OPTIMIZACIÓN DE ALGORITMOS DE AGRUPAMIENTO APLICADOS A METAGENÓMICA USANDO BIG DATA

JULIÁN VANEGAS PIEDRAHITA

Trabajo de grado para optar al título de

INGENIERO DE SISTEMAS Y COMPUTACIÓN

Isis Bonet Cruz Ph.D

UNIVERSIDAD EIA
INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
ENVIGADO
2018

CONTENIDO

pág.

LISTA DE TABLAS

pág.

LISTA DE FIGURAS

pág.

LISTA DE ANEXOS

pág.

RESUMEN

En el campo de la metagenómica, está incrementando el uso de la metagenómica balística como proceso de extracción de cadenas de aminoácidos de microorganismos previamente no identificados, basándose en muestras ambientales de diversas fuentes. Estas cadenas de aminoácidos, debido al proceso de extracción, son separadas en subcadenas de tamaños variables que luego buscan ser comparadas e identificadas con una base de datos para no sólo determinar qué especies ya reconocidas habitan en las muestras tomadas, sino también qué porciones de estas secuencias de aminoácidos no han sido previamente categorizadas.

En búsqueda de que este método de identificación produzca mayores resultados, se usan algoritmos de agrupamiento como facilitadores en el proceso de identificación de las diferentes especies. Estos algoritmos agrupan secuencias de aminoácidos que tienen cierto grado de similitud, produciendo clústeres de subcadenas, para que luego estos puedan ser comparados en grupo y ser más rápidamente analizadas.

Con el objetivo de mejorar los tiempos de ejecución, se usaron plataformas como Apache Spark y TensorFlow, que dentro de sus librerías incluyen implementaciones nativas de estos algoritmos de agrupamiento. A partir de estas librerías se implementó el K-means iterativo que fue usado como punto de comparación.

En los resultados se puede apreciar que el uso de K-means Iterativo mejora la pureza comparado con la alternativa de una sola iteración, para el caso de uso de una base de metagenómica usando los 4mer como rasgos, y usando el coseno como distancia. Debido a este último punto, y a que la implementación de Apache Spark de K-means no tiene la distancia coseno, se utilizó TensorFlow principalmente para la toma de resultados. El uso de TensorFlow muestra una mejora en general de tiempos de ejecución, siendo mucho más significativa en el caso de K-means Iterativo, teniendo como desventaja que requiere mucho más poder de procesamiento.

Palabras clave: metagenómica, tensorflow, spark, k-means, clusterización

ABSTRACT

In the field of metagenomics, the use of ballistic metagenomics as the extraction process of amino acids chains of previously unidentified microorganisms, using environmental samples from diverse sources as its base. These amino acid chains, due to the extraction process, are separated in sub-chains of variable sizes, which will be used afterwards for comparison and identification with a database to not only determine which of the already recognized species are present in the samples, but also what portion of these amino acid sequences have not been previously categorized.

Seeking for this method for identification to produce better results, clustering algorithms will be used as enablers in the identification process for the different species. These algorithms group amino acid chains with a certain similarity rate, producing sub-chain clusters, so these can then be compared in group and be analyzed faster.

Platforms like Apache Spark and TensorFlow were used with the objective of reducing the execution times, as they include native implementations of these clustering algorithms in their libraries. With these libraries as a base, an implementation of Iterative K-means was created and then used as a comparison point.

In the results it is apparent that Iterative K-means improves the cluster purity compared to the single-iteration alternative, using a metagenomics base with 4mer as features, and using the cosine distance. Due to the latter, and the fact that the K-means implementation in Apache Spark doesn't include the cosine distance, TensorFlow was the primary platform used for the gathering of the results. The use of TensorFlow improved the execution times in general, with a more significative difference in the case of the Iterative K-means, with the disadvantage that it requires much more processing power.

Palabras clave: metagenomics, tensorflow, spark, k-means, clusterization

INTRODUCCIÓN

Los microorganismos, a pesar de ser los seres vivos más abundantes y dominar nuestro planeta en prácticamente todos los ambientes, siguen siendo en su gran parte un misterio para su investigación. En un estudio reciente se predijo que la tierra podría albergar aproximadamente 1 billón (10^{12}) de especies microbianas, de las cuales sólo se han identificado un 0.01% de estas. (Locey & Lennon, 2016). Muchos de estos microorganismos son relevantes en muchas áreas de estudio humano, particularmente en la agricultura y medicina. El interés por incrementar este porcentaje de identificación de especies microbianas ha crecido en los últimos años, ya que podría traer soluciones a problemas que nuestra especie humana todavía sufre día a día.

El aumento de la popularidad del estudio de los microorganismos ha llevado a que se busquen cada vez formas más efectivas de su identificación. Sin embargo, un obstáculo muy grande en este estudio es la limitación de estas formas de vida microbianas de sobrevivir en laboratorios, y que no permiten ser cultivadas artificialmente, o requieren de ciertas características en su ambiente que lo hacen difícil o poco viable. A raíz de esto, la disciplina de la metagenómica busca permitir el estudio genómico de microorganismos que no permiten ser cultivados artificialmente, mediante la secuenciación de los microbios directamente desde su hábitat ambiental, sin necesidad de su cultivo en un laboratorio. (Wooley, Godzik, & Friedberg, 2010).

El análisis de los resultados a partir de la metagenómica conlleva retos grandes, debido a que mientras en los cultivos del laboratorio se puede garantizar que las secuencias genómicas pertenecen a una sola especie, la metagenómica regresa secuencias pertenecientes a grupos homogéneos de microorganismos presentes en la muestra ambiental tomada, sin distinción de los grupos a los que pertenece, y puede llegar a contener más de 10000 especies presentes en los resultados. (Wooley, Godzik, & Friedberg, 2010)

Una posible solución a este obstáculo es el uso de las bases de datos existentes para filtrar subcadenas en los microorganismos sobre los que se tiene un registro, y los que no han sido identificados hasta el momento. Sin embargo, esto también lleva a otro reto que se debe solucionar, la velocidad de la comparación teniendo en cuenta no sólo la gran cantidad de subcadenas regresadas por el proceso de metagenómica, sino también el volumen masivo de las bases de datos de especies identificadas, dificultando mucho un análisis rápido y eficiente de los resultados.

La metagenómica, específicamente la metagenómica balística, está siendo utilizada actualmente en el Centro Nacional de Secuenciación Genómica de la Universidad de Antioquia en la búsqueda de la identificación de cadenas de aminoácidos provenientes de los microorganismos presentes en diferentes muestras del ambiente. Estas cadenas luego son comparadas con una base de datos de todas las cadenas de aminoácidos conocidas, para comprobar si hay alguna coincidencia sobre esta, o si la secuencia original no está indexada en el momento. (I. Bonet Cruz, comunicación personal, 9 de marzo de 2016).

Para el caso puntual analizado, esta comparación puede tomar de días a semanas debido al tamaño de la base de datos que se toma como referencia, y debido a que inicialmente las cadenas se comparaban una por una. Usando algoritmos de agrupamiento e inteligencia artificial, es posible agrupar cadenas que posean características similares, y que posiblemente pertenezcan al mismo organismo. Usando algoritmos como K-means con diversas distancias y tomando diferentes características se realizó un agrupamiento de subcadenas, buscando que en cada uno de los clústeres se pudiera identificar microorganismos específicos. Estos agrupamientos de clústeres luego son analizados en conjunto, en lugar de sobre cada secuencia particular.

Mediante el uso de plataformas de computación distribuida y aprendizaje de máquina, como Apache Spark y TensorFlow, es posible lograr una implementación más optimizada de los algoritmos de agrupamiento utilizados en trabajos antecedentes. Estas implementaciones logran hacer un procesamiento más veloz del gran volumen de datos que se requieren procesar, usando computación en la nube, computación masiva y algoritmos optimizados para esto.

Se llegó a la conclusión de que el algoritmo de K-means Iterativo planteado por trabajos antecedentes, como las tesis de Widerman Montoya, (Montoya Ramírez, 2014) y de Adriana Escobar (Escobar Vasco, 2015), sí produce mejores resultados comparado con la alternativa de una sola iteración. Y mediante el uso de plataformas como Apache Spark y herramientas como TensorFlow, además de la computación en la nube y masiva, se pudo reducir significativamente el tiempo de ejecución de esto algoritmos.

1. PRELIMINARES

1.1 PLANTEAMIENTO DEL PROBLEMA

Los análisis sobre las secuencias de ADN que están siendo encontradas mediante el uso de la metagenómica les permiten a los investigadores determinar no sólo los ambientes típicos en los que se encuentran diversos microorganismos que ya han sido identificados, sino que también ayuda a identificar organismos que no habían sido identificados previamente. Sin embargo, para que este tratamiento de datos pueda ser utilizado con mayor frecuencia, debe entregar resultados en un tiempo razonable. Mientras más corto sea el tiempo más análisis se pueden dar y se puede llegar a más conclusiones. (I. Bonet Cruz, comunicación personal, 9 de marzo de 2016).

Es por esta razón que se busca la optimización de la velocidad que se tardan corriendo los algoritmos de agrupamiento usados en la secuenciación metagenómica. La optimización de los algoritmos que están siendo usados para realizar este tratamiento de las secuencias, siendo enfocado especialmente al procesamiento distribuido y el uso de plataformas de aprendizaje de máquina, puede permitir que los resultados sean entregados en menos tiempo y, al estar implementado con inteligencia artificial para el agrupamiento de las secuencias, es posible que entregue resultados con un mayor nivel de confiabilidad. La validación con casos de prueba permitirá verificar que los resultados obtenidos sean adecuados, y que la diferencia de tiempo sí sea significativa con respecto al proceso original.

La comparación entre los diferentes métodos de optimización de los algoritmos permitirá encontrar una solución que no sólo sea actual en el estado del arte, sino que se busca que también sea apta para el nivel de procesamiento que es requerido por el Centro Nacional de Secuenciación Genómica de la Universidad de Antioquia,

1.2 OBJETIVOS DEL PROYECTO

1.2.1 Objetivo General

Optimizar el tiempo de ejecución de algoritmos de agrupamiento, para bases de datos de muestras metagenómicas, utilizando Big Data y programación distribuida.

1.2.2 Objetivos Específicos

- Identificar los métodos de programación en Apache Spark y TensorFlow para almacenar e implementar los algoritmos de agrupamiento.

- Diseñar e implementar dos variantes de un método de agrupamiento iterativo, basado en K-means, una con Spark y otra con TensorFlow.

- Comparar tiempos de ejecución y resultados de las implementaciones de optimización de los algoritmos, con una base de metagenómica.

1.3 MARCO DE REFERENCIA

1.3.1 Antecedentes

El estudio de algoritmos para el procesamiento y análisis de secuencias genómicas y de cadenas de ADN ha tomado gran fuerza en los últimos años, gracias a la búsqueda de reducir el porcentaje de especies de microorganismos que no han sido identificados. Muchos de estos estudios buscaban maneras de obtener resultados sobre la composición taxonómica de las cadenas de aminoácidos, creando herramientas para facilitar este procesamiento. Estas herramientas facilitan el análisis de muestras de secuencias de ADN encontradas, buscando hacer una relación con las bases de datos existentes de especies microbianas.

El enfoque principal del desarrollo de estas nuevas herramientas es hacia algoritmos que clasifican microorganismos a partir de conocimiento ya adquirido previamente, es decir, algoritmos de aprendizaje supervisado. La aplicación buscada en este trabajo y por las tesis antecedentes de este tienen un enfoque no supervisado, pero estas herramientas anteriores proveen de una base de conocimiento que puede ser aplicada a la metagenómica, especialmente en el hecho de que los resultados del agrupamiento y clasificación de microorganismos a partir de sus cadenas de ADN han sido efectivos y relevantes para desarrollos futuros.

Dröge, Gregor, y McHardy (2014) utilizaron una serie de programas de análisis taxonómico llamado Taxator-tk para asignar secuencias de nucleótidos a diferentes grupos aproximándolos a sus vecinos evolutivos. Los resultados que obtuvieron fueron satisfactorios y Taxator-tk fue exitoso y preciso en la asignación taxonómica de diferentes secuencias de prueba, también clasificando los organismos en sus diferentes familias taxonómicas. Taxator-tk tiene también una implementación utilizando computación paralela.

Moheedhar, Reddy, Haque Mohammed, y S. Mande (2014) presentaron MetaCAA, un ensamblador específico para metagenomas, que utiliza un procedimiento de ensamblaje ayudado por clústeres para mejorar la eficiencia en el ensamblaje de cóntigos. Después de evaluar el rendimiento de MetaCAA haciendo pruebas con metagenomas reales y simulados, pudieron llegar a la conclusión de que producía resultados con una mayor pureza y longitud comparándolo con herramientas basadas en ensambladores ayudados por un solo genoma.

En 2013 se propuso el uso de Quikr, una herramienta cuadrática, iterativa, basada en K-mer, desarrollada por Koslicki, Foucart, y Rosen (2013). Esta propone reconstruir comunidades bacterianas computando asignaciones taxonómicas y sus proporciones en la muestra y usando una técnica de optimización motivada desde la teoría matemática de *compressive sensing*. En sus resultados, mostraron que la herramienta no era afectada por la presencia de “quimeras” en los datos, removiendo la necesidad de filtrar éstas antes de realizar la clasificación. También se pudo apreciar que tenía menos error y mejor velocidad que otras técnicas de asignación taxonómica como el Naïve Bayesian Classifier usado por el Ribosomal Database Project.

Estos estudios y herramientas orientados hacia la genómica son una base también para la metagenómica, pudiendo aplicar los mismos y analizar resultados sobre bases que no han sido cultivadas, tomadas directamente desde muestras ambientales. Se pudo determinar que el uso de algoritmos de agrupamiento con diferentes distancias lleva a producir resultados para el análisis de subcadenas genómicas de microorganismos, buscando que se puedan crear grupos de secuencias similares y usar esto como base en el análisis metagenómico.

En el trabajo de Widerman Montoya, basado a partir de estudios similares, se realizó una exploración de algoritmos de inteligencia artificial aplicados al agrupamiento de secuencias metagenómicas, implementando una variación del algoritmo de K-means, llamado K-means iterativo, que a partir de una primera iteración de K-means se guardan los clústeres más puros y con el resto el algoritmo es aplicado nuevamente, produciendo al final como resultado clústeres más puros que la variante original. (Montoya Ramírez, 2014)

En el trabajo consecuente realizado por Adriana Escobar, a partir de los resultados del trabajo de Widerman Montoya se buscó realizar una alteración al algoritmo, en el que los clústeres con mayor distancia promedio a los demás se reagrupan para formar un grupo nuevo, basándose también en diferentes características de las secuencias genómicas pertenecientes a ellos. (Escobar Vasco, 2015)

En los últimos años, el uso de herramientas de computación en la nube y 'Big Data' se ha desarrollado también en el campo de la metagenómica, llevando a mejoras en el procesamiento masivo de los datos que son producidos y necesitan ser analizados a partir del estudio de los genomas. En 2013, O'Driscoll, Daugelaite, y Sleator realizaron un análisis sobre el papel que puede llegar a tener Big Data y la computación en la nube para el análisis genómico, llegando a la conclusión que estas nuevas tecnologías serán de gran importancia para el procesamiento de volúmenes masivos de datos. (O'Driscoll, Daugelaite, & Sleator, 2013)

Herramientas específicas también han sido desarrolladas en busca de aprovechar estas tecnologías. En el 2017, un grupo de académicos desarrolló MetaSpark, una herramienta de procesamiento distribuido de secuencias metagenómicas basada en Apache Spark. (Zhou, y otros, 2017). En sus resultados pudieron ver que el uso de computación distribuida a partir de Apache Spark lograba un incremento de secuencias analizadas en el mismo tiempo, comparado con otras herramientas como SOAP2, BWA y LAST.

1.3.2 Metagenómica

La metagenómica, en su definición, es la ciencia que tiene como objetivo el estudio de conjuntos de genomas pertenecientes a un determinado entorno tomando muestras directamente del ambiente. Mediante el uso de tecnologías genómicas y bioinformática permite el acceso al contenido genético de comunidades enteras de organismos sin necesidad de ser cultivadas en un laboratorio. (Thomas, Gilbert, & Meyer, 2012)

Uno de los problemas más grandes que esta ciencia intenta resolver es la secuenciación del ADN de microorganismos que no han sido identificados previamente, debido a la dificultad de su estudio sin llegar a ser cultivados en un ambiente controlado. También

ayuda a controlar el sesgo que es introducido a las muestras y experimentos cuando estos son producidos en un laboratorio, y permitió descubrir nuevas líneas de vida microbiana. (Handelsman, 2004).

Las herramientas de secuenciación producen un rompecabezas de fragmentos de secuencias, que luego serán ensambladas con un proceso de superposición usando secuencias largas llamadas cóntigos. El mayor problema viene después, ya que se debe obtener el genoma completo a partir de estos cóntigos, y teniendo en cuenta que la muestra ambiental contiene diversos individuos. (Bonet, Montoya, Mesa Múnera, & Alzate, 2014)

La metagenómica requiere un proceso de *binning* (discretización) en el que los cóntigos de diferentes especies serán asignados a su correspondiente grupo filogenético. Estos métodos pueden clasificarse en basados en similitud, y basados en composición. Una de las estrategias más utilizadas para este fin es la de usar métodos de agrupamiento en clústeres. Los métodos también pueden ser diferenciados según si tienen un aprendizaje supervisado o no supervisado. Los métodos supervisados suelen ser más precisos, pero más lentos debido a la cantidad de organismos presentes en la muestra. Esto quiere decir que si un método de discretización recibe una muestra que se sabe es puramente de un solo organismo, el proceso se vuelve más rápido. (Bonet, Montoya, Mesa Múnera, & Alzate, 2014)

En un estudio anteriormente realizado sobre la comparación de diferentes métodos de discretización se encontró que una aplicación iterativa y no supervisada de un algoritmo de agrupamiento basado en *k*-means mejora la pureza de los clústeres encontrados con respecto a una aplicación simple del mismo algoritmo, y es un método prometedor para el uso futuro en problemas de metagenómica. (Bonet, Montoya, Mesa Múnera, & Alzate, 2014) Es en las implementaciones de este algoritmo que se aplicará la optimización buscada mediante computación distribuida y plataformas de aprendizaje de máquina.

1.3.3 Computación Paralela y Computación Distribuida

En el sentido más simple, la computación paralela es el uso de múltiples recursos de cómputo para resolver un problema computacional. Para realizar este proceso, se deben seguir una serie de pasos. En primer lugar, el problema debe ser separado en partes que pueden ser resueltas simultáneamente. Luego, cada parte es separada en una serie de instrucciones. Las instrucciones de cada parte son ejecutadas en diferentes procesadores, y se implementa un mecanismo de control y coordinación sobre todo el sistema. (Barney, 2016)

Para poder implementar computación paralela sobre un problema dado, este debe cumplir ciertas características: debe ser posible separarlo en partes discretas que puedan resolverse simultáneamente; debe poder ejecutar múltiples instrucciones en un momento del tiempo; y debe ser resulto en menos tiempo con múltiples recursos de cómputo comparado con un solo recurso. Recursos computacionales pueden ser un solo computador con múltiples procesadores o núcleos, o un número arbitrario de computadores conectados en una red.

La computación distribuida se puede tomar como un tipo de computación paralela, sólo que llevada a un nivel más general. En lugar de repartir instrucciones en diferentes procesadores, la idea de la computación distribuida es la repartición de tareas a diferentes nodos de un clúster computacional, cada uno siendo compuesto de uno o varios servidores que harán operaciones localmente para luego ser enviadas al orquestador que se encarga de recibir estos resultados y agruparlos. Todo este clúster corre como un solo sistema que puede estar en una red local, o estar distantes geográficamente y conectados por una red de área. (IBM, 2018)

Los requerimientos de los problemas posibles para ambas son muy similares, la diferencia principal es sobre su implementación.

1.3.4 Apache Spark

Apache Spark es un motor de procesamientos open source basado en velocidad, facilidad de uso, y análisis sofisticado de datos. Desde su inicio, Spark ha visto gran adopción por empresas en un gran rango de industrias. Se basa en proporcionar un ecosistema que facilita a los usuarios el uso de diferentes tecnologías en el análisis y procesamiento de datos, usando conceptos como *Machine Learning* (Aprendizaje de máquinas) y *Graph Computation* (Computación de Grafos). (Databricks, 2016)

El principal beneficio de Apache Spark es su aprovechamiento de computación distribuida para su funcionamiento. En la siguiente figura se grafica el concepto de un clúster con Apache Spark, y su distribución de tareas.

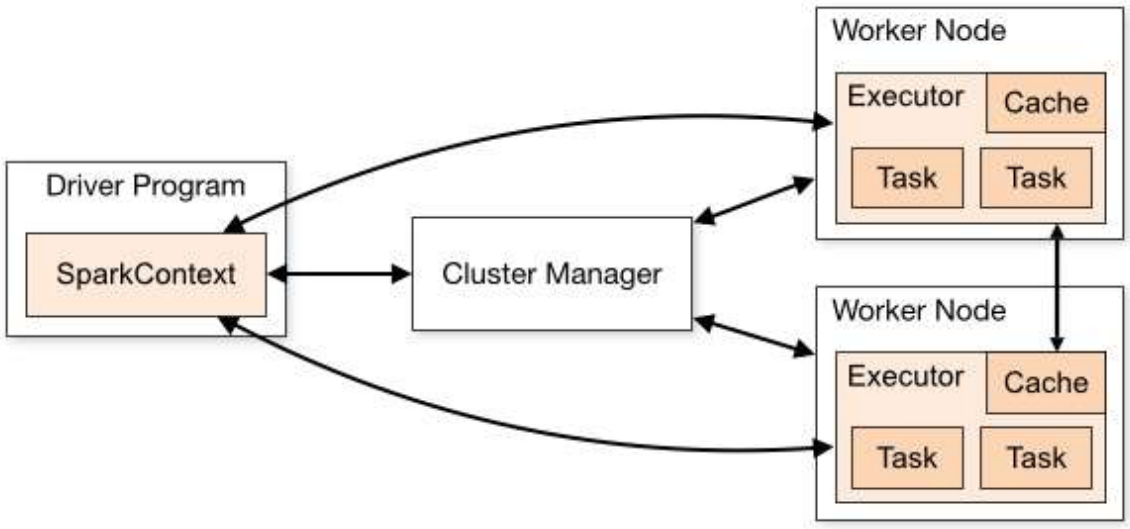

Figura 1. Componentes de Apache Spark.

A partir de la posibilidad de tener diferentes nodos en los que se puede distribuir el trabajo de procesamiento, es posible reducir el tiempo de ejecución paralelizando tareas, facilitando el uso generalizado de computación distribuida, usando las APIs proveídas por la herramienta y sin necesidad de implementar la lógica de distribución de procesos.

2. METODOLOGÍA

Para el desarrollo del proyecto se comenzó investigando sobre las alternativas existentes en algoritmos de agrupamiento y clusterización disponibles para las plataformas de Apache Spark y TensorFlow. Ambas de estas herramientas son de código abierto y ampliamente usadas en la comunidad científica y de software. Apache Spark es distribuido por el Apache Software Foundation, mientras que TensorFlow es mantenido por Google. Estas dos plataformas también poseen librerías relacionadas al aprendizaje de máquinas, y tienen implementaciones de diversos modelos de algoritmos de agrupamiento.

Se usó como precedente para la realización de este proyecto la tesis "Análisis de algoritmos para el agrupamiento de muestras metagenómicas" de Adriana María Escobar Vasco, egresada de Ingeniería de Sistemas y Computación de la Universidad EIA, cuyos algoritmos utilizados en su tesis son los que se buscan optimizar para mejorar su tiempo de ejecución.

2.1 MÉTODOS USADOS

2.1.1 K-means

El algoritmo de K-means es un método de agrupación no supervisado, es decir, no necesita saber la clasificación de cada uno de los puntos para entrenarse. Tiene como objetivo asignar diferentes puntos de datos a un número determinado de clústeres, llamado k, mediante aproximación de los centros usando las medias de los puntos asignados a cada centro durante cada iteración del algoritmo. Para el caso del proyecto los clústeres iniciales se tomaban aleatoriamente, mediante un modelo llamado *kmeans++*, en el que los centros se toman de una muestra aleatoria de los datos y se recalculan por cierto número de repeticiones hasta decidir los que están más alejados del resto.

Para este modelo de agrupamiento el valor de k debe ser sabido con anterioridad, por lo que a veces es compleja su utilización sobre datos en los que no se está seguro cuántos grupos pueden existir.

Las distancias usadas para este algoritmo durante el desarrollo de este trabajo tienen las siguientes fórmulas.

Distancia Euclidiana $dE(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$

Distancia Coseno $dC(x,y) = \frac{\sum_{i=1}^{n}(x_i \times y_i)}{\sqrt{\sum_{i=1}^{n} x_i} \times \sqrt{\sum_{i=1}^{n} x_i}}$

2.1.2 K-means Iterativo

El K-means iterativo es una variación del K-means, en el que un algoritmo completo de K-means se completa durante cada iteración, y cada iteración siguiente usa los resultados de la anterior. En cada una de las iteraciones se determinan clústeres "buenos", a partir de cierto margen, y los clústeres que no cumplen esta característica serán pasados a la siguiente iteración para que sus puntos se vuelvan a agrupar. (Escobar Vasco, 2015)

En cada repetición del algoritmo se pueden usar diferentes k, y es también posible usar diferentes algoritmos de distancia para evaluar las combinaciones que producen los mejores resultados.

Para el desarrollo del K-means Iterativo en el trabajo, se calculó el margen utilizando como medidas las distancias de los centroides de todos los clústeres, calculando el promedio de las distancias de un clúster con el resto, haciendo un promedio de este resultado para todos los clústeres, y sumándole la desviación estándar de este último. Se interpretó que los clústeres que tuvieran una distancia promedio al resto mayor que este margen, es porque estaban más lejanos, mientras que los que la tuvieran menor estaban todavía demasiado agrupados y podían volver a ser procesados por el algoritmo de agrupamiento.

2.2 MODELOS DE AGRUPAMIENTO EN APACHE SPARK

Apache Spark provee un paquete de *machine learning*, o aprendizaje de máquina, llamado *spark.mllib*. (Apache Software Foundation, 2018) Dentro de este paquete existen diversos modelos disponibles para hacer agrupamiento, entre ellos el K-means, que es el algoritmo usado en el desarrollo del trabajo precedente. También existen implementaciones de otros modelos como Gaussian mixture model, Power Iteration Clustering (PIC), y Latent Dirichlet allocation (LDA), entre otros.

Principalmente se identificó el modelo de K-means, compartido, ya que es también compartido con Tensorflow, la otra plataforma que se analizó para el desarrollo del trabajo, y es el usado en el trabajo previo

Aunque Spark posee APIs para diversos lenguajes de programación, como Scala, Java, y Python entre otros, se decidió usar Scala para el desarrollo del proyecto en lo referente a esta librería. Esto es debido a que Scala es el lenguaje nativo sobre el que está escrito Spark, y es el que posee la API más reciente y actualizada.

Una desventaja del uso de K-means en Spark es que sólo permite usar la distancia euclidiana al cuadrado, mientras que otras plataformas permiten el uso de una distancia basada en la similitud coseno.

Un ejemplo utilizando la implementación de este algoritmo de Apache Spark se presenta en la figura 2.

En este ejemplo primero se configura y se crea el contexto de Spark donde se ejecutará la aplicación. Este contexto es el que se encargará de encapsular la aplicación, y coordinar el clúster de computadores para operar sobre los datos y realizar las operaciones necesarias, para luego agrupar los resultados.

Como siguiente paso se lee un archivo de texto con datos de prueba y se guarda en un set de datos distribuido (*distributed data* set, o RDD). Este formato de datos propio de Spark es el que le permite dividir los datos para que cada nodo del clúster computacional pueda hacer operaciones independientes sobre su trozo asignado. Se instruye que estos datos sean almacenados en el caché para su acceso más rápido.

```scala
object KMeansExample {

  def main(args: Array[String]) {

    val conf = new SparkConf().setAppName("KMeansExample")
    val sc = new SparkContext(conf)

    // Load and parse the data
    val data = sc.textFile("data/mllib/kmeans_data.txt")
    val parsedData = data.map(s => Vectors.dense(s.split(' ')
      .map(_.toDouble))).cache()

    // Cluster the data into two classes using KMeans
    val numClusters = 2
    val numIterations = 20
    val clusters = KMeans.train(parsedData, numClusters, numIterations)

    // Evaluate clustering by computing Within Set Sum of Squared Errors
    val WSSSE = clusters.computeCost(parsedData)
    println(s"Within Set Sum of Squared Errors = $WSSSE")

    // Save and load model
    clusters.save(sc,
      "target/org/apache/spark/KMeansExample/KMeansModel")
    val sameModel = KMeansModel.load(sc,
      "target/org/apache/spark/KMeansExample/KMeansModel")

    sc.stop()
  }
}
```

Figura 2: Ejemplo utilizando K-means en Spark

Luego se crea y entrena un modelo de K-means con 2 clústeres y 20 iteraciones, usando los datos de prueba. Como último se calcula la suma de errores cuadrados internos para validar el modelo. Este modelo es guardado y se muestra como ejemplo cómo puede ser cargado nuevamente para su uso posterior.

2.3 MODELOS DE AGRUPAMIENTO EN TENSORFLOW

TensorFlow tiene como API nativa la basada en Python, y provee varias librerías para aprendizaje de máquina. Su módulo de *tf.contrib.factorization* tiene varios modelos y operaciones relacionadas a la factorización, o agrupamiento, de datos. Sus dos modelos principales son Gaussian Mixture Model y K-means. (Google, 2018)

Cada uno de estos modelos tiene dos implementaciones, usando dos APIs diferentes: una con *Graphs* (Grafos) y otra con *Estimators* (Estimadores). La diferencia principal en el uso de las implementaciones con cada una de estas APIs es que en la de grafos primero se debe crear un grafo de operaciones que serán ejecutadas en orden dentro de una sesión de TensorFlow, mientras que la API basada en estimadores usa principalmente *eager execution*, lo que lo acerca más a una programación imperativa en la que las operaciones

e instrucciones son ejecutadas inmediatamente, en lugar de crear un grafo previo a la ejecución.

Para este trabajo se utilizó la implementación de K-means basada en estimadores, usando el módulo *KMeansClustering*. Un ejemplo del uso de esta implementación en TensorFlow se presenta en la figura 3.

Este ejemplo está basado en el conjunto de datos de Iris, comúnmente usado en pruebas de aprendizaje de máquina. Este conjunto de datos se lee desde un archivo en formato CSV, y son convertidos a un tensor. La implementación de K-means en TensorFlow requiere una función que le provee los datos para sus operaciones, por lo que la conversión se hace dentro de una función que retorna el tensor de los puntos.

Luego se crea el modelo de K-means con 3 clústeres, y usando la distancia euclídea cuadrada por defecto. Este modelo luego se entrenará, usando la función de los datos, por 10 iteraciones, calculando el delta de los centros en cada iteración y el puntaje como validación.

Luego se llama al modelo para que prediga los clústeres asignados para el conjunto de datos, regresando una lista del índice del clúster al que pertenece cada uno de los puntos. Con este índice se imprime el punto, el índice, y el centro del clúster para todos los datos.

La diferencia principal entre el uso de TensorFlow sobre Apache Spark es el lenguaje de programación utilizado. Sin embargo, Spark también tiene una API disponible para Python, aunque no es tan mantenida como la de Scala.

```
import pandas as pd
import tensorflow as tf

points = pd.read_csv('iris.csv', comment='@', header=None,
                     names=['sepal_length', 'sepal_width',
'petal_length', 'petal_width', 'species'],
                     usecols=range(0, 4)).values


def input_fn():
    return tf.convert_to_tensor(points, dtype=tf.float32)


num_clusters = 3
kmeans = tf.contrib.factorization.KMeansClustering(
    num_clusters=num_clusters, use_mini_batch=False)

# train
num_iterations = 10
previous_centers = None
for _ in range(num_iterations):
    kmeans.train(input_fn)
    cluster_centers = kmeans.cluster_centers()
    if previous_centers is not None:
        print('delta:', cluster_centers - previous_centers)
    previous_centers = cluster_centers
    print('score:', kmeans.score(input_fn))
print('cluster centers:', cluster_centers)

# map the input points to their clusters
cluster_indices = list(kmeans.predict_cluster_index(input_fn))
for i, point in enumerate(points):
    cluster_index = cluster_indices[i]
    center = cluster_centers[cluster_index]
    print('point:', point, 'is in cluster', cluster_index, 'centered at',
center)
```

Figura 3: Ejemplo utilizando K-means en Tensorflow

2.4 DATOS

El conjunto de datos utilizado durante el proyecto está basado en una base proveída por el Centro Nacional de Secuenciación Genómica (CNSG). Cada registro de esta base representa una secuencia de ADN de una especie, extraída por medio de metagenómica balística, y conteniendo 872,576 segmentos en total para la base de datos.

Este conjunto fue procesado para calcular diferentes atributos, que serán usados como rasgos o características para el modelo de agrupamiento, y describen de una forma estándar las diferentes cadenas de aminoácidos. Los atributos que fueron calculados para esta nueva representación son el contenido de Guanina y Citosina, la cantidad de nucleótidos y codones de cada cadena, y los k-mer de longitud 3 y 4. Este último está

basado en los patrones existentes dentro de la cadena de ADN, incluyendo todas las combinaciones posibles de la longitud k para los cuatro aminoácidos.

Se usó estos datos para poder tener un punto de comparación con el trabajo realizado por Adriana Escobar en su tesis Análisis de algoritmos para el agrupamiento de muestras metagenómicas, que implementó los métodos de agrupamiento usando esta misma base. (Escobar Vasco, 2015)

2.5 GOOGLE CLOUD PLATFORM

Para el desarrollo del proyecto a nivel computacional, se usaron diversos servicios proveídos por Google Cloud Platform (GCP), una plataforma de computación en la nube. Se usó Compute Engine para realizar la ejecución de los algoritmos basados en Spark y en TensorFlow, utilizando una máquina de tipo n1-highmem-8, que cuenta con 8 vCPUs y 52 GB de memoria RAM.

Un gran beneficio de usar computación en la nube para el desarrollo es la facilidad con la que se pueden usar máquinas potentes de procesamiento masivo, utilizando recursos que es difícil llegar a conseguir de forma física, y que también aceleran el procesamiento de los datos del trabajo.

Para el almacenamiento y procesamiento de resultados se usaron Google Storage y Google BigQuery, con el primero almacenando los archivos de texto resultado de las ejecuciones, y BigQuery adaptando estos archivos de texto a un sistema lógico basado en tablas, permitiendo realizar un análisis rápido a partir de secuencias SQL.

2.6 VALIDACIÓN DE LOS RESULTADOS

Para el análisis de los resultados y su posterior validación, se utilizó el software Tableau Desktop, que permite conectarse con Google BigQuery para mostrar gráficamente y rápidamente un análisis sobre un volumen grande de datos, como lo son los resultados de la ejecución de los algoritmos.

Como medidas para hacer la validación de los clústeres se usaron la pureza y la dispersión. La pureza de un clúster representa la existencia de una única clase dentro de un clúster, y se calcula como el porcentaje de la clase mayoritaria que está contenida dentro de él.

La dispersión de esta medición para todos los contenidos del clúster, viéndose como una matriz de los porcentajes de todas las clases asignadas a él.

Estas medidas permiten hacer una validación visual y rápida sobre la efectividad de los algoritmos de agrupamiento en lo que se refiere a la separación de las diferentes especies. Se busca que cada especie quede contenida cada una dentro de un clúster únicamente, lo que representaría el resultado ideal, y se mostraría como una pureza del 100%, o 1, para todos los clústeres encontrados.

3. PRESENTACIÓN Y DISCUSIÓN DE RESULTADOS

Usando la implementación del K-means iterativo en Apache Spark y en TensorFlow, que se pueden apreciar en el anexo, se procedió a hacer varias ejecuciones del algoritmo con diferentes parámetros, buscando comparar los diferentes resultados que producía cada uno, y registrando los diferentes tiempos de ejecución. El fin de esto es comparar el resultado de ejecutar el mismo algoritmo con diferentes parámetros, usando como base los mismos parámetros de la tesis antecedente.

Como primer punto de comparación se ejecutó el algoritmo de K-means no iterativo con ambas distancias, usando como k 30. A partir de las observaciones se pudo apreciar que para el caso de nuestra base la distancia de coseno estaba produciendo los mejores resultados, pero al Apache Spark no tener esta distancia implementada en su librería, se procedió a enfocarse primeramente en TensorFlow como plataforma seleccionada.

La gran cantidad de datos obtenida en los resultados fue procesada usando Google BigQuery y Tableau, sobre las que se crearon tablas de dispersión para los diferentes clústeres formados por los algoritmos, así como también analizando la composición por especie de estos.

También se realizaron experimentos usando diferentes rasgos de los datos, primero usando todos los rasgos disponibles y luego usando exclusivamente k-mer de distancia 4, o 4mer. Esto gracias a que en el trabajo antecedente de Adriana Escobar se determinó que los mejores resultados se obtenían a partir del uso exclusivo de estos rasgos. (Escobar Vasco, 2015) Estos resultados fueron confirmados en los experimentos, se veía una diferencia considerable en la pureza de los clústeres a partir de las características tomadas por consideración durante la ejecución del algoritmo.

Los resultados de ejecutar el K-means no iterativo, con distancia coseno, usando 4mer y con k igual a 30 se pueden apreciar en la Figura 4.

En la gráfica se puede ver que la gran mayoría de los clústeres quedaron con una gran dispersión, mientras que los que quedaron con poca dispersión son de los de menor dimensión, como es el caso de los clústeres 0.10, 0.11, y 0.18.

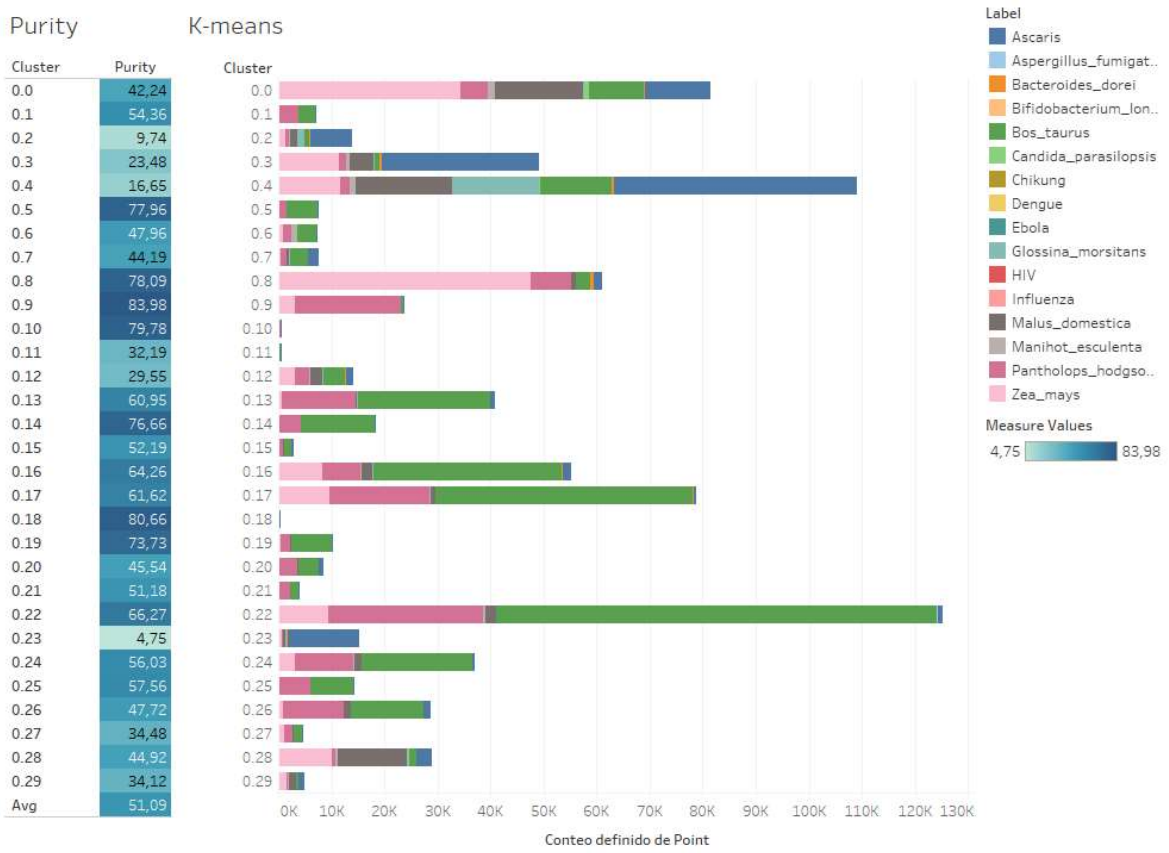

Figura 4: Clústeres creados con distancia coseno, k=30

En el lado izquierdo de la figura se puede apreciar el porcentaje de pureza para cada uno de los clústeres, con un porcentaje mayor siendo mejor para la medición de los clústeres.

Para el cluster 0.22, se puede apreciar una presencia mayoritaria de Bos taurus, igualmente que en los clústeres 0.16 y 0.17. Sin embargo, estos 3 clústeres presentan un perfil muy similar, con una porción de Influenza, de Pantholopos hodgsonii, y un pequeño porcentaje de Malus domestica. Esto puede llevar a la conclusión de que estos clústeres podrían ser agrupados y luego nuevamente separados usando K-means y diferentes parámetros, buscando que cada una de estas especies quede mejor discriminada. Estos clústeres poseen una pureza también muy similar, entre el 60% y 70% aproximadamente. Esto puede indicar que se podría hacer uso de la pureza para la identificación de clústeres de perfil similar, o que pueden llegar a ser subdivididos

Los clústeres 0.10 y 0.18, que tenían poca cantidad de puntos, son de los que tienen una mejor medición de la pureza. La pureza promedio de los 30 clústeres es 51.09%, y existen clústeres como el 0.2 y el 0.23 que tienen una pureza por debajo del 10% y el 5% respectivamente. Estas estadísticas muestran unos resultados de alta dispersión y baja pureza para una cantidad considerable de clústeres.

Para el siguiente experimento, se decidió realizar una segunda iteración sobre estos mismos resultados, para hacer una comparación de la variación al conservar los mejores clústeres de esta primera iteración, y corriendo el algoritmo de nuevo sobre los que estaban por debajo del margen. Como margen para determinar los mejores clústeres, se usó un cálculo con la distancia entre los centros de cada clúster, y comparando con el promedio de esta distancia, usando también la desviación estándar.

Por esto, se ejecutó el K-means iterativo, usando la misma distancia coseno y los 4mer como rasgos, pero esta vez con dos iteraciones, usando k igual a 30 para la primera y k igual a 20 para la segunda. Los resultados están graficados en la figura 5.

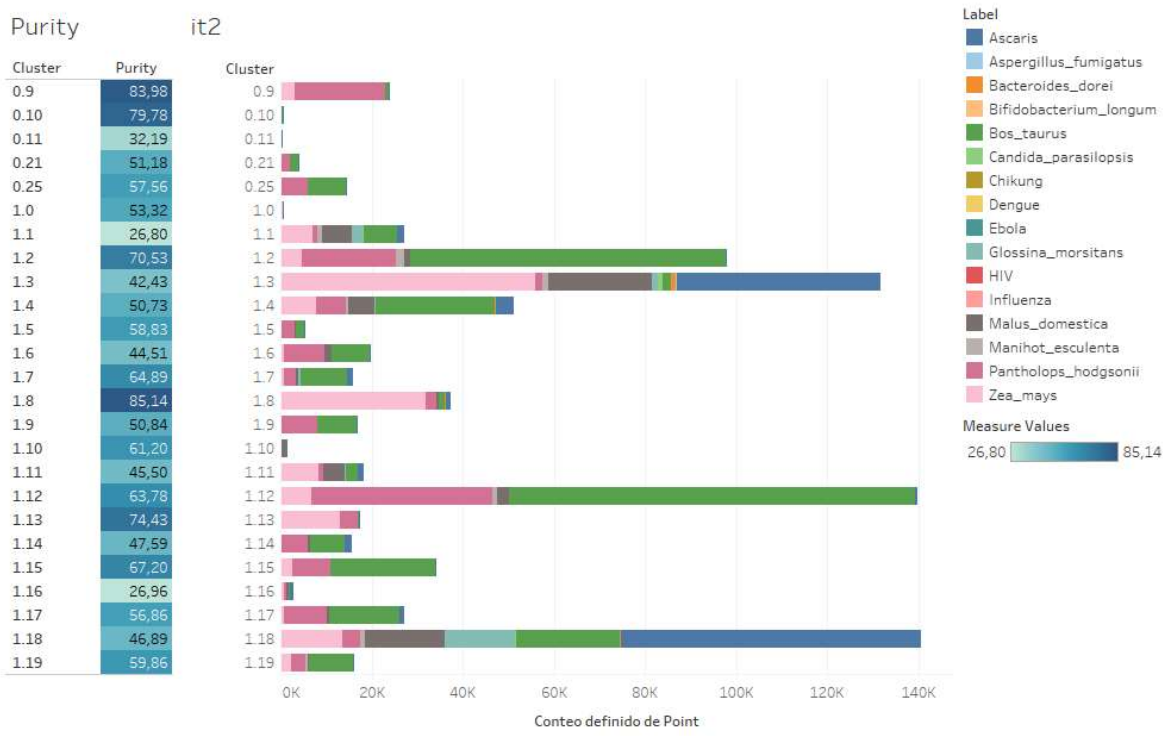

Figura 5: Clústeres creados con K-means iterativo, distancia coseno, y k=30,20

Al añadir una segunda iteración con k igual a 20, el total de los clústeres se redujo, debido a que de los 30 que fueron encontrados en la primera iteración, sólo 5 permanecieron, el resto fue reprocesado. Los clústeres conservados fueron el 0.9, 0.10, 0.11, 0.21 y 0.25. El clúster 0.9 en específico tiene una pureza bastante buena, con el 83.98% de su composición siendo parte de la especie *Pantholops hodgosonii*, mientras que el resto, aunque tienen una pureza entre 30% y 80%, son de dimensión mucho menor.

La especie *Ascaris*, que en la figura 4 es posible ver que está mucho más dispersa en los clústeres para el caso de K-means no iterativo con k=30, en estos resultados está casi completamente en los clústeres 1.3 y 1.18, mostrando una mejora en la separación de esta especie del resto.

Para el caso de esta especie, es posible que, a partir de la unión de estos dos clústeres y su reprocesamiento, quede mucho mejor separada del resto de las especies. Para este caso, se puede aplicar un algoritmo que determine también los clústeres que, a partir de cierto margen determinado, se podrían conjugar, llevando a una siguiente iteración del algoritmo con estos nuevos clústeres creados.

Mientras que para los clústeres de la iteración 1, la pureza mínima llegaba a ser 4.75%, con dos iteraciones la pureza mínima llega a ser 29.80%, mostrando una mejora considerable en la composición de los nuevos clústeres. El promedio de estas nuevas purezas es 56.12%, lo que muestra una mejora general comparado con la alternativa de una sola iteración.

Sin embargo, para ambos casos no se pudo llegar a una separación consistente de las especies a partir de los casos limitados de prueba. La tendencia que muestran los experimentos lleva a deducir que con más iteraciones, una variación en los parámetros, o incluso posiblemente filtrar los datos para reducir el ruido, se puede llegar a una composición más consistente de clústeres.

Buscando reducir el ruido existente en el conjunto de datos, se realizó un análisis de los tamaños de las cadenas de aminoácidos para las diferentes especies, mirando sus máximos, mínimos y promedios, y la cantidad de registros existentes por especie. Los resultados de este análisis están en la tabla 3.

Tamaños por Especie

Especie	Max(Tamaño)	Min(Tamaño)	Prom(Tamaño)	Cantidad
Ascaris	30,000	50	3,566	126,066
Aspergillus_fumigat..	29,660	1,001	3,117	295
Bacteroides_dorei	29,906	500	3,071	1,928
Bifidobacterium_lon..	26,797	540	6,506	18
Bos_taurus	5,000	101	1,472	315,841
Candida_parasilopsis	29,956	1,003	7,305	1,540
Chikung	11,826	11,826	11,826	1
Dengue	10,785	10,392	10,672	64
Ebola	18,957	18,957	18,957	1
Glossina_morsitans	29,996	101	5,636	20,333
HIV	9,181	9,181	9,181	1
Influenza	2,309	853	1,667	8
Malus_domestica	5,000	102	2,796	66,738
Manihot_esculenta	4,998	1,998	2,940	7,192
Pantholops_hodgso..	5,000	50	1,515	159,719
Zea_mays	5,000	102	1,564	161,168

Max(Tamaño), Min(Tamaño), Prom(Tamaño) and Cantidad broken down by Especie.

Tabla 1: Relación de estadísticas del tamaño del contig por especie

Es posible ver que existe una gran variación en los tamaños para las diferentes especies, con algunas teniendo mínimos de 50 y máximos de 5000, y otras teniendo mínimos de 1000 y máximos de 30000 aproximadamente. Con el fin de analizar si filtrando las especies con tamaños más pequeños de la muestra de datos mejora la pureza de los clústeres, se tomó

el promedio de las purezas para los resultados del K-means iterativo, con distancia coseno, dos iteraciones de k=30,20, y se hizo un gráfico de comparación mostrado en la figura 6.

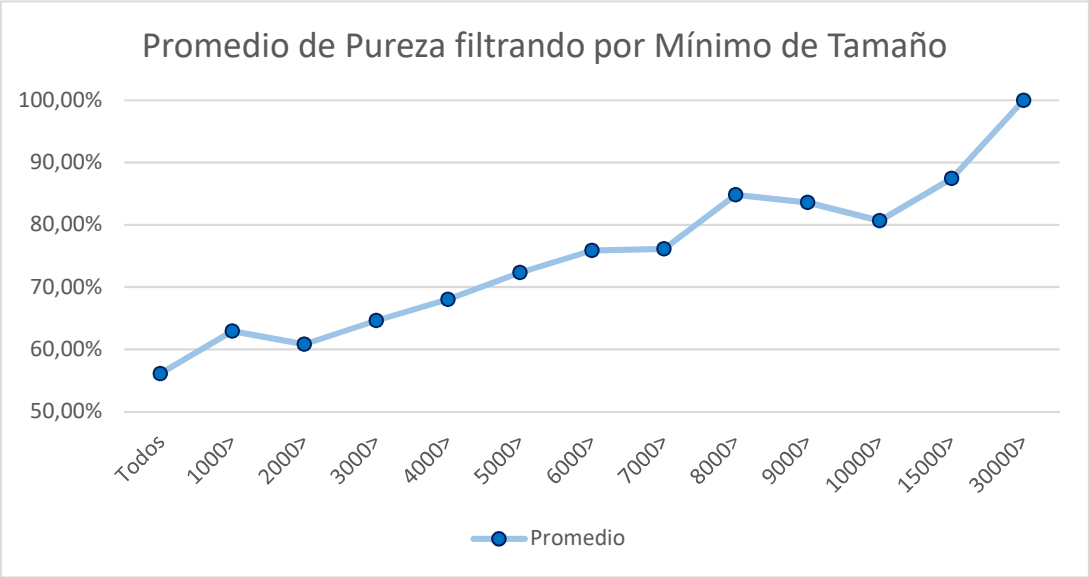

Figura 6: Evolución de la pureza a partir de filtrar por un mínimo de tamaño

Con el incremento del filtro del mínimo de tamaño, se puede apreciar un incremento también de la pureza de los clústeres, reduciendo la cantidad de datos tomados en cuenta. Para el caso de un tamaño mayor a 30000, se tiene sólo 1 registro de una especie, perteneciendo a un solo clúster, por lo que la pureza es del 100%. Esto no entrega información adicional, por lo que se debe buscar un límite de tamaño que siga teniendo suficiente cantidad de datos para proporcionar resultados significativos, mientras se reduce la cantidad de ruido existente.

Como demostración de los posibles resultados a partir del uso del tamaño como filtro, se analizan los resultados del algoritmo de la Figura 5, pero filtrando para las muestras en las que su cadena de aminoácidos tiene un tamaño mínimo de 5000, ya que en la Figura 6 se puede apreciar que es el límite en el que el promedio de la pureza supera el 70%.

El gráfico de los resultados se aprecia en la figura 7.

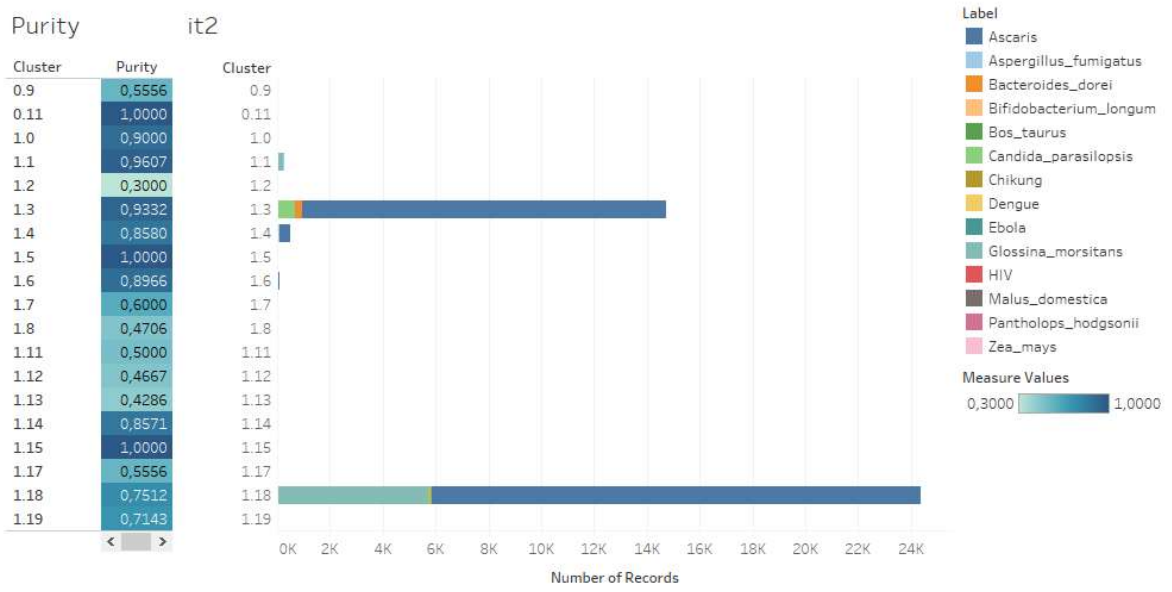

Figura 7. Clústeres creados con K-means iterativo, distancia coseno, y k=30,20 y tamaño mayor a 5000

Se puede apreciar que muchas de las especies, o fueron eliminadas completamente del análisis, o fueron reducidas a unos pocos especímenes. El mayor porcentaje de especies restante pertenece a Ascaris, seguida de Aspergillius fumigatus, y un mucho menor porcentaje se encuentran Candida parasilopsis y Bacteroides dorei.

Esto lleva a la conclusión de que, aunque la reducción del ruido mediante el filtro por tamaño de las cadenas de aminoácidos lleva también a que la base de datos se reduzca demasiado, llevando a clústeres con muy pocas muestras, y también sesgando los datos hacia diferentes especies.

Para medir los tiempos de ejecución de los algoritmos, se usó una máquina de alta memoria (52 GB), y se corrieron varias ejecuciones de estos al tiempo sobre esta. En el caso del K-means de una sola iteración en trabajos antecedentes, usando una implementación con Weka y también ejecutándolo en una máquina de alta memoria, el mínimo tiempo de ejecución era aproximadamente 16 minutos. Para la alternativa desarrollada con TensorFlow, se produjo un tiempo aproximado de 13 minutos. Para este algoritmo específicamente la diferencia no es muy considerable, ya que su ejecución era bastante corta desde un principio.

En el caso del K-means Iterativo, los tiempos de ejecución pueden fluctuar debido a todas las variables relacionadas, como el número de iteraciones, los algoritmos de distancia usados, y los rasgos de los datos que fueron tomados en cuenta. Sin embargo, en trabajos antecedentes usando la implementación de Weka, el mínimo tiempo de ejecución para los casos de dos iteraciones era aproximadamente 1 hora, y para casos más complejos podía llegar hasta ser de 4 y 8 horas. Para la alternativa desarrollada con TensorFlow el mínimo tiempo es de 25 minutos aproximadamente y para casos más complejos llegó a ser de 1 hora. Esto es una mejora considerable, teniendo en cuenta que dependiendo de los

experimentos en busca de mejores resultados se puede llegar a ejecutar el algoritmo varias veces por día, ahorrando horas en cada iteración.

4. CONCLUSIONES Y CONSIDERACIONES FINALES

Al finalizar la realización de este proyecto se confirmó que el uso de tecnologías distribuidas y del estado del arte, como TensorFlow, puede disminuir los tiempos de ejecución de algoritmos de agrupamiento iterativos, como es el caso de K-means iterativo. Apache Spark por su parte también mejora los tiempos, pero debido a que su API es más restringida y no posee las implementaciones de las distancias y algoritmos que proveen los mejores resultados, limita bastante su uso.

Una alternativa existente relacionada es aprovechar las capacidades de computación distribuida de Apache Spark, mientras se usa la API proveída por TensorFlow para la implementación de los algoritmos de agrupamiento deseados. Apache Spark también posee soporte para ejecutar Python, por lo que la ejecución de TensorFlow sobre Apache Spark es posible.

TensorFlow también permite el uso de computación distribuida, pero no es tan transparente ni fácil de usar como Apache Spark. Requiere una programación más a bajo nivel de la coordinación paralela de ejecución de los problemas, mientras que Spark tiene una API más a alto nivel.

En los resultados se puede confirmar la mejora de calidad de clústeres por el uso de K-means Iterativo comparado con el K-means de una sola ejecución, y reafirma que la función coseno usando los 4mer como rasgos es la que hasta ahora ha producido los mejores clústeres con respecto a su pureza y dispersión.

A destacar como producto del trabajo realizado es la disminución del tiempo de ejecución del K-means Iterativo para la implementación en trabajos antecedentes, teniendo reducciones de hasta 7 horas en ejecuciones más complejas.

REFERENCIAS

Apache. (12 de Abril de 2016). *What Is Apache Hadoop?* Obtenido de Apache™ Hadoop®: https://hadoop.apache.org/

Apache Software Foundation. (01 de 05 de 2018). *MLlib Clustering*. Obtenido de Apache Spark Docs: https://spark.apache.org/docs/2.3.0/mllib-clustering.html

Barney, B. (12 de Abril de 2016). *Introduction to Parallel Computing*. Obtenido de Lawrence Livermore National Laboratory: https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

Bonet, I., Montoya, W., Mesa Múnera, A., & Alzate, J. (2014). Iterative Clustering Method for Metagenomic Sequences.

Databricks. (29 de 04 de 2016). *About Spark*. Obtenido de Databricks: https://databricks.com/spark/about

Dröge, J., Gregor, I., & McHardy, A. C. (2014). Taxator-tk: precise taxonomic assignment of metagenomes by fast approximation of evolutionary neighborhoods. *Bioinformatics*.

Escobar Vasco, A. M. (2015). Análisis de algoritmos para el agrupamiento de muestras metagenómicas.

Google. (01 de 05 de 2018). *Module: tf.contrib.factorization* . Obtenido de Tensorflow Python API Docs: https://www.tensorflow.org/api_docs/python/tf/contrib/factorization

Handelsman, J. (2004). Metagenomics: Application of Genomics to Uncultured Microorganisms. *Microbiology and Molecular Biology Reviews*.

Hortonworks. (s.f.). *Apache Hadoop YARN*. Recuperado el 12 de Junio de 2016, de Hortonworks: http://hortonworks.com/apache/yarn/

IBM. (15 de 05 de 2018). *What is distributed computing*. Obtenido de IBM Knowledge Center: https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.2.0/com.ibm.cics.tx.doc/concepts/c_wht_is_distd_comptg.html

Koslicki, D., Foucart, S., & Rosen, G. (2013). Quikr: a method for rapid reconstruction of bacterial communities via compressive sensing. *Bioinformatics*.

Locey, K. J., & Lennon, J. T. (2016). Scaling laws predict global microbial diversity. *National Academy of Sciences*. Obtenido de http://www.pnas.org/content/early/2016/04/26/1521291113

Maheedhar Reddy, R., Haque Mohammed, M., & S. Mande, S. (2014). MetaCAA: A clustering-aided methodology for eficient assembly of metagenomic datasets. *Genomics*.

Montoya Ramírez, W. S. (2014). Exploración y comparación de métodos de inteligencia artificial para la clasificación taxonómica en análisis metagenómicos.

O'Driscoll, A., Daugelaite, J., & Sleator, R. D. (2013). 'Big data', Hadoop and cloud computing in genomics. *Journal of Biomedical Informatics*. doi:https://doi.org/10.1016/j.jbi.2013.07.001

Tanaseichuk, O., Borneman, J., & Jiang, T. (2012). Separating metagenomic short reads into genomes via clustering. *Algorithms for Molecular Biology*.

Thomas, T., Gilbert, J., & Meyer, F. (2012). Metagenomics - a guide from sampling to data analysis. *Microbial Informatics and Experimentation*.

Wang, Y., Ming Leung, H. C., Ming, S., & Yuk Lun, F. (2014). MetaCluster-TA: taxonomic annotation for metagenomic data based on assembly-assisted binning. *BMG Genomics*.

Wooley, J. C., Godzik, A., & Friedberg, I. (2010). A Primer on Metagenomics. *PLoS Comput Biol 6(2): e10006672010*. Obtenido de https://doi.org/10.1371/journal.pcbi.1000667

Zhou, W., Li, R., Liu, C., Yao, S., Luo, J., & Niu, B. (2017). MetaSpark: a spark-based distributed processing tool to recruit metagenomic reads to reference genomes. *Bioinformatics*.

ANEXO 1

```
object IterativeKMeans {
  def run(config: Config) {
    val conf = new SparkConf().setAppName("KMeansMetagenomics")
      .setIfMissing("spark.master", "local[2]")
    val log = LogManager.getLogger(IterativeKMeans.getClass)
    log.setLevel(Level.DEBUG)
    log.debug(s"Started iterative program")

    val spark = SparkSession.builder().config(conf).getOrCreate()

    val data = spark.sparkContext.textFile(config.input)
    var parsedData = data.filter(s => !s.startsWith("@")).map(s => PointWithCategory(s, config.classes))

    var centers: ListBuffer[(String, Vector)] = ListBuffer[(String, Vector)]()

    val numClusters = config.numClusters
    val numIterations = config.numIterations

    parsedData.cache()
    var result = spark.sparkContext.emptyRDD[String]
    for (runIndex <- 1 to config.runs) {
      val model = new KMeans().setK(numClusters)
        .setMaxIterations(numIterations)
        .setEpsilon(1e-4)
        .setInitializationMode("kmeans||")
        .run(parsedData.map( . 1))
      val runCenters = model.clusterCenters
      val score = model.computeCost(parsedData.map(_._1))
      log.debug(s"score: $score")
      log.debug(s"centers: ${runCenters.toString}")
      if (runIndex < config.runs) {
        val distMatrix = ArrayBuffer.fill(centers.length, centers.length)(0.0)
        for ((centerA, i) <- runCenters.zipWithIndex) {
          for ((centerB, j) <- runCenters.zipWithIndex) {
            val dist = Vectors.sqdist(centerA, centerB)
            distMatrix(i)(j) = dist
          }
        }

        val distAvg = distMatrix.map(row => row.sum/row.length)
        val average = distAvg.sum/distAvg.length
        val std = Math.sqrt(distAvg.map(distance => Math.pow(Math.abs(distance - average), 2)).sum /
distAvg.length)
        val margin = average + std

        val clustKeep = ListBuffer[Int]()
        for ((row, index) <- distAvg.zipWithIndex) {
          if (row >= margin) {
            clustKeep.append(index)
            centers.append((runIndex.toString + "." + index.toString, runCenters(index)))
          }
        }

        val output = parsedData.map{case (point, label) => (point, label, model.predict(point))}
        parsedData = output.filter{case ( ,  , cluster) => clustKeep.contains(cluster)}.map{case (point, label,
_) =>
          (point, label)}
        result = result ++ output.filter{case (_, _, cluster) => !clustKeep.contains(cluster)}.map{
          case (point, label, cluster) =>
            (point.toString.replace(",", " "), label, runIndex.toString + "." + cluster)
              .productIterator.mkString(",")
        }
      } else {
        result = result ++ parsedData.map{case (point, label) =>
          (point.toString.replace(",", " "), label, runIndex.toString + "." + model.predict(point))
            .productIterator.mkString(",")}
      }
    }
    println(result.count())
    result.coalesce(1).saveAsTextFile(config.output)
    val file = new File(config.output + "/centers.txt")
    val bw = new BufferedWriter(new FileWriter(file))
    bw.write(centers.map{case (index, center) =>
      (index, center.toArray.map(_.formatted("%.3f")).mkString("[", " ", "]")).productIterator.mkString(",")}
      .mkString(scala.util.Properties.lineSeparator))
    bw.close()
    spark.stop()
  }
}
```

ANEXO 2

```
def input_fn(data):
    return tf.data.Dataset.from_tensors(tf.convert_to_tensor(data, dtype=tf.float32))


num_runs = args.runs
with open('output/{0}_output.txt'.format(args.output), 'w') as pt_f, \
        open('output/{0}_cluster.txt'.format(args.output), 'w') as ct_f:
    print('index', 'point', 'label', 'cluster', sep=',', file=pt_f)
    print('cluster', 'center', sep=',', file=ct_f)
    for run_ix in range(num_runs):
        num_clusters = k_list[run_ix]
        distance = distance_list[run_ix]
        if distance == 'euclidean':
            dist_kmeans = KMeansClustering.SQUARED_EUCLIDEAN_DISTANCE
            dist_scipy = spatial.distance.sqeuclidean
        else:
            dist_kmeans = KMeansClustering.COSINE_DISTANCE
            dist_scipy = spatial.distance.cosine
        kmeans = KMeansClustering(
            num_clusters=num_clusters, use_mini_batch=False,
            initial clusters=KMeansClustering.KMEANS PLUS PLUS INIT,
            distance_metric=dist_kmeans)
        print('Created KMeans using distance', dist_kmeans)

        # train
        num iterations = 100
        kmeans.train(input_fn=lambda: input_fn(points),
                     max_steps=num_iterations)
        cluster_centers = kmeans.cluster_centers()
        print(datetime.now(), 'score:', kmeans.score(input_fn=lambda: input_fn(points)))
        print(datetime.now(), 'cluster centers:', cluster_centers)

        if run_ix < num_runs - 1:
            # calculate margin
            dist_matrix = np.zeros((len(cluster_centers), len(cluster_centers)))
            for i, center_a in enumerate(cluster_centers):
                for j, center b in enumerate(cluster centers):
                    dist = dist_scipy(center_a, center_b)
                    dist matrix[i][j] = dist

            dist_avg = np.average(dist_matrix, axis=1)
            margin = np.average(dist_avg) + np.std(dist_avg)

            clust_keep = list()
            for i, row in enumerate(dist_avg):
                if row.item() >= margin:
                    clust_keep.append(i)
                    print(str(run_ix) + '.' + str(i),
                          np.array2string(cluster_centers[i], formatter={'float_kind': lambda x: "%.3f" % x},
                                          max_line_width=np.inf),
                          sep=',', file=ct_f)

            # map the input points to their clusters and calculate average intracluster distance
            cluster_indices = list(kmeans.predict_cluster_index(lambda: input_fn(points)))

            new_points = list()
            new_labels = list()
            new_indices = list()
            for point_ix, point in enumerate(points):
                cluster = cluster_indices[point_ix]
                if cluster in clust_keep:
                    print(indices[point_ix],
                          np.array2string(point, formatter={'float_kind': lambda x: "%.3f" % x},
max_line_width=np.inf),
                          labels[point_ix].tolist()[0], str(run_ix) + '.' + str(cluster), sep=',', file=pt_f)
                else:
                    new_points.append(point)
                    new_labels.append(labels[point_ix])
                    new_indices.append(indices[point_ix])

            points = np.asarray(new_points)
            labels = np.asarray(new_labels)
            indices = np.asarray(new_indices)
        else:
            print(datetime.now(), 'Writing final clusters and points')
            cluster_indices = list(kmeans.predict_cluster_index(lambda: input_fn(points)))
            for i, point in enumerate(points):
                cluster_index = cluster_indices[i]
                label = labels[i]
                index = indices[i]
                print(index,
```