The Thesis Committee for Gregory Ryan King
Certifies that this is the approved version of the following Thesis:

Comparative Systemic Analysis of Human Immunoglobulin Repertoires

APPROVED BY
SUPERVISING COMMITTEE:

George Georgiou, Supervisor

Gregory Ippolito

Comparative Systemic Analysis of Human Immunoglobulin Repertoires

by

Gregory Ryan King

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

December 2018

Acknowledgements

I would like to acknowledge and extend my gratitude towards my advisor, George Georgiou, and to Professor Gregory Ippolito for their advice and patience throughout my time in the BIGG lab. Working with George and Greg was always exciting, and having the opportunity to learn from them and be a part of a group of so many great scientists is a once in a lifetime experience. I also want to thank the entire lab for their guidance and input along the way, and for support from Professors Jenny Jiang, Jennifer Maynard, Edward Marcotte, Som Mukhopahdyay, and Bryan Davies.

Finally, I could not have come this far without my wife, Rhea King. I am eternally grateful for her endless love and encouragement, and her patience and understanding through the worst times and the best.

Abstract

Comparative Systemic Analysis of Human Immunoglobulin Repertoires

Gregory Ryan King, M.A.

The University of Texas at Austin, 2018

Supervisor: George Georgiou

The humoral immune system is majorly composed of B cells producing effector immunoglobulin molecules, the vast diversity of which allow for the neutralization of pathogenic threats never previously seen by the immune system. High-throughput sequencing technology has allowed this vast repertoire to be characterized and quantified, but understanding this complex system requires methods of comparison to identify and differentiate B cell populations. In this thesis, differences between groups of repertoires within individuals are analyzed at both the cellular and proteomic level. Novel experimental techniques and visualization methods will allow for the analyses of several such high-dimensional complex systems, leading to a fuller picture of the B cells' contribution to the immune system.

Table of Contents

Chapter 1: The Plasma Cell Repertoire and Serological Contribution

BACKGROUND

A hallmark of the vertebrate adaptive immune system – hit upon by evolution in the very early stages of animal life on Earth and a major component of immunity in all mammals – is the receptor and soluble effector molecule of the B cell: the immunoglobulin, also known (for its ability to bind foreign bodies) as the antibody (Kindt et al. 2007). The antibody is a tetrameric glycoprotein made up of two dimers each containing a paired heavy and light chain (heavy and light referring to the molecular masses) joined covalently with cysteine disulfide bonds. The antibody's essential role to play is in determining the presence of an unknown / foreign antigen, and relaying this message to a variety of cell types and other effector molecules to deal with the intruder promptly. The N-terminus and C-terminus of the antibody each have specialized functions that allows them to serve in this function. The N-terminus of both the heavy and light chains include regions of non-germline sequences with a vast diversity of potential tertiary protein configurations which allow the molecule to bind to a theoretically infinite range of ligands, in some cases with extremely high specificity (Murphy and Weaver, 2016). Since this variable region (known as VH for the Variable Heavy chain and VL for the Variable Light chain) is present in duplicate on the antibody, the molecule will not only bind a target but can also crosslink multiple targets, creating aggregates that stimulate an ever-stronger local response.

The C-terminus of the antibody, often known as the Constant region, is the portion of the heavy chain which signals specifically to other cells in the immune system where a potential target of interest lies. All animals have several interchangeable constant regions encoded by different genes, each with a different profile of interaction with various cell subsets. In humans, there are five isotypes for the heavy chain constant region: IgG, IgA, IgM, IgD, and IgE. The IgG and IgA isotypes are further divide into 4 and 2 subgroups, respectively. Each type may specialize in attracting certain cells or other effector molecules: for example, IgG3 is a strong attractor for the complement C1q binding protein that stimulates an innate cell-free attack, while IgA1 is more efficient at bringing neutrophils to assist (Vidarsson et al. 2014). While the canonical concept of a tetrameric “Y” shaped antibody is applicable to the IgG, IgE, and IgD subclasses (Figure 1.1), IgA and IgM can and often do take on polymeric configurations – IgA is commonly found as a dimer of two single IgA molecules (Joined by a J chain tail-to-tail) especially in sites of mucosal secretions, and soluble IgM takes on a large pentameric configuration. In the periphery, IgG subtypes are the most common member by far, making up a combined 85% of serum immunoglobulins (Manz et al. 2005). In the mucosa, the balance shifts heavily in the favor of IgA; IgA production in the mucosal sites most likely surpasses the production of all other antibody isotypes throughout the body (Rifai et al. 2000).

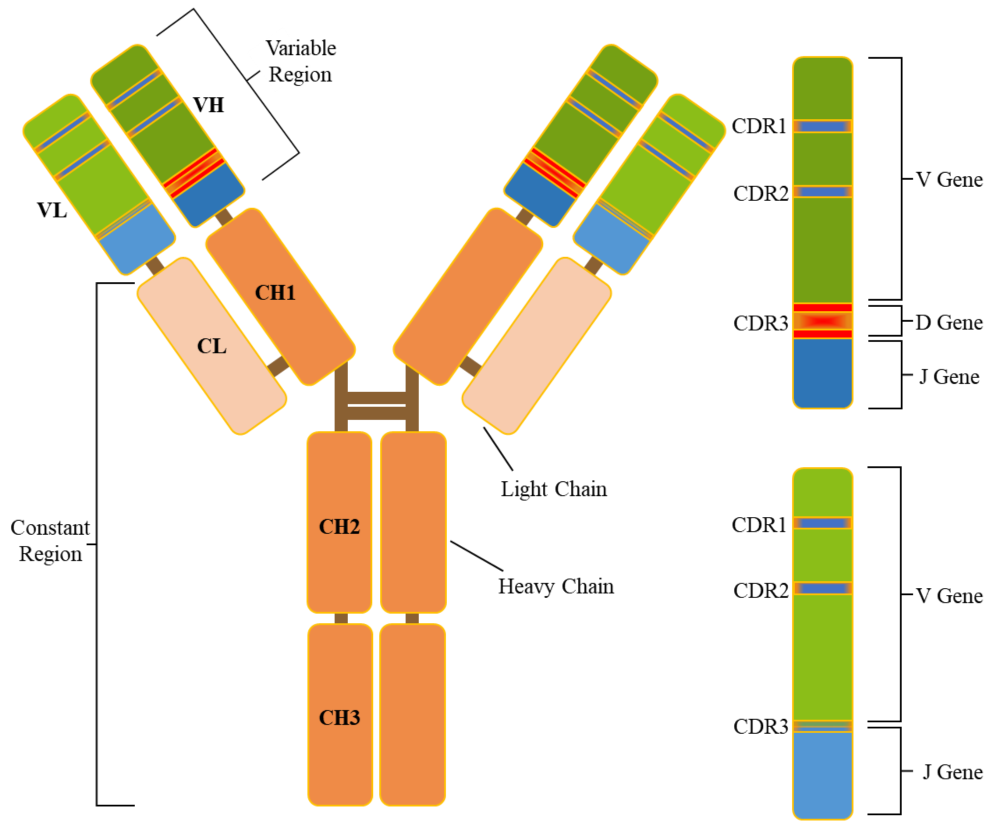


Figure 1.1: The generalized structure of mammalian antibodies (specifically IgG).

The production of antibodies is entirely due to the diverse set of immunological cells known as B cells. B cells begin life in the bone marrow, in stromal sites containing hematopoietic stem cells which are constantly dividing throughout life. Like all germline cells, the precursors to B cells have the fundamental building blocks for any possible antibody in the immunoglobulin heavy-chain locus, on the 14th chromosome in humans. The locus is made up of many variants of three gene types, known as Variable (V), Diversity (D), and Joining (J). The first steps in development require the B cells to begin the process of heavy chain gene component recombination to bring together first a D and J gene at random, after the success of which allow for the second recombination of the V gene to the DJ segment. This bringing together of three random assortments of the dozens of V genes and D genes with one of 6-7 J genes alone yields a sequence space of many thousands of possible heavy chain sequences. However, V-DJ joining is also accompanied by exonuclease digestion at the site of the VDJ recombination, the so-called junction region. As the junction region is chewed back a random number of bases, a low-fidelity DNA polymerase adds back a set a randomized bases that greatly increase the diversity of this site (see Figure 1.2). In the (likely) chance that a deleterious mutation occurs and no pre-B heavy chain is able to express, the V-D-J gene cassette of the second chromosome-14 attempts another recombination, doubling the chances for a particular cell to create a functional heavy chain.

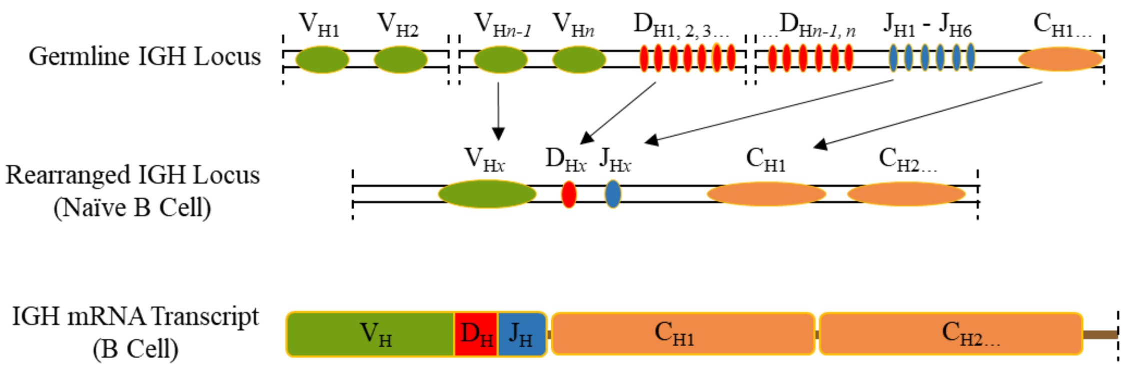


Figure 1.2: The Human germline immunoglobulin heavy chain locus (IGH), with many variants of the Variable, Diversity, and Joining genes are displayed on top. The middle section displays the rearranged IGH locus at the naïve B cell stage with the recombination of a single V-D-J set followed by a constant domain determining the isotype. The bottom section depicts the spliced mRNA transcript as expressed in a functional B cell.

Upon a final productive rearrangement, naïve B cells that show no binding to self-antigens (preventing autoimmunity) are released from the bone marrow into peripheral circulation to allow exposure to potential pathogen-expressed antigens for an immune response. Should a B cell membrane-bound antibody receptor find a cognate antigen, the cell can undergo a variety of proliferation and differentiation steps to allow the cell to respond more fully to the invader. The two major effector B cell forms that allow both a full primary response and the ability to persist for long periods of time and contribute to immunological memory are the Plasma cell and the Memory B cell (figure 1.3). These important steps take place within lymphoid organs in segments known as the germinal center (figure 1.4).

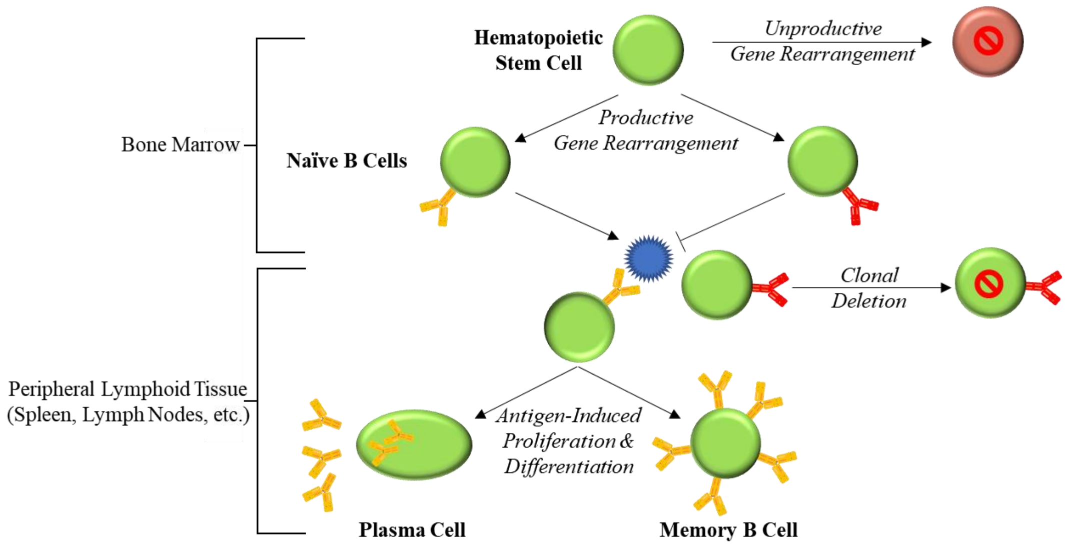

Hematopoietic Stem Cell
Unproductive Gene Rearrangement
Bone Marrow
Naïve B Cells
Productive Gene Rearrangement
Clonal Deletion
Peripheral Lymphoid Tissue (Spleen, Lymph Nodes, etc.)
Antigen-Induced Proliferation & Differentiation
Plasma Cell
Memory B Cell

Figure 1.3: The generalized creation and end states of B Cells. Hematopoietic stem cells in the bone marrow can differentiate into B cells, from which only the minority of cells that undergo a productive heavy and light chain rearrangement can leave the bone marrow as naïve B cells. Upon encountering antigen in the periphery, B cells can mature into Plasma and Memory B Cells – the variants that contribute to an effective immune response and that can remain active for months to years and contribute to a secondary response should the same antigen be encountered in the future.

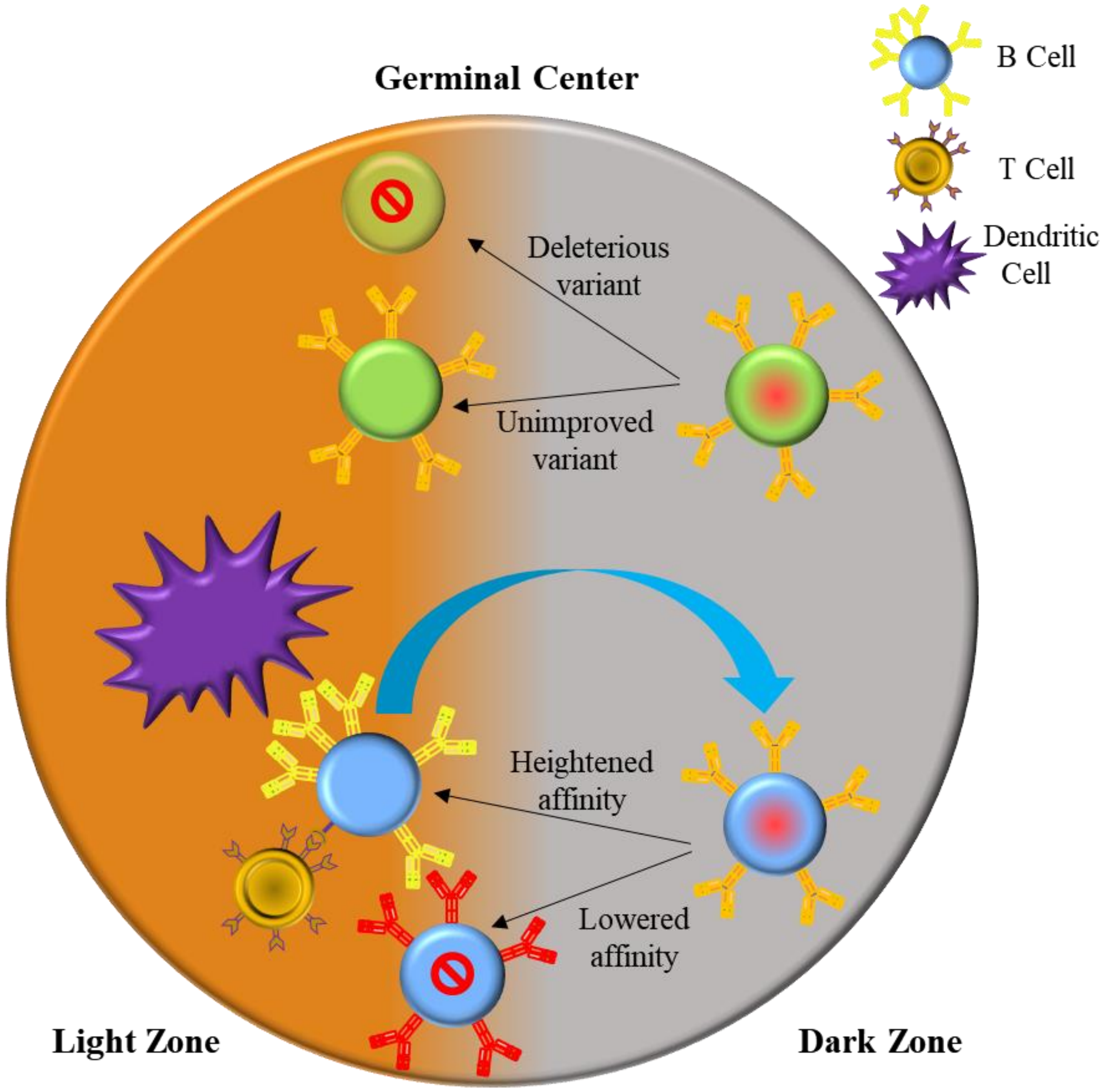


Figure 1.4: Induced hypermutation and proliferation of B cells in the germinal center. Cells that actively bind antigen displayed by antigen presenting cells, such as dendritic cells, in the "light" zone are stimulated to rapidly mutate and divide in the "dark" zone. The minority of cells with enhanced antigenic binding can then be repetitively selected for additional rounds of mutation, while non-functional or less useful variants die off as they are outcompeted for antigen to bind.

Plasma cells, more than any other B cell group, provide the body with humoral immunity through the constant high-level production of secreted antibodies. Plasma cells arise from particularly successful plasmablasts induced during exposure to antigenic challenge; newly formed plasma cells have a lifespan of several days as they exist in the periphery, but long-term antigen protection is truly provided when these new plasma cells find their way to a survival niche in which they can be stimulated to persist and continue to secrete antibody (Radbruch et al. 2006). The niches sought out by plasma cells depend on their origins in the gut or periphery, but the most well understood niche contributing to serological long-term antibody expression is the bone marrow. Plasma cells that migrate back to their developmental origin can receive guidance by signaling from T cells and the bone marrow stroma (Koch and Benner 1982). Once settled, these plasma cells transition from a life cycle of days to weeks to one lasting months to years. Recent studies have begun to identify distinguishing factors between plasma cells of various ages in the bone marrow; the B cell co-receptor and signaling enhancer molecule CD19 has been directly shown to lower in expression levels as plasma cells remain in the bone marrow for longer periods of time (Henrik, E.M. et al. 2014, Halliley, J. et al. 2015). This allows for the selection and differentiation between the populations of plasma cells which may be newly derived and of the super-competent members that may be directly responsible for the protection against pathogens over many years.

MATERIALS AND METHODS

Peripheral Blood and Bone Marrow Mononuclear Cell and Serum Isolation

All peripheral blood and bone marrow aspirate samples were collected by AllCells and shipped overnight at ambient temperature; processing was started immediately upon receipt of the samples in a sterilized biological safety hood. Sterile 1x PBS pH 7.4 plus 2 mM EDTA (without Ca^{2+} or Mg^{2+}) at room temperature was used to dilute samples prior to density centrifugation separation: peripheral blood (60 mL starting volume) was diluted 1:1 by volume, and bone marrow aspirate (25 mL starting volume) was diluted 7:1 by volume. The samples were aliquoted out in 35 mL increments and cautiously layered into 50 mL conical centrifuge tubes containing 15 mL of room temperature Lymphocyte Separation Medium at 1.077 g/mL density (Corning, #25-072), without disturbing the separation medium interface. The vials were centrifuged at room temperature at 800g for 15 minutes, with the brake turned off to ensure gentle deceleration.

The upper serum / aspirate supernatant layers were removed by pipette and stored in 10 mL aliquots at -80 °C for future use. The mononuclear cell layers were then carefully aspirated by glass transfer pipette and pooled into a fresh 50 mL vial and brought to a final volume of 50 mL with PBS plus EDTA. The cell solution was gently mixed by inversion then centrifuged at 300g for 10 minutes at room temperature, and the PBS wash supernatant was discarded. Two additional PBS washes were performed, and the final cell pellet was resuspended in 2 mL 1x PBS pH 7.4 plus 2 mM EDTA and 0.5% Bovine Serum Albumin.

Two 10 μL samples of the cell suspension were diluted 1:10 with 0.4% Trypan Blue for viability and cell concentration measurement via hemocytometer. For peripheral blood samples, ~1-2 x 10^7 mononuclear cells were pelleted and resuspended in 1 mL TRIzol RNA extraction reagent (Thermo Fisher #15596026) and stored at -80°C. The remaining cells were brought to ~2 x 10^7 cells/mL with 2x cell freezing medium: 40% RPMI 1640 (ThermoFisher #12633012), 40% heat-inactivated fetal bovine serum (ThermoFisher #16140089), 20% DMSO (Sigma-Aldrich #D2650). Cell suspensions were then stored in liquid nitrogen after controlled freezing via isopropyl alcohol chilling (Nalgene Mr. Frosty, Sigma-Aldrich #C1562). For the bone marrow mononuclear cells, in addition to the frozen cell suspension and TRIzol aliquots, two additional ~1-2 x 10^7 mononuclear cell aliquots were set aside.

Plasma Cell and CD19+/- FACS Sorting of Bone Marrow Mononuclear Cells

FACS sample staining was performed by Gregory King, Gregory Ippolito, and Sebastian Schaetzle and sample sorting and processing by Richard Salinas.

The first bone marrow mononuclear cell aliquot was prepared for FACS plasma cell plus CD19-expression level sorting. 10^7 cells in 1.1 mL FACS buffer (1x PBS pH 7.4 plus 0.5% bovine serum albumin) were split into two equal aliquots in 1.5 mL Eppendorf tubes. The first aliquot (individual color controls) was then portioned into 5 tubes at 0.1 mL each, and 10 µL of each of the labeled FACS antibody solution was added to each separate tube. The second sample aliquot at 0.5 mL had 10 µL of all 5 labeled antibodies added. All tubes were incubated in the dark for 15 minutes at room temperature, with gentle flicking every five minutes. Cells were washed with 1 mL FACS buffer, pelleted at 300g for 7 minutes, then resuspended with 0.5 mL FACS buffer for the color controls or 1 mL FACS buffer for the full sample. All samples were filtered into FACS tubes with a 45 µm filter cap (Corning #352235) and brought to sorting immediately on ice. The collected plasma cell subsets were immediately stored in 1 mL TRIzol at -80°C.

Antibodies used for the plasma cell and CD19 sorting were Mouse Anti-Human CD138 PE (Miltenyi Clone B-B4, #130-081-301), Mouse Anti-Human CD19 v450 (BD Biosciences, #556633), Mouse Anti-Human CD38 FITC (BD Biosciences Clone HIT2, #555459), Mouse Anti-Human IgD PE (BD Biosciences Clone IA6-2, #555779), and Mouse Anti-Human CD27 APC (BD Biosciences Clone M-T271, #558664).

Library Preparation for VH-Only Sequencing

For mRNA isolation from the lysed cell samples stored at -80 °C in TRIzol, the tubes were thawed and 0.2 mL chloroform was added; the mixture was then vortexed for 15 seconds and allowed to settle for 5 minutes at room temperature. The tubes were then centrifuged at 4 °C for 10 minutes at 12000g, followed by careful removal of the upper clear layer into a fresh Eppendorf tube. 70% ethanol was added to the tubes 1:1 by volume and gently mixed. The samples were then run through an RNA purification centrifugal column and processed as recommended by the manufacturer (RNeasy Mini, Qiagen). 2x elutions with 15 µL each of RNAse-free water were pooled and RNA concentrations were measured by Nanodrop absorbance; 1000ng of RNA was set aside for immediate cDNA synthesis with the remainder stored at -80 °C.

cDNA production was performed following the manufacturer's protocol (ThermoFisher SuperStrand IV) using 1000ng of sample RNA. The cDNA synthesis program used was 50 °C for 4 minutes, 52 °C for 5 minutes, 55 °C for 6 minutes, and finally 80 °C for 10 minutes. After the cDNA synthesis, 1 µL of RNase H was added and incubated for 20 minutes at 37 °C. The samples were then prepared for VH-only amplicon PCR using the manufacturer's recommended FastStart Taq polymerase protocol with 8 µL of the cDNA product for a 400 µL final volume reaction, aliquoted at 50 µL per PCR tube. The program for VH amplification was: 95 °C for 2 minutes, 4 cycles of 92 °C, 50 °C, and 72 °C (each 1 minute), 4 cycles of 92 °C, 55 °C, and 72 °C (each 1 minute), 22 cycles of 92 °C, 63 °C, and 72 °C (each 1 minute), and finally 72 °C for 7 minutes. PCR products were purified using a Zymo-Spin I DNA binding column (Zymo Research #C1003) and eluted 2x with 15 µL water each. The purified DNA was then run on a 1% TAE-agarose gel, and the appropriately sized VH band was cut out and purified with a Zymo-Spin I DNA binding column using agarose dissolving buffer (expected ~400bp band for IgG and IgM or ~500bp for IgA). Once again, the 2x 15 µL water elutions were pooled and the concentration measured by Nanodrop absorbance. DNA was then given to the University of Texas at Austin Genome Sequencing and Analysis Facility sequencing core for MiSeq library preparation by ligation and sequencing.

Bone Marrow B Cell MACS Isolation

Initial 2 donor sample VH-VL pairing was done by Jon McDaniel, and subsequent donor samples were processed by Gregory King.

The remaining 10^7 bone marrow mononuclear cells were then prepared for single cell emulsion VH-VL paired RT-PCR. The cell suspension was first depleted of non-B cells by magnetic-activated cell sorting (MACS) using the negative selection portion of the Miltenyi Human IgG+ Memory B Cell isolation kit (Miltenyi #130-094-350). The cell suspension was pelleted at 300g for 10 minutes at 4 °C, then resuspended in 0.4 mL of chilled MACS buffer (1x PBS pH 7.4 plus 0.5% bovine serum albumin). 0.1 mL of the Miltenyi B Cell Biotin-Antibody Cocktail was added and gently mixed then incubated for 10 minutes at 4 °C. 0.3 mL of chilled MACS buffer and 0.2 mL of Anti-Biotin microbeads were added and mixed, followed by another 15-minute incubation at 4 °C. The cell suspension was centrifuged for 10 minutes at 300g at 4 °C, followed by removal of the supernatant and resuspension of the cell pellet in 1 mL cold MACS buffer. The cell suspension was run through a 45 µm filter (Corning #352235), then added to an equilibrated MACS LD column (Miltenyi #130-042-901). The non-B Cell depleted flowthrough was collected and pooled with 4x 1 mL MACS buffer washes. The cells were then washed with 15 mL chilled PBS and resuspended in 5 mL PBS, and a final viability count was performed prior to emulsion-based lysis and mRNA capture.

Single-cell lysis and mRNA capture for bone marrow B Cell VH-VL RT-PCR was performed as described in McDaniel, J.R. et al. 2016. The B Cell suspension in PBS at 10^5 cells/mL and an equal volume of lysis buffer (100 mM Tris pH 7.4 plus 500 mM LiCl, 10 mM EDTA, 1% LiDS, 5 mM DTT) containing mRNA-capturing magnetic microbeads (Oligo d(T) microbeads, New England Biolabs #S1419S) were run through an in-house pump to form single-cell droplets in emulsions (4.5% Span-80, 0.4% Tween-80, and 0.05%

Triton-X 100 in mineral oil) for mRNA capture of paired heavy and light chain transcripts on beads. Emulsions were formed at a flow rate of 0.5 mL/minute for the aqueous solutions and 3 mL/minute for the oil, and captured in 50 mL conical vials chilled in ice. Emulsions were gently pooled and centrifuged at 4000 RPM for 6 minutes at 4 °C, and the upper layers of oil and non-cell containing emulsions were removed. An equal volume of chilled hydrated diethyl ether was added to the remaining large emulsions and mixed by gently inverting, followed by another 4000 RPM 6 minute centrifugation step at 4 °C. All supernatant was removed, and the pelleted mRNA-containing beads were resuspended in 1 mL of chilled wash buffer (100 mM Tris pH 7.5 plus 500 mM LiCl and 1 mM EDTA). The beads were then pelleted on a magnetic rack at 4 °C and washed once with lysis buffer, followed by two washes with the bead wash buffer and a final equilibration in 0.5 mL of 20 mM Tris pH 7.5 plus 3 mM MgCl and 50 mM KCl. After pelleting a final time, the beads were thoroughly resuspended in chilled emulsion RT-PCR mixture containing RTX polymerase. The ~3 mL prepared RT-PCR sample was then re-emulsified in 9 mL of the mineral oil mixture for 5 minutes using an Ultra-Turrax DT-20 emulsifier tube (IKA #0003700600) and aliquoted out into 96-well chilled PCR plates at 100 μL per well. The plates were sealed and placed in a thermocycler pre-heated to 68 °C, and were kept at 68 °C for 30 minutes for reverse transcription followed by heating to 94 °C for 2 minutes. Amplification was performed with 25 cycles of 94 °C for 30 seconds, 60 °C for 30 seconds, and 68 °C for 2 minutes. A final 68 °C extension was performed for 7 minutes and plates were stored at 4 °C.

All reaction products were pooled and transferred to 2 mL Eppendorf tubes, then centrifuged at 4 °C for 10 minutes at 13000g. The mineral oil supernatant was removed with plastic disposable pipettes, and the tubes were filled with hydrated diethyl ether and vortexed vigorously twice for 5-10 seconds each to break the emulsions followed by a 40 second centrifugation at 16000g. The upper organic layer was carefully removed, and the aqueous DNA-containing samples were pooled to three 2 mL tubes. Two additional ether wash steps were performed, after which the tubes were left open in a chemical hood for 15 minutes then concentrated in a vacuum centrifuge at ambient temperature for 45 minutes to evaporate any residual ether. The mRNA-capturing beads were magnetically pelleted, and the supernatants were pooled. Two washes with Zymo-Spin DNA purification buffer were pooled with the supernatants (final volume of ~5 mL) and purified in a single Zymo-Spin I DNA purification column (Zymo Research #C1003) in a final volume of 30 μL H_2O.

Serum IgG Purification and Preparation

For all donors, 5 mL of serum frozen at -80 °C was thawed to room temperature and centrifuged for 5 minutes at 15000g to pellet debris, then diluted 1:1 with 1x PBS pH 7.4 and filtered through a 0.22 μm syringe filter. For each separate sample, 1.5 mL of Protein G+ agarose (Thermo Scientific) resin was placed in a 5 mL polypropylene protein purification column (pre-washed once with 5 mL 70% ethanol), then equilibrated with 3x 5 mL PBS washes at room temperature. The serum sample was then added to the resin and allowed to flow through by gravity; flowthroughs were collected and run an additional two times to ensure complete binding of antibody to the resin. 2x 1 mL PBS washes were pooled with the flowthroughs, then 3x 10 mL PBS washes were collected separately. 8x 1.5 mL Eppendorf tubes were prepared each containing 0.1 mL 1M Tris pH 8.0 to immediately neutralize the elution fractions; 8x acid elutions using 0.9 mL of 100 mM glycine pH 2.7, collected into the prepared tubes. Protein concentration estimates were performed via Nanodrop absorbance readings, and all fractions containing antibody were pooled. The pooled antibody samples were then dialyzed overnight using a 10 kDa MWCO dialysis membrane (SnakeSkin dialysis membrane) at 4 °C into 1x PBS, pH 7.4.

To remove the antibody constant regions to assist with LC/MS proteomic analysis, the samples were digested using IdeS enzyme, which is IgG-specific and cleaves only in the hinge region (IdeS protease produced in-house). IdeS was added at a 1:50 ratio of enzyme:IgG, and the mix was incubated for 2 hours at 37 °C. After digestion, a 10 μg aliquot of digested IgG was run on a reducing SDS-PAGE gel to ensure proper cleavage.

RESULTS

Experimental Outline and Sample Processing

In order to better understand the human antibody repertoire both in different tissue compartments and at different life stages, a set of six healthy male donors – three younger (ages 20-25) and three older (ages 50-55) – were selected to provide both peripheral blood and a hip bone marrow aspirate. The donors in each age group were matched as closely as possible in terms of activity levels, height and weight, and medications used. To ensure the samples provided would be as fresh as possible and to avoid cross-contamination and working with too many samples at once, each donor's peripheral blood sample was shipped overnight at ambient temperature (never frozen) and immediately processed the next day; the bone marrow aspirate samples were then taken two days later and shipped / processed similarly. The matched blood and bone marrow samples were therefore most likely to be given in the same immunological state of health for each donor, and the sample processing and separation into specific cellular compartments could be performed immediately upon receipt.

While peripheral blood is the most easily accessed immunological tissue and therefore the most commonly studied, the bone marrow is arguably of greater interest as both a site of initial development of B cells and as a store of long-lasting immunocompetent memory B cell and plasma cells that carry out the rapid effector response to previously encountered pathogens. For this reason, we set out to characterize the general bone marrow B cell repertoire population as it compares to the periphery and the important plasma cell population within. Since recent reports have implicated the BCR co-receptor molecule CD19 as an easily selected marker for plasma cell longevity within this compartment, the long-term resident plasma cell subset was chosen as a major area of study to identify the characteristics of the most successful members of the immunoglobulin-producing family.

Mononuclear cell isolation via density centrifugation was used to separate the plasma fractions from both blood and bone marrow aspirates and remove platelets, bone marrow stromal cells, and other non-adaptive immunological cell subsets (see Figure 2.1 for the sample processing overview). The peripheral blood mononuclear cells (PBMCs) for all donors were assayed for viability before processing for long-term liquid nitrogen storage or immediate RNA isolation for VH-only BCR sequencing. The bone marrow mononuclear cells (BMMCs) were split into several fractions for more detailed analysis. Two aliquots of ~10^7 BMMCs each were set aside for VH-VL paired single-cell repertoire sequencing of total B cells and for FACS plasma cell isolation based on CD19 expression levels, respectively. Briefly, the portion for paired repertoire analysis was magnetically depleted of non B cells via MACS, then run through an in-house flow-focusing device for single cell lysis and mRNA capture leading to emulsion-based RT-PCR to pair each cell's VH and VL transcripts into a single cDNA amplicon. The FACS stained BMMC samples were sorted first based on CD38high and CD138high selection of bone marrow plasma cells, from which the cells were isolated into CD19high and CD19low expression groups (for newer plasma cells and long-term resident, respectively) and immediately stored in TRIzol at -80°C for VH-only sequencing and repertoire analysis (see table 1.1 for cell counts per donor and number of cells used for each subset). The final sequencing libraries derived from each donor compartment – Peripheral Blood Mononuclear Cell (PBMC), Bone Marrow Mononuclear Cell (BMMC), and Plasma cells sorted by CD19 expression derived sequences – and the total number of clones for each library after quality filtering are enumerated in table 1.2.

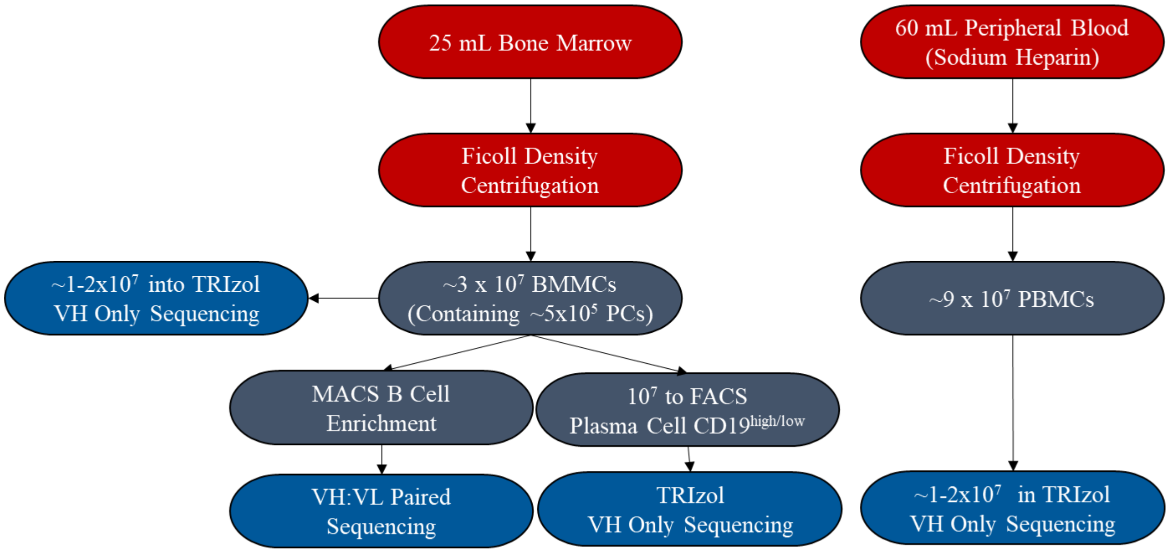

25 mL Bone Marrow
60 mL Peripheral Blood (Sodium Heparin)
Ficoll Density Centrifugation
Ficoll Density Centrifugation
~1-2x10⁷ into TRIzol VH Only Sequencing
~3 x 10⁷ BMMCs (Containing ~5x10⁵ PCs)
~9 x 10⁷ PBMCs
MACS B Cell Enrichment
10⁷ to FACS Plasma Cell CD19high/low
VH:VL Paired Sequencing
TRIzol VH Only Sequencing
~1-2x10⁷ in TRIzol VH Only Sequencing

Figure 1.5: Experimental sample selection and processing pipeline for donor bone marrow and peripheral blood samples.

Donor	**Age**	**Total BMMCs**	**PBMCs**	**B Cells Paired**	**BMMCs to FACS**
Young1	24	1.52×10^8	7.50×10^7	1.35×10^6	1.20×10^7
Young2	22	1.87×10^8	1.70×10^8	1.50×10^6	1.88×10^7
Young3	24	5.40×10^7	8.88×10^7	1.00×10^6	1.35×10^7
Old1	57	6.88×10^7	1.72×10^8	9.35×10^5	1.72×10^7
Old2	56	9.04×10^7	1.26×10^8	1.16×10^6	1.13×10^7
Old3	58	1.74×10^8	1.53×10^8	1.80×10^6	1.70×10^7

Table 1.1: Donor ages and cell counts for all experimental samples.

Donor	**Sample**	**Total Reads**	**Filtered Reads**	**Remaining**	**Total Clones**
Young1	PBMC	881023	229117	26.01%	47761
	BMMC	2390817	710758	29.73%	125409
	CD19high	1989717	405022	20.36%	12596
	CD19low	1609903	360066	22.37%	8810
Young2	PBMC	3123204	902838	28.91%	91254
	BMMC	1646601	571490	34.71%	79635
	CD19high	1909840	460823	24.13%	16835
	CD19low	1701771	471196	27.69%	11629
Young3	PBMC	4108230	1089508	26.52%	73975
	BMMC	3972736	864281	21.76%	37727
	CD19high	2077396	679992	32.73%	68006
	CD19low	1267745	331543	26.15%	3191
Old1	PBMC	2747188	746075	27.16%	120909
	BMMC	1961699	600784	30.63%	110042
	CD19high	2034195	482828	23.74%	25487
	CD19low	1801607	383665	21.29%	12120
Old2	PBMC	2525005	664203	26.31%	248502
	BMMC	1347192	427510	31.73%	118584
	CD19high	1810419	336035	18.56%	9306
	CD19low	2219683	562379	25.34%	60241
Old3	PBMC	1130022	352855	31.23%	57114
	BMMC	1846823	581366	31.48%	54163
	CD19high	3370560	879211	26.09%	11006
	CD19low	2572185	656761	25.53%	8908

Table 1.2: Raw read counts from VH-only sequencing, along with quality filtered read counts and total clones per donor sample.

Gene Usage and Class Switching by Repertoire

The base unit for the diversity in the antibody repertoire is the combination of V,D, and J genes that are further mutated in the hopes of increased antigen binding; as the D gene is so highly mutated, the V and J genes are often used as the main identifying factors of a given clonotype. These paired V-J genes for all given clones in the repertoire are highlighted in Figure 1.6. V-J gene pairings are shown as a double donut plot; the inner ring consists of the V genes in order numerically with segment areas representing clonotype prevalence in the population. The outer ring demonstrates the representative J gene usage by clonotypes for each given V gene in the inner ring. In all donors, high enrichment of V gene families 1 and 4 were seen; IGHV1-69 – a member of the largest V family by total number of genes – along with IGHV4-34 were the top members of all donors when looking at the peripheral blood cell derived members. However, in the plasma cell compartment specifically IGHV4-34 became the single more common gene in all donors. In every single donor and cellular compartment sequenced, IGHJ4 was the most commonly used for all clones.

While few discernable trends were seen in isotype usage either amongst donors or compartments, a few generalizations are noteworthy. While IgM was the most dominant constant region in every single sample (50% of total clones or more), IgG-expressing clones were more prevalent in plasma cells as would be expected of cells which have undergone selection in germinal centers; on average, IgG clonotypes were 13.9% more common in these cells than in the periphery. IgA clones were by far the lowest proportion of the repertoire in all compartments, with similar rates across the sample sets.

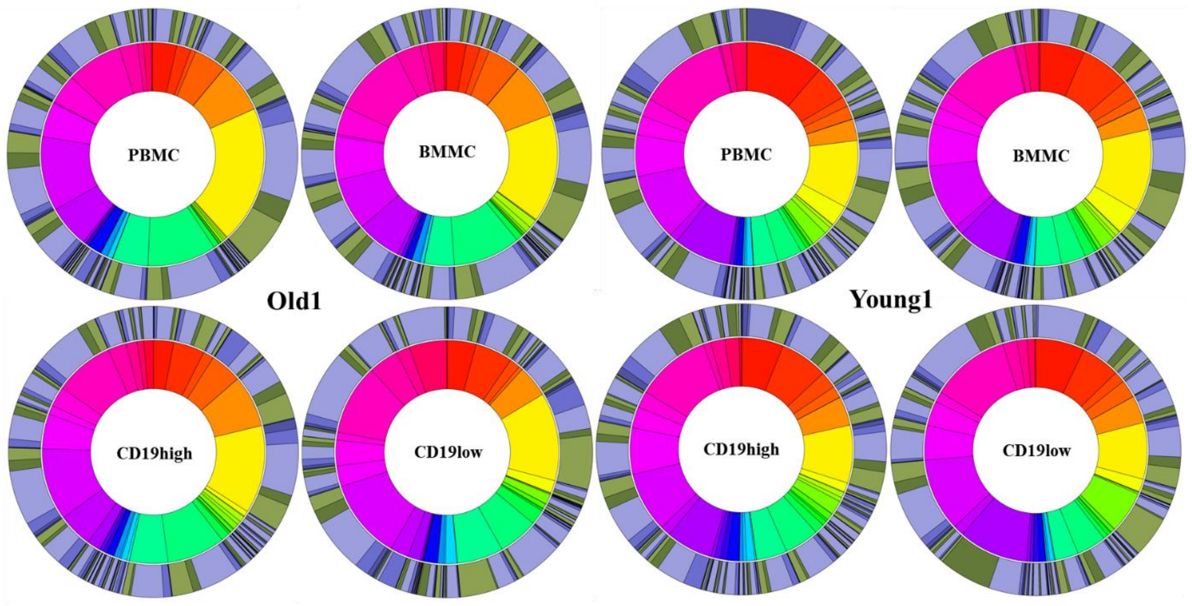


Figure 1.6a: Paired V-J Gene Usage for donors Old1 and Young1; the inner ring segments correspond to donor V genes sorted (clockwise, starting from top center and colored by V family) numerically from V Family 1-on. The outer ring segments show the relative usage of each J gene per V Gene, ordered from J Gene 1-6.

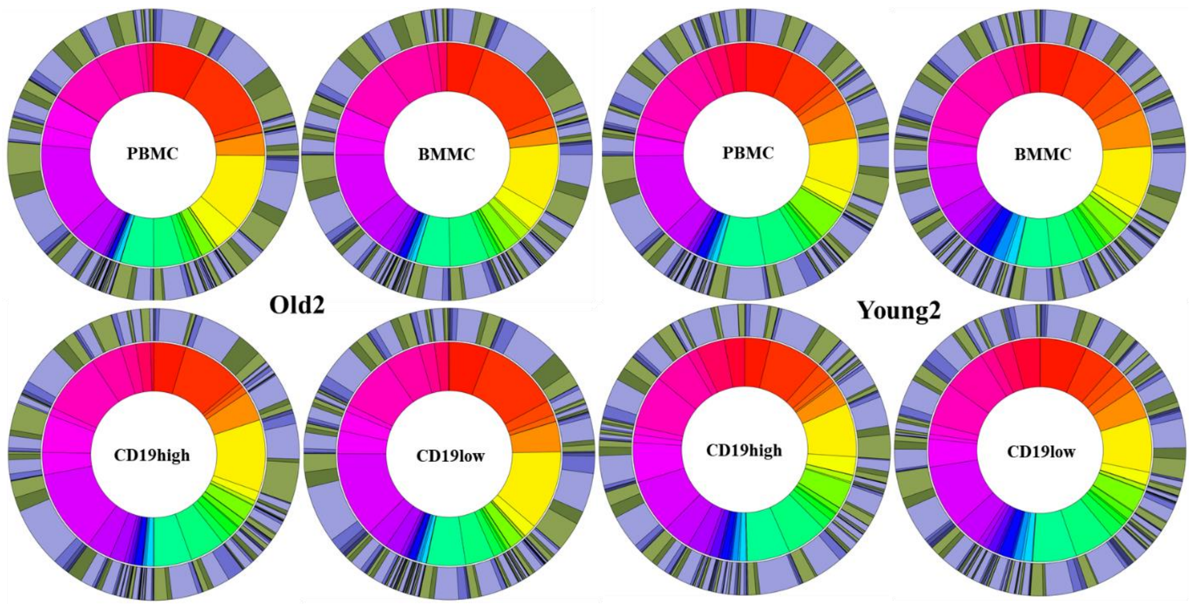


Figure 1.6b: Paired V-J Gene Usage for donors Old2 and Young2; the inner ring segments correspond to donor V genes sorted (clockwise, starting from top center and colored by V family) numerically from V Family 1-on. The outer ring segments show the relative usage of each J gene per V Gene, ordered from J Gene 1-6.

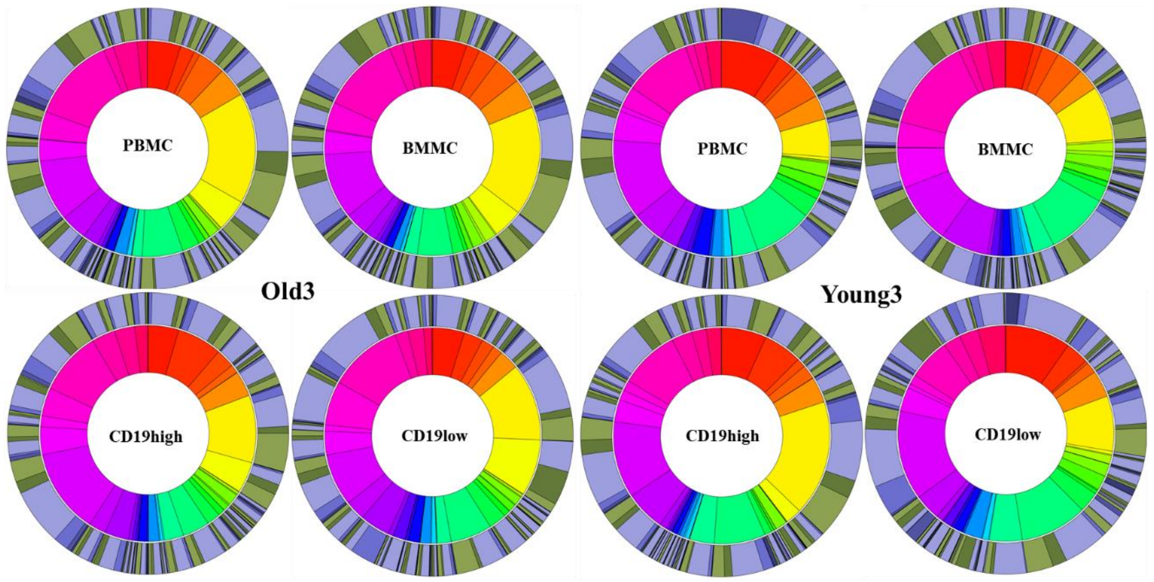


Figure 1.6c: Paired V-J Gene Usage for donors Old3 and Young3; the inner ring segments correspond to donor V genes sorted (clockwise, starting from top center and colored by V family) numerically from V Family 1-on. The outer ring segments show the relative usage of each J gene per V Gene, ordered from J Gene 1-6.

Somatic Hypermutation Rates and Clonal Distribution

As a metric of selection and probable antigen affinity, rates of mutation in a given clonotype's V gene can provide valuable insights into the overall developmental stage of a repertoire. As a population of cells progresses from naivety, levels of somatic hypermutation should show a marked increase as they demonstrate utility. This trend is indeed seen when comparing the peripheral blood cell population to bone-marrow resident cells. Average V gene SHM rates (number of mutations divided by germline V gene length) are lowest for PBMC populations in all sampled donors, while the overall bone marrow B cell population averages around 2.5% higher SHM. Since the major subsets of this bone marrow population are newly forming B cells with little to no V gene mutation and highly mutated plasma cells, the plasma cell subset is expected to have the highest overall mutation rates. This is seen across donors as expected, although the increase is only an additional 2.1% for all. The bimodal population distribution is clearly seen for bone marrow cells in Figure 1.7; in the majority of non-BMMC repertoires, there is a normal tapering off of the SHM distribution with a defined base of near-germline clones.

Also seen in Figure 1.7 is the normalized relative clone sizes in each repertoire subset. The much smaller plasma cell compartments appear to have a polarized distribution; far fewer clones make up a large percent of the total than is seen in the larger and more diverse periphera and bone marrow. The top 100 clones comprise 30% to 60% of the total plasma cell repertoires, while in other repertoires they make up 18% or less. Interestingly, the CD19^{low} fraction appears to be even more heavily weighted by top clones – an observation that suggests fewer remaining members of long-lived populations, as would be expected.

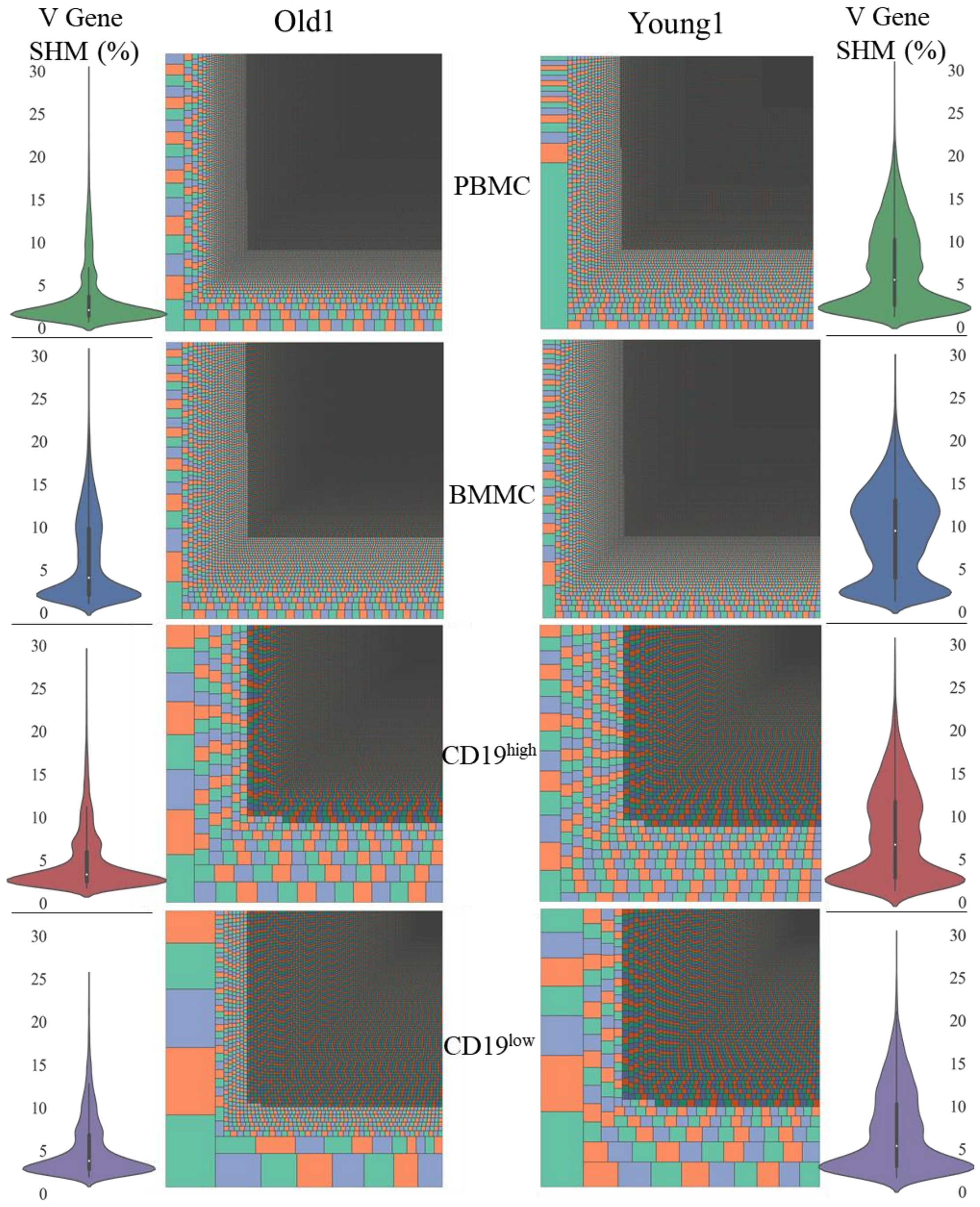


Figure 1.7a: Clonal Mosaics and V gene SHM profile for donors Old1 and Young1. Mosaic patches are ordered from largest clone member to smallest; the bottom 50% of the total clone population is shaded. The violin plots next to each mosaic represent the overall V gene SHM distribution across the population, ranging from 0% to 30%.

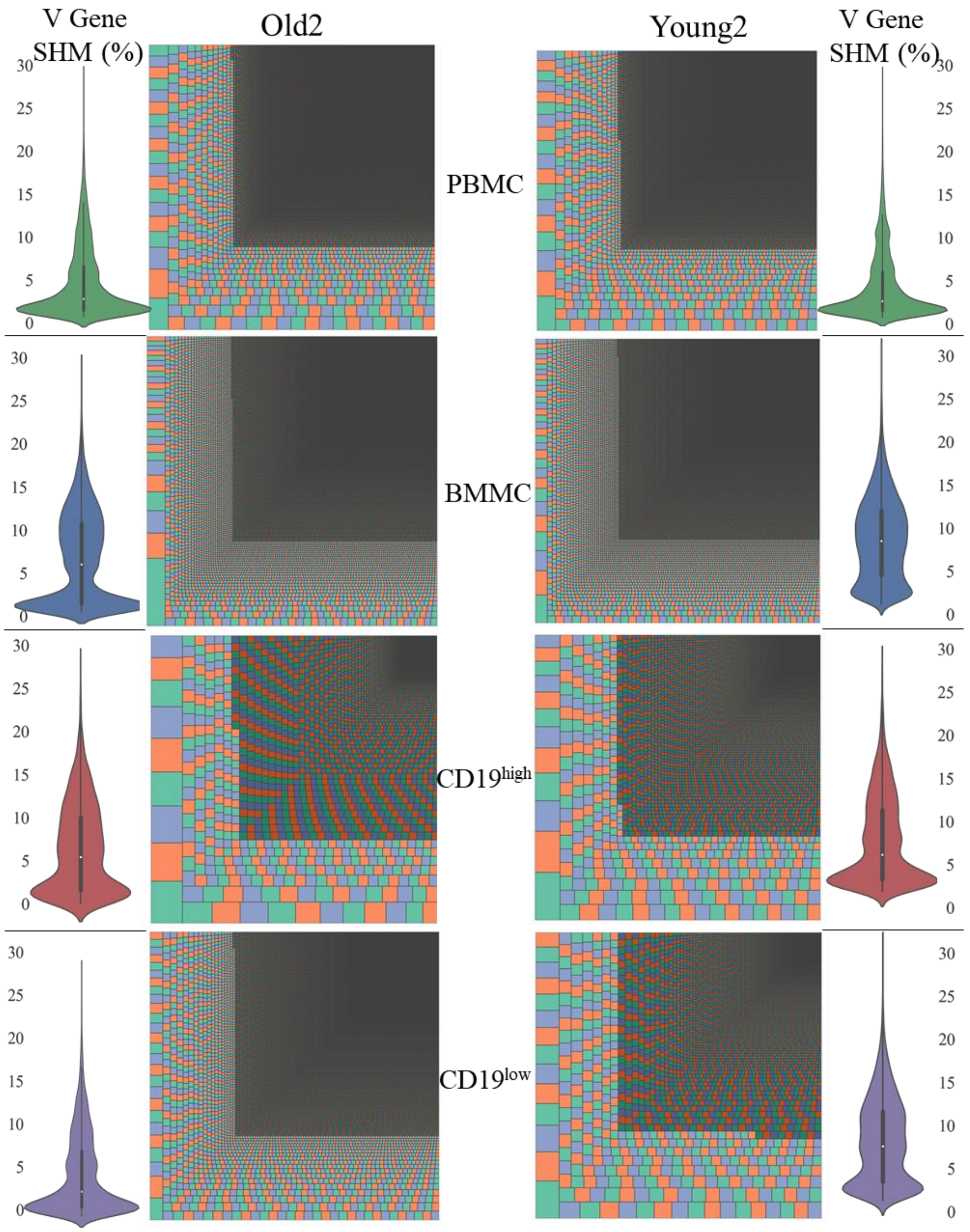


Figure 1.7b: Clonal Mosaics and V gene SHM profile for donors Old2 and Young2. Mosaic patches are ordered from largest clone member to smallest; the bottom 50% of the total clone population is shaded. The violin plots next to each mosaic represent the overall V gene SHM distribution across the population, ranging from 0% to 30%.

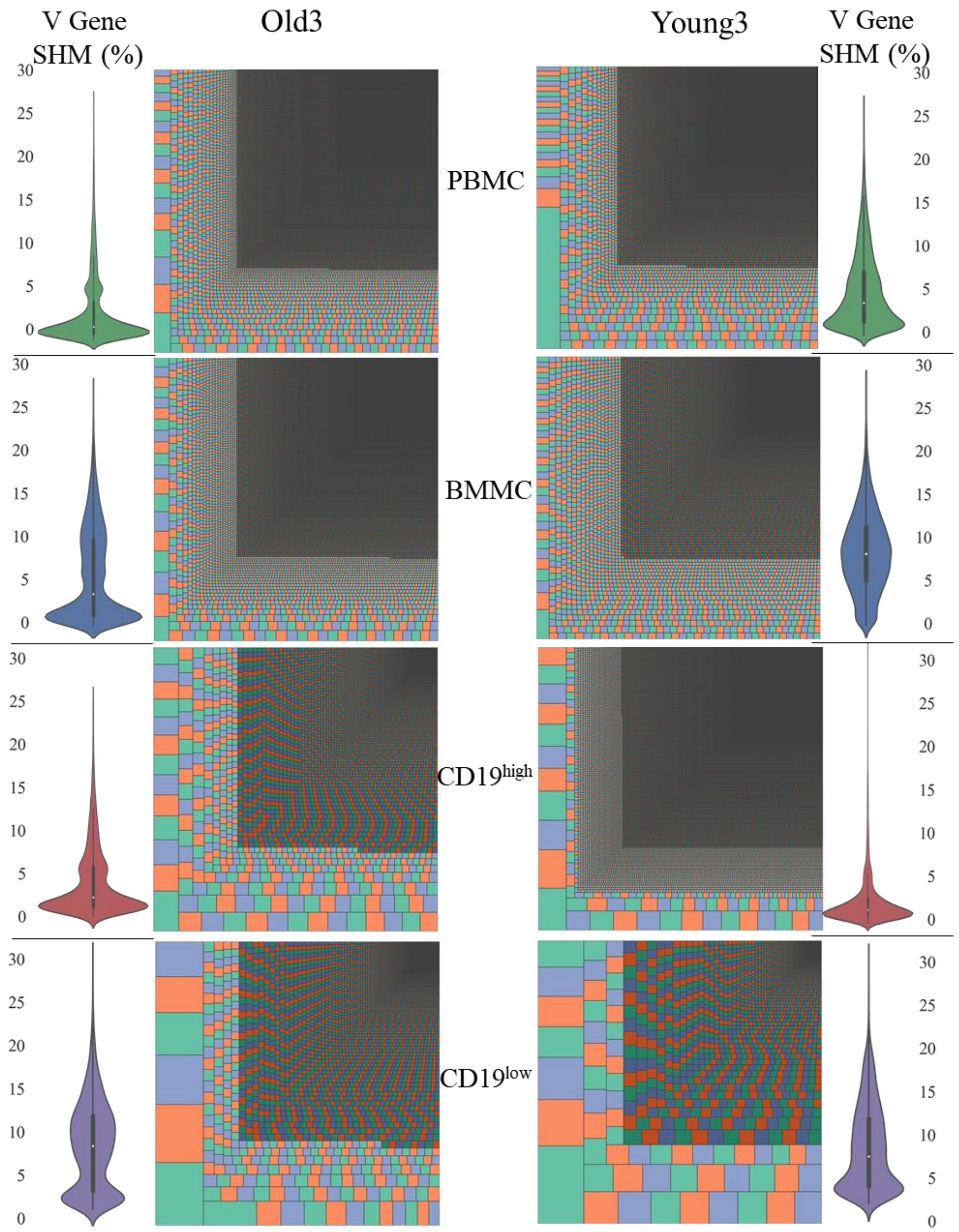


Figure 1.7c: Clonal Mosaics and V gene SHM profile for donors Old3 and Young3. Mosaic patches are ordered from largest clone member to smallest; the bottom 50% of the total clone population is shaded. The violin plots next to each mosaic represent the overall V gene SHM distribution across the population, ranging from 0% to 30%.

Clonal Diversity

Another characteristic of antibody repertoires considered in aggregate is their overall diversity: the number of total unique clones, and their relative frequencies in comparison to all. Compartment clonotype libraries were first pooled by donor age group and analyzed for the total contribution of top clones to the overall repertoire as a crude measure of population polarity (overrepresentation of a few clones compared to the whole repertoire), shown in table 1.3. In both the PBMC and BMMC compartments, repertoire polarity appeared to shift noticeably between age groups. In the periphery, the young donor group population seems to be more dominated by fewer clones, with the top member and top 50 clones being an average of 6.8% and 19.1% of young PBMC repertoires compared to 1.2% and 16.6%, respectively, in the old donors. This trend was reversed in the general bone marrow B cell compartment, with 18.5% of the repertoire consisting of the top 100 clones in the old but only making up 12.7% in the young. This particular trend may stem from a reduction in bone marrow capacity for resident plasma cells in the old, along with a reduced production of naïve B cells (Gibson, K.L. et al. 2009, Cancro, M.P. et al. 2009, Tabibian-Keissar, H. et al. 2016).

Frequency of Top *X* Clones Out of Total Repertoire

Donor	**Compartment**	**Top 1 Clone**	**Top 10 Clones**	**Top 50 Clones**	**Top 100 Clones**
Old	PBMC	1.153%	6.649%	16.852%	23.975%
Young	"	6.814%	11.198%	19.081%	24.889%
Old	BMMC	1.564%	6.127%	13.693%	18.466%
Young	"	0.744%	3.568%	8.765%	12.710%
Old	CD19high	2.328%	14.700%	32.188%	41.930%
Young	"	2.683%	15.163%	33.026%	41.405%
Old	CD19low	5.072%	22.742%	31.705%	36.290%
Young	"	4.325%	19.244%	40.668%	50.940%

Table 1.3: Contribution of top X clones to the overall repertoire for combined age groups, indicative of repertoire polarization and skew towards fewer more prevalent clonotypes.

Many metrics attempt to calculate diversity in an unbiased manner, and while quantifying diversity is not possible with a single equation it can be estimated for comparison best by the Hill diversity number which combines many metrics into one. By this measure, samples are far less likely to be skewed heavily by populations with much larger numbers of unique individuals or extremely polarized populations mostly consisting of a small few individuals (Chao, A. et al. 2014). The Hill diversity index between the orders of 0 and 1 increases as the repertoire is dominated by fewer clones (higher polarization), and orders greater than 2 derives a larger index from a more balanced population of equally sized clones. Graphing the Hill order number for different populations can help visually demonstrate repertoire profile differences between cellular subsets, as seen in figure 1.8. Interestingly, the overall diversity profiles of all compartments compared in the Old donors show remarkable similarity. The BMMC and PBMC compartments show the largest skew towards polarization in the older donors compared to the young. The CD19high and CD19low fractions show a large amount of polarization in most donors, as would be expected from the relatively small number of plasma cells selected; only donor Young1 seemed to show a high diversity of clones with few overly represented members in the newly resident plasma cells. Donor Young2 showed a consistent similarity across all subsets, while donor Young3 showed high polarity in all compartments aside from the very diverse and equally represented bone marrow B cells.

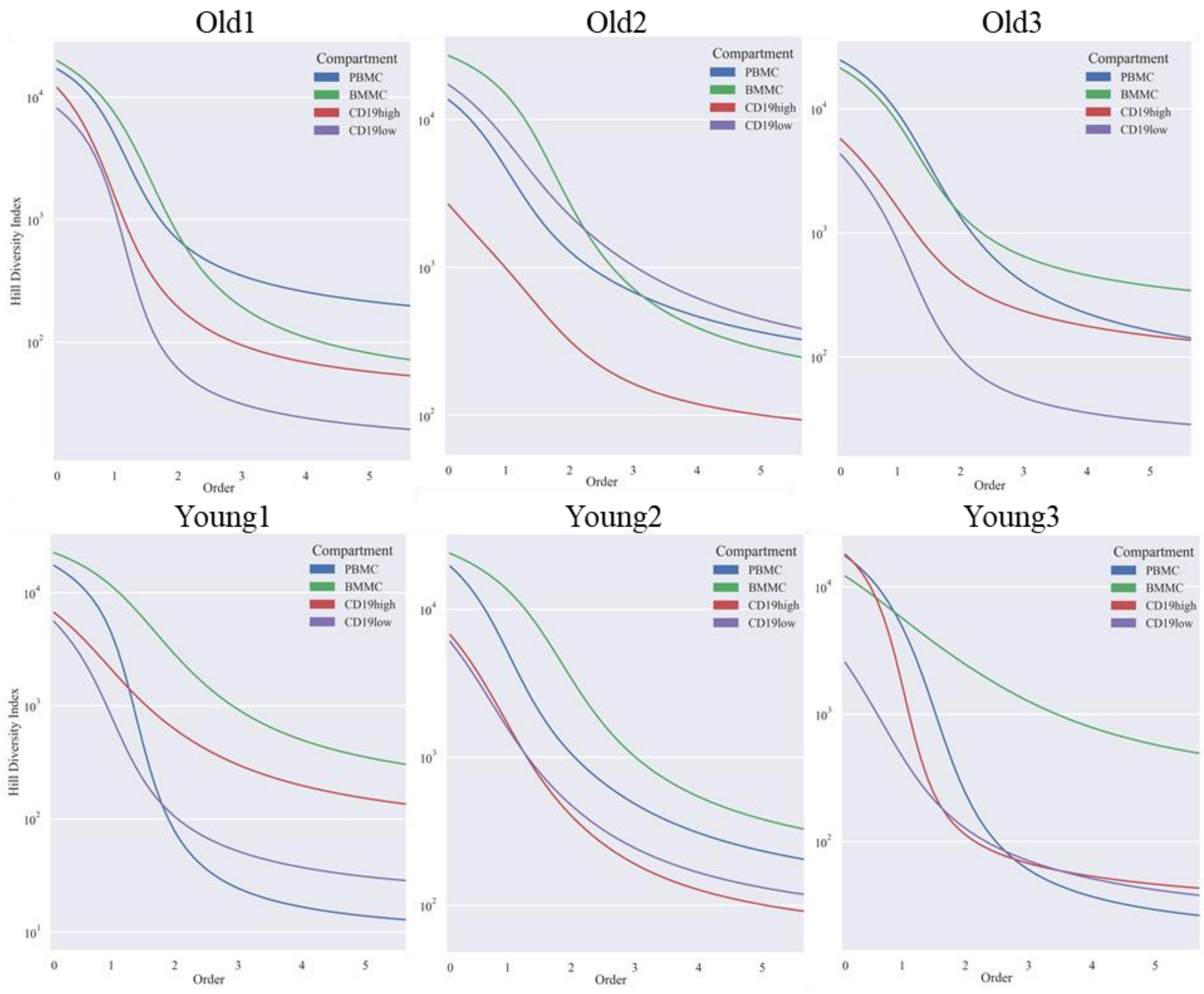


Figure 1.8: Hill number diversity plots for all donors, separated by cellular compartment. The Hill Diversity index from between 0 and 1 order is larger if the repertoire is dominated by fewer clones (higher polarization), whereas at orders greater than 2 a larger index rating derives from a more balanced population of equally sized clones.

Compartmental Repertoire Comparison and Similarity

Since the periphery, bone marrow resident B cells, and plasma cells are all inherently interconnected a natural question that arises is the overall similarity between these compartment repertoires. This similarity is best represented by the shared clones seen in more than one compartment, indicative of members that may still be migratory, or whose highly related cohorts are actively proliferating and can be seen in multiple areas. Since comparisons of more than three compartments are highly complex and visually confusing when presented as Venn diagrams, the clonal overlap for all donors is here presented in figure 1.9 as an UpSet plot generated using an open-source Python library (Conway, J.R. et al. 2017). The bottom segment of the graph shows the compartments with their total clone count, with columns representing specific shared subsets. The bar graph above indicates the total count of shared clones, and is ordered by decreasing similarity. For all donors the shared plasma cell fraction (CD19high and CD19low shared clones) are colored purple, plasma cell clones also seen in the periphery are colored red, plasma cell clones also seen in the BMMC B cells are teal, and gold indicates clones seen in every compartment sequenced.

In all six donors, the BMMC and PBMC residing B cells show extremely high similarity to one another; this large number of shared sequences is to be expected, as both subsets include the highest number of total clones sequenced, and many newer B cells leaving the bone marrow are actively entering the periphery. The only donor with a compartment overlap greater then between the PBMC and BMMC fraction was donor Old2, with over 14000 clones shared between the periphery and the long-term resident CD19low plasma cells. All donors showed a large similarity between the long term and short term resident plasma cell compartment. This may be explained by many plasma cell members being in a transitory state of many months resident in the bone marrow, with the total clones unique to each compartment being the true new residents (for CD19high expression) and very long-term residents (for CD19low expression).

The older donors in general showed the expected trend of a high similarity between the periphery and new CD19high plasma cells, and between the bone marrow B cells and the CD19low older plasma cells. The younger donors seemed to have a wider variability between the periphery and any plasma cell fraction. Only donor Young3 showed a high correlation between the PBMC B cells and all plasma cells, dwarfing even the shared bone marrow B cell clones also seen in the plasma cell fractions.

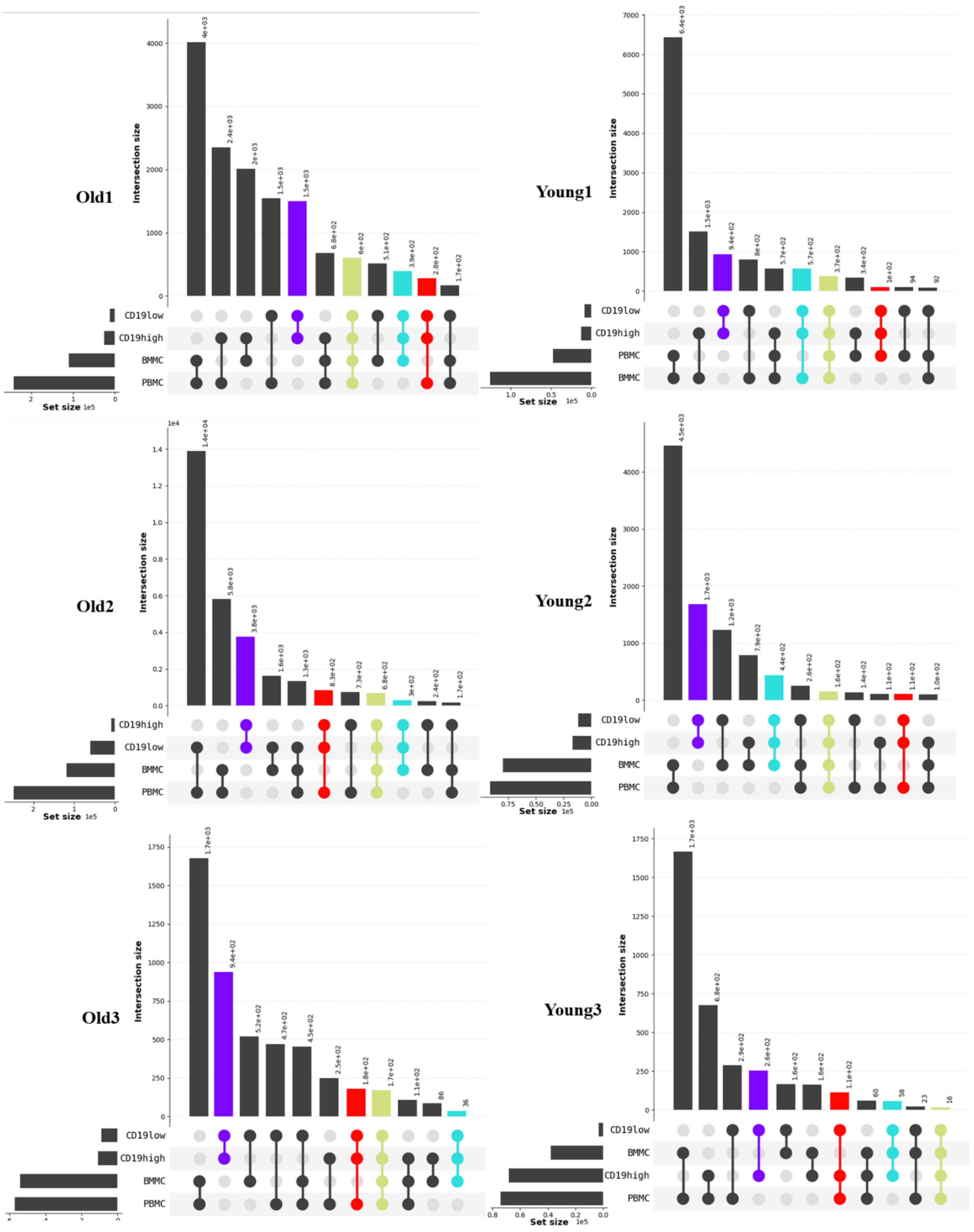

Old1
Young1
Old2
Young2
Old3
Young3
Intersection size
Set size
CD19low
CD19high
BMMC
PBMC

Figure 1.9.

Figure 1.9 (previous page): UpSet repertoire comparisons based on the number of shared clonotypes between population sets. The bottom rows show the donor compartments and their total number of clones, with each column representing the shared subsets of compartments (indicated by darkened circles) ordered by maximum similarity (shared clones). The total number of shared clones is indicated in the bar graph above the sets. Shared sets of interest are colored by type; purple sets are the set of clones only seen in the plasma cell compartment (CD19high and CD19low), red sets are clones seen in the plasma cells and the PBMC derived B cells but not bone marrow, teal sets are seen in all bone marrow compartments (plasma cells and BMMCs) but not the periphery, and gold sets are clones seen in all compartments sequenced.

Proteomic Analysis of Total Serum IgG Repertoires

While the quantitative probing of the actual proteomic repertoires has seen great strides in recent years due to ultra-high resolution mass spectrometry, studies utilizing IgSeq have almost entirely focused on small repertoire subsets. By enriching for antibodies specific to an antigen, the overall diversity of a sample is vastly reduced; this focus on fewer clones has been necessary to overcome resolution limitations of LC/MS. As such, little is known about the composition of total serum IgG. One major complication for clonotype identification is the lack of a robust BCR sequence library – while large PBMC-based datasets do yield high-confidence matches, peripheral blood B cells account for only 2% of all B cells in the body. Access to a spectrum of sequences from many compartments not only greatly increases the chance of identification but may lead to a far more biologically-relevant picture of the serological repertoire.

Total serum IgG that was collected from serum by Protein G-agarose chromatography was trypsinized and run over six hours on a Thermo Orbitrap mass spectrometer after C18 reversed-phase separation to maximize peptide resolution; the resulting raw datasets were then parsed by Thermo Discover 1.4 using SEQUEST with a pooled database of quality filtered clonotype sequences including all donor compartment libraries. Despite the extended separation process, the total number of unique clone identifications did not surpass 200 members for any donor – a count many orders of magnitude less than expected. However, the clones that were identified showed a wide range of origins; aggregation of clone total areas by compartment of origin (unique clone seen in a single sample) suggests a proteomic repertoire heavily skewed towards an origin in the bone marrow (figure 1.10). This serves as some of the first physical evidence of a serological antibody origin in the bone marrow.

Average IgSeq clonotype summed areas for clones only identified in ANY bone marrow sequence dataset: 56.0% (standard deviation 10.0%). At least 40% of clonotypes identified in proteomic IgSeq in all donors were sequenced only in bone marrow compartments. Even more telling is the average of 26.6% of clones by total area deriving only from sequences in any plasma cell subset (standard deviation 12.6%). The longest-lived plasma cell fraction CD19low also showed a consistent prevalence across donors, with at least one unique clonotype found in the top 10 ranked members for all individuals.

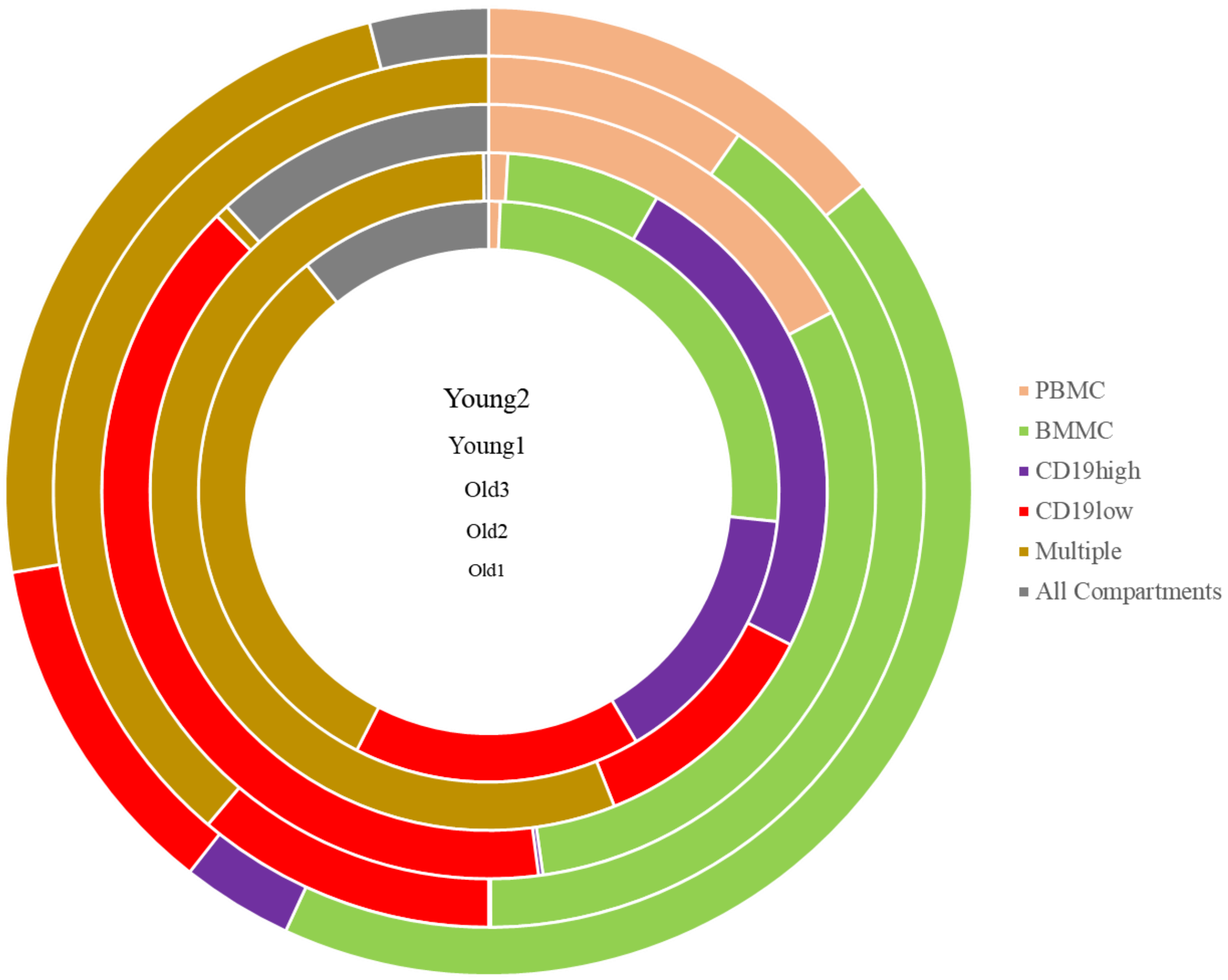


Figure 1.10: Clonotype sequence origin for total detected serum IgG members, weighted by the total elution area each compartment contributes. In order from the innermost ring to the outermost are donors Old1, Old2, Old3, Young1, and Young2 respectively. No data was collected for donor Young3.

Chapter 2: The Anti-Pneumococcus Capsular Polysaccharide Antibody Repertoire and Serological Response

BACKGROUND

A hallmark of the B cell's unique place in the adaptive immune response is in the extreme versatility and de novo complexity of its effector molecule; there are numerous protein families containing a variety of specificities for foreign materials such as bacterial DNA and lipopolysaccharide, but none can boast the many functions of the antibody. Even the T cell receptor, with its similar genetic structure and combinatorial variety, finds itself lacking the multi-faceted roles allowed by a soluble molecule with a constant region that interacts with not only many different cell types, but even non-adaptive ambient immunological proteins such as the complement system (Hoffman et al. 2016). For researchers in the field of the humoral immune system, this provides a whole new method of studying how the body recognizes and removes pathogens – but one that is equally as complicated to probe as it is critical to understand. Proteomic analysis of serum antibody repertoires is currently in its infancy, being far more fraught with challenges than the traditional next-generation sequencing of antibody transcripts or even genomic studies of B cells at a high throughput (Lavinder et al. 2014, Wine et al. 2015). The only real tool currently capable of identifying and characterizing protein sequences at scale is liquid-chromatography in series with an ultra-high sensitivity mass spectrometer (LC/MS); while the technology today is finally at the level allowing for identification of entire proteomes, many issues still hinder progress in the accuracy of sequencing an entire proteomic antibody repertoire. Unlike working with an RNA/DNA template, no method of amplification or selection of specific gene regions of interest exist in proteomics. Additionally, while antibodies as a group display a massive range of unique sequences the majority of the full molecule is identical to most other members. Since current methods of

LC/MS depend on a form of shotgun sequencing (in the form of tryptic peptides), reconstructing a highly variable non-genomic dataset is exceedingly difficult.

To break the daunting problem of antibody proteomics down into more manageable chunks, more specific questions can be targeted than the overall repertoire analysis possible with VH-only transcriptomics. By focusing on a subset of the proteomic repertoire and enriching a sample to only a small number of antibodies with related properties it is possible to cut out much of the noise and quantitatively characterize a population of interest. One method of selecting a smaller set of similar antibodies is by utilizing their intended natural function – purification of members against a specific antigen, such as the pathogen components that have been encountered naturally or via vaccination. The number of antibodies recognizing any specific ligand are at least several orders of magnitude less prevalent than those recognizing others, and the serological component of immunity in terms of a specific disease may be even more rewarding to study than the broader repertoire (Lavinder et al. 2015). Another major benefit of studying a specific response to common vaccines is in the increased control of sample timing; donors can provide samples at important temporal junctions, and the likelihood of creating a memory response and proliferation of desired B cells in the donors leads to a better sequence database for the subsequent LC/MS search.

One such antigen of interest is found in the capsular polysaccharides of the bacterium *Streptococcus pneumoniae*. *S. pneumoniae* is a common and deadly source of many cases of meningitis and pneumonia worldwide, responsible for over a million deaths annually worldwide (Torres et al. 2015). Vaccines against the pathogen exist, targeted mainly at the deadliest variants (serotypes), but these often have a much diminished efficacy in children and the elderly. The widely used Pneumovax-23 vaccine may be especially hypoactive in these groups; rather than a protein antigen, the vaccine is comprised of the bacteria's outer capsule polysaccharides for 23 different serotypes. Unlike protein antigens, which are targeted by both T cell and B cell effectors, polysaccharide antigens cannot be identified by T cell receptors. Even the antibody response to the much less confined and diverse polysaccharide structure tends to be consequently less specific and robust (Gonzalez-Fernandez et al. 2008; Weller et al. 2005). The major responders – especially involved in the secondary memory response – may be derived from an IgM memory B cell population expressing so-called "natural" antibodies (Shi et al. 2005). These antibodies tend to have lower rates of somatic hypermutation and a wider range of ligands for a single antibody than usually expected from an immunocompetent memory-response.

Memory B cells act as a major component of the rapid secondary response by their ability to proliferate into antibody-secreting plasma cells. Memory B cells express the surface CD27 receptor, allowing for easy selection of the subset. While a majority of Memory B cells express IgG antibodies, a small subset does retain the IgM constant domain; these cells may have skipped germinal center mutagenesis entirely, with SHM and proliferation taking place in the marginal zone of the spleen instead (Tangye et al. 2007). This population appears to differ in overall repertoire profile as well, further suggesting a unique origin (Bagnara et al. 2015). Supporting this idea, individuals whose spleens have been removed and the young with underdeveloped marginal zone regions are far more likely to become infected by bacteria with capsules (Zandvoort et al. 2002).

CD27+ Memory B cells are known to be highly enriched for antibodies binding simple bacterial membrane components such as phosphorylcholine and capsular polysaccharides (Fiskesund et al. 2014). IgM Memory B cells in particular often take the role of a "natural effector" cell, targeting membrane antigens even with very little to no germinal center involvement (Weller et al. 2005). Activation of this Memory population may occur not in the systemic lymph, but in the marginal-zone like regions of the gut

(Hamada et al. 2002). As bacterial antigens are omnipresent in the gut, the IgM+ CD27+ subset may be used as specialized detectors of bacterial antigen given the lack of a T cell response. Marginal Zone B cells also express the innate immunity Toll-like receptor, which allows the B cell to respond even if an antigen is not bound by the BCR. Despite their key role in early identification of encapsulated pathogens, the contribution of this Memory B cell population to protective levels of serum antibody (either natural or higher affinity) is not known. This study attempts to characterize and quantify the T-cell independent serum response to the polysaccharide pneumococcal vaccine, using the high-risk 6B serotype antigen as a probe to separate the contributions from various B cell populations involved in the response.

MATERIALS AND METHODS

Serum IgG Purification and Preparation

For all donors and timepoints, 5 mL of serum frozen at -80 °C was thawed to room temperature and centrifuged for 5 minutes at 15000g to pellet debris, then diluted 1:1 with 1x PBS pH 7.4 and filtered through a 0.22 μm syringe filter. For each separate sample, 1.5 mL of Protein G+ agarose (Thermo Scientific) resin was placed in a 5 mL polypropylene protein purification column (pre-washed once with 5 mL 70% ethanol), then equilibrated with 3x 5 mL PBS washes at room temperature. The serum sample was then added to the resin and allowed to flow through by gravity; flowthroughs were collected and run an additional two times to ensure complete binding of antibody to the resin. 2x 1 mL PBS washes were pooled with the flowthroughs, then 3x 10 mL PBS washes were collected separately. 8x 1.5 mL Eppendorf tubes were prepared each containing 0.1 mL 1M Tris pH 8.0 to immediately neutralize the elution fractions; 8x acid elutions using 0.9 mL of 100 mM glycine pH 2.7, collected into the prepared tubes. Protein concentration estimates were performed via Nanodrop absorbance readings, and all fractions containing antibody were pooled. The pooled antibody samples were then dialyzed overnight using a 10 kDa MWCO dialysis membrane (SnakeSkin dialysis membrane) at 4C into 1x PBS, pH 7.4.

To remove the antibody constant regions to assist with LC/MS proteomic analysis, the samples were digested using IdeS enzyme, which is IgG-specific and cleaves only in the hinge region (IdeS protease produced in-house). IdeS was added at a 1:50 ratio of enzyme:IgG, and the mix was incubated for 2 hours at 37C. After digestion, a 10 µg aliquot of digested IgG was run on a reducing SDS-PAGE gel to ensure proper cleavage.

Anti-Pneumococcal Polysaccharide Antibody Enrichment

To create a substrate suitable for column chromatography, purified 6B polysaccharide was conjugated to NHS-agarose beads based on protocol described in Ey, P., 1993. To create an aminohexyl derivative prior to linkage, 1 mg of 6B polysaccharide (ATCC) was first dissolved in 1 mL of 0.2 M sodium bicarbonate, pH 9. 880 µL of 0.1 M 1,6-diaminohexane pH 9 (adjusted with HCl) and 20 µL of 0.1 M sodium periodate were added and incubated for 15 minutes on ice. 100 µL of sodium borohydride was added and the reaction was kept on ice for 60 minutes in the dark. The resulting solution was then dialyzed overnight at 4 °C into 4 L of 0.1 M sodium bicarbonate, and a fresh 4 L of buffer was exchanged for another 12 hours at 4 °C. After this second 12 hour dialysis, the solution was again dialyzed overnight at 4 °C into sodium binding buffer (100 mM sodium phosphate, 150 mM NaCl at pH 7.2).

Aminohexyl-6B conjugation to N-hydroxysuccinimide (NHS) activated agarose was started by dissolving 132 mg of NHS-activated agarose (Pierce) and incubating the solution rotating at room temperature for 60 minutes. The tube was then spun down for 2 minutes at 1,000g and the supernatant pulled out and saved. The resin was then washed 2x with 1.5 mL of PBS and supernatant was saved, then brought to 1.5 mL with 1 M ethanolamine to block the remaining reactive sites on the resin. After 30 minutes rotating at room temperature, the resin was equally split into two columns and centrifuged at 1,000g for 60 seconds followed by 2 washes with 400 µL of PBS.

For purification of 6B-specific antibodies, serum IgG in PBS was run through the 6B column three times, collecting the flowthroughs separately. Four sequential wash steps were done, each step with 400 µL of various wash buffers:

- 5x washes with 1x PBS (binding buffer)
- 3x washes with 50 mM Tris, 100 mM NaCl pH 7.4 (final wash saved)
- 117 mg of NaCl, 1 mL of 1M Tris pH 7.4 in a final volume of 20 mL
- 5x washes with 5 mM phosphocholine / 100 mM borate buffer pH 8.4 (all saved)
- 381 mg of sodium borate, 16.5 mg of phosphocholine in a final volume of 10 mL
- 3x washes with 150 mM NaCl pH 7.2 (final wash saved)
- 438 mg NaCl in a final volume of 50 mL

Columns were eluted 5x with 400 µL of 3.5 M MgCl2 pH 3.5, collected with 40 µL of 1 M Tris pH 8 to neutralize.

RESULTS

Sample Subsets and Experimental Outline

To characterize the humoral immune response to a perceived Pneumococcal challenge, four healthy adult donors were given the Pneumovax-23 vaccine and a matched set of peripheral blood samples were acquired on days 7, 14, and 28 after vaccination. From these samples, peripheral blood mononuclear cells were purified by density gradient centrifugation; from these cells a fraction were stained and sorted by FACS, selecting for the Plasmablast, IgD+ Memory, IgD- Memory, and C27+ memory cellular subsets depending on sample. RT-PCR libraries using either IgG or IgM specific constant primers were created and sequenced using Illumina MiSeq technology. Post read quality filtering and clustering, each donor compartment repertoire contained an average of 5994 unique clonotypes ranging from a minimum 307 clones up to 30050 clones (table 2.1). These sequence databases were pooled by donor, and the pooled dataset was used for identification of tryptic peptides from polysaccharide-binding IgG.

To select for and characterize anti-6B polysaccharide IgG, serum IgG collected by Protein-G agarose resin was run over custom 6B-NHS agarose columns; columns were washed thoroughly to remove general non-binders and general anti-cell wall component binding antibodies (figure 2.1). The elution fraction was then digested with IdeS to remove constant region sections, digested with trypsin, and run on high resolution Orbitrap LC/MS. Raw datasets were identified using the donor sequencing libraries via Thermo Proteome Discoverer software.

Donor	Day	Compartment, Isotype	Raw Reads	Filtered Reads	Clones
448	7	Plasmablast, IgM	66481	12269	307
448	7	Plasmablast, IgG	297346	82338	1157
448	7	PBMC, IgG	552374	148378	2055
448	14	IgD+ Memory, IgM	271322	45974	3462
448	14	IgD- Memory, IgG	205803	57694	14922
449	7	Plasmablast, IgM	150259	27424	292
449	7	Plasmablast, IgG	145220	36679	798
449	7	IgM+ IgD- Memory, IgM	2024572	422053	6763
449	14	IgD+ Memory, IgM	312978	51555	3199
449	14	IgM+ IgD- Memory, IgG	2195939	178389	972
449	14	IgD- Memory, IgG	275054	63519	3728
450	7	Plasmablast, IgM	68716	14773	267
450	7	Plasmablast, IgG	19679	5189	453
450	7	CD27+ IgM+ Memory, IgM	956912	81314	2615
450	14	IgD+ Memory, IgM	182713	28397	7460
450	14	IgM+ IgD- Memory, IgM	198026	17212	3785
450	15	IgM+ IgD- Memory, IgG	947297	84644	16712
450	34	IgD+ Memory, IgM	511599	101803	30050
450	34	IgM+ IgD- Memory, IgG	400823	5268	2904
GCI	7	Plasmablast, IgG	8853	1181	299
GCI	13	IgD+ Memory, IgM	510462	122182	16968
GCI	13	IgM+ IgD- Memory, IgG	277210	59801	5057
GCI	28	CD27+ Memory, IgG	454202	122364	13635

Table 2.1: Donor library sequencing statistics and unique clonotype count.

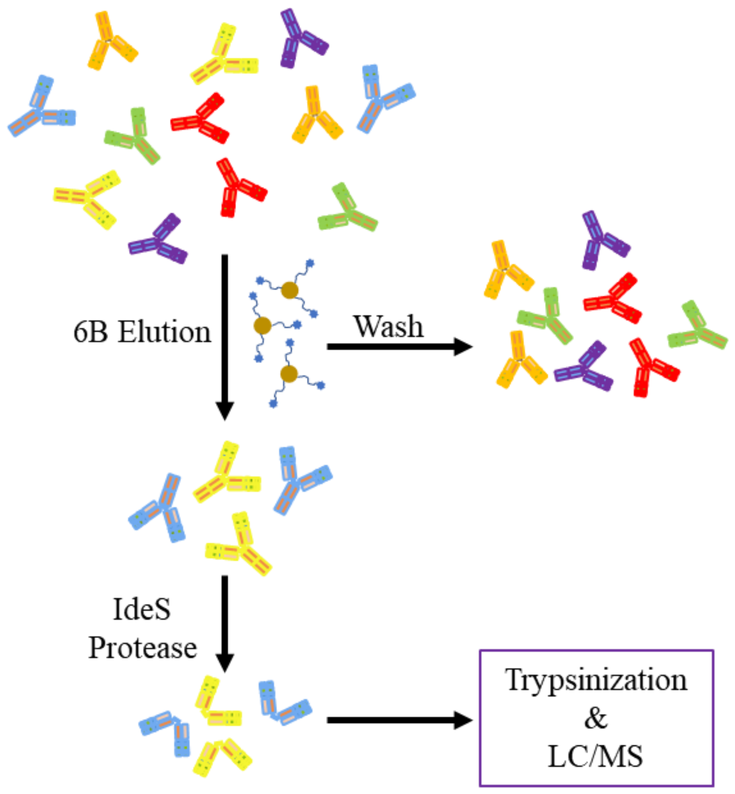


Figure 2.1: Sample processing pipeline for enrichment of anti-6B polysaccharide antibodies for LC/MS proteomic sequencing. Total serum IgG was run over 6B-linked agarose beads, washed to remove non-binders, and eluted by low pH and high salt concentration. Enriched IgG was then buffer exchanged to remove salt followed by IdeS digestion to cleave the constant region and then tryptic digestion. Peptide digest was cleaned by C18 reverse phase chromatography prior to LC/MS.

High-Throughput Sequencing Derived Repertoire Characteristics

For all donor sequencing samples, the resulting datasets were processed via a standardized pipeline that removed low-quality read sequences then aligned validated reads against the human immunoglobulin locus and clustered alignments based on a 95% similarity between CDRH3 sequences using MiXCR. Even accounting for total read sampling, day 7 plasmablast datasets from all donors showed far fewer overall clones than other cell types (893 clonotypes on average in the plasmablast compartment versus 14568 in memory B cells); this smaller subset of clones was expected to be directly resulting from the vaccinal challenge. In three of the four donors, a large proportion of the total plasmablast repertoire corresponded to only the top 5 clones identified (mean of 27.5% of all clonotypes); donor 449 showed a far less polarized plasmablast response, which may result from a hyporesponse to the vaccine itself as far fewer plasmablasts were isolated in the day 7 FACS collection (Figure 2.2).

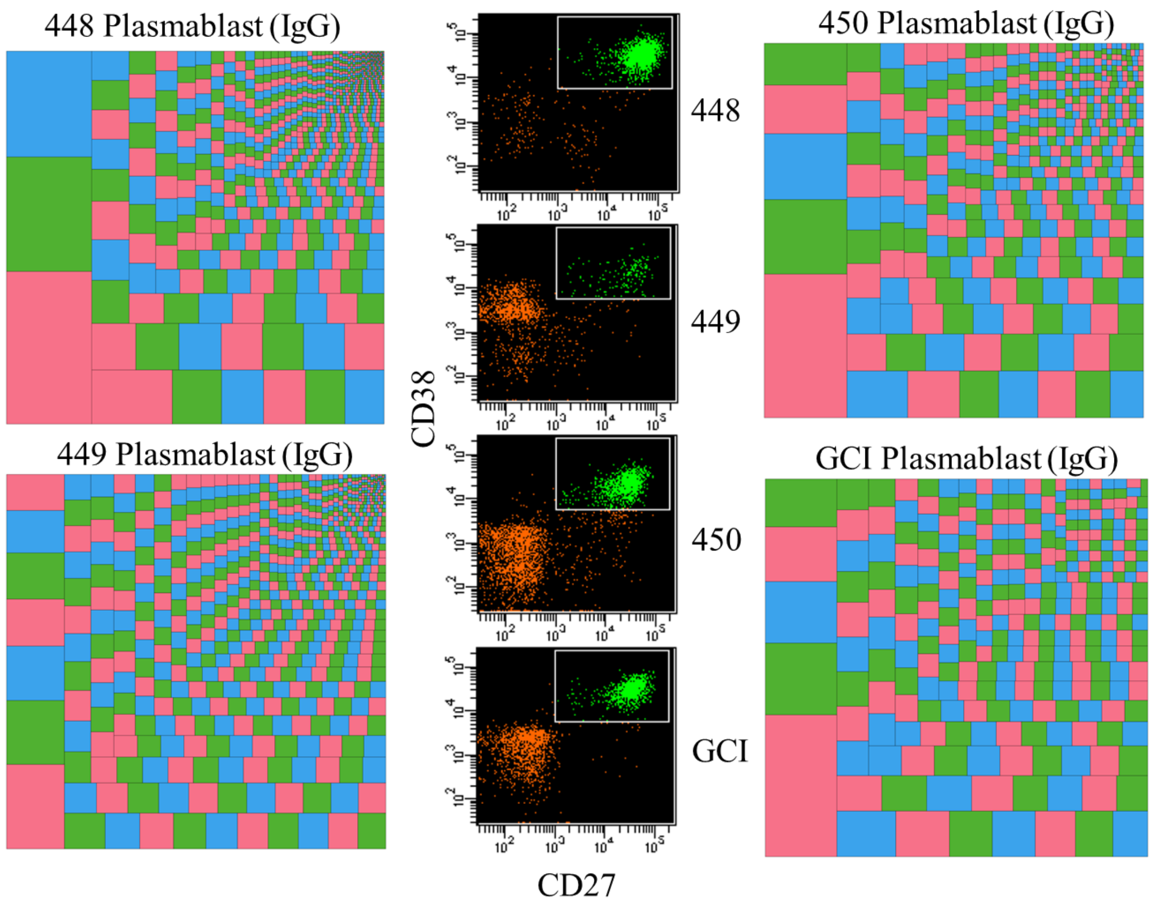


Figure 2.2: Day 7 plasmablast FACS sorting results and IgG plasmablast repertoire clonal mosaic profiles. The plasmablast cell sort was gated based on B cells with high CD27 and CD38 expression levels, as shown in the boxed selection. Clonal mosaics demonstrate the overall size of each clonotype in the repertoire, with the most prevalent clones starting at the bottom left of the diagram. Colors for clones are repeating for clarify and are uninformative.

Overall V gene somatic hypermutation (SHM) rates between cellular subsets were found to be highly dependent not on the compartment but on the isotype of the antibody expressed; average SHM rates among IgG-expressing cells were 8.4%, with IgM at 6.2%. Interestingly, the top responding clones in the plasmablast fractions of both isotypes were found to be more highly mutated by around 2.5% than the average of all clones as shown in the clonal mosaic profile in Figure 2.3. Unlike the differences seen in SHM, there was no appreciable distinction in CDR3 amino acid length when compared either between isotypes or among all sequencing samples (average of 16.3 amino acids).

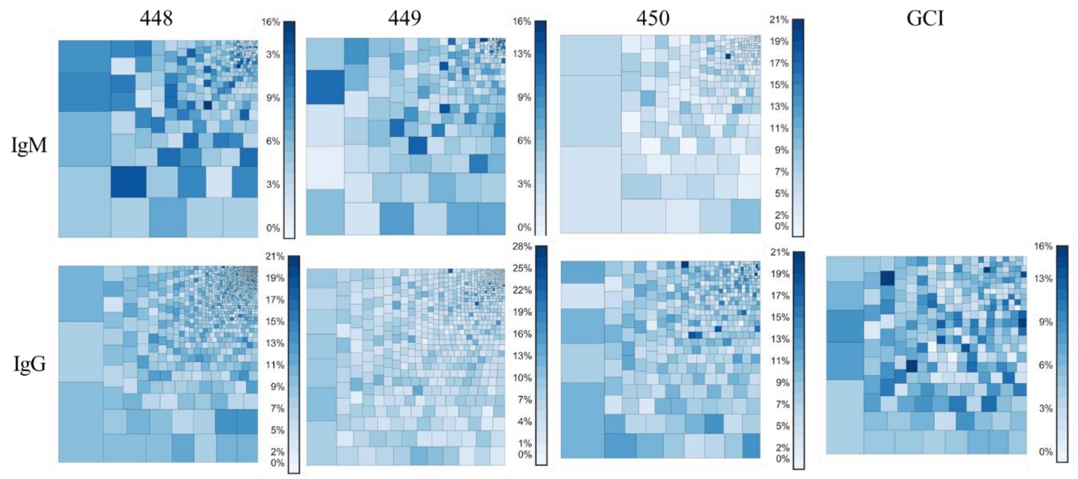


Figure 2.3: Day 7 Plasmablast V gene SHM clonal mosaics for all donors, separated by IgM and IgG isotype. Individual clones are colored as a heatmap representing V gene mutation rates, with corresponding heatmap colorbars to right of each sample.

The overall V gene usage in the presumed vaccine-enriched clonotypes was found to be strikingly similar across donors. The V family 3 genes were used in the majority of day 7 plasmablast clones, and the most commonly used V gene for all donors in the plasmablast compartment was IGHV 3-30. The overall diversity of V gene usage was found to be much higher in the memory compartments, with the V family 1 and 4 genes being slightly more common than family 3. J gene usage was also highly polarized in all donor plasmablast samples, with IGHJ4 being the foremost gene at an average of 45% in all plasmablast clones. In contrast, IGHJ6 was highly enriched in the memory B cell samples and was found in an average of 38% of total clones.

Compartment Repertoire Overlap

As the selected sequenced B cell subsets for each donor were all collected within a close timeframe, some overlap between identified clones between each compartment are to be expected. For example, some members of the expanded plasmablast cells collected would presumably be seen to have high similarity to memory B cells from which they originated. To gauge the overall similarity of clonotypes found in the different cellular samples, sequencing libraries post-quality filtering were pooled and clonotyped in aggregate based on V and J gene usage and 90% amino acid CDRH3 sequence similarity. To visualize these relationships between subsets, a variety of graph formats can be used. While Venn diagrams are succinct and easily distinguished for 2-3 intersecting samples, 4 or more intersecting samples sets are far more clearly seen in a novel format known as the UpSet plot as described by Conway et al. 2017 and shown in Figure 2.4.

Donor 448's clonotypic overlap showed an unsurprisingly high relation between the day 7 plasmablast IgG subset and the overall day 7 PBMC cells. Within this relation however, 120 clones were shared not only by the IgG plasmablast and PBMC cells but were also highly related to the day 14 memory IgM population. Also of note was the similarity of many IgM plasmablast clones and the day 14 memory IgG-expressing cells, suggesting common progenitor memory B IgM-expressing cells that produced both IgM plasmablasts and class-switched IgG memory cells. A smaller group of clones were seen in both the IgM and IgG isotypic plasmablast groups; in general, the day 14 IgM memory B cells were the most dissimilar to all other groups.

As the least responsive of all donors to the vaccination, donor 449's overlap profile appears to show fewer sequenced plasmablast clones that have a high correlation to observed memory B cells. Many clonotypes were seen in the memory B cell IgM-expressing population at both days 7 and 14. The plasmablast groups of both isotypes had relatively high overlap, and in general the plasmablast clones had the closest relation to memory B cells of the same isotype. An unexpected relation of note was seen between the day 14 memory B IgM and IgG cells with 300 clones being found in both populations.

Donor 450 was seen to have the expected high relation of clones in the Memory B IgM-expressing cells at all timepoints. This donor more than any other also seemed to have the highest overall similarity between the vaccine-induced plasmablast IgM cells and the memory B cell population of the same isotype at all time points sampled. At the latest day 34 timepoint sampled, potentially IgG class-switched memory B cells were seen that corresponded to IgM memory B cells at the day 14 timepoint; this may be indicative of vaccine-responsive cells that went through the traditional germinal center hypermutation and proliferation steps. Unexpectedly, 75 clones were seen to be shared between the day 7 plasmablast IgG group and the day 14 memory B cell population.

Finally, donor GCI showed the most similarity between populations of the same isotype; the memory B cell IgG population was self-similar at all timepoints, and many plasmablast IgG B cells appear to derive from this memory population. However, the day 13 memory B cells expressing IgM were only found to overlap with the day 13 memory IgG population; these may be members of a single originating group from which some cells underwent class switching. No IgM-expressing plasmablast population was sampled for this donor.

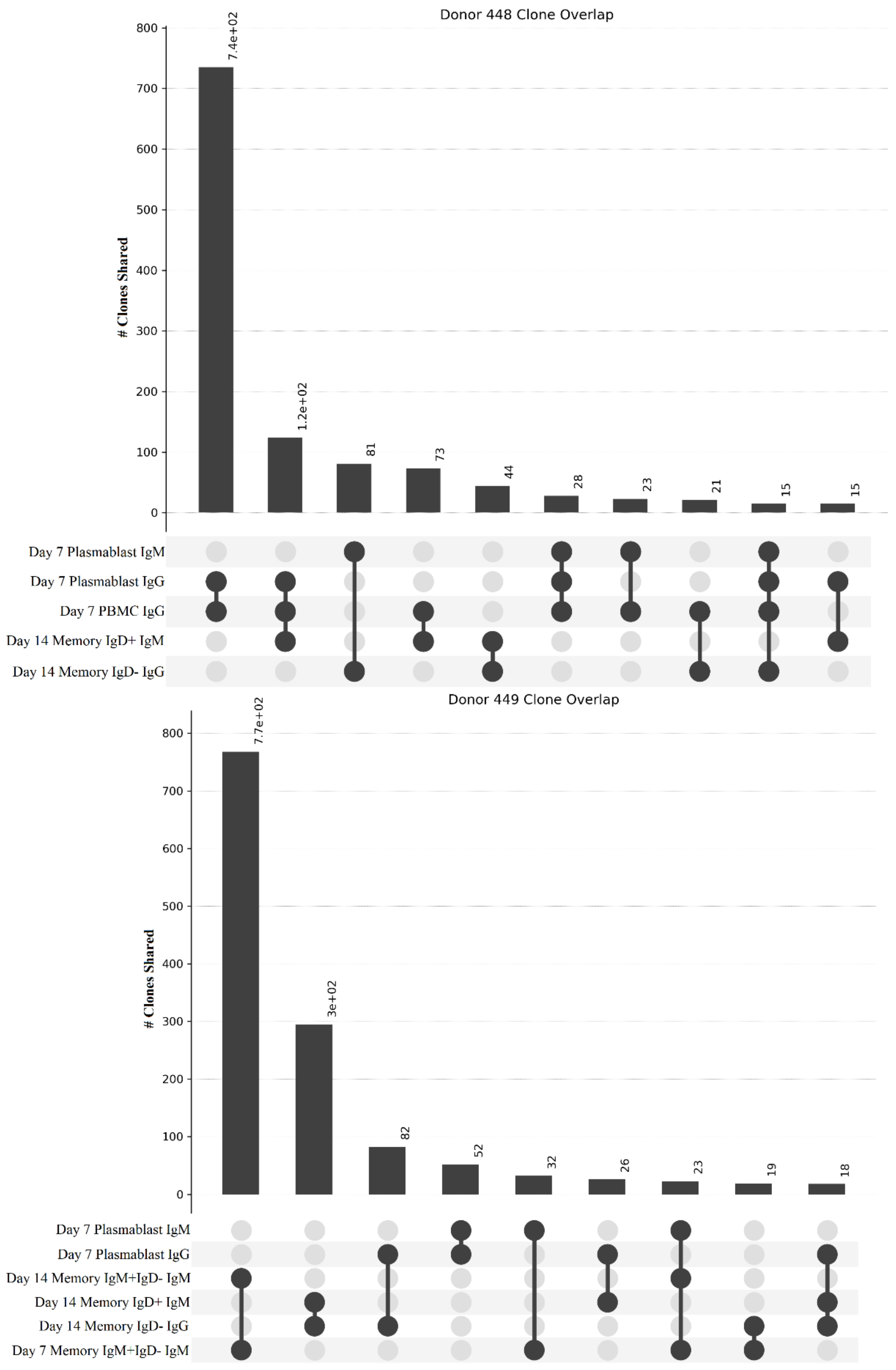
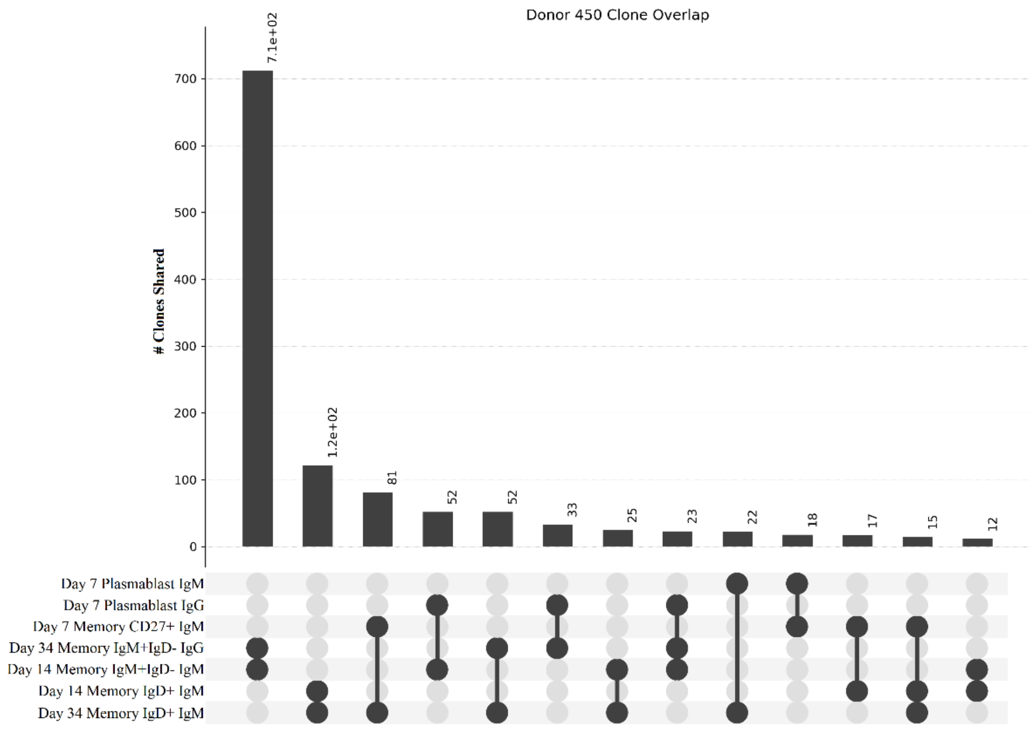

Donor 448 Clone Overlap
# Clones Shared
800
700
600
500
400
300
200
100
0
7.4e+02
1.2e+02
81
73
44
28
23
21
15
15
Day 7 Plasmablast IgM
Day 7 Plasmablast IgG
Day 7 PBMC IgG
Day 14 Memory IgD+ IgM
Day 14 Memory IgD- IgG
Donor 449 Clone Overlap
# Clones Shared
800
700
600
500
400
300
200
100
0
7.7e+02
3e+02
82
52
32
26
23
19
18
Day 7 Plasmablast IgM
Day 7 Plasmablast IgG
Day 14 Memory IgM+IgD- IgM
Day 14 Memory IgD+ IgM
Day 14 Memory IgD- IgG
Day 7 Memory IgM+IgD- IgM

Figure 2.4.



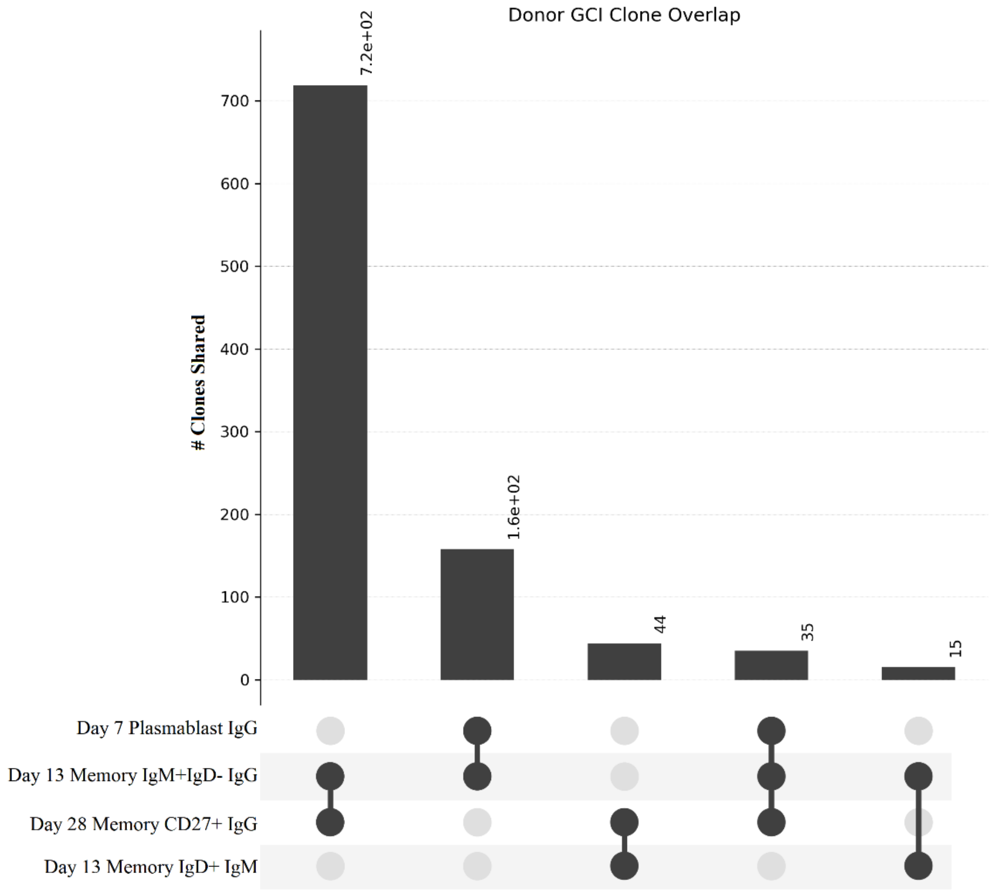


Figure 2.4, cont.

Figure 2.4 (previous page): UpSet clonotype overlap similarity plots for all donor sequencing libraries. The bottom section of each graph consists of each sequenced group, with shaded circles indicating the shared groups that would be displayed as overlapping circles in a Venn diagram. The upper bar plot displays the total number of shared clones in the shared subset indicated below the bar; plot bars are sorted descending by total number of clones shared amongst each sample set.

Vaccine Component Enriched Serological Repertoire

While high-throughput sequencing datasets of specific B cell populations at specific timeframes after vaccination can give a general picture of the changing repertoire in response to immunological challenge, the actual effectors involved in eradication of the perceived threat – the soluble antibody itself – cannot be directly measured by next-generation sequencing alone. As such, quantitative characterization of the serological response requires observation of the actual expression and antigen-binding capacity of these effector molecules. Recent advances in mass spectrometry allow for high-resolution identification of relatively complex protein samples; however, current limitations of these methods do not allow for the depth of sequencing required to identify all antibodies present in whole-blood serum samples as can be performed with the transcriptional repertoire. To simplify this issue, the serological response to vaccination was limited to a single component of the vaccine. The *Pneumococcus* serotype 6B capsular polysaccharide antigen was chosen as a model of a highly pathogenic challenger, and from total serum antibody an enriched subset of 6B-binding antibodies were purified as described in Methods. The 6B elution antibodies from all donors were compared by mass-spectrometry proteomics to the non-binding flowthrough, and top clones were chosen by the amount of enrichment seen in the elution.

As shown in figure 2.5, it is evident that many individual clonotypes appear to derive from a single compartment – the clone is unique to a specific cellular subset. Although the plasmablast subset should consist heavily of newly proliferating cells responding to the vaccine challenge, unique sequences coming from the Memory B cell compartment seem to have a much larger role in the 6B response. Another surprising observation is the high similarity of the anti-6B repertoire compared before immunization and at day 32; the top clone present maintained its rank for all donors, and the overall ratio of top clones in day 32 strongly correlate to day 0. The largest change in top clones was seen in the hyporesponsive donor 449; only two of the top ten clones in the day 0 elution carry over to day 32. The overall anti-6B repertoire also appears to be highly polarized – very few clones make up the majority of the response (top 5 clones making up 38% to 64% of the response by total area). This could indicate one of two explanations; first, prior exposure to the Pneumovax vaccine or the 6B serotype lead to high affinity antibodies that persist long term. Alternatively, the top ranking members could simply be a part of the widespread natural antibodies which are adept at binding the more repetitive and simplistic capsular polysaccharide. The latter case may be the stronger explanation, as one would expect a larger shift after vaccination were the day 32 repertoire representative of a re-expanded memory response.

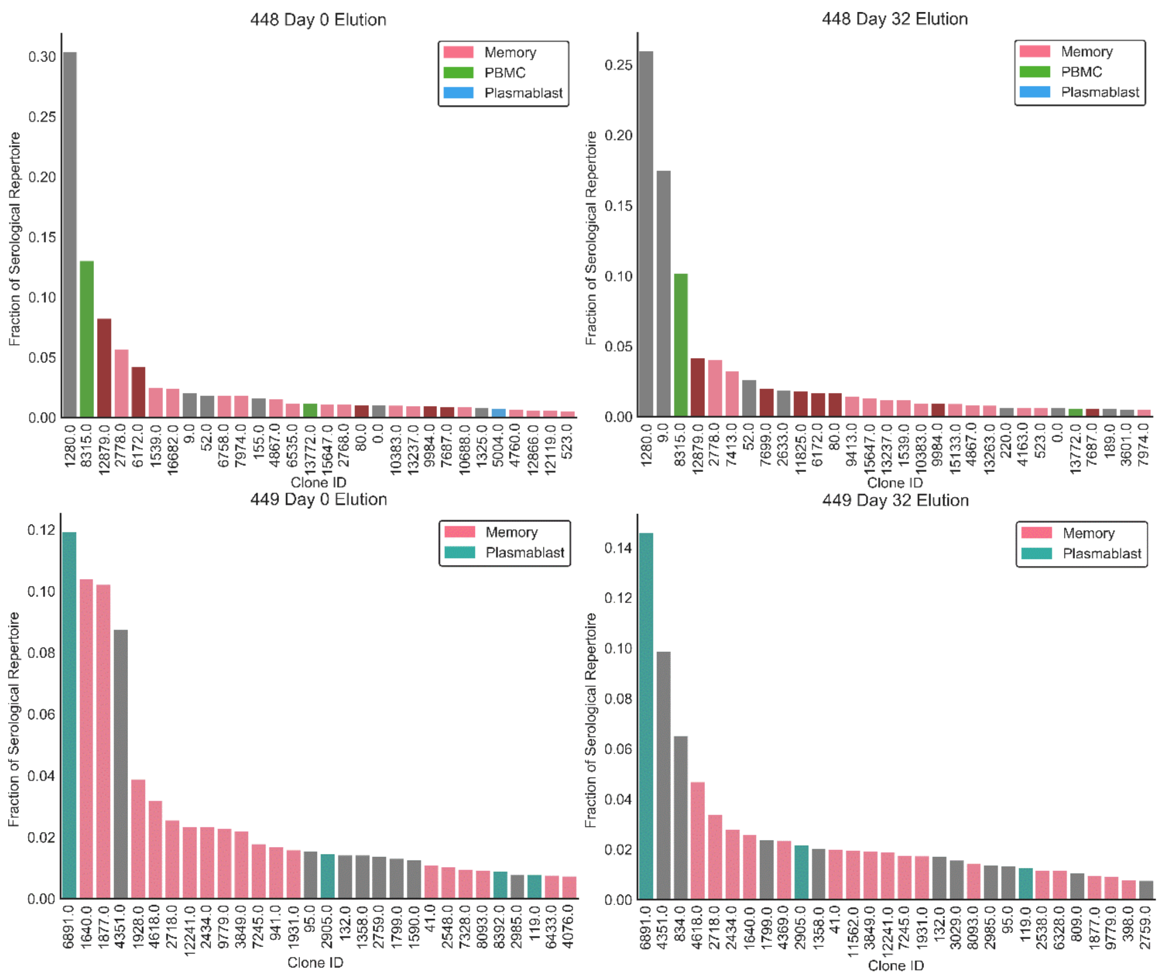


Figure 2.5a: Top proteomic clonotypes enriched by 6B-polysaccharide chromatography and LC/MS IgSeq. Clones identified as a specific compartment such as "Memory" indicate that the clonotype is derived from sequences that were only seen in the Memory B cell libraries (no contribution from other compartments). Grey bars signify that a clone is present in multiple compartments or is shared by all.

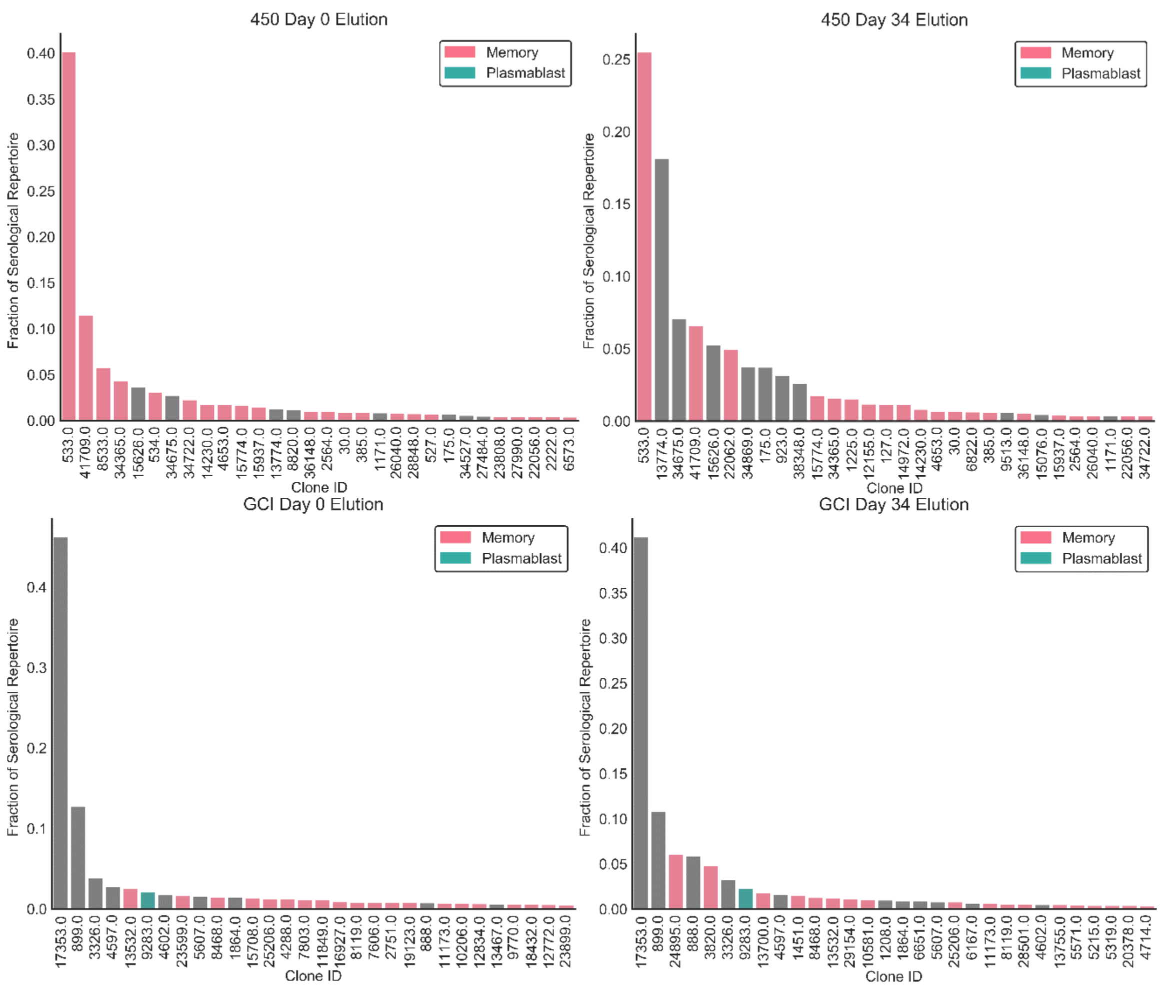


Figure 2.5b: Top proteomic clonotypes enriched by 6B-polysaccharide chromatography and LC/MS IgSeq. Clones identified as a specific compartment such as "Memory" indicate that the clonotype is derived from sequences that were only seen in the Memory B cell libraries (no contribution from other compartments). Grey bars signify that a clone is present in multiple compartments or is shared by all.

Contribution of Shared Clones to Serum IgG Response

Shared public clonotypes between donors were not only found as members of interest in the sequencing data alone. As evidence of their potential importance in the response to T cell independent antigens, some shared clones with similar CDRH3 sequences were found in the enriched 6B-specific proteomic elutions of multiple donors as well (table 2.2). As an example, one prevalent shared clonotype found in the 6B-responding elutions (CDRH3 sequence CARSLWPEDYW) in donors 448 and 449 was found in the day 0 elutions for both donors at rank 67 and 123 respectively, and also in the day 32 elution for donor 449 at rank 106. Every clonotype found in the IgSeq 6B-enriched fraction found shared by multiple donors was from the V gene family 3 or 4, and all had a generally low V gene SHM of 5-9%. These clones are the strongest examples of natural antibodies – often pre-class switched IgM effectors which act as the first responders to bacterial capsular antigens. In support of this idea, several of the identified CDR3 clonotypes from each donor (CARAWRVDSVMPKRYFDFW, CARSRGAMATLRGKRGYYGMDVW, CARGNVDRSMVYNFFDPW, and CVKLGYRAPDDPW) were only found in the day 13-14 Memory IgM-expressing B cells, and were not seen in any IgG-expressing B cell sequence datasets.

CDRH3 Sequence	**Rank in Elution (Per Donor)**	**V Gene**	**V Gene SHM % (Per Donor)**
CARGRNNFRVW	448: 123; 449: 66	IGHV3-7	448: 7.1%, 449: 7.9%
CARAWRVDSVMPKRYFDFW	448: 76, 449: 154	IGHV4-31	448: 9.7%, 449: 9.7%
CARSRGAMATLRGKRGYYGMDVW	449: 3, 450: 43	IGHV4-34	449: 7.2%, 450: 7.0%
CARGNVDRSMVYNFFDPW	449: 5, 450: 31	IGHV3-72	449: 5.1%, 450: 4.9%
CVKLGYRAPDDPW	449: 19, 450: 86	IGHV3-7	449: 8.5%, 450: 8.8%
CARQVQDAMDVW	449: 224, 450: 95	IGHV3-51	449: 2.5%, 450: 2.5%
CARSLWPEDYW	448: 67; 449: 106	IGHV3-7	448: 8.1%, 449: 7.9%

Table 2.2: Shared public CDRH3 sequences found in multiple donors and all in the 6B-enriched IgSeq mass spectrometry proteomics elution fractions. Clone ranks are displayed for each donor, along with the V gene SHM of the clone as seen in each donor.

Chapter 3: Automated Comparative Repertoire Visualization

BACKGROUND

In the new era of massive high-throughput repertoire sequencing datasets, gaining a picture of general sample characteristics is critical but incredibly difficult to visualize. While simple statistics can be easily calculated, much more comprehensive figures that quickly show the distributions of many factors in the high-dimensional repertoire are far better for qualitative analysis and comparison. Spreadsheet / statistical programs such as Excel can be used to create figures, but they are often limited by few chart types, limited configuration of a plot, and file size restrictions that disallow work with entire datasets. These limitations all but require at least an intermediate knowledge of a programming language to reasonably parse and visualize big data like the antibody repertoire. This can be a harsh restriction for many researchers, and often leads to repertoire analyses that are completely different amongst individuals. For this reason, a tool that can aggregate information and produce a variety of figures in a standardized manner for any library would be of great use for initial analysis and discussion between scientists. This chapter describes such a tool, utilizing the Python programming language and the Bokeh data visualization library to create a simple program that produces a dashboard with a variety of clear figures given either a single repertoire or a collection of repertoire datasets for comparison. Thanks to the much more dynamic graphics libraries available to the scientific community, the dashboard figures are interactive; charts can be easily zoomed or panned, and hovering tooltips provide information otherwise lost in a static figure. The analyses and output chart types are discussed below, and the Python source code is provided in the appendix for review and modification.

APPLICATION OVERVIEW

CDR3 Amino Acid Length Histogram (Spectratype Plot)

One of the earliest methods of visualizing characteristics of the BCR / TCR repertoire was through the measurement of the length of the CDR3 region. By using V gene-specific primers for PCR amplification of complex samples and running the results on a polyacrylamide gel to separate by size, the distribution of the product lengths is easily quantified by measurement of the brightness of each band and plotted as a histogram known as a spectratype plot (Gorski 1995). This method served as the first high-throughput means of analyzing immune repertoires and is still a commonly used measure of hypermutation and for general repertoire comparison. As full-scale repertoire sequencing is now possible, CDR3 spectratype analysis can be performed bioinformatically after identification of VDJ genes and the CDR3 region sequence (often using the translated protein sequence for histogram binning).

For the automated comparison of spectratype histograms between repertoires, data is provided as a table of clonotypes including either the CDR3 sequences or their previously calculated amino acid lengths. A clonotype histogram is calculated using the binned segment lengths using Numpy's histogram statistical function, normalizing the total density sum to unity (figure 3.1). For comparation of multiple samples each sample histogram is calculated separately, then each sample bin is plotted sequentially with a customizable color for each repertoire as shown in the legend (figure 3.2). For interactive visualization the plot can be panned and zoomed, allowing for specific box regions of the plot to be highlighted to demonstrate differences of interest between the samples (figure 3.3).

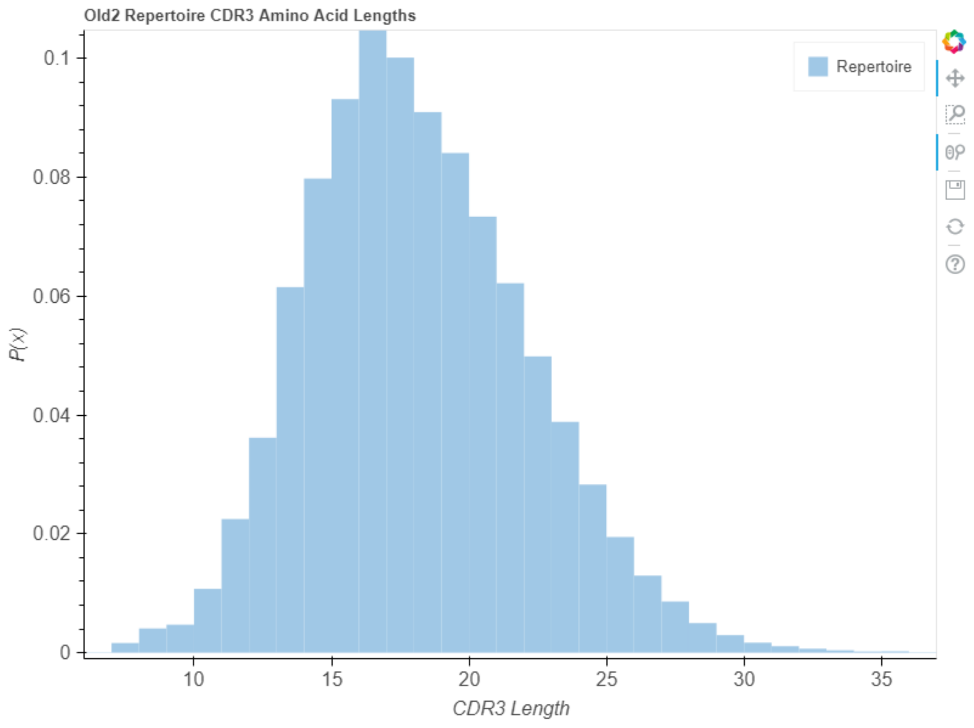


Figure 3.1: A standard CDR3 amino acid length spectratype for a single repertoire.

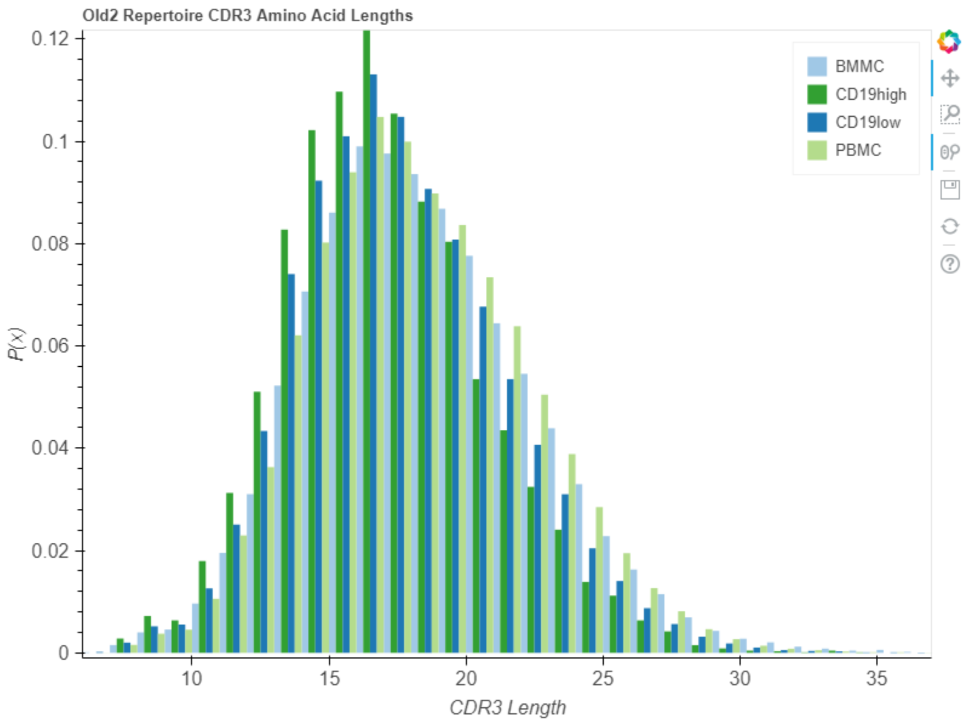


Figure 3.2: Comparative repertoire CDR3 amino acid length spectratype.

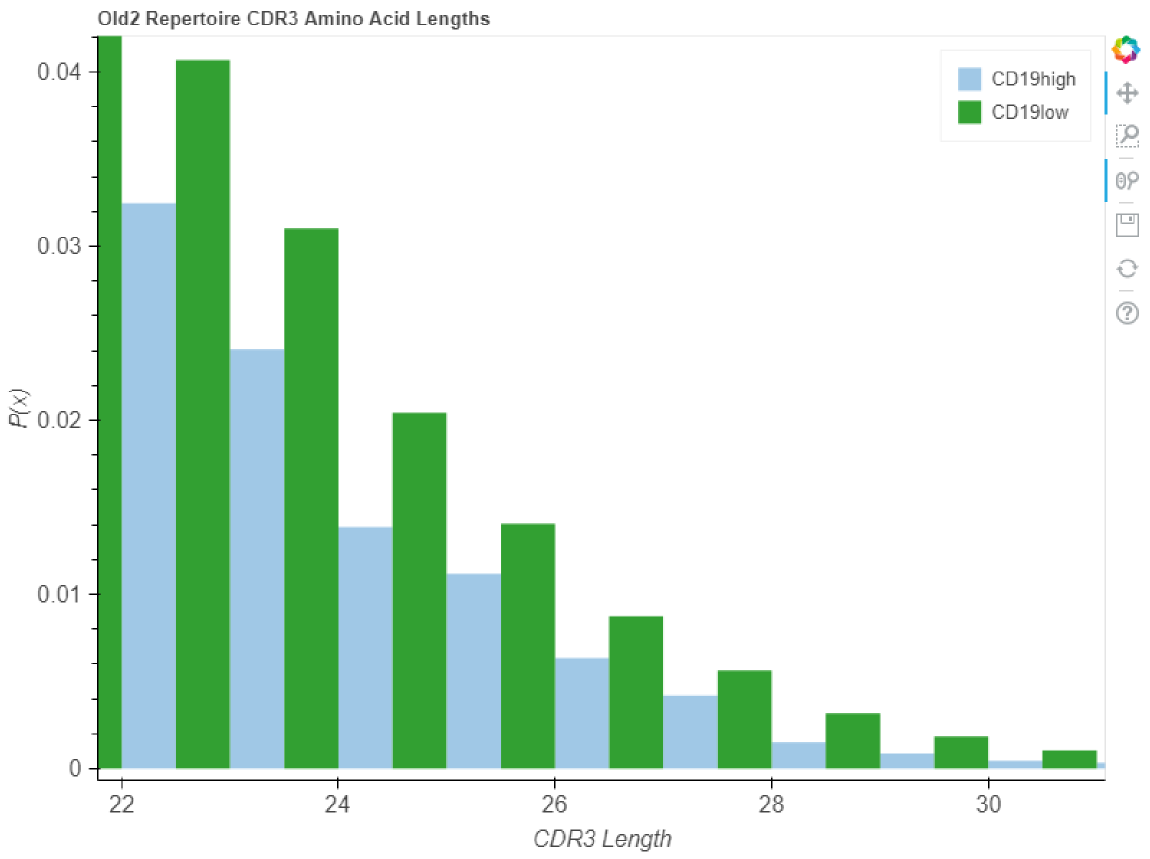


Figure 3.3. Zoomed-in region of CDR3 spectratype highlighting the differences between two repertoires.

V/J Gene Somatic Hypermutation Violin Plot

While CDR3 length analysis has been possible for decades, high-throughput analysis of the average hypermutation of individual clones has only become possible with full heavy chain sequencing of total repertoires. Gene annotation software such as IMGT's HighV-Quest and MIXCR are used to identify the germline V, D, and J genes, and somatic hypermutation (SHM) levels can be calculated by counting the total nucleotide point mutations, insertions, and deletions (Bolotin 2015). Unlike the total length of the CDR3 region, clonal SHM levels serve as a proxy for overall selection and utility; plasma cells having undergone many rounds of SHM in the secondary immune organs can be distinguished from naïve B cell populations which should have much lower rates of mutation. While SHM occurs throughout the VDJ region, identification of the germline D gene is generally unreliable as the insertion / deletion of nucleotides during recombination significantly modifies the original sequence – as such, the V and J genes are used as more reliable measures of overall hypermutation.

For visualizing repertoire V and J gene SHM levels, commonly a histogram or box plot is used to show the distribution in the repertoire. However, a more recent variant of the box plot known as the violin plot has several advantages over other plots in quickly and informatively yielding overall characteristics. The violin plot (named after its visual similarity to the instrument) is a combination of a box plot and the density distribution of the data as with a smoothed histogram (Hintze 1998). The violin plot excels in efficiently communicating the overall statistical trends of a sample as with a box plot, but also allows for distinguishing between uniform or bi/multi-modal distributions as with the histogram.

For automated repertoire analysis, a repertoire's clonal SHM distribution can be plotted either with a separate, mirrored violin plot for the V and J genes separately or by placing the V and J distributions on each side of the violin. While this application can be used to plot the V and J genes separately, for concise graphs allowing quick comparison of samples all figures below split the violins between the genes for each sample. The basic plot takes a table of all repertoire clonotypes and plots each sample as would be done with a categorical boxplot; the violin widths are all normalized to unity, with labeled sample categories on the X axis and the overall gene SHM percentages on the Y axis (figure 3.4). The widths, colors of samples, and separation or combination of the V and J violins are all easily customizable. The violin density distribution is calculated using Scipy's variable kernel density estimator function with the Scott density function as standard (Scott 1992). To further clarify the sample statistics, interactive hovering tooltips are included for each sample violin – the user can highlight a violin region with the mouse (or tap the violin plot if using a mobile device) to reveal the average, maximum, and 25th and 7th percentile SHM values for the repertoire, and additional sample information can be easily added for display if desired (figure 3.5). As with the CDR3 length spectratype plot, the user can pan and zoom in on specific regions to more easily identify sample distributions (figure 3.6).

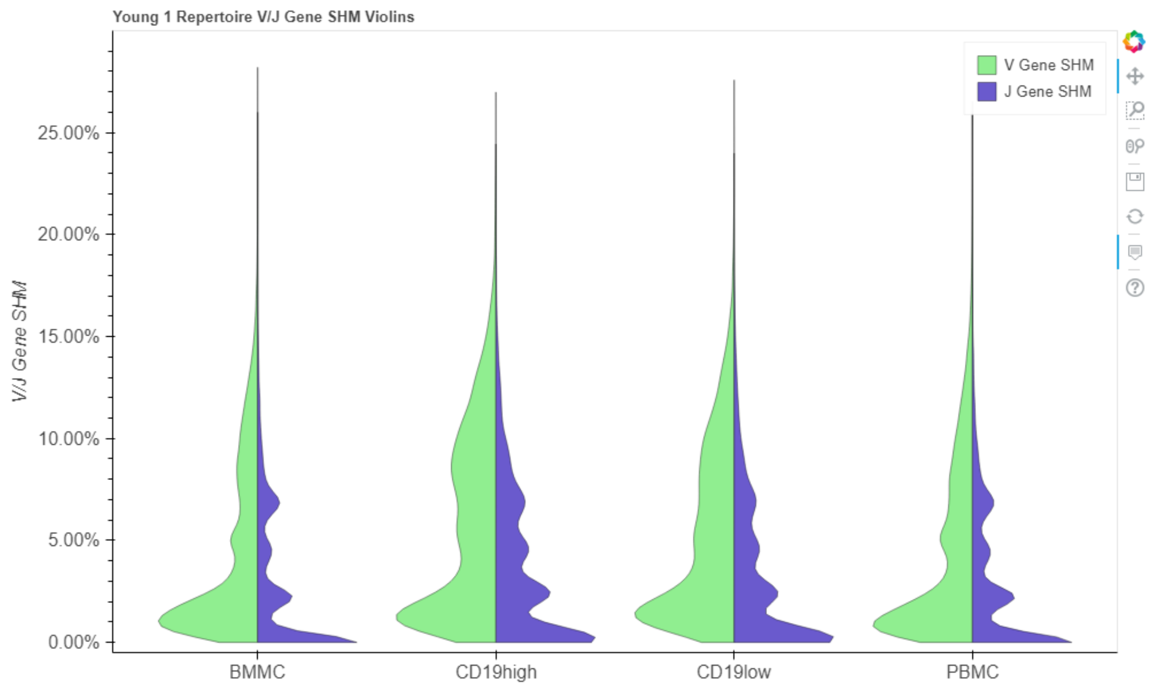


Figure 3.4: Violin plots demonstrating comparative repertoire clonotype V and J gene somatic hypermutation levels.

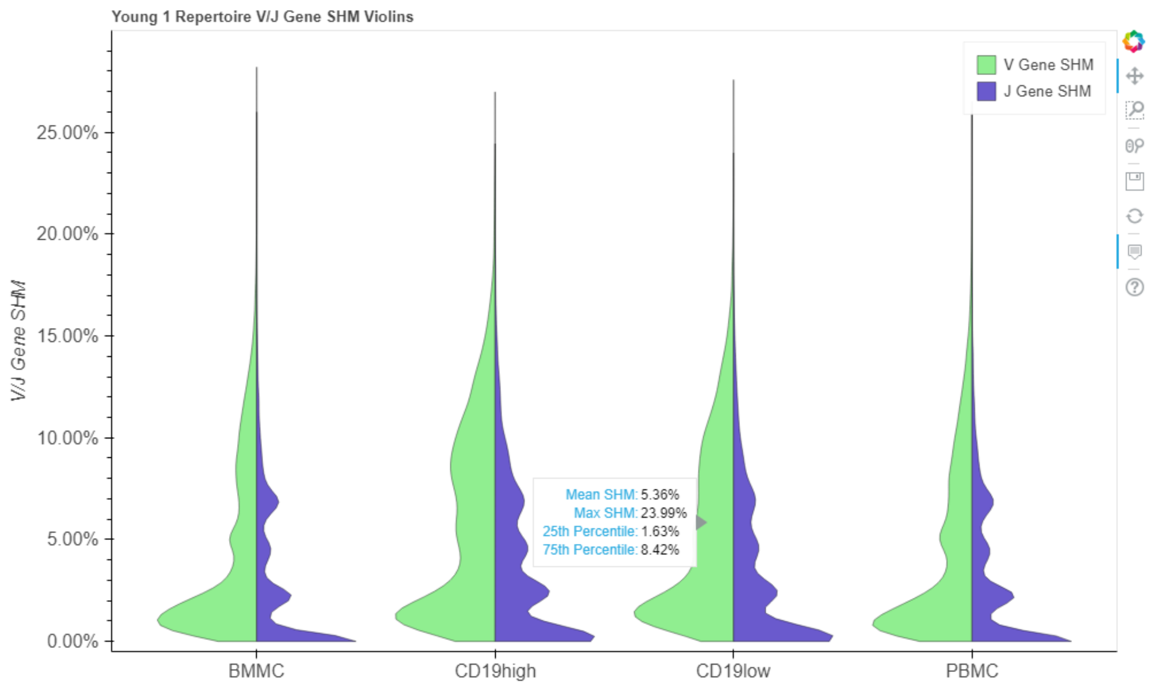


Figure 3.5: Demonstration of hovering tooltips displaying sample average, maximum, and 25th and 75th percentile SHM values upon selection of a sample violin.

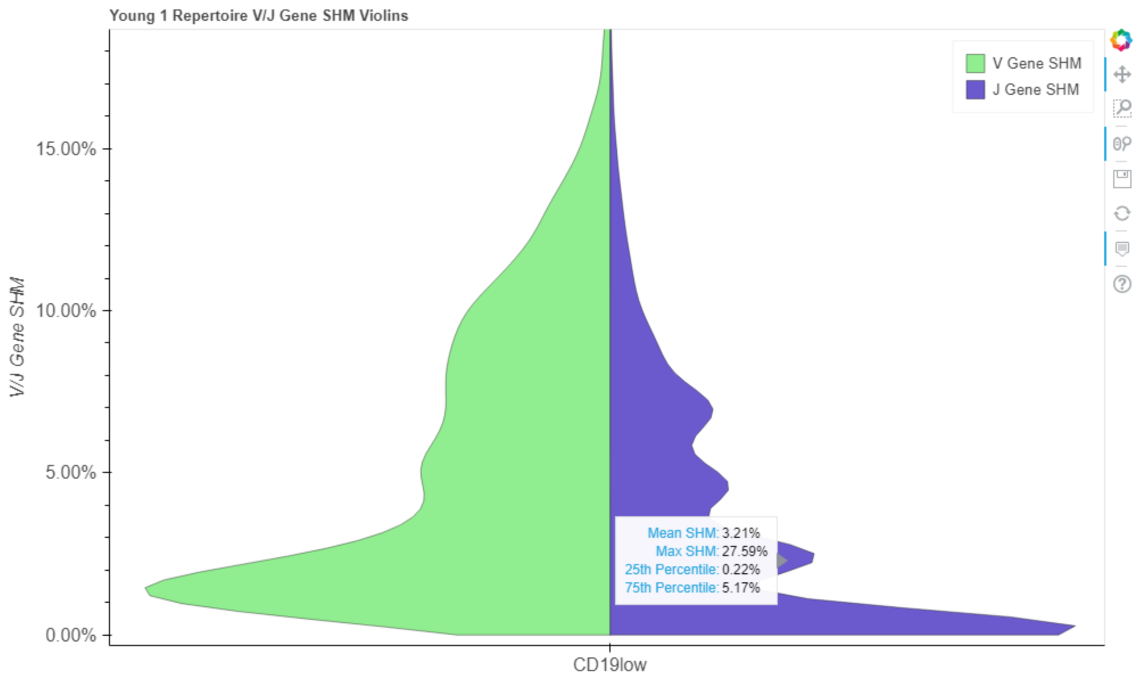
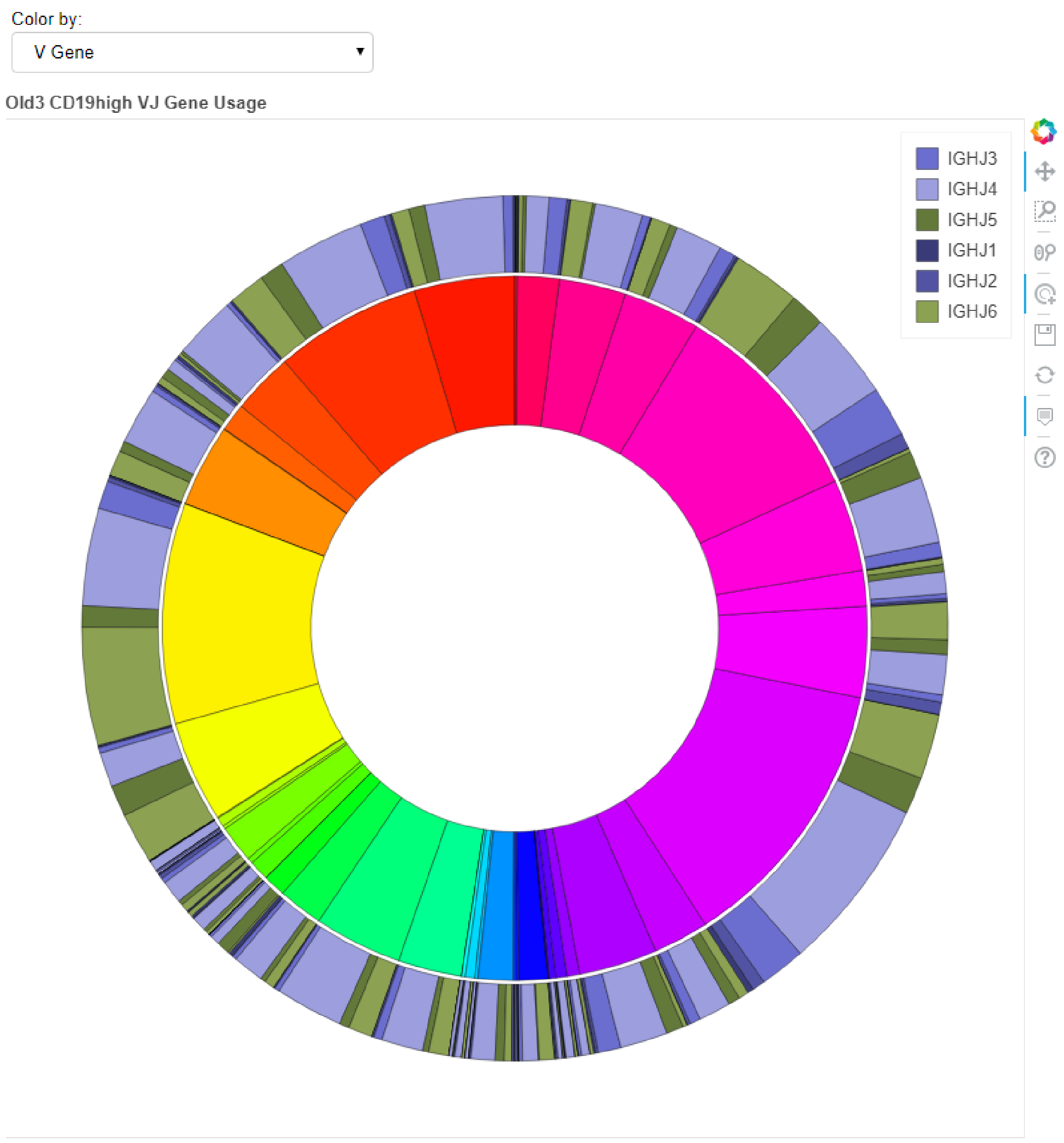


Figure 3.6: Zoomed-in selection of single sample from figure 5 for increased visibility of the sample SHM distributions.

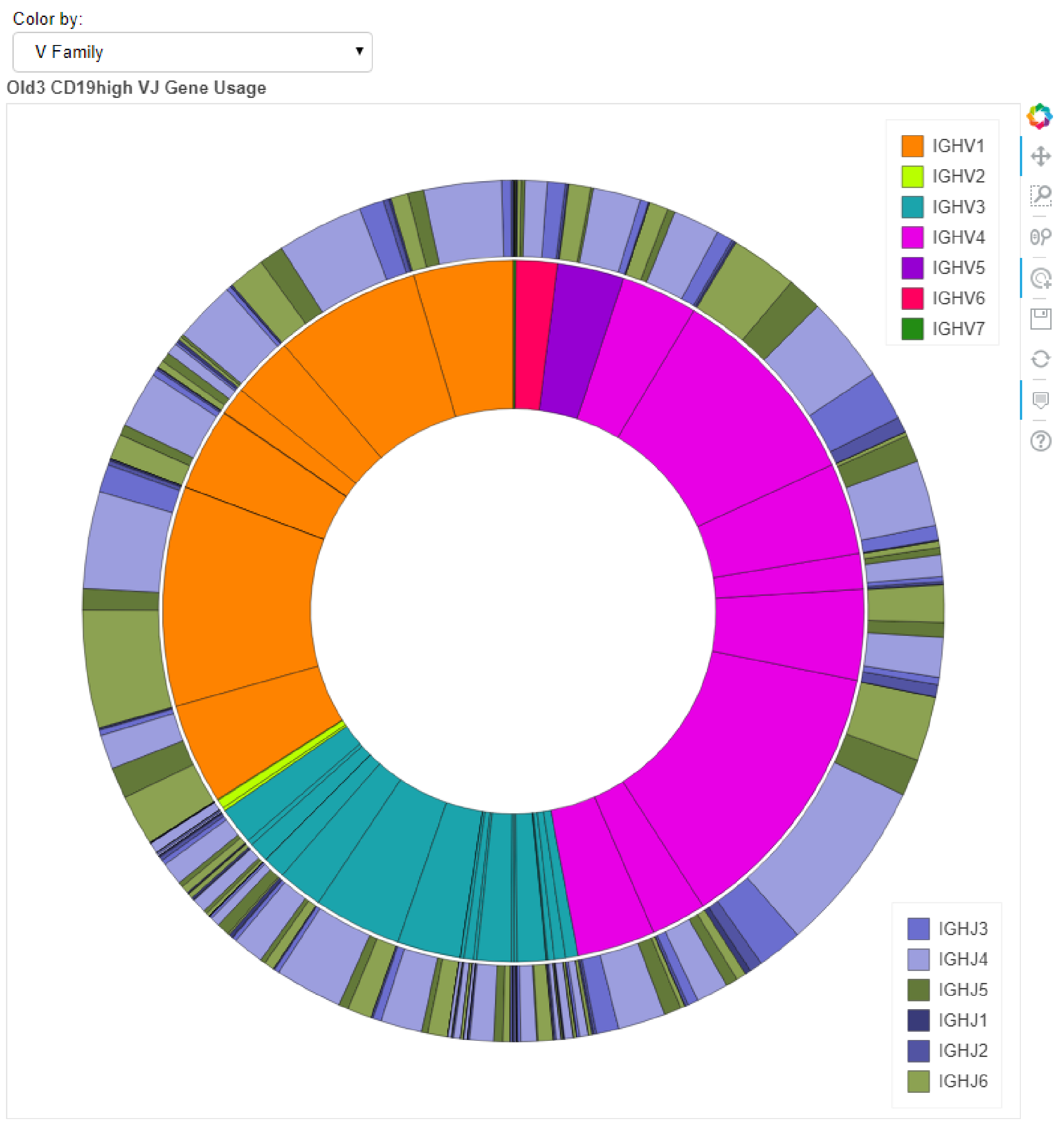
Paired V-J Gene Usage Donut Plot

While the V gene is often considered as the most informative gene region, the J region is also easily identifiable and contributes significantly to antigenic specificity. Correlation tables of the relative frequencies of V-J gene pairings are important to understand, but almost impossible to picture. To clearly demonstrate all sets of V-J pairings in a repertoire, a donut chart within a donut chart may be a superior method to quickly see how common specific pairings are. As seen in figure 3.7 below, the inner ring consists of all V gene frequencies while the outer ring consists of the cognate J gene percentages. For visual clarity, the graph can be colored by V or J gene or V family (figure 3.8). Users can also select specific genes, zoom into a region of the plot, and a hovering tooltip quickly shows which gene is selected and the percent composition in the full repertoire (figures 3.9, 3.10).


Color by:

V Gene

Figure 3.7: The V-J gene paired donut plot. The outer ring consists of the J gene frequencies as paired with the specific V gene in the inner ring. The chart can be colored by V or J gene.


Color by:

V Family

Old3 CD19high VJ Gene Usage

Figure 3.8: Demonstration of the chart colored by V family.

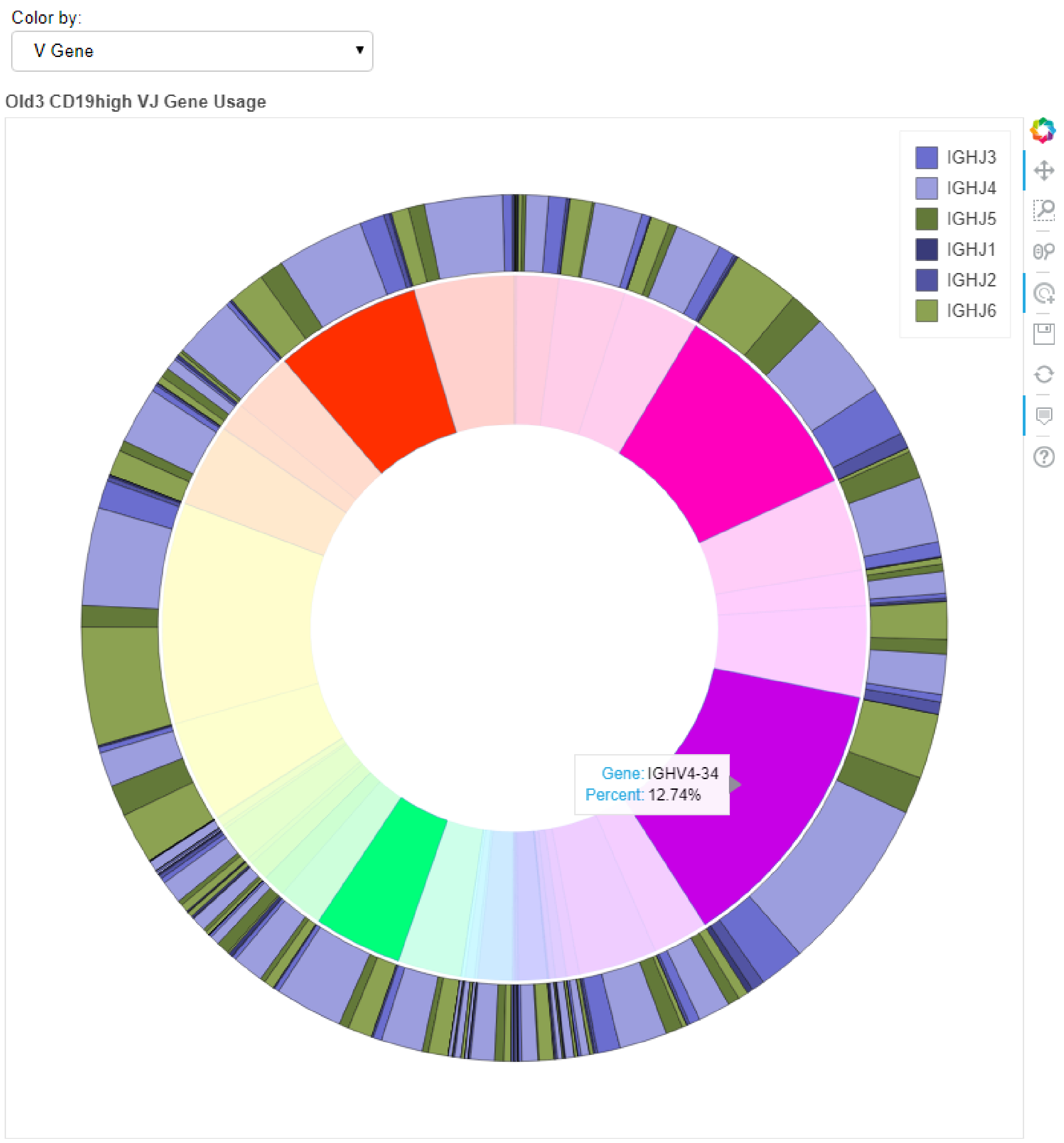


Figure 3.9: Selection of individual genes is available, with hovering tooltips displaying the gene and percent of the total.

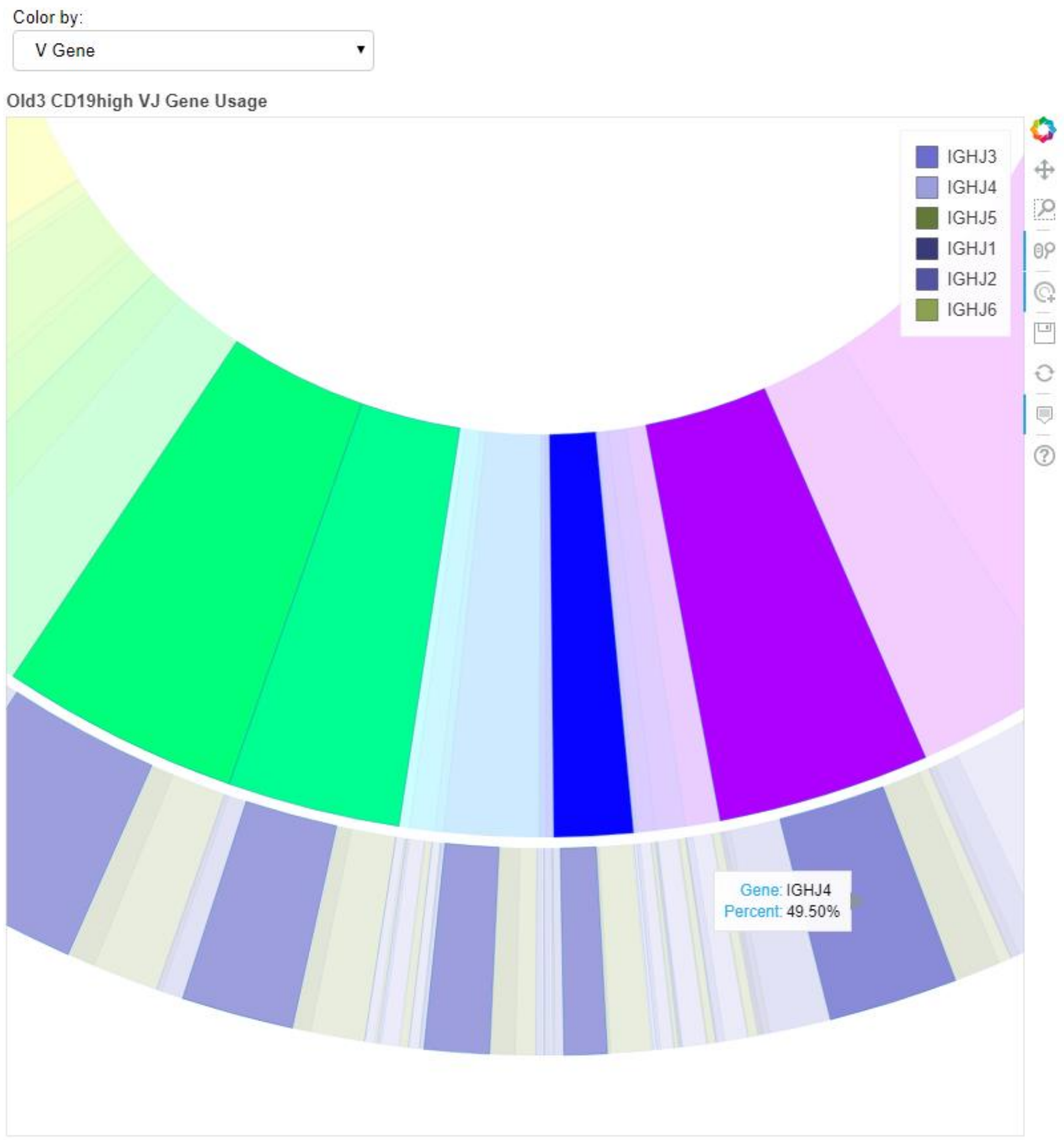
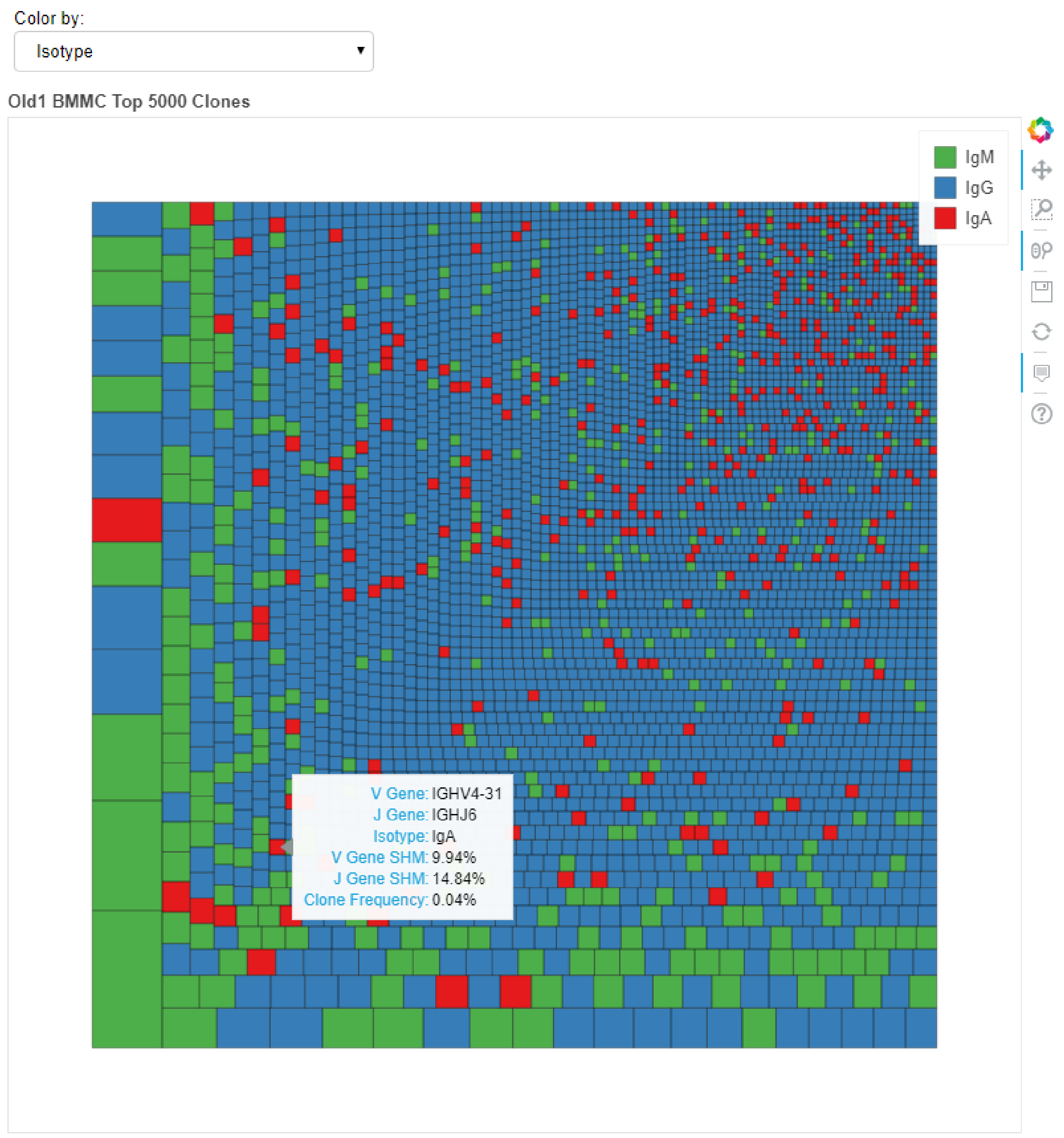


Figure 3.10: Zoomed in section of the chart with several V genes highlighted.

Categorical Clonal Frequency Mosaic Plot

Since the total number and frequencies of clones in a repertoire underlie the overall diversity and polarization, a common method of displaying these relative areas are using a square mosaic plot consisting of sorted rectangles sized in proportion to the clone's prevalence. As a repertoire may have thousands to hundreds of thousands of clones these charts may be difficult to interpret, but trends regarding the top clones are distinct and useful for comparison (figure 3.11). To further increase the utility of the clonal mosaic plot, clone rectangles can be colored based on category (such as isotype, V gene, or V family as seen in figure 3.12) or by continuous metrics like clonal V gene SHM. For continuous data like mutation rates, a heatmap is automatically calculated based on the range of the data and a mapping color bar legend is displayed (figure 3.13).


Color by:

Isotype

Old1 BMMC Top 5000 Clones

V Gene: IGHV4-31
J Gene: IGHJ6
Isotype: IgA
V Gene SHM: 9.94%
J Gene SHM: 14.84%
Clone Frequency: 0.04%

Figure 3.11: A repertoire clonal mosaic depicting all clones and their frequencies. Clones are colored by isotype, and a tooltip displays additional clone information.

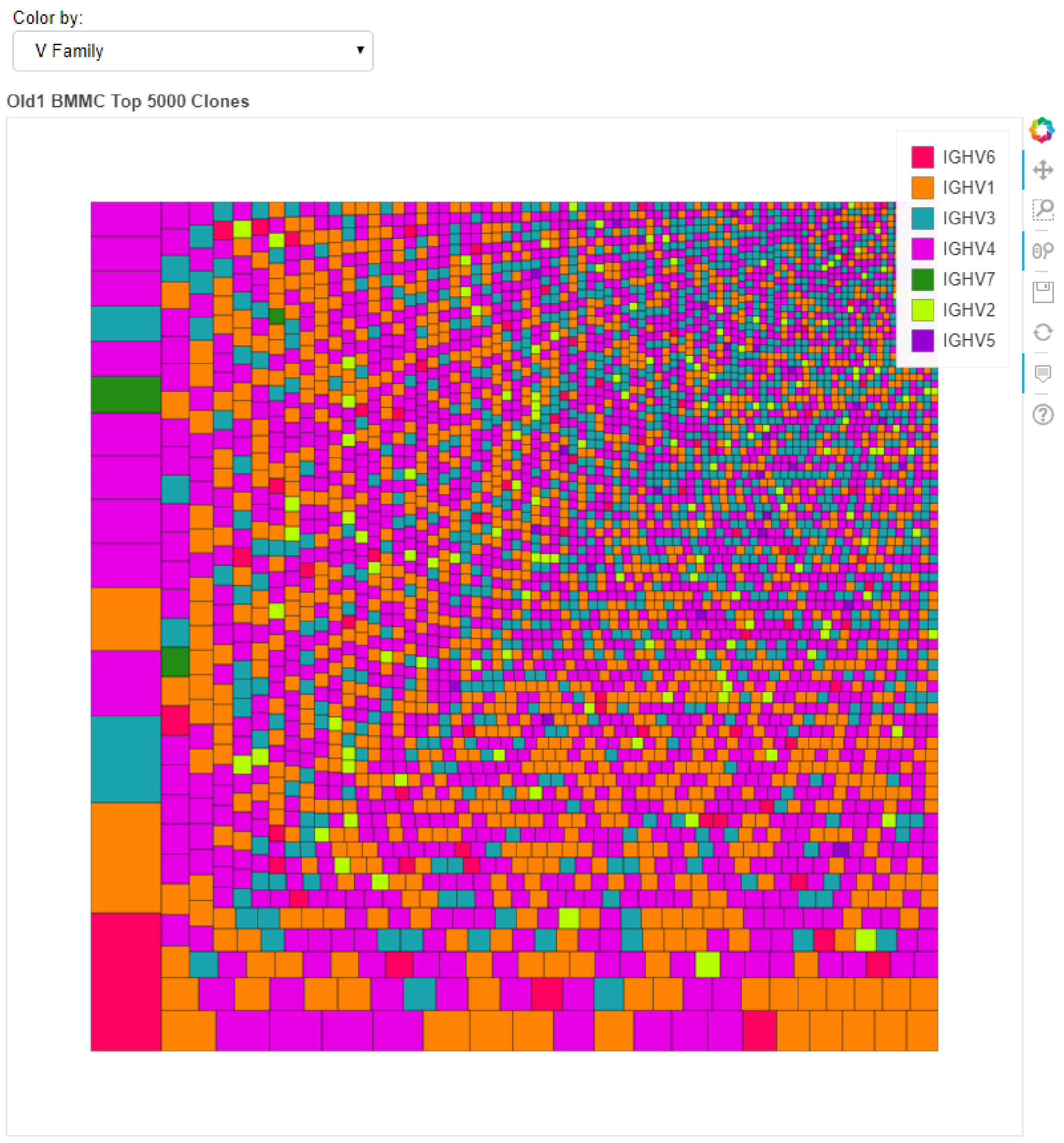


Figure 3.12: Repertoire from figure 3.11 colored by V family.

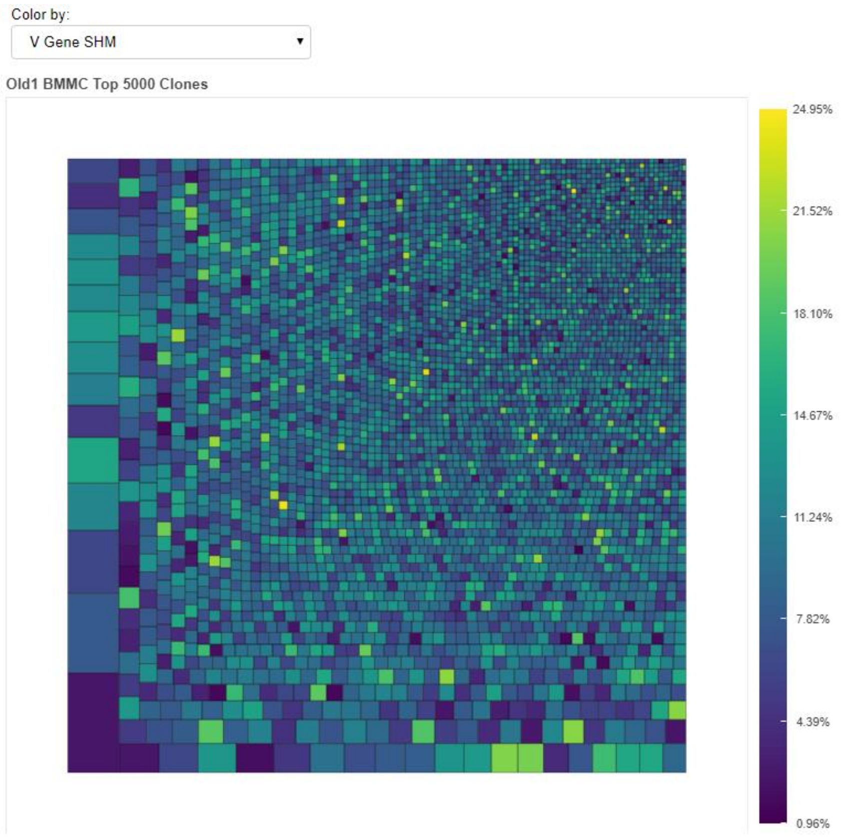


Figure 3.13: Clonal mosaics colored as a heatmap indicating clone V gene mutation levels. For continuous colormaps a legend sidebar is automatically calculated based on the range of the data.

Clonal V Gene Somatic Hypermutation Burtin Plot

In addition to looking at the total V gene hypermutation rates of a population, a novel approach to mutation rates is to consider all clones with specific V genes between two repertoires. The best method for picturing this is the Burtin plot, a form of radial bar chart. Unique V genes are displayed around the ring, with the length of the bar displaying the total SHM levels for all repertoires. As shown in figure 3.14, V gene SHM rates can vary wildly even within a single individual – a fact that is missed by aggregating total V gene SHM.

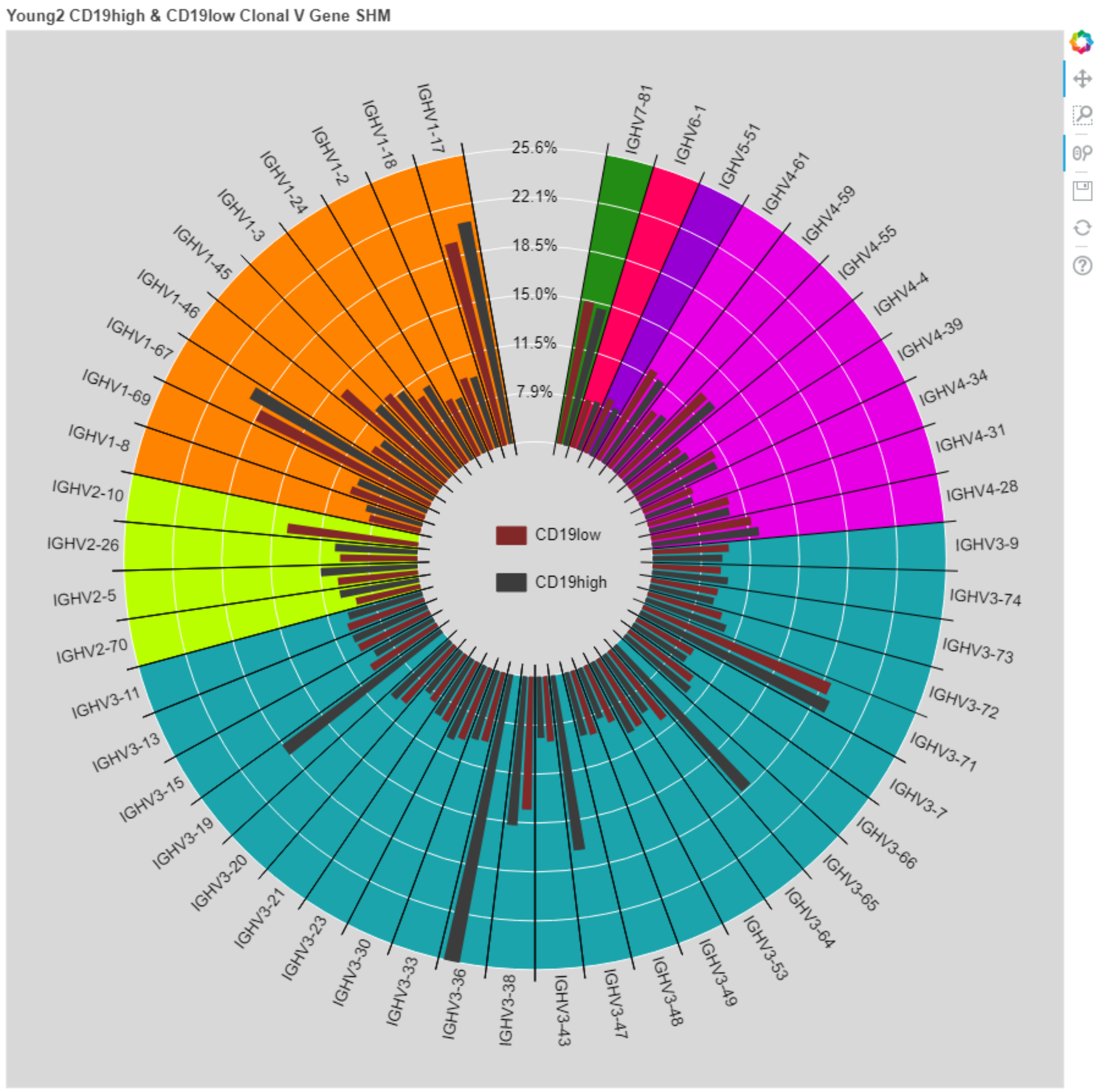


Figure 3.14: A Burtin V gene SHM plot. Bars are colored by sample repertoire, with V genes found in all repertoires represented radially. Total SHM rates for each gene are shown as the radius / length of the bars.

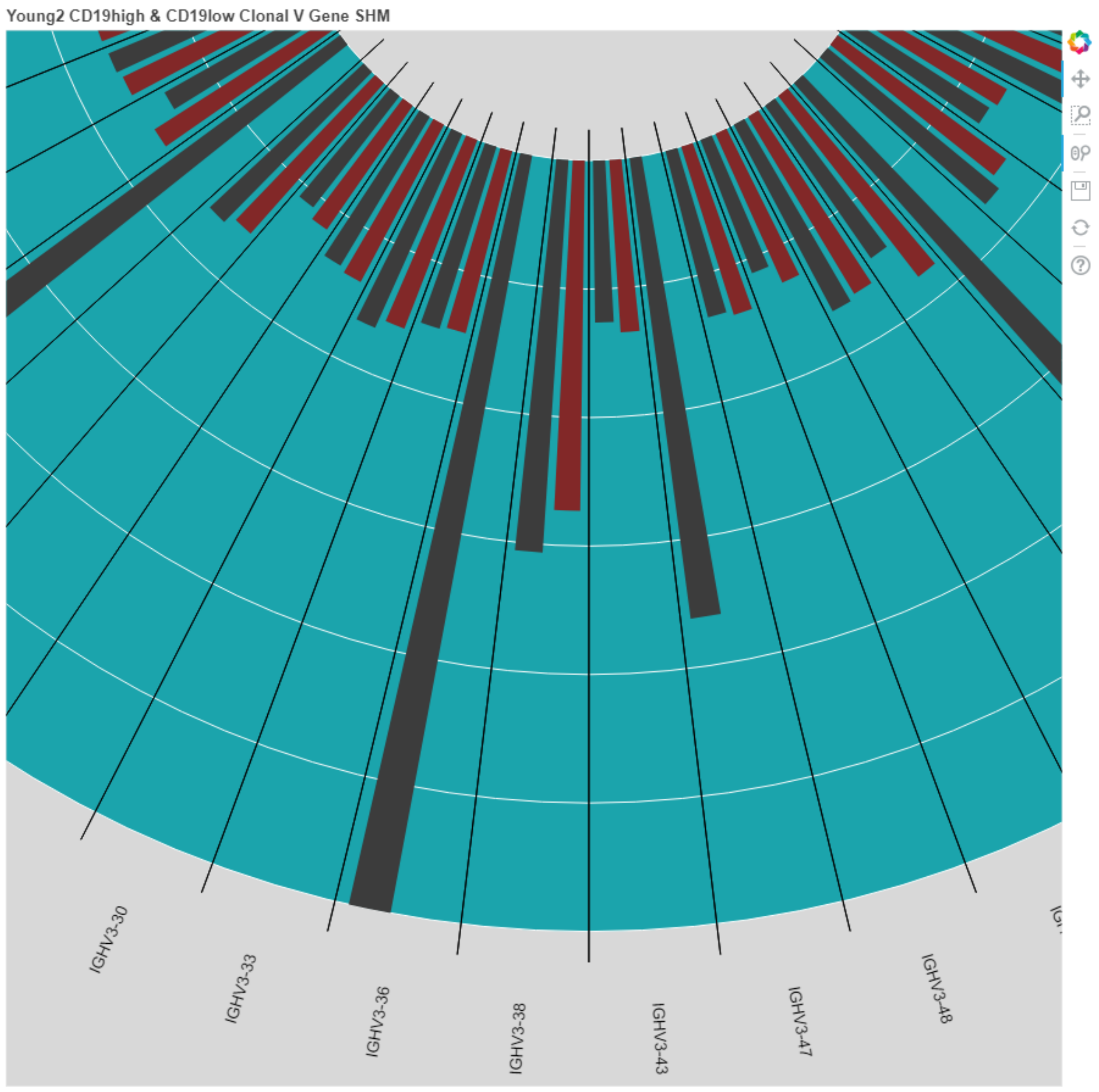


Figure 3.15: Zoomed in section from figure 3.14.

Repertoire Diversity / Polarization Line Plot

Qualitative measures of sample diversity and polarization are key for inter-repertoire comparison. One such measure of diversity is the series of Hill numbers, which are more adept than other measures since the series is far less affected by samples with extreme polarity or high total unique clone counts (Anne, C. et al 2014). As show in figure

3.16, multiple repertoires can be easily seen side-by-side for diversity comparison. For a set of *in silico* controls, several faux repertoires can be generated and displayed along with the actual datasets. These repertoires range from highly polarized to completely equally distributed across the population (figure 3.17).

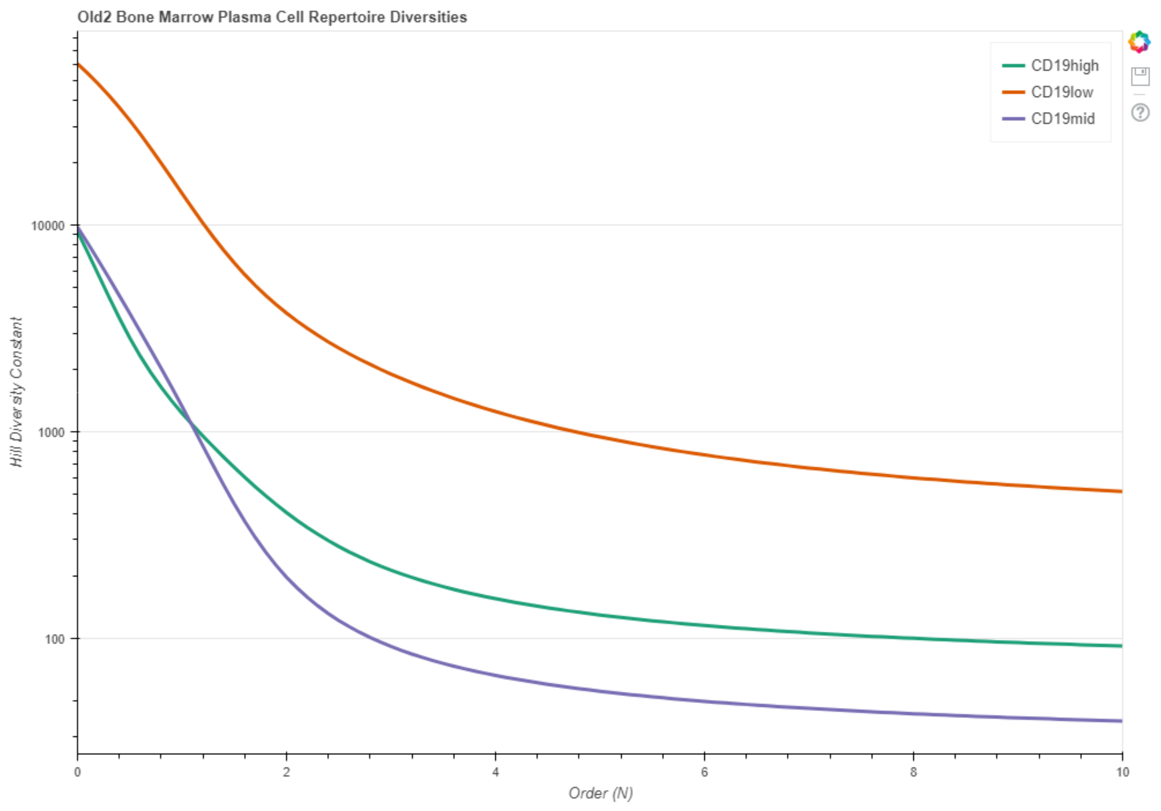


Figure 3.16: A Hill number diversity plot comparing three repertoires.

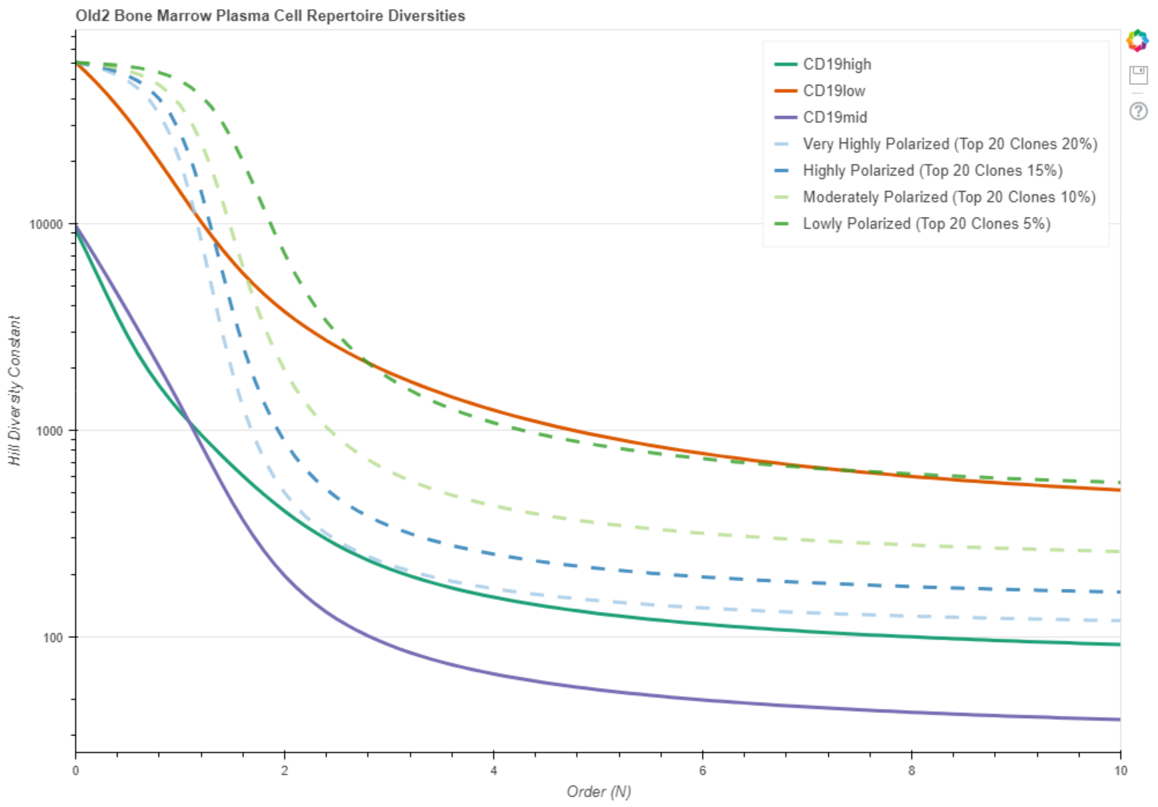


Figure 3.17: The Hill number diversity plot from 3.16 with added faux-repertoire controls for very highly, highly, moderately, and low polarized repertoires.

Appendix

Python Processing & Graphing Scripts

```
##########MIXCR Utilities

import logging
import subprocess
import os
import pandas

MIXCR_alignment_cols = ["descrR1", "cloneId", "vGene", "dGene", "jGene", "cGene",
                                                  "vBestIdentityPercent", "dBestIdentityPercent",
"jBestIdentityPercent"]

def MIXCR_Align(fastx, vdjca_out = None, species = "HomoSapiens", chains = "IG", threads = 8,
other_align_params = None,
                                 java_memory = None, save_reads = False, save_descs = True,
MIXCR_jar = MIXCR_loc, overwrite = False):
        """Perform a MIXCR alignment of the specified FASTQ/FASTA sequence file; wrapper function
for mixcr.jar align.

        Parameters
        ----------
        fastx: str
                Filename for the FASTQ/FASTA file to align with MIXCR.
        vdjca_out: str or None
                Output filename for the MIXCR .vdjca alignment file; by default will use the
prefix filename from fastx.
        species: str
                Species name (as usable by MIXCR) to use for V/D/J gene alignment; default is
"HomoSapiens".
        chains: str or list of str
                Immunological chain genes to compare against as used by MIXCR; default is "IG"
meaning any immunoglobulin.
        threads: int
                Number of CPU threads available for the MIXCR executable; default is 8.
        other_align_params: str or None
                Additional parameters given to MIXCR align as a string; optional.
        java_memory: int or str
                Amount of memory usable by the Java virtual machine in GB of RAM (eg. 8 leads to
"java -Xms8G -Xmx8G").
        save_reads: bool
                Whether or not to save the full read sequences in the output .vdjca file (MIXCR
align -g option); default False.
        save_descs: bool
                Whether or not to save the read descriptions in the output .vdjca file (MIXCR
align -a option); default True.
        MIXCR_jar: str
                Path of the MIXCR executable jar file; see MIXCR_loc.
        overwrite: bool
                If True, any existing .vdjca file of the same name will be overwritten (MIXCR -f
option); default is False.
        """

        logger = logging.getLogger("MIXCR_Align")

        if isinstance(vdjca_out, str):
                if not vdjca_out.lower().endswith(".vdjca"):
                        vdjca_out += ".vdjca"
        else:
                vdjca_out = ".".join(fastx.split(".")[:-1]) + ".vdjca"
        species = species if isinstance(species, str) else "HomoSapiens"

        if hasattr(chains, "__iter__") and not isinstance(chains, str):
                chains = ",".join(chains)
        elif not isinstance(chains, str):
                chains = "IG"
                logger.warning("chains needs to be either a list of chain types or a str;
defaulting to \"IG\"")

        threads = threads if int(threads) > 0 else 8

        MIXCR_call = ["java", "-jar"]

        if java_memory is not None:
                java_memory = str(java_memory).upper()

                if not java_memory.endswith("G"):
                        java_memory += "G"

                java_initial_mem = "-Xms" + java_memory
                java_max_mem = "-Xmx" + java_memory

                MIXCR_call.extend([java_initial_mem, java_max_mem])

        MIXCR_call.append(MIXCR_jar)
        MIXCR_call.append("align")
        MIXCR_call.extend(["-t", str(threads)])
        MIXCR_call.extend(["-s", species])
        MIXCR_call.extend(["-c", chains])

        if save_reads:
                MIXCR_call.append("-g")
        if save_descs:
                MIXCR_call.append("-a")
        if overwrite:
                MIXCR_call.append("-f")

        if other_align_params is not None:
                MIXCR_call.extend(other_align_params.split(" "))

        MIXCR_call.append(fastx)
        MIXCR_call.append(vdjca_out)

        logger.info("Starting alignment for " + fastx + "...")
        logger.debug("Full call to MIXCR:\n" + " ".join(MIXCR_call))

        try:
                align_log = subprocess.check_output(MIXCR_call, stderr = subprocess.STDOUT,
universal_newlines = True)

        except subprocess.CalledProcessError as cpe:
                MIXCR_error = cpe.stdout.strip().lower()

                if "filenotfoundexception" in MIXCR_error:
                        logger.error("Could not access input file " + fastx + "!")

                elif "already exists" in MIXCR_error:
                        logger.error("Output filename " + vdjca_out + " already exists!")
                        logger.error("Run MIXCR_Align again with overwrite = True or use a unique
vdjca_out filename.")

                elif "unable to access jarfile" in MIXCR_error:
                        logger.error("The MIXCR executable could not be found; check if the
provided jarfile path is correct!")
```

```
		else:
			logger.error("An unknown error occurred trying to run MIXCR align!")

		logger.error(MIXCR_error)
		return None

	logger.info(align_log)

def MIXCR_Filter_Alignments(vdjca, vdjca_out, seq_feature = None, cdr3_nt_seq = None,
other_filter_params = None,
								java_memory = None, MIXCR_jar =
MIXCR_loc, overwrite = False):
	"""Filter a MIXCR alignment to remove sequences not containing a specific feature, or with
a specific CDR3 sequence.
	Wrapper function for mixcr.jar filterAlignments.

	Parameters
	----------
	vdjca: str
		Filename for the VDJCA alignment binary file to filter.
	vdjca_out: str
		Output filename for the filtered MIXCR .vdjca alignment file.
	seq_feature: str or None
		Sequence feature to filter on, from any of ["FR1", "CDR1", "FR2", "CDR2", "FR3",
"CDR3", "FR4"].
	cdr3_nt_seq: str or None
		Nucleotide CDR3 sequence that must be present in filtered sequences; optional.
	other_filter_params: str or None
		Additional parameters given to MIXCR filterAlignments as a string; optional.
	java_memory: int or str
		Amount of memory usable by the Java virtual machine in GB of RAM (eg. 8 leads to
"java -Xms8G -Xmx8G").
	MIXCR_jar: str
		Path of the MIXCR executable jar file; see MIXCR_loc.
	overwrite: bool
		If True, any existing .vdjca file of the same name will be overwritten (MIXCR -f
option); default is False.
	"""

	logger = logging.getLogger("MIXCR_Filter_Alignments")

	if isinstance(vdjca_out, str):
		if not vdjca_out.lower().endswith(".vdjca"):
			vdjca_out += ".vdjca"

	MIXCR_call = ["java", "-jar"]

	if java_memory is not None:
		java_memory = str(java_memory).upper()

		if not java_memory.endswith("G"):
			java_memory += "G"

		java_initial_mem = "-Xms" + java_memory
		java_max_mem = "-Xmx" + java_memory

		MIXCR_call.extend([java_initial_mem, java_max_mem])

	MIXCR_call.append(MIXCR_jar)
	MIXCR_call.append("filterAlignments")

	valid_seq_features = ["FR1", "CDR1", "FR2", "CDR2", "FR3", "CDR3", "FR4"]
	if isinstance(seq_feature, str) and any([seq_feature.upper() == feat for feat in
valid_seq_features]):
		seq_feature = seq_feature.upper()
```

```
                MIXCR_call.extend(["-g", seq_feature])

        else:
                logger.error("seq_feature to filter must be a valid CDR/FR region (eg.
\"CDR2\")!")
                return None

        if isinstance(cdr3_nt_seq, str):
                cdr3_nt_seq = cdr3_nt_seq.upper()

                if len(cdr3_nt_seq.replace("A", "").replace("C", "").replace("G",
"").replace("T", "")) > 0:
                        logger.error("Sequences filtered by CDR3 must be nucleotide sequences
only!")
                        return None

                MIXCR_call.extend(["-e", cdr3_nt_seq])

        if overwrite:
                MIXCR_call.append("-f")

        if other_filter_params is not None:
                MIXCR_call.extend(other_filter_params.split(" "))

        MIXCR_call.append(vdjca)
        MIXCR_call.append(vdjca_out)

        logger.info("Filtering alignments from " + vdjca + "...")
        logger.debug("Full call to MIXCR:\n" + " ".join(MIXCR_call))

        try:
                filter_log = subprocess.check_output(MIXCR_call, stderr = subprocess.STDOUT,
universal_newlines = True)

        except subprocess.CalledProcessError as cpe:
                MIXCR_error = cpe.stdout.strip().lower()

                if "filenotfoundexception" in MIXCR_error:
                        logger.error("Could not access input file " + vdjca + "!")

                elif "already exists" in MIXCR_error:
                        logger.error("Output filename " + vdjca_out + " already exists!")
                        logger.error("Run MIXCR_Filter_Alignments again with overwrite = True or
use a unique vdjca_out filename.")

                elif "unable to access jarfile" in MIXCR_error:
                        logger.error("The MIXCR executable could not be found; check if the
provided jarfile path is correct!")

                else:
                        logger.error("An unknown error occurred trying to run MIXCR
filterAlignments!")

                logger.error(MIXCR_error)
                return None

        logger.info(filter_log)

def MIXCR_Assemble(vdjca, clns_out = None, threads = 8, create_index = True, other_assemble_params
= None,
                                    java_memory = None, MIXCR_jar = MIXCR_loc, overwrite = False):
        """Assemble clonotypes from a MIXCR alignment; wrapper function for mixcr.jar assemble.

        Parameters
        ----------
```

```
	vdjca: str
		Filename for the VDJCA alignment binary file to clonotype.
	clns_out: str or None
		Output filename for the MIXCR .clns alignment file; by default will use the
prefix filename from vdjca.
	threads: int
		Number of CPU threads available for the MIXCR executable; default is 8.
	create_index: bool
		Whether to create an index file usable for the export of total alignment reads
with clone IDs; default is True.
	other_assemble_params: str or None
		Additional parameters given to MIXCR assemble as a string; optional.
	java_memory: int or str
		Amount of memory usable by the Java virtual machine in GB of RAM (eg. 8 leads to
"java -Xms8G -Xmx8G").
	MIXCR_jar: str
		Path of the MIXCR executable jar file; see MIXCR_loc.
	overwrite: bool
		If True, any existing .clns file of the same name will be overwritten (MIXCR -f
option); default is False.
	"""

	logger = logging.getLogger("MIXCR_Assemble")

	if isinstance(clns_out, str):
		if not clns_out.lower().endswith(".clns"):
			clns_out += ".clns"
	else:
		clns_out = ".".join(vdjca.split(".")[:-1]) + ".clns"

	threads = threads if int(threads) > 0 else 8

	MIXCR_call = ["java", "-jar"]

	if java_memory is not None:
		java_memory = str(java_memory).upper()

		if not java_memory.endswith("G"):
			java_memory += "G"

		java_initial_mem = "-Xms" + java_memory
		java_max_mem = "-Xmx" + java_memory

		MIXCR_call.extend([java_initial_mem, java_max_mem])

	MIXCR_call.append(MIXCR_jar)
	MIXCR_call.append("assemble")
	MIXCR_call.extend(["-t", str(threads)])

	if create_index:
		index_file = ".".join(vdjca.split(".")[:-1]) + "_index"
		MIXCR_call.extend(["-i", index_file])

	if overwrite:
		MIXCR_call.append("-f")

	if other_assemble_params is not None:
		MIXCR_call.extend(other_assemble_params.split(" "))

	MIXCR_call.append(vdjca)
	MIXCR_call.append(clns_out)

	logger.info("Assembling clonotypes from " + vdjca + "...")
	logger.debug("Full call to MIXCR:\n" + " ".join(MIXCR_call))
```

```
        try:
                assemble_log = subprocess.check_output(MIXCR_call, stderr = subprocess.STDOUT,
universal_newlines = True)

        except subprocess.CalledProcessError as cpe:
                MIXCR_error = cpe.stdout.strip().lower()

                if "filenotfoundexception" in MIXCR_error:
                        logger.error("Could not access input file " + vdjca + "!")

                elif "already exists" in MIXCR_error:
                        logger.error("Output filename " + clns_out + " already exists!")
                        logger.error("Run MIXCR_Assemble again with overwrite = True or use a
unique clns_out filename.")

                elif "unable to access jarfile" in MIXCR_error:
                        logger.error("The MIXCR executable could not be found; check if the
provided jarfile path is correct!")

                else:
                        logger.error("An unknown error occurred trying to run MIXCR assemble!")

                logger.error(MIXCR_error)
                return None

        logger.info(assemble_log)

def Export_MIXCR_Alignments(vdjca, aligns_out = None, export_clone_ids = True, export_descriptions
= True,
                                                        export_top_genes = "VDJC",
export_top_gene_identities = "VDJ", export_read_ids = False,
                                                        nt_feat_seqs = "all", aa_feat_seqs =
"all", export_full_seq = False, export_aligns = None,
                                                        clone_index_file = None,
other_export_fields = None, java_memory = None,
                                                        MIXCR_jar = MIXCR_loc, overwrite =
False):
        """Export specified MIXCR alignment fields to a tab-separated text file for further
downstream analysis.
        Wrapper function for mixcr.jar exportAlignments; most arguments are used to pick the
fields exported to text.

        Parameters
        ----------
        vdjca: str
                Filename for the VDJCA alignment binary file to export to text.
        aligns_out: str or None
                Output filename for the alignment text file; by default will use the prefix of
vdjca plus "_alignments.txt".
        export_clone_ids: bool
                Whether or not to export clone IDs (clone index file must be available, see
clone_index_file); default True.
        export_descriptions: bool
                Whether to export the original FASTX read headers (if descriptions were saved
during align); default True.
        export_top_genes: str
                A string containing any/all of "VDJC" to export the top V/D/J/C gene called per
read; default is "VDJC".
        export_top_gene_identities: str or None
                A string containing any/all of "VDJC" to export the gene region identity (%
similarity to gene); default "VDJ".
        export_read_ids: bool
                Whether the original read ID numbers should be exported; default is False.
        nt_feat_seqs: str, list of str, or None
```

```
		Which CDR and framework region (FR) nucleotide sequences should be exported, or
"all" (default is "all").
		Example for only exporting FR2, FR3, and CDR3: ["FR2", "FR3", "CDR3"]
	aa_feat_seqs: str, list of str, or None
		Which CDR and framework region (FR) amino acid sequences should be exported, or
"all" (default is "all").
		Example for only exporting FR2, FR3, and CDR3: ["FR2", "FR3", "CDR3"]
	export_full_seq: bool
		Whether to export the full read nucleotide sequence for each read; default is
False.
	export_aligns: str or None
		A string containing any/all of "VDJC" to export the V/D/J/C alignment to germline
per read; default is None.
	clone_index_file: str or None
		Index of clone IDs for each alignment; if None the file is assumed to be the
prefix of vdjca plus "_index".
	other_export_fields: str, list of str, or None
		Additional export fields usable by MIXCR; can be a string of options or a list of
option strings.
		Example: "-mutationsDetailed FR3 -lengthOf CDR3"
	java_memory: int or str
		Amount of memory usable by the Java virtual machine in GB of RAM (eg. 8 leads to
"java -Xms8G -Xmx8G").
	MIXCR_jar: str
		Path of the MIXCR executable jar file; see MIXCR_loc.
	overwrite: bool
		If True, any existing output file of the same name will be overwritten (MIXCR -f
option); default is False.
	"""

	logger = logging.getLogger("Export_MIXCR_Alignments")

	if aligns_out is None:
		aligns_out = ".".join(vdjca.split(".")[:-1]) + "_alignments.txt"

	if clone_index_file is None:
		clone_index_file = ".".join(vdjca.split(".")[:-1]) + "_index"

	MIXCR_call = ["java", "-jar"]

	if java_memory is not None:
		java_memory = str(java_memory).upper()

		if not java_memory.endswith("G"):
			java_memory += "G"

		java_initial_mem = "-Xms" + java_memory
		java_max_mem = "-Xmx" + java_memory

		MIXCR_call.extend([java_initial_mem, java_max_mem])

	MIXCR_call.append(MIXCR_jar)
	MIXCR_call.append("exportAlignments")

	if overwrite:
		MIXCR_call.append("-f")

	if export_clone_ids:
		MIXCR_call.extend(["-cloneId", clone_index_file])

	if export_descriptions:
		MIXCR_call.append("-descrR1")

	if isinstance(export_top_genes, str):
		export_top_genes = export_top_genes.upper()
		if "ALL" in export_top_genes:
			export_top_genes = "VDJC"

		if "V" in export_top_genes:
			MIXCR_call.append("-vGene")
		if "D" in export_top_genes:
			MIXCR_call.append("-dGene")
		if "J" in export_top_genes:
			MIXCR_call.append("-jGene")
		if "C" in export_top_genes:
			MIXCR_call.append("-cGene")

	if isinstance(export_top_gene_identities, str):
		export_top_gene_identities = export_top_gene_identities.upper()
		if "ALL" in export_top_gene_identities:
			export_top_gene_identities = "VDJC"

		if "V" in export_top_gene_identities:
			MIXCR_call.append("-vBestIdentityPercent")
		if "D" in export_top_gene_identities:
			MIXCR_call.append("-dBestIdentityPercent")
		if "J" in export_top_gene_identities:
			MIXCR_call.append("-jBestIdentityPercent")
		if "C" in export_top_gene_identities:
			MIXCR_call.append("-cBestIdentityPercent")

	if export_read_ids:
		MIXCR_call.append("-readId")

	if hasattr(nt_feat_seqs, "__iter__") and not isinstance(nt_feat_seqs, str):
		for feat in nt_feat_seqs:
			MIXCR_call.extend(["-nFeature", feat.upper()])
	elif isinstance(nt_feat_seqs, str):
		nt_feat_seqs = nt_feat_seqs.upper()

		if "ALL" in nt_feat_seqs:
			MIXCR_call.extend(["-nFeature", "FR1", "-nFeature", "FR2", "-nFeature",
"FR3", "-nFeature", "FR4"])
			MIXCR_call.extend(["-nFeature", "CDR1", "-nFeature", "CDR2", "-nFeature",
"CDR3"])
		else:
			MIXCR_call.extend(["-nFeature", nt_feat_seqs])

	if hasattr(aa_feat_seqs, "__iter__") and not isinstance(aa_feat_seqs, str):
		for feat in aa_feat_seqs:
			MIXCR_call.extend(["-aaFeature", feat.upper()])
	elif isinstance(aa_feat_seqs, str):
		aa_feat_seqs = aa_feat_seqs.upper()

		if "ALL" in aa_feat_seqs:
			MIXCR_call.extend(["-aaFeature", "FR1", "-aaFeature", "FR2", "-
aaFeature", "FR3", "-aaFeature", "FR4"])
			MIXCR_call.extend(["-aaFeature", "CDR1", "-aaFeature", "CDR2", "-
aaFeature", "CDR3"])
		else:
			MIXCR_call.extend(["-aaFeature", aa_feat_seqs])

	if export_full_seq:
		MIXCR_call.append("-sequence")

	if isinstance(export_aligns, str):
		export_aligns = export_aligns.upper()
		if "ALL" in export_aligns:
			export_aligns = "VDJC"
```

```
                if "V" in export_aligns:
                        MIXCR_call.append("-vAlignment")
                if "D" in export_aligns:
                        MIXCR_call.append("-dAlignment")
                if "J" in export_aligns:
                        MIXCR_call.append("-jAlignment")
                if "C" in export_aligns:
                        MIXCR_call.append("-cAlignment")

        if other_export_fields is not None:
                if isinstance(other_export_fields, str):
                        other_export_fields = other_export_fields.split(" ")

                MIXCR_call.extend(other_export_fields)

        MIXCR_call.append(vdjca)
        MIXCR_call.append(aligns_out)

        logger.info("Exporting alignments from " + vdjca + "...")
        logger.debug("Full call to MIXCR:\n" + " ".join(MIXCR_call))

        try:
                subprocess.check_call(MIXCR_call, stderr = subprocess.STDOUT)

        except subprocess.CalledProcessError as cpe:
                MIXCR_error = cpe.stdout.strip().lower()

                if "filenotfoundexception" in MIXCR_error:
                        logger.error("Could not access input file " + vdjca + "!")

                elif "already exists" in MIXCR_error:
                        logger.error("Output filename " + aligns_out + " already exists!")
                        logger.error("Run Export_MIXCR_Alignments again with overwrite = True or
use a unique aligns_out filename.")

                elif "unable to access jarfile" in MIXCR_error:
                        logger.error("The MIXCR executable could not be found; check if the
provided jarfile path is correct!")

                else:
                        logger.error("An unknown error occurred trying to run MIXCR
exportAlignments.")

                logger.error(MIXCR_error)
                return None

        logger.info("Alignments written to {0}.".format(aligns_out))

def Run_MIXCR(fastx, required_features = None, align_params = None, filter_params = None,
assemble_params = None,
                          export_params = None, delete_temps = False, java_memory = None,
MIXCR_jar = MIXCR_loc, log_file = None):
        """

        Parameters
        ----------
        Returns
        ----------
        SUCCESS
        """

        file_prefix = ".".join(fastx.split(".")[:-1])

        if log_file is None:
                log_file = file_prefix + "_MIXCR_log.txt"
        logging.basicConfig(filename = log_file, level = logging.DEBUG,
                                                datefmt = "%m-%d-%y %H:%M:%S", format =
"%(asctime)s in %(name)s:\n%(message)s\n")
        console = logging.StreamHandler()
        console.setLevel(logging.INFO)
        console.setFormatter(logging.Formatter("%(levelname)s\t%(message)s"))
        logging.getLogger("").addHandler(console)

        if align_params is not None:
                MIXCR_Align(fastx, **align_params)
        else:
                MIXCR_Align(fastx)

        if isinstance(required_features, str):
                required_features = required_features.upper()

                if "ALL" in required_features:
                        required_features = ["FR1", "CDR1", "FR2", "CDR2", "FR3", "CDR3", "FR4"]
                else:
                        required_features = [required_features]

        if required_features is not None:
                cur_in_filename = file_prefix
                cur_out_filename = file_prefix + "_has"

                for feature in required_features:
                        cur_out_filename += "_" + feature
                        MIXCR_Filter_Alignments(cur_in_filename + ".vdjca", cur_out_filename +
".vdjca", feature)
                        cur_in_filename = cur_out_filename

                vdjca = cur_in_filename + ".vdjca"

        else:
                vdjca = file_prefix + ".vdjca"

        MIXCR_Assemble(vdjca)

        if export_params is not None:
                Export_MIXCR_Alignments(vdjca, **export_params)
        else:
                Export_MIXCR_Alignments(vdjca)

def FASTX_Random_Sample(filename, outfile_prefix = None, num_seqs = None, num_resamples = None):
        pass

from .Constants import MIXCR_header_dtypes, MIXCR_headers_renamed, no_stop_feats,
no_frameshift_feats

def MIXCR_to_DataFrame(filename, col_dtypes = MIXCR_header_dtypes, renamed_cols =
MIXCR_headers_renamed,
                                           drop_features_with_stop = no_stop_feats,
drop_features_with_frameshift = no_frameshift_feats,
                                           clone_col = "CloneID", min_clone_count = 2,
simplify_isotypes = True, stop_char = "*",
                                           frameshift_char = "_", assume_heavy_or_light = True,
assume_heavy_light_cutoff = 0.9):
        """
        By default, removes sequences with a stop codon in any gene region or a frameshift in any
region but FR4.

        Parameters
        ----------
        Returns
        ----------
        seq_df: pandas.DataFrame
                A pandas DataFrame consisting of the filtered alignments from filename.
        """

        if not os.path.exists(filename):
                print("Error in MIXCR_to_DataFrame: can't find input alignment file
{0}!".format(filename))
                return None

        seq_df = pandas.read_csv(filename, sep = "\t", usecols = [i for i in col_dtypes], dtype =
col_dtypes)
        total_raw_reads = len(seq_df)

        print("{0} successfully loaded; identified a total of {1} raw reads.".format(filename,
total_raw_reads))

        if renamed_cols:
                seq_df = seq_df.rename(columns = renamed_cols)

        filtered_report = "Dropped {0} sequences ({1:.1%} of raw reads) with a {2} in feature
{3}."
        if drop_features_with_stop:
                for feature in drop_features_with_stop:
                        prefiltered_reads = len(seq_df)

                        if feature in seq_df.columns:
                                seq_df = seq_df[~seq_df[feature].str.contains(stop_char, na =
False, regex = False)]
                                dropped_reads = prefiltered_reads - len(seq_df)
                                print(filtered_report.format(dropped_reads, (dropped_reads /
total_raw_reads), "stop codon", feature))

                        else:
                                print("Error in MIXCR_to_DataFrame: column \"{0}\" not found in
alignment file!".format(feature))
                                return None

        if drop_features_with_frameshift:
                for feature in drop_features_with_frameshift:
                        prefiltered_reads = len(seq_df)

                        if feature in seq_df.columns:
                                seq_df = seq_df[~seq_df[feature].str.contains(frameshift_char, na
= False, regex = False)]
                                dropped_reads = prefiltered_reads - len(seq_df)
                                print(filtered_report.format(dropped_reads, (dropped_reads /
total_raw_reads), "frameshift", feature))

                        else:
                                print("Error in MIXCR_to_DataFrame: column \"{0}\" not found in
alignment file!".format(feature))
                                return None

        cur_reads = len(seq_df)
        if cur_reads < total_raw_reads:
                pct_total = cur_reads / total_raw_reads
                print("{0} reads remaining after filtering features ({1:.1%} of
total).".format(cur_reads, pct_total))

        if clone_col is not None and clone_col in seq_df.columns:
                if min_clone_count > 0:
                        prefiltered_reads = len(seq_df)
                        seq_df = seq_df[seq_df[clone_col].notnull()]
                        seq_df[clone_col] = seq_df[clone_col].astype(int)
                        no_clone = prefiltered_reads - len(seq_df)
                        pct_total = no_clone / total_raw_reads
                        print("{0} reads removed with no assigned clonotype ({1:.1%} of
total).".format(no_clone, pct_total))

                if min_clone_count > 1:
                        prefiltered_reads = len(seq_df)
                        clone_counts = seq_df[clone_col].value_counts()
                        clone_counts_filtered = clone_counts[clone_counts >= min_clone_count]
                        seq_df = seq_df[seq_df[clone_col].isin(clone_counts_filtered.index)]

                        not_enough_clone_members = prefiltered_reads - len(seq_df)
                        pct_total = not_enough_clone_members / total_raw_reads
                        report = "{0} reads removed for being in clonotypes consisting of less
than {1} reads ({2:.1%} of total)."
                        print(report.format(not_enough_clone_members, min_clone_count,
pct_total))

        elif clone_col is not None and clone_col not in seq_df.columns:
                print("Error in MIXCR_to_DataFrame: clone ID column \"{0}\" not found in
alignment file!".format(clone_col))
                return None

        for col in ["VGene", "DGene", "JGene"]:
                if col in seq_df.columns:
                        seq_df[col] = seq_df[col].fillna("Unknown")

        if "V_Identity" in seq_df.columns:
                seq_df["V_SHM"] = 1.0 - seq_df["V_Identity"]
                seq_df = seq_df.drop(["V_Identity"], axis = 1)

        if "D_Identity" in seq_df.columns:
                seq_df["D_SHM"] = 1.0 - seq_df["D_Identity"]
                seq_df = seq_df.drop(["D_Identity"], axis = 1)

        if "J_Identity" in seq_df.columns:
                seq_df["J_SHM"] = 1.0 - seq_df["J_Identity"]
                seq_df = seq_df.drop(["J_Identity"], axis = 1)

        if "CGene" in seq_df.columns:
                seq_df["Isotype"] = seq_df["CGene"].str.split("*").str[0] #Remove MIXCR allele
calls, primers are ambiguous
                seq_df["Isotype"] = seq_df["Isotype"].fillna("Other")
                seq_df = seq_df.drop(["CGene"], axis = 1)

                if assume_heavy_or_light:
                        #if heavy chain is >90% of reads, remove any light chain seqs. and vice
versa
                        heavy_chains = len(seq_df[seq_df["Isotype"].str.contains("IGH")])
                        light_chains = len(seq_df[seq_df["Isotype"].str.contains("IGL") |
seq_df["Isotype"].str.contains("IGK")])
                        heavy_chain_ratio = heavy_chains / len(seq_df)
                        light_chain_ratio = light_chains / len(seq_df)

                        if heavy_chain_ratio > assume_heavy_light_cutoff and light_chains > 0:
                                seq_df = seq_df[seq_df["Isotype"].str.contains("IGH")]
                                cutoff_report = "{0} light chain reads dropped (assumed to be
erroneous as {1:.1%} of reads are IGH)."
                                print(cutoff_report.format(light_chains, heavy_chain_ratio))

                        elif light_chain_ratio > assume_heavy_light_cutoff:
                                seq_df = seq_df[seq_df["Isotype"].str.contains("IGK") |
seq_df["Isotype"].str.contains("IGL")]
```

```
                                cutoff_report = "{0} heavy chain reads dropped (assumed to be 
erroneous as {1:.1%} of reads are IGK/L)."
                                print(cutoff_report.format(heavy_chains, light_chain_ratio))

                if simplify_isotypes:
                        simple_isotypes = {
                                "IGHA1": "IgA", "IGHA2": "IgA", "IGHD": "IgD", "IGHEP1": "IgE", 
"IGHE": "IgE",
                                "IGHG1": "IgG", "IGHG2": "IgG", "IGHG3": "IgG", "IGHG4": "IgG", 
"IGHGP": "IgG",
                                "IGHM": "IgM", "IGLC1": "IgL", "IGLC2": "IgL", "IGLC3": "IgL", 
"IGLC4": "IgL",
                                "IGLC5": "IgL", "IGLC6": "IgL", "IGLC7": "IgL", "IGKC": "IgK"
                        }

                        for isotype in simple_isotypes:
                                seq_df["Isotype"] = seq_df["Isotype"].str.replace(isotype, 
simple_isotypes[isotype])

        seq_df = seq_df.reset_index(drop = True)

        filtered_reads = len(seq_df)
        print("{0} reads remaining in final dataframe!".format(filtered_reads))

        return seq_df


###########Repertoire Comparison

import pandas

def Morisita_Horn_Similarity(combined_freq_df, freq_col1, freq_col2):
        """Calculates the Morisita-Horn index of similarity: 2 * sum(freq(x) * freq(y)) / 
(sum(freq(x)^2) + sum(freq(y)^2))
        This function is intended for use by Repertoire_Similarity().

        Parameters
        ----------
        combined_freq_df: pandas.DataFrame
                DataFrame containing the frequencies of each unique clone found in either of the 
two repertoires.
        freq_col1: str
                Column name in combined_freq_df with the clonal frequencies for all clones in the 
first repertoire.
        freq_col2: str
                Column name in combined_freq_df with the clonal frequencies for all clones in the 
second repertoire.

        Returns
        ----------
        MH_index: float
                The similarity of the repertoires as defined by the Morisita-Horn index.
        """

        freq_product = combined_freq_df[freq_col1] * combined_freq_df[freq_col2]
        freq1_squared = combined_freq_df[freq_col1] ** 2
        freq2_squared = combined_freq_df[freq_col2] ** 2

        summed_freq_product = freq_product.sum()
        summed_freq_squared = freq1_squared.sum() + freq2_squared.sum()

        MH_index = (summed_freq_product / summed_freq_squared) * 2.0
        return MH_index

def Cosine_Similarity(combined_freq_df, freq_col1, freq_col2):
        """Calculates the Cosine index of similarity: sum(freq(x) * freq(y)) /
(sqrt(sum(freq(x)^2)) * sqrt(sum(freq(y)^2)))
        This function is intended for use by Repertoire_Similarity().

        Parameters
        ----------
        combined_freq_df: pandas.DataFrame
                DataFrame containing the frequencies of each unique clone found in either of the
two repertoires.
        freq_col1: str
                Column name in combined_freq_df with the clonal frequencies for all clones in the
first repertoire.
        freq_col2: str
                Column name in combined_freq_df with the clonal frequencies for all clones in the
second repertoire.

        Returns
        ----------
        cosine_index: float
                The similarity of the repertoires as defined by the Cosine index.
        """

        from numpy import sqrt
        freq_product = combined_freq_df[freq_col1] * combined_freq_df[freq_col2]
        freq1_squared = combined_freq_df[freq_col1] ** 2
        freq2_squared = combined_freq_df[freq_col2] ** 2

        summed_freq_product = freq_product.sum()
        freq1_squared_sum_sqrt = sqrt(freq1_squared.sum())
        freq2_squared_sum_sqrt = sqrt(freq2_squared.sum())

        cosine_index = summed_freq_product / (freq1_squared_sum_sqrt * freq2_squared_sum_sqrt)
        return cosine_index

def Jaccard_Similarity(combined_freq_df, freq_col1, freq_col2):
        """Calculates the Jaccard index of similarity: sum(minimum of freqs(x, y)) / sum(maximum
of freqs(x, y))
        This function is intended for use by Repertoire_Similarity().

        Parameters
        ----------
        combined_freq_df: pandas.DataFrame
                DataFrame containing the frequencies of each unique clone found in either of the
two repertoires.
        freq_col1: str
                Column name in combined_freq_df with the clonal frequencies for all clones in the
first repertoire.
        freq_col2: str
                Column name in combined_freq_df with the clonal frequencies for all clones in the
second repertoire.

        Returns
        ----------
        jaccard_index: float
                The similarity of the repertoires as defined by the Jaccard index.
        """

        min_freqs = combined_freq_df[[freq_col1, freq_col2]].min(axis = 1)
        max_freqs = combined_freq_df[[freq_col1, freq_col2]].max(axis = 1)

        jaccard_index = min_freqs.sum() / max_freqs.sum()
        return jaccard_index

def Bray_Curtis_Similarity(combined_freq_df, freq_col1, freq_col2):
        """Calculates the Bray-Curtis index of similarity: sum(minimum of freqs(x, y))
```

```
	This function is intended for use by Repertoire_Similarity().

	Parameters
	----------
	combined_freq_df: pandas.DataFrame
		DataFrame containing the frequencies of each unique clone found in either of the
two repertoires.
	freq_col1: str
		Column name in combined_freq_df with the clonal frequencies for all clones in the
first repertoire.
	freq_col2: str
		Column name in combined_freq_df with the clonal frequencies for all clones in the
second repertoire.

	Returns
	----------
	BC_index: float
		The similarity of the repertoires as defined by the Bray-Curtis index.
	"""

	min_freqs = combined_freq_df[[freq_col1, freq_col2]].min(axis = 1)

	BC_index = min_freqs.sum()
	return BC_index

def Repertoire_Similarity(clone_df1, clone_df2, how = "all", clone_col = "CloneID", count_col =
"Clustered"):
	"""Calculates the similarity of two repertoires that MUST have been clonotyped together
(ie, must share clone IDs).
	Generally returns a fraction ranging from 0.0 (no similarity) to 1.0 (in theory,
identical) depending on the method.
	Implemented methods currently available (for the "how" argument):
		Morisita-Horn, called using "morisita_horn", "morisita", "horn", or "mh".
		Cosine, called using "cosine" or "cos".
		Jaccard, called using "jaccard" or "j".
		Bray-Curtis, called using "bray_curtis", "bray", "curtis", "bc".
		Using "all" yields a dict of all methods formatted as {str: float} for the method
name -> resulting value.

	Parameters
	----------
	clone_df1: pandas.DataFrame
		First DataFrame containing unique clones sharing clone IDs with clone_df2 (must
have been clonotyped together).
	clone_df2: pandas.DataFrame
		Second DataFrame containing unique clones sharing clone IDs with clone_df1 (must
have been clonotyped together).
	how: str
		Method for calculating similarity, defaulting to "all"; can be one of the
following:
			"mh", "morisita", "horn", or "morisita_horn" for Morisita-Horn
similarity.
			"cos" or "cosine" for Cosine similarity.
			"j" or "jaccard" for Jaccard similarity.
			"bc", "bray", "curtis", or "bray_curtis" for Bray-Curtis similarity.
			"all" returns a dict of all methods formatted as {str: float} for the
method name -> resulting value.
	clone_col: str
		Name of column in both DataFrames with the unique clone IDs; default is
"CloneID".
	count_col: str
		Name of column in both DataFrames with the count or frequency for each unique
clone; default is "Clustered".

	Returns
	----------
	similarity or similarities: float or {str: float}
		If how != "all" returns the value of the given similarity index, otherwise a dict 
of the results of all methods.
		An example of the result from how = "all": {"Morisita_Horn": 0.9556, "Cosine": 
0.9239, "Jaccard": 0.8913}
	"""

	valid_method_calls = ["all", "mh", "morisita", "horn", "cos", "j", "bc", "bray", "curtis"]

	if isinstance(how, str) and any([m in how.lower() for m in valid_method_calls]):
		how = how.lower()

		if "all" in how:
			method = "All"
		elif any(["mh" in how, "morisita" in how, "horn" in how]):
			method = "Morisita_Horn"
		elif "cos" in how:
			method = "Cosine"
		elif "j" in how:
			method = "Jaccard"
		elif any(["bc" in how, "bray" in how, "curtis" in how]):
			method = "Bray_Curtis"

	else:
		print("Warning in Repertoire_Similarity: \"how\" argument invalid; defaulting to 
\"all\"!")
		method = "All"

	similarity_funcs = {
		"Morisita_Horn": Morisita_Horn_Similarity,
		"Cosine": Cosine_Similarity,
		"Jaccard": Jaccard_Similarity,
		"Bray_Curtis": Bray_Curtis_Similarity
	}

	if clone_col not in clone_df1.columns or clone_col not in clone_df2.columns:
		print("Error in Repertoire_Similarity: clone ID column \"{0}\" not 
found!".format(clone_col))
		return None

	if count_col not in clone_df1.columns or count_col not in clone_df2.columns:
		print("Error in Repertoire_Similarity: clone count/frequency column \"{0}\" not 
found!".format(count_col))
		return None

	clone_df1_freqs = clone_df1[[clone_col, count_col]]
	clone_df2_freqs = clone_df2[[clone_col, count_col]]

	total_clone_counts_df1 = float(clone_df1_freqs[count_col].sum())
	total_clone_counts_df2 = float(clone_df2_freqs[count_col].sum())
	clone_df1_freqs["Freq_DF1"] = clone_df1_freqs[count_col] / total_clone_counts_df1
	clone_df2_freqs["Freq_DF2"] = clone_df2_freqs[count_col] / total_clone_counts_df2
	clone_df1_freqs = clone_df1_freqs.drop([count_col], axis = 1)
	clone_df2_freqs = clone_df2_freqs.drop([count_col], axis = 1)

	merged_clone_dfs = clone_df1_freqs.merge(clone_df2_freqs, on = clone_col, how = "outer")
	merged_clone_dfs = merged_clone_dfs.fillna(0.0)

	if method == "All":
		similarities = {}

		for func in similarity_funcs:
			Similarity_Function = similarity_funcs[func]
```

```
                        similarities[func] = Similarity_Function(merged_clone_dfs, freq_col1 =
"Freq_DF1", freq_col2 = "Freq_DF2")

                return similarities

        else:
                Similarity_Function = similarity_funcs[method]
                similarity = Similarity_Function(merged_clone_dfs, freq_col1 = "Freq_DF1",
freq_col2 = "Freq_DF2")
                return similarity


##########Graphing
import matplotlib.pyplot as plt
import numpy
from squarify import squarify
from itertools import cycle
from matplotlib.colors import Colormap
from matplotlib.patches import Rectangle, Wedge
from matplotlib.collections import PatchCollection
from matplotlib.ticker import StrMethodFormatter
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

#Add ability to plot several groups
def Rank_Abundance_Graph(clone_counts, max_clones = None, zoom_inset = None, figsize = None):
        if not hasattr(clone_counts, "__iter__") or isinstance(clone_counts, str):
                print("Error in Rank_Abundance_Graph: clone_counts must be a list/iterable of
clone counts/frequencies!")
                return None

        if figsize is None:
                fig = plt.figure()
        else:
                fig = plt.figure(figsize = figsize)

        ax = fig.add_subplot(1, 1, 1)

        total_clone_counts = float(sum(clone_counts))
        total_clones = len(clone_counts)

        clone_freqs = []
        for clone in sorted(clone_counts, reverse = True):
                clone_freq = clone / total_clone_counts
                clone_freqs.append(clone_freq)

        ranks = [i for i in range(1, total_clones + 1)]
        ax.bar(x = ranks, height = clone_freqs)

        if max_clones is not None:
                ax.set_xlim(0, max_clones)

        #zoomed_ax = zoomed_inset_axes(ax, 0.5, loc = 1)
        zoomed_ax = inset_axes(ax, 2, 1)
        zoomed_ax.bar(x = ranks, height = clone_freqs)
        zoomed_ax.set_xlim(0, 20)
        zoomed_ax.set_ylim(0, max(clone_freqs))
        mark_inset(ax, zoomed_ax, loc1 = 2, loc2 = 4, ec = "black")

        return fig

def Mosaic_Plot(members, member_color_data = None, fig_name = None, title = None, colors = "Set2",
num_colors = 3,
                                quant_cmap = "viridis", colorbar_pos = "right", colorbar_width =
0.02, highlight_upper_percent = 0.5,
                                fig = None, ax = None, figsize = (2, 2), dpi = 600, bbox_inches =
"tight", multiple_fig = False):
        """Creates a Mosaic plot figure optionally colored by an additional quantitative factor
using Matplotlib patches.

        Parameters
        ----------
        members: list or other non-string iterable
                An iterable of float or integer values to be used as areas for the mosaic
patches.
        member_color_data: list or other non-string iterable of same length as members, or None
                An optional list of the same length as members with a quantitative range to color
members by (such as gene SHM).
        fig_name: str or None
                Filename to save the output figure; if None, shows the figure on screen
(matplotlib.pyplot.show); default None.
        title: str or None
                Text to be used as the figure title; default is None.
        colors: str, or matplotlib Colormap, or list or other non-string iterable of same length
as members, or None
                Colormap to use for the mosaic patches IF member_color_data is NOT provided, and
ignored otherwise.
                colors can be a str of any matplotlib colormap name, or a Colormap object;
default is "Set2".
                If another iterable is provided, must be a list of color names/values to manually
provide each patch's color.
        num_colors: int
                If member_color_data is None, num_colors sets the mosaic patches to cycle through
the first "num_colors" colors.
                Too many colors to cycle through is visually straining and may be problematic for
a group of many small patches.
                num_colors is ignored if member_color_data is provided, or if colors is a
list/iterable of each patch's colors.
                Default: 3
        quant_cmap: str or matplotlib Colormap
                Describes the colormap to use for mapping the values of member_color_data to
quantitative color values.
                Can be a str name for a matplotlib colormap or a Colormap object; default is
"viridis".
        colorbar_pos: str
                Location on figure (cardinal direction) to place the colorbar if
member_color_data given; default is "right".
        colorbar_width: float
                Width for the colorbar if member_color_data is provided (passed to matplotlib
Rectangle patch); default is 0.02.
        highlight_upper_percent: float or None
                If not None (default), the upper x% of patches will use a lighter color scheme
than the patches smaller than x%.
                Example: if set as 0.6 the top 60% of patches will be normally colored and the
remaining patches will be darker.
        fig: matplotlib Figure or None
                Matplotlib Figure object to use instead of creating a new Figure; default is
None.
        ax: matplotlib Axes or None
                Matplotlib Axes object to use instead of adding a new Axes; default is None.
        figsize: tuple of (int, int) or (float, float)
                Size for the matplotlib Figure object as a tuple of (inches, inches); default is
(2, 2).
        dpi: int
                Dots per inch (DPI) for the matplotlib Figure; default is 600.
        bbox_inches: str, int
                Image bounding box parameter given to matplotlib; default is "tight".
        multiple_fig: bool
                ADD
        """
        if hasattr(members, "__iter__") and not isinstance(members, str):
                total_area = float(sum(members))
                norm_areas = [float(member) / total_area for member in members]
        else:
                print("Error in Mosaic_Plot: members must be a list/iterable (eg. pandas Series,
numpy array) of ints/floats!")
                return None

        sorted_areas = sorted(norm_areas, reverse = True)        #Keeping original list to use
for sorting other lists
        mosaic_rects = squarify(sorted_areas, 0.0, 0.0, 1.0, 1.0)

        if fig is None:
                fig = plt.figure(figsize = figsize, dpi = dpi)

        if ax is None:
                ax = fig.add_subplot(1, 1, 1, aspect = "equal")

        ax.set_axis_off()

        if isinstance(title, str):
                ax.set_title(title)

        if isinstance(colors, str):
                colors = plt.get_cmap(colors).colors
        elif isinstance(colors, Colormap):
                colors = colors.colors #lol
        elif hasattr(colors, "__iter__") and not isinstance(colors, str) and len(colors) ==
len(members):
                num_colors = len(colors)
                #Since member order will have probably changed after sorting, must sort the color
list based on original member.
                colors = [clr for member, clr in sorted(zip(norm_areas, colors), key = lambda x:
x[0], reverse = True)]
        else:
                colors = plt.get_cmap("Set2").colors

        num_colors = num_colors if isinstance(num_colors, int) and num_colors > 1 else 3
        mosaic_colors = cycle(colors[0:num_colors])

        if isinstance(highlight_upper_percent, float) and 0.0 < highlight_upper_percent < 1.0:
                change_colors = True
                lower_colors = []

                for c in colors[0:num_colors]:
                        darker_color = [0.0 if rgb <= 0.25 else rgb - 0.25 for rgb in c]
                        lower_colors.append(darker_color)

        else:
                change_colors = False

        mosaic_patches = []

        area_summed = 0.0
        for rect in mosaic_rects:
                rect_xy = (rect["x"], rect["y"])
                rect_width = rect["dx"]
                rect_height = rect["dy"]

                if change_colors:
                        cur_area = rect_width * rect_height
                        area_summed += cur_area

                        if area_summed > highlight_upper_percent:
                                mosaic_colors = cycle(lower_colors)
                                change_colors = False

                mosaic_patches.append(Rectangle(rect_xy, rect_width, rect_height, facecolor =
next(mosaic_colors)))

        mosaic_collection = PatchCollection(mosaic_patches, edgecolor = "#404040", linewidth =
0.2, match_original = True)

        if hasattr(member_color_data, "__iter__") and not isinstance(member_color_data, str):
                if len(member_color_data) != len(members):
                        print("Error creating Mosaic: member_color_data must be the same length
as the members input!")
                        print("Make sure the correct matching data is used, or remove
member_color_data to repeat default colors.")
                        return None

                member_colors = [j for i, j in sorted(zip(norm_areas, member_color_data), key =
lambda x: x[0], reverse = True)]

                quant_cmap = quant_cmap if isinstance(quant_cmap, str) else "viridis"
                colorbar_width = colorbar_width if isinstance(colorbar_width, float) and
colorbar_width > 0.0 else 0.02

                mosaic_collection.set_array(numpy.array(member_colors))
                mosaic_collection.set_cmap(quant_cmap)

                colorbar_parameters = {
                        "width": 0.1,           #Width of the ticks
                        "length": 2.0,          #Length of the ticks
                        "labelsize": 3.0,       #Tick label text size
                        "pad": 2.0                      #Distance between ticks and tick labels
                }

                #Ax dimensions (left, bottom, width, height) and parameters for colorbar ax
depending on position in figure
                colorbar_pos_params = {
                        "top": ((0.1725, 0.9, 0.679, colorbar_width),
                                        {"top": True, "bottom": False, "labeltop": True,
"labelbottom": False}),
                        "right": ((0.87, 0.11, colorbar_width, 0.77),
                                         {"right": True, "left": False, "labelright": True,
"labelleft": False}),
                        "bottom": ((0.1725, 0.07, 0.679, colorbar_width),
                                          {"top": False, "bottom": True, "labeltop": False,
"labelbottom": True}),
                        "left": ((0.135, 0.11, colorbar_width, 0.77),
                                        {"right": False, "left": True, "labelright": False,
"labelleft": True})
                }

                colorbar_horizontal = False #Display colorbar horizontally if positioned on the
top or bottom of figure

                if isinstance(colorbar_pos, str): #Figure out what cardinal direction the
colorbar axis should be placed:
                        colorbar_pos = colorbar_pos.lower()
                        if any([pos in colorbar_pos for pos in ["top", "north", "up"]]):
                                colorbar_pos = "top"
                                colorbar_horizontal = True
                        elif any([pos in colorbar_pos for pos in ["right", "east"]]):
                                colorbar_pos = "right"
                        elif any([pos in colorbar_pos for pos in ["bottom", "south", "down"]]):
                                colorbar_pos = "bottom"
                                colorbar_horizontal = True
                        elif any([pos in colorbar_pos for pos in ["left", "west"]]):
                                colorbar_pos = "left"
                        else:
                                colorbar_pos = "right"
                else:
                        colorbar_pos = "right"

                colorbar_ax = fig.add_axes(colorbar_pos_params[colorbar_pos][0])
                colorbar_parameters.update(colorbar_pos_params[colorbar_pos][1])

                if colorbar_horizontal:
                        mosaic_colorbar = fig.colorbar(mosaic_collection, cax = colorbar_ax,
orientation = "horizontal")
                        colorbar_ax.xaxis.set_major_formatter(StrMethodFormatter("{x:.1%}"))
                else:
                        mosaic_colorbar = fig.colorbar(mosaic_collection, cax = colorbar_ax,
orientation = "vertical")
                        colorbar_ax.yaxis.set_major_formatter(StrMethodFormatter("{x:.1%}"))

                colorbar_ax.tick_params(**colorbar_parameters)
                mosaic_colorbar.outline.set_linewidth(0.1)

        ax.add_collection(mosaic_collection)

        if not multiple_fig:
                if fig_name is not None:
                        fig.savefig(fig_name, bbox_inches = bbox_inches)
                else:
                        plt.show()

def Multiple_Mosaic(data_list):
        pass

def Rarefaction_Plot():
        pass

def Diversity_Plot(clone_dfs, fig_name = None, title = None):
        fig = plt.figure(dpi = 600)
        ax = fig.add_subplot(1, 1, 1)


vgene_colors = {
        "IGHV1-17": (1.0, 0.0, 0.0, 1.0),
        "IGHV1-18": (1.0, 0.09264715147068088, 0.0, 1.0),
        "IGHV1-2": (1.0, 0.18529430294136176, 0.0, 1.0),
        "IGHV1-24": (1.0, 0.2779414544120426, 0.0, 1.0),
        "IGHV1-3": (1.0, 0.37058860588272352, 0.0, 1.0),
        "IGHV1-45": (1.0, 0.46323575735340439, 0.0, 1.0),
        "IGHV1-46": (1.0, 0.5558829088240852, 0.0, 1.0),
        "IGHV1-58": (1.0, 0.64853006029476612, 0.0, 1.0),
        "IGHV1-67": (1.0, 0.76433899963311702, 0.0, 1.0),
        "IGHV1-68": (1.0, 0.85698615110379794, 0.0, 1.0),
        "IGHV1-69": (0.99595556580850708, 0.94558886838298584, 0.0, 1.0),
        "IGHV1-8": (0.95771954595484032, 1.0, 0.0, 1.0),
        "IGHV2-10": (0.8650723944841594, 1.0, 0.0, 1.0),
        "IGHV2-26": (0.77242524301347859, 1.0, 0.0, 1.0),
        "IGHV2-5": (0.67977809154279767, 1.0, 0.0, 1.0),
        "IGHV2-70": (0.58713094007211675, 1.0, 0.0, 1.0),
        "IGHV3-11": (0.47132200073376579, 1.0, 0.0, 1.0),
        "IGHV3-13": (0.37867484926308459, 1.0, 0.0, 1.0),
        "IGHV3-15": (0.28602769779240411, 1.0, 0.0, 1.0),
        "IGHV3-16": (0.19338054632172286, 1.0, 0.0, 1.0),
        "IGHV3-19": (0.10073339485104227, 1.0, 0.0, 1.0),
        "IGHV3-20": (0.023528747793453701, 1.0, 0.015442504413092598, 1.0),
        "IGHV3-21": (0.0, 1.0, 0.084560769107334968, 1.0),
        "IGHV3-22": (0.0, 1.0, 0.17720733690405541, 1.0),
        "IGHV3-23": (0.0, 1.0, 0.29301554664995555, 1.0),
        "IGHV3-25": (0.0, 1.0, 0.3856621144466757, 1.0),
        "IGHV3-30": (0.0, 1.0, 0.47830868224339584, 1.0),
        "IGHV3-33": (0.0, 1.0, 0.57095525004011594, 1.0),
        "IGHV3-35": (0.0, 1.0, 0.66360181783683614, 1.0),
        "IGHV3-36": (0.0, 1.0, 0.75624838563355612, 1.0),
        "IGHV3-38": (0.0, 1.0, 0.84889495343027632, 1.0),
        "IGHV3-43": (0.0, 1.0, 0.96470316317617644, 1.0),
        "IGHV3-47": (0.0, 0.94264990772343771, 1.0, 1.0),
        "IGHV3-48": (0.0, 0.85000275625275612, 1.0, 1.0),
        "IGHV3-49": (0.0, 0.75735560478207531, 1.0, 1.0),
        "IGHV3-52": (0.0, 0.6647084533113945, 1.0, 1.0),
        "IGHV3-53": (0.0, 0.57206130184071435, 1.0, 1.0),
        "IGHV3-60": (0.0, 0.47941415037003288, 1.0, 1.0),
        "IGHV3-62": (0.0, 0.38676699889935195, 1.0, 1.0),
        "IGHV3-64": (0.0, 0.27095805956100094, 1.0, 1.0),
        "IGHV3-65": (0.0, 0.17831090809032013, 1.0, 1.0),
        "IGHV3-66": (0.0, 0.08566375661963932, 1.0, 1.0),
        "IGHV3-7": (0.023161131617013827, 0.016177736765972346, 1.0, 1.0),
        "IGHV3-71": (0.099630546321722385, 0.0, 1.0, 1.0),
        "IGHV3-72": (0.19227769779240333, 0.0, 1.0, 1.0),
        "IGHV3-73": (0.28492484926308431, 0.0, 1.0, 1.0),
        "IGHV3-74": (0.37757200073376529, 0.0, 1.0, 1.0),
        "IGHV3-76": (0.49338094007211653, 0.0, 1.0, 1.0),
        "IGHV3-9": (0.58602809154279745, 0.0, 1.0, 1.0),
        "IGHV4-28": (0.67867524301347837, 0.0, 1.0, 1.0),
        "IGHV4-31": (0.7713223944841594, 0.0, 1.0, 1.0),
        "IGHV4-34": (0.86396954595484032, 0.0, 1.0, 1.0),
        "IGHV4-39": (0.95661669742552136, 0.0, 1.0, 1.0),
        "IGHV4-4": (0.99558794963206743, 0.0, 0.94632410073586526, 1.0),
        "IGHV4-55": (1.0, 0.0, 0.85808899963311702, 1.0),
        "IGHV4-59": (1.0, 0.0, 0.74228006029476601, 1.0),
        "IGHV4-61": (1.0, 0.0, 0.64963290882408509, 1.0),
        "IGHV5-51": (1.0, 0.0, 0.55698575735340428, 1.0),
        "IGHV5-78": (1.0, 0.0, 0.46433860588272341, 1.0),
        "IGHV6-1": (1.0, 0.0, 0.37169145441204254, 1.0),
        "IGHV7-27": (1.0, 0.0, 0.27904430294136173, 1.0),
        "IGHV7-81": (1.0, 0.0, 0.18639715147068092, 1.0)
}

jgene_colors = {
        "IGHJ1": (0.2235294117647059, 0.23137254901960785, 0.4745098039215686),
        "IGHJ2": (0.3215686274509804, 0.32941176470588235, 0.6392156862745098),
        "IGHJ3": (0.4196078431372549, 0.43137254901960786, 0.8117647058823529),
        "IGHJ4": (0.611764705882353, 0.6196078431372549, 0.8705882352941177),
        "IGHJ5": (0.38823529411764707, 0.4745098039215686, 0.2235294117647059),
        "IGHJ6": (0.5490196078431373, 0.6352941176470588, 0.3215686274509804),
        "IGHJ2P": (0.7098039215686275, 0.8117647058823529, 0.4196078431372549)
}

def VJ_Gene_Plot(clone_df, vgene_col = "VGene", jgene_col = "JGene", count_col = "Clustered", 
v_colormap = vgene_colors,
                                  j_colormap = jgene_colors, vj_gap = 0.1, v_edgecolor = "black", 
j_edgecolor = "black", linewidth = 0.2,
                                  vgene_gap = 0.0, fig_name = None, title = None, figsize = (4, 
4), dpi = 600):
        """Creates a paired V-J gene usage hierarchical donut plot, with V genes in the inner ring 
and J genes in the outer.

        Parameters
        ----------
        clone_df: pandas.DataFrame
                The unique clone member DataFrame with the V and J gene calls plus clone member 
counts / frequencies.
```

```
        vgene_col: str
                The name for the column in clone_df containing the V gene calls; default is
"VGene".
        jgene_col: str
                The name for the column in clone_df containing the J gene calls; default is
"JGene".
        count_col: str
                The name for the column in clone_df containing the clone member counts or
frequencies; default is "Clustered".
        v_colormap: dict of {str: color}
                A dict of V gene names to the corresponding color values for the gene; see
vgene_colors for default.
        j_colormap: dict of {str: color}
                A dict of J gene names to the corresponding color values for the gene; see
jgene_colors for default.
        vj_gap: float
                The amount of space between the outer and inner rings, in figure unit dimensions;
default is 0.1.
        v_edgecolor: color
                Name of a valid matplotlib color, RGB tuple, etc. to use for the V gene segment
borders; default is "black".
        j_edgecolor: color
                Name of a valid matplotlib color, RGB tuple, etc. to use for the J gene segment
borders; default is "black".
        linewidth: float
                The size (width) of the border lines surrounding the V/J gene segments; default
is 0.2.
        vgene_gap: float
                An angle in degrees to use as a gap between the V gene sections; default is 0.
        fig_name: str or None
                Name of the output file for the figure; if left None (the default) the plot is
displayed on screen.
        title: str or None
                Optional title for the figure (matplotlib ax.set_title).
        figsize: tuple of (int, int)
                A tuple of two integers/floats describing the figure width and height in inches;
default is (4, 4).
        dpi: int
                The Dots Per Inch for the figure; default is 600.
        """

        if vgene_col not in clone_df.columns:
                print("Error in VJ_Gene_Plot: V gene column \"{0}\" not found in
DataFrame!".format(vgene_col))
                return None
        if jgene_col not in clone_df.columns:
                print("Error in VJ_Gene_Plot: J gene column \"{0}\" not found in
DataFrame!".format(jgene_col))
                return None
        if count_col not in clone_df.columns:
                print("Error in VJ_Gene_Plot: clone count/frequency column \"{0}\" not found in
DataFrame!".format(count_col))
                return None

        fig = plt.figure(figsize = figsize, dpi = dpi)
        ax = fig.add_subplot(1, 1, 1, aspect = "equal")
        ax.set_axis_off()
        ax.invert_xaxis() #This ensures the plot is displayed clockwise.

        #v_radius and v_width are the inner ring diameter and width; j_radius and j_width are for
the outer ring.
        v_radius = 0.345
        v_width = 0.15
        j_radius = 0.45
        j_width = 0.1
        center = (0.5, 0.5)

        gene_df = clone_df[[vgene_col, jgene_col, count_col]].groupby([vgene_col,
jgene_col]).agg({count_col: sum})
        gene_df = gene_df.sort_index()   #First sorts by V gene ascending, then J gene ascending.
        gene_df = gene_df.reset_index()  #Now returns the VGene and JGene columns to the
DataFrame.

        total_vgenes = len(gene_df[vgene_col].drop_duplicates())
        total_gapsize = total_vgenes * vgene_gap
        remaining_size = 360.0 - float(total_gapsize)
        gap_size = float(vgene_gap)

        total_counts = gene_df[count_col].sum()
        gene_df["Arc_Length"] = gene_df[count_col] / total_counts * remaining_size
        cur_v_start = 90.0 + (gap_size / 2.0) #Starting at 90 degrees (top center of the circle)
plus half the gap size.

        ring_patches = []
        for vgene in gene_df[vgene_col].drop_duplicates():
                cur_vgene_df = gene_df[gene_df[vgene_col] == vgene]

                v_color = v_colormap[vgene]
                v_arc_length = cur_vgene_df["Arc_Length"].sum()
                cur_v_end = cur_v_start + v_arc_length

                inner_patch = Wedge(center, v_radius, cur_v_start, cur_v_end, v_width,
                                                        facecolor = v_color, edgecolor =
v_edgecolor, lw = linewidth)
                ring_patches.append(inner_patch)

                cur_j_start = cur_v_start
                for jgene, jgene_arc_length in zip(cur_vgene_df[jgene_col],
cur_vgene_df["Arc_Length"]):
                        j_color = j_colormap[jgene]
                        cur_j_end = cur_j_start + jgene_arc_length

                        outer_patch = Wedge(center, j_radius, cur_j_start, cur_j_end, j_width,
                                                                facecolor = j_color, edgecolor =
j_edgecolor, lw = linewidth)
                        ring_patches.append(outer_patch)

                        cur_j_start = cur_j_end

                cur_v_start = cur_v_end + gap_size

        ax.add_collection(PatchCollection(ring_patches, match_original = True))

        if title is not None:
                ax.set_title(title)

        if fig_name is not None:
                fig.savefig(fig_name, bbox_inches = "tight")
        else:
                plt.show()

####FOR TESTING:
if __name__ == "__main__":
        test_clone_counts = []
        with open("clones_test.txt", "r") as clone_test:
                for line in clone_test:
                        line = line.strip()
                        test_clone_counts.append(float(line))
        sm = test_clone_counts[:600]
        rank_graph = Rank_Abundance_Graph(sm)
        rank_graph.savefig("t1b", bbox_inches = "tight", dpi = 600)
        rank_graph3 = Rank_Abundance_Graph(sm, figsize = (8, 5))
        rank_graph3.savefig("t3b", bbox_inches = "tight", dpi = 600)
        plt.show()

        test_clones20 = [0.9, 7.5, 6.0, 5.9, 5.5, 4.0, 2.8, 2.1, 1.3, 1.0, 0.5, 0.5, 0.5, 0.4, 
0.4, 0.4, 0.3, 0.2, 0.2, 0.1]
        test_shm20 = [0.0, 0.1, 0.1, 0.2, 0.2, 0.3, 0.1, 0.2, 0.3, 0.1, 0.3, 0.3, 0.1, 0.0, 0.3, 
0.3, 0.3, 0.3, 0.3, 0.3]
        Mosaic_Plot(test_clones20)
        Mosaic_Plot(test_clones20, highlight_upper_percent = 0.7)
        #Mosaic_Plot(test_clones20, test_shm20)
        #Mosaic_Plot(test_clones20, test_shm20, colorbar_pos = "bottom")

##########Repertoire Diversity
def Shannon_Wiener_Index(clone_counts):
        """Calculates the Shannon-Wiener index of diversity: -sum(x * ln(x)) for clone frequencies 
x.

        Parameters
        ----------
        clone_counts: list of ints/floats
                The count or frequencies of each clone in a sample as a list/iterable.

        Returns
        ----------
        sw_index: float
                The Shannon-Wiener index value for clone_counts.
        """

        if not hasattr(clone_counts, "__iter__") or isinstance(clone_counts, str):
                print("Error in Shannon_Wiener_Index: clone_counts must be a list/iterable of 
clone counts/frequencies!")
                return None

        total_clone_counts = float(sum(clone_counts))
        sw_index = 0.0
        for clone in clone_counts:
                clone_freq = clone / total_clone_counts
                sw_index -= (clone_freq * numpy.log(clone_freq))

        return sw_index

def Gini_Simpson_Index(clone_counts):
        """Calculates the Gini-Simpson index of diversity: sum(x ^ 2) for clone frequencies x.

        Parameters
        ----------
        clone_counts: list of ints/floats
                The count or frequencies of each clone in a sample as a list/iterable.

        Returns
        ----------
        gs_index: float
                The Gini-Simpson index value for clone_counts.
        """

        if not hasattr(clone_counts, "__iter__") or isinstance(clone_counts, str):
                print("Error in Gini_Simpson_Index: clone_counts must be a list/iterable of clone 
counts/frequencies!")
                return None

        total_clone_counts = float(sum(clone_counts))
        gs_index = 0.0
        for clone in clone_counts:
                clone_freq = clone / total_clone_counts
                gs_index += (clone_freq * clone_freq)

        return gs_index

def Hill_Diversity_Index(clone_counts, N = (0.0, 10.0), step = 0.1):
        """Calculates the Hill diversity metrics: sum(x ^ Q) ^ 1/(1-Q) for clone frequencies x and
positive order Q.
        By default, returns a list of the Hill indices from 0 to 10 with step of 0.1, but can also
return a single index.
        Order acts as a "true diversity" measure, whether high or low abundance clones have the
most influence on diversity.
        For Q of 0, species frequency has zero effect; this is the "species richness", ie. the
number of unique clones.
        For 0 < Q < 1, rare species contribute more to the diversity rating than abundant species.
        For Q of 1, rare and abundant species are equally weighted contributors to diversity; same
as exp(Shannon-Wiener).
        For Q > 1, abundant clones contribute most to diversity (at Q of 2, equal to 1/Gini-
Simpson).

        Parameters
        ----------
        clone_counts: list of ints/floats
                The count or frequencies of each clone in a sample as a list/iterable.
        N: int/float or (int/float, int/float)
                Either tuple of (start_order, end_order) or single float of the order number to
calculate; default: (0.0, 10.0)
        step: int/float
                The step increment for the count from start_order to end_order if N is not a
single number; default: 0.1

        Returns
        ----------
        hill_index or hill_indices: (float, float) or list of (float, float)
                If N is one order, returns the order and index as (order, index); otherwise
returns a list of (order, index)
        """

        if not hasattr(clone_counts, "__iter__") or isinstance(clone_counts, str):
                print("Error in Hill_Diversity_Index: clone_counts must be a list/iterable of
clone counts/frequencies!")
                return None

        if hasattr(N, "__iter__"):
                if len(N) != 2 or N[1] <= N[0]:
                        print("Error in Hill_Diversity_Index: if N is an iterable it must be of
length 2 for the start/end orders.")
                        return None

                chunks = numpy.floor(N[1] / step) + 1
                orders = numpy.linspace(start = N[0], stop = N[1], num = chunks)

        else:
                orders = [N]

        total_clone_counts = float(sum(clone_counts))
        hill_indices = []

        for order in orders:
                hill_index = 0.0

                if order == 0.0:
                        hill_index = len(clone_counts) #Hill index at zero is simply the species
richness (total number of clones)
```

```
		elif order == 1.0:
			sw_index = Shannon_Wiener_Index(clone_counts)
			hill_index = numpy.exp(sw_index) #Hill index at one is the exponential of 
the Shannon-Wiener index

		else:
			for clone in clone_counts:
				clone_freq = clone / total_clone_counts
				hill_index += (clone_freq ** order)

			order_exponent = 1.0 / (1.0 - order)
			hill_index = hill_index ** order_exponent

		order_index = (order, hill_index)
		hill_indices.append(order_index)

	if len(hill_indices) == 1:
		return hill_indices[0]
	else:
		return hill_indices

def Berger_Parker_Index(clone_counts):
	"""Calculates the Berger-Parker index of diversity: just the frequency of the most 
prevalent clone in a sample.

	Parameters
	----------
	clone_counts: list of ints/floats
		The count or frequencies of each clone in a sample as a list/iterable.

	Returns
	----------
	bp_index: float
		The Berger-Parker index value for clone_counts.
	"""

	if not hasattr(clone_counts, "__iter__") or isinstance(clone_counts, str):
		print("Error in Berger_Parker_Index: clone_counts must be a list/iterable of 
clone counts/frequencies!")
		return None

	total_clone_counts = float(sum(clone_counts))
	bp_index = 0.0
	for clone in clone_counts:
		clone_freq = clone / total_clone_counts

		if clone_freq > bp_index:
			bp_index = clone_freq

	return bp_index

def Diversity_Index(clone_counts, ratio = 0.5):
	"""Calculates the simple diversity index: the minimum fraction of all clones that make up 
x% of the total counts.
	By default, the calculated index is set at 50%.

	Parameters
	----------
	clone_counts: list of ints/floats
		The count or frequencies of each clone in a sample as a list/iterable.
	ratio: float
		The ratio/fraction (0.0 < ratio < 1.0) of total clone counts/frequencies to use; 
default is 0.5.

	Returns
```

```
        ----------
        div_index: float
                The diversity index value for clone_counts.
        """

        if not hasattr(clone_counts, "__iter__") or isinstance(clone_counts, str):
                print("Error in Diversity_Index: clone_counts must be a list/iterable of clone
counts/frequencies!")
                return None

        if not 0.0 < ratio < 1.0:
                print("Error in Diversity_Index: ratio must be a float greater than 0.0 and less
than 1.0!")
                return None

        total_clone_counts = float(sum(clone_counts))
        total_clones = float(len(clone_counts))
        sorted_clones = sorted(clone_counts, reverse = True)

        cur_percent = 0.0
        cur_clones = 0
        for clone in sorted_clones:
                clone_freq = clone / total_clone_counts
                cur_percent += clone_freq
                cur_clones += 1

                if cur_percent >= ratio:
                        div_index = float(cur_clones) / total_clones
                        break

        return div_index


###########File utilities
import gzip
import os

def Decompress_GZIP(filename, output_filename = None):
        """Takes a compressed gzip text file (such as a fastq.gz sequence file) and saves the
decompressed version.

        Parameters
        ----------
        filename: str
                Name of the gzip file to load for decompression.
        output_filename: str or None
                Name to use for the decompressed file; if None, the filename is same as the input
file minus the trailing ".gz".
        """

        if output_filename is None:
                output_filename = ".".join(filename.split(".")[:-1])

        with gzip.open(filename, "rt") as compressed_file:
                file_contents = compressed_file.read()

        with open(output_filename, "w", newline = "\n") as decompressed_file:
                decompressed_file.write(file_contents)

def Textfile_to_GZIP(filename, output_filename = None):
        """Opens a text file (such as a .fastq sequence file) and saves a gzip compressed .gz
version.

        Parameters
        ----------
```

```
	filename: str
		Name of the text file to be compressed.
	output_filename: str or None
		Name to use for the compressed file; if None, the input filename is used with an
appended ".gz" extension.
	"""

	if output_filename is None:
		output_filename = filename + ".gz"

	with open(filename, "r") as text_file:
		file_contents = text_file.read()

	with gzip.open(output_filename, "wt") as compressed_file:
		compressed_file.write(file_contents)

def Stitch_Reads():
	pass

def Filter_Trim_Reads():
	pass

def Rename_FASTQ_Headers(filename, output_filename, header_prefix, keep_illumina_pos_and_read =
True):
	"""Creates a copy of the "filename" FASTQ file with modified read headers, optionally
keeping Illumina read info.
	Useful for ensuring the origin of reads when multiple different sequence files are merged
before MIXCR processing.

	Parameters
	----------
	filename: str
		Filename of the FASTQ file to modify.
	output_filename: str
		Name for the modified output file.
	header_prefix: str
		Header prefix used for each modified read.
	keep_illumina_pos_and_read: bool
		Whether to save the Illumina MiSeq/HiSeq flowcell lane, xy position, and read
direction (R1/R2) for all reads.
		If False, a simple read counter (starting at 1) is appended to the header prefix.
		Example: "@M02288:110:000000000-AE88A:1:1101:8994:1790 1:N:0:2" ->
"@Prefix_1101:8994:1790:R1"
	"""

	with open(filename, "r") as fastq_in:
		with open(output_filename, "w", newline = "\n") as fastq_out: #Non-windows
newline is important for MIXCR!
			cur_line = 0
			cur_seq = 1
			modified_prefix = "@" + header_prefix + "_"

			for line in fastq_in:
				line = line.strip()

				if not line.startswith("@") and cur_line == 0:
					fastq_out.write(line + "\n") #Write any pre-data comment
lines and move on
					continue

				if line.startswith("@") and cur_line % 4 == 0:
					if keep_illumina_pos_and_read:
						read_num = ""
						if len(line.split(" ")) > 1:
							illumina_read = line.split(" ")[1]
							read_num = "_R" + 
illumina_read.split(":")[0]

							line = line.split(" ")[0]

						illumina_fields = line.split(":")
						lane_xy = ":".join(illumina_fields[4:7])
						cur_header = modified_prefix + lane_xy + read_num

						fastq_out.write(cur_header + "\n")

					else:
						fastq_out.write(modified_prefix + str(cur_seq) + 
"\n")

					cur_seq += 1

				else:
					fastq_out.write(line + "\n")

				cur_line += 1

def Concat_FASTQ(filenames, output_filename, rename_headers = None, keep_illumina_pos_and_read = 
True):
	"""Concatenates all FASTQs in filenames with the ability to modify read headers, 
optionally keeping Illumina info.

	Parameters
	----------
	filenames: list of str
		File names of the input FASTQ files to concatenate.
	output_filename: str
		Name for the merged output FASTQ file.
	rename_headers: list of str
		Optional; header prefixes to use for each file in filenames - must be the same 
length as filenames.
	keep_illumina_pos_and_read: bool
		Whether to save the Illumina MiSeq/HiSeq flowcell lane, xy position, and read 
direction (R1/R2) for all reads.
		If False, a simple read counter (starting at 1) is appended to the header prefix.
		Example: "@M02288:110:000000000-AE88A:1:1101:8994:1790 1:N:0:2" -> 
"@Prefix_1101:8994:1790:R1"

	Returns
	----------
	success: bool
		Returns True if no errors occur and False otherwise.
	"""

	for fastq_file in filenames:
		if not os.path.exists(fastq_file):
			print("Error in Concat_FASTQ - can't find input file: " + fastq_file)
			return False

	if rename_headers is not None:
		if not all([hasattr(rename_headers, "__iter__"), len(rename_headers) == 
len(filenames)]):
			print("Error in Concat_FASTQ - rename_headers must be a list of the same 
length as filenames.")
			return False

	with open(output_filename, "w", newline = "\n") as fastq_out:
		if rename_headers is not None:
			for fastq, renamed_header in zip(filenames, rename_headers):
```

```
                                Rename_FASTQ_Headers(fastq, "temp_renamed_headers.fastq", 
renamed_header, keep_illumina_pos_and_read)

                                with open("temp_renamed_headers.fastq", "r") as fastq_in:
                                        for line in fastq_in:
                                                line = line.strip()
                                                fastq_out.write(line + "\n")

                                try:
                                        os.remove("temp_renamed_headers.fastq")
                                except:
                                        print("Warning in Concat_FASTQ - couldn't delete 
temporary file.")

                else:
                        for fastq in filenames:
                                with open(fastq, "r") as fastq_in:
                                        for line in fastq_in:
                                                line = line.strip()
                                                fastq_out.write(line + "\n")

        return True

def FASTQ_to_FASTA(filename, output_filename = None, seqs_to_upper = False, remove_ambig_seqs = 
False, line_maxlen = 0):
        """Converts a FASTQ sequence file to FASTA, optionally with ambiguous N-containing 
sequences removed.

        Parameters
        ----------
        filename: str
                Filename of the FASTQ file to convert.
        output_filename: str or None
                Name for the converted FASTA output file; if None, the FASTQ extension is 
replaced with ".fasta".
        seqs_to_upper: bool
                If True, convert all sequence characters to uppercase (default is False).
        remove_ambig_seqs: bool
                If True, sequences containing any ambiguous bases (N) will be dropped from the 
output FASTA (default is False).
        line_maxlen: int
                Number of characters per sequence-containing line to write for the output FASTA; 
default of 0 means no limit.
        """

        if output_filename is None:
                output_filename = ".".join(filename.split(".")[:-1]) + ".fasta"

        line_maxlen = line_maxlen if isinstance(line_maxlen, int) and line_maxlen >= 0 else 0

        headers = []
        sequences = []

        with open(filename, "r") as fastq_file:
                cur_line = 0

                for line in fastq_file:
                        line = line.strip()

                        if not line.startswith("@") and cur_line == 0:
                                continue #In case the FASTQ file doesn't immediately start off 
with a sequence, move on to the next line

                        if line.startswith("@") and cur_line % 4 == 0:
                                headers.append(">" + line[1:])
                        if cur_line % 4 == 1:
                                if seqs_to_upper:
                                        line = line.upper()

                                sequences.append(line)

                        cur_line += 1

        with open(output_filename, "w", newline = "\n") as fasta_file: #Non-windows newline is
important for MIXCR!
                for header, sequence in zip(headers, sequences):
                        if remove_ambig_seqs and "N" in sequence.upper():
                                continue

                        fasta_file.write(header + "\n")

                        if line_maxlen == 0:
                                fasta_file.write(sequence + "\n")
                        else:
                                for i in range(0, len(sequence), line_maxlen):
                                        sub_seq = sequence[i : i + line_maxlen]
                                        fasta_file.write(sub_seq + "\n")


#########Repertoire utilities

import pandas
import os

compartment_types = {
        "PBMC": (),
        "BMMC": (),
        "Memory": (),
        "PlasmaCell": (),
        "Plasmablast": ()
}

clone_agg_funcs = {
        "CDR3_NT": "top",
        "CDR1_AA": "top",
        "CDR2_AA": "top",
        "CDR3_AA": "top",
        "VGene": "top",
        "JGene": "top",
        "Isotype": "top",
        "V_SHM": "mean",
        "J_SHM": "mean",
        "Compartment": "multiple",
        "Sample": "multiple",
        "Donor": "uniques"
}

default_clonotyping_params = {
        "identity": 0.96,
        "min_len": 5,
        "method": "usearch",
        "max_accepts": 0,
        "max_rejects": 0        #, "require_same_V_gene" (or family): False,
"require_same_J_gene" (or family): False
}

def Repertoire_Stats(clone_df):
        pass
```

```
def Group_Clones(seq_df, clone_col = "CloneID", agg_funcs = clone_agg_funcs, read_count_col =
None, descending = True,
                                 clone_count_col_out = "Clustered", clone_freq_col_out =
"Frequency"):
        """Groups a DataFrame of reads together by clone ID into a DataFrame of clones.
        Allows customization of clone feature aggregation and sorting by clone prevalence.
        By default Group_Clones does not take into account read counts since data from cDNA IgSeq
can be skewed during PCR.
        Read information columns to use and the functions for aggregation are defined by the
agg_funcs argument.
        Special functions for aggregation are:
                "top": picks the most common value for a feature (most common gene called out of
all genes called for a clone).
                "multiple": yields "Multiple" if a clone feature has more than one unique value,
else returns the sole value.
                "uniques": gives the number of unique values found for a clone (for example, 3 if
three donors share a clone).
        Any other functions found in agg_funcs must be usable by pandas agg() function (examples:
"mean", "sum", "count").

        Parameters
        ----------
        seq_df: pandas.DataFrame
                Sequence dataframe containing sequences to be clustered by clone ID.
        clone_col: str
                Column name containing the clonotype ID for each sequence; default is "CloneID".
        agg_funcs: dict of {str: str or function}
                Dictionary of column name to aggregation function to be used for the feature; for
default see clone_agg_funcs.
        read_count_col: str or None
                Column containing the counts per read to use for calculating clonotype abundance.
                If not provided, read counts are discarded; default is None, since cDNA IgSeq
read counts are biased by PCR.
        descending: bool
                Whether to sort the clones by descending clone relative frequency / abundance;
default is True.
        clone_count_col_out: str
                Name of output column in clone_df with the number of reads aggregated per clone;
default is "Clustered".
        clone_freq_col_out: str
                Name of output column in clone_df with the clone abundance/frequency (from 0.0 to
1.0); default is "Frequency".

        Returns
        ----------
        clone_df: pandas.DataFrame
                Pandas DataFrame containing the aggregated clone members and their properties.
        """

        if clone_col not in seq_df.columns:
                print("Error in Group_Clones: clone ID column \"{0}\" not found in sequence
DataFrame!".format(clone_col))
                return None

        agg_dict = {}
        for feature_col in agg_funcs:
                if feature_col in seq_df:
                        agg_dict[feature_col] = agg_funcs[feature_col]

        selected_cols = [clone_col] + [col for col in agg_dict]
        read_feature_df = seq_df[selected_cols]
        total_initial_reads = len(read_feature_df)

        read_feature_df = read_feature_df[read_feature_df[clone_col].notnull()]
        cur_reads = len(read_feature_df)
        if cur_reads < total_initial_reads:
                dropped = total_initial_reads - cur_reads
                print("{0} reads dropped with no clone ID ({1:.1%} of total).".format(dropped,
(dropped / total_initial_reads)))

        top_lambda = lambda x: x.value_counts().index[0]
        has_multiple_lambda = lambda x: "Multiple" if len(x.drop_duplicates()) > 1 else
x.drop_duplicates().tolist()[0]
        count_unique_lambda = lambda x: len(x.drop_duplicates())

        clone_aggregator = {clone_col: "count"}

        for col, func in agg_dict.items():
                if col in read_feature_df.columns:
                        if col == read_count_col:
                                continue

                        if isinstance(func, str) and "top" in func.lower():
                                read_feature_df[col] = read_feature_df[col].fillna("")
                                clone_aggregator[col] = top_lambda

                        elif isinstance(func, str) and "multiple" in func.lower():
                                clone_aggregator[col] = has_multiple_lambda

                        elif isinstance(func, str) and "unique" in func.lower():
                                clone_aggregator[col] = count_unique_lambda

                        else:
                                clone_aggregator[col] = func

        clone_df = read_feature_df.groupby([clone_col]).agg(clone_aggregator)
        clone_df = clone_df.rename(columns = {clone_col: clone_count_col_out})

        if read_count_col is not None and read_count_col in read_feature_df.columns:
                clone_df[clone_count_col_out] =
read_feature_df.groupby([clone_col])[read_count_col].sum()

        if descending:
                clone_df = clone_df.sort_values([clone_count_col_out], ascending = [False])

        if clone_freq_col_out is not None:
                total_counts = float(clone_df[clone_count_col_out].sum())
                clone_df[clone_freq_col_out] = clone_df[clone_count_col_out].astype(float) /
total_counts

        clone_df = clone_df.reset_index()

        return clone_df

def Clonotype_Sequences(seq_df, feature_col = "CDR3_NT", clonotyping_params =
default_clonotyping_params,
                                                 clone_col_out = None, for_rarefaction = False,
usearch_loc = usearch_exe):
        """Clonotypes sequences by gene feature(s) such as the nucleotide CDR3 sequence, returning
a column of the determined clone IDs.
        Currently the only available clonotyping method is "usearch".

        By default function will return the seq_df with a new column containing the clone IDs, but
can also return the count of total clones
        as an int by setting for_rarefaction to True (faster return when clonotyping many
subsamples during rarefaction analysis).

        Parameters
        ----------
        seq_df: pandas.DataFrame
                Input sequence dataframe to be clonotyped.
        method: str
                Method for determining clonal membership; current implementation only allows for
"usearch".
                Default: "usearch"
        parameters: dict
                A dict of str parameter names to values used to customize clonotype
identification.
                Default: see default_clonotyping_params
        for_rarefaction: bool
                For clone vs read count rarefaction analysis, only return the number of unique
clones as an int (more efficient quick return).
                Default: False; the full seq_df with the additional clone identification column
will be returned.
        usearch_loc: str
                Path location of the binary executable for usearch clonotyping.
                Default: see usearch_exe_loc
        uc_dtypes: dict of str: type
                Column names and datatypes for processing the .uc tab-separated output file
produced by usearch.
                Default: see usearch_col_dtypes

        Returns
        ----------
        seq_df: pandas.DataFrame
                The original input seq_df plus the additional column of clone membership ID for
each sequence.

        OR

        total_clones: int
                The total number of unique clones in the dataframe (returned when for_rarefaction
= True)
        """

        method_funcs = ["usearch"]

        if "method" not in clonotyping_params:
                print("Warning in Clonotype_Sequences: no clonotyping method given, defaulting to
USEARCH.")
                clonotyping_method = "usearch"

        elif isinstance(clonotyping_params["method"], str) and
clonotyping_params["method"].lower() not in method_funcs:
                print("Error in Clonotype_Sequences: invalid method \"{0}\"; options
are:".format(clonotyping_params["method"]))
                for m in method_funcs:
                        print("--" + m)

                return None

        else:
                clonotyping_method = clonotyping_params["method"].lower()

        if "identity" not in clonotyping_params:
                print("Warning in Clonotype_Sequences: no sequence identity % given, defaulting
to 0.96.")
                identity = 0.96

        else:
                if not 0.0 < clonotyping_params["identity"] < 1.0:
                        if 0 < clonotyping_params["identity"] < 100:
                                identity = clonotyping_params["identity"] / 100.0
                        else:
                                print("Error in Clonotype_Sequences: \"identity\" should be a
float between 0.0 and 1.0!")
                                return None

                else:
                        identity = clonotyping_params["identity"]

        column_not_found_error = "Error in Clonotype_Sequences: column \"{0}\" not found in
sequence DataFrame!"
        if hasattr(feature_col, "__iter__") and not isinstance(feature_col, str):
                for col in feature_col:
                        if col not in seq_df.columns:
                                print(column_not_found_error.format(col))
                                return None

                seqs = seq_df[feature_col].sum(axis = 1)
                cols_clonotyped = "".join(feature_col)

        elif isinstance(feature_col, str):
                if feature_col not in seq_df.columns:
                        print(column_not_found_error.format(feature_col))
                        return None

                seqs = seq_df[feature_col]
                cols_clonotyped = feature_col

        else:
                print("Error in Clonotype_Sequences: feature_col to clonotype by must be a column
name / list of column names!")
                return None

        if clone_col_out is None:
                clone_col_out = cols_clonotyped + "_CloneID"

        elif isinstance(clone_col_out, str):
                if clone_col_out in seq_df.columns:
                        print("Error in Clonotype_Sequences: output column \"{0}\" already in
DataFrame!".format(clone_col_out))
                        return None

                if "_cloneid" not in clone_col_out.lower():
                        clone_col_out = clone_col_out + "_CloneID"

        else:
                print("Warning in Clonotype_Sequences: clone_col_out should be a column name or
None to append \"_CloneID\"!")
                clone_col_out = cols_clonotyped + "_CloneID"
                print("Defaulting to \"{0}\"!".format(clone_col_out))

        if "min_len" not in clonotyping_params:
                min_len = 5

        elif isinstance(clonotyping_params["min_len"], int) or
isinstance(clonotyping_params["min_len"], float):
                if clonotyping_params["min_len"] < 1:
                        print("Warning in Clonotype_Sequences: parameter \"min_len\" should be an
int >0; defaulting to 5!")
                        min_len = 5

                else:
                        min_len = int(clonotyping_params["min_len"])

        else:
                print("Warning in Clonotype_Sequences: parameter \"min_len\" should be an int >0;
defaulting to 5!")
                min_len = 5

        if clonotyping_method == "usearch":
                if "max_accepts" in clonotyping_params:
                        if isinstance(clonotyping_params["max_accepts"], int):
                                max_accepts = clonotyping_params["max_accepts"]

                        else:
                                print("Warning in Clonotype_Sequences: parameter \"max_accepts\"
should be an int; defaulting to 0!")
                                max_accepts = 0

                else:
                        max_accepts = 0

                if "max_rejects" in clonotyping_params:
                        if isinstance(clonotyping_params["max_rejects"], int):
                                max_rejects = clonotyping_params["max_rejects"]

                        else:
                                print("Warning in Clonotype_Sequences: parameter \"max_rejects\"
should be an int; defaulting to 0!")
                                max_rejects = 0

                else:
                        max_rejects = 0

                clone_IDs = Clonotype_Usearch(seqs, identity = identity, max_accepts =
max_accepts, max_rejects = max_rejects,
                                                                          min_len = min_len,
usearch_loc = usearch_loc, for_rarefaction = for_rarefaction)

        if clone_IDs is None:
                print("Error in Clonotype_Sequences: clonotyping function failed!")
                return None

        if not for_rarefaction:
                clone_IDs = clone_IDs.rename(columns = {"Clone_ID": clone_col_out})
                seq_df = seq_df.join(clone_IDs, how = "left")

                return seq_df

        else:
                return clone_IDs

def Clonotype_Usearch(seqs, identity = 0.96, max_accepts = 0, max_rejects = 0, min_len = 5,
usearch_loc = usearch_exe,
                                          for_rarefaction = False, delete_temp = True):
        """Clonotypes sequences via Usearch, returning the read clone IDs or the number of clones
for rarefaction analysis.
        By default returns a Series mapping the reads to clone IDs.

        Parameters
        ----------
        seqs: pandas.Series
                Input sequences with the appropriate index to merge resulting clone IDs back to
original reads.
        identity: float or int
                Sequence similarity cutoff for grouping clones; must be a float between 0.0 and
1.0 or an int between 0 and 100.
                Default is 0.96 (sequences having a 96% similarity or more can be grouped
together).
        max_accepts: int
```

```
                Parameter passed to Usearch cluster_fast as maxaccepts; should be an int >= 0 (0 
being disabled); default is 0.
        max_rejects: int
                Parameter passed to Usearch cluster_fast as maxrejects; should be an int >= 0 (0 
being disabled); default is 0.
        min_len: int
                Minimum length of sequence to be clonotyped for placement in a clonal group; 
default is 5.
        usearch_loc: str
                Path / location of the binary executable for Usearch; for default see 
usearch_exe.
        for_rarefaction: bool
                Returns only the number of clones for faster clone vs input read rarefaction 
analysis; default is False.
        delete_temp: bool
                If True, the temporary .fasta Usearch input file and Usearch output .uc files 
will be removed; default is True.

        Returns
        ----------
        clones or total_clones: pandas.DataFrame or int
                The DataFrame mapping read ID to clone ID for the input sequences, or the total 
number of clones identified.
        """

        if not os.path.exists(usearch_loc):
                print("Error in Clonotype_Usearch: Usearch executable not found!")
                return None

        uc_col_names = ("Type", "Clone_ID", "Len_Size", "Identity_to_Centroid", "Orientation",
                                   "NA1", "NA2", "NA3", "Read_ID", "Centroid_Label")

        uc_col_dtypes = {
                "Type": str,
                "Clone_ID": int,
                "Len_Size": int,
                "Identity_to_Centroid": str,
                "Orientation": str,
                "NA1": str,
                "NA2": str,
                "NA3": str,
                "Read_ID": int,
                "Centroid_Label": str
        }

        cluster_seqs_file = "to_cluster.fasta"
        uc_file = "usearch_cluster_IDs.uc"

        with open(cluster_seqs_file, "w") as cluster_seqs_fasta:
                for idx, seq in seqs.iteritems():
                        cluster_seqs_fasta.write(">" + str(idx) + "\n" + str(seq) + "\n")

        if not os.path.exists(cluster_seqs_file):
                print("Error in Clonotype_Usearch: error writing FASTA input file for Usearch!")
                return None

        usearch_call = "{0} --cluster_fast {1}".format(usearch_loc, cluster_seqs_file)
        usearch_call += " --id {0} --maxaccepts {1} --maxrejects {2}".format(identity, 
max_accepts, max_rejects)
        usearch_call += " --minseqlength {0} --top_hit_only --uc {1}".format(min_len, uc_file)
        os.system(usearch_call)

        if not os.path.exists(uc_file):
                print("Error in Clonotype_Usearch: Usearch .uc results file was not found, error 
occurred during clonotyping!")
```

```
                return None

        clones = pandas.read_csv(uc_file, sep = "\t", header = None, names = uc_col_names, dtype = 
uc_col_dtypes)

        if delete_temp:
                os.remove(cluster_seqs_file)
                os.remove(uc_file)

        if not for_rarefaction:
                clones = clones[clones["Type"] != "C"]
                clones = clones[["Clone_ID", "Read_ID"]]
                total_clones = len(clones)
                reads_per_clone = len(seqs) / total_clones

                print("{0} clonotypes identified; average of {1:.1f} reads per 
clone.".format(total_clones, reads_per_clone))

                clones = clones.set_index("Read_ID")
                return clones

        else:
                total_clones = len(clones[clones["Type"] == "S"])
                return total_clones


#########Mass Spec utilities

import os

def DataFrame_to_FASTA(seq_df, filename, header_prefix = None, seq_col = "Sequence", header_cols = 
"ReadID", overwrite = False):
        """

        Parameters
        ----------

        Returns
        ----------
        success: bool
                Whether the FASTA file was successfully written or not.
        """

        if os.path.exists(filename) and not overwrite:
                print("Error in DataFrame_to_FASTA: " + filename + " already exists!")
                print("Pick a new file name, or run DataFrame_to_FASTA with overwrite = True.")
                return False

        if seq_col not in seq_df.columns:
                print("Error in DataFrame_to_FASTA: sequence column \"" + seq_col + "\" not found 
in DataFrame!")
                return False

        if header_prefix is None:
                header_prefix = ">"

        elif isinstance(header_prefix, str):
                if not header_prefix.startswith(">"):
                        header_prefix = ">" + header_prefix

        else:
                print("Error in DataFrame_to_FASTA: header_prefix must be a string or left as the 
default None for no prefix!")
                return False
        fasta_df = seq_df[[seq_col]]
        fasta_df["Header"] = header_prefix

        if hasattr(header_cols, "__iter__") and not isinstance(header_cols, str):
                for field in header_cols:
                        if field in seq_df.columns:
                                pass

                        else:
                                pass

        elif isinstance(header_cols, "str") and header_cols in seq_df.columns:
                pass

        with open(filename, "w", newline = "\n") as fasta:
                for header, seq in zip(seq_df[header_cols], seq_df[seq_col]):
                        fasta.write(">" + header + "\n")
                        fasta.write(seq + "\n")

        success = os.path.exists(filename)
        return success


#########Constants
#Constant region VH-only primer sequences used to identify additional isotypes missed during
annotation
VHonly_primer_IgG = "TCCACCAAGGGCCCAT"
VHonly_primer_IgA = "CTTCTTCCCCCAGGAG"
VHonly_primer_IgM = "AGTGCATCCGCCCCAA"

#Constant region pairing primer sequences used to identify additional isotypes missed during
annotation
pairing_primer_IgA = "ACCAGCCCCAAGCAGGGCCC"
pairing_primer_IgG = "TCCACCAAGGGCCCATC"
pairing_primer_IgM = "AGTGCATCCGCCCCAACCCA"

#By default, remove sequences with a stop codon in any gene region or a frameshift in any region
but FR4 (primer site)
no_stop_feats = ("FR1_AA", "CDR1_AA", "FR2_AA", "CDR2_AA", "FR3_AA", "CDR3_AA", "FR4_AA")
no_frameshift_feats = ("FR1_AA", "CDR1_AA", "FR2_AA", "CDR2_AA", "FR3_AA", "CDR3_AA")

#Columns and data types to parse USEARCH tab-separated .uc output files
usearch_col_names = ("Type", "ClusterID", "Len_Size", "Identity_to_Centroid", "Orientation",
                                          "NA1", "NA2", "NA3", "Query_Label", "Centroid_Label")

usearch_col_dtypes = {
        "Type": str,
        "ClusterID": int,
        "Len_Size": int,
        "Identity_to_Centroid": str,
        "Orientation": str,
        "NA1": str,
        "NA2": str,
        "NA3": str,
        "Query_Label": int,
        "Centroid_Label": str
}

MIXCR_header_dtypes = {
        "cloneId": float,
        "descrR1": str,
        "bestVGene": str,
        "bestDGene": str,
        "bestJGene": str,
        "bestCGene": str,
        "vBestIdentityPercent": float,
        "jBestIdentityPercent": float,
        "nSeqFR1": str,
        "nSeqFR2": str,
        "nSeqFR3": str,
        "nSeqFR4": str,
        "nSeqCDR1": str,
        "nSeqCDR2": str,
        "nSeqCDR3": str,
        "aaSeqFR1": str,
        "aaSeqFR2": str,
        "aaSeqFR3": str,
        "aaSeqFR4": str,
        "aaSeqCDR1": str,
        "aaSeqCDR2": str,
        "aaSeqCDR3": str
}

MIXCR_headers_renamed = {
        "cloneId": "CloneID",
        "descrR1": "ReadID",
        "bestVGene": "VGene",
        "bestDGene": "DGene",
        "bestJGene": "JGene",
        "bestCGene": "CGene",
        "vBestIdentityPercent": "V_Identity",
        "dBestIdentityPercent": "D_Identity",
        "jBestIdentityPercent": "J_Identity",
        "nSeqFR1": "FR1_NT",
        "nSeqFR2": "FR2_NT",
        "nSeqFR3": "FR3_NT",
        "nSeqFR4": "FR4_NT",
        "nSeqCDR1": "CDR1_NT",
        "nSeqCDR2": "CDR2_NT",
        "nSeqCDR3": "CDR3_NT",
        "aaSeqFR1": "FR1_AA",
        "aaSeqFR2": "FR2_AA",
        "aaSeqFR3": "FR3_AA",
        "aaSeqFR4": "FR4_AA",
        "aaSeqCDR1": "CDR1_AA",
        "aaSeqCDR2": "CDR2_AA",
        "aaSeqCDR3": "CDR3_AA"
}

#All Homo sapiens heavy/light V and J genes, as recorded by IMGT as of 06-17-2017
human_VH_genes = (
        "IGHV1-12", "IGHV1-14", "IGHV1-17", "IGHV1-18", "IGHV1-2", "IGHV1-24", "IGHV1-3", "IGHV1-
38-4",
        "IGHV1-45", "IGHV1-46", "IGHV1-58", "IGHV1-67", "IGHV1-68", "IGHV1-69", "IGHV1-69-2",
"IGHV1-69D",
        "IGHV1-8", "IGHV2-10", "IGHV2-26", "IGHV2-5", "IGHV2-70", "IGHV2-70D", "IGHV3-11", "IGHV3-
13",
        "IGHV3-15", "IGHV3-16", "IGHV3-19", "IGHV3-20", "IGHV3-21", "IGHV3-22", "IGHV3-23",
"IGHV3-23D",
        "IGHV3-25", "IGHV3-29", "IGHV3-30", "IGHV3-30-3", "IGHV3-30-5", "IGHV3-32", "IGHV3-33",
"IGHV3-33-2",
        "IGHV3-35", "IGHV3-36", "IGHV3-37", "IGHV3-38", "IGHV3-38-3", "IGHV3-41", "IGHV3-42",
"IGHV3-42D",
        "IGHV3-43", "IGHV3-43D", "IGHV3-47", "IGHV3-48", "IGHV3-49", "IGHV3-50", "IGHV3-52",
"IGHV3-53",
        "IGHV3-54", "IGHV3-57", "IGHV3-6", "IGHV3-60", "IGHV3-62", "IGHV3-63", "IGHV3-64", "IGHV3-
64D",
        "IGHV3-65", "IGHV3-66", "IGHV3-69", "IGHV3-7", "IGHV3-71", "IGHV3-72", "IGHV3-73", "IGHV3-
74",
        "IGHV3-75", "IGHV3-76", "IGHV3-79", "IGHV3-9", "IGHV3-NL1", "IGHV4-28", "IGHV4-30-1",
"IGHV4-30-2",
        "IGHV4-30-4", "IGHV4-31", "IGHV4-34", "IGHV4-38-2", "IGHV4-39", "IGHV4-4", "IGHV4-55",
"IGHV4-59",
        "IGHV4-61", "IGHV4-80", "IGHV5-10", "IGHV5-10-1", "IGHV5-51", "IGHV5-78", "IGHV6-1",
"IGHV7-27",
        "IGHV7-40", "IGHV7-40D", "IGHV7-4-1", "IGHV7-56", "IGHV7-77", "IGHV7-81"
)

human_VL_genes = (
        "IGKV1-12", "IGKV1-13", "IGKV1-16", "IGKV1-17", "IGKV1-27", "IGKV1-33", "IGKV1-39",
"IGKV1-5",
        "IGKV1-6", "IGKV1-8", "IGKV1-9", "IGKV1-NL1", "IGKV1D-12", "IGKV1D-13", "IGKV1D-16",
"IGKV1D-17",
        "IGKV1D-33", "IGKV1D-39", "IGKV1D-43", "IGKV1D-8", "IGKV2-24", "IGKV2-28", "IGKV2-29",
"IGKV2-30",
        "IGKV2-40", "IGKV2D-26", "IGKV2D-28", "IGKV2D-29", "IGKV2D-30", "IGKV2D-40", "IGKV3-11",
"IGKV3-15",
        "IGKV3-20", "IGKV3D-11", "IGKV3D-15", "IGKV3D-20", "IGKV3D-7", "IGKV4-1", "IGKV5-2",
"IGKV6-21",
        "IGKV6D-21", "IGLV1-36", "IGLV1-40", "IGLV1-44", "IGLV1-47", "IGLV1-51", "IGLV10-54",
"IGLV2-11",
        "IGLV2-14", "IGLV2-18", "IGLV2-23", "IGLV2-8", "IGLV3-1", "IGLV3-10", "IGLV3-12", "IGLV3-
16", "IGLV3-19",
        "IGLV3-21", "IGLV3-22", "IGLV3-25", "IGLV3-27", "IGLV3-9", "IGLV4-3", "IGLV4-60", "IGLV4-
69", "IGLV5-37",
        "IGLV5-39", "IGLV5-45", "IGLV5-52", "IGLV6-57", "IGLV7-43", "IGLV7-46", "IGLV8-61",
"IGLV9-49"
)

human_JH_genes = ("IGHJ1", "IGHJ2", "IGHJ3", "IGHJ4", "IGHJ5", "IGHJ6")

human_JL_genes = ("IGKJ1", "IGKJ2", "IGKJ3", "IGKJ4", "IGKJ5", "IGLJ1", "IGLJ2", "IGLJ3", "IGLJ6",
"IGLJ7")

#Gene names to (int, int) indicating its relative start and end position in the human IGH locus
(see NCBI NG_001019)
human_VH_gene_locations = {
        "IGHVIII-82": (501, 792), "IGHV7-81": (5328, 5764), "IGHV4-80": (7159, 7562), "IGHV3-79":
(12253, 12705),
        "IGHVII-78-1": (14450, 14724), "IGHV5-78": (28799, 29233), "IGHVIII-76-1": (48276, 48582),
        "IGHV3-76": (52035, 52484), "IGHV3-75": (56198, 56667), "IGHVII-74-1": (58935, 59103),
"IGHV3-74": (69450, 69905),
        "IGHV3-73": (77192, 77653), "IGHV3-72": (89192, 89653), "IGHV3-71": (104727, 105188),
"IGHV2-70": (109325, 109768),
        "IGHV1-69D": (117815, 118253), "IGHV1-69-2": (142798, 143235), "IGHV3-69-1": (151730,
152182),
        "IGHV2-70D": (156328, 156771), "IGHV1-69": (165225, 165663), "IGHV1-68": (176059, 176538),
        "IGHVIII-67-4": (184948, 185253), "IGHVIII-67-3": (187457, 187731), "IGHVIII-67-2":
(193145, 193243),
        "IGHVII-67-1": (193803, 193952), "IGHV1-67": (199303, 199742), "IGHV3-66": (204880,
205330),
        "IGHVII-65-1": (208301, 208573), "IGHV3-65": (213813, 214307), "IGHV3-64": (222167,
222622),
        "IGHV3-63": (227665, 228132), "IGHVII-62-1": (229557, 229829), "IGHV3-62": (236760,
237215),
        "IGHV4-61": (240789, 241226), "IGHVII-60-1": (242372, 242640), "IGHV3-60": (248692,
249148),
        "IGHV4-59": (252665, 253096), "IGHV1-58": (257551, 257988), "IGHV3-57": (261131, 261436),
        "IGHV7-56": (270149, 270583), "IGHV4-55": (273794, 274229), "IGHV3-54": (278553, 279009),
        "IGHVII-53-1": (280421, 280690), "IGHV3-53": (287219, 287669), "IGHV3-52": (293519,
293969),
        "IGHVII-51-2": (295369, 295631), "IGHVIII-51-1": (296538, 296844), "IGHV5-51": (301168,
301603),
        "IGHV3-50": (313790, 314245), "IGHVII-49-1": (315747, 316017), "IGHV3-49": (322948,
323409),
        "IGHV3-48": (342080, 342535), "IGHVIII-47-1": (348909, 349212), "IGHV3-47": (361311,
361766),
        "IGHVII-46-1": (364221, 364509), "IGHV1-46": (368793, 369230), "IGHV1-45": (372912,
373349),
        "IGHVII-44-2": (385964, 386200), "IGHVIV-44-1": (390575, 391003), "IGHVIII-44": (401952,
402128),
        "IGHVII-43-1": (407345, 407554), "IGHV3-43": (409625, 410082), "IGHV3-42": (416657,
417091),
        "IGHV3-41": (436761, 437216), "IGHVII-40-1": (439335, 439411), "IGHV7-40": (454784,
454986),
        "IGHV4-39": (458198, 458636), "IGHVIII-38-1": (462035, 462327), "IGHV3-38": (469402,
469851),
        "IGHV3-37": (483231, 483680), "IGHV3-36": (487114, 487576), "IGHV3-35": (490502, 490955),
        "IGHV7-34-1": (502612, 503045), "IGHV4-34": (506252, 506684), "IGHV3-33-2": (510799,
511247),
        "IGHVII-33-1": (512681, 512954), "IGHV3-33": (520101, 520554), "IGHV3-32": (523743,
524200),
        "IGHVII-30-21": (525641, 525897), "IGHV4-30-2": (530625, 531062), "IGHV3-30-2": (535512,
535960),
        "IGHVII-30-1": (537392, 537665), "IGHV3-30": (544812, 545265), "IGHV3-29": (548782,
549239),
        "IGHVII-28-1": (550659, 550913), "IGHV4-28": (555657, 556091), "IGHV7-27": (562109,
562522),
        "IGHVII-26-2": (565385, 565697), "IGHVIII-26-1": (570633, 570939), "IGHV2-26": (578507,
578950),
        "IGHVIII-25-1": (586538, 586779), "IGHV3-25": (590866, 591319), "IGHV1-24": (603362,
603799),
        "IGHV3-23": (611284, 611739), "IGHVIII-22-2": (615472, 615659), "IGHVII-22-1": (616717,
616985),
        "IGHV3-22": (622122, 622583), "IGHV3-21": (644830, 645283), "IGHVII-20-1": (667521,
667559),
        "IGHV3-20": (668954, 669409), "IGHV3-19": (683355, 683647), "IGHV1-18": (695010, 695446),
        "IGHV1-17": (705567, 706003), "IGHVIII-16-1": (709294, 709599), "IGHV3-16": (714689,
715142),
        "IGHVII-15-1": (716541, 716811), "IGHV3-15": (726262, 726723), "IGHV1-14": (734214,
734463),
        "IGHVIII-13-1": (737761, 738059), "IGHV3-13": (750353, 750805), "IGHV1-12": (757636,
757925),
        "IGHVIII-11-1": (759870, 760056), "IGHV3-11": (763261, 763710), "IGHV5-10-1": (771832,
772373),
        "IGHV3-64D": (791772, 792262), "IGHV3-7": (817741, 818196), "IGHV3-6": (824347, 824801),
        "IGHVIII-5-2": (834485, 834745), "IGHVIII-5-1": (840581, 840679), "IGHV2-5": (842000,
842443),
        "IGHV7-4-1": (854764, 855200), "IGHV4-4": (867989, 868423), "IGHV1-3": (874813, 875250),
        "IGHVIII-2-1": (878510, 878811), "IGHV1-2": (893326, 893763), "IGHVII-1-1": (934954,
935407),
        "IGHV6-1": (940144, 940591)
}

human_JH_gene_locations = {
        "IGHJ1": (1014887, 1014940), "IGHJ2": (1015094, 1015148), "IGHJ2P": (1015493, 1015552),
"IGHJ3": (1015709, 1015760),
        "IGHJ4": (1016083, 1016132), "IGHJ5": (1016481, 1016533), "IGHJ3P": (1016880, 1016930),
"IGHJ6": (1017085, 1017149)
}

#Regex for potentially N-linked glycosylation sites with the mammalian consensus sequence: Asn +
NOT Pro + Ser or Thr
N_linked_gly_consensus = "N[^P][ST]"


#########Repertoire Graphing / Visualization Dashboard
import numpy
import json
import pandas
import math
from scipy.stats import gaussian_kde
from squarify import squarify
from itertools import cycle, combinations

from bokeh.plotting import figure
from bokeh.models import (Range1d, HoverTool, ColumnDataSource, CustomJS, ColorBar,
LinearColorMapper,
                                                    NumeralTickFormatter, FixedTicker,
BasicTickFormatter)
from bokeh.models.widgets import Select
from bokeh.colors import RGB
from bokeh.palettes import viridis, all_palettes
from bokeh.io import save, show, output_file
from bokeh.io.export import export_png
from bokeh.embed import components
from bokeh.layouts import column, layout, gridplot, Spacer
from bokeh.transform import linear_cmap
from bokeh.util.hex import hexbin

vgene_colors = {
        "IGHV1-17": RGB(255, 0, 0),
        "IGHV1-18": RGB(255, 24, 0),
        "IGHV1-2": RGB(255, 47, 0),
        "IGHV1-24": RGB(255, 71, 0),
        "IGHV1-3": RGB(255, 95, 0),
        "IGHV1-45": RGB(255, 118, 0),
        "IGHV1-46": RGB(255, 142, 0),
        "IGHV1-58": RGB(255, 165, 0),
        "IGHV1-67": RGB(255, 195, 0),
        "IGHV1-68": RGB(255, 219, 0),
        "IGHV1-69": RGB(254, 241, 0),
        "IGHV1-8": RGB(244, 255, 0),
        "IGHV2-10": RGB(221, 255, 0),
        "IGHV2-26": RGB(197, 255, 0),
        "IGHV2-5": RGB(173, 255, 0),
        "IGHV2-70": RGB(150, 255, 0),
        "IGHV3-11": RGB(120, 255, 0),
        "IGHV3-13": RGB(97, 255, 0),
        "IGHV3-15": RGB(73, 255, 0),
        "IGHV3-16": RGB(49, 255, 0),
        "IGHV3-19": RGB(26, 255, 0),
        "IGHV3-20": RGB(6, 255, 4),
        "IGHV3-21": RGB(0, 255, 22),
        "IGHV3-22": RGB(0, 255, 45),
        "IGHV3-23": RGB(0, 255, 75),
        "IGHV3-25": RGB(0, 255, 98),
        "IGHV3-30": RGB(0, 255, 122),
        "IGHV3-33": RGB(0, 255, 146),
        "IGHV3-35": RGB(0, 255, 169),
        "IGHV3-36": RGB(0, 255, 193),
        "IGHV3-38": RGB(0, 255, 217),
        "IGHV3-43": RGB(0, 255, 246),
        "IGHV3-47": RGB(0, 240, 255),
        "IGHV3-48": RGB(0, 217, 255),
        "IGHV3-49": RGB(0, 193, 255),
        "IGHV3-52": RGB(0, 169, 255),
        "IGHV3-53": RGB(0, 146, 255),
        "IGHV3-60": RGB(0, 122, 255),
        "IGHV3-62": RGB(0, 99, 255),
        "IGHV3-64": RGB(0, 69, 255),
        "IGHV3-65": RGB(0, 45, 255),
        "IGHV3-66": RGB(0, 22, 255),
        "IGHV3-7": RGB(6, 4, 255),
        "IGHV3-71": RGB(25, 0, 255),
        "IGHV3-72": RGB(49, 0, 255),
        "IGHV3-73": RGB(73, 0, 255),
        "IGHV3-74": RGB(96, 0, 255),
        "IGHV3-76": RGB(126, 0, 255),
        "IGHV3-9": RGB(149, 0, 255),
        "IGHV4-28": RGB(173, 0, 255),
        "IGHV4-31": RGB(197, 0, 255),
        "IGHV4-34": RGB(220, 0, 255),
        "IGHV4-39": RGB(244, 0, 255),
        "IGHV4-4": RGB(254, 0, 241),
        "IGHV4-55": RGB(255, 0, 219),
        "IGHV4-59": RGB(255, 0, 189),
        "IGHV4-61": RGB(255, 0, 166),
        "IGHV5-51": RGB(255, 0, 142),
        "IGHV5-78": RGB(255, 0, 118),
        "IGHV6-1": RGB(255, 0, 95),
        "IGHV7-27": RGB(255, 0, 71),
        "IGHV7-81": RGB(255, 0, 47)
}

vfamily_colors = {
        "IGHV1": RGB(254, 131, 0),
        "IGHV2": RGB(185, 255, 0),
        "IGHV3": RGB(27, 164, 172),
        "IGHV4": RGB(232, 0, 229),
        "IGHV5": RGB(150, 0, 210),
        "IGHV6": RGB(255, 0, 95),
        "IGHV7": RGB(35, 140, 20)
}

jgene_colors = {
        "IGHJ1": RGB(57, 59, 121),
        "IGHJ2": RGB(82, 84, 163),
        "IGHJ3": RGB(107, 110, 207),
        "IGHJ4": RGB(156, 158, 222),
        "IGHJ5": RGB(99, 121, 57),
        "IGHJ6": RGB(140, 162, 82),
        "IGHJ2P": RGB(181, 207, 107)
}

isotype_colors = {
        "IgG": RGB(55, 126, 184),
        "IgG1": RGB(55, 126, 184),
        "IgG2": RGB(55, 126, 184),
        "IgG3": RGB(55, 126, 184),
        "IgG4": RGB(55, 126, 184),
        "IGHG": RGB(55, 126, 184),
        "IGHG1": RGB(55, 126, 184),
        "IGHG2": RGB(55, 126, 184),
        "IGHG3": RGB(55, 126, 184),
        "IGHG4": RGB(55, 126, 184),
        "IgA": RGB(228, 26, 28),
        "IgA1": RGB(228, 26, 28),
        "IgA2": RGB(228, 26, 28),
        "IGHA": RGB(228, 26, 28),
        "IGHA1": RGB(228, 26, 28),
        "IGHA2": RGB(228, 26, 28),
        "IgM": RGB(77, 175, 74),
        "IGHM": RGB(77, 175, 74),
        "IgD": RGB(152, 78, 163),
        "IGHD": RGB(152, 78, 163),
        "IgE": RGB(255, 127, 0),
        "IGHE": RGB(255, 127, 0)
```

```
}

def Shannon_Wiener_Index(clone_counts):
        total_clone_counts = float(sum(clone_counts))
        sw_index = 0.0
        for clone in clone_counts:
                clone_freq = clone / total_clone_counts
                sw_index -= (clone_freq * numpy.log(clone_freq))
        return sw_index

def Hill_Diversity_Index(clone_counts, N = (0.0, 10.0), step = 0.1):
        if hasattr(N, "__iter__"):
                if len(N) != 2 or N[1] <= N[0]:
                        print("Error in Hill_Diversity_Index: if N is an iterable it must be of
length 2 for the start/end orders.")
                        return None
                chunks = numpy.floor(N[1] / step) + 1
                orders = numpy.linspace(start = N[0], stop = N[1], num = chunks)
        else:
                orders = [N]
        total_clone_counts = float(sum(clone_counts))
        hill_indices = []
        for order in orders:
                hill_index = 0.0
                if order == 0.0:
                        hill_index = len(clone_counts) #Hill index at zero is simply the species
richness (total number of clones)
                elif order == 1.0:
                        sw_index = Shannon_Wiener_Index(clone_counts)
                        hill_index = numpy.exp(sw_index) #Hill index at one is the exponential of
the Shannon-Wiener index
                else:
                        for clone in clone_counts:
                                clone_freq = clone / total_clone_counts
                                hill_index += (clone_freq ** order)
                        order_exponent = 1.0 / (1.0 - order)
                        hill_index = hill_index ** order_exponent
                order_index = (order, hill_index)
                hill_indices.append(order_index)
        if len(hill_indices) == 1:
                return hill_indices[0]
        else:
                return hill_indices

class Cyrcos_Repertoire_Comparison_Plot(object):
        def __init__(self, clone_dfs, title = "", top_clones = None, normalize_segments = True,
gap_size = 10,
                                 start_pos = "top", clockwise = True, offset_segments = None,
segment_face_colors = "Category10",
                                 segment_outline_colors = None, fade_segments = True, clone_col =
"CloneID", count_col = "Clustered",
                                 sample_col = "Sample", figsize = (1000, 1000)):
                """Creates a Circos-like Chord graph for comparing multiple immune repertoire
clonotype profiles."""

                #Plot visual aspect definitions
                segment_width = 0.07 #Thickness of the circle arc segments
                offset_shift_amount = 0.1 #Increase in radius for segments to offset
                min_segment_alpha = 0.2

                #Gather the samples and their respective DataFrames
                df_cols = [clone_col, count_col]
                self.samples = []
                comparison_dfs = []
```

```
		if isinstance(clone_dfs, dict): #Input is a dictionary of {sample_name: 
DataFrame}
			for sample in clone_dfs:
				self.samples.append(sample)
				clone_df = clone_dfs[sample][df_cols].sort_values([count_col], 
ascending = [False])

				if top_clones is not None:
					clone_df = clone_df.head(top_clones)

				comparison_dfs.append(clone_df)

		else: #Input is a single DataFrame with all samples; sample_col gives the sample 
to split on
			df_cols.append(sample_col)
			for sample, df in clone_dfs[df_cols].groupby([sample_col]):
				self.samples.append(sample)
				clone_df = df.sort_values([count_col], ascending = [False])

				if top_clones is not None:
					clone_df = clone_df.head(top_clones)

				comparison_dfs.append(clone_df)

		self.total_samples = len(self.samples)

		#Create the figure plot
		self.Create_Plot(title = title, figsize = figsize)

		sample_clone_counts = [len(df) for df in comparison_dfs]
		total_gap_size = gap_size * self.total_samples
		total_segment_len = 360 - total_gap_size

		if normalize_segments:
			self.segment_lengths = [total_segment_len / self.total_samples] * 
self.total_samples
		else:
			total_clones = sum(sample_clone_counts)
			self.segment_lengths = [clones / total_clones * total_segment_len for 
clones in sample_clone_counts]

		#Convert description of the first segment's start location to the angular 
position in degrees:
		start_pos = start_pos.lower()
		if "top" in start_pos or "north" in start_pos:
			start_position = 90
		elif "right" in start_pos or "east" in start_pos:
			start_position = 0
		elif "bottom" in start_pos or "south" in start_pos:
			start_position = -90
		elif "left" in start_pos or "west" in start_pos:
			start_position = -180
		else:
			start_position = 90

		#Shift the start position to align the first gap's center with the start 
position.
		self.direction = "clock" if clockwise else "anticlock"
		start_position -= gap_size / 2 if clockwise else 0

		radius = 0.8
		self.inner_radii = [radius] * self.total_samples

		if offset_segments is not None:
			if isinstance(offset_segments, int):
```

```
                                self.inner_radii[offset_segments] += offset_shift_amount

                        elif hasattr(offset_segments, "__iter__") and not
isinstance(offset_segments, str):
                                for seg in offset_segments:
                                        self.inner_radii[seg] += offset_shift_amount

                self.outer_radii = [r + segment_width for r in self.inner_radii]

                if isinstance(segment_face_colors, str):
                        if segment_face_colors in all_palettes:
                                self.segment_face_colors =
all_palettes[segment_face_colors][self.total_samples]
                        else:
                                self.segment_face_colors = [segment_face_colors] *
self.total_samples
                elif hasattr(segment_face_colors, "__iter__") and not
isinstance(segment_face_colors, str):
                        if len(segment_face_colors) == self.total_samples:
                                self.segment_face_colors = segment_face_colors
                        else:
                                raise IndexError("List provided to segment_face_colors is not the
same length as total segments!")
                else:
                        print("Warning: segment_face_colors should be a colormap or list of
colors for each segment!")
                        print("Defaulting to \"Category10\"...")
                        self.segment_face_colors = all_palettes["Category10"][self.total_samples]

                if segment_outline_colors is None:
                        self.segment_outline_colors = ["transparent"] * self.total_samples
                elif isinstance(segment_outline_colors, str):
                        self.segment_outline_colors = [segment_outline_colors] *
self.total_samples
                elif hasattr(segment_outline_colors, "__iter__") and not
isinstance(segment_outline_colors, str):
                        self.segment_outline_colors = segment_outline_colors
                else:
                        raise TypeError("segment_outline_colors should be a color name, list of
colors, or None!")

                #Add the repertoire circle segments to the figure
                self.Create_Segments(start_position, gap_size, clockwise, fade_segments,
min_alpha = min_segment_alpha)

                #Add rank and segment position columns to the clone DataFrames
                for idx, _ in enumerate(comparison_dfs):
                        #Rank clones by total count / frequency (largest clone being rank 0,
second largest rank 1, etc.)
                        clone_ranks = comparison_dfs[idx][count_col].rank(method = "first",
ascending = False).astype(int)
                        comparison_dfs[idx]["Rank"] = clone_ranks - 1

                        #Convert the rank to a relative position from 0.0 to 1.0 for placement
along the segments
                        comparison_dfs[idx]["Position"] = comparison_dfs[idx]["Rank"] /
len(comparison_dfs[idx])

                #Iterate through all combinations of two samples and create the links from clone
to clone
                for comb in combinations(range(self.total_samples), 2):
                        idx1 = comb[0]
                        idx2 = comb[1]
                        sample1 = self.samples[idx1]
                        sample2 = self.samples[idx2]
                        sample_df1 = comparison_dfs[idx1].set_index([clone_col])
                        sample_df2 = comparison_dfs[idx2].set_index([clone_col])

                        #Join the samples into a DataFrame containing only the shared clones and
their positions
                        joined_df = sample_df1.join(sample_df2, how = "inner", lsuffix = "1",
rsuffix = "2")

                        #Calculate the angular position for the clones (segment start location +
clone position * segment length)
                        #If the plot is drawn clockwise, subtract the segment start instead of
adding it
                        seg1_start = -self.segment_starts[idx1] if self.direction == "clock" else
self.segment_starts[idx1]
                        seg2_start = -self.segment_starts[idx2] if self.direction == "clock" else
self.segment_starts[idx2]
                        pos1 = joined_df["Position1"] * self.segment_lengths[idx1] + seg1_start
                        pos2 = joined_df["Position2"] * self.segment_lengths[idx2] + seg2_start

                        #Convert the positions to the start and end xy coordinates
                        inner_rad1 = self.inner_radii[idx1]
                        inner_rad2 = self.inner_radii[idx2]
                        xy1 = self.Angle_to_XY(angles = pos1, radius = inner_rad1)
                        xy2 = self.Angle_to_XY(angles = pos2, radius = inner_rad2)
                        #Convert the list of (x, y) tuples to a list of xs and a list of ys
                        xs1, ys1 = zip(*xy1)
                        xs2, ys2 = zip(*xy2)

                        link_data = {
                                "x0": xs1,
                                "y0": ys1,
                                "x1": xs2,
                                "y1": ys2
                        }
                        link_source = ColumnDataSource(link_data)

                        #Plot the links matching clone positions between repertoires; control
points are the center of the circle
                        self.plot.quadratic(x0 = "x0", y0 = "y0", x1 = "x1", y1 = "y1", cx = 0.5,
cy = 0.5, source = link_source,
                                                                        color = "black", line_width = 1)

        def Create_Plot(self, title, figsize):
                plot_params = {
                        "plot_width": figsize[0],
                        "plot_height": figsize[1],
                        "x_range": Range1d(-0.5, 1.5, bounds = (-1.0, 2.0)),
                        "y_range": Range1d(-0.5, 1.5, bounds = (-1.0, 2.0)),
                        "title": title,
                        "tools": "pan, wheel_zoom, box_zoom, save, reset, help",
                        "active_scroll": "wheel_zoom",
                        "toolbar_location": "right",
                        "outline_line_alpha": 0.0
                }
                self.plot = figure(**plot_params)
                self.plot.grid.visible = False
                self.plot.axis.visible = False

        def Create_Segments(self, start_position, gap_size, clockwise, fade_segments, fade_steps =
1000, min_alpha = 0.01):
                self.segment_starts = []
                self.segment_ends = []

                if clockwise:
                        segment_deltas = [-seg_len for seg_len in self.segment_lengths]
```

```
				gap_delta = -gap_size
			else:
				segment_deltas = [seg_len for seg_len in self.segment_lengths]
				gap_delta = gap_size

			cur_position = start_position

			segment_border_starts = []
			segment_border_ends = []

			for seg_len in segment_deltas:
				segment_border_starts.append(cur_position)
				cur_position += seg_len
				segment_border_ends.append(cur_position)
				cur_position += gap_delta

			border_source_dict = {
				"start_angle": segment_border_starts,
				"end_angle": segment_border_ends,
				"inner_radius": self.inner_radii,
				"outer_radius": self.outer_radii,
				"line_color": self.segment_outline_colors
			}
			border_data = ColumnDataSource(data = border_source_dict)

			self.borders = self.plot.annular_wedge(x = 0.5, y = 0.5, start_angle = 
"start_angle", end_angle = "end_angle",

inner_radius = "inner_radius", outer_radius = "outer_radius",

fill_color = None, line_color = "line_color", line_width = 1,

direction = self.direction, start_angle_units = "deg",

end_angle_units = "deg", source = border_data)

			cur_position = start_position
			cur_seg_starts = []
			cur_seg_ends = []

			if fade_segments:
				self.segment_alphas = []

				#Extend the radius and fill color lists to account for the extra alpha 
segments:
				inner_seg_radii = [r for r in self.inner_radii for _ in 
range(fade_steps)]
				outer_seg_radii = [r for r in self.outer_radii for _ in 
range(fade_steps)]
				self.segment_face_colors = [seg_color for seg_color in 
self.segment_face_colors for _ in range(fade_steps)]

				for segment_delta in segment_deltas:
					alpha_segment_delta = segment_delta / fade_steps
					alpha_delta = (1.0 - min_alpha) / fade_steps
					cur_alpha = 1.0

					self.segment_starts.append(cur_position)

					for _ in range(fade_steps):
						cur_seg_starts.append(cur_position)
						cur_position += alpha_segment_delta
						cur_seg_ends.append(cur_position)

						self.segment_alphas.append(cur_alpha)
```

```
                                        cur_alpha -= alpha_delta

                                self.segment_ends.append(cur_position)
                                cur_position += gap_delta

                        cur_legend = [label for label in self.samples for _ in range(fade_steps)]

                else:
                        self.segment_alphas = [1.0] * self.total_samples
                        inner_seg_radii = self.inner_radii
                        outer_seg_radii = self.outer_radii

                        for segment_delta in segment_deltas:
                                self.segment_starts.append(cur_position)
                                cur_seg_starts.append(cur_position)

                                cur_position += segment_delta
                                self.segment_ends.append(cur_position)
                                cur_seg_ends.append(cur_position)

                                cur_position += gap_delta

                        cur_legend = self.samples

                source_data_dict = {
                        "start_angle": cur_seg_starts,
                        "end_angle": cur_seg_ends,
                        "inner_radius": inner_seg_radii,
                        "outer_radius": outer_seg_radii,
                        "fill_color": self.segment_face_colors,
                        "fill_alpha": self.segment_alphas,
                        "legend": cur_legend
                }
                source_data = ColumnDataSource(data = source_data_dict)

                self.wedges = self.plot.annular_wedge(x = 0.5, y = 0.5, direction =
self.direction, start_angle = "start_angle",

end_angle = "end_angle", inner_radius = "inner_radius",

outer_radius = "outer_radius", fill_color = "fill_color",

fill_alpha = "fill_alpha", line_color = None, start_angle_units = "deg",

end_angle_units = "deg", legend = "legend", source = source_data)

        def Angle_to_XY(self, angles, radius, angles_in_degrees = True, offset = (0.5, 0.5)):
                """Converts an angular position to X, Y coordinates.

                Parameters
                ----------
                angles: int/float or iterable of int/float
                        Angle(s) to convert to X, Y coordinate(s).
                radius: int/float
                        Radius of the circle for which the X, Y coordinates map to.
                angles_in_degrees: bool
                        Whether the provided angle(s) are in degrees or radians; default is True.
                offset: tuple of (float, float)
                        The x and y location of the center of the circle; default is (0.5, 0.5).

                Returns
                ----------
                xy_coords: tuple of (float, float) or list of tuples (float, float)
                        X and Y coordinate(s) of the angles provided on a circle with provided
radius.
```

```
                """

                if angles_in_degrees:
                        x = radius * numpy.sin(numpy.deg2rad(angles)) + offset[0]
                        y = radius * numpy.cos(numpy.deg2rad(angles)) + offset[1]
                else:
                        x = radius * numpy.sin(angles) + offset[0]
                        y = radius * numpy.cos(angles) + offset[1]

                #Return a tuple of (x, y) if a single angle was provided, or a list of (x, y)
tuples if multiple angles
                if hasattr(x, "__iter__") and hasattr(y, "__iter__"):
                        xy_coords = list(zip(x, y))
                else:
                        xy_coords = (x, y)

                return xy_coords

        def Show(self):
                """Call Bokeh show function to display the current plot in a browser window."""

                show(self.plot)

        def Get_Plot_Components(self):
                """Call Bokeh components function to get the HTML script and div elements
representing the current plot.

                Returns
                ----------
                plot_script: str
                        The HTML code for the Bokeh JavaScript plot to display.
                plot_div: str
                        The HTML code for the div element used to place the plot in a page.
                """

                plot_script, plot_div = components(self.plot)
                return plot_script, plot_div

        def Save_HTML(self, filename, title = "Repertoire Comparison Cyrcos Graph"):
                """Calls Bokeh save function to save an HTML file containing the current plot.

                Parameters
                ----------
                filename: str
                        The desired filename / filepath for the output HTML file.
                title: str
                        A title to use for the HTML file; default is "Repertoire Comparison
Cyrcos Graph".
                """

                save(self.plot, filename = filename, title = title)

class Repertoire_Upset_Plot(object):
        def __init__(self, clone_dfs, title = "", min_shared = 2, max_shared = None,
overlap_bounds = None,
                                 clone_col = "CloneID", sample_col = None, highlighted_sets =
None, figsize = (1200, 900)):
                """Creates a Repertoire comparison UpSet overlap plot."""

                df_cols = [clone_col]

                if isinstance(clone_dfs, dict):
                        df_dict = {sample: clone_dfs[sample][df_cols] for sample in clone_dfs}
                        comparison_df = pandas.concat(list(df_dict.values()), ignore_index =
True)
                        samples = [i for i in df_dict]

                else:
                        df_cols.append(sample_col)

                        comparison_df = clone_dfs[df_cols]
                        samples = clone_dfs[sample_col].drop_duplicates().tolist()

                #Collapse by clone IDs and concatenate the multiple repertoire sample names
containing each clone using JSON
                serializer = lambda x: json.dumps(x.tolist())
                grouped_df = comparison_df.groupby([clone_col])[sample_col].agg({serializer,
"count"})
                grouped_df = grouped_df.rename(columns = {"<lambda>": "Samples", "count":
"Sample_Count"})

                if min_shared is not None:
                        grouped_df = grouped_df[grouped_df["Sample_Count"] >= min_shared]
                if max_shared is not None:
                        grouped_df = grouped_df[grouped_df["Sample_Count"] <= max_shared]

                #Calculate the total number of clones shared by each combination of samples
                overlap_counts = grouped_df["Samples"].value_counts()
                if overlap_bounds is not None:
                        overlap_counts = overlap_counts[overlap_counts >= overlap_bounds[0]]
                        overlap_counts = overlap_counts[overlap_counts <= overlap_bounds[1]]

                total_sets = len(overlap_counts)
                total_samples = len(samples)
                MAIN_BAR_WIDTH = 0.5
                set_colors = ("#82C882", "#BEB4D2", "#FABE82", "#FFFF96", "#326EB4", "#F00082",
"#BE5A14", "#646464")

                main_plot_params = {
                        "plot_width": int(figsize[0] * 0.6),
                        "plot_height": int(figsize[1] * 0.75),
                        "x_range": Range1d(-MAIN_BAR_WIDTH, total_sets - 1 + MAIN_BAR_WIDTH),
                        "title": title,
                        "tools": "save, reset, help",
                        "outline_line_alpha": 0.0
                }
                self.main_plot = figure(**main_plot_params)
                self.main_plot.grid.visible = False
                self.main_plot.xaxis.visible = False
                largest_overlap = overlap_counts.max()
                self.main_plot.yaxis.bounds = (0, largest_overlap)
                self.main_plot.yaxis.axis_label_text_font_size = "12pt"

                sample_sets_xs = [i for i in range(total_sets)]
                default_bar_color = "#96AAC8"
                sample_sets_colors = []
                cur_color_idx = 0
                cur_color = default_bar_color

                if highlighted_sets is not None:
                        for main_bar_set in overlap_counts.index:
                                for highlighted_subset in highlighted_sets:
                                        cur_set = json.loads(main_bar_set)
                                        if set(cur_set) == set(highlighted_subset):
                                                cur_color = set_colors[cur_color_idx]
                                                cur_color_idx += 1
                                                break
                                        else:
```

```
                                        cur_color = default_bar_color
                                sample_sets_colors.append(cur_color)
                else:
                        sample_sets_colors += [default_bar_color] * len(overlap_counts)

                main_bars_data = {
                        "x": sample_sets_xs,
                        "top": overlap_counts.tolist(),
                        "color": sample_sets_colors
                }
                main_bars_source = ColumnDataSource(main_bars_data)
                self.main_bars = self.main_plot.vbar(x = "x", top = "top", width = 
MAIN_BAR_WIDTH, bottom = 0,
                                                                                  color = 
"color", source = main_bars_source)
                self.main_plot.yaxis.axis_label = "Total Shared Clones"

                sample_sets_plot_params = {
                        "plot_width": int(figsize[0] * 0.6),
                        "plot_height": int(figsize[1] * 0.25),
                        "x_range": self.main_plot.x_range,
                        "y_range": Range1d(-MAIN_BAR_WIDTH, total_samples - 1 + MAIN_BAR_WIDTH, 
bounds = (0, total_samples)),
                        "tools": "save, reset, help",
                        "background_fill_color": "#C8C896",
                        "background_fill_alpha": 0.1,
                        "outline_line_alpha": 0.0
                }
                self.sample_sets_plot = figure(**sample_sets_plot_params)
                self.sample_sets_plot.grid.visible = False
                self.sample_sets_plot.xaxis.visible = False
                self.sample_sets_plot.yaxis.axis_label = " "  #Helps keep plot at the same width 
as the main plot dimensions
                self.sample_sets_plot.yaxis.axis_label_text_font_size = "12pt"

                #Used to pad the Y axis to align properly with the main bar graph
                self.sample_sets_plot.yaxis[0].ticker = [i for i in range(total_samples)]
                self.sample_sets_plot.yaxis.major_label_overrides = {"0": str(largest_overlap)}

                #"Draw" invisible axis line / labels / ticks to match main plot width
                self.sample_sets_plot.yaxis.axis_label_text_color = None
                self.sample_sets_plot.yaxis.axis_line_color = None
                self.sample_sets_plot.yaxis.major_tick_line_color = None
                self.sample_sets_plot.yaxis.major_label_text_color = None

                sample_set_circle_radius = MAIN_BAR_WIDTH * 0.3

                sample_sets_data = {
                        "x": sample_sets_xs * total_samples,
                        "y": [i for i in range(total_samples) for _ in range(total_sets)]
                }
                sample_sets_source = ColumnDataSource(sample_sets_data)
                self.sample_sets_plot.circle(x = "x", y = "y", radius = sample_set_circle_radius, 
fill_color = "#787878",
                                                                  line_color = None, 
alpha = 0.5, source = sample_sets_source)

                #Get the total clones per sample for the total clone counts bar graph
                clone_counts = comparison_df[sample_col].value_counts()

                #Create the linked circles that mark the compared samples
                ypos_to_sample = clone_counts.reset_index()["index"].to_dict()
                sample_to_ypos = {ypos_to_sample[key]: key for key in ypos_to_sample}
                cur_set_color = "black"
                cur_color_idx = 0
```

```
		for x_pos, cur_set in enumerate(overlap_counts.index):
			sample_set = json.loads(cur_set)
			cur_set_count = len(sample_set)
			set_ys = [sample_to_ypos[sample] for sample in sample_set]

			if highlighted_sets is not None:
				for subset in highlighted_sets:
					if set(sample_set) == set(subset):
						cur_set_color = set_colors[cur_color_idx]
						cur_color_idx += 1
						break
					else:
						cur_set_color = "black"

			#Draw the bars linking the sample set circles
			min_circle_y = min(set_ys)
			max_circle_y = max(set_ys)
			self.sample_sets_plot.line(x = [x_pos, x_pos], y = [min_circle_y,
max_circle_y],
											     line_width = 5,
line_color = cur_set_color)
			#Draw the sample set circles
			self.sample_sets_plot.circle(x = [x_pos] * cur_set_count, y = set_ys,
radius = sample_set_circle_radius,
												fill_color =
cur_set_color, line_color = None)

		largest_repertoire_clones = clone_counts.max()
		clone_bar_plot_params = {
			"plot_width": int(figsize[0] * 0.4),
			"plot_height": int(figsize[1] * 0.25),
			"x_range": Range1d(largest_repertoire_clones, 0),
			"tools": "save, reset, help",
			"outline_line_alpha": 0.0,
			"y_axis_location": "right"
		}
		self.clone_bar_plot = figure(**clone_bar_plot_params)
		self.clone_bar_plot.grid.visible = False
		self.clone_bar_plot.xaxis.axis_label_text_font_size = "12pt"
		self.clone_bar_plot.yaxis[0].ticker = [i for i in range(total_samples)]
		sample_ticks_to_labels = {tick_y: sample for tick_y, sample in
enumerate(clone_counts.index.tolist())}
		self.clone_bar_plot.yaxis.major_label_overrides = sample_ticks_to_labels
		self.clone_bar_plot.yaxis.major_tick_line_color = None
		self.clone_bar_plot.yaxis.major_label_text_baseline = "middle"
		self.clone_bar_plot.yaxis.major_label_text_font_size = "12pt"

		clone_bars_data = {
			"y": [i for i in range(total_samples)],
			"right": clone_counts.tolist()
		}
		clone_bars_source = ColumnDataSource(clone_bars_data)
		self.clone_bars = self.clone_bar_plot.hbar(y = "y", right = "right", height =
0.5, left = 0,

color = "#96AAC8", source = clone_bars_source)
		self.clone_bar_plot.xaxis.axis_label = "Total Clones"

		top_left_spacer = Spacer(width = int(figsize[0] * 0.4), height = int(figsize[1] *
0.75))
		self.plots_grid = gridplot([top_left_spacer, self.main_plot, self.clone_bar_plot,
self.sample_sets_plot],
									    ncols = 2, tools = "save,
reset, help", toolbar_location = "right")
```

```
	def Show(self):
		"""Call Bokeh show function to display the current plot in a browser window."""

		show(self.plots_grid)

	def Get_Plot_Components(self):
		"""Call Bokeh components function to get the HTML script and div elements 
representing the current plot.

		Returns
		----------
		plot_script: str
			The HTML code for the Bokeh JavaScript plot to display.
		plot_div: str
			The HTML code for the div element used to place the plot in a page.
		"""

		plot_script, plot_div = components(self.plots_grid)
		return plot_script, plot_div

	def Save_HTML(self, filename, title = "Repertoire Comparison UpSet Graph"):
		"""Calls Bokeh save function to save an HTML file containing the current plot.

		Parameters
		----------
		filename: str
			The desired filename / filepath for the output HTML file.
		title: str
			A title to use for the HTML file; default is "Repertoire Comparison UpSet 
Graph".
		"""

		save(self.plots_grid, filename = filename, title = title)

def VJ_Gene_Plot(clone_df, png = None, title = "", vgene_col = "VGene", jgene_col = "JGene", 
count_col = "Clustered",
				 vgene_colors = vgene_colors, vfamily_colors = vfamily_colors, 
jgene_colors = jgene_colors,
				 vj_gap = 0.008, vgene_gap = 0.0, line_width = 0.4, figsize = 
(800, 800), hover_tooltip = True):
	"""Creates a donut (??) chart for prevalence of all V/J gene pairs in a Repertoire.

	Parameters
	----------
	clone_df: pandas DataFrame

	Returns
	----------
	script: str
	div: str
	"""

	figure_params = {
		"plot_width": figsize[0],
		"plot_height": figsize[1],
		#"sizing_mode": "scale_both",
		"x_range": Range1d(-0.5, 1.5, bounds = (-1.5, 2.5)),
		"y_range": Range1d(-0.5, 1.5, bounds = (-1.5, 2.5)),
		#"outline_line_alpha": 0.0,
		"title": title,
		"tools": "pan, wheel_zoom, box_zoom, tap, save, reset, help",
		"active_scroll": "wheel_zoom",
		"toolbar_location": "right"
	}
```

```
        plot = figure(**figure_params)
        plot.grid.visible = False
        plot.axis.visible = False

        if hover_tooltip:
                hover_tool = HoverTool(tooltips = [("Gene", "@legend"), ("Percent", 
"@percent{(0.00%)}")],
                                                                          point_policy = 
"snap_to_data")
                plot.add_tools(hover_tool)

        gene_df = clone_df[[vgene_col, jgene_col, count_col]].groupby([vgene_col, 
jgene_col]).agg({count_col: sum})
        #Sort by V gene ascending, then J gene ascending
        gene_df = gene_df.sort_index()
        gene_df = gene_df.reset_index()

        total_vgenes = len(gene_df[vgene_col].drop_duplicates())
        total_gapsize = total_vgenes * vgene_gap
        remaining_size = 360.0 - float(total_gapsize)
        gap_size = float(vgene_gap)

        total_counts = gene_df[count_col].sum()
        gene_df["Arc_Length"] = gene_df[count_col] / total_counts * remaining_size
        #Starting at 90 degrees (top center of the circle) plus half the gap size
        #cur_v_start = -90.0 + (gap_size / 2.0)
        cur_v_start = 90.0 + (gap_size / 2.0)

        v_start_angles = []
        v_end_angles = []
        vgene_facecolors = []
        vgene_hover_colors = []
        vfamily_facecolors = []
        vfamily_hover_colors = []
        v_legend_text = []
        v_legend_percent = []

        j_start_angles = []
        j_end_angles = []
        jgene_facecolors = []
        jgene_hover_colors = []
        j_legend_text = []
        j_legend_percent = []

        for vgene in gene_df[vgene_col].drop_duplicates():
                cur_vgene_df = gene_df[gene_df[vgene_col] == vgene]
                vfamily = vgene.split("-")[0]

                vgene_color = vgene_colors[vgene]
                vgene_hover_color = vgene_color.darken(0.05)
                vfamily_color = vfamily_colors[vfamily]
                vfamily_hover_color = vfamily_color.darken(0.05)

                v_arc_length = cur_vgene_df["Arc_Length"].sum()
                cur_v_end = cur_v_start + v_arc_length

                v_start_angles.append(cur_v_start)
                v_end_angles.append(cur_v_end)

                vgene_facecolors.append(vgene_color)
                vgene_hover_colors.append(vgene_hover_color)
                vfamily_facecolors.append(vfamily_color)
                vfamily_hover_colors.append(vfamily_hover_color)

                v_legend_text.append(vgene)
                cur_vgene_counts = cur_vgene_df["Clustered"].sum()
                v_legend_percent.append(cur_vgene_counts / total_counts)

                cur_j_start = cur_v_start
                for jgene, jgene_arc_length in zip(cur_vgene_df[jgene_col],
cur_vgene_df["Arc_Length"]):
                        cur_j_end = cur_j_start + jgene_arc_length

                        jgene_color = jgene_colors[jgene]
                        jgene_hover_color = jgene_color.darken(0.05)

                        j_start_angles.append(cur_j_start)
                        j_end_angles.append(cur_j_end)

                        jgene_facecolors.append(jgene_color)
                        jgene_hover_colors.append(jgene_hover_color)

                        cur_j_start = cur_j_end

                        j_legend_text.append(jgene)
                        cur_jgene_counts = cur_vgene_df[cur_vgene_df[jgene_col] ==
jgene]["Clustered"].sum()
                        j_legend_percent.append(cur_jgene_counts / cur_vgene_counts)

                cur_v_start = cur_v_end + gap_size

        v_wedge_data = {
                "start_angle": v_start_angles,
                "end_angle": v_end_angles,
                "fill_color": vgene_facecolors,
                "legend": v_legend_text,
                "percent": v_legend_percent,
                "vgene_facecolors": vgene_facecolors,
                "vfamily_facecolors": vfamily_facecolors,
                "hover_fill_color": vgene_hover_colors,
                "vgene_hover_colors": vgene_hover_colors,
                "vfamily_hover_colors": vfamily_hover_colors
        }
        v_source = ColumnDataSource(v_wedge_data)

        v_inner_rad = 0.4
        v_outer_rad = 0.692

        plot.annular_wedge(x = 0.5, y = 0.5, start_angle = "start_angle", end_angle = "end_angle",
                                           fill_color = "fill_color", selection_fill_color =
"fill_color",
                                           nonselection_fill_color = "fill_color",
selection_fill_alpha = 1.0,
                                           nonselection_fill_alpha = 0.2, hover_fill_color =
"hover_fill_color", inner_radius = v_inner_rad,
                                           outer_radius = v_outer_rad, line_color = "black",
line_width = line_width, source = v_source,
                                           legend = "legend", start_angle_units = "deg",
end_angle_units = "deg")

        j_wedge_data = {
                "start_angle": j_start_angles,
                "end_angle": j_end_angles,
                "fill_color": jgene_facecolors,
                "legend": j_legend_text,
                "percent": j_legend_percent,
                "hover_fill_color": jgene_hover_colors
        }

        j_source = ColumnDataSource(j_wedge_data)
        j_inner_rad = v_outer_rad + vj_gap
        j_outer_rad = j_inner_rad + 0.15

        plot.annular_wedge(x = 0.5, y = 0.5, start_angle = "start_angle", end_angle = "end_angle",
                                            fill_color = "fill_color", selection_fill_color =
"fill_color",
                                            nonselection_fill_color = "fill_color",
selection_fill_alpha = 1.0,
                                            nonselection_fill_alpha = 0.2, hover_fill_color =
"hover_fill_color", inner_radius = j_inner_rad,
                                            outer_radius = j_outer_rad, line_color = "black",
line_width = line_width, source = j_source,
                                            legend = "legend", start_angle_units = "deg",
end_angle_units = "deg")

        if png is not None:
                export_png(plot, png)

        change_v_color = CustomJS(args = {"source": v_source}, code = """
                var selection = cb_obj.value;
                var new_color_array;
                var new_hover_array;
                if(selection.toLowerCase().indexOf("gene") !== -1) {
                        new_color_array = source.data["vgene_facecolors"];
                        new_hover_array = source.data["vgene_hover_colors"];
                } else {
                        new_color_array = source.data["vfamily_facecolors"];
                        new_hover_array = source.data["vfamily_hover_colors"];
                }
                var fill_color = source.data["fill_color"];
                var hover_fill_color = source.data["hover_fill_color"];
                for(idx = 0; idx < fill_color.length; idx++) {
                        fill_color[idx] = new_color_array[idx];
                        hover_fill_color[idx] = new_hover_array[idx];
                }
                source.change.emit();
        """)

        v_data_color_by = Select(title = "Color by:", options = ["V Gene", "V Family"],
                                                          value = "V Gene", callback =
change_v_color)

        plot_layout = column(v_data_color_by, plot)
        return plot_layout

def Violin_SHM_Plot(clone_df, png = None, title = "", vshm_col = "V_SHM", jshm_col = "J_SHM",
split_col = None,
                                         quads = True, violin_width = 0.8, line_width = 0.4,
figsize = (1000, 600), hover_tooltip = True):
        """Creates a SHM violin plot that can be used to compare multiple categories in a
Repertoire.

        Parameters
        ----------
        clone_df: pandas DataFrame

        Returns
        ----------
        script: str
        div: str
        """

        figure_params = {
                "plot_width": figsize[0],
			"plot_height": figsize[1],
			"y_range": Range1d(-0.005, 0.3, bounds = (-0.01, 0.31)),
			"title": title,
			"tools": "pan, wheel_zoom, box_zoom, save, reset, help",
			"active_scroll": "wheel_zoom",
			"toolbar_location": "right"
		}

		plot = figure(**figure_params)
		plot.grid.visible = False
		plot.xaxis.minor_tick_line_color = None
		plot.xaxis.major_label_text_font_size = "10pt"
		plot.yaxis.axis_label = "V/J Gene SHM"
		plot.yaxis.major_label_text_font_size = "10pt"
		plot.yaxis.formatter = NumeralTickFormatter(format = "0.00%")

		if hover_tooltip:
			hover_tooltips = [("Mean SHM", "@mean{(0.00%)}"), ("Max SHM", "@max{(0.00%)}"),
			                                   ("25th Percentile", "@quantile25{(0.00%)}"),
("75th Percentile", "@quantile75{(0.00%)}")]
			hover_tool = HoverTool(point_policy = "follow_mouse", tooltips = hover_tooltips)
			plot.add_tools(hover_tool)

		shm_cols = []
		if vshm_col is not None:
			shm_cols.append(vshm_col)
		if jshm_col is not None:
			shm_cols.append(jshm_col)

		#To compare samples, add the sample column to split on to the DataFrame
		if split_col is not None:
			shm_cols.append(split_col)
			shm_df = clone_df[shm_cols]

			samples = []
			shm_dfs = []
			for sample, df in shm_df.groupby([split_col]):
				samples.append(sample)
				shm_dfs.append(df)

		else:
			samples = ["Repertoire"]
			shm_dfs = [clone_df[shm_cols]]

		vshm_violin_color = "lightgreen"
		jshm_violin_color = "slateblue"
		violin_xs = []
		violin_ys = []
		violin_colors = []
		violin_legends = []
		hover_means = []
		hover_maxes = []
		hover_25quantiles = []
		hover_75quantiles = []
		x_location_to_category = {}
		violin_x_offset = 0

		for sample, df in zip(samples, shm_dfs):
			#Create the density functions
			if vshm_col in df.columns:
				vshm_mean = df[vshm_col].mean()
				vshm_max = df[vshm_col].max()
				hover_means.append([vshm_mean])
				hover_maxes.append([vshm_max])
				hover_25quantiles.append([df[vshm_col].quantile(0.25)])
```

```
                        hover_75quantiles.append([df[vshm_col].quantile(0.75)])

                        y_points = numpy.linspace(0.0, vshm_max, 300)  #Create the y range of 300 
points from min to max
                        reversed_y_points = numpy.flipud(y_points)
                        v_kernel = gaussian_kde(df[vshm_col], "scott")
                        vshm_x_points = v_kernel(y_points)

                        #Normalize the x range to standard width; negate V SHM points to place it 
on the left half of the violin
                        vshm_x_points = -vshm_x_points / vshm_x_points.max() * violin_width / 2.0

                        #Return to the patch starting points if a different violin is drawn for 
the other half, or mirror data
                        if jshm_col in df.columns:
                                vshm_x_points = numpy.append(vshm_x_points, 
abs(vshm_x_points).min())
                                vshm_y_points = numpy.append(y_points, y_points.min())
                        else:
                                reversed_vshm_x = numpy.flipud(-vshm_x_points)
                                vshm_x_points = numpy.append(vshm_x_points, reversed_vshm_x)
                                vshm_y_points = numpy.append(y_points, reversed_y_points)

                        violin_xs.append(vshm_x_points + violin_x_offset)
                        violin_ys.append(vshm_y_points)
                        violin_colors.append(vshm_violin_color)
                        violin_legends.append("V Gene SHM")

                if jshm_col in df.columns:
                        jshm_mean = df[jshm_col].mean()
                        jshm_max = df[jshm_col].max()
                        hover_means.append([jshm_mean])
                        hover_maxes.append([jshm_max])
                        hover_25quantiles.append([df[jshm_col].quantile(0.25)])
                        hover_75quantiles.append([df[jshm_col].quantile(0.75)])

                        y_points = numpy.linspace(0.0, jshm_max, 300)  #Create the y range of 300 
points from min to max
                        reversed_y_points = numpy.flipud(y_points)
                        j_kernel = gaussian_kde(df[jshm_col], "scott")
                        jshm_x_points = j_kernel(y_points)

                        #Normalize the x range to standard width
                        jshm_x_points = jshm_x_points / jshm_x_points.max() * violin_width / 2.0

                        #Return to the patch starting points if a different violin is drawn for 
the other half, or mirror data
                        if vshm_col in df.columns:
                                jshm_x_points = numpy.append(jshm_x_points, 
abs(jshm_x_points).min())
                                jshm_y_points = numpy.append(y_points, y_points.min())
                        else:
                                reversed_jshm_x = numpy.flipud(-jshm_x_points)
                                jshm_x_points = numpy.append(jshm_x_points, reversed_jshm_x)
                                jshm_y_points = numpy.append(y_points, reversed_y_points)

                        violin_xs.append(jshm_x_points + violin_x_offset)
                        violin_ys.append(jshm_y_points)
                        violin_colors.append(jshm_violin_color)
                        violin_legends.append("J Gene SHM")

                if quads:
                        pass

                x_location_to_category[violin_x_offset] = sample
                violin_x_offset += violin_width * 1.2

        violin_data = {
                "xs": violin_xs,
                "ys": violin_ys,
                "fill_color": violin_colors,
                "legend": violin_legends,
                "mean": hover_means,
                "max": hover_maxes,
                "quantile25": hover_25quantiles,
                "quantile75": hover_75quantiles
        }
        violin_source = ColumnDataSource(violin_data)

        plot.patches(xs = "xs", ys = "ys", fill_color = "fill_color", line_color = "black", 
line_width = line_width,
                                 legend = "legend", source = violin_source)

        #Replace / remap the X axis tickers to the categorical samples
        plot.xaxis.ticker = FixedTicker(ticks = [loc for loc in x_location_to_category])
        plot.xaxis.major_label_overrides = x_location_to_category
        plot.x_range.bounds = (min(x_location_to_category.keys()) - 1, 
max(x_location_to_category.keys()) + 1)

        if png is not None:
                export_png(plot, png)

        return plot

def Mosaic_Plot(clone_df, png = None, title = "", top_clones = 5000, count_col = "Clustered", 
vgene_col = "VGene",
                                jgene_col = "JGene", isotype_col = "Isotype", vshm_col = "V_SHM", 
jshm_col = "J_SHM",
                                vgene_colors = vgene_colors, vfamily_colors = vfamily_colors, 
jgene_colors = jgene_colors,
                                isotype_colors = isotype_colors, line_width = 0.3, figsize = 
(600, 600), hover_tooltip = True):
        """"""

        figure_params = {
                "plot_width": figsize[0],
                "plot_height": figsize[1],
                #"sizing_mode": "scale_both",
                "x_range": Range1d(-0.1, 1.1, bounds = (-1.0, 2.0)),
                "y_range": Range1d(-0.1, 1.1, bounds = (-1.0, 2.0)),
                #"outline_line_alpha": 0.0,
                "title": title,
                "tools": "pan, wheel_zoom, box_zoom, save, reset, help",
                "active_scroll": "wheel_zoom",
                "toolbar_location": "right"
        }

        plot = figure(**figure_params)
        plot.grid.visible = False
        plot.axis.visible = False

        hover_tooltips = [("Clone ID", "@CloneID")]

        info_cols = [count_col]
        if vgene_col is not None:
                info_cols.append(vgene_col)
                hover_tooltips.append(("V Gene", "@" + vgene_col))
        if jgene_col is not None:
                info_cols.append(jgene_col)
                hover_tooltips.append(("J Gene", "@" + jgene_col))
```

```
if isotype_col is not None:
        info_cols.append(isotype_col)
        hover_tooltips.append(("Isotype", "@" + isotype_col))
if vshm_col is not None:
        info_cols.append(vshm_col)
        hover_tooltips.append(("V Gene SHM", "@" + vshm_col + "{(0.00%)}"))
if jshm_col is not None:
        info_cols.append(jshm_col)
        hover_tooltips.append(("J Gene SHM", "@" + jshm_col + "{(0.00%)}"))

if hover_tooltip:
        hover_tool = HoverTool(point_policy = "snap_to_data", tooltips = hover_tooltips)
        plot.add_tools(hover_tool)

mosaic_df = clone_df[info_cols]
mosaic_df = mosaic_df.sort_values([count_col], ascending = [False])

if top_clones:
        mosaic_df = mosaic_df.head(top_clones)

total_area = float(mosaic_df[count_col].sum())
mosaic_df["Clone_Frequencies"] = mosaic_df[count_col].astype(float) / total_area

hover_tooltips.append(("Clone Frequency", "@Clone_Frequencies{(0.00%)}"))

mosaic_rects = squarify(mosaic_df["Clone_Frequencies"].tolist(), 0.0, 0.0, 1.0, 1.0)
#Add half width/height to x/y position for center points
mosaic_df["x"] = [rect["x"] + rect["dx"] / 2.0 for rect in mosaic_rects]
mosaic_df["y"] = [rect["y"] + rect["dy"] / 2.0 for rect in mosaic_rects]
mosaic_df["width"] = [rect["dx"] for rect in mosaic_rects]
mosaic_df["height"] = [rect["dy"] for rect in mosaic_rects]

#By default there is no legend text, since colors are alternating and non-informative
mosaic_df["legend"] = ""
mosaic_df["Empty_Legend"] = ""

alternating_colors = [RGB(102, 194, 165), RGB(252, 141, 98), RGB(141, 160, 203)]
alt2_color_cycle = cycle(alternating_colors[0:2])
alt3_color_cycle = cycle(alternating_colors)
mosaic_df["alternating2_colors"] = [next(alt2_color_cycle) for _ in mosaic_rects]
mosaic_df["alternating3_colors"] = [next(alt3_color_cycle) for _ in mosaic_rects]
#Default color scheme is alternating 3 colors
mosaic_df["fill_color"] = mosaic_df["alternating3_colors"]

#Set up various mosaic coloring options and associated legends
color_select_options = ["Alternating (2)", "Alternating (3)"]
if vgene_col in mosaic_df.columns:
        mosaic_df["vgene_colors"] = mosaic_df[vgene_col].map(vgene_colors)
        vfamilies = mosaic_df[vgene_col].str.split("-").str[0]
        mosaic_df["vfamily_colors"] = vfamilies.map(vfamily_colors)
        color_select_options.append("V Gene")
        color_select_options.append("V Family")
        mosaic_df["VGene_Legend"] = mosaic_df[vgene_col]
        mosaic_df["VFamily_Legend"] = mosaic_df[vgene_col].str.split("-").str[0]
if jgene_col in mosaic_df.columns:
        mosaic_df["jgene_colors"] = mosaic_df[jgene_col].map(jgene_colors)
        color_select_options.append("J Gene")
        mosaic_df["JGene_Legend"] = mosaic_df[jgene_col]
if isotype_col in mosaic_df.columns:
        mosaic_df["isotype_colors"] = mosaic_df[isotype_col].map(isotype_colors)
        color_select_options.append("Isotype")
        mosaic_df["Isotype_Legend"] = mosaic_df[isotype_col]

#Using viridis as a quantitative heatmap color scheme for SHM values
```

```
        #The SHM values are binned into 180 groups; viridis in >180 bins uses some values twice,
which pandas.cut can't use
        shm_viridis = list(viridis(180))
        colorbar_tick_formatter = NumeralTickFormatter(format = "0.00%")

        if vshm_col in mosaic_df.columns:
                vshm_min = mosaic_df[vshm_col].min()
                vshm_max = mosaic_df[vshm_col].max()
                #Use pandas.cut to bin the V gene SHM values into the heatmap colors
                mosaic_df["vshm_colors"] = pandas.cut(mosaic_df[vshm_col], bins = 180, labels =
shm_viridis)
                color_select_options.append("V Gene SHM")

                vshm_color_mapper = LinearColorMapper(palette = shm_viridis, low = vshm_min, high
= vshm_max)
                vshm_ticks = FixedTicker(ticks = numpy.linspace(vshm_min, vshm_max, 8))
                vshm_colorbar = ColorBar(color_mapper = vshm_color_mapper, location = (0, 0),
name = "vshm_colorbar",
                                                                  label_standoff = 12, formatter
= colorbar_tick_formatter, ticker = vshm_ticks)
                plot.add_layout(vshm_colorbar, "right")

        if jshm_col in mosaic_df.columns:
                jshm_min = mosaic_df[jshm_col].min()
                jshm_max = mosaic_df[jshm_col].max()
                #Use pandas.cut to bin the J gene SHM values into the heatmap colors
                mosaic_df["jshm_colors"] = pandas.cut(mosaic_df[jshm_col], bins = 180, labels =
shm_viridis)
                color_select_options.append("J Gene SHM")

                jshm_color_mapper = LinearColorMapper(palette = shm_viridis, low = jshm_min, high
= jshm_max)
                jshm_ticks = FixedTicker(ticks = numpy.linspace(jshm_min, jshm_max, 8))
                jshm_colorbar = ColorBar(color_mapper = jshm_color_mapper, location = (0, 0),
name = "jshm_colorbar",
                                                                  label_standoff = 12, formatter
= colorbar_tick_formatter, ticker = jshm_ticks)
                plot.add_layout(jshm_colorbar, "right")

        mosaic_source = ColumnDataSource(mosaic_df)

        plot.rect(x = "x", y = "y", width = "width", height = "height", fill_color = "fill_color",
legend = "legend",
                          line_color = "black", line_width = line_width, source = mosaic_source)

        #By default, the plot legend and ColorBar should be turned off (since the color is
repeating and uninformative)
        plot.legend[0].visible = False
        vshm_colorbar = plot.select("vshm_colorbar")[0]
        jshm_colorbar = plot.select("jshm_colorbar")[0]
        vshm_colorbar.visible = False
        jshm_colorbar.visible = False

        if png is not None:
                export_png(plot, png)

        change_args = {
                "source": mosaic_source,
                "legend_obj": plot.legend[0],
                "vshm_colorbar_obj": vshm_colorbar,
                "jshm_colorbar_obj": jshm_colorbar
        }
        change_rect_color = CustomJS(args = change_args, code = """
                var selection = cb_obj.value.toLowerCase();
                var new_color_array;
```

```
		var new_legend_array;

		if(selection.indexOf("v gene shm") !== -1) {
			new_color_array = source.data["vshm_colors"];
			new_legend_array = source.data["Empty_Legend"];
			legend_obj.visible = false;
			vshm_colorbar_obj.visible = true;
			jshm_colorbar_obj.visible = false;
		} else if(selection.indexOf("j gene shm") !== -1) {
			new_color_array = source.data["jshm_colors"];
			new_legend_array = source.data["Empty_Legend"];
			legend_obj.visible = false;
			vshm_colorbar_obj.visible = false;
			jshm_colorbar_obj.visible = true;
		} else if(selection.indexOf("v gene") !== -1) {
			new_color_array = source.data["vgene_colors"];
			new_legend_array = source.data["VGene_Legend"];
			legend_obj.visible = true;
			vshm_colorbar_obj.visible = false;
			jshm_colorbar_obj.visible = false;
		} else if(selection.indexOf("v family") !== -1) {
			new_color_array = source.data["vfamily_colors"];
			new_legend_array = source.data["VFamily_Legend"];
			legend_obj.visible = true;
			vshm_colorbar_obj.visible = false;
			jshm_colorbar_obj.visible = false;
		} else if(selection.indexOf("j gene") !== -1) {
			new_color_array = source.data["jgene_colors"];
			new_legend_array = source.data["JGene_Legend"];
			legend_obj.visible = true;
			vshm_colorbar_obj.visible = false;
			jshm_colorbar_obj.visible = false;
		} else if(selection.indexOf("isotype") !== -1) {
			new_color_array = source.data["isotype_colors"];
			new_legend_array = source.data["Isotype_Legend"];
			legend_obj.visible = true;
			vshm_colorbar_obj.visible = false;
			jshm_colorbar_obj.visible = false;
		} else if(selection.indexOf("2") !== -1) {
			new_color_array = source.data["alternating2_colors"];
			new_legend_array = source.data["Empty_Legend"];
			legend_obj.visible = false;
			vshm_colorbar_obj.visible = false;
			jshm_colorbar_obj.visible = false;
		} else {
			new_color_array = source.data["alternating3_colors"];
			new_legend_array = source.data["Empty_Legend"];
			legend_obj.visible = false;
			vshm_colorbar_obj.visible = false;
			jshm_colorbar_obj.visible = false;
		}

		var fill_color = source.data["fill_color"];
		var legend = source.data["legend"];
		for(idx = 0; idx < fill_color.length; idx++) {
			fill_color[idx] = new_color_array[idx];
			legend[idx] = new_legend_array[idx];
		}
		source.change.emit();
	""")

	patch_coloring_select = Select(title = "Color by:", options = color_select_options, value
= "Alternating (3)",
							callback = change_rect_color)
```

```
        plot_layout = column(patch_coloring_select, plot)

        return plot_layout

def Burtin_VGene_SHM_Plot(clone_df, png = None, title = "", vgene_col = "VGene", vshm_col =
"V_SHM", split_col = None,
                                                  vfamily_colors = vfamily_colors, label_arc =
20, figsize = (900, 900)):
        """"""

        figure_params = {
                "plot_width": figsize[0],
                "plot_height": figsize[1],
                "x_axis_type": None,
                "y_axis_type": None,
                "x_range": Range1d(-45, 45, bounds = (-50, 50)),
                "y_range": Range1d(-45, 45, bounds = (-50, 50)),
                "title": title,
                "tools": "pan, wheel_zoom, box_zoom, save, reset, help",
                "active_scroll": "wheel_zoom",
                "toolbar_location": "right",
                "background_fill_color": RGB(216, 216, 216)
        }

        plot = figure(**figure_params)
        plot.grid.visible = False
        plot.axis.visible = False

        label_offset = 90 #Offset the SHM % labels to the top of the plot
        plot_data_degrees = 360 - label_arc
        initial_angle = label_offset + label_arc / 2
        ending_angle = label_offset + 360 - label_arc / 2
        plot_inner_rad = 10
        plot_outer_rad = 35
        plot_thickness = plot_outer_rad - plot_inner_rad

        df_cols = [vgene_col, vshm_col]
        #If comparing multiple samples, add the sample column to split on to the DataFrame
        if split_col is not None:
                df_cols.append(split_col)

        vgene_shm_df = clone_df[df_cols].sort_values([vgene_col])

        total_vgenes = len(vgene_shm_df[vgene_col].drop_duplicates())
        vgene_arc_degrees = plot_data_degrees / total_vgenes

        #Create and color arc backgrounds by V family
        vgene_family_df = vgene_shm_df[[vgene_col]].drop_duplicates().reset_index(drop = True)
        vgene_family_df["VFamily"] = vgene_family_df[vgene_col].str.split("-").str[0]
        vgene_family_df["fill_color"] = vgene_family_df["VFamily"].map(vfamily_colors)
        vfamily_arc_length = plot_data_degrees / total_vgenes
        vgene_family_df["start_angle"] = vgene_family_df.index * vfamily_arc_length +
initial_angle
        vgene_family_df["end_angle"] = vgene_family_df["start_angle"] + vfamily_arc_length

        vfamily_source = ColumnDataSource(vgene_family_df)
        plot.annular_wedge(x = 0, y = 0, start_angle = "start_angle", end_angle = "end_angle",
fill_color = "fill_color",
                                            inner_radius = plot_inner_rad, outer_radius =
plot_outer_rad, line_color = None,
                                            source = vfamily_source, start_angle_units = "deg",
end_angle_units = "deg")

        if split_col in vgene_shm_df:
                vgene_shm_dfs = [sample_df_tup for sample_df_tup in 
vgene_shm_df.groupby([split_col])]

                samples = []
                grouped_vgene_shm_dfs = []
                for sample, df in vgene_shm_dfs:
                        samples.append(sample)

        grouped_vgene_shm_dfs.append(df.groupby([vgene_col])[vshm_col].agg({"mean"}))

                #Add the V genes that may be present in one sample but not in the current one
                all_vgenes = vgene_shm_df[vgene_col].drop_duplicates().tolist()
                grouped_vgene_shm_dfs = [df.reindex(all_vgenes).reset_index() for df in 
grouped_vgene_shm_dfs]

                vshm_min = min([df["mean"].min() for df in grouped_vgene_shm_dfs])
                vshm_max = max([df["mean"].max() for df in grouped_vgene_shm_dfs])

        else:
                grouped_vgene_shm_df = 
vgene_shm_df.groupby([vgene_col])[vshm_col].agg({"mean"}).reset_index()
                grouped_vgene_shm_df = 
grouped_vgene_shm_df.sort_values([vgene_col]).reset_index(drop = True)

                vshm_min = grouped_vgene_shm_df["mean"].min()
                vshm_max = grouped_vgene_shm_df["mean"].max()

                samples = ["All"]
                grouped_vgene_shm_dfs = [grouped_vgene_shm_df]

        #Create the labels and radial axis lines for the SHM data
        shm_labels = ["{0:.1%}".format(shm) for shm in numpy.linspace(vshm_min, vshm_max, 7)]
        shm_label_radii = numpy.linspace(plot_inner_rad, plot_outer_rad, 7)
        plot.circle(x = 0, y = 0, radius = shm_label_radii, fill_color = None, line_color = 
"white")
        plot.text(x = 0, y = shm_label_radii[1:], text = shm_labels[1:], text_font_size = "10pt",
                          text_align = "center", text_baseline = "middle")

        #Create line-width annular wedges to separate V genes
        sep_angles = numpy.linspace(initial_angle, ending_angle, total_vgenes + 1)
        sep_inner_radius = plot_inner_rad - 1
        sep_outer_radius = plot_outer_rad + 1
        plot.annular_wedge(x = 0, y = 0, start_angle = sep_angles, end_angle = sep_angles, 
fill_color = None,
                                           inner_radius = sep_inner_radius, outer_radius = 
sep_outer_radius, line_color = "black",
                                           start_angle_units = "deg", end_angle_units = "deg")

        #Gene text labels; text angle location is the midpoint of the V gene separation lines
        text_radius = plot_outer_rad + 3.5
        text_radian_locs = numpy.deg2rad((sep_angles[1:] + sep_angles[:-1]) / 2)
        text_x = text_radius * numpy.cos(text_radian_locs)
        text_y = text_radius * numpy.sin(text_radian_locs)
        #Angle the text based on the position around the circle; reverse the left half so the text 
isn't upside-down
        mid_graph_radian = numpy.deg2rad(label_offset + 180)
        text_angles = [rad if rad > mid_graph_radian else rad + numpy.pi for rad in 
text_radian_locs]
        plot.text(x = text_x, y = text_y, text = vgene_family_df[vgene_col], angle = text_angles,
                          text_font_size = "10pt", text_align = "center", text_baseline = 
"middle")

        #Finally draw the bars and legend for the mean SHM values for all clones of a specific V 
gene
        total_samples = len(grouped_vgene_shm_dfs)
        vgene_arc_radians = numpy.deg2rad(vgene_arc_degrees)
        bar_width = vgene_arc_radians / (total_samples + 1)
        spacer_width = bar_width / (total_samples + 1)
        sample_colors = (RGB(60, 60, 60), RGB(130, 40, 40), RGB(60, 60, 130), RGB(10, 50, 100),
RGB(150, 100, 20))
        sample_label_ys = numpy.linspace(-total_samples, total_samples, total_samples)
        arc_starts = text_radian_locs - (vgene_arc_radians / 2) + spacer_width

        for sample, cur_df in enumerate(grouped_vgene_shm_dfs):
                bar_start_angles = arc_starts + sample * (bar_width + spacer_width)
                bar_end_angles = bar_start_angles + bar_width
                cur_df["Normalized_SHM"] = cur_df["mean"] / vshm_max

                shm_bars = cur_df["Normalized_SHM"] * plot_thickness + plot_inner_rad
                plot.annular_wedge(x = 0, y = 0, start_angle = bar_start_angles, end_angle =
bar_end_angles, line_color = None,
                                            inner_radius = plot_inner_rad, outer_radius =
shm_bars, fill_color = sample_colors[sample])

                if total_samples > 1:
                        plot.rect(x = -2, y = sample_label_ys[sample], width = 2.5, height = 1.5,
color = sample_colors[sample])
                        plot.text(x = 0, y = sample_label_ys[sample], text = {"value":
samples[sample]}, text_font_size = "10pt",
                                          text_baseline = "middle")

        if png is not None:
                export_png(plot, png)

        return plot

def Diversity_Plot(clone_df, png = None, title = "", count_col = "Clustered", split_col = None,
line_width = 3,
                                    add_control_diversities = True, y_axis_type = "log", figsize =
(1000, 700)):
        """"""

        figure_params = {
                "plot_width": figsize[0],
                "plot_height": figsize[1],
                "x_range": Range1d(0, 10),
                "y_axis_type": y_axis_type,
                "title": title,
                "tools": "save, help",
                "toolbar_location": "right"
        }

        plot = figure(**figure_params)
        plot.xgrid.grid_line_alpha = 0.0
        plot.xaxis.axis_label = "Order (N)"
        plot.yaxis.axis_label = "Hill Diversity Constant"
        plot.yaxis.formatter = BasicTickFormatter()

        #If comparing multiple samples, add the sample column to split on to the DataFrame
        if split_col is not None:
                diversity_df = clone_df[[count_col, split_col]]

                samples = []
                diversity_dfs = []
                for sample, df in diversity_df.groupby([split_col]):
                        samples.append(sample)
                        diversity_dfs.append(df)

        else:
                samples = ["Repertoire"]
```

```
		diversity_dfs = [clone_df[[count_col]]]

	sample_colors = (RGB(30, 160, 120), RGB(220, 90, 0), RGB(120, 110, 180), RGB(230, 40, 
140))
	for sample, df, line_color in zip(samples, diversity_dfs, sample_colors[:len(samples)]):
		hill_indices = Hill_Diversity_Index(df[count_col])
		n_orders = [i[0] for i in hill_indices]
		order_diversities = [i[1] for i in hill_indices]

		#ADD MORE LINE STYLES (dotted, etc.)
		plot.line(x = n_orders, y = order_diversities, color = line_color, line_width = 
line_width, legend = sample)

	if add_control_diversities:
		total_clones = max([len(i) for i in diversity_dfs])
		total_counts = max([df[count_col].sum() for df in diversity_dfs])

		#Very highly polarized data creates a sample in which the top 20 clones are 20% 
of the total by prevalence
		top20_20_data = [total_counts * 0.2 / 20] * 20
		top20_20_data += [total_counts * 0.8 / (total_clones - 20) for _ in 
range(total_clones - 20)]
		#Highly polarized data has the top 20 clones at 15% of the total
		top20_15_data = [total_counts * 0.15 / 20] * 20
		top20_15_data += [total_counts * 0.85 / (total_clones - 20) for _ in 
range(total_clones - 20)]
		#Moderately polarized data has the top 20 clones at 10% of the total
		top20_10_data = [total_counts * 0.1 / 20] * 20
		top20_10_data += [total_counts * 0.9 / (total_clones - 20) for _ in 
range(total_clones - 20)]
		#Lowly polarized data has the top 20 clones at 5% of the total
		top20_5_data = [total_counts * 0.05 / 20] * 20
		top20_5_data += [total_counts * 0.95 / (total_clones - 20) for _ in 
range(total_clones - 20)]

		top20_20_diversities = [i[1] for i in Hill_Diversity_Index(top20_20_data)]
		top20_15_diversities = [i[1] for i in Hill_Diversity_Index(top20_15_data)]
		top20_10_diversities = [i[1] for i in Hill_Diversity_Index(top20_10_data)]
		top20_5_diversities = [i[1] for i in Hill_Diversity_Index(top20_5_data)]
		plot.line(x = n_orders, y = top20_20_diversities, color = RGB(160, 200, 230), 
alpha = 0.8, line_dash = (12,),
					line_width = line_width, legend = "Very Highly Polarized (Top 
20 Clones 20%)")
		plot.line(x = n_orders, y = top20_15_diversities, color = RGB(30, 120, 180), 
alpha = 0.8, line_dash = (12,),
					line_width = line_width, legend = "Highly Polarized (Top 20 
Clones 15%)")
		plot.line(x = n_orders, y = top20_10_diversities, color = RGB(180, 220, 140), 
alpha = 0.8, line_dash = (12,),
					line_width = line_width, legend = "Moderately Polarized (Top 20 
Clones 10%)")
		plot.line(x = n_orders, y = top20_5_diversities, color = RGB(50, 160, 40), alpha 
= 0.8, line_dash = (12,),
					line_width = line_width, legend = "Lowly Polarized (Top 20 
Clones 5%)")

	if png is not None:
		export_png(plot, png)

	return plot

def CDR_Length_Histogram_Plot(clone_df, png = None, title = "", cdr_col = "CDR3_AA", split_col = 
None,
								quantile_boundries = (0.0001, 0.9999), 
figsize = (800, 600)):
        figure_params = {
                "plot_width": figsize[0],
                "plot_height": figsize[1],
                "title": title,
                "tools": "pan, wheel_zoom, box_zoom, save, reset, help",
                "active_scroll": "wheel_zoom",
                "toolbar_location": "right"
        }

        plot = figure(**figure_params)
        plot.grid.visible = False
        plot.xaxis.minor_tick_line_color = None
        plot.xaxis.axis_label = "CDR3 Length"
        plot.xaxis.axis_label_text_font_size = "12pt"
        plot.xaxis.major_label_text_font_size = "12pt"
        plot.yaxis.axis_label = "P(x)"
        plot.yaxis.axis_label_text_font_size = "12pt"
        plot.yaxis.major_label_text_font_size = "12pt"

        #To compare samples, add the sample column to split on to the DataFrame
        if split_col is not None:
                cdr3_df = clone_df[[cdr_col, split_col]]

                samples = []
                cdr3_lens = []
                for sample, df in cdr3_df.groupby([split_col]):
                        samples.append(sample)
                        cdr3_lens.append(df[cdr_col].str.len())

        else:
                samples = ["Repertoire"]
                cdr3_lens = [clone_df[cdr_col].str.len()]

        bin_min = min([cdr_len_series.min() for cdr_len_series in cdr3_lens])
        bin_max = max([cdr_len_series.max() for cdr_len_series in cdr3_lens]) + 1
        bin_range = [i for i in range(bin_min, bin_max)]

        bar_colors = ["#A0C8E6", "#32A032", "#1E78B4", "#B4DC8C"]
        bar_offset = 0.0
        bar_width = 1 / len(samples)

        upper_y = 0.0

        for idx, (sample, cdr_len_series) in enumerate(zip(samples, cdr3_lens)):
                heights, lefts = numpy.histogram(cdr_len_series, density = True, bins = 
bin_range)

                #Ensure proper Y axis scrolling boundaries are set
                if heights.max() > upper_y:
                        upper_y = heights.max()

                #Shift bars if multiple samples are being plotted
                lefts = lefts.astype(float)
                lefts += bar_offset

                bar_lefts = lefts[:-1]
                bar_rights = bar_lefts + bar_width

                plot.quad(top = heights, bottom = 0, left = bar_lefts, right = bar_rights, 
fill_color = bar_colors[idx],
                                  line_color = None, legend = sample)

                bar_offset += bar_width

        plot.y_range.start = -0.001
        plot.y_range.end = upper_y
        plot.y_range.bounds = (-0.05, upper_y * 1.5)

        if quantile_boundries is not None:
                lower_x = clone_df[cdr_col].str.len().quantile(quantile_boundries[0])
                upper_x = clone_df[cdr_col].str.len().quantile(quantile_boundries[1])

                plot.x_range.start = lower_x
                plot.x_range.end = upper_x

        plot.x_range.bounds = (0, bin_max + 4)

        if png is not None:
                export_png(plot, png)

        return plot

def Rarefaction_Plot(align_df, png = None, title = "", cdr_col = "CDR3_AA", split_col = None,
cdr_identity = 0.96,
                                         steps = 50, reads = None, figsize = (800, 600),
hover_tooltip = True, save_to_file = False):
        figure_params = {
                "plot_width": figsize[0],
                "plot_height": figsize[1],
                "title": title,
                "tools": "save, help",
                "toolbar_location": "right"
        }
        plot = figure(**figure_params)
        plot.xgrid.grid_line_color = None
        plot.xaxis.axis_label = "Total Sampled Reads"
        plot.yaxis.axis_label = "Total Clonotypes"
        plot.axis.formatter = NumeralTickFormatter(format = "0")

        tooltips = [("Total Sampled Reads", "@xs"), ("Total Clones", "@ys")]
        if hover_tooltip:
                hover_tool = HoverTool(point_policy = "snap_to_data", tooltips = tooltips, mode =
"hline", names = ["rar_line"])
                plot.add_tools(hover_tool)

        #If comparing multiple samples, add the sample column to split on to the DataFrame
        if split_col is not None:
                reads_df = align_df[[cdr_col, split_col]]

                samples = []
                reads_dfs = []
                for sample, df in reads_df.groupby([split_col]):
                        samples.append(sample)
                        reads_dfs.append(df)

                if hover_tooltip and len(samples) > 1:
                        tooltips.append(("Sample", "@sample"))

        else:
                samples = ["Repertoire"]
                reads_dfs = [align_df[[cdr_col]]]

        sample_colors = ["#1EA078", "#DC5A00", "#786EB4", "#E6288C", "#B4D28C", "#A028B4"]
        for sample, df, color in zip(samples, reads_dfs, sample_colors[:len(samples)]):
                total_reads = len(df)
                subsamp_sizes = []
                cur_total = 0

                if reads is not None:
                        subsamp_steps = reads
```

```
		else:
			subsamp_steps = math.floor(total_reads / steps)

		#Create the list of all read subsample counts to clonotype
		while cur_total < total_reads:
			if cur_total != 0:
				subsamp_sizes.append(cur_total)

			cur_total += subsamp_steps

		subsamp_sizes.append(total_reads)

		subsamp_clones = []
		for n in subsamp_sizes:
			sub_read_df = df.sample(n)
			sub_total_clones = Clonotype_Usearch(sub_read_df[cdr_col], identity =
cdr_identity)
			subsamp_clones.append(sub_total_clones)

		rarefaction_data = {
			"reads": subsamp_sizes,
			"clones": subsamp_clones,
			"sample": [sample if len(samples) > 1 else None] * len(subsamp_sizes)
		}
		rar_source = ColumnDataSource(rarefaction_data)

		plot.line(x = "reads", y = "clones", color = color, line_width = 3, source =
rar_source,
				  legend = "sample", name = "rar_line")
		plot.scatter(x = "reads", y = "clones", color = color, source = rar_source)

		if save_to_file:
			with open(sample + "_Rarefaction_Data.txt", "w") as
rarefaction_text_file:
				rarefaction_text_file.write("Reads\tClones\n")
				for read_count, clone_count in zip(subsamp_sizes,
subsamp_clones):

	rarefaction_text_file.write("{0}\t{1}\n".format(read_count, clone_count))

	if png is not None:
		export_png(plot, png)

	return plot

def Repertoire_Dashboard(clone_dfs, filename = None, title = "Repertoire Analysis Dashboard",
plot_title_prefix = "",
						 mosaic_top_clones = 5000, cyrcos_top_clones =
1000, upset_highlighted_sets = None,
						 clone_col = "CloneID", vgene_col = "VGene",
jgene_col = "JGene", isotype_col = "Isotype",
						 count_col = "Clustered", vshm_col = "V_SHM",
jshm_col = "J_SHM", cdr_col = "CDR3_AA",
						 sample_col = None, sizing_mode = "scale_width",
show_plots = True, bokeh_resources = "cdn"):
	"""Creates an interactive dashboard HTML page displaying all the comparative
visualizations.

	Parameters
	----------
	clone_dfs: pandas DataFrame or dict of {str: DataFrame}
		Input repertoire(s) with sample names; input should be formatted as a dict of
sample name: DataFrame or a single
		concatenated pandas DataFrame with a column sample_col specifying the samples of
origin
```

```
        filename: str or None
                Name for the saved output HTML file, or None if user wishes to save manually;
default is None
        title: str
                Page title for the output HTML file; default is "Repertoire Analysis Dashboard"
        plot_title_prefix: str
                Optional prefix added to the title of each plot, for example to prefix titles
with a Donor name; default is ""
        mosaic_top_clones: int
                Limit for the total clones to display for Mosaic plots (over 5000 is often
visually jarring); default is 5000
        cyrcos_top_clones: int
                Limit for the total clones to display for Cyrcos plots; default is 1000
        upset_highlighted_sets: list of tuples or None
                Specific shared sample sets to highlight in different colors for the UpSet
comparison plot; default is None
        clone_col: str
                Header / name for the column containing the sample clone IDs; default is
"CloneID"
        vgene_col: str
                Header / name for the column containing the sample V genes; default is "VGene"
        jgene_col: str
                Header / name for the column containing the sample J genes; default is "JGene"
        isotype_col: str
                Header / name for the column containing the sample isotypes; default is "Isotype"
        count_col: str
                Header / name for the column containing the sample clone counts or frequencies;
default is "Clustered"
        vshm_col: str
                Header / name for the column containing the sample clone V gene SHMs; default is
"V_SHM"
        jshm_col: str
                Header / name for the column containing the sample clone J gene SHMs; default is
"J_SHM"
        cdr_col: str
                Header / name for the column containing the sample clone CDR3 amino acid
sequences; default is "CDR3_AA"
        sample_col: str or None
                Header / name for the column containing sample names if all samples are in one
DataFrame; default is "Sample"
        sizing_mode: str
                How to scale the plots in the dashboard layout (see Bokeh layout function);
default is "scale_width"
        show_plots: bool
                Whether the dashboard page will be immediately shown upon creation; default is
True
        bokeh_resources: str
                BokehJS resource location used for the dashboard (see Bokeh output_file
documentation); default is "cdn"
                "cdn" gets the required files from the Bokeh CDN (requires internet connection)
                "inline" adds all necessary stylesheets and scripts to the HTML page itself

        Returns
        ----------
        dashboard: bokeh nested layout of Column and Row
                The output dashboard Layout object representing the final plots and their
placements
        """

        #Set up output file if user wants to save the dashboard page
        if filename is not None:
                output_file(filename = filename, title = title, mode = bokeh_resources)

        repertoire_cols = [clone_col, vgene_col, jgene_col, isotype_col, count_col, vshm_col,
jshm_col, cdr_col]
```

```
        if isinstance(clone_dfs, dict):
                if sample_col is None:
                        sample_col = "Sample"

                dfs = []
                for sample in clone_dfs:
                        clone_df = clone_dfs[sample][repertoire_cols]
                        clone_df[sample_col] = sample
                        dfs.append(clone_df)

                comparison_df = pandas.concat(dfs, ignore_index = True)

        elif isinstance(clone_dfs, pandas.DataFrame):
                if sample_col is not None:
                        repertoire_cols.append(sample_col)
                        comparison_df = clone_dfs[repertoire_cols]
                else:
                        raise KeyError("No sample-name column header was found in the repertoire
DataFrame!")

        ############################################
        ##    Paired V-J Gene Usage Donut Plots    ##
        ############################################
        vj_gene_plots = []
        for sample, df in comparison_df.groupby([sample_col]):
                plot_title = "{0} {1} Paired V-J Gene Usage".format(plot_title_prefix, sample)
                vj_gene_plot = VJ_Gene_Plot(df, title = plot_title, vgene_col = vgene_col,
jgene_col = jgene_col,
                                                                        count_col = count_col,
vgene_colors = vgene_colors, jgene_colors = jgene_colors,
                                                                        vfamily_colors =
vfamily_colors)
                vj_gene_plots.append(vj_gene_plot)

        ############################################
        ##        V/J Gene SHMs Violin Plot        ##
        ############################################
        vj_shm_plot = Violin_SHM_Plot(comparison_df, title = plot_title_prefix + " Gene SHM
Levels", vshm_col = vshm_col,
                                                                  jshm_col = jshm_col, split_col
= sample_col)

        ############################################
        ## Repertoire Clone Frequency Mosaic Plots ##
        ############################################
        mosaic_plots = []
        for sample, df in comparison_df.groupby([sample_col]):
                plot_title = "{0} {1} Clonotype Frequencies Mosaic".format(plot_title_prefix,
sample)
                mosaic_plot = Mosaic_Plot(df, title = plot_title, top_clones = mosaic_top_clones,
vgene_col = vgene_col,
                                                                  jgene_col = jgene_col,
isotype_col = isotype_col, count_col = count_col,
                                                                  vshm_col = vshm_col, jshm_col
= jshm_col, vgene_colors = vgene_colors,
                                                                  jgene_colors = jgene_colors,
vfamily_colors = vfamily_colors,
                                                                  isotype_colors =
isotype_colors)
                mosaic_plots.append(mosaic_plot)

        ############################################
        ##      Clonal V Gene SHM Burtin Plot      ##
        ############################################
        clonal_vgene_shm_plot = Burtin_VGene_SHM_Plot(comparison_df, title = plot_title_prefix + "
Clonal V Gene Mean SHM",

vgene_col = vgene_col, vshm_col = vshm_col, split_col = sample_col,

vfamily_colors = vfamily_colors)

        #############################################
        ##        Repertoire Diversity Plot        ##
        #############################################
        diversity_plot = Diversity_Plot(comparison_df, title = plot_title_prefix + " Repertoire
Diversity & Polarization",
                                                                          count_col = count_col,
split_col = sample_col)

        #############################################
        ##  CDR3 Amino Acid Length Histogram Plot  ##
        #############################################
        cdr_len_plot = CDR_Length_Histogram_Plot(comparison_df, title = plot_title_prefix + " CDR3
Length Spectratype",
                                                                                           cdr_col
= cdr_col, split_col = sample_col)

        #############################################
        ## Shared Repertoire Clonotypes UpSet Plot ##
        #############################################
        upset_plot = Repertoire_Upset_Plot(comparison_df, title = plot_title_prefix + " Shared
Clone Set UpSet Plot",
                                                                             clone_col =
clone_col, sample_col = sample_col,
                                                                             highlighted_sets =
upset_highlighted_sets)

        #############################################
        ## Shared Clone Rank/Frequency Circos Plot ##
        #############################################
        cyrcos_plot = Cyrcos_Repertoire_Comparison_Plot(comparison_df, title = " Shared Repertoire
Clonal Frequency",

        top_clones = cyrcos_top_clones, clone_col = clone_col,

        count_col = count_col, sample_col = sample_col)

        dashboard_layout = [[upset_plot.plots_grid], [vj_shm_plot, clonal_vgene_shm_plot],
[cdr_len_plot, diversity_plot]]

        #Arrange the mosaic and V-J gene plots into two columns
        mosaic_plots = [list(plots) for plots in numpy.array_split(mosaic_plots,
numpy.ceil(len(mosaic_plots) / 2))]
        vj_gene_plots = [list(plots) for plots in numpy.array_split(vj_gene_plots,
numpy.ceil(len(vj_gene_plots) / 2))]

        dashboard_layout += mosaic_plots
        dashboard_layout += vj_gene_plots

        dashboard_layout += [[cyrcos_plot.plot]]
        dashboard = layout(children = dashboard_layout, sizing_mode = sizing_mode)

        if show_plots:
                show(dashboard)
        else:
                save(dashboard)
```

References

Agematsu, K., Nagumo, H., Yang, F.C., Nakazawa, T., Fukushima, K., Ito, S., Sugita, K., Mori, T., Kobata, T., Morimoto, C. and Komiyama, A. 1997. B Cell Subpopulations Separated by CD27 and Crucial Collaboration of CD27+ B Cells and Helper T Cells in Immunoglobulin Production. *European Journal of Immunology*, 27(8): 2073-2079.

Ahuja, A., Anderson, S.M., Khalil, A. and Shlomchik, M.J. 2008. Maintenance of the Plasma Cell Pool is Independent of Memory B Cells. Proceedings of the National Academy of Sciences, 105(12): 4802-4807.

Amanna, I.J., Carlson, N.E. and Slifka, M.K. 2007. Duration of Humoral Immunity to Common Viral and Vaccine Antigens. *New England Journal of Medicine*, 357(19): 1903-1915.

Amanna, I.J. and Slifka, M.K., 2010. Mechanisms that Determine Plasma Cell Lifespan and the Duration of Humoral Immunity. *Immunological Reviews*, 236(1): 125-138.

Angelin-Duclos, C., Cattoretti, G., Lin, K.I. and Calame, K. 2000. Commitment of B Lymphocytes to a Plasma Cell Fate is Associated with Blimp-1 Expression in vivo. *The Journal of Immunology*, 165(10): 5462-5471.

Anttila, M., Voutilainen, M., Jäntti, V., Eskola, J. and Käyhty, H. 1999. Contribution of Serotype-Specific IgG Concentration, IgG Subclasses and Relative Antibody Avidity to Opsonophagocytic Activity Against Streptococcus pneumoniae. *Clinical and Experimental Immunology*, 118(3): 402.

Arce, S., Luger, E., Muehlinghaus, G., Cassese, G., Hauser, A., Horst, A., Lehnert, K., Odendahl, M., Honemann, D., Heller, K.D. and Kleinschmidt, H. 2004. CD38 low IgG-Secreting Cells are Precursors of Various CD38 high-Expressing Plasma Cell Populations. *Journal of Leukocyte Biology*, 75(6): 1022-1028.

Arnon, T.I., Horton, R.M., Grigorova, I.L. and Cyster, J.G. 2013. Visualization of Splenic Marginal Zone B-Cell Shuttling and Follicular B-Cell Egress. *Nature*, 493(7434): 684-688.

Bagnara, D., Squillario, M., Kipling, D., Mora, T., Walczak, A.M., Da Silva, L., Weller, S., Dunn-Walters, D.K., Weill J., and Reynaud, C. 2015. A Reassessment of IgM Memory Subsets in Humans. *Journal of Immunology* 195(8): 3716-3724.

Banerjee, M., Mehr, R., Belelovsky, A., Spencer, J. and Dunn-Walters, D.K. 2002. Age- and Tissue-Specific Differences in Human Germinal Center B Cell Selection Revealed by Analysis of IgVH Gene Hypermutation and Lineage Trees. *European Journal of Immunology*, 32(7): 1947-1957.

Bashford-Rogers, R.J., Palser, A.L., Huntly, B.J., Rance, R., Vassiliou, G.S., Follows, G.A. and Kellam, P. 2013. Network Properties Derived from Deep Sequencing of Human B-Cell Receptor Repertoires Delineate B-Cell Populations. *Genome Research*, 23(11): 1874-1884.

Bienvenu, J., Whicher, J., Chir, B. and Aguzzi, F. 1996. Immunoglobulins. *Serum Proteins in Clinical Medicine*, ed. Ritchie, R.F. and Navolotskaia, O. 1: 1-16.

Bolotin, D.A., Poslavsky, S., Mitrophanov, I., Shugay, M., Mamedov, I.Z., Putintseva, E.V. and Chudakov, D.M. 2015. MiXCR: Software for Comprehensive Adaptive Immunity Profiling. *Nature Methods*, 12(5): 380-381.

Boyd, S.D., Gaeta, B.A., Jackson, K.J., Fire, A.Z., Marshall, E.L., Merker, J.D., Maniar, J.M., Zhang, L.N., Sahaf, B., Jones, C.D. and Simen, B.B. 2010. Individual Variation in the Germline Ig Gene Repertoire Inferred from Variable Region Gene Rearrangements. *The Journal of Immunology*, 184(12): 6986-6992.

Boyd, S.D. and Joshi, S.A. 2015. High-Throughput DNA Sequencing Analysis of Antibody Repertoires. *Antibodies for Infectious Diseases*, ed. Rappuoli, R. 345-362.

Briney, B.S., Willis, J.R., McKinney, B.A. and Crowe, J.E. 2012. High-Throughput Antibody Sequencing Reveals Genetic Evidence of Global Regulation of the Naive and Memory Repertoires that Extends Across Individuals. *Genes and Immunity*, 13(6): 469-473.

Cambridge, G., Leandro, M.J., Edwards, J.C., Ehrenstein, M.R., Salden, M., Bodman-Smith, M. and Webster, A.D. 2003. Serologic Changes Following B Lymphocyte Depletion Therapy for Rheumatoid Arthritis. *Arthritis & Rheumatology*, 48(8): 2146-2154.

Cancro, M.P., Hao, Y., Scholz, J.L., Riley, R.L., Frasca, D., Dunn-Walters, D.K. and Blomberg, B.B. 2009. B Cells and Aging: Molecules and Mechanisms. *Trends in Immunology*, 30(7): 313-318.

Carrasco, Y.R. and Batista, F.D. 2007. B Cells Acquire Particulate Antigen in a Macrophage-Rich Area at the Boundary between the Follicle and the Subcapsular Sinus of the Lymph Node. *Immunity*, 27(1): 160-171.

Cerutti, A., Cols, M. and Puga, I. 2012. Activation of B Cells by Non-Canonical Helper Signals. *EMBO Reports*, 13(9): 798-810.

Cerutti, A., Cols, M. and Puga, I. 2013. Marginal Zone B Cells: Virtues of Innate-Like Antibody-Producing Lymphocytes. *Nature Reviews Immunology*, 13(2): 118-132.

Chao, A., Gotelli, N. Hsieh, T., Sander, E., Ma, K.H., Colwell, R.K., and Ellison, A. 2014. Rarefaction and Extrapolation with Hill Numbers: a Framework for Sampling and Estimation in Species Diversity Studies. *Ecological Monographs* 84(1): 45-67.

Chu, V.T. and Berek, C. 2013. The Establishment of the Plasma Cell Survival Niche in the Bone Marrow. *Immunological Reviews*, 251(1): 177-188.

Cobaleda, C., Schebesta, A., Delogu, A. and Busslinger, M. 2007. Pax5: The Guardian of B Cell Identity and Function. *Nature Immunology*, 8(5): 463-470.

Conway, J.R., Lex, A., and Gehlenborg, N. 2017. UpSetR: an R Package for the Visualization of Intersecting Sets and Their Properties. Bioinformatics 33(18): 2938-2940.

Crotty, S., Felgner, P., Davies, H., Glidewell, J., Villarreal, L. and Ahmed, R. 2003. Cutting Edge: Long-Term B Cell Memory in Humans after Smallpox Vaccination. *The Journal of Immunology*, 171(10): 4969-4973.

DeKosky, B.J., Ippolito, G.C., Deschner, R.P., Lavinder, J.J., Wine, Y., Rawlings, B.M., Varadarajan, N., Giesecke, C., Dörner, T., Andrews, S.F. and Wilson, P.C. 2013.

High-Throughput Sequencing of the Paired Human Immunoglobulin Heavy and Light Chain Repertoire. *Nature Biotechnology*, 31(2): 166-169.

Delogu, A., Schebesta, A., Sun, Q., Aschenbrenner, K., Perlot, T. and Busslinger, M. 2006. Gene Repression by Pax5 in B Cells is Essential for Blood Cell Homeostasis and is Reversed in Plasma Cells. *Immunity*, 24(3): 269-281.

Depoil, D., Fleire, S., Treanor, B.L., Weber, M., Harwood, N.E., Marchbank, K.L., Tybulewicz, V.L. and Batista, F.D. 2008. CD19 is Essential for B Cell Activation by Promoting B Cell Receptor–Antigen Microcluster Formation in Response to Membrane-Bound Ligand. *Nature Immunology*, 9(1): 63-72.

Dunn-Walters, D.K. and Ademokun, A.A. 2010. B Cell Repertoire and Ageing. *Current Opinion in Immunology*, 22(4): 514-520.

Dunn-Walters, D.K. 2016. The Ageing Human B Cell Repertoire: a Failure of Selection?. *Clinical & Experimental Immunology*, 183(1): 50-56.

Edwards, J.C., Szczepański, L., Szechiński, J., Filipowicz-Sosnowska, A., Emery, P., Close, D.R., Stevens, R.M. and Shaw, T. 2004. Efficacy of B-Cell–Targeted Therapy with Rituximab in Patients with Rheumatoid Arthritis. *New England Journal of Medicine*, 350(25): 2572-2581.

Ey, P. L. 1993. Use of Stable 6-aminohexyl Derivatives for Labelling Polysaccharides with Haptens and for Preparing Polysaccharide Immunoadsorbents. *Journal of Immunological Methods* 160(1): 135-137.

Ferguson, F.G., Wikby, A., Maxson, P., Olsson, J. and Johansson, B. 1995. Immune Parameters in a Longitudinal Study of a Very Old Population of Swedish People: a Comparison Between Survivors and Nonsurvivors. *The Journals of Gerontology Series A: Biological Sciences and Medical Sciences*, 50(6): B378-B382.

Fiskesund, R., Steen, J., Amara, K., Murray, F., Szwajda, A., Liu, A., Douagi, V., and Frostegard, J. 2014. Naturally Occurring Human Phosphorylcholine Antibodies Are Predominantly Products of Affinity-Matured B Cells in the Adult. *Journal of Immunology*, 192(10): 4551-4559.

Ganusov, V.V. and De Boer, R.J. 2007. Do Most Lymphocytes in Humans Really Reside in the Gut?. *Trends in Immunology*, 28(12): 514-518.

Ghetie, V. and Ward, E.S. 2000. Multiple Roles for the Major Histocompatibility Complex Class I–Related Receptor FcRn. *Annual Review of Immunology*, 18(1): 739-766.

Gibson, K.L., Wu, Y.C., Barnett, Y., Duggan, O., Vaughan, R., Kondeatis, E., Nilsson, B.O., Wikby, A., Kipling, D. and Dunn-Walters, D.K. 2009. B-Cell Diversity Decreases in Old Age and is Correlated with Poor Health Status. *Aging Cell*, 8(1): 18-25.

Gorski, J., Piatek, T., Yassai, M., Gorski, J., Maslanka, K. 1995. Improvements in Repertoire Analysis by CDR3 Size Spectratyping. *Annals of the New York Academy of Sciences*, 756(1): 99-102.

Gray, D. 2002. A Role for Antigen in the Maintenance of Immunological Memory. *Nature Reviews Immunology*, 2(1): 60-65.

Greiff, V., Bhat, P., Cook, S.C., Menzel, U., Kang, W. and Reddy, S.T. 2015. A Bioinformatic Framework for Immune Repertoire Diversity Profiling Enables Detection of Immunological Status. *Genome Medicine*, 7(1): 49.

Halliley, J.L., Tipton, C.M., Liesveld, J., Rosenberg, A.F., Darce, J., Gregoretti, I.V., Popova, L., Kaminiski, D., Fucile, C.F., Albizua, I. and Kyu, S. 2015. Long-Lived Plasma Cells are Contained within the CD19− CD38 hi CD138+ Subset in Human Bone Marrow. *Immunity*, 43(1): 132-145.

Hamada, H., Hiroi, T., Nishiyama, Y., Takahashi, H., Masunaga, Y., Hachimura, S., et al. 2002. Identification of multiple isolated lymphoid follicles on the antimesenteric wall of the mouse small intestine. *The Journal of Immunology*, 168(1), 57-64.

Heesters, B.A., Chatterjee, P., Kim, Y.A., Gonzalez, S.F., Kuligowski, M.P., Kirchhausen, T. and Carroll, M.C. 2013. Endocytosis and Recycling of Immune Complexes by Follicular Dendritic Cells Enhances B Cell Antigen Binding and Activation. *Immunity*, 38(6): 1164-1175.

Heesters, B.A., Myers, R.C. and Carroll, M.C. 2014. Follicular Dendritic Cells: Dynamic Antigen Libraries. *Nature Reviews Immunology*, 14(7): 495-504.

Heesters, B.A., van der Poel, C.E., Das, A. and Carroll, M.C. 2016. Antigen Presentation to B Cells. *Trends in Immunology*, 37(12): 844-854.

Hertz, M. and Nemazee, D. 1998. Receptor Editing and Commitment in B Lymphocytes. *Current Opinion in Immunology*, 10(2): 208-213.

Hintze, J., Nelson, R. 1998. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician*, 52(2): 181-184.

Mei, H.E., Wirries, I., Frolich, D., Brisslert, M., Giesecke, C., Grun, J.R., Alexander, T, Schmidt, S., Luda, K., Kuhl, A.A., Engelmann, R., Durr, M., Scheel, T., Bokarewa, M., Perka, C., Radbruch, A., and Dorner, T. 2015. A Unique Population of IgG-Expressing Plasma Cells Lacking CD19 is Enriched in Human Bone Marrow. *Blood*, 125: 1739 - 1748.

Hoehn, K.B., Fowler, A., Lunter, G. and Pybus, O.G., 2016. The Diversity and Molecular Evolution of B-Cell Receptors During Infection. *Molecular Biology and Evolution*, 33(5): 1147-1157.

Hoffman, W., Lakkis, F.G. and Chalasani, G. 2016. B cells, Antibodies, and More. *Clinical Journal of the American Society of Nephrology*, 11(1): 137-154.

Jost, L. 2006. Entropy and Diversity. *Oikos*, 113(2): 363-375.

Julien, J.P., Sok, D., Khayat, R., Lee, J.H., Doores, K.J., Walker, L.M., Ramos, A., Diwanji, D.C., Pejchal, R., Cupo, A. and Katpally, U. 2013. Broadly Neutralizing Antibody PGT121 Allosterically Modulates CD4 Binding via Recognition of the HIV-1 gp120 V3 Base and Multiple Surrounding Glycans. *PLoS Pathogens*, 9(5).

Kindt, T.J., Goldsby, R.A., Osborne, B.A. and Kuby, J. 2007. *Kuby Immunology*. Macmillan.

Klein, U., Rajewsky, K. and Kuppers, R. 1998. Human Immunoglobulin (Ig) M+ IgD+ Peripheral Blood B Cells Expressing the CD27 Cell Surface Antigen Carry Somatically Mutated Variable Region Genes: CD27 as a General Marker for Somatically Mutated (Memory) B Cells. *Journal of Experimental Medicine*, 188(9): 1679-1689.

Klein, U. and Dalla-Favera, R. 2008. Germinal Centres: Role in B-cell Physiology and Malignancy. *Nature Reviews Immunology*, 8(1): 22-33.

Koch, G. and Benner, R. 1982. Differential Requirement for B-Memory and T-Memory Cells in Adoptive Antibody Formation in Mouse Bone Marrow. *Immunology*, 45: 697–704.

Kolibab, K., Smithson, S.L., Rabquer, B., Khuder, S. and Westerink, M.J. 2005. Immune Response to Pneumococcal Polysaccharides 4 and 14 in Elderly and Young Adults: Analysis of the Variable Heavy Chain Repertoire. *Infection and Immunity*, 73(11): 7465-7476.

Kraus, M., Alimzhanov, M.B., Rajewsky, N. and Rajewsky, K. 2004. Survival of Resting Mature B Lymphocytes Depends on BCR Signaling via the Igα/β Heterodimer. *Cell*, 117(6): 787-800.

Kruetzmann, S., Rosado, M.M., Weber, H., Germing, U., Tournilhac, O., Peter, H.H., Berner, R., Peters, A., Boehm, T., Plebani, A. and Quinti, I. 2003. Human Immunoglobulin M Memory B Cells Controlling Streptococcus pneumoniae Infections are Generated in the Spleen. *Journal of Experimental Medicine*, 197(7): 939-945.

Laserson, U., Vigneault, F., Gadala-Maria, D., Yaari, G., Uduman, M., Vander Heiden, J.A., Kelton, W., Jung, S.T., Liu, Y., Laserson, J. and Chari, R. 2014. High-Resolution Antibody Dynamics of Vaccine-Induced Immune Responses. *Proceedings of the National Academy of Sciences*, 111(13): 4928-4933.

Leandro, M.J., Edwards, J.C., Cambridge, G., Ehrenstein, M.R. and Isenberg, D.A. 2002. An Open Study of B Lymphocyte Depletion in Systemic Lupus Erythematosus. *Arthritis & Rheumatology*, 46(10): 2673-2677.

Leinster, T. and Cobbold, C.A. 2012. Measuring Diversity: The Importance of Species Similarity. *Ecology*, 93(3): 477-489.

Liao, H.X., Lynch, R., Zhou, T., Gao, F., Alam, S.M., Boyd, S.D., Fire, A.Z., Roskin, K.M., Schramm, C.A., Zhang, Z. and Zhu, J. 2013. Co-evolution of a Broadly Neutralizing HIV-1 Antibody and Founder Virus. *Nature*, 496(7446): 469-476.

Mamani-Matsuda, M., Cosma, A., Weller, S., Faili, A., Staib, C., Garçon, L., Hermine, O., Beyne-Rauzy, O., Fieschi, C., Pers, J.O. and Arakelyan, N. 2008. The Human Spleen is a Major Reservoir for Long-Lived Vaccinia Virus–Specific Memory B Cells. *Blood*, 111(9): 4653-4659.

Manz, R.A., Lohning, M., Cassese, G., Thiel, A. and Radbruch, A. 1998. Survival of Long-Lived Plasma Cells is Independent of Antigen. *International Immunology*, 10(11): 1703-1711.

Manz, R.A., Arce, S., Cassese, G., Hauser, A.E., Hiepe, F. and Radbruch, A. 2002. Humoral Immunity and Long-Lived Plasma Cells. *Current Opinion in Immunology*, 14(4): 517-521.

Manz, R.A. and Radbruch, A. 2002. Plasma Cells for a Lifetime?. *European Journal of Immunology*, 32(4): 923-927.

Manz, R.A., Hauser, A.E., Hiepe, F. and Radbruch, A. 2005. Maintenance of Serum Antibody Levels. *Annual Review of Immunology*, 23: 367-386.

Martensson, I.L., Keenan, R.A. and Licence, S. 2007. The Pre-B-Cell Receptor. *Current Opinion in Immunology*, 19(2): 137-142.

McDaniel, J.R., DeKosky, B.J., Tanno, H., Ellington, A.D. and Georgiou, G. 2016. Ultra-High-Throughput Sequencing of the Immune Receptor Repertoire from Millions of Lymphocytes. *Nature Protocols*, 11(3): 429-442.

Mei, H.E., Wirries, I., Frölich, D., Brisslert, M., Giesecke, C., Grun, J.R., Alexander, T., Schmidt, S., Luda, K., Kühl, A.A. and Engelmann, R. 2015. A Unique Population of IgG-Expressing Plasma Cells Lacking CD19 is Enriched in Human Bone Marrow. *Blood*, 125(11): 1739-1748.

Mohr, E., Serre, K., Manz, R.A., Cunningham, A.F., Khan, M., Hardie, D.L., Bird, R. and MacLennan, I.C. 2009. Dendritic Cells and Monocyte/Macrophages that Create the IL-6/APRIL-rich Lymph Node Microenvironments where Plasmablasts Mature. *The Journal of Immunology*, 182(4): 2113-2123.

Mora, T., Walczak, A.M., Bialek, W. and Callan, C.G. 2010. Maximum Entropy Models for Antibody Diversity. *Proceedings of the National Academy of Sciences*, 107(12): 5405-5410.

Mroczek, E.S., Ippolito, G.C., Rogosch, T., Hoi, K.H., Hwangpo, T.A., Brand, M.G., Zhuang, Y., Liu, C.R., Schneider, D.A., Zemlin, M. and Brown, E.E. 2014. Differences in the Composition of the Human Antibody Repertoire by B Cell Subsets in the Blood. *Frontiers in Immunology*, 5(96).

Murphy, K. and Weaver, C. 2016. *Janeway's Immunobiology*. Garland Science.

Vancouver Park, S. and Nahm, M.H. 2011. Older Adults have a Low Capacity to Opsonize Pneumococci due to Low IgM Antibody Response to Pneumococcal Vaccinations. *Infection and Immunity*, 79(1): 314-320.

Pritz, T., Lair, J., Ban, M., Keller, M., Weinberger, B., Krismer, M. and Grubeck-Loebenstein, B. 2015. Plasma Cell Numbers Decrease in Bone Marrow of Old Patients. *European Journal of Immunology*, 45(3): 738-746.

Quail, M.A., Smith, M., Coupland, P., Otto, T.D., Harris, S.R., Connor, T.R., Bertoni, A., Swerdlow, H.P. and Gu, Y. 2012. A Tale of Three Next Generation Sequencing Platforms: Comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq Sequencers. *BMC Genomics*, 13(1): 341.

Radbruch, A., Muehlinghaus, G., Luger, E.O., Inamine, A., Smith, K.G., Dorner, T. and Hiepe, F., 2006. Competence and Competition: The Challenge of Becoming a Long-Lived Plasma Cell. *Nature Reviews Immunology*, 6(10): 741-750.

Radl, J., Sepers, J.M., Skvaril, F., Morell, A. and Hijmans, W. 1975. Immunoglobulin Patterns in Humans over 95 Years of Age. *Clinical and Experimental Immunology*, 22(1): 84-90.

Rajewsky, K., 1996. Clonal Selection and Learning in the Antibody System. *Nature*, 381(6585): 751-758.

Renyi, A. 1961. On Measures of Entropy and Information. *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability (1): Contributions to the Theory of Statistics*.

Rifai, A., Fadden, K., Morrison, S.L. and Chintalacharuvu, K.R. 2000. The N-glycans Determine the Differential Blood Clearance and Hepatic Uptake of Human Immunoglobulin (Ig) A1 and IgA2 Isotypes. *Journal of Experimental Medicine*, 191(12): 2171-2182.

Robinson, W.H. 2015. Sequencing the Functional Antibody Repertoire—Diagnostic and Therapeutic Discovery. *Nature Reviews Rheumatology*. 11(3): 171–182.

Roldan, E., Rodriguez, C., Navas, G., Parra, C. and Brieva, J.A. 1992. Cytokine Network Regulating Terminal Maturation of Human Bone Marrow B Cells Capable of Spontaneous and High Rate Ig Secretion in vitro. *The Journal of Immunology*, 149(7): 2367-2371.

Rowland, S.L., Tuttle, K., Torres, R.M. and Pelanda, R. 2013. Antigen and Cytokine Receptor Signals Guide the Development of the Naive Mature B Cell Repertoire. *Immunologic Research*, 55(1-3): 231-240.

Sanz, I., Wei, C., Lee, F.E.H. and Anolik, J. 2008. Phenotypic and Functional Heterogeneity of Human Memory B Cells. *Seminars in Immunology*, 20(1): 67-82.

Scott, D., Terrell, G. 1992. Variable Kernel Density Estimation. *The Annals of Statistics*, 1992: 1236-1265.

Scheeren, F.A., Nagasawa, M., Weijer, K., Cupedo, T., Kirberg, J., Legrand, N. and Spits, H. 2008. T Cell–Independent Development and Induction of Somatic Hypermutation in Human IgM+ IgD+ CD27+ B Cells. *Journal of Experimental Medicine*, 205(9): 2033-2042.

Slifka, M.K., Matloubian, M. and Ahmed, R. 1995. Bone Marrow is a Major Site of Long-Term Antibody Production after Acute Viral Infection. *Journal of Virology*, 69(3): 1895-1902.

Slifka, M.K., Antia, R., Whitmire, J.K. and Ahmed, R. 1998. Humoral Immunity due to Long-Lived Plasma Cells. *Immunity*, 8(3): 363-372.

Smith, K.G., Light, A., Nossal, G.J.V. and Tarlinton, D.M. 1997. The Extent of Affinity Maturation Differs between the Memory and Antibody-Forming Cell Compartments in the Primary Immune Response. *The EMBO Journal*, 16(11): 2996-3006.

Smithson, S.L., Kolibab, K., Shriner, A.K., Srivastava, N., Khuder, S. and Westerink, M.J. 2005. Immune Response to Pneumococcal Polysaccharides 4 and 14 in Elderly and Young Adults: Analysis of the Variable Light Chain Repertoire. *Infection and Immunity*, 73(11): 7477-7484.

Steens, A., Vestrheim, D.F., Aaberge, I.S., Wiklund, B.S., Storsaeter, J., Bergsaker, M.R., Ronning, K. and Furuseth, E. 2014. A Review of the Evidence to Inform Pneumococcal Vaccine Recommendations for Risk Groups Aged 2 Years and Older. *Epidemiology & Infection*, 142(12): 2471-2482.

Suzuki, K., Maruya, M., Kawamoto, S. and Fagarasan, S. 2010. Roles of B-1 and B-2 Cells in Innate and Acquired IgA-Mediated Immunity. *Immunological Reviews*, 237(1): 180-190.

Tabibian-Keissar, H., Hazanov, L., Schiby, G., Rosenthal, N., Rakovsky, A., Michaeli, M., Shahaf, G.L., Pickman, Y., Rosenblatt, K., Melamed, D. and Dunn-Walters, D. 2016. Aging Affects B-Cell Antigen Receptor Repertoire Diversity in Primary and Secondary Lymphoid Tissues. *European Journal of Immunology*, 46(2): 480-492.

Tangye, S.G., Avery, D.T., Deenick, E.K. and Hodgkin, P.D. 2003. Intrinsic Differences in the Proliferation of Naive and Memory Human B Cells as a Mechanism for

Enhanced Secondary Immune Responses. *The Journal of Immunology*, 170(2): 686-694.

Tangye, S.G. and Good, K.L. 2007. Human IgM+CD27+ B Cells: Memory B Cells or "Memory" B Cells?. *The Journal of Immunology*, 179(1): 13-19.

Tarlinton, D., Radbruch, A., Hiepe, F. and Dorner, T. 2008. Plasma Cell Differentiation and Survival. *Current Opinion in Immunology*, 20(2): 162-169.

Tavares, S.M.Q.M.C., Junior, W.D.L.B. and e Silva, M.R.L. 2014. Normal Lymphocyte Immunophenotype in an Elderly Population. *Revista Brasileira de Hematologia e Hemoterapia*, 36(3): 180-183.

Thomas, M.D., Srivastava, B. and Allman, D. 2006. Regulation of Peripheral B Cell Maturation. *Cellular Immunology*, 239(2): 92-102.

Tokoyoda, K., Hauser, A.E., Nakayama, T. and Radbruch, A. 2010. Organization of Immunological Memory by Bone Marrow Stroma. *Nature Reviews Immunology*, 10(3): 193-200.

Venturi, V., Kedzierska, K., Turner, S.J., Doherty, P.C. and Davenport, M.P. 2007. Methods for Comparing the Diversity of Samples of the T Cell Receptor Repertoire. *Journal of Immunological Methods*, 321(1): 182-195.

Victora, G.D. and Nussenzweig, M.C. 2012. Germinal Centers. *Annual Review of Immunology*, 30: 429-457.

Vidarsson, G., Dekkers, G. and Rispens, T. 2014. IgG Subclasses and Allotypes: From Structure to Effector Functions. *Frontiers in Immunology*, 5: 520.

Weill, J.C., Weller, S. and Reynaud, C.A. 2009. Human Marginal Zone B Cells. *Annual Review of Immunology*, 27: 267-285.

Weiss-Ottolenghi, Y. and Gershoni, J.M. 2014. Profiling the IgOme: Meeting the Challenge. *FEBS Letters*, 588(2): 318-325.

Weller, S., Reynaud, C., and Weill, J. 2005. Vaccination Against Encapsulated Bacteria in Humans: Paradoxes. *Trends in Immunology* 26(2): 85-89.

Wu, Y.C.B., Kipling, D. and Dunn-Walters, D.K. 2012. Age-Related Changes in Human Peripheral Blood IGH Repertoire Following Vaccination. *Frontiers in Immunology*, 3: 193.

Yaari, G. and Kleinstein, S.H. 2015. Practical Guidelines for B-Cell Receptor Repertoire Sequencing Analysis. *Genome Medicine*, 7(1): 121.

Yoshida, T., Mei, H., Dorner, T., Hiepe, F., Radbruch, A., Fillatreau, S. and Hoyer, B.F. 2010. Memory B and Memory Plasma Cells. *Immunological Reviews*, 237(1): 117-139.

Yoshikawa, T.T. and Marrie, T.J. 2000. Community-Acquired Pneumonia in the Elderly. *Clinical Infectious Diseases*, 31(4): 1066-1078.

Zandvoort, A. and W. Timens. 2002. The Dual Function of the Splenic Marginal Zone: Essential for Initiation of Anti-TI-2 Responses but Also Vital in the General First-line Defense Against Blood-Borne Antigens. *Clinical and Experimental Immunology* 130(1): 4-11.

Zhang, W., Brahmakshatriya, V. and Swain, S.L. 2014. CD4 T Cell Defects in the Aged: Causes, Consequences and Strategies to Circumvent. *Experimental Gerontology*, 54: 67-70.