8-2019

Machine Learning Techniques as Applied to Discrete and Combinatorial Structures

Samuel David Schwartz
Utah State University

MACHINE LEARNING TECHNIQUES AS APPLIED TO DISCRETE AND COMBINATORIAL STRUCTURES

by

Samuel David Schwartz

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Mathematics

Approved:

David E. Brown, Ph.D. Major Professor	Adele Cutler, Ph.D. Committee Member
Todd Moon, Ph.D. Committee Member	Richard S. Inouye, Ph.D. Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2019

ABSTRACT

Machine Learning Techniques as Applied to Discrete and Combinatorial Structures

by

Samuel David Schwartz, Master of Science
Utah State University, 2019

Major Professor: David E. Brown, Ph.D.
Department: Mathematics and Statistics

Machine learning techniques, while well established for many observation types, have only recently come onto the scene for graphs and other combinatorial objects. Further, the use and efficacy of machine learning techniques in predicting computationally difficult invariants on discrete and combinatorial objects is next to unknown. This thesis outlines methodologies useful for articulating discrete structures in the paradigm of many machine learning algorithms. Moreover, we examine several NP hard problems in different representations. We then report on the results of various techniques and methodologies in solving certain families of these problems.

(106 pages)

PUBLIC ABSTRACT

Machine Learning Techniques as Applied to
Discrete and Combinatorial Structures
Samuel David Schwartz

Machine Learning Techniques have been used on a wide array of input types: images, sound waves, text, and so forth. In articulating these input types to the almighty machine, there have been all sorts of amazing problems that have been solved for many practical purposes.

Nevertheless, there are some input types which don't lend themselves nicely to the standard set of machine learning tools we have. Moreover, there are some provably difficult problems which are abysmally hard to solve within a reasonable time frame.

This thesis addresses several of these difficult problems. It frames these problems such that we can then attempt to marry the allegedly powerful utility of existing machine learning techniques to the practical solvability of said problems.

DEDICATION

Most thesis dedications are yawn inducing. Often dedicated to the author's loved ones in a trite one liner, it's no wonder that the dedication is the most skipped over section of the typical graduate student produced manuscript. In fact, I'm shocked you're even reading this. Alas, since you're here reading this page anyway, I assure you to fear not; I fall in line and wax typically sentimental in my acknowledgements. At this time, however, I beg of you a moment of indulgence in my overtly political dedication: I dedicate this thesis to the *haves* and *have-nots*.

I am most definitely a "have". I have had a wonderful family that inculcated love of learning and discovery at a young age. I have had teachers and public school systems that fostered opportunities to explore curiosities all throughout my teenagehood. I have had a vast array of academic resources made readily available during my affordable undergraduate experience at USU. I have had a department and advisor who never gave up on me during my darkest hours as a graduate student.

Many of my fellow graduate student peers in the department are similarly situated "haves". I salute them in their diligent tackling of the unique internal and external pressures they face. From the varied complaints of undergraduate curriculum they field on a regular basis (complaints sourced from the lowliest undergraduate rabble to the lofty professor in some other department), to the abysmal pay compared with peers in industry, my fellow graduate student "haves" tolerate much and I stand with them in solidarity. It is to them I dedicate this thesis.

I dedicate this thesis equally to those who have-not. To those who were not born into a family supportive of academic endevors. To those who, by misfortunate accident of their circumstances of birth or environment of origin, did not have a background sufficient for subsequent collegiate success. To those who have-not a place at the banquet table of opportunity, I dedicate this thesis.

This is my solemn prayer: That we may utilize the gifts of machine learning to benefit the many, not the few. That the fruits of these gifts will lead to the boundless and equitable distribution of opportunity. That the paths from *have-not* to *have* may become wider and more numerous. To this dream and hope I commend and dedicate my thesis.

ACKNOWLEDGMENTS

- **David E. Brown** It is only appropriate that any acknowledgements page begin with Dave. I thank him for the tireless service he has provided over the years. A previous MS student likened Dave to “Yoda” in his thesis; the wisdom has yet to cease.

- **Adele Cutler and Todd Moon** For sticking with me from semester one till now.

- **Andreas Malmendier** For having the audacity to work late into the wee hours of the morning, and for being willing to lend a listening ear at both 3 p.m. and 3 a.m.

- **Andrew Lloyd** For being my first advisor and sounding board. Andy may be gone from this life, but his wisdom is never far from mind.

- **Brent Thomas** For speaking to me of the unspoken rules, and for having the courage to tell me what I needed to hear even when I didn’t want to hear it.

- **Katie Sweet** For being my stalwart supporter since second grade.

- **Kevin Moon** For thoughtful technical remarks, and for the opportunity to utilize his research group as a fertilizer for new ideas and directions.

- **My Family** For their unwavering dedication through thick and thin.

- **My Fellow Graduate Students** For cheering me on and teaching me the things which research and classrooms cannot. While I can’t enumerate everyone, I’d like to mention Michael Schultz, Tyler Bowles, and Kaitlin Murphy as pivotal influences during each semester I’ve been a graduate student.

- **The Department of Mathematics and Statistics** For, as an institution, allowing me the opportunity to cyclically fail, learn from the failure, and grow from it.

For all the support given, I am grateful.

Samuel David Schwartz

CONTENTS

CHAPTER 1
INTRODUCTION

Machine learning techniques are well established for many observation types and application domains, ranging from ecology to image recognition to language translation. Nevertheless, machine learning techniques have only recently come onto the scene for use in graphs and other combinatorial objects, such as matrices operating under algebraic operations outside the fields of $\mathbb{R}$ and $\mathbb{C}$.

Further, the use and efficacy of machine learning techniques in predicting computationally difficult invariants on discrete combinatorial objects is next to unknown. This thesis outlines methodologies useful for articulating discrete structures in the paradigm of many machine learning algorithms. Moreover, we examine several NP hard problems in different articulations. We then report on the results of various techniques and methodologies in solving certain families of these problems.

1.1 Motivating Problems

This thesis focuses on three exemplary decision problems:

- The Boolean Matrix Factorization Problem
- The Tournament Isomorphism Problem
- The Feedback Arcset Number Problem

These problems are selected as motivating examples since the underlying decidability of these problems lies in the computational class NP. In particular, the Boolean Matrix Factorization problem is NP Complete [1], as well as the Feedback Arcset Number Problem [2]. While the Tournament Isomorphism problem resides in DLOGSPACE (a subset of P class problems), the best known algorithm has a time complexity of $O(n^{\log_2(n)})$ [3], making the problem intractable in practical settings.

Formally, a decision problem X asks the following question: "*Given a set of n constraints, does there exist at least one solution? Yes or no?*" If X can be decided in polynomial time with respect to n, then X is said to reside in the computational complexity class P.

As a side note, we are often interested in asking subsidiary questions: Assuming the response to X is "*Yes, there exists at least one solution*", we subsequently ask, "*What is (are) the solution(s)? Further, if there are multiple solutions which satisfy the n constraints, which is the optimal solution given a particular optimality criterion?*"

Suppose X is articulated as an optimization problem and the ordering of potential solutions can be done in polynomial time. Then, as the optimal ordering of solutions is predicated on our knowledge of existence of solutions to order, X falls into the same complexity class as its underlying decision problem.

The class NP is defined as the set of decision problems which have polynomial time verifiers. A verifier is an algorithm which, when given decision problem X and additional information c (perhaps given from the oracle), can determine whether there exists at least one solution to X. In practice, this additional information c is often a solution to X itself.

As an illustrative example, consider the following articulation of the Traveling Salesman Problem: "*Given a set of cities S and routes between cities R, does there exist a sequence of routes $(r_1, r_2, \cdots, r_n)$ with $r_i \in R$ such that each city $s \in S$ is visited exactly once, save for the starting city, which is returned to at the very end of the journey?*"

We recognize this problem to be in the computational complexity class NP since, if we had a proposed solution $(r_1, r_2, \cdots, r_n)$ given by the oracle which matched the requirements of visiting each city s exactly once (save for the first city, which we return to at the conclusion of r_n), we could verify that this is in fact a solution in polynomial time.

Other variants of the Traveling Salesman Problem, such as those which apply weights to the routes, often induce a kind of optimization ordering among possible solutions. The underlying decision problem, "*Does there exist at least one solution, regardless of route weighting?*" remains the same.

There are other computational complexity classes, such as NP Hard and NP Complete, which give finer granularity to the classification of a decision problem being within NP. We refer the curious reader to [4] for a full exposition of these classes. Our immediate motivation behind examining of each of the three problems of Boolean Matrix Factorization, Tournament Isomorphism, and the Feedback Arcset Number, is due to all being decision problems (or optimization corollaries) which are computationally expensive to compute.

1.2 Motivating the Use of Existing Techniques

Much of the natural world can be framed in terms of real-valued numbers, and many problems are solvable by algorithms which work by manipulating real values under the usual operations of addition, additive negation, multiplication, and multiplicative invertibility. In other words, for many situations we can take advantage of the techniques available to us when working with an ordered field, particularly when bounded by a set of constraints given by some idiosyncratic problem.

There are many techniques we can use to solve various types of polynomial-time decision problems and their optimization corollaries. One of the more generalized ways to solve these kinds of problems is through articulation as a *Linear Program* (LP). In the next few sections we will use LP formulations, and their variants, as an analogy for the use of appropriate representations when inputting decision problems to machine learning techniques.

In canonical form, a linear program is articulated as:

$$\begin{aligned} &\text{Maximize } \vec{c} \cdot \vec{x} \\ &\text{subject to } \mathbf{A}\vec{x} \leq \vec{b} \\ &\text{and } \vec{x} \geq \vec{0}. \end{aligned}$$

Or, articulated as its dual problem:

$$\text{Minimize } \vec{b} \cdot \vec{y}$$

$$\text{subject to } \mathbf{A}^T \vec{y} \geq \vec{c}$$

$$\text{and } \vec{y} \geq \vec{0}.$$

George Dantzig's 1947 simplex method is one of the most well known algorithms for solving this problem in the general case [5]. Indeed, the simplex method is still widely used today due to its amortized efficiency, despite an exponential time complexity in the worst case.

Leonid Khachiyan developed the first polynomial time bounded algorithm in 1979 with an ellipsoid method [6]. This achievement was important for the development of the theory that the linear programming problem was provably solvable in polynomial time, although practically unimportant as the simplex method continued to outperform the ellipsoid method in all but a few edge cases.

Both methods take advantage of the geometry of the problem. As a concrete example in two dimensions, let $\mathbf{A} = \begin{bmatrix} -2 & 1 \\ 0 & 1 \\ -1 & -1 \\ 1.25 & 1 \\ -1 & 1 \end{bmatrix}$, $\vec{b} = \begin{bmatrix} 2 \\ 3.5 \\ -5 \\ 10 \\ 0 \end{bmatrix}$, and $\vec{c} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

As it happens, we can geometrically visualize the set of feasible solutions (that is, the solutions of $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ such that $\mathbf{A}\vec{x} \leq \vec{b}$ and $\vec{x} \geq \vec{0}$) as noted in Figure 1.1.

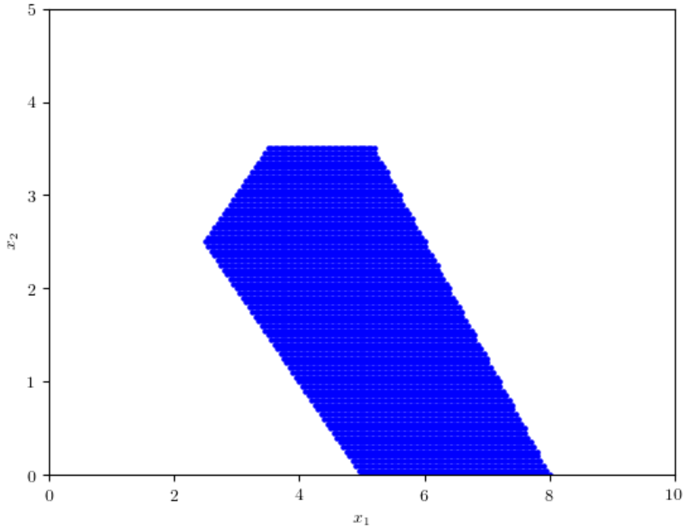


Fig. 1.1: The feasible solution space of our example linear program.

Dantzig and others determined that the solution space could always be further restricted to the vertices of these polytopes [5]. The simplex algorithm quickly finds these vertices and, in the worst case, iterates through all of them in order to find the optimal solution.

This is to say, when the constraints of some optimization problem X are describable as linear inequalities and solutions $\vec{x} \in \mathbb{R}^n$, a linear program formulation solved by Dantzig's simplex algorithm is an appropriate representation X.

1.2.1 Limitations of Existing Techniques

While linear programming can be used to obtain a solution in many types of optimization problems, there are salient limits. The linear integer programming problem is nearly identical to the generalized linear programming problem with one additional restriction: the solution $\vec{x}$ must reside in the set $\mathbb{Z}^n$. This is known as the *Integer Programming* (IP) problem.

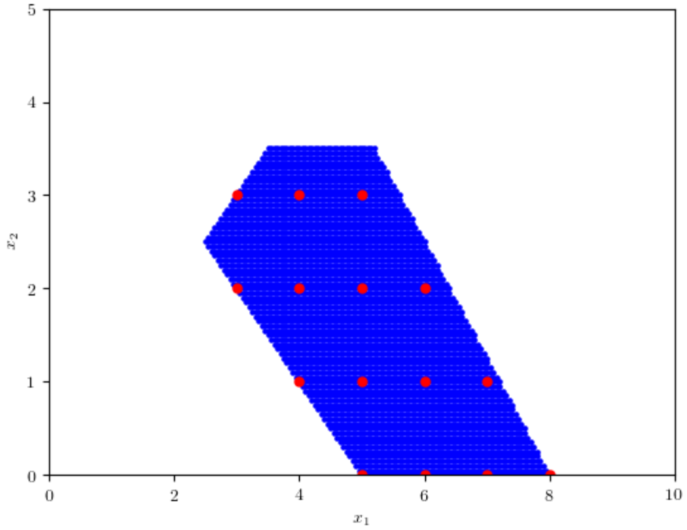


Fig. 1.2: The feasible solution space of our example linear program.

Feasible solutions for $\vec{x} \in \mathbb{R}^2$ in blue.

Feasible solutions for $\vec{x} \in \mathbb{Z}^2$ in red.

In the case illustrated in figure 1.2 we observe, given $\vec{c} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, that the optimal solution in $\mathbb{Z}^2$ is $\begin{bmatrix} 5 \\ 3 \end{bmatrix}$. However, if we were to simply round the optimal solution in $\mathbb{R}^2$ componentwise we would result in an alleged solution of $\begin{bmatrix} 6 \\ 4 \end{bmatrix}$; a solution which isn't even in our feasible region. Indeed, by observing all four of the nearest integer neighbors of the optimal linear solution it is clear that three are not feasible.

In fact, this restricted linear programming problem of finding solutions in $\mathbb{Z}^n$ has been proven to be computationally difficult to solve. It resides in the class of NP Hard, with special cases having been proved to be NP Complete, such as when integer solutions are restricted to 1s and 0s. The IP problem was included on Richard Carp's seminal list of 21 NP Complete problems in 1972 [2].

In other words, while a linear program representation may be appropriate for some kinds of problems, it may not be a sufficient representation for closely related variants (e.g., IPs) to be solved by the ellipsoid or simplex methods. Chapter 2 discusses a number of representations for matrices for input to machine learning techniques.

Some work has been done to overcome the limitations of integer programs when casting them to a reframed linear program. This is particularly true in the larger context of networks and combiniatorics where the techniques fall under the umbrella of "Fractional Graph Theory" [7]. Indeed, despite the pitfalls of relaxation described above, consider the following motivating example of fractional matching (definitions paraphrased from [7]):

> A *matching* in a graph $\mathcal{G} = (V, E)$ is a set of edges no two of which are adjacent. The *matching number* $\alpha(\mathcal{G})$ is the size of a largest matching of $\mathcal{G}$. A *fractional matching* is a function $f : E \to [0, 1]$ that assigns to each edge of a graph a number in $[0, 1]$ such that, for each vertex v, we have $\sum f(e) \leq 1$ where the sum is taken over all edges incident to v.
>
> The *fractional matching number* $\alpha_f(\mathcal{G})$ of $\mathcal{G}$ is the supremum of $\sum_{e \in E} f(e)$ over all fractional matching f.

With this framework of definitions in place, we can thus think of a matching as a $0 - 1$ program where we wish to

$$\text{maximize } \vec{c} \cdot \vec{x}$$

$$\text{subject to } A\vec{x} \leq \vec{b}$$

$$\text{with } \vec{x} \geq \vec{0}$$

where A represents the vertex–edge incidence matrix of $\mathcal{G}$, $\vec{b} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$, and $\vec{c} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$. $\vec{x} \in \{0, 1\}^n$ thus represents whether an edge indexed by column j of A is included in our matching based on whether or not $\vec{x}_j = 1$. Consequently, we see that $\alpha(\mathcal{G}) = \vec{c} \cdot \vec{x}$ when $\vec{c} \cdot \vec{x}$

is maximized. We further see the close relationship that $\alpha(\mathcal{G})$ has with $\alpha_f(\mathcal{G})$ by noting that $\alpha_f(\mathcal{G}) = \vec{c} \cdot \vec{x}$ when $\vec{x} \in \mathbb{R}^n$ and $\vec{c} \cdot \vec{x}$ is maximized.

By utilizing fractional matching (that is, a kind of relaxation) we can exploit the relationships we observe between $\alpha_f(\mathcal{G})$ and $\alpha(\mathcal{G})$ to find bounds. In [8], for example, the authors proved that $\alpha_f(\mathcal{G}) \leq \alpha(\mathcal{G}) + \frac{|V| - 2}{6}$ for a large number of graph families. Their motivation was driven, in part, by the dynamics they observed in these integer and linear programs. This is all to illustrate a larger point: by relaxation or other approximations, we may gain intuition as to the bounds of a problem. Intuition which may motivate future analytic work on a problem thought to be intractable at first brush.

1.3 Motivating the Use of Machine Learning Techniques

Even with tools utilizing integer and linear programming at our disposal, determining a solution(s) to an underlying decision problem in NP may be time consuming.

Machine learning techniques have had fantastic success in domains ranging from ecology to medicine to image recognition. Given their success in quickly predicting a wide array of natural phenomena, we will show their efficacy in predicting solutions for the three decision problems described in 1.1.

We will touch on the efficacy of many techniques, including Support Vector Machines, k Nearest Neighbors classifiers, Random Forests, and Neural Networks. As a prerequisite to this demonstration, we will provide a survey in Chapter 2 of how discrete structures can be rearticulated as input to a wide array of existing techniques.

1.3.1 Limitations of Machine Learning Techniques

Ultimately, any machine learning technique applied to a decision problem will result in heuristic predictions. There will be instances where the machine learning apparatus gives a wrong answer; perhaps correctly asserting that there does exist a solution to a given decision problem, yet simultaneously providing an alleged solution which doesn't conform to the constraints of the problem. This core issue of providing incorrect answers with a non-zero probability, as well as its derivative subproblems (such as the best way to measure accuracy of a method in the NP context), is the paramount limitation of utilizing machine learning techniques.

Nevertheless, there is precedence in utilizing probabilistic algorithms, particularly if the algorithm can be paramaterized such that the probability of error in the algorithm's response is less than the probability of hardware error on the system in which the algorithm is running [4]. One such example is the Miller–Rabin test for determining if a number $x \in \mathbb{N}$ is prime [9].

1.4 Roadmap

Looking forward, the rest of this manuscript will be divided into five areas:

1. **General Representation Techniques for Graphs and Matrices**

   Chapter Two gives a survey of existing and new ways in which discrete and combinatorial objects can be recast to different paradigms such that their representation to existing machine learning techniques is straightforward.

2. **The Boolean Matrix Factorization Problem**

   Chapter Three focuses on the Boolean Matrix Factorization problem, localizing in the space of matrices from $\{0,1\}^{3\times 3}$ as an illuminating example.

3. **The Tournament Isomorphism Problem and Feedback Arcset Problem**

   Chapter Four pivots back to problems on tournaments; namely the Isomorphism problem and Feedback Arcset problem. There, an entropy function is developed for further use in a metric. By utilizing this metric, some existing classification techniques work quite well in deciding and optimizing these exemplary problems.

4. **General Feature Selection Techniques for Real Valued Predictors**

   Chapter Five illustrates two new techniques for general unsupervised feature selection based on entropy, using the MNIST dataset as a proof of concept. The use of entropy in this chapter is inspired by results from Chapter Four.

5. **Retrospective Remarks**

   Chapter Six then concludes the manuscript by providing a brief reflection and identifies areas for future work. In doing so, the chapter also gives a brief technical summary of the main results found in this thesis.

CHAPTER 2

INPUT REPRESENTATIONS OF GRAPHS AND MATRICES TO MACHINE LEARNING ALGORITHMS

This chapter will present existent and new ways in which discrete and combinatorial objects can be represented to machine learning techniques, providing motivating examples when appropriate. In particular, we will talk about a commonly encountered algebraic structure, the *Boolean semiring*, followed by a discussion on ways to cast integers to reals and vice-versa. We then discuss matrix representations of graphs, and then how to represent those matrices as vectors for input to machine learning techniques. Subsequently, due to the large size of some of these vectors, we discuss dimensionality reduction, concluding with a technique to bijectively map a matrix from $\{0,1\}^{m\times m}$ to $n \in \mathbb{N}$.

2.1 The Boolean Semiring

Many combinatorial objects make use of the set $\{0,1\}$ to describe their structure. For example, the *adjacency matrix* of a graph, formally defined in section 2.4.1, utilizes this set to describe the presence or absence of edges between vertices.

The Boolean semiring is an algebraic structure with two operations, $+$ and $\cdot$ (or juxtaposition), operating over the set $\{0,1\}$. One of the key utilities of this algebraic structure is its relationship with the logical operators $\wedge$ and $\vee$ over the elements in the set $\{\text{TRUE}, \text{FALSE}\}$. We describe this relationship in the following tables.

Elements: {TRUE, FALSE}.

Addition Operator: $\vee$

Multiplication Operator: $\wedge$

FALSE	$\vee$	FALSE	=	FALSE
FALSE	$\vee$	TRUE	=	TRUE
TRUE	$\vee$	FALSE	=	TRUE
TRUE	$\vee$	TRUE	=	TRUE
FALSE	$\wedge$	FALSE	=	FALSE
FALSE	$\wedge$	TRUE	=	FALSE
TRUE	$\wedge$	FALSE	=	FALSE
TRUE	$\wedge$	TRUE	=	TRUE

Elements: {1, 0}.

Addition Operator: $\vee$

Multiplication Operator: $\wedge$

0	$\vee$	0	=	0
0	$\vee$	1	=	1
1	$\vee$	0	=	1
1	$\vee$	1	=	1
0	$\wedge$	0	=	0
0	$\wedge$	1	=	0
1	$\wedge$	0	=	0
1	$\wedge$	1	=	1

Elements: {1, 0}.

Addition Operator: $+$

Multiplication Operator: $\cdot$

0	+	0	=	0
0	+	1	=	1
1	+	0	=	1
1	+	1	=	1
0	$\cdot$	0	=	0
0	$\cdot$	1	=	0
1	$\cdot$	0	=	0
1	$\cdot$	1	=	1

In summary: FALSE may be mapped to 0, TRUE to 1, "OR" or $\vee$ to $+$, and "AND" or $\wedge$ to $\cdot$. We will interchange the notation for the operators as appropriate for the context. That said, we will utilize the notation $\mathbb{B}$ to describe this semiring generally.

2.2 Relaxation to $\mathbb{R}$ and "Unit Casting"

Despite the utility of the Boolean semiring, in certain situations it is also convenient to relax variables, constraints, and other articulations to the full set $\mathbb{R}$. Recall the utility of relaxing the matching of a graph as discussed in 1.2.1, for example. That is, in fractional graph theory we can take a graph and apply various algorithms which treat the existence of an edge between vertex i and j not as a value of 0 or 1, but rather as some value in $[0,1]$ to help us gain intuition or provide an approximation to a difficult optimization problem.

The usual underlying motivation for relaxations, whether in fractional graph theory or otherwise, is to employ the full power of an ordered field. This is often done by strictly relaxing an integer to its real valued analog (e.g., $2 \mapsto 2.0$ where $2 \in \mathbb{Z}$, $2.0 \in \mathbb{R}$).

In practice, however, integers in a computational problem always come from a closed and bounded subset $A \subset \mathbb{Z}$. As such, one such casting to $\mathbb{R}$ might be better articulated as a mapping $f : A \to [0,1] \subset \mathbb{R}$ where $f(\sup(A)) = 1$, $f(\inf(A)) = 0$ and $\exists k > 0$ such that $\dfrac{|f(x) - f(y)|}{|x - y|} = k$, $\forall x, y \in A$. That is, a mapping which maps the maximal value of A to 1, the minimal value of A to 0, and preserves relative linear distances between all elements of A. We will refer this type of transformation as "*Unit Casting*" or "*Unit Relaxation*" throughout the rest of this text. In particular, we will refer to the kinds of transformations embodied by f as a *casting.*

2.3 Recasting to $\mathbb{Z}$

When faced with a single element $x \in \mathbb{R}$, there are three typical ways to transform it into an integer: $\lceil x \rceil$, $\lfloor x \rfloor$, and $\text{round}(x) = \begin{cases} \lceil x \rceil & \text{decimal portion of } x > 0.5 \\ \lfloor x \rfloor & \text{otherwise} \end{cases}$.

Once again, we note that in practical cases we really can consider x to be an element of a closed and bounded subset B of $\mathbb{R}$. In considering this larger subset B, we can utilize different kinds of casting back to $\mathbb{Z}$ from $\mathbb{R}$.

2.3.1 Threshold Casting in the $[0,1]$ interval

For example, suppose we have a closed and bounded set of reals $C \subseteq [0,1] \subset \mathbb{R}$. We can create a casting f such that $f_c : C \to \{0,1\}$ by articulating f as

$$f_C(x) = \begin{cases} 1 & x > \text{threshold}(C) \\ 0 & \text{otherwise} \end{cases},$$

where threshold(C) in this case may be defined as one of the following:

- $\text{threshold}(C) = \dfrac{\sup(C) - \inf(C)}{2}$

Or, assuming C is also finite:

- $\text{threshold}(C) = \text{algebraicMean}(C)$
- $\text{threshold}(C) = \text{geometricMean}(C)$
- $\text{threshold}(C) = \text{median}(C)$
- $\text{threshold}(C) = \text{mode}(C)$

2.3.2 Distance Preserving Casting

Suppose we have an ordered, closed, bounded, and finite set $D = \{d_1, d_2, \cdots, d_n\} \subset \mathbb{R}$ such that $d_1 \leq d_2 \leq \cdots \leq d_n$. There are times where we wish to perfectly encapsulate the relative distances between each d_i and d_j when transforming each $d \in D$ to a number in $\mathbb{Z}$. One way to ensure that each element is mapped to an integer while also maintaining relative distances is by applying a suitable rescaling.

Consider the following example of one such scaling. Supposing that we have a finite amount of space to store numbers in a computational setting, we can readily restrict ourselves to the set $D \subset \mathbb{Q}$. Consequentially, let's re-write $D = \{d_1, d_2, \cdots, d_n\}$ as $D = \{\frac{p_1}{q_1}, \frac{p_2}{q_2}, \cdots, \frac{p_n}{q_n}\}$ where $p_i, q_i \in \mathbb{Z}$, $\forall i$, and p_i, q_i are relatively prime. With this assumption, we can create one such distance preserving mapping from $\mathbb{Q} \longmapsto \mathbb{Z}$ by asserting $d_i \mapsto d_i' = d_i \cdot \prod_i^n q_i$. We can re-center this distribution as needed by simply adding or subtracting an appropriate $z \in \mathbb{Z}$ to d_i for all i.

2.4 Matrix Representations of Digraphs

The primary combinatorial objects of interest to us are *digraphs*. In general, we will denote a digraph $\mathcal{G}$ as an ordered pair $\mathcal{G} = (V, E)$ where $V = \{v_1, v_2, \cdots, v_n\}$ is a set of vertices and E is a set of ordered pairs $(v_i, v_j) \in E$ such that $v_i, v_j \in V$. In this section we will describe existing and novel ways in which a digraph can be articulated as a matrix.

2.4.1 Adjacency Matrix

A digraph can be represented by a matrix in many ways. Here we will use the notation $[\mathcal{G}]$ to denote one such matrix, the *adjacency matrix* of $\mathcal{G}$. If the digraph $\mathcal{G} = (V, E)$ is unweighted, then $[\mathcal{G}] \in \{0,1\}^{|V| \times |V|}$.

In particular, the entries of $[\mathcal{G}]$ are defined as follows: $[\mathcal{G}]_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & (v_i, v_j) \notin E \end{cases}$.

In the event of a weighting $w : E \to \mathbb{R}$ on $\mathcal{G} = (V, E)$, the adjacency matrix will be given as $[\mathcal{G}] \in \mathbb{R}^{|V| \times |V|}$, with the entry at row i and column j given by $[\mathcal{G}]_{ij} = w((v_i, v_j))$.

One subtlety to note is the implicit ordering of the elements of V when creating our adjacency matrix. Given our arbitrary ordering, we can see that for any unlabeled graph we impose a labeling when we articulate it as an adjacency matrix. While for certain families of graphs we can bound the number of different adjacency matrix articulations, in general there are $|V|!$ distinct labelings for any given graph $\mathcal{G}$.

Indeed, given adjacency matrices $[\mathcal{G}_1]$ and $[\mathcal{G}_2]$, determining if $\mathcal{G}_1$ and $\mathcal{G}_2$ are in fact the same graph is known as the *Graph Isomorphism Problem*, of which our problem of interest, the Tournament Isomorphism Problem, is a special case.

2.4.2 Laplacian Matrix

A matrix similar to the adjacency matrix is the *Laplacian*. For our purposes, let the graph $\mathcal{G} = (V, E)$ have the Laplacian matrix $L(\mathcal{G})$ given by $L(\mathcal{G}) = D - [\mathcal{G}]$ where

$$D = \begin{bmatrix} \deg(v_1) & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \deg(v_n) \end{bmatrix}.$$

This matrix articulation has several nice linear-algebraic features, such as being positive semidefinite. A survey of the properties of the Laplacian can be found in [10].

2.4.3 Planar Embeddings of a Graph as Set of Matrices

A graph is considered planar if there exists a drawing in the plane of the graph's vertices and edges such that no edges cross. Determining whether a graph is planar is a special case of computing the graph's *genus*.

The genus of a graph evokes the definition of genus in a topological setting. Genus is defined in the graph theoretic context as the minimal integer n such that the graph can be drawn without crossing itself on a sphere with n handles.

The question of "*Is graph $\mathcal{G}$ planar?*" is identical to asking "*Is the genus of graph $\mathcal{G}$ 0?*". While this question is answerable in $O(n)$ time, as first demonstrated by Hopcroft and Tarjan with an algorithm in 1974 [11], the generalized question, "*Is the genus of graph $\mathcal{G}$ g, for $g \in \mathbb{N}$?*" was proved to be NP Complete by Thomassen in 1989 [12].

There are five commonly used families of polynomial time graph visualizations which, when taken together, may be utilized as image inputs to a machine learning apparatus. Many machine learning techniques have their basis in image recognition. All of the visualization algorithms noted below are deterministic in nature. Combining all, or some subset of, these embeddings as a joint image may render fruit when attempting to fit image oriented machine learning techniques to graph inputs.

Our motivation, therefore, in discussing these visualization algorithms is due to the realization that, through stitching the resultant images together, we can create a larger, image-based predictor set for a graph. Hence, we can use machine learning techniques optimized for images on these particular graph visualizations.

2.4.3.1 Orthogonal Layouts

One of the key areas of research done is in the area of orthogonal layouts. Here, the motivation is in printed computer chip layout: given many edges representing silicone electrical paths, chip manufactures have a vested interest in designing circuit-boards with the minimal number of crossings (and therefore circuit-board layers) required. These techniques are well summarized by Abboud et al. in their 2008 literature review paper [13].

2.4.3.2 Spring Based Layouts

Spring based layouts stem from a physical analogy where edges of a graph are imagined as tension and extension springs. The graph is thought of as a system in a vacuum and left to revert to equilibrium. The algorithms find the positions of these springs and uses this physical analogy to then plot the graph in the plane. Force, entropy, and energy layouts are variants on this idea of using attributes of physical systems coming to equilibrium to visualize graphs.

As a motivating example, consider a labeled digraph D on 15 vertices under two commonly used force based algorithms: GraphVis' `neato` and `fdp` [14], as seen in figures 2.1 and 2.2.

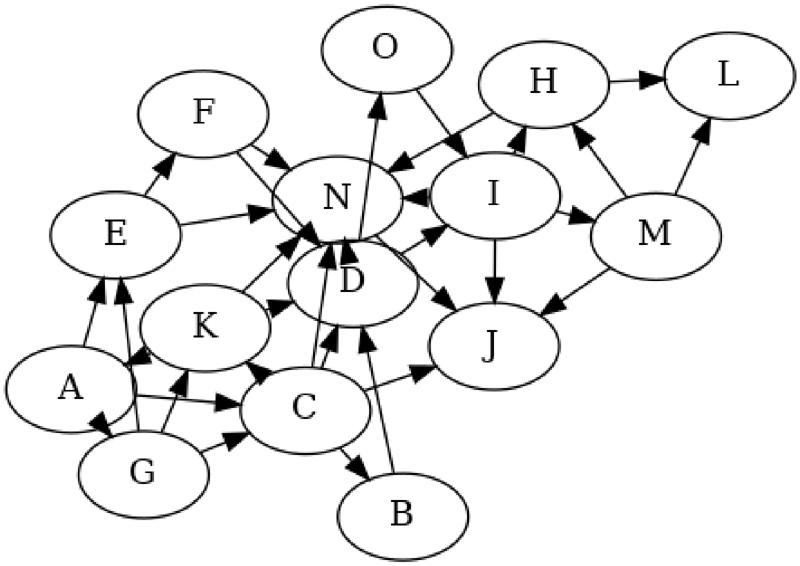


Fig. 2.1: `neato` applied to D

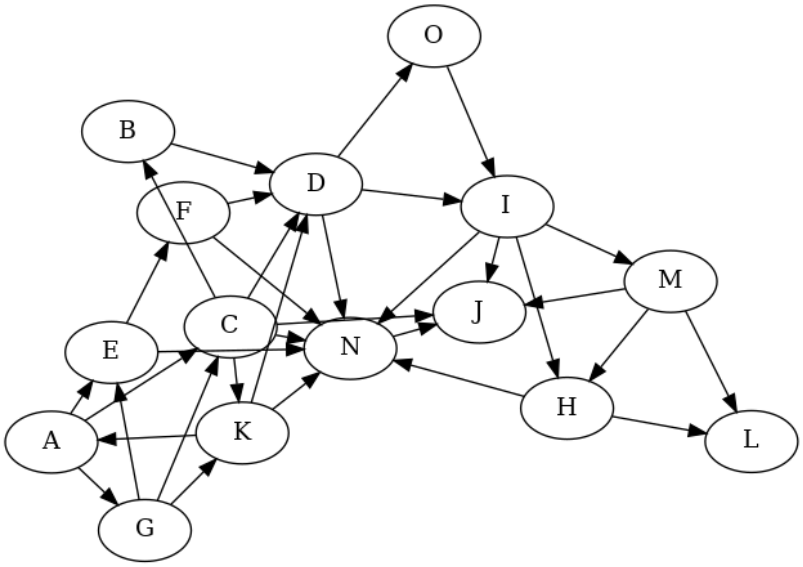


Fig. 2.2: `fdp` applied to D

2.4.3.3 Layered Layouts

A layered approach attempts to partition partially ordered data (such as the nodes in a mostly acyclic graph) into distinct layers and then visualize that data in a quasi-sequential way. Much of the work done is this arena was undertaken by Warfield in 1977 [15], Carpano in 1980 [16], and Sugiyama in 1981 [17].

The `dot` program formally described in [14] utilizes these methods and can be seen in figure 2.3.

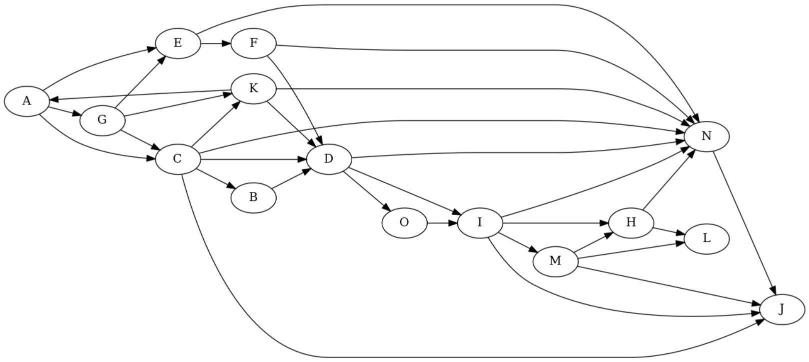


Fig. 2.3: `dot` applied to D

2.4.3.4 Circular Layouts

Circular layouts, as the name implies, utilize radial symmetries in their layout policies. In general, circular layouts either place nodes along a circumference or select a single node as a central point and attempt to create a packing of nodes analogous to that of a sphere, usually by some sort of concentric approach. This latter layout is often seen in genealogical graphs or other multi-tree variants.

Two algorithms which implement circular or radial layouts include `circo` and `twopi`, also described in [14]. They are shown in figures 2.4 and 2.5, respectively.

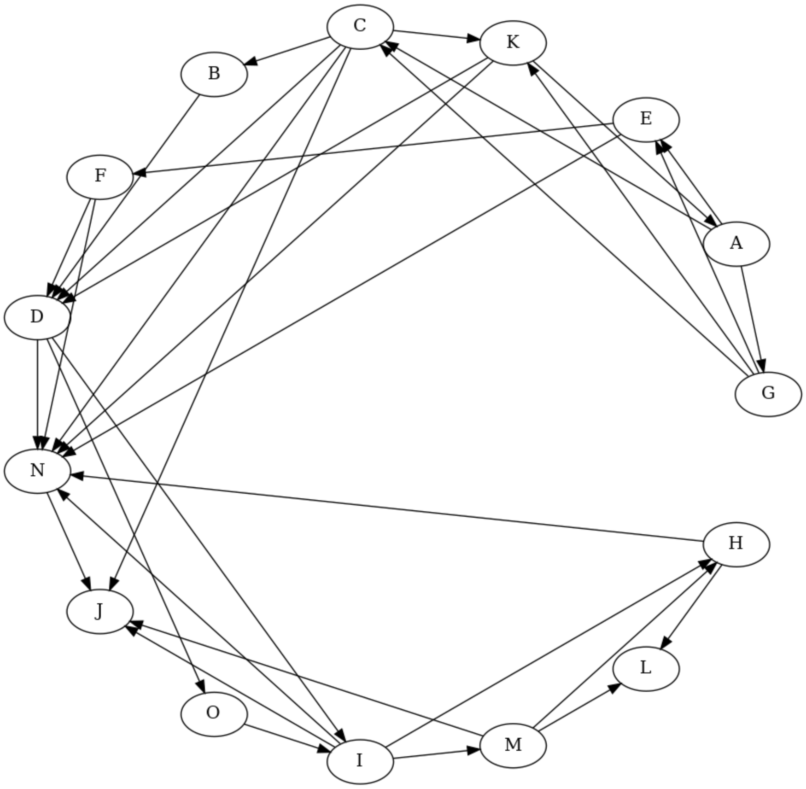


Fig. 2.4: `circo` applied to D

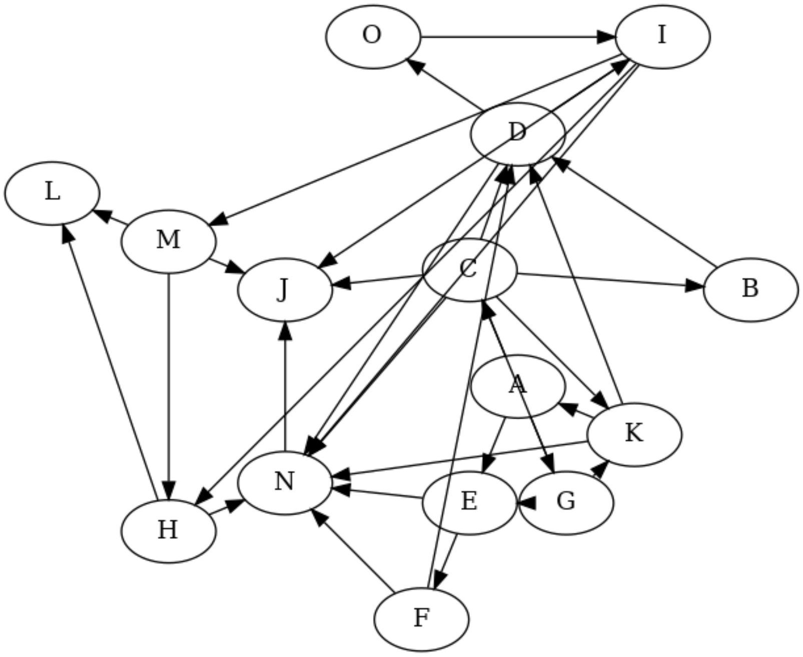


Fig. 2.5: `twopi` applied to D

2.4.3.5 Spectral Layouts

Spectral layouts are analogous to principal component analysis visualizations, in that they utilize the eigenvectors and eigenvalues of either the adjacency or Laplacian matrix of $\mathcal{G}$ to plot nodes in the Cartisian plane.

2.5 The Motivation for the Vectorization of Matrices

While there exist some machine learning techniques which are able to directly use a matrix from $\mathbb{R}^{n \times m}$ as input, these techniques nearly universally stem from an image-centric motivation. Treating a graph's adjacency matrix as analogous to a matrix describing pixels in an image can be dangerous. As a motivating example, consider the grayscale image in figure 2.6:

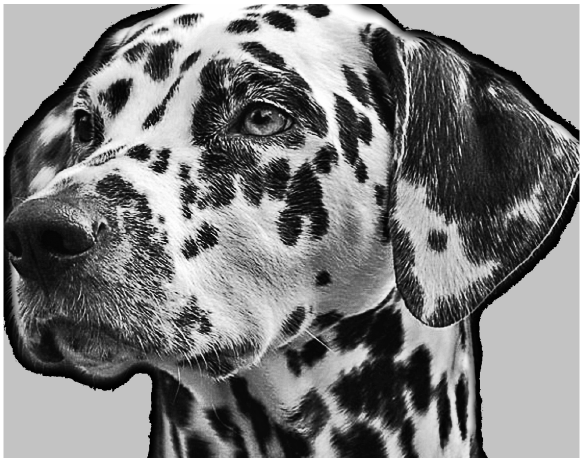


Fig. 2.6: A grayscale image.

This figure is represented by a matrix of pixel values. Here is a sampling of them:

$$\begin{bmatrix} 54 & 48 & 39 & 28 & 22 & 15 & 20 & 23 & 21 \\ 40 & 36 & 30 & 24 & 23 & 20 & 26 & 34 & 33 \\ 35 & 33 & 31 & 26 & 27 & 26 & 30 & 34 & 33 \\ 38 & 42 & 44 & 38 & 35 & 28 & 26 & 31 & 30 \\ 45 & 54 & 59 & 51 & 43 & 29 & 20 & 16 & 15 \\ 43 & 52 & 59 & 55 & 49 & 35 & 24 & 16 & 13 \\ 31 & 39 & 49 & 50 & 52 & 44 & 35 & 28 & 22 \\ 22 & 28 & 34 & 39 & 48 & 53 & 48 & 39 & 29 \\ 23 & 23 & 26 & 30 & 39 & 47 & 48 & 42 & 37 \end{bmatrix}$$

Grayscale values $[0, 255]$ of the small, centered area outlined in red in the image below.

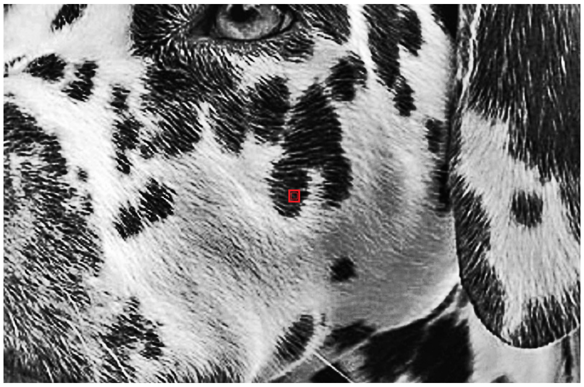


Fig. 2.7: The pixels noted in the matrix above are outlined by the very small red box.

When we take the values of a particular area and plot them, we see how there appears to be a kind of continuity. We can see this visually in the following surface plot:

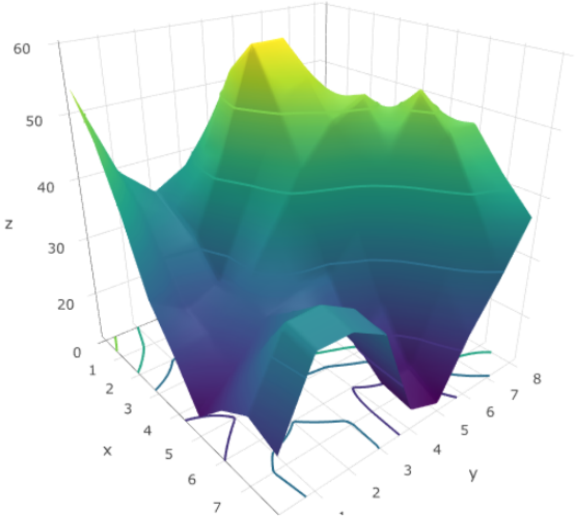


Fig. 2.8: The matrix values visualized as a surface plot. The x axis represents the row; y axis indicates the column; z axis the matrix entry at the xth row and yth column.

Contrast this with the following graph in the Peterson family with random weights applied to the edges.

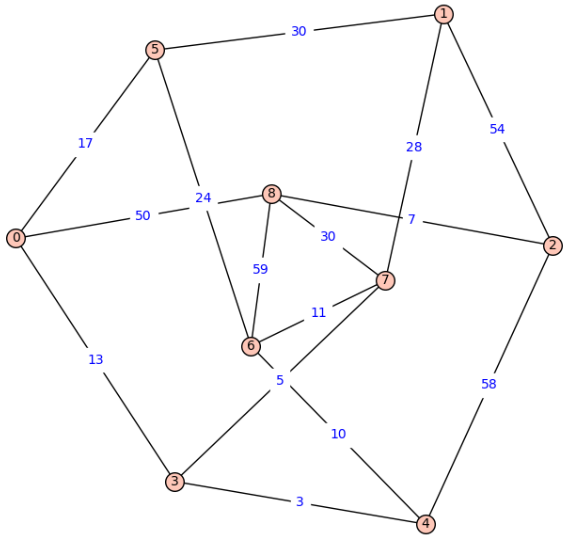


Fig. 2.9: The 9-vertex graph from the Petersen family of graphs, with weights taken from a uniform distribution over $[1, 60] \subset \mathbb{N}$

The graph embedded in figure 2.9 has the adjacency matrix

$$\begin{bmatrix} 0 & 0 & 0 & 13 & 0 & 17 & 0 & 0 & 50 \\ 0 & 0 & 54 & 0 & 0 & 30 & 0 & 28 & 0 \\ 0 & 54 & 0 & 0 & 58 & 0 & 0 & 0 & 7 \\ 13 & 0 & 0 & 0 & 3 & 0 & 0 & 5 & 0 \\ 0 & 0 & 58 & 3 & 0 & 0 & 10 & 0 & 0 \\ 17 & 30 & 0 & 0 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 & 24 & 0 & 11 & 59 \\ 0 & 28 & 0 & 5 & 0 & 0 & 11 & 0 & 30 \\ 50 & 0 & 7 & 0 & 0 & 0 & 59 & 30 & 0 \end{bmatrix},$$

and its surface plot shown in figure 2.10:

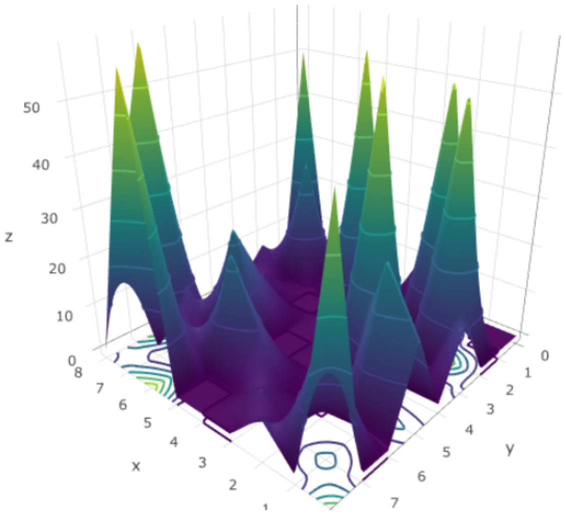


Fig. 2.10: The surface plot of the 9-vertex graph's adjacency matrix. The x axis represents the row; y axis indicates the column; z axis the matrix entry at the xth row and yth column.

Images taken of natural phenomena tend to have smooth surface plots based on the pixel matrices. Surface plots of the adjacency or Laplacian matrices of graphs, by contrast, tend to have very "spiked" surface plots.

This should not come as a great surprise: in the case of an image, the rows and columns of an image's pixels have relative meaning. By rearranging the rows and columns of an image described by matrix M through multiplication by PMP^T, where P is a permutation matrix, can erase any meaning in the image. A graph's adjacency matrix, by contrast, utilizes the rows and columns to convey information as well, but the relative distance of those rows and columns from each other matter little. Rearranging the rows and columns of adjacency matrix M by PMP^T results in the same graph up to isomorphism; the information is preserved.

Hence, many existing machine learning techniques which take in matrices as input, such as convolutional neural networks, are ill equipped to deal with matrices based on graphs. There is an implicit assumption that relative distance between data entries is important; an assumption which is simply not true in the case of graphs.

This is just one among many reasons as to why we should be skeptical when applying machine learning techniques optimized for images to matrices representing graphs. Since many machine learning techniques which take as input matrices from $\mathbb{R}^{n\times m}$ were designed for images, and noting those techniques are largely restricted to convolutional neural network approaches, we might first reach for a representation of a graph in $\mathbb{R}^n$. This is to maximize the availability of off-the-shelf machine learning algorithms open for our use.

Many machine learning techniques require input to be articulated as a single, real valued vector. Given that simply reshaping an $n\times m$ matrix to a vector of shape $n{\cdot}m\times 1$ could result in practical limitations, we have a strong motivation to find representations of graphs and matrices which could act as reasonable vector inputs to a machine learning apparatus. The next several subsections describe ways to do so. Each of the vector representations below has limitations, but each also has potential to be an apt set of characteristic descriptors of graphs and matrices for certain contexts.

2.6 Converting Matrices Representing Digraphs to Vectors

In the following subsections we describe several ways through which we convert matrices (with a focus on matrices which represent digraphs) to vectors. Vectors which are then able to be inputted as training data for machine learning algorithms.

2.6.1 Remark on Vectors in $\mathbb{C}^n$

As we begin to discuss transformation of a graph or matrix to a vector $\vec{v}$, we note an area of caution. Namely, we may fall into a problem if our entries of $\vec{v}$ are complex, given that we need to arrive at a real valued vector for input into many machine techniques. There are a number of ways to preform this conversion from $\mathbb{C}^n$ to $\mathbb{R}^n$, the most convenient of which is to apply some function $f:\mathbb{C}\to\mathbb{R}$ elementwise. One such function is $f(z) = |z|^2$, where $z = x + y\iota$ and $|z| = \sqrt{x^2+y^2}$, with ι denoting the imaginary unit.

2.6.2 Naive Singular Value Representation

Consider the graph $\mathcal{G} = (V, E)$ as an element in $V \times E \subset V \times (V \times V)$, where $E \subseteq V \times V$. One can think of the adjacency matrix or Laplacian of $\mathcal{G}$ as a bijective mapping of $V \times (V \times V) \to \mathbb{R}^{|V| \times |V|}$.

One vector invariant of $\mathcal{G}$ is to take the singular value decomposition of this $\mathbb{R}^{|V| \times |V|}$ matrix. This will result in a set of ordered pairs $(\sigma_1, \vec{v}_1), (\sigma_2, \vec{v}_2), \cdots, (\sigma_{|V|}, \vec{v}_{|V|})$, where σ_i is a singular value and $\vec{v}_i$ is the corresponding singular vector. We're motivated to look at the singular values and vectors in particular due to their relationship with principal component analysis, as noted in Chapter 10 of [18].

We can apply an ordering such as $(|\sigma_1|)^2 \geq (|\sigma_2|)^2 \geq \cdots \geq (|\sigma_{|V|}|)^2$. In the event that $(|\sigma_i|)^2 = (|\sigma_j|)^2$ for $i \neq j$, a secondary ordering based on the magnitude of $(\vec{v}_k \cdot \vec{v}_k)^2$ (where $\cdot$ is the dot-product) can be imposed. In creating this ordered sequence of singular values and singular vectors, we are then able to create a vector of the form $\begin{bmatrix} \sigma_1 \\ \vec{v}_1 \\ \sigma_2 \\ \vec{v}_2 \\ \vdots \\ \sigma_{|V|} \\ \vec{v}_{|V|} \end{bmatrix}$.

2.6.3 Ordered Singular Value Representation

Given that the number of elements in the vector described in 2.6.2 may be large, consider a subset of elements of this larger vector. Instead, utilize only the elements directly derived from the σ_is. For example, the vector $\begin{bmatrix} |\sigma_1|^2 \\ |\sigma_2|^2 \\ \vdots \end{bmatrix}$ where σ_is are ordered strictly by their squared magnitude.

2.6.4 Melting Variants

Melting is a data reshaping technique used in many data cleansing applications (see R's `reshape2` melt function).

Consider a matrix of the following form:

$$M = \begin{array}{cc} & \begin{array}{ccccc} c_1 & c_2 & c_3 & \cdots & c_n \end{array} \\ \begin{array}{c} r_1 \\ r_2 \\ r_2 \\ \vdots \\ r_m \end{array} & \begin{bmatrix} a & b & c & \cdots & i \\ d & e & & & \vdots \\ f & & \ddots & & \vdots \\ \vdots & & & \ddots & x \\ j & \cdots & \cdots & y & z \end{bmatrix} \end{array}$$

where $a, b, c, \cdots, x, y, z \in \mathbb{R}$.

Melting this matrix reshapes the data to a sequence of vectors of the form

$$M' = \begin{array}{c} \begin{array}{ccc} r & c & M_{r,c} \end{array} \\ \begin{bmatrix} [1 & 1 & a] \\ [1 & 2 & b] \\ & \vdots & \\ [m & n & z] \end{bmatrix} \end{array}$$

That is, given a matrix $M \in \mathbb{R}^{m \times n}$, we define

$$M' = \begin{bmatrix} 1 & 1 & M_{11} \\ & \vdots & \\ i & j & M_{ij} \\ & \vdots & \\ m & n & M_{mn} \end{bmatrix}$$

Articulating our data in this way allows us to treat each row of our new matrix as an ordered triple. For example, suppose M represents a digraph's adjacency matrix. In looking at a pair of connected vertices such as $V_i \xrightarrow{M_{ij}} V_j$ we have the following ordered triple: $(i,\ j,\ M_{ij})$. This is a useful articulation, as it allows for the reduction of rows in simple graphs by ensuring there does not exist the situation where there are two rows of the form $[i, j, M_{ij}]$ and $[j, i, M_{ji}]$ where $M_{ij} = M_{ji}$; only one is needed. Hence, just under half our matrix is eliminated (the cases where $i = j$ remaining).

This, of course, is simply a particular articulation of E. Specifically, it's a representation when E is thought of as the set of edges when articulated by two vertices, and the associated weight. Further simplifications can result when we restrict ourselves to the unweighted case, where $M_{ij} = 1$ when $v_i v_j \in E$ and 0 otherwise. One such example is when we eliminate all rows of the form $[i, j, M_{ij} \neq 1]$ and then truncate the last column leaving a matrix consisting only of rows in the form $[i, j]$. Such a matrix can be readily cast to a vector in $\mathbb{C}^{|E|}$, where each element of such a vector takes the form $i + j\iota$, where ι is the imaginary unit. This vector in $\mathbb{C}^{|E|}$ can then be cast to a vector in $\mathbb{R}^{|E|}$ by applying any given function $f : \mathbb{C} \to \mathbb{R}$ elementwise.

This notion can be generalized by applying any function $f : \mathbb{N}^2 \to \mathbb{R}$ row-wise to the initially given $[i, j]$ rows.

2.6.5 Melted Singular Value Representation

Many weighted graphs have sparse adjacency matrices. Consequentially, the resultant melted matrix with rows of the form $[i, j, M_{ij}]$, where rows when $M_{ij} = 0$ are not included, is considerably smaller than an initially given weighted adjacency matrix $M \in \mathbb{R}^{n \times n}$.

Consider the melted matrix

$$M' = \begin{bmatrix} [a, b, M_{ab}] \\ [c, d, M_{cd}] \\ \cdots \\ [y, z, M_{yz}] \end{bmatrix}$$

such that rows where $M_{ij} = 0$ are excluded. Assume that $M' \in \mathbb{R}^{n-1 \times 3}$.

This melted matrix M' can then be re-imagined as a tridiagonal matrix T of the form

$$T = \begin{bmatrix} \chi & b & & & \mathbf{0} \\ a & M_{ab} & d & & \\ & c & M_{cd} & \ddots & \\ & & \ddots & \ddots & z \\ \mathbf{0} & & & y & M_{yz} \end{bmatrix}$$

where χ is a free variable which might be populated with any value in $\mathbb{R}$. Some common choices might include values from the set $\{0, 1, |E|, |V|\}$ or graph invariants easily determined from M.

The advantage of having a matrix articulated in this fashion is that the number of elements is bounded above by $(2|E|)^2$ for undirected graphs, which is often considerably smaller than the sparse, $|V|^2$ element-sized adjacency matrix.

Once we have T, we can now find its singular values and singular vectors. Utilizing the techniques in 2.6.2, we arrive with a vector $\vec{v}$ which links to the graph initially given by M. The key caveat is that, as a tridiagonal matrix, there exist algorithms, usually utilizing techniques first pioneered in the Ehrlich-Aberth iterative algorithm [19,20], which can find these singular values and singular vectors very efficiently.

Naturally, the dimensionality of T will vary based on the sparsity of M; if M has lots of zeros, the dimensionality of T may be small. Given a set of Ts with differing dimensions, however, one need only pad with zeros to result in vectors $\vec{v}$ of the same size.

2.6.6 Polynomial Representations of Graphs and Matrices as Vectors

Many structures can be considered invariant under various families of polynomials. In knot theory, for example, the Jones polynomial comes to mind. In the case of graphs, the characteristic polynomial (which is the same as the characteristic polynomial of its adjacency matrix) is a well utilized, exemplary polynomial.

These polynomials are often motivated in their definition by the information they convey. The characteristic polynomial of a square matrix, for example, has eigenvalues as roots and the trace and determinant as coefficients. Bottom line: these polynomials pack a lot of information.

Suppose we have some easily computed polynomial $P(t)$ with a finite number of coefficients in $\mathbb{C}$. Right off the bat, we have a set $C = \{\text{coefficients of } P(t)\} \subset \mathbb{C}$ handed to us on a platter. In addition, we can also utilize algorithms, such as Ehrlich-Albert [19], to find the $\{\text{roots of } P(t)\} = R$ and $\{\text{maximums, minimum, and inflection points of } P(t)\} = M$ quickly; the latter by finding the roots of $P'(t)$.

Given that C is finite, and therefore bounded, so are R and M also finite and bounded. Suppose that $C = \{c_1, c_2, \cdots, c_i\}$, $R = \{r_1, r_2, \cdots, r_j\}$, and $M = \{m_1, m_2, \cdots, m_k\}$. We can then arrive with a vector $\vec{v} = [c_1, \cdots, c_i, \bar{*}, r_1, \cdots, r_j, \bar{*}, m_1, \cdots, m_k, \bar{*}]^T \in \mathbb{C}^n$. Here $\bar{*}$ represents an array of placeholders (such as a sequence of 0s) to ensure that $\vec{v}$ has the same number of elements across all objects in our universe for which we are deriving polynomials.

Given this vector $\vec{v} \in \mathbb{C}^n$, we can now use any function $f : \mathbb{C} \to \mathbb{R}$ of our liking to cast $\vec{v}$ to $\mathbb{R}^n$ by applying f elementwise.

2.7 Dimensionality Reduction

In the last several sections, we have discussed various ways to map graphs to real valued matrices and real valued vectors. Our overarching motivation: the articulation of graphs so that they could be readily used as training and testing data for existing machine learning techniques. Nonetheless, these methods frequently result in dimensions of vectors and matrices which are too large for practical training. Dimensionality reduction is needed. We discuss existing approaches below.

2.7.1 A Warning on Convolutional Approaches and Pitfalls

For large, sparse matrices which are characteristic of many types of simple graphs, it is tempting to utilize dimensionality reducing techniques made for images. One such approach is applying a sequence of filters over the matrix as in a convolutional layer in a neural network. While this may be enticing at the outset, it relies on, for a few layers, the underlying relationship between the relative positioning of the data within the matrix.

This idea alludes back to the aforementioned idea in section 2.5 that a permutation matrix applied to an image can distort the image beyond human recognition. At the same time, a permutation matrix applied to a graph's adjacency matrix is an isomorphism. Analogously, a filter over an image may preserve or enhance key features, such as edges. Yet over an adjacency matrix such a filter has the potential to destroy any semblance of recoverable structure of a graph. In utilizing convolutional or filtering techniques on adjacency matrices, one should tread with great caution.

2.7.2 Dimension Selection

In representing objects as vectors of the form $\vec{v} \in \mathbb{R}^n$, we may find that n is prohibitively large. Although the goal is to then find some reduced vector $\vec{v'} \in \mathbb{R}^m$ such that $m \ll n$, many dimensionality reducing algorithms require m as an initial parameter. One way to determine m is by utilizing PCA, LASSO, or other algorithms which give an indication of predictor importance or underlying dimensionality in either a supervised or unsupervised setting. Often, this will result in an "L" shaped curve: one with an elbow.

Consequentially, one way to select for m is to take an i.i.d. sample of vectors, apply the predictor-importance or dimensionality-reduction algorithm in question on all n components for this subset, determine the elbow point, and set m equal to that elbow point. This approach, while more computationally intense than other rules of thumb such as simply using the floor of $\log(n)$ or $\sqrt{n}$ for m, may result in better accuracy for more sensitive data outcomes due to the heuristic prediction of the number of latent and expressed variables underlying the data.

2.7.3 Existing Techniques

It should be noted that linear techniques via PCA and its variants are at the forefront to reduce the dimensionality of these kinds of vectors. In data exploration particularly, the ability to visualize these graphs by plotting the two principle components of some set of vectors is invaluable.

That said, non-linear techniques and manifold learning algorithms, such as t-SNE [21] and Isomap [22], are quickly coming to the scene, with one such example being Moon et al.'s PHATE [23]. These techniques have particular promise for graphs which are sequenced as time series, with edges being added or deleted with each discrete time step. A full survey of commonly used techniques can be found in [22].

2.8 Lossless Compressive Traversal of Matrices in $\mathbb{Z}_n$

At this point, much of our energy has been spent describing various mappings from matrices to vectors. One concern, however, is that vectors and matrices are computationally expensive to store, no matter how small dimensionally they are. It would be convenient if there was a bijective mapping f such that $f : \mathbb{Z}_n^{m \times m} \to \mathbb{N}$, particularly when $n = 2$. That is, a mapping which encapsulated a square matrix with elements in $\{0, 1\}$ as a single natural number.

Further, this mapping should stay consistent regardless of dimension, given 0 padding. That is, a mapping such that matrix $\begin{bmatrix} M \end{bmatrix} \mapsto n \in \mathbb{N}$ and $\begin{bmatrix} & - & \vec{0} & - \\ | & & & \\ \vec{0} & & M & \\ | & & & \end{bmatrix} \mapsto n \in \mathbb{N}$.

One way to do so is as follows. Consider the natural numbers $0, 1, 2, 3, \cdots$ in binary:

- 000000000
- 000000001
- 000000010
- 000000011

If we traverse over a matrix in a particular way, we have a bijection between the natural numbers and Boolean matrices via the natural number's binary representation. For example, consider the number 42. 42 is written as 000101010 in binary. We can then insert each digit into the matrix according to a traversal of our choosing. For example, as follows:

$$42 \to 000101010 \to \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

This type of traversal is seen more clearly in the following case when we have a matrix with 16 entries and a corresponding binary number with 16 digits. That is, some number n in base 10 gets mapped to some binary number B, where B is defined as a sequence of digits $B = b_{16}b_{15}b_{14}\cdots b_1$; $b_i \in \{0,1\}$.

$$n \mapsto B = b_{16}b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1 \mapsto \begin{bmatrix} b_{16} & b_{15} & b_{13} & b_{11} \\ b_{14} & b_9 & b_8 & b_6 \\ b_{12} & b_7 & b_4 & b_3 \\ b_{10} & b_5 & b_2 & b_1 \end{bmatrix}$$

More formally, this nested zig-zag mapping from $x \in \mathbb{N}$ to a matrix M can be given as follows:

```
input: natural number $x$;
output: Boolean matrix $M$;

Let $b$ be $x$ articulated as a binary number.
Let $Q$ be a queue composed of the digits of $b$.
Let $d$ be the dimension of $M$. That is, $M \in \{0,1\}^{d \times d}$.
Set $d = \lfloor \sqrt{|Q|} \rfloor$
Let $M$ be initialized.
for $i$ in $1...d$:
    for $j$ in $1...d$:
        if i=j:
            $M_{ij} = Q$.pop()
        else:
            $M_{ij} = Q$.pop()
            $M_{ji} = Q$.pop()
return $M$
```

The inverse, mapping from $M \to b \to x$, has a similar algorithm:

```
input: Boolean matrix $M$ of size $d \times d$;
output: natural number $x$;

Initialize $Q$ as an empty queue.
for i in 1...d
    for j in 1...d:
        if i=j:
            $Q$.append($M_{ij}$)
        else:
            $Q$.append($M_{ij}$)
            $Q$.append($M_{ji}$)

Let $b$ be a binary number, with digits given by the items in $Q$.
Let $x$ be the natural number corresponding with $b$.
return $x$
```

The consequence of this nested zig-zag traversal and rudimentary lossless compression results in several properties, including:

- Each Boolean matrix can be compressed and totally recovered from a single natural number.
- The Boolean matrix associated with each natural number is consistent from dimension to dimension: you only need to pad the rows or columns with zeros.
- This indexing has a several graph-theoretic properties. For example, the sequence of matrices indexed by $0, 1, \cdots, n$ forms a Hamiltonian path in the hypercube created by the set of Boolean matrices as vertices.
- The number of elements which differ between matrices indexed by i and $i \pm 1$ is 1.
- This technique can be generalized to any matrix with elements in $\mathbb{Z}_n$ by considering a conversion from base n to base m, where $m \ll n$.

CHAPTER 3

BOOLEAN MATRIX FACTORIZATION

One of the first mathematical tasks we learn in elementary school is how to factor numbers. For example, 10 has factors: $1, 10, 2, 5$. This concept of factorization can be extended to any algebraic structure for which a binary multiplicative operator, $\cdot$, is defined.

The first time we see a factorization of an object outside of $\mathbb{Z}$ is typically in a foundational linear algebra class, where we are taught how to factor (also said by some authors "to decompose") some matrix $C \in \mathbb{R}^{n \times m}$ into two other matrices A and B such that $A \cdot B = C$. There are a wide variety of ways to do this when the entries of the matrix are from $\mathbb{R}$: LU factorization, QR, and so on. The rest of this chapter will focus on matrix factorizations when matrix elements are from $\mathbb{B}$.

3.1 Motivations

Our primary motivation for creating a matrix factorization algorithm when elements are sourced from $\mathbb{B}$ is due to the problem's NP hardness [1]. As such, it takes $O(\ell \cdot 2^{m \cdot n})$ operations to find solutions for ℓ matrices of size $m \times n$ in the general case. Consequentially, it would be nice to have a solution which, in the amortized case, is solvable in polynomial time given a large number of matrices to factor.

An optimization corollary of Boolean matrix factorization is the determination of *Boolean rank*. The Boolean rank of matrix C is the minimal k such that $A \cdot B = C$ for $A \in \mathbb{B}^{m \times k}$, $B \in \mathbb{B}^{k \times n}$, and $C \in \mathbb{B}^{m \times n}$. If we were to find all $A_1 B_1, \cdots, A_z B_z$ factorizations of a matrix C in polynomial time, we could then find the Boolean rank of C in polynomial time by observing the k_i for each $A_i B_i$ factorization and taking the minimum. Furthermore, Boolean rank has several practical applications as noted in [24].

3.2 Formal Problem Statement and Definitions

As a reminder from 2.1, let $\mathbb{B}$ denote the Boolean semiring; $\mathbb{B} = (\{0,1\}, +, \cdot)$ where $+$ and $\cdot$ denote the usual Boolean operations over the elements in $\{0,1\}$. That is, $1+1=1$ and the other operations as in the reals. Consequentially, there is no subtraction, nor is there division.

For our purposes, we are going to restrict our factorization to a strictly square case, as this allows us some computational flexibility. In all cases, we can simply treat existent non-square matrices by padding rows or columns with zeros until we reach square dimensions.

Hence, our problem is as follows: Given a matrix C such that $C \in \mathbb{B}^{n \times n}$ for some $n \in \mathbb{N}$, determine matrices A, $B \in \mathbb{B}^{n \times n}$ such that $AB = C$.

As it turns out, this matrix multiplication problem is really a Boolean satisfiablity problem in disguise. Specifically, we recall that each element C_{ij} in C is defined as

$$C_{ij} = \sum_{\ell=1}^{n} A_{i\ell} \cdot B_{\ell j}$$

when working over the reals.

The binary operators $\cdot$ and $+$ can be articulated in terms of $\wedge$ and $\vee$. Assume $TRUE \iff 1$ and $FALSE \iff 0$. Notice that $x \cdot y = 1 \iff x \wedge y = 1$, when x, y are treated as logical propositions. Likewise, $x + y = 1 \iff x \vee y = 1$.

Therefore, we can treat

$$C_{ij} = \sum_{\ell=1}^{n} A_{i\ell} \cdot B_{\ell j}$$

as

$$C_{ij} = \bigvee_{\ell=1}^{n} (A_{i\ell} \wedge B_{\ell j})$$

3.2.1 Lemma: Square Boolean Matrix Decomposition is a SAT Problem

Let $\phi(x_1, \cdots, x_n)$ be a boolean formula. ϕ is said to be satisfiable if there are values for $x_1, \cdots, x_n$ such that $\phi(x_1, \cdots, x_n) = TRUE$.

Further, consider three square Boolean matrices, A, B, C, each of size $n \times n$.

Let $AB = C$ and let the entries of C be known.

Given $C_{ij} = \bigvee_{\ell=1}^{n} (A_{i\ell} \wedge B_{\ell j})$, we can consider $\phi_{ij} = \bigvee_{\ell=1}^{n} (A_{i\ell} \wedge B_{\ell j})$.

From De Morgan's laws, we can observe the following:

$$\begin{aligned}
\phi_{ij} &= \bigvee_{\ell=1}^{n} (B_{i\ell} \wedge B_{\ell j}) \\
\neg\phi_{ij} &= \neg \left(\bigvee_{\ell=1}^{n} (A_{i\ell} \wedge B_{\ell j}) \right) \\
\neg\phi_{ij} &= \left(\bigwedge_{\ell=1}^{n} \neg (A_{i\ell} \wedge B_{\ell j}) \right) \\
\neg\phi_{ij} &= \left(\bigwedge_{\ell=1}^{n} (\neg A_{i\ell} \vee \neg B_{\ell j}) \right)
\end{aligned}$$

At this point we note that if ϕ_{ij} is satisfiable, then $C_{ij} = 1$. Similarly, if $\neg\phi_{ij}$ is satisfiable, then $C_{ij} = \neg 1 = 0$.

Naturally, for a decomposition to exist, ϕ_{ij} or $\neg\phi_{ij}$ (depending on whether $C_{ij} = 0$ or $C_{ij} = 1$) must be satisfiable over all i, j.

This can be articulated by the formula

$$\phi_C(A_{11}, \cdots, A_{nn}, B_{11}, \cdots, B_{nn}) = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \begin{cases} \bigvee_{\ell=1}^{n} (A_{i\ell} \wedge B_{\ell j}) & C_{ij} = 1 \\ \bigwedge_{\ell=1}^{n} (\neg A_{i\ell} \vee \neg B_{\ell j}) & C_{ij} = 0 \end{cases}$$

Hence, if $\phi_C(A_{11}, \cdots, A_{nn}, B_{11}, \cdots, B_{nn})$ is satisfiable, then a solution of $\phi_C(A_{11}, \cdots, A_{nn}, B_{11}, \cdots, B_{nn})$ articulates a decomposition of $C = AB$.

3.3 Defining Metrics: Datasets, Sample Spaces, Expected Values, and Accuracy Evaluation

In the subsequent subsections we outline how we will sample data and assess accuracy.

3.3.1 The Dataset and Sample Spaces

In coming up with any predictive algorithm which does not yield exact results, we have a number of questions we wish to answer:

- What does it mean to ensure our data is i.i.d.?
- Is our data set ordinal or nominal?
- How can we best visualize our data?

For our data set, we will consider all 3×3 Boolean matrices.

For any matrix of dimension $m \times m$, there are m^2 entries. If this square matrix is composed of elements from Z_n , then there must be n^{m^2} distinct matrices, as each element in the matrix can be one of n items. Hence, in our case we have $2^{3^2} = 512$ square Boolean matrices in our data set.

In section 2.8, we discussed a bijective way to enumerate the n^{m^2} matrices in an $m \times m$ matrix with elements from $\mathbb{Z}_n$. One way to ensure that Boolean matrices are sampled i.i.d., then, is to sample an integer z from a uniform distribution in the range $[0, 2^{m^2} - 1]$. That is, in our 3×3 case, from $[0, 511]$. Our i.i.d. selected matrix $M \in \mathbb{B}^{3\times 3}$ is then the Boolean matrix associated with z.

As z is clearly an element from an ordered set while M may appear to be strictly nominal, this gives rise to a question: "*How do we treat our data? Ordinally or nominally?*" Since z was constructed with the caveat that $|z - z'| = 1 \iff M$ differs from M' by only one element, we can assert that our data can be seen as both ordered and nominal data: the data forms a poset.

3.3.2 Visualization

One way to visualize this data, then, is to treat the axis as zs and the response value as a colorized version of z. That is, let A_z be the Boolean matrix associated with z via our traversal in 2.8. Suppose $A_i \cdot B_j = C_k$. Just as each pixel in an image can be treated as the tuple $(x, y, value)$, we can create the pixel (i, j, k). In the 3×3 Boolean case, then, we can create an image 512 pixels high and 512 wide, as shown below:

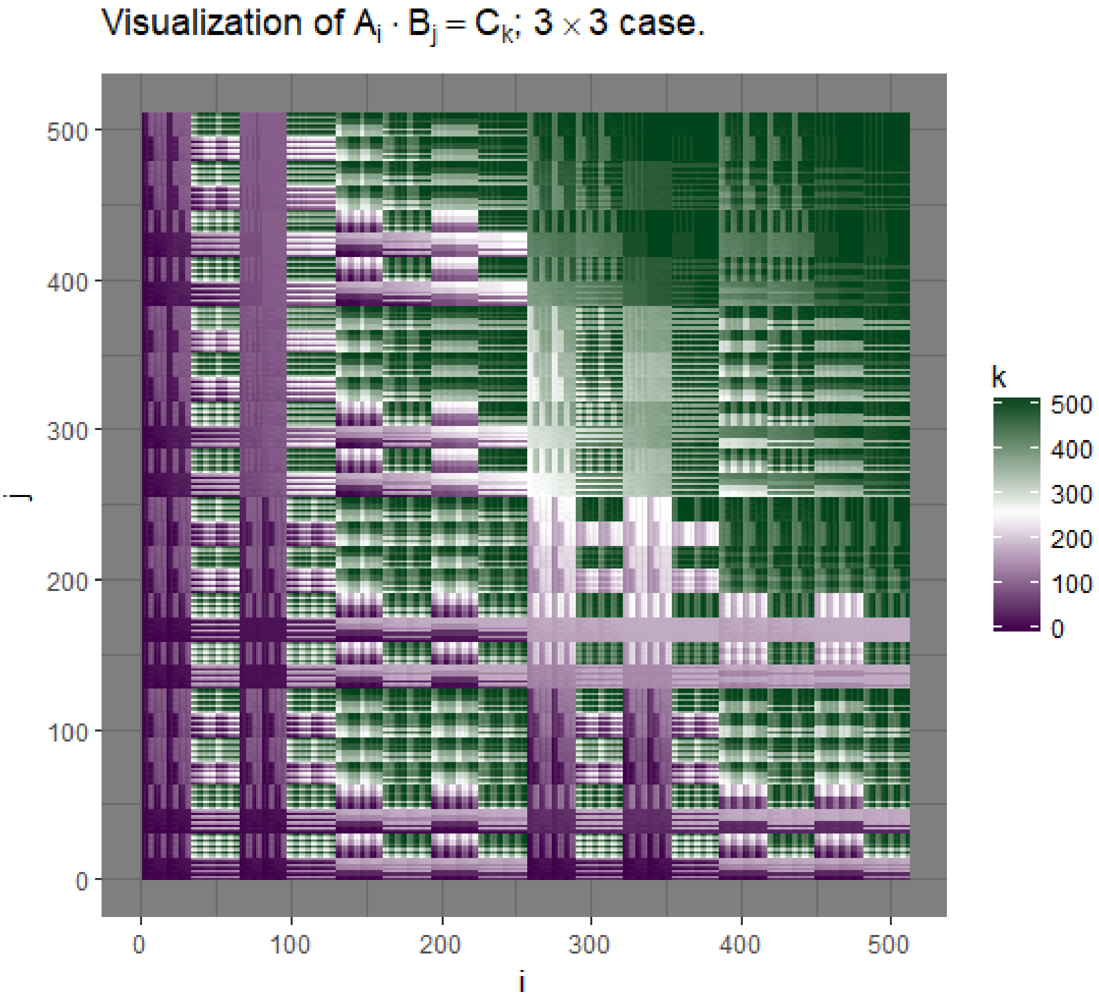


Fig. 3.1: A visualization of all possible matrix multiplications in $\mathbb{B}^{3\times3}$.

3.3.3 Image Completion Approach

Given that we can articulate our problem of matrix factorization as an image, we might consider the idea of image completion. Consider the following motivating example using human faces:

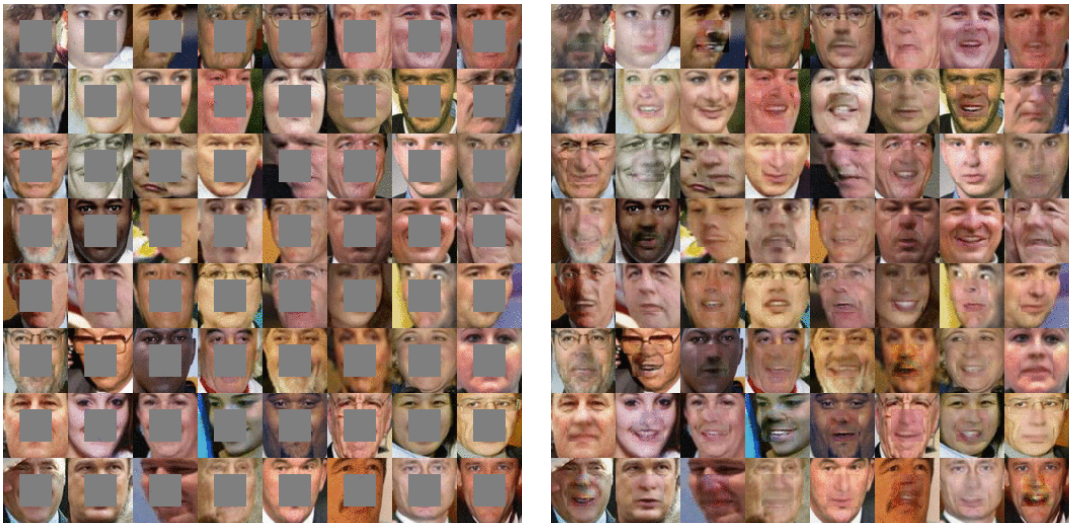


Fig. 3.2: Faces automatically completed by a neural network.

This completion was done using a *deep convolutional generative adversarial network* (DCGAN) first pioneered in 2015 by Radford et al. [25]

Our motivation is similar. Given a partially complete image which represents the data of Boolean matrix products, can we recover the rest of the image?

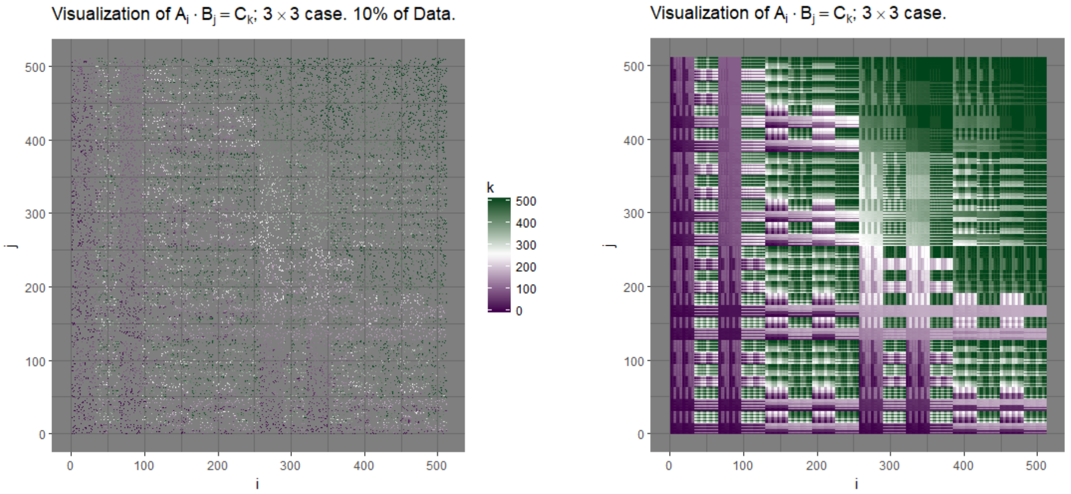


Fig. 3.3: From an image which is only partially complete, can we recover the full image?

For example, if we compute the matrix products for some small number of matrices in our universe, and then utilize those products to predict the rest of the image, we will be able to do a traversal of the image to quickly determine the factorizations of all Boolean matrices. Formally, our process looks like this:

1. Determine by brute force the matrix products for $x\%$ of all matrices. This becomes our training data.

2. Train a machine learning apparatus on the brute-forced data.

3. Quickly predict the rest of the image with high accuracy. "Quickly" being relative to the time required to brute force. All factorizations for all matrices are thus known.

3.3.4 Accuracy

Paramount to our success is defining what accuracy looks like. In this regard, we will consider two metrics for defining accuracy: *absolute accuracy* and *granular accuracy.*

While formally defined below, we consider the motivation for absolute accuracy to be the accuracy when the matrix predicted is exactly the correct matrix. Granular accuracy, by contrast, represents the percentage of entries in the matrix correctly predicted.

3.3.4.1 Absolute Accuracy

Suppose we have Boolean matrices $A, B, C, D, \cdots$. Further, suppose we have predicted Boolean Matrices $A', B', C', D', \cdots$.

Let $f(X, X') = \begin{cases} 1 \iff X_{ij} = X'_{ij} \quad \forall i, j. \\ \\ 0 \text{ otherwise.} \end{cases}$.

We define absolute accuracy as the arithmetic mean of $f(A, A'), f(B, B'), f(C, C'), f(D, D'), \cdots$.

3.3.4.2 Granular Accuracy

Suppose we have Boolean matrices $A, B, C, D, \cdots$. Further suppose we have predicted Boolean Matrices $A', B', C', D', \cdots$. Each matrix is of dimension $n \times n$.

Let $g(X, X') = \frac{1}{n^2} \sum_{i,j} \begin{cases} 1 \iff X_{ij} = X'_{ij} \\ \\ 0 \text{ otherwise.} \end{cases}$.

We define granular accuracy as the arithmetic mean of $g(A, A'), g(B, B'), g(C, C'), g(D, D'), \cdots$.

3.3.4.3 Expected Values

With accuracy defined as either absolute or granular, we can thus determine the expected accuracy of two Boolean matrices N and M when N, M are selected from a uniform distribution. That is, if $N, M \in \mathbb{B}^{n \times n}$, what is $E[f(N, M)]$ and $E[g(N, M)]$?

Claim: $E[f(N, M)] = \frac{1}{2^{n^2}}$.

Rationale:

If N, M are sampled i.i.d., then the probability of $N_{i,j} = M_{i,j}$ for any i, j is 0.5. If there are n^2 entries, then the likelihood they all are equal is given by $\prod^{n^2}(0.5) = \frac{1}{2^{n^2}}$.

Claim: $E[g(N, M)] = \frac{1}{2}$.

Rationale:

If N, M are sampled i.i.d., then the probability of $N_{i,j} = M_{i,j}$ for any i, j is 0.5. Hence, the arithmetic mean that any one of them are equal is given by $\frac{1}{n^2}(\sum^{n^2} 0.5) = \frac{1}{2}$.

Consequentially, a machine learning apparatus must perform with granular accuracy greater than 0.5 and absolute accuracy better than $\frac{1}{2^{3^1}} = \frac{1}{512} \approx 0.002$ to be doing better than chance in the $\mathbb{B}^{3\times3}$ case.

3.4 Models

We showed an example in 3.3 of how Radford et al. [25] utilized a *Deep Convolutional Generative Adversarial Network* (DCGAN) to do image completion. This approach, while inspirational, is not quite applicable to the problem here. For example, Radford had access to multiple images to use for training. For each dimension n, we only have one image. Further, the images representative of objects in the natural world have the kind of partial continuity noted in 2.5. Our sole image doesn't have that.

Still, the results of DCGAN motivates the idea that utilizing some sort of machine learning apparatus to construct an image is feasible. We utilized two techniques: (1) an automatically tuned neural-network and (2) an off-the-self random forest from Python's `Scikit-Learn` package with 100 trees per forest. While we hope that the off-the-self random forest is self-explanatory (and if not, details can be found at [26]), allow us to describe our network.

3.4.1 Auto-Tuned Neural Network

One of the most curious things about neural networks is how mediocrely they preform if the hyperparameters are not quite right. How many hidden layers should we use? What optimizer is the best? Before describing the hyperparameter search space, allow us to describe what remains constant among all networks considered. Each layer in the network (except for the final layer) uses ReLUs[1] as the activation function. Each layer is fully connected to its adjacent layers. The loss function is binary crossentropy[2], and, as an expedient to time, training terminates after 5 epochs. Dropout is not utilized. Initial weights on all layers are drawn from a uniform distribution.

[1] Rectified Linear Unit. Defined as the function $f(x) = \max(0, x)$.

[2] Binary cross entropy is also called "log loss" by some authors.

If the desired output of the network is an integer from a subset of $\mathbb{Z}$ (say, the subset $[0, 2^{3^2} - 1] = [0, 511]$), the network predicts that integer by way of one-hot encoding[3]. The final activation layer of the network is the softmax function, with the returned integer being the argmax of the returned output vector.

If the desired output of the network is a vector of Booleans, the network predicts that vector by utilizing a "hard sigmoid" function. The hard sigmoid function is a linear approximation to the sigmoid function, defined as $\sigma(x) = \begin{cases} 0 & x < -2.5 \\ 1 & x > 2.5 \\ 0.2x + 0.5 & -2.5 \leq x \leq 2.5 \end{cases}$. By using this linear approximation, we significantly speed up training time. The returned vector rounded all network outputs to the nearest 0 or 1.

Thus, the hyperparamaters up for consideration were batch size, optimizers, hidden layers, and hidden layer sizes. In the case of batch size, the following options were given: $\{16, 32, 64, 130, 260, 522\}$. These were chosen by considering sufficiently large batch sizes in the log space from 10^0 to 10^e.

The possible optimizers considered were Stochastic Gradient Descent, RMSprop, Adadelta, Adam, and Nesterov-Adam. Optimization implementation details can be found in [27].

The number of hidden layers considered were 0, 1, and 2. The length of each of these hidden layers was given as a linear spacing between the first and last layer. That is, if the input layer had x nodes and the output layer had y nodes, then if there was 1 hidden layer, that hidden layer had $\lfloor \frac{x+y}{2} \rfloor$ nodes. In the event of two hidden layers, the number of nodes in each layer was given by $\lfloor \frac{1}{3} \cdot (x + y) \rfloor$ and $\lfloor \frac{2}{3} \cdot (x + y) \rfloor$ nodes, ordered so that the layer sizes were always monotonic.

Hence, given six batch sizes, six possible optimizers, and three hidden layer options, there were are possible total of 108 network models to choose from. Including even more options, such as trying different loss functions, would have resulted in an impermissibly large search space.

[3] One-hot encoding is also called "dummy coding" by some authors.

As is, we utilized the hyperparameter package Talos to subset our options further [28]. Talos utilizes linear correlation of hyperparamaters to do a restricted grid search. Hence, of these 108 network models, we cut off our grid search after 10 models were evaluated with training data. The best model of these ten was returned as the network to use for prediction on testing data.

3.4.2 Model Inputs and Outputs

As noted in Chapter 2, there are a wide array of ways to articulate Boolean matrices in terms of graphs. We considered four input representations.

- One articulation is simply to utilize the integer associated with the nested zig-zag traversal described in section 2.8, z. This can be placed in vector form for two matrices A, B as $[z_A, z_B]^T$. We will denote this encoding as $\mathbf{z}$.

- Another articulation is to express z as a binary number; essentially a particular traversal of the Boolean matrix. Hence, one representation is $[asBinary(z_A), asBinary(z_B)]^T$. We will denote this encoding as $\mathbf{b}$.

- The third articulation we will consider is that described in section 2.6.2, the Naive Singular Value Representation. As in the above, the final vector will be composed of the juxtaposition of the two vectors associated with A and B. We will denote this encoding as $\mathbf{n}$.

- The last articulation we will consider is that described in section 2.6.3, the Ordered Eigenvalue Representation. The final vector will also be composed of the juxtaposition of the two vectors associated with A and B. We will denote this encoding as $\mathbf{e}$.

Given $\mathbf{z}$ and $\mathbf{b}$ are bijective with the matrix they represent, we will utilize those possibilities as target outputs in our models.

3.4.3 Model Descriptions and Names

Given four input representations, two output representations, and two underlying models (auto-tuned neural network and 100-tree random forest), we have $4 \cdot 2 \cdot 2 = 16$ larger models to consider. We label these models from A to P as in table 3.1:

Model Name	**Inputs**	**Outputs**	**Underlying ML Technique**
A	**r**	**r**	Auto-Tuned Neural Network
B	**r**	**b**	Auto-Tuned Neural Network
C	**b**	**r**	Auto-Tuned Neural Network
D	**b**	**b**	Auto-Tuned Neural Network
E	**e**	**r**	Auto-Tuned Neural Network
F	**e**	**b**	Auto-Tuned Neural Network
G	**n**	**r**	Auto-Tuned Neural Network
H	**n**	**b**	Auto-Tuned Neural Network
I	**r**	**r**	100-Tree Random Forest
J	**r**	**b**	100-Tree Random Forest
K	**b**	**r**	100-Tree Random Forest
L	**b**	**b**	100-Tree Random Forest
M	**e**	**r**	100-Tree Random Forest
N	**e**	**b**	100-Tree Random Forest
O	**n**	**r**	100-Tree Random Forest
P	**n**	**b**	100-Tree Random Forest

Table 3.1: Model Names and Descriptions.

3.5 Results

We considered five training-validation splits, where the data was sampled i.i.d.: 10% of data used for training, 30%, 50%, 70%, and 90%. Given that there were $512 \cdot 512 = 262144$ total samples, this corresponded to the breakdown of training-prediction counts shown in table 3.2:

Percentage of data used for validation	**No. of samples used for training**	**No. of samples predicted**
0.9	26215	235929
0.7	78644	183500
0.5	131072	131072
0.3	183501	78643
0.1	235930	26214

Table 3.2: Training – Validation Splits

This results in a total of 80 experiments to run across 16 major model types (A – P), of which the eight neural networks consider 108 separate hyperparameter configurations each.

Model	**In**	**Out**	**Underlying ML Technique**	**Granular Acc.**	**Absolute Acc.**
A	**r**	**r**	Auto-Tuned Neural Network	0.6025	0.0792
B	**r**	**b**	Auto-Tuned Neural Network	0.6339	0.0810
C	**b**	**r**	Auto-Tuned Neural Network	0.8704	0.3541
D	**b**	**b**	Auto-Tuned Neural Network	0.8608	0.3078
E	**e**	**r**	Auto-Tuned Neural Network	0.6316	0.0904
F	**e**	**b**	Auto-Tuned Neural Network	0.6624	0.0841
G	**n**	**r**	Auto-Tuned Neural Network	0.8132	0.2890
H	**n**	**b**	Auto-Tuned Neural Network	0.7806	0.1555
I	**r**	**r**	100-Tree Random Forest	0.8186	0.2691
J	**r**	**b**	100-Tree Random Forest	0.8444	0.2799
K	**b**	**r**	100-Tree Random Forest	0.9660	0.7305
L	**b**	**b**	100-Tree Random Forest	0.9997	0.9973
M	**e**	**r**	100-Tree Random Forest	0.6180	0.0850
N	**e**	**b**	100-Tree Random Forest	0.6587	0.0811
O	**n**	**r**	100-Tree Random Forest	0.8256	0.3308
P	**n**	**b**	100-Tree Random Forest	0.8543	0.3400

Table 3.3: Results of predicting 90% of the data when training on 10%.

Model	**In**	**Out**	**Underlying ML Technique**	**Granular Acc.**	**Absolute Acc.**
A	**r**	**r**	Auto-Tuned Neural Network	0.5350	0.0258
B	**r**	**b**	Auto-Tuned Neural Network	0.6046	0.0827
C	**b**	**r**	Auto-Tuned Neural Network	0.9983	0.9853
D	**b**	**b**	Auto-Tuned Neural Network	0.9173	0.5284
E	**e**	**r**	Auto-Tuned Neural Network	0.6329	0.0908
F	**e**	**b**	Auto-Tuned Neural Network	0.6627	0.0844
G	**n**	**r**	Auto-Tuned Neural Network	0.8296	0.3543
H	**n**	**b**	Auto-Tuned Neural Network	0.8076	0.2091
I	**r**	**r**	100-Tree Random Forest	0.8522	0.3397
J	**r**	**b**	100-Tree Random Forest	0.8775	0.3586
K	**b**	**r**	100-Tree Random Forest	0.9881	0.8965
L	**b**	**b**	100-Tree Random Forest	1.0000	0.9999
M	**e**	**r**	100-Tree Random Forest	0.6237	0.0888
N	**e**	**b**	100-Tree Random Forest	0.6620	0.0839
O	**n**	**r**	100-Tree Random Forest	0.8466	0.3981
P	**n**	**b**	100-Tree Random Forest	0.8722	0.4095

Table 3.4: Results of predicting 70% of the data when training on 30%.

Model	**In**	**Out**	**Underlying ML Technique**	**Granular Acc.**	**Absolute Acc.**
A	**r**	**r**	Auto-Tuned Neural Network	0.5784	0.0744
B	**r**	**b**	Auto-Tuned Neural Network	0.6157	0.0708
C	**b**	**r**	Auto-Tuned Neural Network	0.9668	0.8122
D	**b**	**b**	Auto-Tuned Neural Network	0.9033	0.4700
E	**e**	**r**	Auto-Tuned Neural Network	0.6255	0.0906
F	**e**	**b**	Auto-Tuned Neural Network	0.6623	0.0833
G	**n**	**r**	Auto-Tuned Neural Network	0.8078	0.2775
H	**n**	**b**	Auto-Tuned Neural Network	0.7929	0.1770
I	**r**	**r**	100-Tree Random Forest	0.8612	0.3647
J	**r**	**b**	100-Tree Random Forest	0.8891	0.3913
K	**b**	**r**	100-Tree Random Forest	0.9939	0.9465
L	**b**	**b**	100-Tree Random Forest	1.0000	1.0000
M	**e**	**r**	100-Tree Random Forest	0.6259	0.0891
N	**e**	**b**	100-Tree Random Forest	0.6627	0.0839
O	**n**	**r**	100-Tree Random Forest	0.8536	0.4248
P	**n**	**b**	100-Tree Random Forest	0.8762	0.4367

Table 3.5: Results of predicting 50% of the data when training on 50%.

Model	**In**	**Out**	**Underlying ML Technique**	**Granular Acc.**	**Absolute Acc.**
A	**r**	**r**	Auto-Tuned Neural Network	0.6175	0.0789
B	**r**	**b**	Auto-Tuned Neural Network	0.6093	0.0833
C	**b**	**r**	Auto-Tuned Neural Network	0.9988	0.9897
D	**b**	**b**	Auto-Tuned Neural Network	0.9027	0.4658
E	**e**	**r**	Auto-Tuned Neural Network	0.6284	0.0918
F	**e**	**b**	Auto-Tuned Neural Network	0.6625	0.0840
G	**n**	**r**	Auto-Tuned Neural Network	0.8361	0.3790
H	**n**	**b**	Auto-Tuned Neural Network	0.8124	0.2359
I	**r**	**r**	100-Tree Random Forest	0.8636	0.3708
J	**r**	**b**	100-Tree Random Forest	0.8908	0.3899
K	**b**	**r**	100-Tree Random Forest	0.9966	0.9695
L	**b**	**b**	100-Tree Random Forest	1.0000	1.0000
M	**e**	**r**	100-Tree Random Forest	0.6318	0.0888
N	**e**	**b**	100-Tree Random Forest	0.6628	0.0842
O	**n**	**r**	100-Tree Random Forest	0.8557	0.4344
P	**n**	**b**	100-Tree Random Forest	0.8763	0.4494

Table 3.6: Results of predicting 30% of the data when training on 70%.

Model	**In**	**Out**	**Underlying ML Technique**	**Granular Acc.**	**Absolute Acc.**
A	**r**	**r**	Auto-Tuned Neural Network	0.6729	0.1181
B	**r**	**b**	Auto-Tuned Neural Network	0.6286	0.0707
C	**b**	**r**	Auto-Tuned Neural Network	0.9994	0.9945
D	**b**	**b**	Auto-Tuned Neural Network	0.9090	0.4846
E	**e**	**r**	Auto-Tuned Neural Network	0.6316	0.0899
F	**e**	**b**	Auto-Tuned Neural Network	0.6617	0.0842
G	**n**	**r**	Auto-Tuned Neural Network	0.8374	0.3745
H	**n**	**b**	Auto-Tuned Neural Network	0.8104	0.2176
I	**r**	**r**	100-Tree Random Forest	0.8644	0.3739
J	**r**	**b**	100-Tree Random Forest	0.8940	0.3934
K	**b**	**r**	100-Tree Random Forest	0.9983	0.9845
L	**b**	**b**	100-Tree Random Forest	1.0000	1.0000
M	**e**	**r**	100-Tree Random Forest	0.6303	0.0867
N	**e**	**b**	100-Tree Random Forest	0.6617	0.0846
O	**n**	**r**	100-Tree Random Forest	0.8555	0.4397
P	**n**	**b**	100-Tree Random Forest	0.8744	0.4563

Table 3.7: Results of predicting 10% of the data when training on 90%.

3.6 Analysis

Across our 80 data points, our first question burns: "*Which model was the most accurate across a wide variety of training-prediction splits?*" To gain a sense of which did well, we first observe figure 3.4.

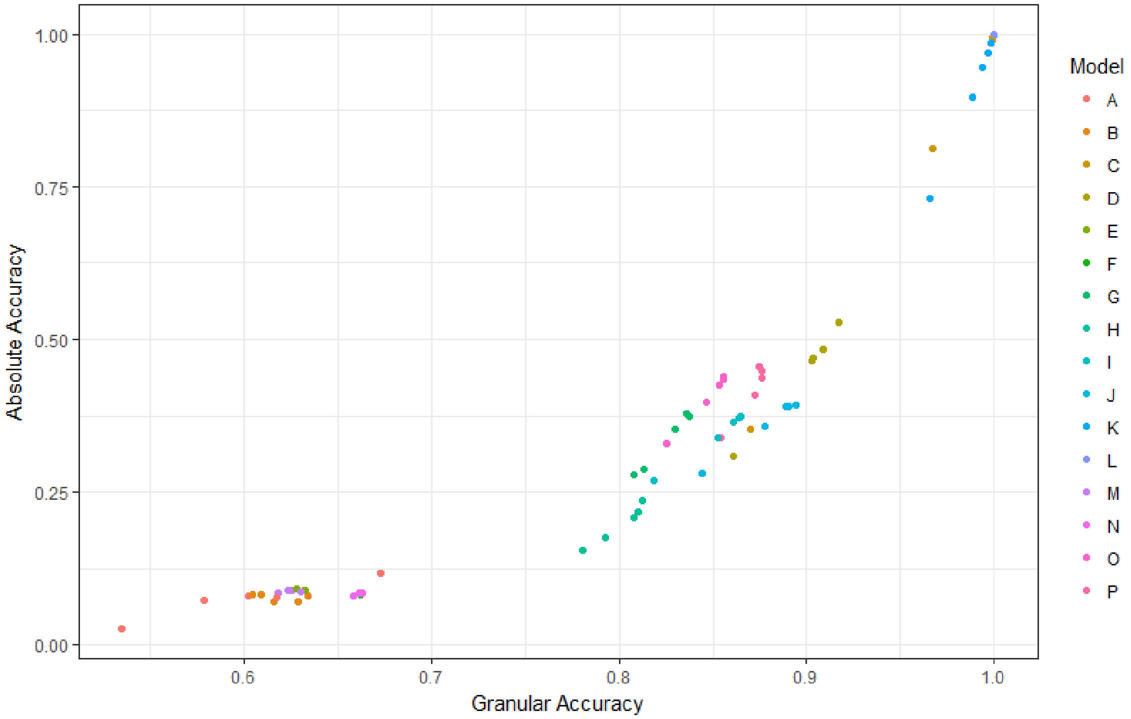


Fig. 3.4: Accuracy of all models across all training-prediction splits.

The top performing models, defined as the models which achieved absolute accuracy greater than 75% or granular accuracy greater than 95% for any training-validation split, were models C, K, and L. Model L in particular did quite well, with an astounding absolute and granular accuracy rate of 0.9997 and 0.9973, respectively, when predicting the remaining 90% of the image data from only 10% of the figure. We also note that all techniques did better than chance in both granular and absolute accuracy.

In figure 3.4 we also observe that there seems to be a tight trend relating the granular and absolute accuracies. Since we are working in the Boolean case, we conjecture that the trend follows a scaled variant of the geometric distribution. In any case, the visualization seems to indicate that an acceptable absolute accuracy of, say, 75%, would be highly correlated with a granular accuracy of 95% or above.

One way to visualize how well these models A through P do across all types of training-prediction ratios is through the box-plots in figure 3.5:

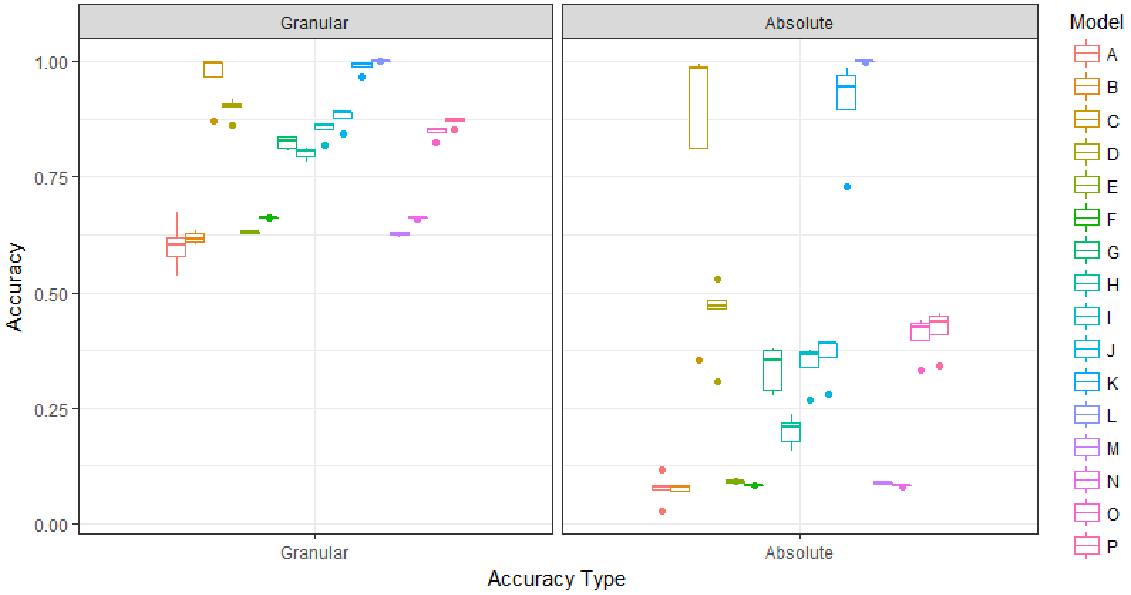


Fig. 3.5: Accuracy across all models.

This visualization is notable in that the thin gray line in each accuracy type separates the neural network based models from the random forest based models. Visually, the spread would seem to indicate that, as a whole, there is not a great deal of difference between the predictive capabilities between the two (albeit with some trends towards forests having greater absolute accuracy predictions).

One commonality between models C, K, and L, however, was the input type: in each case, the binary representation was used. Perhaps this is not surprising, as the binary representation in this case is a particular traversal of each matrix. We can compare other representation types as a reflection of accuracy across all models as noted in the box plots in figure 3.6:

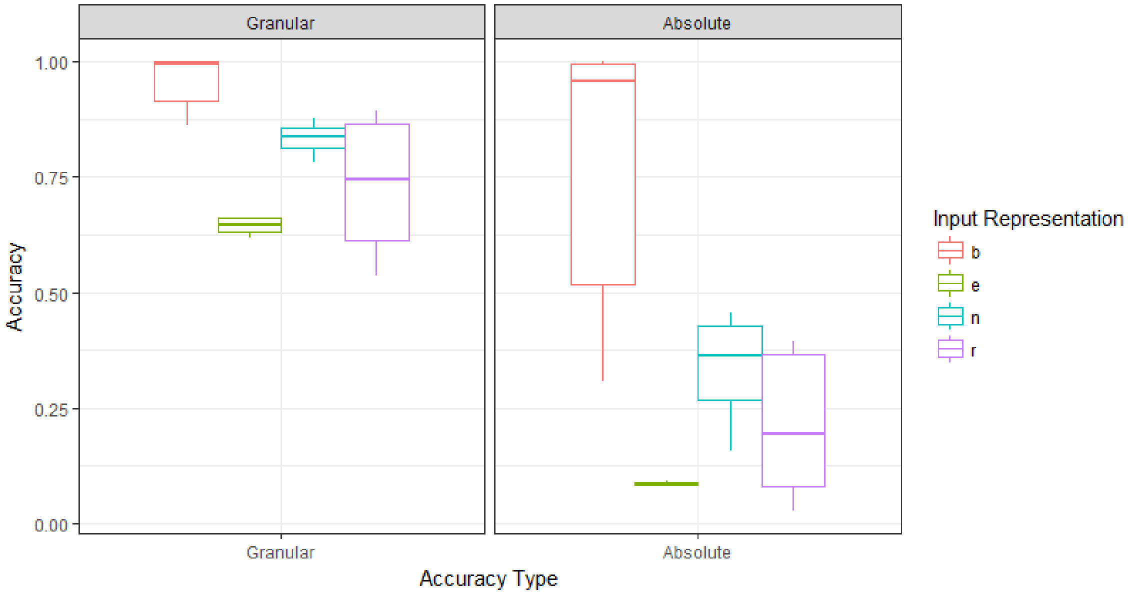


Fig. 3.6: Accuracy across all models with respect to input type.

This box plot provides evidence that random forests in particular can learn how to do Boolean matrix multiplication with very small amounts of binary training data relative to the universe. In the case of the runner up, the naive singular value representation, there is some motivation as to why, in this case, the accuracy could not breach a threshold. There are only so many $n \times n$ Boolean matrices up to permutation, each of which have the same ordered naive singular value representation. While this vector embedding may be useful in the wide variety of applications where linear permutation does not matter, in this case it does.

CHAPTER 4

GRAPH ATTRIBUTE COMPUTATION USING ENTROPY-BASED KERNELS

4.1 Introduction

Graph theory has long been filled with questions which are in the computational class of NP: NP Hard, NP Complete, etc. While many papers in academia have so far pointed to reductions or clever algorithms to answer specific questions relating to the practical computation of these computationally difficult invariants, very few authors have so far used classification techniques – now widely used in statistics and data science – to predict, with quality accuracy, the answers to these questions. Least of all in graph theory.

We hope to rectify at least one corner of this oversight by employing a technique on digraphs which generates a vector of predictors based on the graph in question. This set of predictor vectors can then be used to train a number of common classifiers so that predicting computationally difficult graph invariants then becomes trivial to a high degree of accuracy.

The rest of this chapter will be dedicated to describing the technique we've developed, experiments done to demonstrate the practical validity of the technique on two NP-level problems, and will close with a brief discussion as to future arenas of research in this area.

4.2 Technique

In this section we reiterate a definition for a *graph kernel* described in [29]. We generalize this definition from graphs to real valued matrices. We then use this kernel to create a *kernel matrix*. We apply principal component analysis to this kernel matrix to create a derived set of vectors. These vectors may thus be utilized as predictors for a machine learning apparatus.

4.2.1 Ye et al. Kernel

In 2012, Ye et al., pioneered a technique illustrated in [29]. Their core idea was to create a kernel described as follows:

Let $\mathcal{G} = (V, E)$ be a digraph. For all $v \in V$, let the out-degree of v be denoted as d^{out}, and the in-degree of v as d^{in}. Define $E' = \{(u, v) \mid (u, v) \in E \text{ and } (v, u) \in E\}$. Define the entropy of $\mathcal{G}$, denoted $H(\mathcal{G})$, as

$$H(\mathcal{G}) = \sum_{(u,v)\in E} \frac{d_u^{in}}{d_v^{in} \cdot d_u^{out^2}} + \sum_{(u',v')\in E'} \frac{1}{d_u^{out\prime} \cdot d_v^{out\prime}}$$

Denote the disjoint union graph of $\mathcal{G}_1$, $\mathcal{G}_2$ as $\mathcal{G}_1 \oplus \mathcal{G}_2$. That is, $\mathcal{G}_1 \oplus \mathcal{G}_2 = (V_1 \cup V_2, E_1 \cup E_2)$. From these above definitions, the authors in [29] presented their kernel, $k(\mathcal{G}_1, \mathcal{G}_2)$, as

$$k(\mathcal{G}_1, \mathcal{G}_2) = \exp\left(\frac{H(\mathcal{G}_1) + H(\mathcal{G}_2)}{2} - H(\mathcal{G}_1 \oplus \mathcal{G}_2)\right)$$

We invite the reader to examine [29] directly for the motivation underpinning this definition.

4.2.2 Matrix Generalization of the Ye et al. Kernel

If we consider the adjacency matrix of $\mathcal{G}$, denoted $[\mathcal{G}]$, we can generalize the kernel described in 4.2.1 to two matrices, M_1, M_2, subject to a few constraints. Namely:

1. M_1, M_2 are square (although not necessarily of equal dimension).

2. All entries in both M_1 and M_2 are non-negative.

These constraints are due to the adjacency matrices of two graphs also having these same properties.

Consider the following generalizations of the definitions provided in [29]:

- Let M be a matrix subject to the aforementioned constrains.
- Define $M_1 \oplus M_2$ as the block matrix $\begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix}$. This is analogous to the adjacency matrix of the disjoint union graph of $\mathcal{G}_1$ and $\mathcal{G}_2$. That is, $[\mathcal{G}_1 \oplus \mathcal{G}_2] = \begin{bmatrix} [\mathcal{G}_1] & 0 \\ 0 & [\mathcal{G}_2] \end{bmatrix}$.
- Define $E = \{(i,j) \mid M_{i,j} \neq 0\}$, which generalizes the edge set of a graph.
- Define $E' = \{(i,j) \mid (i,j) \in E \text{ and } (j,i) \in E\}$, keeping the original definition from [29].
- Define $r_M(x) = \sum_{i=0} M_{i,x}$, as the in-degree of a vertex indexed as x in a graph's adjacency matrix. That is, $r_M(x)$ is defined as the sum of the entries of each row of column x.
- Define $c_M(x) = \sum_{j=0} M_{x,j}$, as the out-degree of a vertex indexed as x in a graph's adjacency matrix. That is, $c_M(x)$ is defined as the sum of the entries of each column of row x.
- Define the entropy of M, denoted $H(M)$, as

$$H(M) = \sum_{(i,j) \in E} \frac{r_M(i)}{r_M(j) \cdot c_M(i)^2} + \sum_{(i',j') \in E'} \frac{1}{c_M(i') \cdot c_M(j')}$$

From this relaxed entropy, we can write a kernel for two matrices M_1, M_2 as

$$k(M_1, M_2) = \exp\left(\frac{H(M_1) + H(M_2)}{2} - H(M_1 \oplus M_2)\right)$$

By generalizing this way, we now have the following broader flexibilities:

- We can now use the kernel with other matrices which represent graphs, such as the signless Laplacian matrix.
- Positively weighted digraphs can be considered.
- Other objects represented by square matrices, such as knots or certain algebraic structures, can now utilize the kernel directly without first bijectively relating them to a digraph for classification purposes.

4.2.3 Prediction Using a Variant of Kernel Principal Components Analysis

Consider a set of matrix training data $T_x = \{M_1, M_2, \cdots, M_j\}$, with a corresponding set of classes $T_y = \{C_1, C_2, \cdots, C_j\}$, where C_i is one of a finite number of classes. Further, consider a set of validating matrix data $V_x = \{\mathcal{M}_1, \mathcal{M}_2, \cdots, \mathcal{M}_k\}$, with a corresponding set of classes $V_y = \{\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_k\}$. To predict the classes in V_x, we undergo a few intuitive steps:

1. Create the $j \times j$ kernel matrix from the training matrices:

$$K = \begin{bmatrix} k(M_1, M_1) & k(M_1, M_2) & \cdots & k(M_1, M_j) \\ k(M_2, M_1) & k(M_2, M_2) & \cdots & k(M_2, M_j) \\ \vdots & \vdots & \ddots & \vdots \\ k(M_j, M_1) & k(M_j, M_2) & \cdots & k(M_j, M_j) \end{bmatrix}$$

2. Apply PCA to K, with the resulting set of component vectors denoted as $P = \{\vec{p_1}, \vec{p_2}, \cdots, \vec{p_j}\}$ and ordered by the explained variance (that is, the magnitude of the associated λ_i with each p_i) of each component.

3. Consider the following function $F_P : \mathbb{R}_{\geq 0}^{a \times b} \to \mathbb{R}^j$, $a, b \in \mathbb{N}$.

$$F_P(A) = [\vec{p_1} \cdot \vec{G(A)}, \vec{p_2} \cdot \vec{G(A)}, \cdots, \vec{p_j} \cdot \vec{G(A)}]$$
$$\text{where } G(A) = [k(M_1, A), k(M_2, A), \cdots, k(M_j, A)]$$

4. Define

$$T'_x = \{F(M_1), F(M_2), \cdots, F(M_j)\}$$
$$V'_x = \{F(\mathcal{M}_1), F(\mathcal{M}_2), \cdots, F(\mathcal{M}_k)\}$$

5. Using T'_x, T_y as training data, with V'_x, V_y as validation data, utilize an algorithm of your choice, such as a linear classifier or k-nearest neighbors, to subsequently classify the results. From experiments described in the next section and in [29], the data should already be both quite clustered and linearly separable.

4.3 Experiments

In the following sections we describe experiments undertaken. These experiments validate the efficacy of the technique described in 4.2. We define our data set and then use that data set for predicting the results of two computationally intractable problems.

4.3.1 Data Used

Tournaments, formally defined as orientations of complete graphs, while interesting structures in their own right, have been used to model paired comparisons in domains ranging from animal social hierarchies [30] to voting phenomena in democracies [31].

There are two classical NP level computational problems which are frequently found in conjunction with tournaments. Namely, the *Tournament Isomorphism Class problem*, and the *Minimum Feedback Arc Set Size (FAST) problem.* For both problems, formally defined below, the following data sets were used:

- Adjacency Matrices Representing Labeled Tournaments of Order 4 ($n = 64$)
- Signless Laplacian Matrices Representing Labeled Tournaments of Order 4 ($n = 64$)
- Adjacency Matrices Representing Labeled Tournaments of Order 5 ($n = 1024$)
- Signless Laplacian Matrices Representing Labeled Tournaments of Order 5 ($n = 1024$)

4.3.2 Isomorphism Class on Labeled Tournaments

Formally, an isomorphism of two graphs $\mathcal{G}_1 = (V_1, E_1), \mathcal{G}_2 = (V_1, V_2)$ is a bijection between their vertex sets $f : V_1 \mapsfrom\mapsto V_2$ such that $(v, u) \in E_1 \iff (f(v), f(u)) \in E_2$.

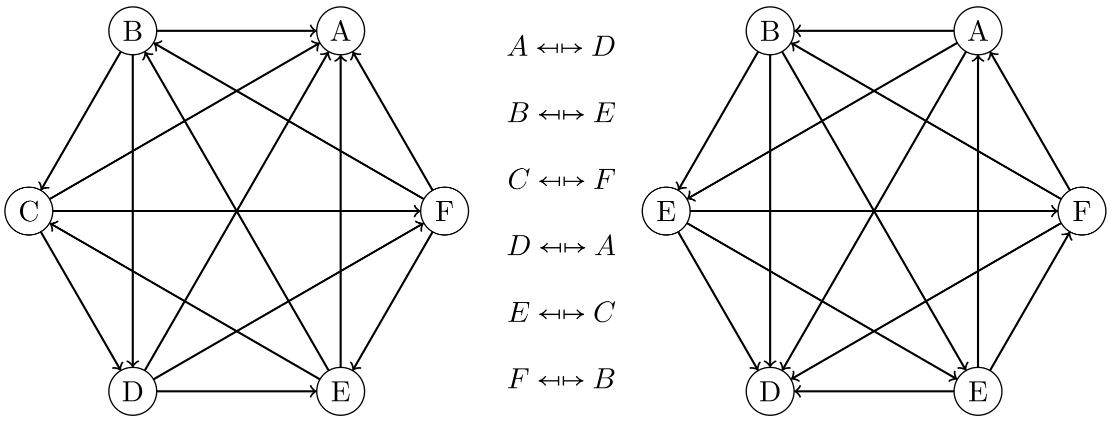


Fig. 4.1: Tournaments in the same isomorphism class.

A set of graphs which are isomorphic to each other are said to be of the same isomorphism class. Determining whether two graphs belong to the same isomorphism class is, in general, of computational complexity $2^{O(\log(n)^c)}$ for $c > 0$ [32].

On labeled tournaments of orders 4 and 5, there are, respectively, 4 and 12 isomorphism classes. Given that each class is ascribed an artificial label (e.g., one class is labeled "A", the next "B", etc.), train a classifier based on some data for which the isomorphism class is known, and predict the class for a new observation for which the isomorphism class is unknown.

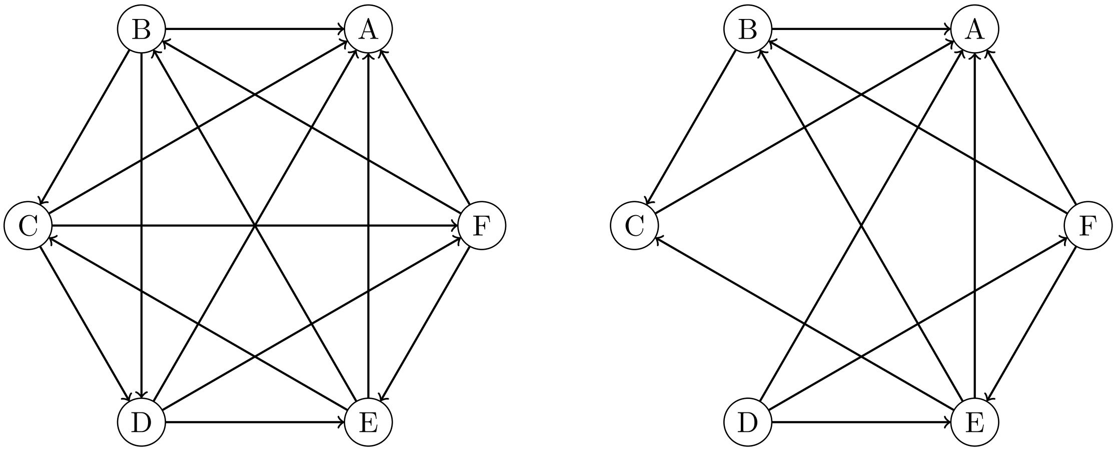
Beforehand, the isomorphism class was computed for each tournament in our training set by brute force. The training instances were then randomized and results validated by 10-fold validation, as noted in section 4.5.

4.3.3 Minimum Feedback Arc Set Size on Labeled Tournaments

A feedback arc set is a set of edges on a digraph, which if removed from the parent graph, would result in the digraph becoming acyclic.

It is interesting to determine the minimum size of the arc set needed to make a particular digraph acyclic, and the problem is in fact NP-Complete, as shown by Karp [2]. When the graphs are tournaments, the problem is known as the *minimum Feedback Arc Set on Tournaments*, or FAST.

While others, notably Bessy et al., have focused on deriving polynomial time kernel algorithms specific to the FAST problem [33], there is little to no literature describing how the generalized kernel methods, such as those outlined in this paper, preform. Indeed, Bessy et al. specifically mention this as an area of future work in their own article.



Some cycles in the above tournament: BCF, BDE, BDF, CDE, CFE, CDFE, BCDF, BCDFE.

Derived acyclic digraph. The three edges BD, CD, CF have been removed. Minimal arc set size: 3.

Fig. 4.2: A visualization of FAST

Beforehand, the minimum feedback arc set size was computed for each tournament in our training set by brute force. Since the minimum feedback arc set size is an integer in the range of, at most, $[0, |E|]$, we can treat the problem as classification with up to $|E|$ classes.

The training instances were then randomized and results validated by 10-fold validation, as noted in 4.5.

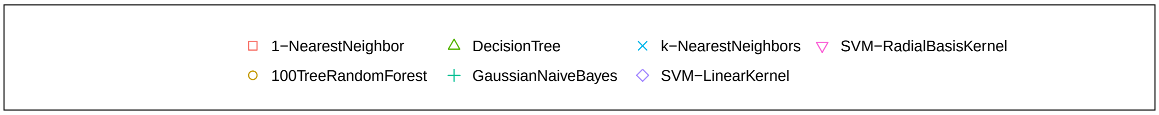
4.3.4 Final Classifiers

The reader will recall that an additional classifier is needed to actually classify the data once the steps outlined in section (2) are preformed. The classifiers selected for our comparison are:

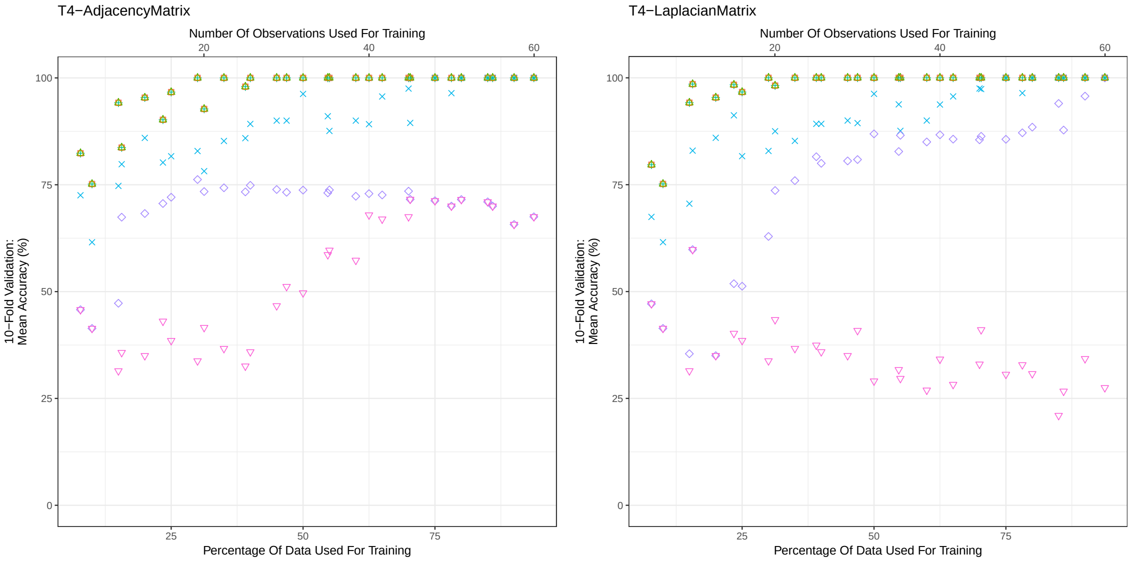
- k-nearest neighbors, where $k = 1$.
- k-nearest neighbors, where $k = \lfloor\sqrt{\text{Number of Training Observations}}\rfloor$.
- A single decision tree, using Gini impurity.
- A random forest with 100 trees, each using Gini impurity.
- A support vector machine, with a linear kernel used.
- A support vector machine, with a radial basis kernel used.
- A Gaussian naive Bayes classifier.

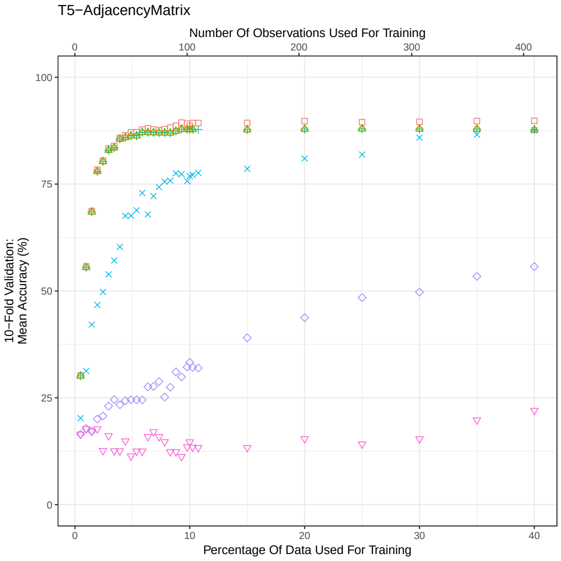
All classifiers were implemented using the Python `Scikit-Learn` library [26]. An arbitrary seed was set during 10-fold validation to ensure that, while the selected training and testing samples remained random, each classifier would evaluate the same dataset.

4.4 Results

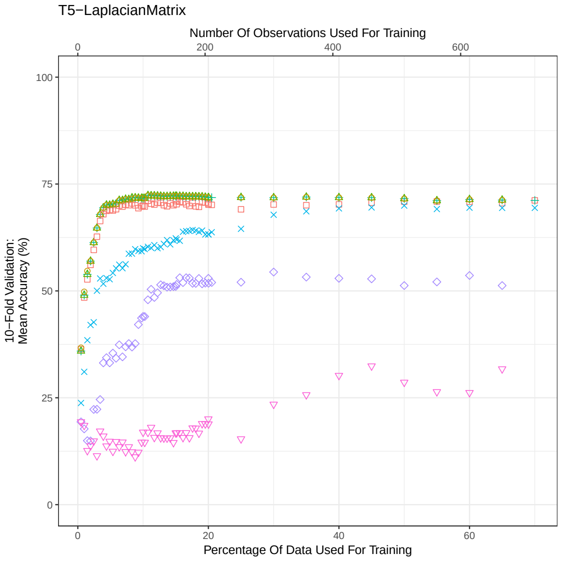


4.4.1 Isomorphism Class on Labeled Tournaments



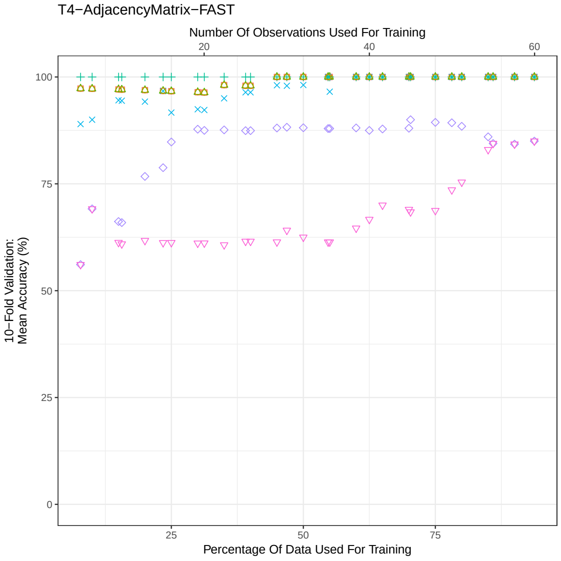


T5−AdjacencyMatrix

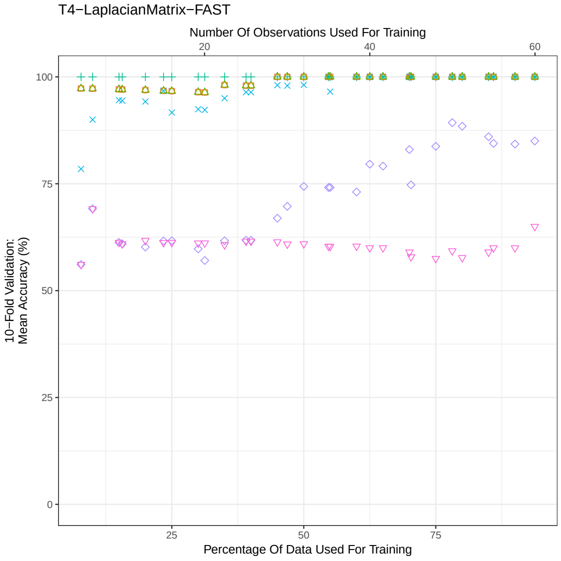


T5−LaplacianMatrix

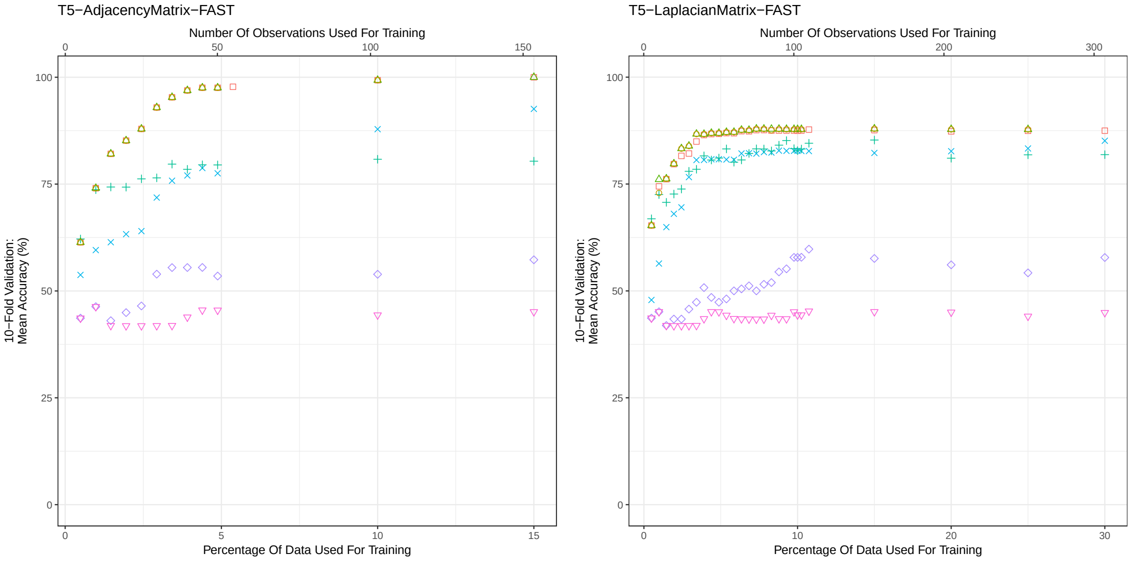
4.4.2 Minimum Feedback Arc Set Size on Labeled Tournaments



T4−AdjacencyMatrix−FAST



T4−LaplacianMatrix−FAST



4.5 Analysis

It is remarkable to note how quickly the highest accuracy is obtained relative to the amount of training data provided. Among the two problems and eight data sets evaluated, the plateau of accuracy for the most accurate classifier tended to be at no more than 30% of the data in the universe.

What was even more intriguing was the lack of linear separability, as inferred by the usually poor performance of the two support vector machines.

In any case, the consistently high accuracy of the 1-nearest neighbor, singular decision tree, and 100 tree forest classifiers perhaps warrants further investigation.

While the Ye-Wilson-Hancock kernel based on Shannon-Jensen entropy was generalized for generic graphs, it may be that it is not as useful for certain subtypes of graphs such as the tournaments. An area of further work, therefore, is in analyzing other entropy based kernels across other graph families.

Another future area of investigation include graphs of larger order. In practice, the problems of this magnitude have already been solved, utilizing algorithms such as the `nauty` program [34]. The rigour of ramping up to data of higher dimension needs to be considered.

CHAPTER 5

ENTROPY BASED VARIABLE SELECTION

In section 2.5 we discussed ways to vectorize matrices as preparation for input to a supervised machine learning apparatus. In section 2.7 we discussed the importance of dimensionality reduction in combinatorial contexts. This gives rise to the question "*What elements of a vector (especially a vector created from methods described in section 2.5) can be safely removed from consideration in machine learning prediction, regardless of the associated response? Can we minimize the number of elements in our eventual machine learning input vector without using potentially unstable and computationally expensive dimensionality reduction techniques described in section 2.7?*"

We see in Chapter 4 the utility in using the broadly constructed idea of entropy (that is, the amount of (dis)order in a system) to generate predictors for graphs. We suspect that some definition of entropy may be applied to generalized unsupervised feature selection. Indeed, upon inspection of the literature we found that unsupervised feature selection algorithms have been developed before, with the techniques described in [35] being one of the first papers to utilize entropy in any sort of way.

Existing methods such as those described in [35], however, utilize a pairwise definition of entropy rooted in linear correlation coefficents, or via nearest neighbor entropy as in [36]. Here we propose two new methods for feature selection, *Componentwise Observation for Variable Selection with Entropy* (COVSE) and *Vectorwise Observation for Variable Selection with Entropy* (VOVSE). Both methods utilize Shannon entropy at their crux, and both consider an entire probability distribution via kernel density estimation. We then show the efficacy of COVSE by using the MNIST dataset as a case study.

5.1 Componentwise Observation for Variable Selection with Entropy (COVSE)

Suppose we have a set of vectors $\vec{v_1} = \begin{bmatrix} a_1 \\ b_1 \\ \vdots \\ n_1 \end{bmatrix}, \vec{v_2} = \begin{bmatrix} a_2 \\ b_2 \\ \vdots \\ n_2 \end{bmatrix}, \cdots, \vec{v_k} = \begin{bmatrix} a_k \\ b_k \\ \vdots \\ n_k \end{bmatrix}$ where $\vec{v_i} \in \mathbb{R}^n$, each vector $\vec{v_i}$ is chosen i.i.d. from our universe of interest, and k is a sufficiently large sample size.

Our initial motivation is to find the entropy of each component independently. Recall the following definition for the Shannon entropy of X, denoted $H(X)$, where X is a random variable:

$$H(X) = -\sum_{x \in X} \Pr(X = x) \log_2(\Pr(X = x)) \qquad [37]$$

Let $\mathcal{A} = \{a_1, a_2, \cdots, a_k\}$, $\mathcal{B} = \{b_1, b_2, \cdots, b_k\}$, $\cdots$, $\mathcal{N} = \{n_1, n_2, \cdots, n_k\}$. We could construct the entropy for each $H(\mathcal{A})$, $H(\mathcal{B}), \cdots, H(\mathcal{N})$ if only we had probability distributions for each of $\mathcal{A}$, $\mathcal{B}$, and so on.

5.1.1 Kernel Density Estimation

Kernel density estimation (KDE), as developed in [38, 39], is a tool for providing such a distribution. We define a kernel density estimator, $f_b(x) : X \subset \mathbb{R} \to [0, 1] \subset \mathbb{R}$, such that $\int_X f_b(x)\, dx = 1$, as follows:

$$f_b(x) = \frac{1}{|X|} \sum_{x' \in X} K_b(x - x')$$

where

- $b \in \mathbb{R}_{>0}$ is a smoothing parameter called *bandwidth*. While we strive to find this parameter by cross validation, we nonetheless utilize rules of thumb as a computational expedient. One such rule is given by Silverman in [40]. We prefer the rule given by Scott [41] due to its optimality in the Gaussian case.

- $K_b(x) : \mathbb{R} \to \mathbb{R}$ is a non-negative function called the *scaled kernel* in this context. We choose the scaled Gaussian normal kernel, defined as:

  $$K_b(x) = \frac{\phi\left(\frac{x}{b}\right)}{b}$$

  Where $\phi(x)$ is the standard normal density function:

  $$\phi(x) = \frac{1}{\sqrt{2\pi}} \cdot \exp\left(-\frac{x^2}{2}\right)$$

Given that we set up KDEs for each of $\mathcal{A}$, $\cdots$, $\mathcal{N}$ we can now determine the entropy, $H(X)$, for each vector component.

Recalling that entropy is a metric of the information contained within our set of random variables actualized from X [37], we can determine a threshold t of our choosing such that all $z_i \in \mathcal{Z} \in \{\mathcal{A}, \cdots, \mathcal{N}\}$ are discarded as elements in our vectors when $H(\mathcal{Z}) < t$. One way to select such a t is to plot all $H(\mathcal{Z})$s monotonically and attempt to identify an elbow.

Another way to determine which set of elements to deselect is by taking a *meta entropy* and utilizing it as a response variable to some sort of linear model. That is, let $\chi = \{H(\mathcal{A}), H(\mathcal{B}), \cdots, H(\mathcal{N})\}$. Using a KDE as described in 5.1.1, we can determine $H(\chi)$, which we call meta entropy. We can then utilize tools of explained variance with respect to $H(\chi)$ to similarly order and threshold the given $H(\mathcal{Z})$s.

This idea of meta entropy, however, is hinting at a larger idea of considering all of the elements in juxtaposition with each other. We survey this idea more directly than merely aggregating componentwise entropy values in section 5.3.

5.2 COVSE Experiments on the MNIST Dataset

MNIST is a classic dataset used for image recognition. It consists of a set of handwritten digits, each labeled 0 through 9. Traditionally, 60000 of these images are used for training a machine learning apparatus, and 10000 are used for testing [42]. Each pixel of the 28×28

sized image serves as a feature for a machine learning apparatus to predict on. Several examples of these images are in figure 5.1.[1]

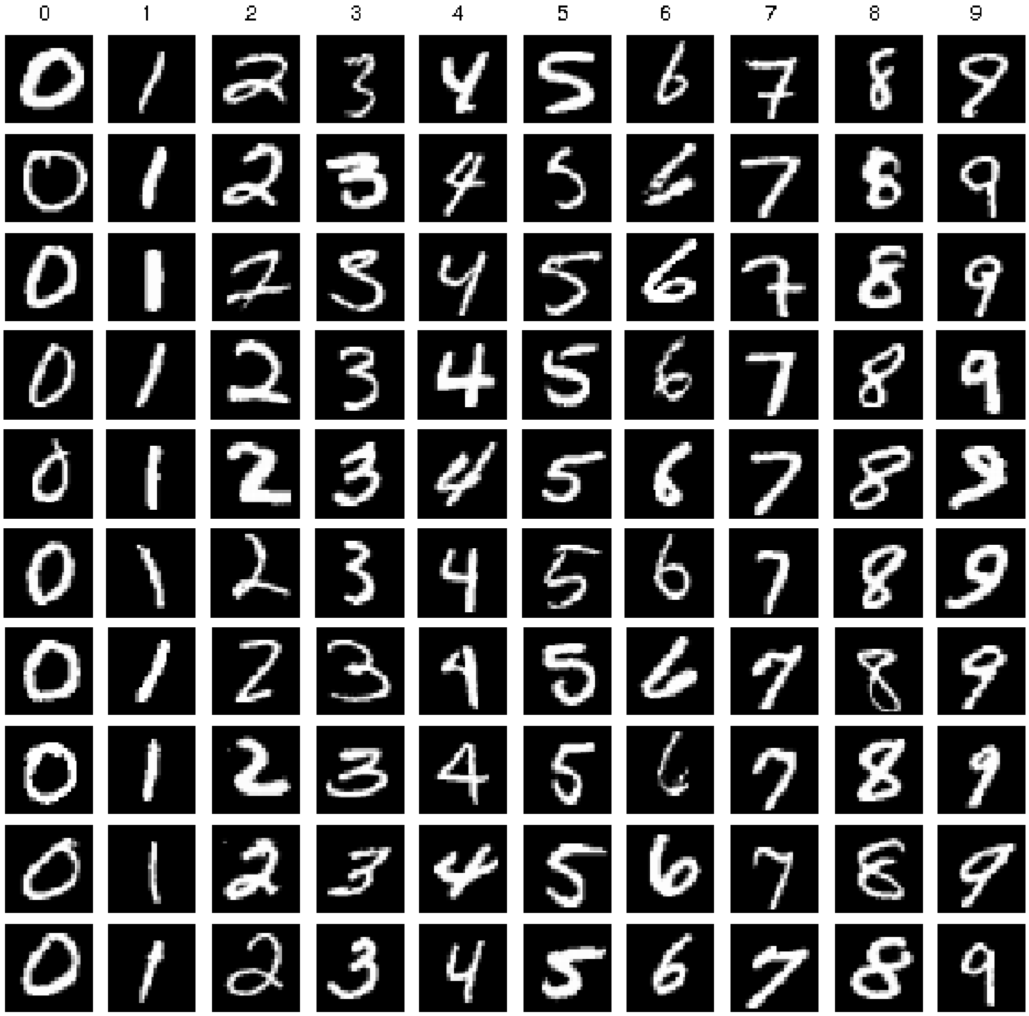


Fig. 5.1: Examples of Images and Labels in the MNIST Dataset

[1] This image of MNIST examples is sourced from [43].

In the experiment below we will do the following:

1. Determine the entropy of each pixel based on the method described in 5.1. We will do this for several different sizes of training data.

2. Based on the entropies calculated in step 1, remove pixels from consideration in our training data. We then train a random forest to predict an image's class based on only the features selected. Accuracy of the training set is then reported.

5.2.1 COVSE Predictor Selection Technique

In 5.1 we assumed the following before describing COVSE:

Suppose we have a set of vectors $\vec{v_1} = \begin{bmatrix} a_1 \\ b_1 \\ \vdots \\ n_1 \end{bmatrix}, \vec{v_2} = \begin{bmatrix} a_2 \\ b_2 \\ \vdots \\ n_2 \end{bmatrix}, \cdots, \vec{v_k} = \begin{bmatrix} a_k \\ b_k \\ \vdots \\ n_k \end{bmatrix}$ where $\vec{v_i} \in \mathbb{R}^n$, each vector $\vec{v_i}$ is chosen i.i.d. from our universe of interest, and k is a sufficiently large sample size.

This begs the question: "*How large is a sufficiently large k?*" COVSE's efficacy in determining how many n predictors to keep (and of those n, which ones) depends on obtaining a true representation of the entropy of each predictor in our dataset. Ideally, then, k would be equal to the number of training instances to gain the truest representation of the entropy of the underlying data.

Nevertheless, calculating entropy is expensive, and adding more data could quickly result in diminishing returns. Hence, given that we must include at least two samples to run COVSE, and number of training samples is 60000, we took the following values for potential $k \in D$; $K = \{2, 4, 12, 40, 133, 452, 1533, 5205, 17617\}$. These K values were chosen because they are the first 9 values evenly spaced on the log scale $[0, 60000]$[2]

[2] Refer to the Python code `numpy.logspace(0,math.log10(60000), 10, dtype="int64") + 1`

For each k in $K = \{2, 4, 12, 40, 133, 452, 1533, 5205, 17617\}$, we selected without replacement k MNIST images from our training set. We then determined the entropy for each predictor in the entire training set by applying COVSE to those k images.

5.2.2 COVSE Results

Below are the results for each k in $K = \{2, 4, 12, 40, 133, 452, 1533, 5205, 17617\}$. For each k we plot the entropy of each pixel as a colorized image using the `viridis` color pallet in the Python library `matplotlib` [44]. Additionally, we also determine the number of pixels (that is, predictor entries) without entropy (i.e., entropy is 0), as well as reporting that number as a percentage of all $28 \cdot 28$ pixels. With that information, we suggest a threshold t of entropy for which predictors with entropy $< t$ should be discarded based on the calculated elbow of the curve.

The elbow was calculated by the point which had the maximized orthogonal distance between the line given by the first and last points in a set of ordered numbers. As a concrete example, consider the following ordered set: $\{0.9, 1.1, 1.1, 1.9, 2.5, 2.8, 4.9, 8.5\}$.

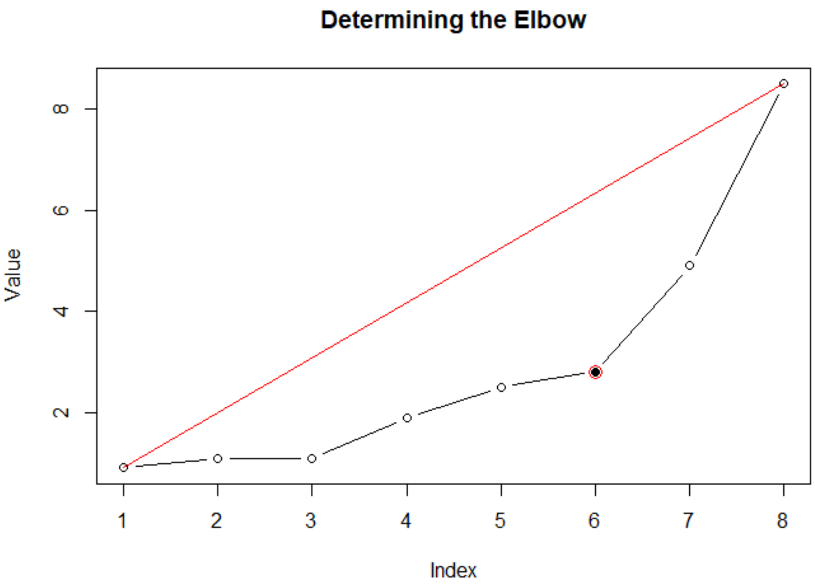
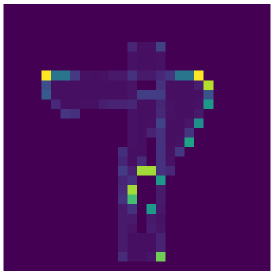
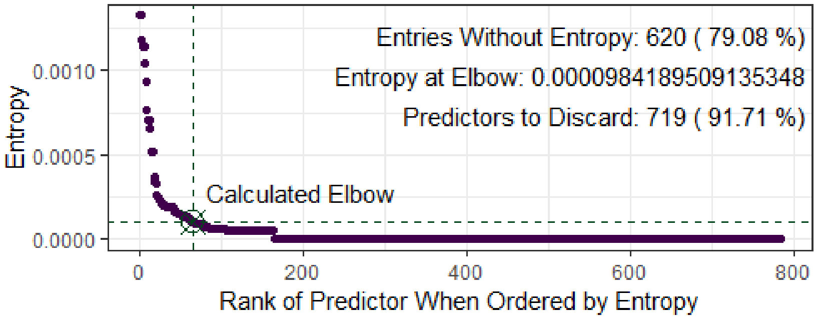


Fig. 5.2: Finding The Elbow of an Ordered Set of Data

In this example, the value 2.8 is the elbow due to it having the maximal orthogonal distance from the red line. We note that the algorithm is formally defined in the `findElbow` method of [45]. While other techniques to find the elbow are based on maximizing the absolute value of a curve's second derivative [46], we utilize this method precisely because it doesn't depend on the derivative. Our ordered data may not have a sufficiently robust piecewise continuity to take an accurate numerical approximation of the second-order derivative.

Using the entropy at the elbow as a threshold, we then recommend that we discard all predictors with COVSE calculated entropy less than that of the elbow from subsequent consideration. We report the entropy of the elbow, and the resulting number (and percentage of the initial $28 \cdot 28$ entries) of predictors to discard.

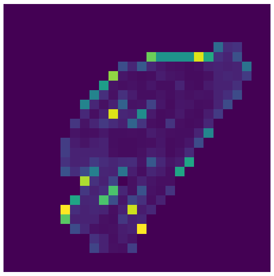
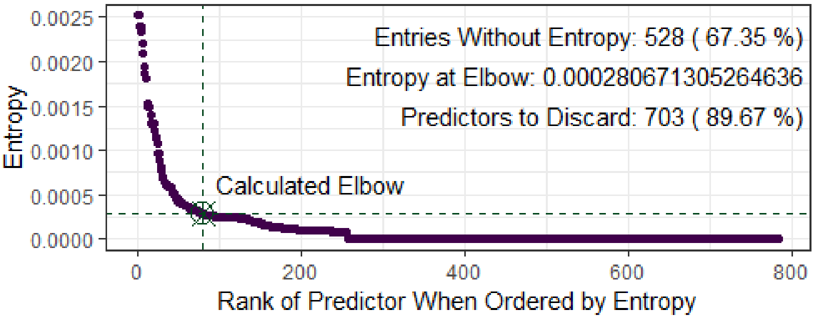
5.2.2.1 COVSE Calculated With 2 Randomly Selected MNIST Training Images

Entropy of Pixels	**Summary Results**
	

In this experiment, only two images from our MNIST training data were selected to determine the entropy for all pixels in the set. As one might suspect from the image of the pixel entropies plotted above, the labels corresponding to the randomly selected samples were indeed "1" and "7". One notes that entropy is highest around the apparent edge of the strokes.

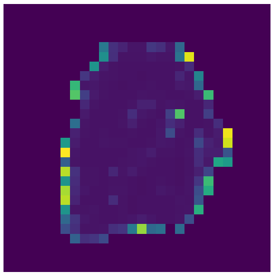
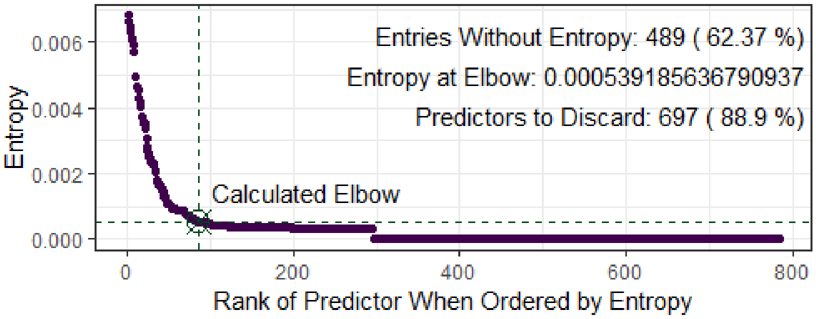
We also note that the sum of entropy across all entries is primarily concentrated in only $(28 \cdot 28) - 719 = 65$ pixels. If the reduction of predictors by $\geq 90\%$ continues for COVSE when its trained on more samples, then this technique has promise for a general purpose feature reduction algorithm.

5.2.2.2 COVSE Calculated With 4 Randomly Selected MNIST Training Images

Entropy of Pixels	Summary Results
	

As we increase the number of samples used for training, the entropy of each pixel seems to paint a more blurred picture. The sample of images used for training COVSE had labels of $\{4, 7, 7, 8\}$. The overall shape of the curve of entropies, however, seems to be very analogous to the $k = 2$ case. Indeed, the number of predictors to keep is $(28 \cdot 28) - 703 = 81$, implying we discarding close to 90% of predictors; this is similar to the $k = 2$ case as well. Based on the visualized entropy of the pixels, however, it appears that the predictors to keep would differ somewhat from the $k = 2$ case.

5.2.2.3 COVSE Calculated With 12 Randomly Selected MNIST Training Images

Entropy of Pixels	Summary Results
	

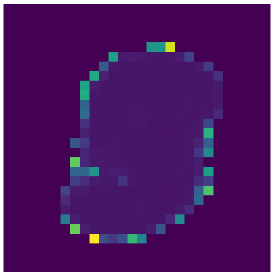
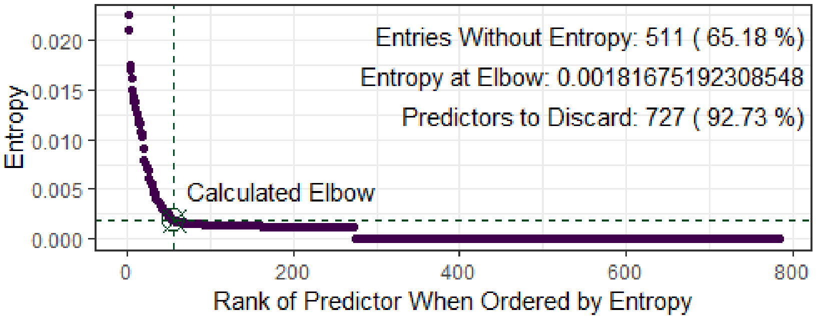
Class	Count	Frequencies
0	1	0.08333333333333333
1	1	0.08333333333333333
2	2	0.16666666666666666
3	0	0.0
4	1	0.08333333333333333
5	0	0.0
6	4	0.3333333333333333
7	1	0.08333333333333333
8	0	0.0
9	2	0.16666666666666666

Table 5.1: Label Distribution of the k Randomly Selected Training Samples

Here we see a bit more diversity in the samples provided. What is notable is how the entropies seem to be maximized when the edge strokes are on the extremes. As in the $k = 2$ and $k = 4$ case, we continue to find that the number of predictors to discard remains at $\approx 90\%$.

Let's observe what happens for the next several values of k.

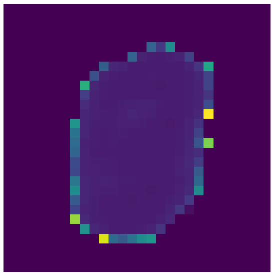
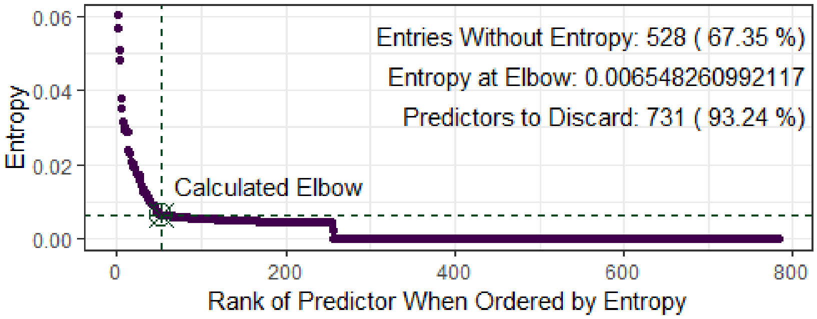
5.2.2.4 COVSE Calculated With 40 Randomly Selected MNIST Training Images

Entropy of Pixels	Summary Results
	

Class	Count	Frequencies
0	6	0.15
1	2	0.05
2	5	0.125
3	3	0.075
4	2	0.05
5	7	0.175
6	3	0.075
7	8	0.2
8	2	0.05
9	2	0.05

Table 5.2: Label Distribution of the k Randomly Selected Training Samples

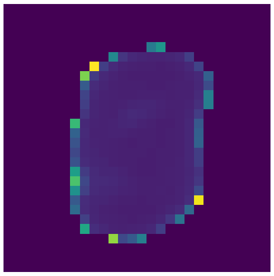
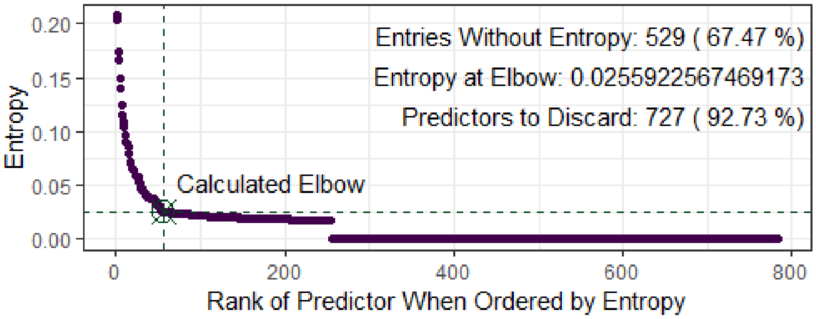
5.2.2.5 COVSE Calculated With 133 Randomly Selected MNIST Training Images

Entropy of Pixels	Summary Results
	

Class	Count	Frequencies
0	15	0.112781954887218 04
1	17	0.12781954887218044
2	8	0.06015037593984962
3	13	0.09774436090225563
4	10	0.07518796992481203
5	8	0.06015037593984962
6	16	0.12030075187969924
7	17	0.12781954887218044
8	11	0.08270676691729323
9	18	0.13533834586466165

Table 5.3: Label Distribution of the k Randomly Selected Training Samples

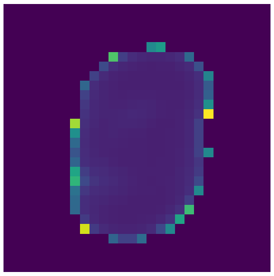
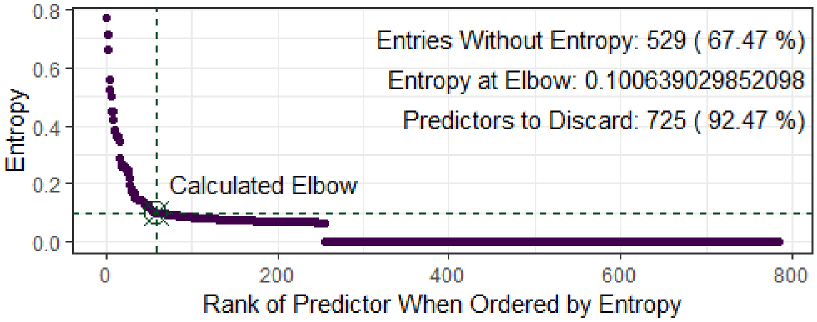
5.2.2.6 COVSE Calculated With 452 Randomly Selected MNIST Training Images

Entropy of Pixels	Summary Results
	

Class	Count	Frequencies
0	36	0.0796460176991150 4
1	57	0.126106194690265 5
2	47	0.103982300884955 75
3	46	0.101769911504424 78
4	57	0.1261061946902655
5	48	0.10619469026548672
6	48	0.10619469026548672
7	39	0.086283185840707 96
8	31	0.06858407079646017
9	43	0.09513274336283185

Table 5.4: Label Distribution of the k Randomly Selected Training Samples

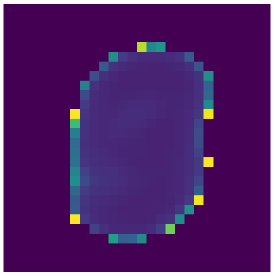
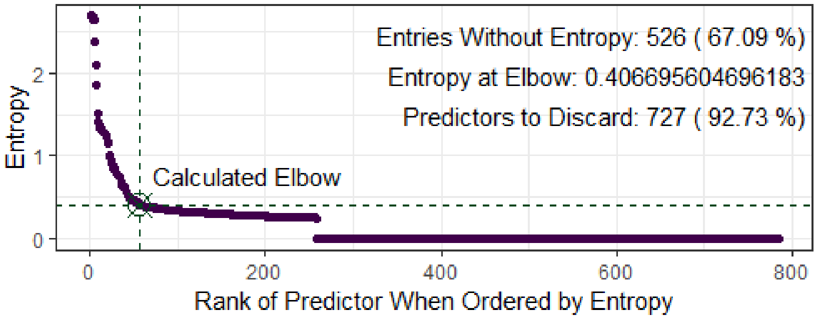
5.2.2.7 COVSE Calculated With 1533 Randomly Selected MNIST Training Images

Entropy of Pixels	Summary Results
	

Class	Count	Frequencies
0	145	0.09458577951728636
1	182	0.1187214611872146
2	147	0.0958904109589041
3	176	0.11480756686236138
4	168	0.1095890410958904
5	119	0.0776255707762557
6	163	0.10632746249184605
7	148	0.09654272667971298
8	131	0.08545335942596216
9	154	0.1004566210045662

Table 5.5: Label Distribution of the k Randomly Selected Training Samples

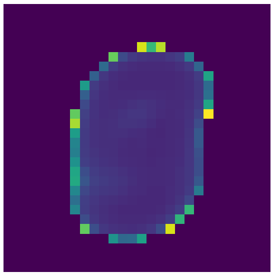
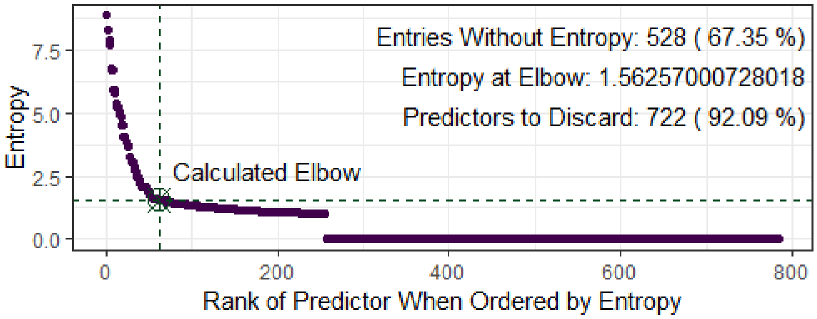
5.2.2.8 COVSE Calculated With 5205 Randomly Selected MNIST Training Images

Entropy of Pixels	Summary Results
	

Class	Count	Frequencies
0	537	0.103170028818444381
1	595	0.114313160422670 5
2	519	0.0997118155619 5965
3	525	0.100864553314121 04
4	495	0.0951008645533 1412
5	483	0.0927953890489 9136
6	479	0.0920268972142171
7	577	0.1108549471661 8637
8	483	0.0927953890489 9136
9	512	0.0983669548511 0471

Table 5.6: Label Distribution of the k Randomly Selected Training Samples

5.2.2.9 COVSE Calculated With 17671 Randomly Selected MNIST Training Images

Entropy of Pixels	Summary Results
	

Class	Count	Frequencies
0	1711	0.09712209797354827
1	2006	0.11386728727933246
2	1684	0.09558948742691718
3	1768	0.10035760912754725
4	1723	0.09780325821649544
5	1605	0.09110518249418176
6	1764	0.10013055571323154
7	1788	0.10149287619912585
8	1735	0.09848441845944259
9	1833	0.10404722711017766

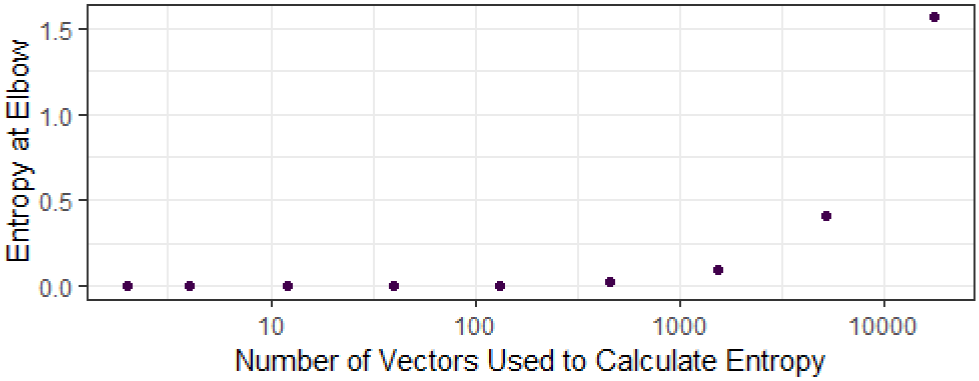
Table 5.7: Label Distribution of the k Randomly Selected Training Samples

5.2.3 Meta COVSE Results

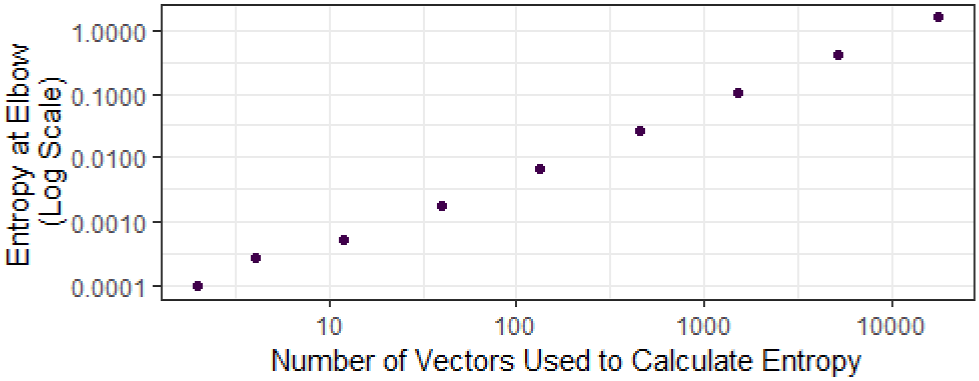
In 5.2.2 we examined the results for several values of k, where k is the number of samples (or, more correctly, the number of vectors which have entries composed of a sample's predictors) to train COVSE. The overarching motivation for doing so was to gain intuition as to the smallest value of k needed to sufficiently capture the entropy of the entire dataset. In this section we compare the metrics of the selected ks side by side to help us glean further understanding.

5.2.3.1 The Entropy at the Selected Elbow

We hypothesized that the entropy of the elbow would be a reasonable threshold. If the entropy at the threshold stabilizes, then the elbow of the elbows might be a good cutoff for k. We observe the following plots to determine if that hypothesis had merit.

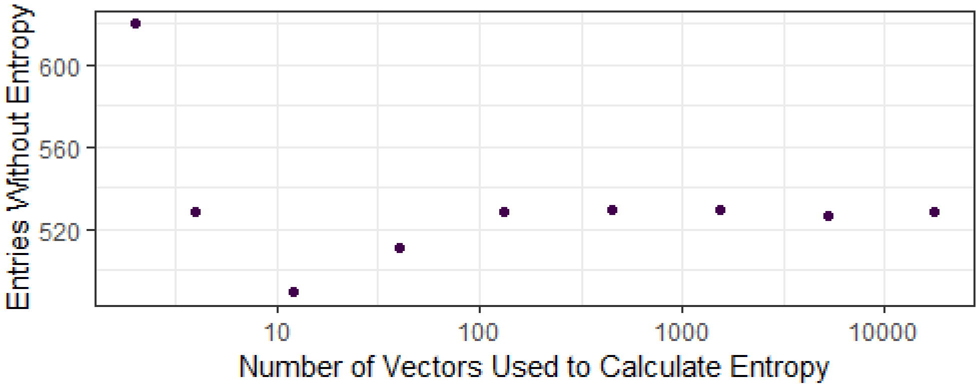


Far from stabilizing to a singular value, we see that the entropy at the elbow consistently increased as k increased. Contrast the above plot when the same entropy data is shown on a log scale.



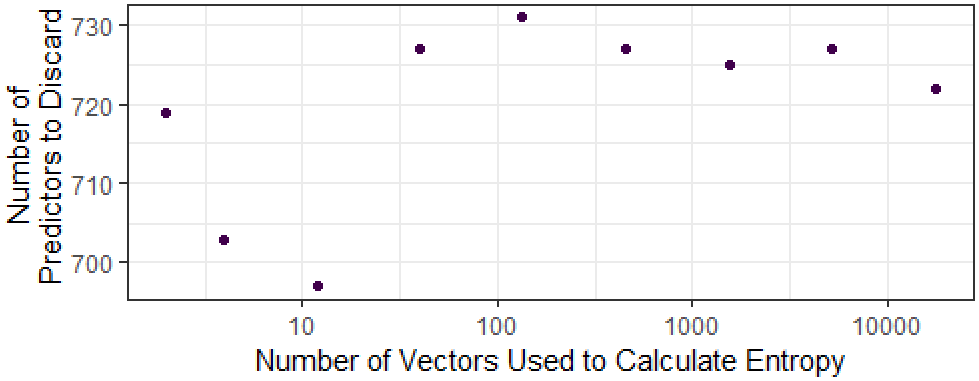
The plot indicates that, instead of converging, entropy increased linearly as k increased. Retrospectively, it is not a surprising observation that the disorder of a system increases as more elements are added. Nonetheless, we still haven't answered the question as to what, if any, k is small enough to sufficiently train COVSE.

5.2.3.2 Number of Entries Without Entropy



Despite the linearity of the elbow entropy noted in section 5.2.3.1, we find that there does appear to be convergence in the number of predictor entries (i.e., pixels) without entropy around $k \approx 100$, and certainly by $k = 452$.

5.2.3.3 Number of Predictors to Discard



We also see the number of predictors to discard converge (or at least become more tightly bounded) once k exceeds 100. Certainly when $k = 452$.

With this in mind, recall that the sample size s for a infinite population is given by

$$s = \frac{z^2 \cdot (p) \cdot (1 - p)}{c^2} \qquad [47]$$

where z is the Z-score of the standard normal distribution, p is probability of the choice, and c is the confidence interval. Noting that s is maximized when $p = 0.5$, we can say that with a confidence interval of 0.05 and a confidence level of 0.95 ($z = 1.96$), then

$$s = \frac{1.96^2 \cdot (0.5) \cdot (1 - 0.5)}{(0.05)^2}$$

$$s = 385.$$

As $k = 452$ is greater than $s = 385$, and noting that two indicative attributes in sections 5.2.3.2 and 5.2.3.3 seemed to converge for $k > 452$, there seems to be an indication that the smallest sufficient k is that given by the sample size s for some chosen c and z.

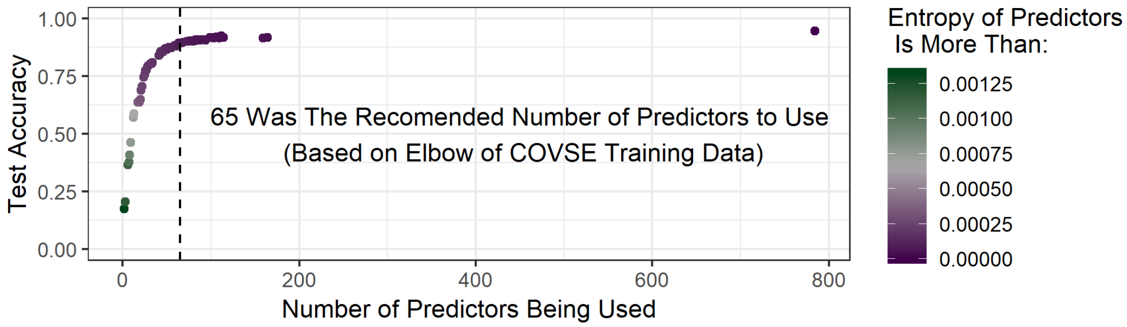
5.2.4 Comparative Accuracy via Random Forest

In section 5.2.2, we claimed that we should ignore the ≈ 725 predictors with the lowest entropy. We put that claim to the test here. For each estimation ($k = 2$, 4, 12, and so forth), we considered all distinct entropies returned by COVSE. Using each one of those distinct entropies as a threshold for predictors to include for training and testing, we utilized a 10 tree random forest to classify the image. Each forest was trained on the full 60000 image training dataset, and evaluated on the 10000 image test dataset.

We utilized small random forests due to their turnkey nature (as opposed to a neural network with many hyperparameters to choose) and speed in training. The particular software package we used is `Scikit-Learn`'s `RandomForestClassifier` [26].

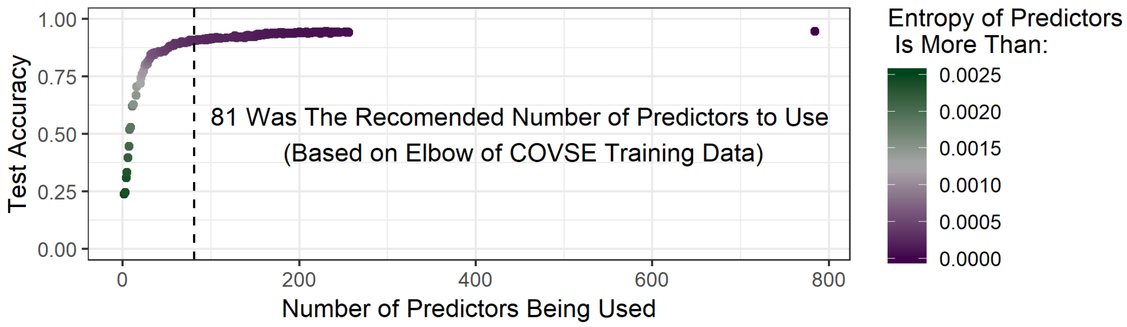
We give the predicted accuracy for each entropy threshold in the results below. The vertical dashed line indicates the final accuracy when the recommended number of predictors was evaluated. As a reminder from section 5.2.2, that recommendation came from an unsupervised selection based on the elbow of the COVSE entropies calculated across each predictor.

5.2.4.1 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.1 (i.e., COVSE trained on 2 images)



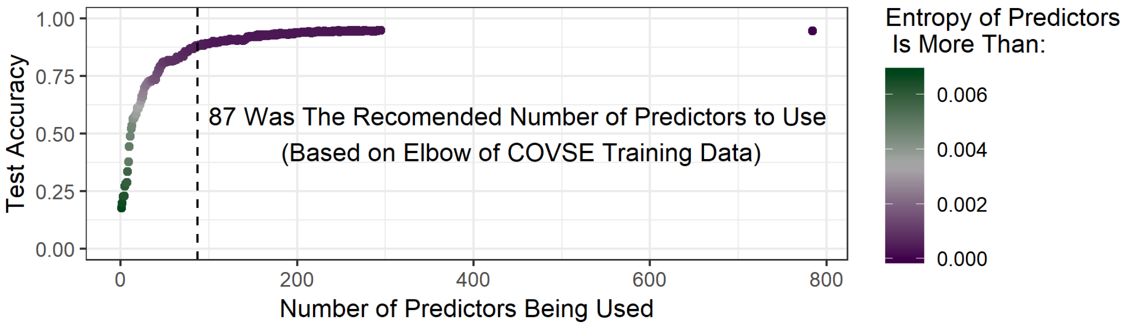
We note the large gap between *Number of Predictors Being Used* ≈ 175 and *Number of Predictors Being Used* $= 28 \cdot 28$ is due to many predictors having no entropy at all. We also see here that, even with entropy calculated from just two training images, how well a random forest classifier performs when trained only on the selected predictors. For example, note that the classifier achieves more than 90% accuracy when utilizing only 65 pixels for prediction.

5.2.4.2 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.2 (i.e., COVSE trained on 4 images)

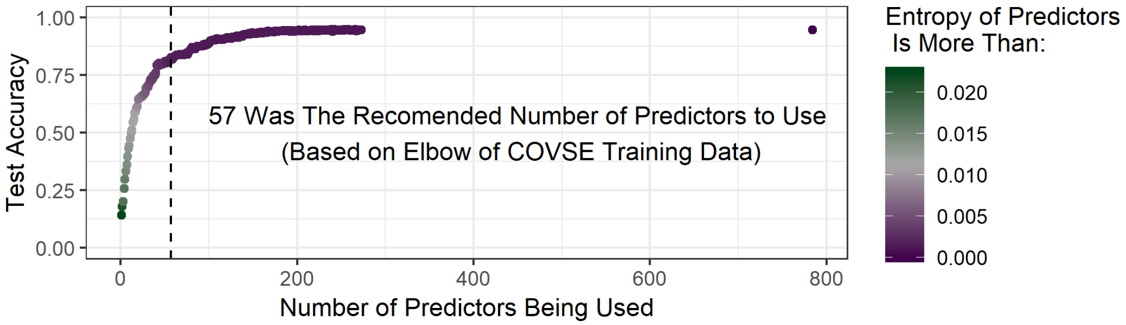


We see a similarly shaped curve when four images were used for entropy calculation, although it resulted in fewer predictors having no entropy, as noted in section 5.2.3.2. Subsequent results will follow this trend of a Γ shaped curve.

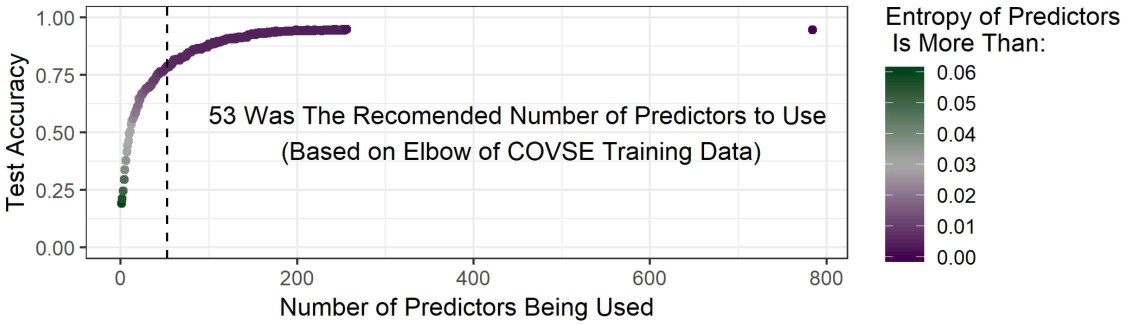
5.2.4.3 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.3 (i.e., COVSE trained on 12 images)



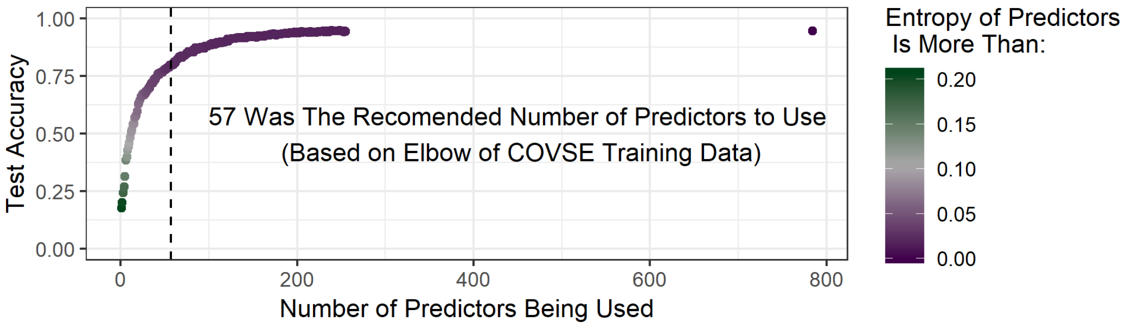
5.2.4.4 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.4 (i.e., COVSE trained on 40 images)



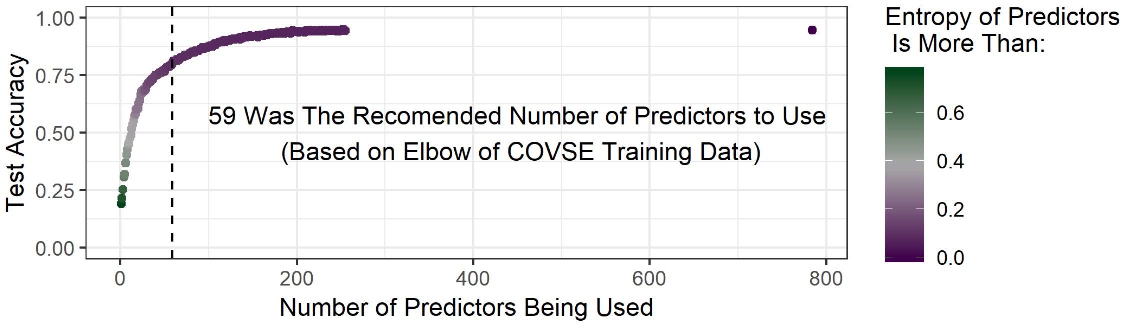
5.2.4.5 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.5 (i.e., COVSE trained on 133 images)



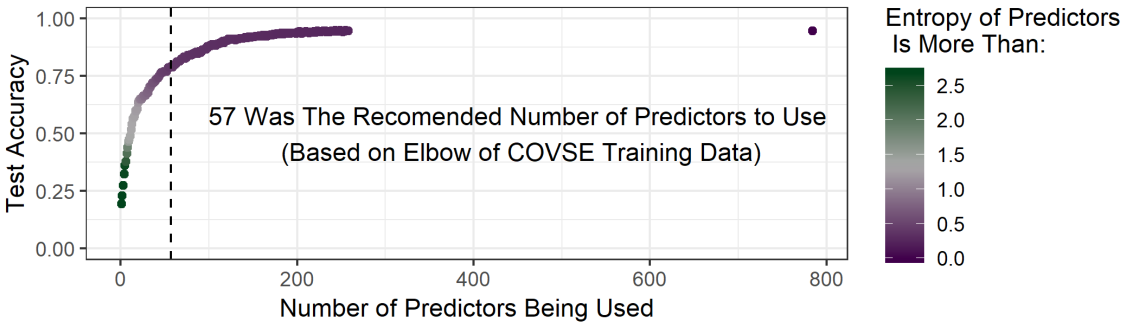
5.2.4.6 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.6 (i.e., COVSE trained on 452 images)



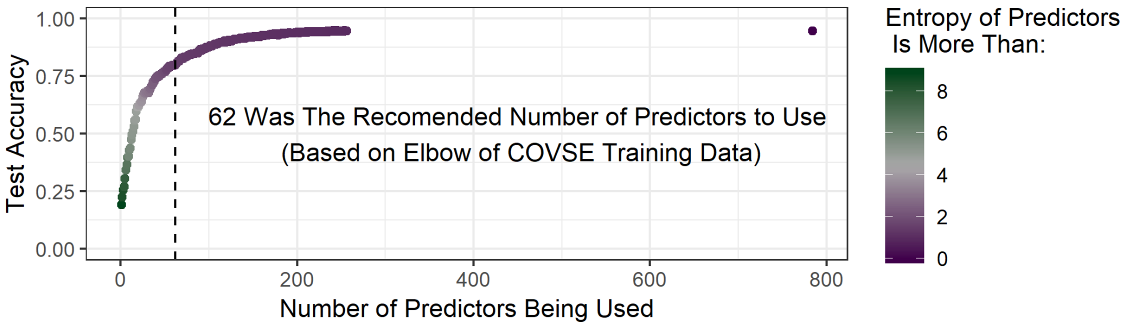
5.2.4.7 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.7 (i.e., COVSE trained on 1533 images)



5.2.4.8 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.8 (i.e., COVSE trained on 5205 images)



5.2.4.9 MNIST Prediction Accuracy; Predictors Chosen Based on Calculations in Section 5.2.2.9 (i.e., COVSE trained on 17617 images)



5.2.5 Analysis

Seeing the experiments juxtaposed against each other, we note that the suggested number of features to include tended to match the elbow of the plotted accuracy data. This suggests that we will see significantly diminishing returns for any predictors included beyond that indicated by the elbow. Furthermore, due to the exponential reduction of predictors while still maintaining quality accuracy, we conclude that this method of feature selection is viable for the MNIST data set.

5.3 Vectorwise Observation for Variable Selection with Entropy (VOVSE)

In section 5.1 we discussed a componentwise entropy based method to select predictors. Suppose we want to consider the relative entropy of each vector in tandem. For example, a photo of a forest fire might pack more information (that is, have a higher entropy) than a small candle against a dark background. The entropy of an image may not be realized when considering an aggregation of pixel entropy individually. At the same time, considering individual component entropy is not dismissable either, as shown in our experiments in section 5.2.2. Hence, our initial motivation is to first define and find the internal entropy of each prediction vector as a whole, while simultaneously taking into account the entropy of individual components. We then may use that entropy to provide insights into feature selection.

Suppose we have a set of vectors $\vec{v}_1 = \begin{bmatrix} a_1 \\ b_1 \\ \vdots \\ n_1 \end{bmatrix}, \vec{v}_2 = \begin{bmatrix} a_2 \\ b_2 \\ \vdots \\ n_2 \end{bmatrix}, \cdots, \vec{v}_k = \begin{bmatrix} a_k \\ b_k \\ \vdots \\ n_k \end{bmatrix}$ where $\vec{v}_i \in \mathbb{R}^n$, each vector $\vec{v}_i$ is chosen i.i.d. from our universe of interest and k is a sufficiently large sample size. Once again, our initial motivation is to find the internal entropy of each vector $\vec{v}_\iota$, denoted $H_{\vec{v}_\iota}$, as a whole. To do so, recall the following definition for the conditional Shannon entropy of X given Y, denoted $H(X|Y)$, where X and Y are random variables:

$$H(X|Y) = -\sum_{x \in X, y \in Y} \Pr(X = x, Y = y) \cdot \log_2 \left(\frac{\Pr(X = x, Y = y)}{\Pr(Y = y)} \right)$$

Let X represent the set of entries in a vector, and Y the set of vector indices. Under this motivation, we can define for some vector $\vec{v}_\iota \in \mathbb{R}^n$

$$H_{\vec{v}_\iota} = -\sum_{\substack{x \in \{a_\iota, b_\iota, \cdots, n_\iota\}, \\ y \in \{1, 2, \cdots, n\}}} \Pr(X = x, Y = y) \cdot \log_2 \left(\frac{\Pr(X = x, Y = y)}{\Pr(Y = y)} \right)$$

Since the number of entries of each vector remains fixed at n, we can assert the probability of $P(Y = y) = \frac{1}{n}$ for all $y \in Y$. This immediately simplifies our construction to

$$H_{\vec{v}_\iota} = -\sum_{\substack{x\in\{a_\iota,b_\iota,\cdots,n_\iota\},\\ y\in\{1,2,\cdots,n\}}} \Pr(X=x,Y=y)\cdot\log_2\left(\frac{\Pr(X=x,Y=y)}{\Pr(Y=y)}\right)$$

$$H_{\vec{v}_\iota} = -\sum_{\substack{x\in\{a_\iota,b_\iota,\cdots,n_\iota\},\\ y\in\{1,2,\cdots,n\}}} \Pr(X=x,Y=y)\cdot\log_2\left(\frac{\Pr(X=x,Y=y)}{\frac{1}{n}}\right)$$

$$H_{\vec{v}_\iota} = -\sum_{\substack{x\in\{a_\iota,b_\iota,\cdots,n_\iota\},\\ y\in\{1,2,\cdots,n\}}} \Pr(X=x,Y=y)\cdot\log_2\left(n\cdot\Pr(X=x,Y=y)\right)$$

The crux is in determining $\Pr(X=x,Y=y)$. To approximate this function, we may utilize a *multivariate kernel density estimation* (mKDE). That is to say, we assert that the PDF of $\Pr(X=x,Y=y)$ is the same as the mKDE over X and Y.

5.3.1 Multivariate Kernel Density Estimation

Formally, we define a multivariate kernel density estimator, $\vec{f_B}(x,y) : X\times Y\subset\mathbb{R}^2\to[0,1]\subset\mathbb{R}$, such that $\int_X\int_Y \vec{f_B}(x,y)\,dy\,dx = 1$, as a generalization of the univariate kernel density estimator in the following way:

$$\vec{f_B}(x,y) = \frac{1}{|S|}\sum_{\vec{s}\in S} K_B\left(\begin{bmatrix} x \\ y \end{bmatrix} - \vec{s}\right)$$

where

- $S = \left\{\begin{bmatrix} a_1 \\ 1 \end{bmatrix},\begin{bmatrix} a_2 \\ 1 \end{bmatrix},\cdots,\begin{bmatrix} a_k \\ 1 \end{bmatrix},\quad \begin{bmatrix} b_1 \\ 2 \end{bmatrix},\begin{bmatrix} b_2 \\ 2 \end{bmatrix},\cdots,\begin{bmatrix} b_k \\ 2 \end{bmatrix},\quad \cdots,\quad \begin{bmatrix} n_1 \\ n \end{bmatrix},\begin{bmatrix} n_2 \\ n \end{bmatrix},\cdots,\begin{bmatrix} n_k \\ n \end{bmatrix}\right\}$
- $B\in\mathbb{R}^{2\times2}$, with B symmetric and positive definite, is our multivariate bandwidth (smoothing) parameter, found using cross validation.
- $K_B(\vec{x}) : \mathbb{R}^n\to\mathbb{R}$ is a scaled standard multivariate Gaussian normal kernel, defined as:

$$K_B(\vec{x}) = \frac{1}{2\pi}\cdot|B|^{-\frac{1}{2}}\cdot\exp(-\frac{1}{2}\vec{x}^T B^{-1}\vec{x})$$

It should be noted that other kernels have been found to have success in the multivariate case, such as those pioneered by Breiman et al. in [48], but for simplicity we will hold to the multivariate Gaussian kernel here. Hence, with this machinery in place, we now may compute an ordered pair of the form $(H_{\vec{v}_i}, \vec{v}_i)$ for all our initially given vectors.

While $H_{\vec{v}_i}$ is useful as a descriptive attribute in-and-of-itself, the real benefit is utilizing the power of linear regression. At this point, we may set up a linear model of the form:

$$H_{\vec{v}_i} \sim a_i + b_i + \cdots + n_i$$

Given that $H_{\vec{v}_i}$ represents the intrinsic entropy of the vector in juxtaposition with other vectors in $\{\vec{v}_1, \cdots, \vec{v}_k\}$, we can utilize it as a response variable for which $a_i, \cdots, n_i$ act as predictors. Consequentially, we may now utilize techniques such as LASSO, ridge regression, backwards elimination, or any other linear feature selection technique to identify which elements of our vectors to leave in.

CHAPTER 6
CONCLUDING REMARKS AND FUTURE WORK

The panorama of discrete and combinatorial exploration via machine learning is an area ripe for further investigation. The results presented in this thesis indicate that, when represented by appropriate invariants, graphs and matrices (particularly matrices from non-field algebraic structures) can be used successfully in supervised learning algorithms. This tends to hold true even for computationally difficult problems, as noted when we examined computationally intractable problems in Chapters 3 and 4.

In particular, we conclude that by representing Boolean matrices as a vector described by the binary representation of a particular indexing (described in section 2.8), random forests were able to reconstruct 99% of all matrix products from only 10% of samples in our universe of data (reported in section 3.6). This insight indicates the feasibility of solving the Boolean factorization problem in the amortized case in polynomial time, and to a high degree of accuracy.

We further conclude, from representations discussed in Chapter 4, that graphs are also able to be appropriately articulated as predictors for certain machine learning techniques. Articulations for which machine learning techniques do well in supervised learning contexts. For example, we noted in section 4.5 that with just 5% of the data in our universe of tournaments on 5 vertices, we were able to gain over 95% accuracy in determining the feedback arcset number.

The crux is in determining which invariant representations are most appropriate as input for the supervised problem at hand. While this thesis explored a handful of problems and input articulations, there is much more work which needs to be done to understand which invariant representations work well when supervised against problems in a generic sense, versus those invariant representations which are idiosyncratically beneficial to only a particular problem. Exploring more structures, and the invariant articulations of those structures, is certainly an area which needs to be surveyed.

Another area of investigation is that of scalablity. We are under no pretense that the inputs utilized in Chapters 3 and 4 are large. After all, in Chapter 3 matrices were of dimension 3×3. In Chapter 4 the largest matrix processed was of dimension 5×5. Although the search spaces undeniably expand exponentially as matrix dimension increases, this hurdle of combinatorial explosion must become tractable if machine learning techniques are to be utilized in practice. More experiments done on higher dimensions should be conducted. In parallel, currently nonexistent techniques to prune a search space composed of highly structured data need to be developed.

We addressed one facet of the challenge of scalablity in Chapter 5 by demonstrating that entropy based methods have merit in feature selection, and hence data reduction. For example, we showed in section 5.2.2 that using COVSE we could, using only several dozen MNIST training instances, parry our feature set down to 87 or fewer predictors; a reduction of $1 - \dfrac{87}{28 \cdot 28} \implies 88.9\%$ or more. Other authors have expounded much more on these ideas of entropy based feature reduction. Where they may have use, and where future work needs to be done, is in their efficacy in paring down features used to represent graphs.

As is, this thesis serves primarily as a “proof of concept” approach to find solutions for many currently intractable problems in their amortized cases. We are hopeful that the methodologies outlined here provide fodder to the lector in solving similarly flavored problems in other fields, such as in topology (e.g., knot and braid classification) and algebraic geometry (e.g., cylindrical algebraic decomposition). ■

REFERENCES

[1] M. Kutz, "The complexity of boolean matrix root computation," *Theoretical Computer Science*, vol. 325, no. 3, pp. 373 – 390, 2004, selected Papers from COCOON 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397504004165

[2] R. M. Karp, *Reducibility among Combinatorial Problems.* Boston, MA: Springer US, 1972, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2_9

[3] F. Wagner, "Hardness results for tournament isomorphism and automorphism," in *Mathematical Foundations of Computer Science 2007*, L. Kučera and A. Kučera, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 572–583.

[4] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, 1996.

[5] G. Dantzig, "The simplex method," *RAND Corporation*, p. 891, 1956.

[6] L. Khachiyan, "Polynomial algorithms in linear programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 20, no. 1, pp. 53 – 72, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0041555380900610

[7] E. R. Scheinerman and D. H. Ullman, *Fractional graph theory: a rational approach to the theory of graphs.* Courier Corporation, 2011.

[8] I. Choi, J. Kim, and S. O, "The difference and ratio of the fractional matching number and the matching number of graphs," *Discrete Mathematics*, vol. 339, no. 4, pp. 1382 – 1386, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0012365X15004380

[9] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128 – 138, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022314X80900840

[10] E. W. Weisstein, "Laplacian matrix." Wolfram Mathworld, 2019. [Online]. Available: http://mathworld.wolfram.com/LaplacianMatrix.html

[11] J. Hopcroft and R. Tarjan, "Efficient planarity testing," *J. ACM*, vol. 21, no. 4, pp. 549–568, Oct. 1974. [Online]. Available: http://doi.acm.org/10.1145/321850.321852

[12] C. Thomassen, "The graph genus problem is np-complete," *J. Algorithms*, vol. 10, no. 4, pp. 568–576, Dec. 1989. [Online]. Available: https://doi.org/10.1016/0196-6774(89)90006-0

[13] N. Abboud, M. Grötschel, and T. Koch, "Mathematical methods for physical layout of printed circuit boards: An overview," *OR Spectrum*, vol. 30, pp. 453–468, 06 2008.

[14] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph – static and dynamic graph drawing tools," in *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pp. 127–148.

[15] J. N. Warfield, "Crossing theory and hierarchy mapping," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. SMC-7, pp. 505 – 523, 08 1977.

[16] M.-J. Carpano, "Automatic display of hierarchized graphs for computer-aided decision analysis," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. SMC-10, pp. 705 – 715, 12 1980.

[17] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, pp. 109–125, 1981.

[18] C. B. Moler, "Numerical computing with matlab, revised reprint." Philadelphia: SIAM, 2008, ch. 0. [Online]. Available: http://epubs.siam.org/doi/book/10.1137/1.9780898717952

[19] D. Bini, L. Gemignani, and F. Tisseur, "The ehrlich–aberth method for the nonsymmetric tridiagonal eigenvalue problem," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, 01 2005.

[20] K. Fernando, "Computation of exact inertia and inclusions of eigenvalues (singular values) of tridiagonal (bidiagonal) matrices," *Linear Algebra and its Applications*, vol. 422, no. 1, pp. 77 – 99, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0024379506004228

[21] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008. [Online]. Available: http://www.jmlr.org/papers/v9/vandermaaten08a.html

[22] L. van der Maaten, E. O. Postma, and H. J. van den Herik, "Dimensionality reduction: A comparative review," 2008.

[23] K. R. Moon, D. van Dijk, Z. Wang, S. Gigante, D. B. Burkhardt, W. S. Chen, K. Yim, A. van den Elzen, M. J. Hirn, R. R. Coifman, N. B. Ivanova, G. Wolf, and S. Krishnaswamy, "Visualizing structure and transitions for biological data exploration," *bioRxiv*, 2019. [Online]. Available: https://www.biorxiv.org/content/early/2019/04/04/120378

[24] C. Damm, K. H. Kim, and F. Roush, "On covering and rank problems for boolean matrices and their applications," in *Computing and Combinatorics*, T. Asano, H. Imai, D. T. Lee, S.-i. Nakano, and T. Tokuyama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 123–133.

[25] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2015. [Online]. Available: http://arxiv.org/abs/1511.06434

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[27] F. Chollet *et al.*, "Keras," https://github.com/keras-team/keras, 2019.

[28] M. Kotila, "Talos," https://github.com/autonomio/talos, 2019.

[29] C. Ye, R. C. Wilson, and E. R. Hancock, "A jensen-shannon divergence kernel for directed graphs," in *Structural, Syntactic, and Statistical Pattern Recognition*, A. Robles-Kelly, M. Loog, B. Biggio, F. Escolano, and R. Wilson, Eds. Cham: Springer International Publishing, 2016, pp. 196–206.

[30] H. G. Landau, "On dominance relations and the structure of animal societies: Iii the condition for a score structure," *The bulletin of mathematical biophysics*, vol. 15, no. 2, pp. 143–148, Jun 1953. [Online]. Available: https://doi.org/10.1007/BF02476378

[31] D. C. McGarvey, "A theorem on the construction of voting paradoxes," *Econometrica*, vol. 21, no. 4, pp. 608–610, 1953. [Online]. Available: http://www.jstor.org/stable/1907926

[32] L. Babai, "Graph isomorphism in quasipolynomial time," *CoRR*, vol. abs/1512.03547, 2015. [Online]. Available: http://arxiv.org/abs/1512.03547

[33] S. Bessy, F. V. Fomin, S. Gaspers, C. Paul, A. Perez, S. Saurabh, and S. Thomassé, "Kernels for feedback arc set in tournaments," *CoRR*, vol. abs/0907.2165, 2009. [Online]. Available: http://arxiv.org/abs/0907.2165

[34] B. D. McKay and A. Piperno, "Practical graph isomorphism, ii," *Journal of Symbolic Computation*, vol. 60, pp. 94 – 112, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0747717113001193

[35] P. Mitra, C. A. Murthy, and S. K. Pal, "Unsupervised feature selection using feature similarity," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, pp. 301–312, 2002.

[36] A. Mariello and R. Battiti, "Feature selection based on the neighborhood entropy," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, pp. 6313–6322, 2018.

[37] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 7 1948. [Online]. Available: https://ieeexplore.ieee.org/document/6773024/

[38] M. Rosenblatt, "Remarks on some nonparametric estimates of a density function," *Ann. Math. Statist.*, vol. 27, no. 3, pp. 832–837, 09 1956. [Online]. Available: https://doi.org/10.1214/aoms/1177728190

[39] E. Parzen, "On estimation of a probability density function and mode," *Ann. Math. Statist.*, vol. 33, no. 3, pp. 1065–1076, 09 1962. [Online]. Available: https://doi.org/10.1214/aoms/1177704472

[40] B. W. Silverman, *Density estimation for statistics and data analysis.* London: Chapman & Hall, 1986.

[41] D. W. Scott, "On optimal and data-based histograms," *Biometrika*, vol. 66, no. 3, pp. 605–610, 12 1979. [Online]. Available: https://doi.org/10.1093/biomet/66.3.605

[42] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[43] S.-H. Lim, S. Young, and R. Patton, "An analysis of image storage systems for scalable training of deep neural networks," 04 2016.

[44] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[45] B. A. Hanson, *ChemoSpecMarkeR: Functions to identify biomarkers in nmr spectra*, 2016, r package: version 0.1.44. [Online]. Available: https://github.com/bryanhanson/ChemoSpecMarkeR

[46] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a kneedle in a haystack: Detecting knee points in system behavior," 07 2011, pp. 166 – 171.

[47] B. Bowerman, *Essentials of Business Statistics.* McGraw Hill Education, 2014. [Online]. Available: https://books.google.com/books?id=sHRzCgAAQBAJ

[48] L. Breiman, W. Meisel, and E. Purcell, "Variable kernel estimates of multivariate densities," *Technometrics*, vol. 19, no. 2, pp. 135–144, 1977. [Online]. Available: https://amstat.tandfonline.com/doi/abs/10.1080/00401706.1977.10489521