UCLV
Universidad Central
"Marta Abreu" de Las Villas

MFC
Facultad de Matemática
Física y Computación

TRABAJO DE DIPLOMA

Título: Implementación de algoritmos de cálculo de similitud de proteínas utilizando Apache Spark

Autor: Andrés Puerto Pacheco

Tutores: Dra. Deborah Galpert Cañizares, Dr. Reinaldo Molina Ruiz.

Santa Clara

UCLV
Universidad Central
"Marta Abreu" de Las Villas

MFC
Facultad de Matemática
Física y Computación

Academic Department of Computing

Career in Computer Science

DIPLOMA THESIS

Title: Implementation of protein similarity calculation algorithms using Apache Spark

Author: Andrés Puerto Pacheco

Thesis Director: Dra. Deborah Galpert Cañizares, Dr. Reinaldo Molina Ruiz.

Santa Clara, June 2019

RESUMEN

La comparación de proteínas dentro del análisis de secuencias de proteínas o enzimas resulta un área de investigación activa en bioinformática ya que la cantidad de secuencias aumenta considerablemente a ritmo acelerado por el perfeccionamiento de las técnicas de secuenciación de genomas. El reconocimiento de la función o la clasificación estructural de las proteínas o enzimas continúa siendo un reto por la divergencia en las secuencias que hace que los métodos de comparación basados en alineamiento fallen cuando se comparan secuencias homólogas con baja identidad. Es por esto que la combinación de medidas basadas en alineamiento y libres de este puede ser útil en las distintas aplicaciones. La implementación PySpark de cálculo de medidas de similitud entre proteínas como parte de un sistema de big data analítica que se desarrolla en este trabajo logra este fin y además mejora la versión anterior implementada en la UCLV al permitir un aumento en el rango de los parámetros de estas medidas. La nueva implementación también reduce la carga computacional en el cálculo de la frecuencia de subsecuencias de longitud *k* en las secuencias. Las pruebas de software fueron satisfactorias en el clúster de Spark de la UCLV.

Palabras Clave: Medidas de Similitud, Alineamiento de proteínas, Descriptores de proteínas libres de alineamiento, Spark, PySpark

Abstract

Protein comparison is an active bioinformatics research topic in sequence analysis of proteins or enzymes since the amount of sequences increases in a rapid rate because of the improvements achieved in genome sequencing techniques. The recognition of the function or the structural classification of proteins or enzymes continues to be a challenge due to the divergence of sequences causing that the alignment-based comparison methods fail when they compare homolog sequences with low identity percent. For this reason the combination of alignment-based and alignment-free similarity measures may be useful in different applications. The PySpark implementation of the protein similarity measures as part of a big data analytics system developed in this theses paper is pursuing this goal and also improves the previous version of the calculations implemented in the UCLV when it allows an increased range of the parameter values. Besides, the new implementation reduce the computational load in the calculation of the frequency of the subsequences of length *k*. The software testing was acceptable in the Spark cluster of the UCLV.

Keywords: Similarity measures, Protein alignment, Alignment-free protein descriptors, Spark, PySpark

TABLA DE CONTENIDOS

LISTA DE FIGURAS

LISTA DE TABLAS

INTRODUCCIÓN

La comparación de proteínas con diversos fines ha sido objetivo de múltiples herramientas bioinformáticas basadas en alineamiento de secuencias y otras libres de alineamiento que representan las proteínas como vectores numéricos (Zielezinski et al., 2017). Entre las aplicaciones de estas herramientas se encuentra el análisis evolutivo, la clasificación en familias y subfamilias, así como la clasificación estructural de proteínas, todas aplicaciones de gran importancia por su posterior uso en la industria biotecnológica y farmacéutica.

En la consulta de la bibliografía se aprecia una tendencia a la mezcla de información de proteínas basada en alineamiento y libre de alineamiento para lograr predicciones más eficaces (Chen et al., 2016). Precisamente, en los trabajos realizados en el Centro de Investigaciones de Informática de la Universidad Central "Marta Abreu" de Las Villas (UCLV) como (Galpert, 2018) se propone la mezcla de rasgos de proteínas para la clasificación evolutiva de genes. Se incorpora a su vez el uso de big data a la comparación de proteínas. Seguidamente en (Arteaga, 2018), se propone como parte de un sistema de Big Data Analítica, el cálculo de descriptores de proteínas libres de alineamiento en el modelo de programación Spark con vistas a aplicarlo a la comparación por pares de grandes conjuntos de datos de proteínas incluyendo proteomas completos. Sin embargo, la primera implementación de estos cálculos debe ser mejorada desde los puntos de vista siguientes: (i) cálculo de descriptores libres de alineamiento de proteínas con una ampliación en el rango de valores de los parámetros, (ii) inclusión de la comparación por pares de proteínas utilizando diversas medidas de similitud libres de alineamiento y (iii) inclusión de la similitud basada en alineamiento.

La primera versión de los cálculos Spark de descriptores de proteínas ha sido concebida para ejecutar en el clúster de computadoras de la Universidad Central "Marta Abreu" de las Villas por lo que las mejoras a dicha versión deben ser desplegadas y probadas en la misma infraestructura de manejo de big data. De la problemática planteada se deriva el siguiente problema de investigación:

Problema de investigación

La anterior implementación de los cálculos de descriptores de proteínas debe ser renovada y probada para realizar la comparación par a par en grandes conjuntos de datos de proteínas.

Objetivo general

Desarrollar una nueva versión de los métodos de cálculo de medidas de similitud basadas en alineamiento y libres de alineamiento en pares de proteínas para grandes conjuntos de datos utilizando la plataforma Apache Spark.

Objetivos específicos

1. Reconocer las limitaciones de la primera implementación de los cálculos de descriptores de proteínas.
2. Desarrollar renovaciones al cálculo de descriptores de proteínas en cuanto al aumento en el rango de valores de los parámetros.
3. Desarrollar el cálculo de similitud por pares de proteínas basados en medidas de alineamiento y libres de alineamiento.
4. Validar los algoritmos en el clúster de Spark de la UCLV.

Tareas de investigación

1. Estudio de aspectos teóricos necesarios para la implementación de las medidas de similitud basadas en alineamiento y libres de alineamiento con el uso de big data.
2. El estudio de las limitaciones de la primera versión de los cálculos de descriptores de proteínas.
3. La selección de tecnologías de Spark necesarias para realizar las mejoras al código anterior.
4. El diseño e implementación de la nueva versión.
5. La validación de la implementación en el clúster de la UCLV.

Justificación y Viabilidad de la investigación:

El presente trabajo forma parte del proyecto de investigación: “Herramientas bioinformáticas para la búsqueda de secuencias codificadoras de enzimas dextranasacarasas”, Proyecto

Empresarial de UCLV con el código 10654, de la línea científica Ciencia e ingeniería de la computación, cuyo líder es el Profesor Titular, Dr.C. Rafael E. Bello Pérez. Los resultados que se obtengan en este trabajo de diploma deben apoyar los objetivos específicos del proyecto relacionados con:

- Desarrollar herramientas que integren varios descriptores de proteínas para la detección de secuencias de enzimas tipo GH70, y que sean escalables al análisis amplio de diversos proteomas. Estas enzimas son altamente perjudiciales en la producción de azúcar ocasionando pérdidas millonarias y es necesario elevar la precisión de los algoritmos de detección de las mismas mediante búsqueda por similitud para que puedan encontrar secuencias incluso divergentes en proteomas poco explorados. Las nuevas secuencias que se encuentre deben contribuir a la construcción de modelos inhibidores más efectivos. Además, dichas enzimas tienen diversos usos en la industria biotecnológica.

- Desarrollar algoritmos de detección relaciones evolutivas entre proteínas como los algoritmos de detección de ortólogos capaces de manejar grandes volúmenes de secuencias que permitan la detección funcional de miembros ortólogos de la familia GH70.

Valor práctico

Las herramientas bioinformáticas que se desarrollen en este trabajo deben ser de utilidad a la comunidad bioinformática de nuestro país y de fuera de este. Representarían un paso de avance en el uso del clúster de la UCLV en función de problemas reales de alta aplicabilidad en la industria biotecnológica.

Estructura del Documento

El trabajo consta de tres capítulos:

El **Capítulo I** trata los aspectos generales sobre la comparación de proteínas en el ámbito de big data analítica.

El **Capítulo II** describe el proceso de diseño e implementación de la aplicación a través de diagramas y algoritmos empleados para el desarrollo de la misma.

El **Capítulo III** describe las pruebas de software para validar la funcionalidad de la aplicación.

CAPÍTULO 1. Comparación de proteínas en el ámbito de big data analítica

En este capítulo se abordan aspectos teóricos relacionados con el cálculo de las medidas de similitud de proteínas basadas en alineamiento y libres de alineamiento para comparar pares de proteínas. Seguidamente se presentan las características de avances y limitaciones en los sistemas Big Data Analítica para el análisis de proteínas y en particular las limitaciones en cuanto a facilidades de programación empleadas para su desarrollo de la implementación Spark realizada en la UCLV. Finalmente, se especifican las características de la plataforma instalada, las bibliotecas y otros requerimientos de implementación para el trabajo en el clúster de Spark de la UCLV.

1.1 Comparación par a par de proteínas

El alineamiento de dos secuencias de proteínas ha sido ampliamente utilizado en la comparación par a par de proteínas para la detección de proteínas homólogas, es decir, proteínas que tienen un origen evolutivo común (Galpert, 2016). Compara cada aminoácido en una posición de una secuencia con el aminoácido correspondiente en la otra, de esta forma penaliza la aparición de aminoácidos diferentes ubicando los llamados "huecos" o gaps en las respectivas secuencias alineadas. Algunos algoritmos de alineamiento basados en la programación dinámica, como el algoritmo de alineamiento global (NW) (Needleman and Wunsch, 1970) y el algoritmo de alineamiento local (SW) (Smith and Waterman, 1981), devuelven la puntuación óptima (score) del alineamiento que se calcula sumando el valor de la llamada matriz de sustitución para cada par de correspondencias en el alineamiento obtenido, y restando el valor de penalización de gaps en los casos que lo requieran (Mount, 2004a). En el modelo de penalización extendida (Smith and Waterman, 1981), el algoritmo de alineamiento asigna una penalización a la apertura de un gap (GOP), y una, a los gaps sucesivos después de una apertura (GEP).

Entre las matrices de sustitución más utilizadas se encuentra la familia PAM (Dayhoff, 1978), que contienen las probabilidades de cambiar un aminoácido por otro en secuencias de proteínas homólogas durante el proceso evolutivo. Son muy usadas en alineamientos de secuencias distantes en la evolución cuya identidad se encuentra aproximadamente en un 20% (Pearson, 2013). Otras son las matrices BLOSUM (Henikoff, 1992) basadas en la observación de sustituciones de aminoácidos en grandes regiones conservadas en bases de datos de proteínas. De este modo, el modelo PAM está diseñado para determinar el origen evolutivo de las secuencias de proteínas, mientas que el BLOSUM, para encontrar los dominios conservados entre las proteínas. Algunas combinaciones de parámetros de alineamiento son recomendadas en (Pearson, 2013) para valores de penalización y matrices de sustitución de acuerdo a los porcientos de identidad de secuencias. De estas recomendaciones y de (Eddy, 2004) se selecciona BLOSUM62 en este trabajo por haber sido tomada por defecto en muchas herramientas de alineamiento considerándose las regiones con un 62% de identidad o menos. De igual forma se toman valores recomendados de penalización para apertura y continuidad de gaps, respectivamente.

Por otra parte, los métodos de alineamiento basados en palabras o *k*-tuplas como FASTA (Pearson, 1990) y BLAST (Altschul et al., 1990) son heurísticas que no garantizan encontrar la solución óptima de un alineamiento, pero son significativamente más eficientes que los basados en programación dinámica (Mount, 2004b). En particular, BLASTp se utiliza para encontrar regiones de similitud local entre proteínas. Estos métodos realizan el alineamiento de dos secuencias rápidamente, buscando a través de cortas prolongaciones idénticas, llamadas palabras o *k*-tuplas (*k*=3 por defecto en BLASTp), y luego uniendo estas palabras dentro de un alineamiento producido por un método de programación dinámica. Determinan si un alineamiento es significativo o no, para disminuir la cantidad de comparaciones a realizar. En el caso del BLAST, la significación del alineamiento E-value representa el número de alineamientos diferentes con puntuaciones equivalentes a, o mejores que, una puntuación dada que se espera se produzcan por azar en una búsqueda en una base de datos de secuencias. Aunque los algoritmos heurísticos son más eficientes, principalmente, en la búsqueda secuencias en bases de datos (Altschul et al., 1990), en este trabajo se propone el uso de los

algoritmos de programación dinámica por la exactitud de su solución al realizar comparaciones por pares de secuencias.

El alineamiento global entre proteínas puede brindar información sobre la similitud estructural, mientras que el local, puede brindarla sobre la similitud funcional. Los algoritmos de alineamiento global calculan el alineamiento sobre la longitud total de las secuencias comparadas determinando las identidades o similitudes entre los aminoácidos. Las secuencias que son altamente similares, con longitud análoga, son las adecuadas para ser comparadas mediante un algoritmo de alineamiento global. Sin embargo, los alineamientos locales son más útiles para secuencias en las que se conoce que existen regiones muy similares, secuencias que difieren en sus longitudes, o secuencias que comparten una región conservada o dominio (Christianini and Hahn, 2006).

Otra medida basada en el alineamiento global ha sido propuesta en (Galpert, 2016) denominada perfil físico-químico que permite analizar la similitud de proteínas midiendo la correlación entre las medias móviles de las energías de contacto de los aminoácidos en las regiones alineadas sin gaps. Esta medida resultó promisoria en experimentos de clasificación evolutiva.

A partir del alineamiento global se calcula el porcentaje de identidad que indica las coincidencias de aminoácidos con relación a la longitud del alineamiento obtenido. Este porcentaje define la zona twilight (<30% identidad de secuencias) en la cual se presentan fallas en la detección de la homología (Chapin, 2013). Las secuencias con una identidad menor del 40% presentan dificultad de igual forma para la detección de relaciones estructurales como se especifica en (Vinga et al., 2004, Zielezinski et al., 2017). Es por esto que en estas referencias se presenta el uso de otras medidas de similitud llamadas libres de alineamiento para comparar proteínas.

Como se plantea en (Davies et al., 2008), (Iqbal et al., 2014) las técnicas de clasificación basadas en alineamiento están afectadas en su funcionamiento ante secuencias que tienen muy baja similitud. Asumen que se preserva la contigüidad en segmentos homólogos, pero esto no debe ser cierto ante la ocurrencia de eventos genéticos. Por estas razones, surgen dichos métodos libres de alineamiento para la comparación de secuencias. Uno de los métodos libres

de alineamiento es el perfil de composición de aminoácidos (AAC) (Bhasin and Raghava, 2004), que describe la proporción de cada aminoácido en una proteína representada por un vector $P = \left(\frac{N_1}{L}, \frac{N_2}{L}, \dots, \frac{N_{20}}{L}\right)$, donde N_i representa la cantidad de aminoácidos del tipo i en la secuencia y L es la longitud de la secuencia.

Por otra parte, la Pseudo-composición de aminoácidos (PseAAC) de tipo I fue definida en (Chou, 2001) con el objetivo de incorporar la información sobre el orden de aminoácidos al perfil de frecuencia de los mismos. Chou define la Pseudo-composición a partir de una cadena de aminoácidos R_1 R_2 R_3... R_L, como un conjunto de factores de correlación de orden λ de acuerdo a las distintas propiedades físico-químicas de los aminoácidos. Una proteína puede ser representada como $P = [\psi_1 \psi_2 \dots \psi_{20} \psi_{20+1} \dots \psi_{20+\lambda}]^T$ con los primeros veinte componentes similares a los de AAC, mientras que los λ componentes adicionales, aparecen en correspondencia con diferentes factores de correlación dentro de la cadena de aminoácidos. En este descriptor se incluyen las propiedades de hidrofobicidad, el valor hidrofílico y la masa de encadenamiento de los aminoácidos.

La composición de aminoácidos es otro de métodos libres de alineamiento para describir proteínas. Se representa en tres descriptores: Composición (C), Transición (T) y Distribución (D) (CTD) que recogen información a partir de la división de aminoácidos en tres clases de acuerdo con el valor de sus atributos como la hidrofobicidad, el volumen normalizado de van der Waals, la polaridad, etc. De esta forma cada aminoácido es acompañado por cada uno de los índices correspondientes a las clases 1, 2 y 3. El descriptor C: representa el por ciento global de cada clase (1, 2 y 3) en la secuencia, el descriptor T, el por ciento de frecuencia con la cual la clase 1 es seguida por la clase 2 o la 2 es seguida por la 1. El descriptor D representa la distribución de cada rasgo en la secuencia codificada (Dubchak et al., 1995, Dubchak et al., 1999).

La autocorrelación se representa en cuatro descriptores: Norm Moreau Broto, Moran, Geary y Total. Los tres primeros que están basados en determinadas propiedades de los aminoácidos que son normalizadas en conjunto en el Total (Cao et al., 2013). Por otra parte, los descriptores Quasi-Sequence-Order (QSO) representan una combinación de la composición y

la correlación de las propiedades de los aminoácidos definidas por Chou KC (2000) (Chou, 2000).

Otro de los métodos libres de alineamiento consiste en la determinación de la abundancia de cada secuencia de longitud *k* (*k*-mer) en las secuencias a comparar (Melsted and Pritchard, 2011), (Boden et al., 2013, Rizk et al., 2013). Cada proteína puede ser representada como un vector con la frecuencia de aparición de cada posible *k*-mer en el alfabeto de aminoácidos. Se encuentran en la literatura referencias a varios programas que pueden ser utilizados para el cálculo de los *k*-mers con diversas soluciones a la alta complejidad espacial requerida para almacenar la frecuencia de posibles *k*-mers para genomas completos, como (Rizk et al., 2013), (Marcais and Kingsford, 2011) y (Melsted and Pritchard, 2011). Sucede que estos programas al igual que otras herramientas de cálculo de medidas de similitud tanto basadas en alineamiento como libres de alineamiento no se encuentran implementados en una plataforma integradora de diversos rasgos de comparación de proteínas.

La importancia de la integración de diversas medidas de similitud en los procesos de clasificación funcional o evolutiva se ha evidenciado en (Chen et al., 2016) y trabajos previos realizado en el Centro de Investigaciones de Informática de la Universidad Central "Marta Abreu" de Las Villas (UCLV) (Galpert et al., 2018). Las limitaciones del cálculo de las comparaciones para a par integrando varias medidas se ha estudiado desde el punto de vista de la escalabilidad, conduciendo a la implementación big data de dichos cálculos en (Pérez et al., 2018) dentro de un sistema big data analítica en Spark.

1.2 Sistema de big data analítica en Spark

Los sistemas de big data analítica constituyen una de las áreas de investigación en la ejecución con una gran cantidad de retos y necesita para cumplir con los requisitos computacionales de análisis masivo de datos. Una "armazón" eficiente es esencial para diseñar, implementar y manejar los algoritmos y tuberías requeridas. Apache Spark ha aparecido como un motor unificado para análisis de datos de gran escala a través de una colección variada de cargas de trabajos. Ha introducido un acercamiento nuevo para la ciencia de datos e ingeniería donde una gran variedad de problemas de datos puede ser solucionada usando un almacenamiento sencillo y un motor de procesamiento con lenguajes para usos generales. Es por esto que

Apache Spark ha sido adoptado como esta “armazón” rápida y dimensionable requerida para desarrollar proyectos de análisis de datos a gran escala en (Salman Salloum, 2016).

Spark es diseñado para ser altamente accesible, proponiendo APIs simples en Python, Java, Scala, y SQL conjuntamente con bibliotecas incorporadas. También se integra estrechamente con otras herramientas Big Data. En particular, Spark puede acceder a cualquier fuente de datos Hadoop, incluyendo a Cassandra en (Holden Karau, february 2015). El sistema de archivos distribuidos Hadoop Distributed Archive Sistema (HDFS) provee rendimiento específico alto de acceso de datos, brinda facilidades como la elasticidad para la falla del hardware, el acceso fluyente de datos, soporte para grandes conjuntos de datos, la dimensionalidad para los nodos con ancho de banda elevado, la localidad aplicativa para datos, la portabilidad a través de hardware heterogéneo y plataformas del software en (Mahmoud Soufi, july 2018).

Spark soporta un conjunto de herramientas del nivel más alto incluyendo a Spark SQL para SQL y los datos estructurados, MLlib para el aprendizaje automático, GraphX para gráfica, y Spark Streaming en (Isaac Triguero, 2015-2016)

La abstracción básica de programación en Spark son los llamados Resilient Distributed Dataset (RDD), colección de objetos inmutables y distribuidos. Cada RDD es particionado en varias nuevas particiones, las cuales son procesadas en diferentes nodos del clúster. Los RDDs pueden contener varios tipos de objetos, que pueden ser de distintos lenguajes, ya sea Java, Scala, o Python, además de clases definidas por el usuario. En Spark el procesamiento se expresa creando nuevos RDDs, transformando RDDs existentes, o llamando operaciones sobre un RDD para obtener un resultado. Oculto a la vista del usuario, Spark automáticamente distribuye los datos contenidos en los RDDs en el clúster y paraleliza las operaciones que realiza sobre los mismos. A su vez, Spark puede ejecutar sobre un clúster Hadoop (Karau et al., 2015). Spark reutiliza datos a través de múltiples cómputos, a la vez que mantiene la escalabilidad y tolerancia a fallas (Fernández et al., 2014).

En Spark se pueden crear RDDs de dos formas, una es cargando datos externos, y otra es distribuyendo una colección de objetos en el programa controlador (llamado driver). Una vez creados, los RDDs ofrecen dos tipos de operaciones sobre ellos, transformaciones y acciones,

las primeras construyen un nuevo RDD a partir de uno ya existente, y las segundas computan resultados a basados en un RDD, y estos son devueltos al programa controlador o salvados en un sistema de archivos externo (Karau et al., 2015). La **¡Error! No se encuentra el origen de l a referencia.** contiene las transformaciones básicas sobre RDD mientras que la **¡Error! No se encuentra el origen de la referencia.** contiene las acciones básicas a realizar sobre un RDD.

Tabla 1: Transformaciones básicas sobre RDD

Función	**Descripción**
Map	Aplica una función a cada elemento del RDD y retorna un nuevo RDD con el resultado.
Flatmap	Aplica una función a cada elemento del RDD y retorna un RDD con los contenidos de los iteradores retornados. Son utilizadas para extraer palabras.
Filter	Retorna un RDD que contiene los elementos que pasaron la condición enviada para filtrar.
Distinct	Remueve duplicados.
sample (withReplacement, fraction, [seed])	Toma una muestra de un RDD.
Union	Produce un RDD que contiene los elementos de dos RDD.
Intersection	Retorna un RDD que contiene sólo los elementos que coinciden en dos RDD.
Subtract	Remueve los elementos contenidos en un RDD. Por ejemplo: datos de entrenamiento.
Cartesian	Producto cartesiano con otro RDD.
zipWithIndex	Indexa los elementos contenidos en un RDD

Tabla 2: Acciones básicas sobre RDD

Función	**Descripción**
collect()	Retorna todos los elementos de un RDD.
count()	Retorna el número de elementos en un RDD.
take(num)	Retorna el número de elementos del RDD.
top(num)	Retorna los top num elementos del RDD.
takeOrdered(num)(ordering)	Retorna num elementos ordenados según ordering.
takeSample(withReplacement, num, [seed])	Retorna num elementos aleatorios.
reduce(func)	Combina los elementos del RDD juntos en paralelo.
fold(zero)(func)	Igual que reduce pero con el zero value indicado.
aggregate(zeroValue)(seqOp, combOp)	Similar a reduce pero utilizado para retornar un tipo diferente.

Las transformaciones y las acciones son diferentes a causa de la forma en que Spark maneja computacionalmente los RDDs, aunque se puedan definir nuevos RDDs en cualquier momento, Spark solo procesa los mismos en modo perezoso, computándose estos la primera vez que son utilizados en una acción (Karau et al., 2015). Estas características son esenciales cuando se trabaja con enormes cantidades de datos.

En específico, el manejo de datos de secuencias de proteínas se favorece con la estructura llave, valor de los RDD en (Holden Karau, february 2015) ya que las proteínas se representan por su identificador y su secuencia siendo estos los componentes de la estructura de los RDD propuestos en el sistema de big data analítica abordado en la siguiente sección.

1.3 Avances y limitaciones del sistema de big data analítica para el análisis de proteínas en la UCLV

Para el entorno de big data de la UCLV utilizando Spark, se propone en (Arteaga, 2018) una implementación del cálculo de descriptores de proteínas libres de alineamiento en el modelo de programación Apache Spark, como parte de un sistema de big data analítica para comparar pares en grandes conjuntos de datos de proteínas incluyendo proteomas completos. En la Ilustración 1 se muestra el esquema general del sistema.

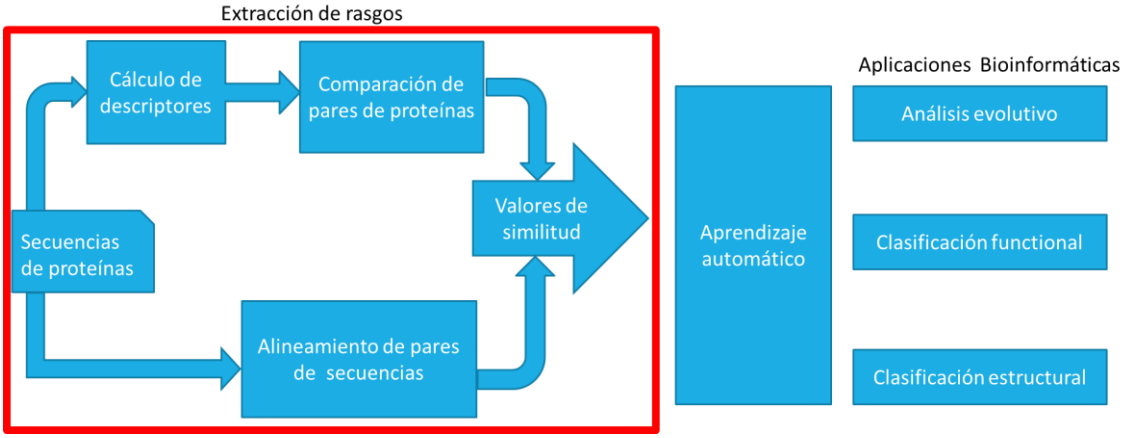

Ilustración 1: Esquema general del sistema de big data analítica propuesto para la UCLV.

La implementación se realiza utilizando PySpark con acceso a los archivos .fasta de las proteínas desde el sistema HDFS. Una vez cargadas las secuencias se manejan en RDDs con la estructura identificador de la proteína y secuencia que la compone. Estas estructuras son transformadas mediante la llamada a los cálculos de cada descriptor aplicado a cada secuencia y los vectores numéricos resultantes del cálculo se aparean, luego pasan hacer comparadas o alineadas según la medida de similitud seleccionada, los resultados son depositados en carpetas por descriptor en HDFS.

La implementación referida calcula los siguientes descriptores libres de alineamiento:

- Pseudo-composición de aminoácidos
- Auto correlación
  - I. Norm Moreau Broto Autocorrelation
  - II. Moran's Autocorrelation

III. Geary's Autocorrelation

IV. Total Autocorrelation

- Composición, Transición y Distribución

  I. Composition descriptor (CTD_C)

  II. Transition descriptor (CTD_T)

  III. Distribution descriptor (CTD_D)

  IV. Composition, Transition, Distribution descriptor (CTD)

- Quasi Sequence Order Descriptors

  I. Quasi-sequence Order Coupling Number

  II. Quasi-sequence order descriptor

- *k*-mers
- *k*-mers espaciados

Sin embargo, dicha implementación debe ser mejorada desde los puntos de vista siguientes:

(i) El cálculo de descriptores libres de alineamiento de proteínas debe permitir la ampliación en el rango de valores de los parámetros. De acuerdo a la experimentación realizada en un Spark local el cálculo de *k*-mers admite valores de $k < 4$ al igual que la Pseudo-composición de aminoácidos. En el caso del cálculo de *k*-mers se generan inicialmente las 20^k variaciones con repetición de todos los aminoácidos lo que conduce a una sobre carga en la búsqueda de los *k*-mers que existen en cada secuencia, La función de *k*-mers espaciados funciona de manera similar excepto que en un paso intermedio paraleliza la formación de patrones de *k*-mers con espacios, donde no se incluyen espacios en los extremos, por ejemplo: "101" para *k*-mers con *k* igual a 2 y un espacio, y "10101", "10011" y "11001" para *k*-mers de tamaño 3 con dos espacios.

(ii) La comparación por pares de proteínas debe realizarse utilizando diversas medidas de similitud aplicadas a descriptores libres de alineamiento

(iii) El cálculo de similitud por pares utilizando las medidas de similitud basada en alineamientos no están implementadas en la versión anterior.

1.4 Consideraciones finales del capítulo

En este capítulo se presentan los aspectos teóricos básicos para implementar los cálculos de medidas de similitud entre pares de proteínas. Se presentan las características de los sistemas big data analítica en Spark y los avances y limitaciones del sistema propuesto para el análisis de proteínas en la UCLV, aclarando de este modo la problemática por la cual se renueva la implementación. La transformación del cálculo de descriptores de proteínas debe reducir la carga computacional en el caso del cálculo de *k*-mers y las pruebas que se realicen a otros descriptores deben demostrar un aumento el rango de los parámetros de los descriptores. Se deben incluir comparaciones por pares utilizando diversas medidas de similitud y en general la aplicación propuesta debe permitir la combinación de medidas de similitud para pares de proteínas incluyendo tanto las libres de alineamiento como las basadas en alineamiento.

CAPÍTULO 2. Diseño e implementación de la aplicación

En este capítulo se presentan las medidas de similitud combinadas en un sistema de información o en un sistema de decisión si se conocen las clases de las proteínas en una aplicación bioinformática dada. Se muestra el diseño del cálculo de estas medidas en PySpark y la implementación de las mismas. Se muestra además cómo debe realizarse el despliegue y el modo de ejecución de la aplicación por parte del usuario.

2.1 Medidas de similitud combinadas

Partiendo de la representación de dos conjuntos de proteínas (incluyendo proteomas completos) $P_1 = \{x_1, x_2, \dots, x_n\}$ y $P_2 = \{y_1, y_2, \dots, y_m\}$, con n y m secuencias de proteínas, respectivamente, en (Galpert, 2016) se definen los rasgos que representan los valores continuos de las medidas de similitud $S_r: X \times X \rightarrow \mathbb{R}$, donde r es la identificación del rasgo de comparación de proteínas y X el dominio de cada rasgo. Estas medidas cumplen con las propiedades de una similitud según (Deza, 2006).

La medida de similitud global de las secuencias, S_1 en la expresión 2.1, se basa en la puntuación óptima en bit-score del alineamiento global c_g de Needleman-Wunsch (Needleman and Wunsch, 1970), mientras que la medida de similitud local S_2 (expresión 2.2), se basa en la puntuación óptima en bit-score del alineamiento local de Smith-Waterman (Smith and Waterman, 1981). Ambos alineamientos son calculados con una matriz de sustitución M y penalizaciones de gaps especificados (*go*- GOP y *ge*- GEP).

$$S_1(X_i, Y_j) = \begin{cases} c_g(x_i, y_j), & c_g(x_i, y_j) > 0 \\ 0, & c_g(x_i, y_j) \leq 0 \end{cases}$$
$$c_g(X_i, Y_j) = \frac{nwalign(x_i, y_j, M, go, ge)}{max\left(nwalign(x_k, y_p, M, go, ge)\right)}, \quad (2.1)$$
$$\forall k \in [1, n], \forall p \in [1, m]$$

$$S_2(x_i, y_j) = \begin{cases} c_l(x_i, y_j), & c_l(x_i, y_j) > 0 \\ 0, & c_l(x_i, y_j) \leq 0 \end{cases}$$
$$c_l(X_i, Y_j) = \frac{swalign(x_i, y_j, M, go, ge)}{max(swalign(x_k, y_p, M, go, ge))}, \quad (2.2)$$
$$\forall k \in [1, n], \forall p \in [1, m]$$

La medida S_3 (expresión 2.3) se calcula a partir de la longitud *(L)* de las secuencias de proteínas utilizando la diferencia normalizada para valores continuos (Deza, 2006). Precisamente, la longitud de las secuencias puede ser vista como las posiciones relativas de los nucleótidos/aminoácidos dentro del mismo gen/proteína en diferentes especies, y en regiones genómicas duplicadas dentro de la misma especie. Considerando que la longitud está afectada por la inserción y eliminación de segmentos de ADN a través del tiempo, esta medida puede ser utilizada para estudiar las diferencias en la longitud de regiones homólogas. Tiene en cuenta que las especies relacionadas más lejanas tienen mayor probabilidad de tener diferencias en las longitudes (Kumar and Filipski, 2007).

$$S_3(x_i, y_j) = 1 - \frac{|L(x_i) - L(y_j)|}{max(L(z_k)) - min(L(z_k))}, \quad (2.3)$$
$$z = x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m$$
$$\forall k \in [1, n + m]$$

La medida S_4 (expresión 2.4), construida a partir del método de Codificación Predictiva Lineal (Deza, 2006), está basada en la representación espectral de las secuencias a partir del alineamiento de un par de proteínas. Primeramente, cada aminoácido perteneciente a una región de correspondencia sin gaps entre las dos secuencias alineadas, es reemplazado por su energía de contacto (Miyazawa and Jernigan, 1999). Seguidamente, se calcula el promedio de esta propiedad físico-química dentro de un tamaño de ventana predefinido *W*, el cual se denomina media móvil para cada espectro. La medida de similitud $Corr(MX, MY)$ entre dos representaciones espectrales de regiones de correspondencia se calcula utilizando el coeficiente de correlación de Pearson (Deza, 2006) y su correspondiente valor de significación. Finalmente, las similitudes significativas de las *R* regiones sin gaps son agregadas considerando la longitud len_k de cada región *k*. De este modo, este rasgo está encaminado a manejar secuencias con similitud funcional, a pesar de tener baja identidad de aminoácidos (<30%). Para comparar vectores de datos continuos se pueden utilizar otras

medidas como Bray–Curtis, Euclidian, Canberra, Minkowski, Kulsinski, Hamming, Chebyshev, City Block (Manhattan) (Deza, 2006).

$$S_4(x_i, y_j) = \frac{\sum_{k=0}^{R} Corr(MX_{ik}, MY_{jk}) \times len_k}{\sum_{k=0}^{R} len_k}$$
$$Corr(MX, MY) = \begin{cases} Corr(MX, MY) & , sig \leq 0{,}05 \\ 0 & , sig > 0{,}05 \end{cases} \quad (2.4)$$

Para el cálculo de una medida de similitud S_r para un par de secuencias para un descriptor en la (expresión 2.5) se muestra la función para comparar los vectores numéricos del descriptor. En este caso a modo ilustrativo se ha utilizado la correlación de Pearson y $DescriptorX$ y $DescriptorY$ representan los vectores correspondientes a las dos secuencias a comparar.

$$S_r(x_i, y_j) = \begin{cases} Corr(DescriptorX, DescriptorY) & , sig \leq 0{,}05 \\ 0 & , sig > 0{,}05 \end{cases} \quad (2.5)$$

La combinación de medidas se expresa al conformar un sistema de información o un sistema de decisión a partir de las diversas medidas de similitud calculadas. De esta forma, dado un conjunto $A = \{S_r(x_i, y_j)\}$ de rasgos o atributos de los pares de proteínas, definidos como los valores continuos o discretos de r funciones de medidas de similitudes es posible representar un sistema de decisión de la clasificación de pares de proteínas como $DS = (U, A \cup \{d\})$, donde $U = \{(x_i, y_j)\}, \forall x_i \in P_1, \forall y_j \in P_2$ es el universo de pares de proteínas de los conjuntos de proteínas P_1 y P_2, y $d \notin A$ es el atributo de decisión obtenido de una clasificación curada. En varias aplicaciones como la clasificación estructural de proteínas, pudieran coincidir los conjuntos P_1 y P_2, no siendo así para aplicaciones como la comparación de genomas (o proteomas) de especies diferentes. Las comparaciones a realizar son aquellas comparaciones todos-vs.-todos entre los por pares por encima de la diagonal principal de la matriz de similitud de cada rasgo. El sistema DS o el sistema de información correspondiente sería la entrada de información para la fase de aplicación de algoritmos de aprendizaje automático dentro del sistema de big data analítica (ver Ilustración 1).

2.2 Diseño del cálculo de medidas de similitud en PySpark

El programa “BigD_DescriptorMod.py” calcula las medidas de similitud a partir de un archivo FASTA que contiene el conjunto de secuencias de proteínas o el proteoma. Este archivo

FASTA tiene la estructura siguiente: dos líneas para cada proteína, donde la primera línea comienza con el carácter “>” y seguidamente incluye el identificador de la proteína, y la segunda línea contiene la cadena de aminoácidos de un alfabeto de veinte de ellos. La Ilustración 2 muestra una sección de un archivo FASTA de la familia de enzimas GH70.

```
1  >AJE22990.1 dextransucrase [Azotobacter chroococcum NCIMB 8003]
2  MRASPSQFFAISLLSIAISGLLSGAAVAAPAPTALEQVPDGKGGVKWQEVTHDASAEEEQKGQDPKKFLGIQAITTEPDGSVKVEMGKPEVRQP
3  >AJH79253.1 alpha amylase, catalytic domain protein [Bacillus coagulans DSM 1 = ATCC 7050]
4  MEKKFFSRLSILMLSLLLVAGSISYFPKSAKAYTSGTSLDNRVIFQSFSLYMPYESNMYKILSAKGNELKDWGITDIWLPPAYRSFNAARYMEG
5  >AVD57793.1 dextransucrase [Bacillus coagulans]
6  MLSLLLVAGSISYFPKSAKAYTSGTSLDNRVIFQSFSLYQPYESNMYKILSAKGSELKDWGITDIWLPPAYRSFNMARYMEGYAIADRYDLGEF
7  >AFS71545.1 Dextransucrase [Exiguobacterium antarcticum B7]
8  MKNTKKVSVGLLATLVATSSFGVAPKQAAAYTSGEKLDNHVIFQSFSLYQPYDSNMYRTLAKKGDLLNSWGVTDVWMPPAYRSFDMARYMEGYA
9  >ACB62096.1 Dextransucrase [Exiguobacterium sibiricum 255-15]
10 MKNTKKVSAGLLATLVATSSFGVAPKQAAAYTSGEKLDNHVIFQSFSLYQPYDSNMYRTLAKKGDLLNSWGVTDVWMPPAYRSFDMARYMEGYA
11 >AOS98796.1 dextransucrase [Exiguobacterium sp. U13-1]
12 MNKTKKVSTGLLAALVATSGLTYAPESAKAFAPSEKLDNRVIFQSFSLYQPYESNMYRTLAKKGELLNSWGVTDVWLPPAYRSFDMARYMEGYA
13 >GAP05007.1 glycosyl hydrolase, partial [Fructobacillus tropaeoli]
14 MRKKLYKSGKMWVAASVAISFSALSISVGNNGAKADDSQQSSTQIQSTQVTTALPAGGQYSTTNGGQSWNYLVNGVAIKGMYQDGQGQLRYFNF
15 >AKM18207.1 Glucosyltransferase-SI precursor [Geobacillus sp. 12AMOR1]
16 MIKKYVHRTVALAVALLIIFGQIGVFPGKAHAYSSGPELDNRVIFQSFSLYQPYESNMYKILATKGDLLKEWGITDVWLPPAYRSFNMARYMEG
17 >SFV41414.1 Choline binding protein A [Lactobacillus acidipiscis]
18 MERKQRYKMYKSGKNWVIAPLVFFGIALGFQAGVHNVFADETIQKPQVVNLNSPVGDQKENSTSTDSQEDTASDKDPIQTNGEEKVANQSQSEK
19 >CCK33644.1 dextransucrase [Lactobacillus animalis]
20 MFEKKLHYKMYKAGKHWVFAAITVGIFGFASTTSALADETSSSNEAQTEQTLNINEAADTTTDVSNEEKATEAKLTTQAVDTASSEEKTTNVEK
21 >ANK68614.1 glycosyl hydrolase family 70 (plasmid) [Lactobacillus backii]
22 MTGLRHYGNNKLEYYGTDHVQYRNRYASQGNQLYYFGSNGDAMVTIRGAIENGKFNIYDMRTNKLIKSLDAGTWENLAYSMDANSINNVDGYLS
23 >CCK33643.1 dextransucrase [Lactobacillus curvatus]
24 MLRNNYFGETKTHYKLYKCGKNWAVMGISLFPLGLGMLVTSQPVSADVTATSTSSSAVRTDAISESSSSAAKAETTSASSSSAVKAETTSASSS
25 >ASA47807.1 putative GTFB [Lactobacillus delbrueckii]
26 MNSRKKMMNEPSSEKQRWSLRKISVGMTSVLLGATMFWASGAGSQVKADTTTASEQAAQTQTNTARKAAGTESTEQTEDNNANADAGKTAETPA
27 >ASA47803.1 putative GTFB [Lactobacillus delbrueckii]
28 MYHMKGNPQNANNHLSYNEGYHSGAARMLNKKGNPQLYMDSGEFYSLENVLGRANNRDNISDLVTNSIVNRQNDVTENEATPNWSFVTNHDQRK
29 >ASA47809.1 putative GTFB [Lactobacillus delbrueckii]
30 MYYFGSNGDAVTGLRHYGNNKLEYYGADHVQYRNRYYQEGNKFYYFGGNGDAMVTIRGAIENGKFNIYDIRTNKLIKSLDAGTWENLAYSMDAN
31 >ASA47808.1 putative GTFB [Lactobacillus delbrueckii]
32 MSVGMTSVLLGATMFWASGAGSQVKADTTTASEQAAQTQTNTARKAAGTESTEQTEDNNANADAGKTAETPAATNNGSQETTTESTAASSSAQN
33 >ASA47810.1 putative GTFB [Lactobacillus delbrueckii]
34 MNSRKKMMNEPSSEKQRWSLRKISVGMTSVLLGATMFWASGAGSQVKADTTTASEQAAQTQTNTARKAAGTESTEQTEDNNANADAGKTAETPA
35 >ASA47811.1 putative GTFB [Lactobacillus delbrueckii]
36 MSFQGFIVLFGVGSLSNDNSDNSPELLVGNDIDNSNPVVQAENLNWEYFLLNYGKLMGYNQDGNFDGFRIDAADNIDADVFDQMGQLMNDMYHM
37 >ASA47812.1 putative GTFB [Lactobacillus delbrueckii]
38 MVTIRGAIENGKFNIYDIRTNKLIKSLDAGTWENLAYSMDANSINNVDGYLSYSGWYRPIGTSQDGKTWYKTGAGDWRPILMYVWPNKDVQAQF
39 >ASA47813.1 putative GTFB [Lactobacillus delbrueckii]
40 MVTIRGAIENGKFNIYDIRTNKLIKSLDAGTWENLAYSMDANSINNVDGYLSYSGWYRPIGTSQDGKTWYKTGAGDWRPILMYVWPNKDVQAQF
41 >ASA47816.1 putative GTFB [Lactobacillus delbrueckii]
42 MVTIRGAIENGKFNIYDIRTNKLIKSLDAGTWENLAYSMDANSINNVDGYLSYSGWYRPIGTSQDGKTWYKTGAGDWRPILMYVWPNKDVQAQF
43 >ASA47814.1 putative GTFB [Lactobacillus delbrueckii]
44 MVTIRGAIENGKFNIYDIRTNKLIKSLDAGTWENLAYSMDANSINNVDGYLSYSGWYRPIGTSQDGKTWYKTGAGDWRPILMYVWPNKDVQAQF
```

Ilustración 2: Fragmento de archivo GH70.fasta

La estructura general del programa para calcular los distintos descriptores se muestra en el diagrama de actividades de la Ilustración 3. El programa carga la cantidad de secuencias pasada por parámetros realiza el apareo del conjunto de datos a través de la trasformación “cartesian” seguido por un filtrado de las secuencias a ser comparadas (las que se encuentran por encima de la diagonal principal) ver diagrama de la Ilustración 4. Luego del apareamiento se procede al cálculo de las medidas para cada par de manera distribuida como se muestra en

el diagrama de actividades de la Ilustración 5. Cada medida de similitud tiene varios argumentos que serán especificados en la siguiente sección donde se presenta la propuesta Spark de sus implementaciones, también se pueden ver en Tabla 6.

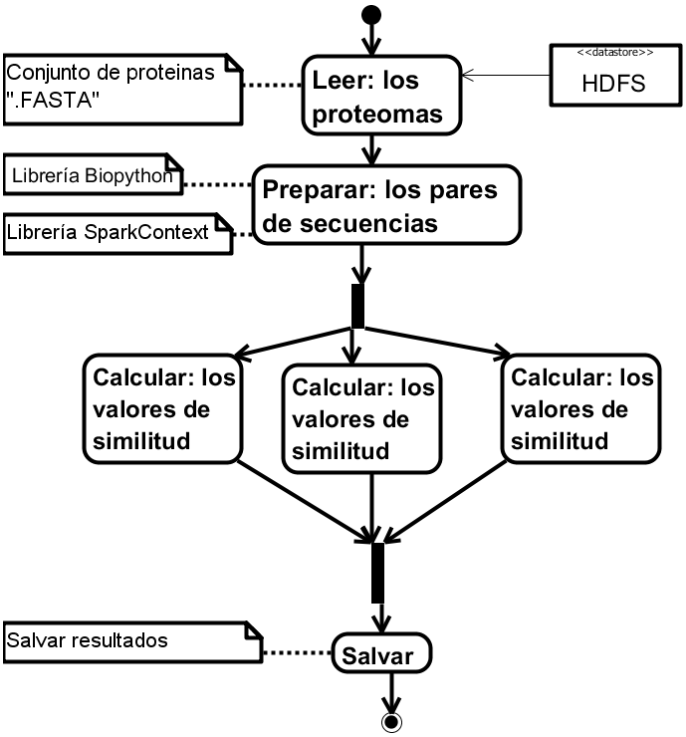

Ilustración 3: Estructura general del cálculo paralelo de los valores de similitud

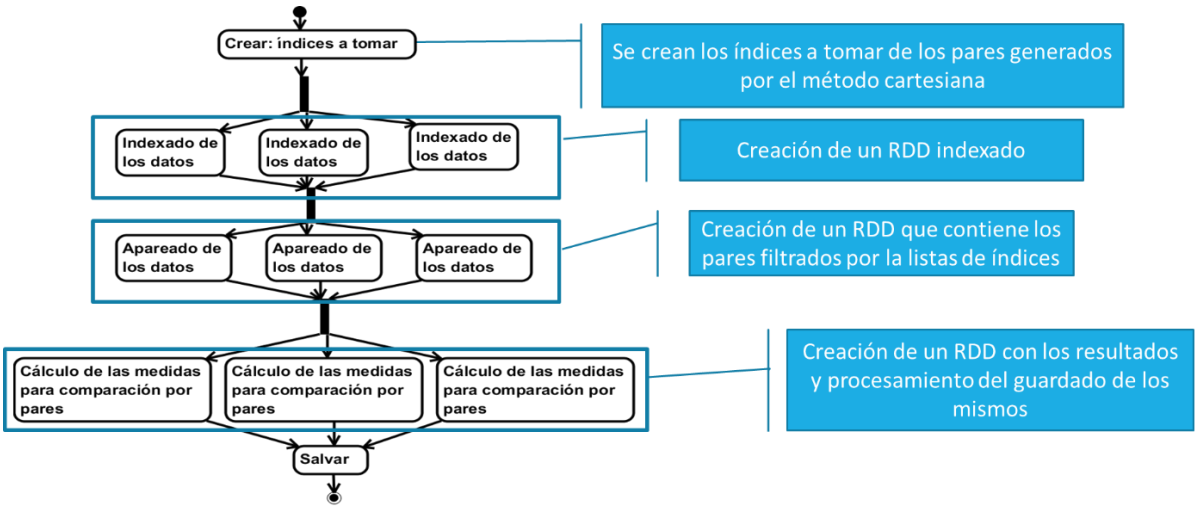

Ilustración 4: Creación de los pares

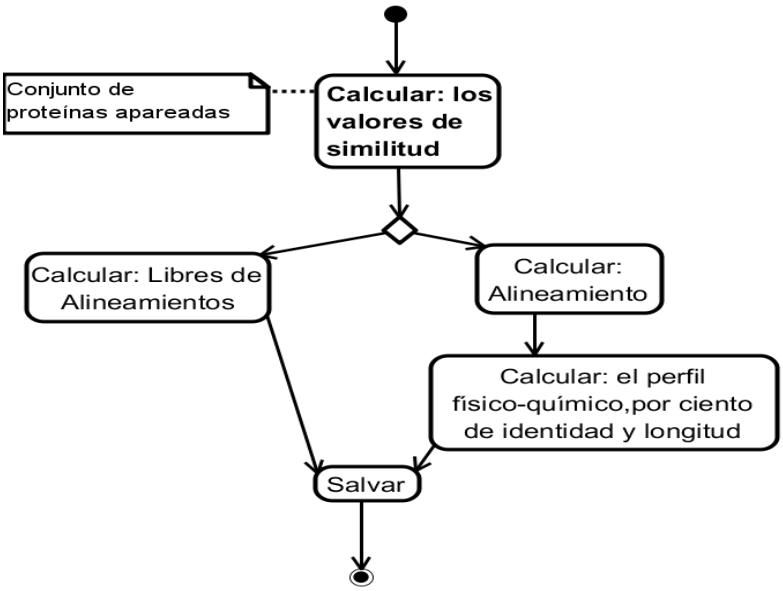

Ilustración 5: Estructura de elección del cálculo de descriptores o del alineamiento

En el caso de la implementación del cálculo basado en alineamiento de un par de proteínas. Primeramente, se calcula el alineamiento global de las secuencias, obtenido las secuencias alineadas para cada aminoácido perteneciente a una región de correspondencia sin gaps entre las dos secuencias alineadas, es reemplazado por su energía de contacto (Miyazawa and Jernigan, 1999). Seguidamente, se calcula el promedio de esta propiedad físico-química dentro de un tamaño de ventana predefinido W, el cual se denomina media móvil para cada espectro. A continuación, se calcula la medida de similitud $Corr(MX, MY)$ entre dos representaciones espectrales de regiones de correspondencia, utilizando el coeficiente de correlación de Pearson y su correspondiente valor de significación. Después, las similitudes significativas de las R regiones sin gaps son agregadas considerando la longitud len_k de cada región k. Finalmente, con las dos secuencias alineadas se calcula el porcentaje de identidad. Incluido a este algoritmo también se procesa el cálculo de la longitud renormalizada y el alineamiento local que presenta las dos secuencias que se están analizando (ver Ilustración 6).

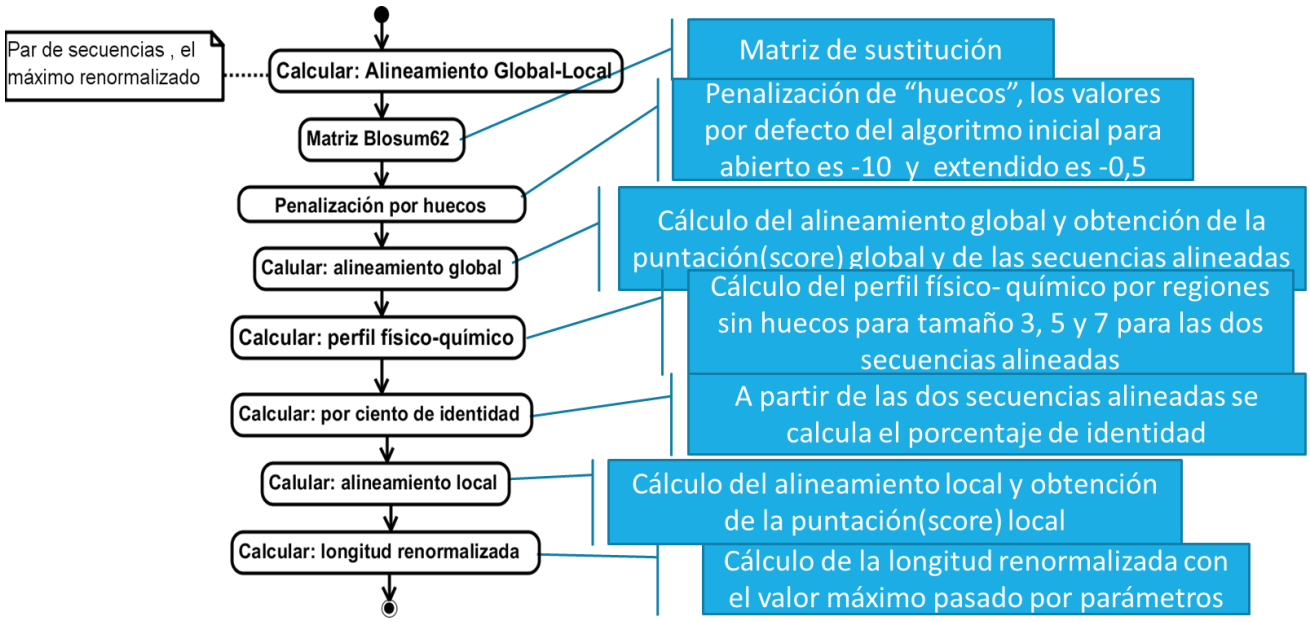

Ilustración 6: Diagrama de actividad para el alineamiento

Una vez calculadas las medidas de similitud, el resultado de estas comparaciones se devuelve en una estructura que contiene un tipo de dato diccionario.

2.3 Implementación

Todo el código se encuentra dentro de un script de nombre "BigD_ProtDescritorsMod.py" el cual utiliza librerías, como "PEARSONR" para el cálculo de la correlación Pearson, "DISTANCE" para el procesamiento de los cálculos de distancia cuando se hallan los valores de similitud, "BIO" con esta se trabajan las cadenas de aminoácidos de las proteínas y "SparkContext" para paralelizar la ejecución y salvar los resultados del cálculo en archivos, dentro de una carpeta en el sistema HDFS como en la Ilustración 9.**¡Error! No se encuentra el origen de la referencia.**

De forma general, todas las medidas de similitud utilizan como entrada los archivos FASTA, ubicados en el sistema de archivos distribuidos HDFS, distribuyen el cálculo en los distintos procesadores de los nodos del clúster mediante la transformación Map que ejecuta una función de cálculo de una medida de similitud, previamente implementada en el programa "BigD_ProtDescritorsMod.py" (Código fuente 1), y como resultado salvan el cálculo en archivos dentro de una carpeta en el sistema HDFS.

El flujo de creación de RDDs se especifica a continuación, igualmente de manera genérica indicando el orden de los pasos.

Código fuente 1 conjunto de argumentos y asume valores por defecto

```
 #El programa recibe un conjunto de argumentos y asume valores por
defecto como se especifican a continuación:

1- parser = argparse.ArgumentParser (description='Script for Protein
Descriptors Calculations.')

parser.add_argument('-lm', '--LocalMode', action='store_true', help='
Cluster or Local Mode')

parser.add_argument ('-k', type=int, help='K-mers size (from 1 to
7)')

parser.add_argument ('-s', type=int, help='Space size (from 1 to 4)')

parser.add_argument ('-sp','--sizesplit', type=int, help='Splitting
size')

parser.add_argument ('-f', '--fasta', required=True, help='input
fasta file name')

parser.add_argument ('-l', '--lambdax', type=int, help='Pseudo amino
acid composition, (input Lamda)')

parser.add_argument ('-an', '--NMB', action='store_true',
help='Autocorrelation for Norm Moreau Broto')

parser.add_argument ('-fs', '--filter', action='store_true',
help='filter for small selection')

parser.add_argument ('-sf', '--savefile', action='store_true',
help='saving pairing tuples')

parser.add_argument ('-am', '--Moran', action='store_true',
help='Autocorrelation for Moran')

parser.add_argument ('-ag', '--Geary', action='store_true',
help='Autocorrelation for Geary')

parser.add_argument ('-at', '--Total', action='store_true',
help='Autocorrelation total')

parser.add_argument ('-cc', '--CTD_C', action='store_true',
help='Composition descriptors (CTD_C)')

parser.add_argument ('-ct', '--CTD_T', action='store_true',
help='Transition descriptors (CTD_T)')

parser.add_argument ('-cd', '--CTD_D', action='store_true',
help='Distribution descriptors (CTD_D)')

parser.add_argument ('-ctd', '--CTD', action='store_true',
help='Composition, Transition, Distribution descriptors (CTD)')
```

```
parser.add_argument ('-d', '--dst', action='store_true', help='Avoid
to calculate Distances')
parser.add_argument ('-qcn', '--QSOCN', action='store_true',
help='Quasi sequence order coupling numbers ')
parser.add_argument ('-qso', '--QSO', action='store_true',
help='Quasi-sequence order descriptors')
parser.add_argument ('-m', '--maxlag', type=int, help='Maxlag
(default 30)')
parser.add_argument ('-w', '--weight', type=float, help='Weight
(default 0.1)')
parser.add_argument ('-z', '--running', type=str, help='running
number')
parser.add_argument ('-al', '--align', action='store_true',
help='cálculo de alinamiento por parte')
```

En la Tabla 3 se especifica el proceso de cálculo de cada medida de similitud.

Tabla 3: Descripción del cálculo de cada medida de similitud.

Medidas	**Proceso de cálculo**
Pseudo-composición de aminoácidos	Carga el archivo .FASTA y lo distribuye entre los nodos de clúster. Llama para cada uno de los fragmentos a la función calculePseudoAcc, la cual se encarga del cálculo llamando al método _GetPseudoAAC implementado en PseudoACC de la librería propy de Python. Luego guarda en un archivo texto.
Auto correlación	<u>Norm Moreau Broto Autocorrelation</u> Carga el archivo .FASTA y lo distribuye entre los nodos de clúster. Llama para cada uno de los fragmentos a la función calculeNormMoreauBroto que se encarga del cálculo de esta Auto correlación con el método CalculateNormalizedMoreauBrotoAuto implementado en Autocorrelation que se encuentra en la librería propy. <u>Moran's Autocorrelation</u> Llama a la función calculeMoran que calcula esta Auto correlación con el método CalculateMoranAuto implementado en

	Autocorrelation de la librería propy. Geary's Autocorrelation Llama a la función calculeGearyAuto que calcula esta Auto correlación con el método CalculateGearyAuto implementado en Autocorrelation de la librería propy. Total Autocorrelation Llama a la función calculateAutoTotal que se encarga del cálculo de esta Auto correlación con el método CalculateAutoTotal implementado en Autocorrelation de la librería propy de Python.
Composición, Transición y Distribución	Composition descriptor (CTD_C) Llama a la función calculateC que se encarga del cálculo de este descriptor mediante el método calculateC implementado en CTD de la librería propy. Transition descriptor (CTD_T) Llama a la función calculateT que calcula este descriptor mediante el método calculateT implementado en CTD de la librería propy de Python. Distribution descriptor (CTD_D) Llama a la función calculateD que calcula mediante el método calculateD implementado en CTD de la librería propy. Composition, Transition, Distribution descriptor (CTD) Llama a la función calculateCTD que se encarga del cálculo de este descriptor mediante el método calculateCTD implementado en CTD de la librería propy.
Quasi Sequence Order Descriptors	Quasi-sequence Order Coupling Number Llama a la función GetSequenceOrderCouplingNumberTotal que

	se encarga del cálculo de este descriptor mediante el método GetSequenceOrderCouplingNumberTotal implementado en QuasiSequenceOrder de la librería propy. <u>Quasi-sequence order descriptor</u> Llama a la función GetQuasiSequenceOrder que se encarga del cálculo de estos descriptores mediante el método GetQuasiSequenceOrder implementado en QuasiSequenceOrderde la librería propy.
k-mers	# Crea un RDD con todos los *k*-mers que se encuentran en las secuencias con el método ownKmers y lo deposita en un nuevo RDD a este se le indexan todos los elementos con transformación zipWithIndex, después se aparean los elementos con la transformación cartesian y finalmente genera las subsecuencias de tamaño *k* con owKmers se pasa al cálculo del valor de las distancias de los *k*-mers a través del método measuresDistsNumeric3
k-mers espaciados	Esta función ejecuta de forma similar a la de *k*-mers, excepto que primeramente crea patrones con espacios a partir de los patrones formados que se encuentran en cada secuencia. Para esto utiliza el método ownsSpacedKmers para generar las subsecuencias de tamaño *k* y espaciado S
Alineamiento, perfil físico-químico y longitud	Esta se ejecuta primero la longitud normalizada con el método calculeLongitud, seguido se le indexan todos los elementos con transformación zipWithIndex, después se aparean los elementos con la transformación cartesian y finalmente calcula el alineamiento global y local, con el perfil físico-químico, el porciento de identidad y la longitud con el método alignGapMatGloLoc

En el caso específico del cálculo de la frecuencia de *k*-mers en una secuencia de proteínas, seguido se paraleliza el cálculo y ejecuta la función measuresDistsNumeric3 que se encarga del cálculo de las medidas de similitud por pares y se le pasa por parámetro el par de proteínas a comparar y el resultado de la función owKmers que conformar vectores de frecuencias de subsecuencias del tamaño de *k* deseado existentes en cada proteína, es decir que se encarga de contar la aparición de cada secuencia de tamaño *k* en una secuencia de proteína. Finalmente, une las frecuencias calculadas en paralelo para conformar el resultado completo. La función de *k*-mers espaciados funciona de manera similar excepto que la función que realiza la formación de patrones de *k*-mers con espacios, donde no se incluyen espacios en los extremos es ownsSpacedKmers.

Después con respecto a la medida de similitud escogida, es decir, pasada por parámetros para calcular se procesan estos cálculos a través de un RDD el cual distribuye los cálculos, finalmente guarda los resultados en una carpeta con los datos particionados, utilizando la función de la librería “SparkContext” (Código fuente 2). En la Ilustración 7 se especifica el diagrama de componentes de la aplicación.

Código fuente 2: Inicialización del SparkContext

```
# Inicialización del SparkContext. La inicialización de este puede
ser tanto local cuando la llamada es “spark-submit
BigD_ProtDescritorsMod.py –f <nombre del archivo .FASTA> [conjunto de
argumentos de los descriptores a calcular]”, o la otra variante puede
ser “spark-submit BigD_ProtDescritorsMod.py –f <nombre del archivo
.FASTA> [conjunto de argumentos de los descriptores a calcular]”


if args.LocalMode:
    sc = SparkContext("local[*]","ProteinMolecularDescriptors")
else:
    sc = SparkContext("yarn", "ProteinMolecularDescriptors")
#Carga del archivo FASTA en un RDD llamado “fullfasta” con todos los
elementos en una cadena de la forma “idproteina<+>secuencia”
fullfasta = sc.wholeTextFiles(logFile).flatMap(spliterData)
#Carga en un RDD llamado “fasta” mapeando “fullfasta” con creando
cada elemento conteniendo idproteina, secuencia
fasta = fullfasta.map(lambda x:(x.split('<+>')[0],x.split('<+>')[1]))
```

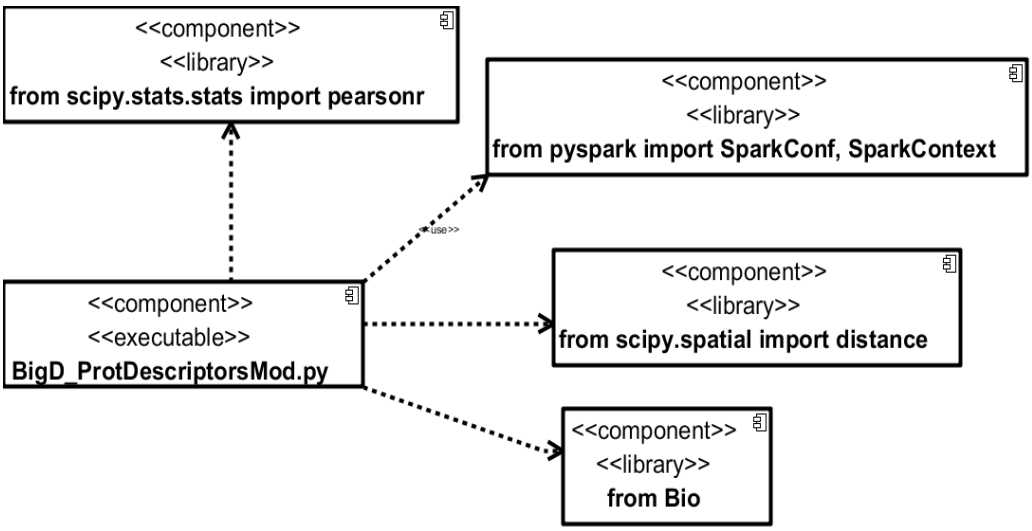

Ilustración 7: Diagrama de componentes

2.4 Despliegue

El clúster de big data de la UCLV es IBM iDataPlex d360 M2 y está compuesto por 30 nodos con dos procesadores Intel Xeon L5520 (8M Cache, 2.26 GHz, 5.86 GT/s Intel® QPI), 8 núcleos, 12 GB RAM, 2 x 1 Gbit Ethernet y una capacidad de almacenamiento de 148 GB (4.44 TB en general) cada uno. Su arquitectura se muestra en Ilustración 8 la incluyendo un sistema de archivos distribuidos HDFS, un sistema de operación de datos YARN que permite la interacción entre Spark y HDFS, y el propio Spark entre otros componentes.

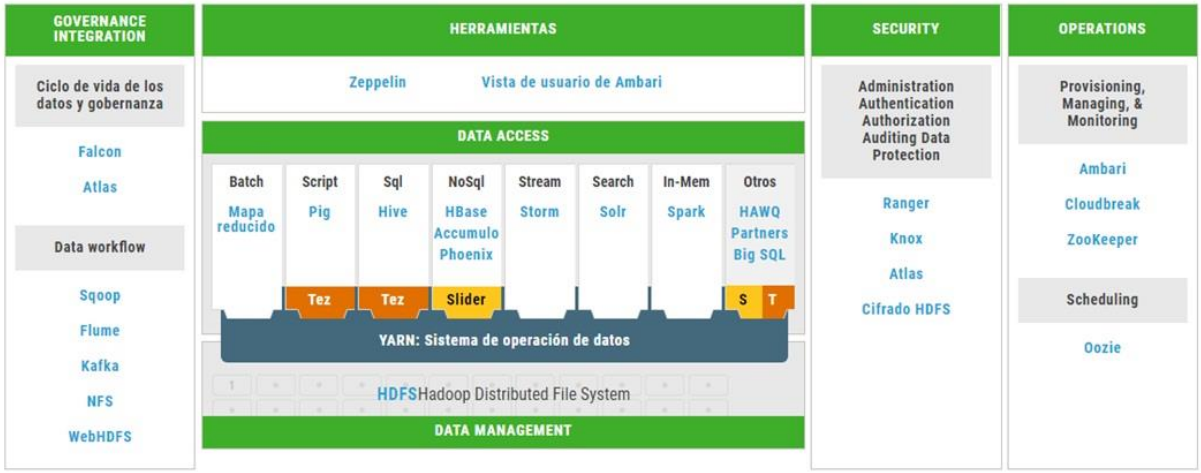

Ilustración 8: Arquitectura del clúster de big data en la UCLV

Para ejecutar el programa o aplicación en el clúster se debe utilizar el comando de Spark spark-submit (Karau et al., 2015) que de forma general se representa como:

```
spark-submit [options] <app jar | python file> [app options]
```

[options] Es una lista de banderas para spark-submit. Una lista de banderas comunes es enumerada la Tabla 4.

<app jar | python file> Se refiere al JAR o Script de Python conteniendo el punto de entrada hacia la aplicación.

[app options] Son opciones que serán pasadas a la aplicación. Si el método main () del programa llama a argumentos, se utiliza sólo [app options] y no las banderas específicas para spark-submit.

Tabla 4: Banderas comunes para spark-submit.

Bandera	Explicación
--master	Indica el manejador del clúster a quien conectarse. Las opciones para esta bandera están descritas en la Tabla 5
--deploy-mode	Para lanzar el programa del controlador localmente (“client”) o en una de las máquinas trabajadoras dentro del clúster (“cluster”). En spark-submit modo cliente el controlador ejecuta el programa en la misma máquina donde spark-submit está siendo invocado. En el modo clúster, el controlador envía para ejecutar en un nodo trabajador en el clúster. Por defecto está en modo cliente.
--class	La clase “principal” de la aplicación si se ejecutara un programa Java o Scala.
--name	Un nombre legible para la aplicación. Esto será exhibido en la web UI de Spark.
--jars	Una lista de archivos JAR para subir y colocar en el classpath de la aplicación. Si la aplicación depende en un pequeño número de JARs de terceros, se pueden añadir aquí.

--files	Una lista de archivos a ser colocados en el directorio de trabajo de su aplicación. Esto puede servir para archivos de datos a distribuir para cada nodo.
--py-files	Una lista de archivos a ser añadida al PYTHONPATH de la aplicación. Esta lista puede contener archivos .py, egg, o .zip.
--executor-memory	La cantidad de memoria a usar por los ejecutores, en bytes. Los sufijos pueden usarse para especificar mayores cantidades como “512m” (512 megabytes) o “15g” (15 gigabytes).
--driver-memory	La cantidad de memoria a usar para el proceso del controlador, en bytes. Los sufijos pueden usarse para especificar mayores cantidades como “512m” (512 megabytes) o “15g” (15 gigabytes).

Tabla 5: Valores posibles para la bandera --master en spark-submit

Valor	**Explicación**
spark://host:port	Conecta a un clúster de Spark Standalone en el puerto especificado. Por defecto los masters de Spark Standalone usan el puerto 7077.
mesos://host:port	Conecta a un master del clúster Mesos con el puerto especificado. Por defecto los masters de Mesos escuchan por el puerto 5050.
yarn	Conecta a un clúster de YARN. Al correr en YARN se necesita colocar la variable de ambiente HADOOP_CONF_DIR a apuntar la posición del directorio de configuración de Hadoop, el cual contiene la información acerca del clúster.
local	Ejecuta en modo local con un solo core.
local[N]	Ejecuta en modo local con N cores.
local[*]	Ejecuta en modo local y usa tantos cores como tenga la máquina.

En el caso específico del script BigD_DescriptorMod.py el spark-submit se ejecuta de la siguiente forma

spark-submit \

--master yarn \

--deploy-mode cluster \

--name "ProtDescriptors" \

2.5 Modo de uso de la aplicación

Primeramente, se abre una consola de Linux que tenga acceso a la instalación de Apache Sparck, te colocas sobre el directorio que contiene el script "BigD_ProtDescritorsMod.py", que tiene el código fuente a ejecutar, después existen dos formas de su ejecución:

1. Escribe en la consola el comando "`$ spark-submit --master yarn --deploy-mode cluster --num-executors 8 BigD_ProtDescriptorsMod_P5.py -f <nombre del archivo .FASTA> [conjunto de argumentos de los descriptores a calcular]`" , estos argumentos se muestra en la Tabla 6. Esta variante es para conectar a un clúster de YARN
2. Otra variante es "`$ spark-submit BigD_ ProtDescriptorsMod_P5.py -f <nombre del archivo .FASTA> -lm [conjunto de argumentos de los descriptores a calcular]`", estos argumentos se muestra en la Tabla 6. Esta variante es para conectar a un clúster de local

Tabla 6: Descriptores empleados en la experimentación con sus valores de parámetros.

Descriptor	Parámetros y el rango de valores si es necesario
Alineamiento perfil físico-químico y longitud	$-al$
Pseudo-composición de aminoácidos	$-\lambda\ Valores\ mayores\ igual\ que\ 1$
k-mers	$-k\ Valores\ mayores\ igual\ que\ 1$

k-mers /espaciado	$-k\ Valores\ mayores\ igual\ que\ 1$ $-s\ Valores\ mayores\ igual\ que\ 1\ y\ menores\ que\ k$
Auto correlación de Geary	$-ag$
Auto correlación de Moran	$-am$
Auto correlación Total	$-at$
Composición, Distribución y Transición (Composición) (CTD_C)	$-cc$
Composición, Distribución y Transición (Distribución) (CTD_D)	$-cd$
Composición, Distribución y Transición (Transición) (CTD_T)	$-ct$
Composición, Distribución and Transición (Total) (CTD)	$-ctd$
Quasi-Sequence-Order (QSO)	$-qso$ después $-m$ para maxlag, que este recibe 30 por defecto, seguido $-w$ que recibe 0.1 por defecto
Quasi-Sequence-Order Coupling Numbers (QSOCN)	$-qcn$ después $-m$ para maxlag, que este recibe 30 por defecto

2.6 Conclusiones parciales del capítulo

La utilización de la transformación zipWithIndex y seguidamente la transformación cartesian con filtrado permite aparear las secuencias antes de efectuar la comparación.

La forma de cálculo del descriptor *k*-mers al limitar la búsqueda de subsecuencias reduce la carga computacional de este cálculo permitiendo la ampliación en el rango del parámetro *k*.

La incorporación de los cálculos de medidas basadas en alineamiento permite que las secuencias sean analizadas sobre la base de descriptores y de medidas de homología de secuencias.

El PySpark permite el uso de bibliotecas de Phyton que facilitan la implementación y que se encuentran instaladas en el clúster de la UCLV.

CAPÍTULO 3. Validación de la aplicación

En este capítulo se abordan aspectos relacionados con los conjuntos de datos a utilizar para la ejecución de la validación de la aplicación. Seguidamente los experimentos realizados para la validación del código utilizando el clúster de Spark de la UCLV. Luego aparecen los resultados de tiempo de ejecución de los cálculos de medidas de similitud que se probaron. Finalmente, aparecen los resultados de prueba del software durante el testeo de los cálculos.

3.1 Conjuntos de datos

Para la experimentación se utilizaron las enzimas del conjunto GH70 con 482 proteínas. La conformación de los pares de enzimas seleccionados para los experimentos aparece en la dirección **/user/uclv_apuerto** del servicio HDFS del clúster de Spark de la UCLV como se muestra en la Ilustración 9. En esta figura las carpetas con el encabezado de "GH70_Align_Global_Local_results_run..." contienen los resultados de la ejecución de la medida similitud basada en alineamiento, este proceso se realizó en el clúster UCLV para 8 ejecutores.

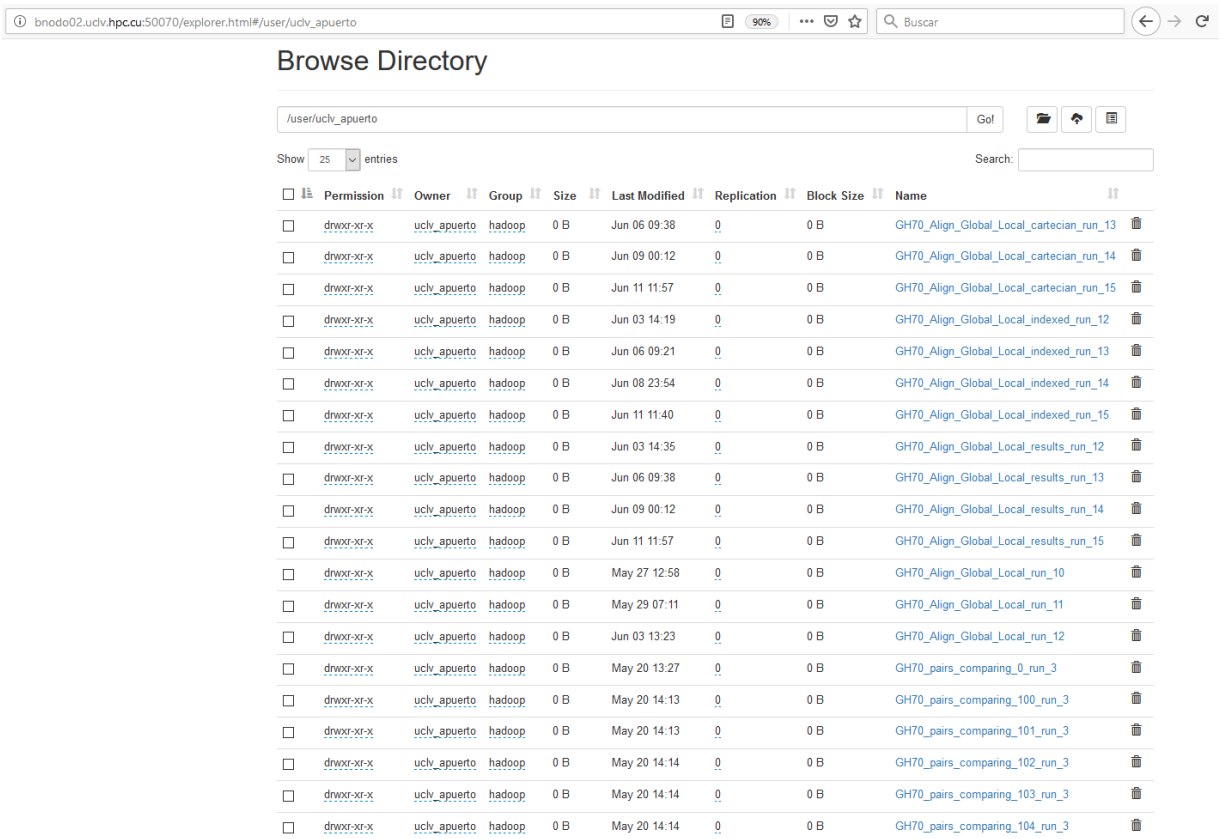

Ilustración 9: Datos guardados en el Hadoop

3.2 Experimentos realizados

Para la experimentación se utilizó el archivo GH70.fasta la Tabla 7. Con respecto a la infraestructura, se utilizaron la local del Centro Bioquímico (CBQ) y el clúster de la UCLV en la Tabla 8. Para este conjunto de datos se generan como resultado 115921 pares de secuencias

Tabla 7: Archivo fasta utilizado en la medición con cantidad de secuencias antes y después de un filtrado realizado

Archivo Fasta	Cantidad de Secuencias	Longitud Mínima de las Secuencias	Longitud Máxima de las Secuencias
GH70	482	74	2869

Tabla 8: Propiedades de los clúster utilizados

Rasgos	**Clúster CBQ**	**Clúster UCLV (big data)**
Modelo	Inspur Servir NFS 279M3	IBM iDataPlex dx360 M2
Nodos	1	30 en Total
Memoria	3,5 TB	148 GB (4.44 TB Total)
Procesador	Intel(R) Xoon(R) CPU ES-2630 2.66hz	2 x Intel Xeon L5520 (Gainestown or Nehalem-EP)
Núcleos	12	240 (8 Núcleos por cada Nodos)
RAM	64 GB	12 GB por cada Nodos
Ethernet	1 GB	2 GB

Los experimentos de tiempo de ejecución aparecen desglosados de la siguiente forma:

1. Tiempo de comparación de alineamiento en el clúster de la UCLV contra clúster CBQ

2. Tiempo de la comparación de alineamiento en el clúster UCLV para dos versiones de código

3. Tiempo de cálculo de *k*-mers en el clúster CBQ

4. Tiempo de cálculo de Pseudo-composición de aminoácidos en el clúster CBQ.

3.3 Resultados de tiempo de ejecución

1. Tiempo de la comparación de alineamiento en el clúster de la UCLV contra clúster CBQ

En la Tabla 7 se muestran los datos del conjunto de enzimas elegido para hacer las mediciones. En esta versión (Código fuente 3) se crean los pares a analizar para cada secuencia, y se realizan los cálculos de similitud en este caso el alineamiento por pares.

Código fuente 3 la primera versión para el cálculo del alineamiento

```
# Código del método aparear para el cálculo del alineamiento
def aparear(seqentrada):
    min = sys.float_info.max
    max = sys.float_info.min
    for seq in seqentrada:
        if len(seq[1]) < min:
            min = len(seq[1])
        if len(seq[1]) > max:
            max = len(seq[1])
    print("    Longitud m\u00EDnima: ", min)
    print("    Longitud m\u00E1xima: ", max)
    max -= min
    data = []
    N = len(seqentrada)
    for i in range(N):
      for j in range(i+1,N):
          data.append({'name1':seqentrada[i][0],
'name2':seqentrada[j][0],'seq1':seqentrada[i][1],'seq2':seqentrada[j]
[1],'longitud': max})
    return data


# Código de la primera versión para el cálculo del alineamiento, la
varaible results es un rdd que posee todo el conjunto de los datos


rddstore = sc.parallelize(aparear(results.collect()))
rddstore.saveAsTextFile(splitname + '_pairs_comparing_run_' +
args.running)
rddalign=rddstore.map(lambda x :
alignGapMatGloLoc(x['name1'],x['name2'],x['seq1'],x['seq2'],
x['longitud']))
rddalign.saveAsTextFile(splitname + '_Align_Global_Local_run_' +
args.running)
```

Tabla 9: Tiempo de ejecución de la primera variante del cálculo alineamientos comparada con el clúster del CBQ vs. UCLV

Archivo fasta	Alineamiento en el clúster CBQ	Alineamiento en el clúster UCLV para 8 ejecutores
GH70	1306min	2313min

Aunque el clúster de la UCLV para 8 ejecutores presenta más prestaciones que el clúster del CBQ, el resultado el tiempo de ejecución de la UCLV fue mayor debido a que en ocasiones según las bibliografías consultadas, el tráfico en la red para la comunicación entre los nodos se dificulta en la recopilación de la información y trabajo con la misma para grandes flujos de datos. También tener en cuenta que los recursos de los nodos en clúster big data de la UCLV están compartidos por todos los usuarios que estén trabajando sobre el mismo, es decir, que se compite por obtener los recursos que se van liberando.

2. Tiempo de la comparación de alineamiento en el clúster UCLV con el archivo GH70.fasta. En la Tabla 10 aparece la comparación de la primera variante del código del cálculo del alineamiento con respecto a la final para la misma medida de similitud el alineamiento por pares de secuencias. La versión final (Código fuente 4) primero calcula la longitud, guarda los pares generados por método “cartesian” con un filtrado de los pares que aparecen por encima de la diagonal principal de la matriz de similitud y después pasa a calcular de modo paralelo el alineamiento por pares y a cada par que está siendo analizado se le calcula la longitud normalizada.

Código fuente 4 la versión final para el cálculo del alineamiento

```
# código de la versión final para el cálculo del alineamiento
longitud = calculeLongitud(results.collect())
rddnew = results.zipWithIndex().map(lambda key: (key[1],
key[0])).cache()
rddnew.saveAsTextFile(splitname +
'_Align_Global_Local_indexed_run_' + args.running)
rddcartesian = rddnew.cartesian(rddnew).filter(lambda (x, y):
(x[0], y[0]) in tuplelist)
rddcartesian.saveAsTextFile(splitname+'_Align_Global_Local_cart
ecian_run_'+ args.running)
```

```
rddalign=rddcartesian.map(lambda ((n1, x), (n2, y)):
alignGapMatGloLoc(x[0],y[0],x[1],y[1], longitud))
rddalign.saveAsTextFile(splitname +
'_Align_Global_Local_results_run_' + args.running)
```

Tabla 10: Comparación entre la primera versión y la final del código de los alineamientos

Archivo fasta: GH70	Alineamiento primera versión	Alineamiento versión final
UCLV	2313min	2080min

Como el tiempo de obtención del resultado en la ejecución de la versión final fue menor con respecto a la primera versión como se muestra en la Tabla 10, se opta por el código de la versión final para la extracción de las medidas de similitud basadas en alineamiento.

3. Tiempo de cálculo de *k*-mers en el clúster CBQ con el archivo GH70.fasta para distintos valores de *k* para obtener el tiempo y demostrar la superioridad del aumento del valor de *k* con respecto a la implementación anterior (Tabla 11).

Tabla 11: Cálculo de *k*-mers para distintos valores de *k*

Medidas de Similitud	GH70
k-mers para $k = 3$	219min
k-mers para $k = 5$	355min
k-mers para $k = 6$	366min

La Tabla 11 muestra los resultados de los tiempos de ejecución una vez aplicados los *k*-mers de orden 3,5 y 6. Se puede notar que según aumenta el orden, los valores de los tiempos de ejecución de los descriptores son mayores respectivamente, debido a que el cálculo del descriptor al aumentar el valor de *k* necesita más tiempo de ejecución, en contraste, se soluciona el limitante de espacio de memoria que presentó el cálculo de este descriptor en la versión anterior de la implementación.

4. Tiempo de cálculo de Pseudo-composición de aminoácidos en el clúster CBQ con el archivo GH70.fasta para distintos valores λ para obtener el tiempo y demostrar la superioridad del aumento del valor de λ con respecto a la implantación anterior (Tabla 12).

Tabla 12: Cálculo de Pseudo-composición de aminoácidos para distintos valores λ

Medidas de Similitud	GH70
Pseudo-composición de aminoácidos $\lambda = 3$	84min
Pseudo-composición de aminoácidos $\lambda = 5$	131min
Pseudo-composición de aminoácidos $\lambda = 7$	188min

La Tabla 12 muestra los resultados de los tiempos de ejecución una vez aplicados los Pseudo-composición de aminoácidos de orden 3,5 y 7, en donde se puede notar que según aumenta el orden el valor de los tiempos de ejecución de los descriptores son mayores respectivamente, debido a que el cálculo del descriptor al aumentar el valor de λ necesita más tiempo de ejecución, en contraste se soluciona la limitante de espacio de memoria una vez realizado el apareamiento de proteínas que presentó el cálculo de este descriptor en la implementación anterior.

3.4 Resultados de pruebas de software

En la implementación realizada en (Arteaga, 2018), los cálculos de los descriptores no tienen implementada la comparación por pares, lográndose con los experimentos el cumplimiento de este objetivo.

En el método relacionado con el descriptor de los *k*-mers, se le aumentó el rango de los valores admisibles por *k* con esto se “salta una nueva barrera” para el cálculo de los *k*.

Para eliminar la ausencia de la implementación de los métodos del cálculo basado en alineamiento, se agregó a esta nueva implementación tanto el cálculo del alineamiento global como el local. A través de este cálculo se extrae la puntación global y local, el por ciento de identidad entre los pares de cada proteína analizada el perfil físico-químico presente por cada región que se analizas en los pares.

En cuanto al límite del rango de los valores admisibles por lambda (λ) en el método relacionado con el descriptor de los Pseudo-composición de aminoácidos, se aumentó dicho rango, con esto se mejora el cálculo de la Pseudo-composición de aminoácidos.

3.5 Conclusiones parciales del capítulo

La Tabla 13 muestra el resumen de los resultados a partir de la experimentación:

Tabla 13: Resumen de los resultados a partir de la experimentación

Aspectos a comparar	**Implementación anterior**	**Implementación actual**
La comparación por pares	No implementada	Para todas las medidas de similitud
Similitudes basadas en alineamiento de pares de secuencias	No implementada	Implementada, conjuntamente con el cálculo del perfil físico-químico, por ciento de identidad y la longitud
La implementación de *k*-mers	Admite valores de $k < 4$	Validada para valores de K hasta 6
Pseudo-composición de aminoácidos valor de lambda	Admite valores de lambda < 4	Validada para valores de lambda hasta 7

CONCLUSIONES

- Las facilidades de Spark para acceder al HDFS, conjuntamente con la representación de los RDD estructurados de la forma llave-valor, permiten el manejo de archivos de proteomas completos u otros conjuntos de proteínas.
- La implementación del cálculo del descriptor *k*-mers, reduciendo el número de subsecuencias a buscar a solo las existentes en la secuencia permite el aumento del rango del parámetro *k* al reducir el espacio a utilizar.
- La implementación del cálculo de la comparación por pares, filtrando el número de comparaciones a realizar a sólo las que aparecen por encima de la diagonal principal de la matriz de similitud permite reducir considerablemente el número de comparaciones y, por tanto, aumentar el rango de los parámetros de las medidas al disminuir el tiempo y el espacio de cómputo.
- El cálculo de similitud por pares de proteínas basados en medidas de alineamiento y libres de alineamiento, permite extraer diversos valores de similitud de las proteínas.
- La ejecución los cálculos de similitud tanto en el clúster de Spark de la UCLV, como el de CBQ para 115921 pares de proteínas, permitió la validación de la implementación.

RECOMENDACIONES

1. Ampliar la experimentación para conjunto de datos de proteínas más grandes, valores de parámetros más altos y diferentes niveles de recursos.
2. Desarrollar la exportación de archivos con los cálculos en formato .csv incluyendo rasgos de clase de pares de secuencias.

BIBLIOGRAFÍA

ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, W. & LIPMAN, D. J. 1990. Basic local alignment search tool. *Journal Molecular Biology,* 215**,** 403-410.

BHASIN, M. & RAGHAVA, G. P. S. 2004. Classification of Nuclear Receptors Based on Amino Acid Composition and Dipeptide Composition. *J. Bio. Chem.,* 279**,** p.23262.

BODEN, M., SCHÖNEICH, M., HORWEGE, S., LINDNER, S., LEIMEISTER, C. & MORGENSTERN, B. 2013. Alignment-free genome comparison with feature frequency profiles (FFP) and optimal resolutions. *In:* T. BEIßBARTH, M. K., A. LEHA, B. MORGENSTERN, A.-K. SCHULTZ, S. WAACK, E. WINGENDER (ed.) *German Conference on Bioinformatics (GCB'13).* Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing.

CAO, D.-S., XU, Q.-S. & LIANG, Y.-Z. 2013. propy: a tool to generate various modes of Chou's PseAAC. *Bioinformatics,* 29**,** 960-962.

CHAPIN, G. A. 2013. *A Graphical and Numerical Approach for Functional Annotation and Phylogenetic Inference.* Doctor in Biology University of Porto.

CHEN, J., LIU, B. & HUANG, D. 2016. Protein Remote Homology Detection Based on an Ensemble Learning Approach. *BioMed Research International, Hindawi Publishing Corporation***,** 11 pages.

CHOU, K.-C. 2000. Prediction of protein subcellular locations by incorporating quasi-sequence-order effect. *Biochemical and biophysical research communications,* 278**,** 477-483.

CHOU, K.-C. 2001. Prediction of Protein Cellular Attributes Using Pseudo-Amino Acid Composition. *PROTEINS: Structure, Function, and Genetics,* 43**,** 246–255.

CHRISTIANINI, N. & HAHN, M. W. 2006. Introduction to computational genomics. A case studies approach. New York: Cambridge University Press.

DAVIES, M. N., SECKER, A., FREITAS, A. A., TIMMIS, J., CLARK, E. & FLOWER, D. R. 2008. Alignment-Independent Techniques for Protein Classification. *Current Proteomics,* 5**,** pp. 217-223(7).

DAYHOFF, M. O. 1978. *Survey of new data and computer methods of analysis,* Washington, D.C, National Biomedical Research Foundation, Georgetown University.

DEZA, E. 2006. *Dictionary of Distances*, Elsevier.

DUBCHAK, I., MUCHNIK, I., HOLBROOK, S. R. & KIM, S. H. 1995. Prediction of protein folding class using global description of amino acid sequence. *Proc Natl Acad Sci U S A,* 92, 8700-8704.

DUBCHAK, I., MUCHNIK, I., MAYOR, C., DRALYUK, I. & KIM, S. H. 1999. Recognition of a protein fold in the context of the SCOP classification. *Proteins: Structure, Function, and Bioinformatics,* 35**,** 401-407.

EDDY, S. R. 2004. Where did the BLOSUM62 alignment score matrix come from? *Nature Biotechnology,* 22**,** 1035-1036.

FERNÁNDEZ, A., RÍO, S. D., LÓPEZ, V., BAWAKID, A., JESUS, M. J. D., BENÍTEZ, J. M. & HERRERA, F. 2014. Big Data with Cloud Computing: an insight on the computing environment, MapReduce, and programming frameworks. *WIREs Data Mining and Knowledge Discovery,* 4**,** 380-409.

GALPERT, D. 2016. *Contribuciones al enfoque de comparación par a par en la detección de genes ortólogos.* Tesis para optar por el grado de Doctor en Ciencias Técnicas, Universidad Central "Marta Abreu" de las Villas.

GALPERT, D., FERNÁNDEZ, A., HERRERA, F., ANTUNES, A., MOLINA-RUIZ, R. & AGÜERO-CHAPIN, G. 2018. Surveying alignment-free features for Ortholog detection in related yeast proteomes by using supervised big data classifiers. *BMC Bioinformatics,* 19**,** 166.

HENIKOFF, S. 1992. Amino Acid Substitution Matrices from Protein Blocks. *Proceedings of the National Academy of Sciences of the United States of America,* 89**,** 10915-10919.

HOLDEN KARAU, A. K., PATRICK WENDELL, AND MATEI ZAHARIA february 2015. O'Reilly Learning Spark Lightning-Fast Data Analysis

IQBAL, M. J., FAYE, I., SAMIR, B. B. & SAID, A. M. 2014. Efficient Feature Selection and Classification of Protein Sequence Data in Bioinformatics. *The Scientific World Journal* [Online], 2014. [Accessed 2015].

ISAAC TRIGUERO, A. Y. S. 2015-2016. An introductory session of Apache Spark. *Intelligence Systems Course Ghent University***,** 25.

KARAU, H., KONWINSKI, A., WENDELL, P. & ZAHARIA, M. 2015. Learning Spark. *In:* BEAUGUREAU, A. S. A. M. (ed.) *Lightning-Fast Data Analysis.* United States of America: O'Reilly Media, Inc.

KUMAR, S. & FILIPSKI, A. 2007. Multiple sequence alignment: In pursuit of homologous DNA positions. *Genome Research,* 17**,** 127–135.

MAHMOUD SOUFI, A. A. A. E.-A., AND HESHAM A. HEFNY july 2018. A Survey on Big Data and Knowledge Acquisition Techniques. *IPASJ International Journal of Computer Science (IIJCS),* 6**,** 16.

MARCAIS, G. & KINGSFORD, C. 2011. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics,* 27**,** 764–770.

MELSTED, P. & PRITCHARD, J. K. 2011. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics,* 12**,** 1-7.

MIYAZAWA, S. & JERNIGAN, R. L. 1999. Self-Consistent Estimation of Inter-Residue Protein Contact Energies Based on an Equilibrium Mixture Approximation of Residues. *PROTEINS: Structure, Function, and Genetics,* 34**,** 49–68.

MOUNT, D. W. 2004a. Alignment of Pairs of Sequences. *Bioinformatics Sequence and Genome Analysis.* CSHL Press.

MOUNT, D. W. 2004b. Database Searching for Similar Sequences. *Bioinformatics Sequence and Genome Analysis.* CSHL Press.

NEEDLEMAN, S. B. & WUNSCH, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.,* 48**,** 443-453.

PEARSON, W. R. 1990. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods Enzymology,* 183**,** 63-98.

PEARSON, W. R. 2013. Selecting the Right Similarity-Scoring Matrix. *Current Protocols in Bioinformatics,* 43**,** 3.5.1-3.5.9.

PÉREZ, A. A., CAÑIZARES, D. G. & MOLINA-RUIZ, R. 2018. *Aplicación del modelo de programación Spark al cálculo de medidas de similitud para pares de genes.* Trabajo de Diploma para optar por la Licenciatura en Ciencia de la Computación, Universidad Central "Marta Abreu" de Las Villas.

RIZK, G., LAVENIER, D. & CHIKHI, R. 2013. DSK: k-mer counting with very low memory usage. *Bioinformatics,* 29**,** 652-653.

SALMAN SALLOUM, R. D., XIAOJUN CHEN, PATRICK XIAOGANG PENG AND JOSHUA ZHEXUE HUANG 2016. Big data analytics on Apache Spark. *Int J Data Sci Anal (2016)***,** 1:145-164.

SMITH, T. F. & WATERMAN, M. S. 1981. Identification of common molecular subsequences. *Journal Molecular Biology,* 147**,** 195-197.

VINGA, S., GOUVEIA-OLIVEIRA, R. & ALMEIDA, J. S. 2004. Comparative evaluation of word composition distances for the recognition of SCOP relationships. *Bioinformatics,* 20**,** 206–215.

ZIELEZINSKI, A., VINGA, S., ALMEIDA, J. & KARLOWSKI, W. M. 2017. Alignment-free sequence comparison: benefits, applications, and tools. *Genome Biology,* 18.

ANEXOS

Código fuente 5 el script “BigD_ProtDescritorsMod.py”,

```
from Bio import SeqIO
from itertools import product
import numpy as np
import pandas as pd
from propy import Autocorrelation as AC
from propy import PseudoAAC as PAAC
from propy import CTD as CTD
from propy import QuasiSequenceOrder as QSO
from pyspark import SparkConf, SparkContext
from pyspark.sql import *
from pyspark.sql.types import *
from scipy.stats.stats import pearsonr
from scipy.spatial import distance
import argparse
import csv
import os
import sys
import re

from pydoc import help
from Bio.SubsMat import MatrixInfo
from Bio import pairwise2

# calculo de los alinamientos RDD

def replace_contact_energy(secuencia):
    # Funcion para sustituir las energias de contacto de los
nucleotidos function[num_seq] = replace_contact_energy(seq)
    # Convierte la cadena de nucleotidos en una cadena numerica
```

```
    num_secuencia=[]
    # Define las energia de contacto = [('A', -0.03), ('R', 0.08),
('N',  0.12),  ('D',  0.18),  ('C',  -0.35),  ('Q',  0.14),  ('E',
0.22),('G', 0.05), ('H', -0.03), ('I', -0.32), ('L', -0.1), ('K',
0.32), ('M', -0.28), ('F', -0.35),('P', 0.17), ('S', 0.11), ('T',
0.03), ('W', -0.28), ('Y', -0.25), ('V', -0.27)]


    contacto=dict(zip('ARNDCQEGHILKMFPSTWYV',[-0.03,   0.08,   0.12,
0.18, -0.35, 0.14, 0.22, 0.05, -0.03, -0.32, -0.1, 0.32, -0.28, -
0.35, 0.17,0.11, 0.03, -0.28, -0.25, -0.27]))


    # Cantidad de valores de energia de contacto es el len de la
secuencia
    # % Reemplaza cada nucleotido por su respectiva energia de
contacto
    for caracter in secuencia:
        num_secuencia+=[contacto[caracter]]
    return(num_secuencia)



# Running mean/Moving average
def running_mean(l, N):
    sum = 0
    result = list( 0 for x in l)
    for i in range( 0, N ):
        sum = sum + l[i]
        result[i] = sum / (i+1)

    for i in range( N, len(l) ):
        sum = sum - l[i-N] + l[i]
        result[i] = sum / N

    return result



def calculate_profile(secuencia1,secuencia2,ventana):
    # Funcion que calcula la similitud entre dos subsecuencias basada
en el perfil fisico - quimico
```

```
    # de las secuencias
    #function[correlation] = calculate_profile(seq1, seq2, window)
    # Transforma la representacion de la secuencia por su energia de
contacto
    num_sec1=replace_contact_energy(secuencia1)
    num_sec2=replace_contact_energy(secuencia2)
    #Calcula el MoveringAverage para el tamano de ventana indicado
    ma_sec1 = running_mean(num_sec1,ventana)
    ma_sec2 = running_mean(num_sec2, ventana)
    #Calcula la correlacion entre los valores de MoveringAverage
    pearsoncoef, signification = pearsonr(ma_sec1,ma_sec2)

    if signification > 0.05:
         pearsoncoef = 0.0

    #print ("ventana", ventana,"pearsoncoef",pearsoncoef, 'ma_sec1',
ma_sec1, "ma_sec2", ma_sec2)
    return abs(pearsoncoef)

def      calculate_profile_feature(secuencia1=[],      secuencia2=[],
ventana=[]):
     # Funcion que calcula la similitud entre dos secuencias basada
en el perfil fisico - quimico de las secuencias
     #           function[profile,           region_count]           =
calculate_profile_feature(seq1, seq2, window)
     # Longitud total del alineamiento
    N=len(secuencia1)
    if N > len(secuencia2):
        N = len(secuencia2)
     # Cantidad de ventanas
    W=len(ventana)
      # Contador de las regiones a las que se le calcula el perfil
    region_count = [0,0,0]
      # Similitud de las secuencias
    profile = [0,0,0]
      # Iterador para recorrer el alineamiento
```

```
    itr = 0
      #      Inicializa      los      contenedores      temporales,
Key:[(correlacion,len_secuencia)]
    correlation = {3:[],5:[],7:[]}
      # Recorre cada posicion del alineamiento
    while itr < N:
            # Inicializa las subsecuencias
        itr1 = itr
        itr2 = itr1
                 # Recorre las secuencias mientras no existan gaps
en ninguna de las  secuencias
        while    itr    <    N    and    secuencia1[itr]!='-'    and
secuencia2[itr]!='-':
            itr2 = itr
            itr = itr + 1
                 # Toma la longitud de las subsecuencias
        N1 = itr2 - itr1 + 1
                 #  Recorre  cada  tamanno  de  ventana  pasado  como
parametro
        for wnd in range(W):
                 #  Verifica  si  se  puede  calcular  la  informacion
asociada al perfil fisico - quimico de las proteinas
           if N1 > ventana[wnd]:
              region_count[wnd] = region_count[wnd] + 1
              correlation[ventana[wnd]]+=
[(calculate_profile(secuencia1[itr1:itr2],     secuencia2[itr1:itr2],
ventana[wnd]),N1)]
           # Recorre las secuencias mientras exista un gap en una de
las  secuencias
        if itr == N:
             break
        while itr < N and secuencia1[itr] == '-' or secuencia2[itr]
== '-':
             itr = itr + 1
             if itr == N:
                 break
     # Recorre cada tamanno de ventana pasado como parametro
    for wnd in range(W):
```

```
     # Verifica si se le calculo el perfil al menos a una secuencia
        sum_len_sec=0
        if region_count[wnd] > 0:
            for itr in range(region_count[wnd]):
                (corr_sec,len_sec)=correlation[ventana[wnd]][itr]
                profile[wnd] = profile[wnd] +  corr_sec * len_sec
                sum_len_sec+=len_sec
        if sum_len_sec > 0:
            profile[wnd] = profile[wnd] / sum_len_sec


    return (profile, region_count)


def alignGapMatGloLoc( name1, name2, seq_a, seq_b,max, g_open=-10,
g_extend=-0.5):
    matrix = MatrixInfo.blosum62
    gap_open = g_open  # valor por defecto del algoritmo inicial -10
    gap_extend = g_extend  # valor del algoritmo inicial -0,5


    # Only using the first 60 aas for visualization reasons..


    alns = pairwise2.align.globalds(seq_a, seq_b, matrix, gap_open,
gap_extend)


    top_aln = alns[0]


    aln_seq_a, aln_seq_b, score, begin, end = top_aln
    matches = [(aln_seq_a[i] == aln_seq_b[i] and aln_seq_a[i] != '-'
and aln_seq_b[i] != '-') for i in xrange(end)]
    porciento = (100 * sum(matches)) / end
    (profile, region_count) = calculate_profile_feature(aln_seq_a,
aln_seq_b, [3, 5, 7])



#print('++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++')
    #print '(o)(o)'+name1 + '<++>' + name2
```

```
    alnsl = pairwise2.align.localxx(seq_a, seq_b)
    aaln_seq_al, aln_seq_bl, scorel, beginl, endl = alnsl[0]

    #calculo de la longitud incluido
    tmp = (abs(len(seq_a) - len(seq_b)) * 1.0) / max
    longitud = 1 - tmp

    return {'name1': name1, 'name2': name2, "scoreglobal": score,
"scorelocal": scorel, "porciento": porciento, "perfil3": profile[0],
            "perfil5":     profile[1],     "perfil7":     profile[2],
"region_count3": region_count[0],
            "region_count5":    region_count[1],    "region_count7":
region_count[2],'longitud':longitud}

def spliterData2(fileNameContents):
    allcontents = fileNameContents[1].split('\n')
    depures = [x.split('<+>') for x in allcontents if x != '']
    return depures

def calculeLongitud(seqentrada):
    min = sys.float_info.max
    max = sys.float_info.min
    for seq in seqentrada:
        if len(seq[1]) < min:
            min = len(seq[1])
        if len(seq[1]) > max:
            max = len(seq[1])
    print("    Longitud m\u00EDnima: ", min)
    print("    Longitud m\u00E1xima: ", max)
    #print("    Entradad: ",seqentrada)
    return max - min

def aparear(seqentrada):
    #min = 0
    #max = 0
    min = sys.float_info.max
```

```
    max = sys.float_info.min
    for seq in seqentrada:
        if len(seq[1]) < min:
            min = len(seq[1])
        if len(seq[1]) > max:
            max = len(seq[1])
    print("    Longitud m\u00EDnima: ", min)
    print("    Longitud m\u00E1xima: ", max)
    #print("    Entradad: ",seqentrada)
    max -= min
    data = []
    N = len(seqentrada)
    for i in range(N):
      for j in range(i+1,N):
         #data                                                    +=
[(seqentrada[i][0],seqentrada[j][0],seqentrada[i][1],seqentrada[j][1]
,max)]
         data.append({'name1':seqentrada[i][0],
'name2':seqentrada[j][0],'seq1':seqentrada[i][1],'seq2':seqentrada[j]
[1],'longitud': max})
         #data.append(seqentrada[i][0] + '<+>' + seqentrada[j][0] +
'<+>' + seqentrada[i][1] + '<+>' + seqentrada[j][1]+ '<+>' + max)
    print("OK")
    return data


def measuresDistsNumeric3(nam1,nam2,dic1,dic2):
    name1 = nam1
    name2 = nam2
    d1 = dic1
    d2 = dic2
    v1 = []
    v2 = []


    for key in sorted(d1.iterkeys()):
        if key not in d2.keys():
            v1.append(d1[key])
```

```
            v2.append(0)
        elif key not in d1.keys():
            v1.append(0)
            v2.append(d2[key])
        else:
            v1.append(d1[key])
            v2.append(d2[key])
    for key in sorted(d2.iterkeys()):
        if key not in d2.keys():
            v1.append(d1[key])
            v2.append(0)
        elif key not in d1.keys():
            v1.append(0)
            v2.append(d2[key])

    wk1 = list(map(convertFloat, v1[:]))
    wk2 = list(map(convertFloat, v2[:]))

    pearsoncoef, signification = pearsonr(wk1, wk2)
    if signification > 0.05:
        pearsoncoef = 0.0

    euclidean = distance.euclidean(wk1, wk2)
    canberra = distance.canberra(wk1, wk2)
    braycurtis = distance.braycurtis(wk1, wk2)
    chebyshev = distance.chebyshev(wk1, wk2)
    minkowski = distance.minkowski(wk1, wk2, p=2)
    kulsinski = distance.kulsinski(wk1, wk2)
    hamming = distance.hamming(wk1, wk2)
    cityblock = distance.cityblock(wk1, wk2)
    print
('measuresDistsNumeric2+++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++')
    return   {'name1':   name1,   'name2':   name2,   'pearsoncoef':
pearsoncoef, 'braycurtis': braycurtis,
```

```
            'euclidean':     euclidean,     'canberra':     canberra,
'minkowski': minkowski, 'kulsinski': kulsinski,
            'hamming': hamming, 'chebyshev': chebyshev, 'cityblock':
cityblock }


def getData(line):
    elements = line[1].split('\n')
    return elements


def getRowElement(line):
    elementDict = {}
    #sx                                                              =
re.compile("\((.*)\)").search(str(line)).group(1).split(',')
    markers = []
    #for x in sx:
    #    if '=' not in x:
    #        markers.append(x)
    #if markers.__len__() == 0:
    #    for x in sx:
    #        parts = x.split('=')
    #        elementDict[str(parts[0]).rstrip()] = parts[1].rstrip()
    #else:
    #    for x in markers:
    #        wherex = sx.index(x)
    #        sx[wherex-1] = sx[wherex-1] + ',' + sx[wherex]


    #    for x in sx:
    #        if '=' in x:
    #            parts = x.split('=')
    #                           elementDict[str(parts[0]).rstrip()]  =
parts[1].rstrip()



print('++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++>',li
ne[4:len(line)-1])
    data1=line.split("name1=")[1][2:len(line.split("name1=")[1])-2]
    elementDict['name1']= data1
```

```
    data2=line.split("name2=")[1][2:len(line.split("name2=")[1])-
(len(line.split("name1=")[1])+len("', name1="))]
    elementDict['name2']= data2
    data3=line.split("seq1=")[1][2:len(line.split("seq1=")[1])-
(len(line.split("name2=")[1])+len("', name2="))]
    elementDict['seq1']= data3.replace("u\'", "'")
    data4=line.split("seq2=")[1][2:len(line.split("seq2=")[1])-
(len(line.split("seq1=")[1])+len("', seq1="))]
    elementDict['seq2']= data4.replace("u\'", "'")



print('+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++>',
elementDict)


    if elementDict['name1'] == "name1":
        elementDict = {}


    return elementDict


# splitting data to build RDD


def convertFloat(x):
    return float(x)


def without_keys(d, keys):
    return {x: d[x] for x in d if not x in keys}


def spacedk_mers(k, l):
    result = []
    istring = k * 'X' + l * '.'
    c = [''.join(x) for x in product(istring, repeat=k + l)]
    for i in c:
        if i.count('X') == k:
            if (i.find('.') != 0) and (i.rfind('.') != k + l - 1):
                result.append(i)
    result = list(set(result))
```

```
    return result

# Kmers calculations

def spacedk_mers(k, l):
    result = []
    istring = k * 'X' + l * '.'
    c = [''.join(x) for x in product(istring, repeat=k + l)]
    for i in c:
        if i.count('X') == k and i.count('.') == l:
            if (i.find('.') != 0) and (i.rfind('.') != k + l - 1):
                result.append(i)
    result = list(set(result))
    return result

def setmuster(patronelement, musters):
    modmusters = []
    modmusters.append(patronelement)
    for muster in musters:
        cad = ''
        counter = 0
        for j in range(len(muster)):
            if muster[j] == 'X':
                cad = cad + patronelement[counter]
                counter += 1
            else:
                cad = cad + '.'
        modmusters.append(cad)
    return modmusters

def countKmers_spaced(listakms, wheretolook, id):
    kmersview = {}
    kmersview['name'] = id
    counter = 0
    for xlists in listakms:
```

```
        for kms in xlists:
            kmersview[kms] = re.findall(kms, wheretolook).__len__()
    return kmersview

def makeMuster(k, alphabet):
    """
     k: mustter size
     alphabet : mustter context
    :return:
    """
    return [''.join(x) for x in product(alphabet, repeat=k)]

def kmercount(id, seq, patrones):
    """ seq : Sequence
        patrones: mutter to look for
        id: id for sequence
        :return a dictionary patronDict
    """
    patronDict = {}
    patronDict['name'] = id
    for k in patrones:
        patronDict[k] = seq.count(k)
    return patronDict

def ownKmers(id, seq, size):
    go_on = False
    counter = 0
    #founded = {'name': id}
    founded = {}
    if len(seq) >= size:
        seqsearch = seq[:size]
        founded[seqsearch] = seq.count(seqsearch)
        go_on = True
    else:
        founded['check Sequence'] = 0
```

```
    while go_on:
        counter += 1
        seqsearch = seq[counter:counter + size]

        if len(seqsearch) == size:
            if seqsearch not in founded.keys():
                founded[seqsearch] = seq.count(seqsearch)
                if len(founded) - 1 >= 20**size:
                    go_on = False
        else:
            go_on = False

    return founded

def ownsSpacedKmers(id, seq, size, space, target):
    go_on = False
    # get pattern to search
    # founded = {'name': id}
    founded = {}
    counter = 0
    if len(seq) >= size + space:
        patronelement = seq[:size]
        seqsearch = setmuster(patronelement, target)
        for x in seqsearch:
            founded[x] = re.findall(x, seq).__len__()
        go_on = True
    else:
        founded['check Sequence'] = 0
    while go_on:
        counter += 1
        patronelement = seq[counter:counter + size]
        if len(patronelement) == size:
            seqsearch  = setmuster(patronelement, target)
            for x in seqsearch:
```

```
                if x not in founded.keys():
                    founded[x] = re.findall(x, seq).__len__()
                    if len(founded) - 1 >= 20**(size + space):
                        go_on = False
        else:
            go_on = False
    return founded

def aligmentKmers(x, y):
    #aligment of the founded kmers
    name1, element1 = x[0]
    name2, element2 = y[0]
    xelement = without_keys(element1, 'name')
    yelement = without_keys(element2, 'name')
    inX = set(xelement.keys())
    inY = set(yelement.keys())
    onlyX = list(inX.difference(inY))
    onlyY = list(inY.difference(inX))
    commonKmers = list(inX.intersection(inY))
    fx = []
    fy = []
    for i in commonKmers:
        fx.append(xelement[i])
        fy.append(yelement[i])
    for i in onlyX:
        fx.append(xelement[i])
        fy.append(0)
    for i in onlyY:
        fy.append(yelement[i])
        fx.append(0)
    return (name1, fx, name2, fy)

def newcounter(data, patternx ):
    key1, seq, valdictx   = data
    valdictx[patternx] = seq.count(patternx)
```

```
    return (key1, seq, valdictx)

def newkmers(id, s, alphabet, size):
    g = (''.join(i) for i in product(alphabet, repeat=size))
    outputx = {'name': id}
    for x in g:
        outputx[x] = s.count(x)
    return outputx

def putOrder(element):
    newlist = []
    nameX = element['name']
    newdict = without_keys(element, 'name')
    for x in sorted(newdict.keys()):
        newlist.append(element[x])
    return nameX, newlist

def countKmer(seqdict, kpattern):
    wherelook = seqdict
    wherelook[kpattern] =seqdict['seq'].count(kpattern)
    return wherelook

def measuresDists(x, y):
    name1 = x['name']
    name2 = y['name']
    d1 = without_keys(x, ['name'])
    d2 = without_keys(y, ['name'])
    u = []
    v = []
    for key in sorted(d1.iterkeys()):
        u.append(d1[key])
        v.append(d2[key])
    wk1 = list(map(convertFloat, u[:]))
    wk2 = list(map(convertFloat, v[:]))
    pearsoncoef, signification = pearsonr(wk1, wk2)
```

```
    if signification > 0.05:
        pearsoncoef = 0.0

    euclidean = distance.euclidean(wk1, wk2)
    canberra = distance.canberra(wk1, wk2)
    braycurtis = distance.braycurtis(wk1, wk2)
    chebyshev = distance.chebyshev(wk1, wk2)
    minkowski = distance.minkowski(wk1, wk2, p=2)
    kulsinski = distance.kulsinski(wk1, wk2)
    hamming = distance.hamming(wk1, wk2)
    cityblock = distance.cityblock(wk1, wk2)

    return {'name1': name1, 'name2': name2, 'pearsoncoef':
pearsoncoef, 'braycurtis': braycurtis,
            'euclidean': euclidean, 'canberra': canberra,
'minkowski': minkowski, 'kulsinski': kulsinski,
            'hamming': hamming, 'chebyshev': chebyshev, 'cityblock':
cityblock }

def measuresDistsNumeric(x, y):
    name1 = x[0]
    name2 = y[0]
    u = x[1][:]
    v = y[1][:]
    wk1 = list(map(convertFloat, u[:]))
    wk2 = list(map(convertFloat, v[:]))
    pearsoncoef, signification = pearsonr(wk1, wk2)
    if signification > 0.05:
        pearsoncoef = 0.0
    euclidean = distance.euclidean(wk1, wk2)
    canberra = distance.canberra(wk1, wk2)
    braycurtis = distance.braycurtis(wk1, wk2)
    chebyshev = distance.chebyshev(wk1, wk2)
    minkowski = distance.minkowski(wk1, wk2, p=2)
    kulsinski = distance.kulsinski(wk1, wk2)
    hamming = distance.hamming(wk1, wk2)
```

```
    cityblock = distance.cityblock(wk1, wk2)
    return {'name1': name1, 'name2': name2, 'pearsoncoef': pearsoncoef, 'braycurtis': braycurtis,
            'euclidean': euclidean, 'canberra': canberra, 'minkowski': minkowski, 'kulsinski': kulsinski,
            'hamming': hamming, 'chebyshev': chebyshev, 'cityblock': cityblock }


def spliterData(fileNameContents):
    allcontents = fileNameContents[1].split('>')
    depures = [x.rstrip().replace('\n', '<+>') for x in allcontents if x != '']
    return depures


# Autocorrelations
# Autocorrelation modules


def calculeNormMoreauBroto(id, proteinSequence):
    temp = AC.CalculateNormalizedMoreauBrotoAuto(proteinSequence, [AC._ResidueASA], ['ResidueASA'])
    result = temp['ResidueASA']
    result['name'] = id
    return result



def calculeMoran(id, proteinSequence):
    temp = AC.CalculateMoranAuto(str(proteinSequence), [AC._ResidueASA], ['ResidueASA'])
    result = temp['ResidueASA']
    result['name'] = id
    return result


def calculeGearyAuto(id, proteinSequence):
    temp = AC.CalculateGearyAuto(str(proteinSequence), [AC._ResidueASA], ['ResidueASA'])
    result = temp['ResidueASA']
    result['name'] = id
```

```
    return result


def calculateAutoTotal(id, proteinSequence):
    temp = AC.CalculateAutoTotal(str(proteinSequence))
    # result = temp['ResidueASA']
    temp['name'] = id
    return temp


# SOCN: sequence order coupling numbers (depend on the choice of maxlag, the default is 60)
# QSO: quasi-sequence order descriptors (depend on the choice of maxlag, the default is 100)


def getSequenceOrderCouplingNumberTotal(id, proteinSequence, maxlag=30):
    temp = QSO.GetSequenceOrderCouplingNumberTotal(str(proteinSequence), maxlag)
    temp['name'] = id
    return temp


def getQuasiSequenceOrder(id, proteinSequence, maxlag=30, weight=0.1):
    temp = QSO.GetQuasiSequenceOrder(str(proteinSequence), maxlag, weight)
    temp['name'] = id
    return temp


# modules for CTD
# CTD: Composition, Transition, Distribution descriptors (CTD) (21+21+105=147)


def calculateC(id, proteinSequence):
    temp = CTD.CalculateC(str(proteinSequence))
    temp['name'] = id
    return temp


def calculateT(id, proteinSequence):
```

```
    temp = CTD.CalculateT(str(proteinSequence))
    temp['name'] = id
    return temp

def calculateD(id, proteinSequence):
    temp = CTD.CalculateD(str(proteinSequence))
    temp['name'] = id
    return temp

def calculateCTD(id, proteinSequence):
    temp = CTD.CalculateCTD(str(proteinSequence))
    temp['name'] = id
    return temp

# Pseudo Amino acid compositionPago 1er semenstre 2018
def calculePseudoAcc(id, proteinSequence, lamdasize):
    # return a dictionary with PseudoAAC
    if len(proteinSequence) > lamdasize:
        temp  =  PAAC._GetPseudoAAC(str(proteinSequence),  lamdasize,
weight=0.05)
        temp['name'] = id
    else:
        temp = {}
        c = ['pac{}'.format(i) for i in range(21 + lamdasize)]
        d = c[1:]
        temp['name'] = id
        for x in d:
            temp[x] = 0.0

    return temp

# Arguments for ProteinDescriptors.pyargs.fasta
parser  =  argparse.ArgumentParser(description='Script  for  Protein
Descriptors calculations.')
parser.add_argument('-lm', '--LocalMode', action='store_true', help='
Cluster or Local Mode')
```

```
parser.add_argument('-kw', type=int, help='K-mers size ( from 1 to n)')
parser.add_argument('-k', type=int, help='K-mers size ( from 1 to 7)')
parser.add_argument('-s', type=int, help='Space size ( from 1 to 4)')
parser.add_argument('-sp','--sizesplit', type=int, help='Splitting size')
parser.add_argument('-f', '--fasta', required=True, help='input fasta file name')
parser.add_argument('-l', '--lambdax', type=int, help='Pseudo amino acid composition, (input Lamda)')
parser.add_argument('-an', '--NMB', action='store_true', help='Autocorrelation for Norm Moreau Broto')
parser.add_argument('-fs', '--filter', action='store_true', help='filter for small selection')
parser.add_argument('-sf', '--savefile', action='store_true', help='saving pairing tuples')
parser.add_argument('-am', '--Moran', action='store_true', help='Autocorrelation for Moran')
parser.add_argument('-ag', '--Geary', action='store_true', help='Autocorrelation for Geary')
parser.add_argument('-at', '--Total', action='store_true', help='Autocorrelation total')
parser.add_argument('-cc', '--CTD_C', action='store_true', help='Composition descriptors (CTD_C)')
parser.add_argument('-ct', '--CTD_T', action='store_true', help='Transition descriptors (CTD_T)')
parser.add_argument('-cd', '--CTD_D', action='store_true', help='Distribution descriptors (CTD_D)')
parser.add_argument('-ctd', '--CTD', action='store_true', help='Composition, Transition, Distribution descriptors (CTD)')
parser.add_argument('-d', '--dst', action='store_true', help='Avoid to calculate Distances')
parser.add_argument('-qcn', '--QSOCN', action='store_true', help='Quasi sequence order coupling numbers ')
parser.add_argument('-qso', '--QSO', action='store_true', help='Quasi-sequence order descriptors')
parser.add_argument('-m', '--maxlag', type=int, help='Maxlag (default 30)')
parser.add_argument('-w', '--weight', type=float, help='Weight (default 0.1)')
```

```
parser.add_argument('-z',   '--running',   type=str,   help='running
number')
parser.add_argument('-al',      '--align',      action='store_true',
help='calculo de alinamiento por parte')



if __name__ == "__main__":
    k = 0  # Kmers
    s = 0  # Spaced
    l = 0  # lamda for Pseudo amino acid composition
    kw = 0 # Kmers owns
    alphabet = 'ACDEFGHIKLMNPQRSTVWY'  # Protein alphabet
    ejecutar = False
    calcule = False
    calcule2 = False
    calcule3 = False
    calcdist = True
    sizesplit = 10

    args = parser.parse_args()
    if args.k:
        k = int(args.k)
        calcule = True

    if args.kw:
        kw = int(args.kw)
        calcule = True

    if args.dst:
        calcdist = False

    if args.s:
        s = int(args.s)
        calcule = True
```

```
    if args.lambdax:
        l = int(args.lambdax)
        calcule2 = True

    if args.maxlag:
        maxlag = int(args.maxlag)
    else:
        maxlag = 30

    if args.weight:
        weight = float(args.weight)
    else:
        weight = 0.1

    # input checking

    if k > 7 or k < 0:
        parser.print_help()
        print('Error input for kmers ... k_{}'.format(k))
        sys.exit()

    if s < 0 or s > 4:
        parser.print_help()
        print ('Error input for spaced kmers......s_{}'.format(s))
        sys.exit()

    if l < 0:
        parser.print_help()
        print('Error input for Lambda ......l_{}'.format(l))
        sys.exit()

    logFile = args.fasta
    splitname = logFile.split('.')[0]
    # Initializing SparkContext(local and cluster)
    if args.LocalMode:
```

```
        sc = SparkContext("local[*]", "ProteinMolecularDescriptors")
    else:
        sc = SparkContext("yarn", "ProteinMolecularDescriptors")

    # Declarations of Dataframe
    sqlContext = SQLContext(sc)

    #sc = SparkContext("yarn", "ProteinMolecularDescriptors")

    # load text file as RDD

    fullfasta = sc.wholeTextFiles(logFile).flatMap(spliterData)
    # pair RDD
    fasta    =    fullfasta.map(lambda    x:    (x.split('<+>')[0],
x.split('<+>')[1]))

    print('{} secuencias cargadas'.format(fasta.count()))
    # filter applied to get small sequences

    if args.filter:
        results = fasta.filter(lambda keyValues: len(keyValues[1]) >
3 and len(keyValues[1]) < 100).cache()
        checkseq = results.map(lambda key: len(key[1])).collect()
        np.savetxt(splitname  +  '_seqdata_run_'  +  args.running  +
'.txt', checkseq, delimiter=',')
    else:
        results = fasta.filter(lambda keyValues: len(keyValues[1]) >
10).cache()

    tuplelist = []
    for i in range(results.count()):
        for j in range(i + 1, results.count()):
            tuplelist.append((i, j))

    #only to save tuple list
    #rddxt = sc.parallelize(tuplelist)
```

```
    #rddxt.saveAsTextFile(splitname     +     '_tuplelist_run_'     +
args.running)



    print('{} Sequences to calculate ...'.format(results.count()))


    if args.savefile:
        testx = []
        rddnew  =  results.zipWithIndex().map(lambda  key:  (key[1],
key[0])).cache()
        nElements = rddnew.count()
        datax = {'name1': 'name1', 'name2': 'name2', 'seq1': 'seq1',
'seq2': 'seq2'}
        predf = pd.DataFrame(datax, columns=datax.keys(), index=[0])
        print(predf)
        counterx = 0
        for i in range(nElements-1):

            rddstore = sqlContext.createDataFrame(predf)
            point1 = rddnew.lookup(i)
            testx = []
            for j in range(i+1, nElements):
                counterx += 1
                point2 = rddnew.lookup(j)
                datax['name1'] = point1[0][0]
                datax['name2'] = point2[0][0]
                datax['seq1'] = point1[0][1]
                datax['seq2'] = point2[0][1]

                rddtoadd                                          =
sqlContext.createDataFrame(pd.DataFrame(datax,  columns=datax.keys(),
index=[0]))
                rddstore = rddstore.union(rddtoadd).coalesce(8)
            rddtosave = rddstore.rdd
            rddtosave.saveAsTextFile(splitname  +  '_pairs_comparing_'
+ str(i) + '_run_' + args.running)
```

```
        print("The data was processed. Done.")


    if args.align:
            #rddstore = sc.parallelize(aparear(results.collect()))
            #rddstore.saveAsTextFile(splitname                          +
'_pairs_comparing_run_' + args.running)
            #rddalign              =rddstore.map(lambda              x:
alignGapMatGloLoc(x['name1'],x['name2'],x['seq1'],x['seq2'],x['longit
ud']))
            #rddalign.saveAsTextFile(splitname                          +
'_Align_Global_Local_run_' + args.running)
            #rddalign.toDF().write.format("csv").save(splitname       +
'_Align_Global_Local_CSV_run_' + args.running)
        longitud = calculeLongitud(results.collect())
        rddnew  =  results.zipWithIndex().map(lambda  key:  (key[1],
key[0])).cache()
        rddnew.saveAsTextFile(splitname                                 +
'_Align_Global_Local_indexed_run_' + args.running)
        rddcartesian = rddnew.cartesian(rddnew).filter(lambda (x, y):
(x[0], y[0]) in tuplelist)
        rddcartesian.saveAsTextFile(splitname
+'_Align_Global_Local_cartecian_run_' + args.running)
        rddalign  =  rddcartesian.map(lambda  ((n1,  x),  (n2,  y)):
alignGapMatGloLoc(x[0],y[0],x[1],y[1], longitud))
        rddalign.saveAsTextFile(splitname                               +
'_Align_Global_Local_results_run_' + args.running)



    if calcule:
        if k and s:
            patternx = spacedk_mers(k, s)
            rddnew = results.zipWithIndex().map(lambda key: (key[1],
key[0])).cache()
            rddnew.saveAsTextFile(
                splitname  +  '_k_'  +  str(k)  +  '_s_'  +  str(s)  +
'_Kmers_own_indexed_run_' + args.running)
            rddcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            rddcartesian.saveAsTextFile(
```

```
                splitname + '_k_' + str(k) + '_s_' + str(s) +
'_Kmers_own_cartecian_run_' + args.running)
            comparing = rddcartesian.map(lambda ((n1, x), (n2, y)):
measuresDistsNumeric3(x[0], y[0],

ownsSpacedKmers(x[0], x[1], k,

s, patternx),

ownsSpacedKmers(y[0], y[1], k,

s, patternx)))
            comparing.saveAsTextFile(
                splitname + '_k_' + str(k) + '_s_' + str(s) +
'_Kmers_owns_compare_results_run_' + args.running)


        elif k != 0:


            rddnew = results.zipWithIndex().map(lambda key: (key[1],
key[0])).cache()
            rddnew.saveAsTextFile(splitname + '_k_' + str(k) +
'_Kmers_own_indexed_run_' + args.running)
            rddcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            rddcartesian.saveAsTextFile(splitname + '_k_' + str(k) +
'_Kmers_own_cartecian_run_' + args.running)
            comparing = rddcartesian.map(
                lambda ((n1, x), (n2, y)):
measuresDistsNumeric3(x[0], y[0], ownKmers(x[0], x[1], k),

ownKmers(y[0], y[1], k)))
            comparing.saveAsTextFile(splitname + '_k_' + str(k) +
'_Kmers_owns_compare_results_run_' + args.running)


    if calcule2:
        if l > 0:
            psedoAAC = results.map(lambda key:
calculePseudoAcc(key[0], key[1], l))
            psedoAAC.saveAsTextFile(splitname + '_PseudoAC_lambda_' +
str(l) + '_results_run_' + str(args.running))
```

```
            rddpure = psedoAAC.map(lambda key: (putOrder(key)))
            rddnew = rddpure.zipWithIndex().map(lambda key: (key[1],
key[0])).cache()
            rddpure.saveAsTextFile(splitname + '_PseudoAC_lambda_' +
str(l) + '_numberonly__run_' + args.running)


            if calcdist:
                newcartesian = rddnew.cartesian(rddnew).filter(lambda
(x, y): (x[0], y[0]) in tuplelist)
                comparing = newcartesian.map(lambda (x, y):
measuresDistsNumeric(x[1], y[1]))
                comparing.saveAsTextFile(splitname +
'_PseudoAC_lambda_' + str(l) + '_compare_results_run_' +
str(args.running))


    if args.NMB:
        normMoreauBroto = results.map(lambda key:
calculeNormMoreauBroto(key[0], key[1]))
        normMoreauBroto.saveAsTextFile(splitname +
'_Norm_MoreauBroto_results_run_' + args.running)
        rddnew = normMoreauBroto.zipWithIndex().map(lambda key:
(key[1], key[0]))
        if calcdist:
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing = newcartesian.map(lambda (x, y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname +
'_Norm_MoreauBroto_compare_results_run_' + args.running)


    if args.Moran:
        moranAuto = results.map(lambda key: calculeMoran(key[0],
key[1]))
        moranAuto.saveAsTextFile(splitname +
'_MoranAuto_results_run_' + args.running)
        if calcdist:
            rddnew = moranAuto.zipWithIndex().map(lambda key:
(key[1], key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
```

```
            comparing    =    newcartesian.map(lambda    (x,    y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname                      +
'_MoranAuto_compare_results_run_' + args.running)


    if args.Geary:
        gearyAuto = results.map(lambda key: calculeGearyAuto(key[0],
key[1]))
        gearyAuto.saveAsTextFile(splitname                          +
'_GearyAuto_results_run_' + args.running)
        if calcdist:
            rddnew   =   gearyAuto.zipWithIndex().map(lambda   key:
(key[1], key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing    =    newcartesian.map(lambda    (x,    y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname                      +
'_GearyAuto_compare_results_run_' + args.running)


    if args.Total:
        xtotalAuto         =         results.map(lambda         key:
calculateAutoTotal(key[0], key[1]))
        xtotalAuto.saveAsTextFile(splitname                         +
'_TotalAuto_results_run_' + args.running)
        if calcdist:
            rddnew   =   xtotalAuto.zipWithIndex().map(lambda   key:
(key[1], key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing    =    newcartesian.map(lambda    (x,    y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname                      +
'_TotalAuto_compare_results_run_' + args.running)


        # Composition, transition, distribution descriptors (CTD)


    if args.CTD_C:
        xctdc = results.map(lambda key: calculateC(key[0], key[1]))
```

```
        xctdc.saveAsTextFile(splitname   +   '_CTD_C_results_run_'   +
args.running)
        if calcdist:
            rddnew  =  xctdc.zipWithIndex().map(lambda  key:  (key[1],
key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing    =    newcartesian.map(lambda    (x,    y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname                      +
'_CTD_C_compare_results_run_' + args.running)


    if args.CTD_T:
        xctdt = results.map(lambda key: calculateT(key[0], key[1]))
        xctdt.saveAsTextFile(splitname   +   '_CTD_T_results_run_'   +
args.running)
        if calcdist:
            rddnew  =  xctdt.zipWithIndex().map(lambda  key:  (key[1],
key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing    =    newcartesian.map(lambda    (x,    y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname                      +
'_CTD_T_compare_results_run_' + args.running)


    if args.CTD_D:
        xctdd = results.map(lambda key: calculateD(key[0], key[1]))
        xctdd.saveAsTextFile(splitname   +   '_CTD_D_results_run_'   +
args.running)
        if calcdist:
            rddnew  =  xctdd.zipWithIndex().map(lambda  key:  (key[1],
key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing    =    newcartesian.map(lambda    (x,    y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname                      +
'_CTD_D_compare_results_run_' + args.running)
```

```
    if args.CTD:
        xctd = results.map(lambda key: calculateCTD(key[0], key[1]))
        xctd.saveAsTextFile(splitname + '_CTD_results_run_' +
args.running)
        if calcdist:
            rddnew = xctd.zipWithIndex().map(lambda key: (key[1],
key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing = newcartesian.map(lambda (x, y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname +
'_CTD_compare_results_run_' + args.running)


    if args.QSO:
        qso = results.map(lambda key: getQuasiSequenceOrder(key[0],
key[1], maxlag=maxlag, weight=weight))
        qso.saveAsTextFile(
            splitname + '_QSO_maxlag_' + str(maxlag) + 'weight_' +
str(weight) + '_results_run_' + args.running)
        if calcdist:
            rddnew = qso.zipWithIndex().map(lambda key: (key[1],
key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing = newcartesian.map(lambda (x, y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname + '_QSO_maxlag_' +
str(maxlag) + 'weight_' + str(weight) + '_compare_results_run_' +
args.running)



    if args.QSOCN:
        # QSO Calculationsm, distances and save to directory
        qsoresults = results.map(lambda key:
getSequenceOrderCouplingNumberTotal(key[1], key[0]))
        qsoresults.saveAsTextFile(splitname + '_QSOCN_results_run_' +
args.running)
        if calcdist:
```

```
            rddnew    =    qsoresults.zipWithIndex().map(lambda    key:
(key[1], key[0]))
            newcartesian = rddnew.cartesian(rddnew).filter(lambda (x,
y): (x[0], y[0]) in tuplelist)
            comparing    =    newcartesian.map(lambda    (x,    y):
measuresDists(x[1], y[1]))
            comparing.saveAsTextFile(splitname                      +
'_QSOCN_compare_results_run_' + args.running)


    sc.stop()
```