UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Taller de Sistemas Ciber Físicos

Autor:
Alexis ARRIOLA GARCIA

20 de diciembre de 2019

Indice

1 Introducción

1.1. Problema Planteado

El objetivo principal de este proyecto era poder realizar un análisis de datos de la red Ceibal, para poder encontrar los momentos de mayor demanda y saturación de la red, tratando de identificar las características principales de estas consultas. Para realizar esto, en un principio, se comenzó a trabajar con los datos en menor volumen, manipulando información correspondientes a un solo día de uso de la red.

La idea de trabajo con estos datos era poder realizar primeramente una reducción de dimensionalidad del grupo de datos que se tenía. Para esta primera parte se tenía como objetivo comparar el resultados de dos técnicas distintas, una lineal que era aplicando PCA[1] y otra no lineal aplicando T-SNE[2].

En una siguiente etapa, se tenía planificado tratar de agrupar los datos. Para esta etapa, al igual que la anterior, se tenía como idea principal tratar de aplicar dos técnicas de clustering y poder luego comparar los resultados obtenidos. Las técnicas que se tenía pensado usar eran K-MEANS[3] y algún tipo de clustering jerárquico.

Otro punto que se quería estudiar, era poder distinguir que características de los datos eran las más "importantes" y/o relevantes. Y de esta manera poder hacer una abstracción de estas características, permitiendo trabajar con un universos de los datos más acotado y disminuyendo el "ruido" que podrían agregar estas otras características que no aportaban una información relevante a la realidad.

Para poder llevar a cabo este último punto, se tenía como objetivo aplicar una técnica de clasificación como random forest[4] pero orientándola en un ambiente de aprendizaje no supervisado. Este tipo de aplicación de este método puede apreciarse con mayor profundidad en el paper de la referencia[5].

1.2. Procedimientos

Los datos estaban provistos en formato CSV, en donde se tiene un aproximado de seis archivos CSV, correspondientes a las consultas realizadas al sistema DNS, cada diez minutos de funcionamiento del sistema.

El primer desafío que se tuvo fue el de poder manipular este volumen de datos. En una primera instancia, se procedió a realizar, mediante un script python, la unión de estos CSV en un único archivo de extensión CSV.

Para el procesamiento de la matriz de datos se decidió trabajar con la librería Pandas[6] de python. En donde se levantaba este único archivo CSV y se procesaban los

datos para crear una nueva matriz, que sería la utilizada como entrada para los algoritmos de learning. Esta nueva matriz también era almacenada posteriormente en formato CSV. En este punto ya se encontró el inconveniente de que el almacenamiento en este formato era impráctico debido que a que los archivos ocupan demasiado espacio en disco y a su vez la carga del archivo en memoria era muy lenta.

Como alternativa al uso del formato CSV, se buco otro formato que permita una rápida carga en memoria así como también que utilice alguna especie de compresión de los datos, a la hora del almacenamiento, para disminuir el espacio en disco. El nuevo formato por el que se opto fue PARQUET[7], el cual permitía una rápida carga en memoria así como también una muy notable disminución del espacio de almacenamiento en disco.

Como paso siguiente, se procedió a trabajar con los algoritmos de learning, para esto se utilizó la librería Scikit-Learn[8] de python. Como se mencionó anteriormente, como primer paso se tenía planeado hacer una reducción de la dimensionalidad del problema, utilizando PCA y T-SNE, teniendo como finalidad no solo la reducción de dimensiones sino también la comparación de los resultados. Al utilizar PCA no hubo mayores inconvenientes pero al usar T-SNE se presentó el problema de que el algoritmo no finalizaba. El problema que se presentaba era que el algoritmo no es escalable para grandes volúmenes de datos. Se trato de ejecutar otras versión del algoritmo que en un principio aceleran el proceso, pero no se tuvo gran éxito con esto por lo que se decidió proseguir con los algoritmos de clustering.

La idea en este punto era poder ejecutar algún tipo de clustering jerárquico y además ejecutar K-MEANS, para luego poder comparar los resultado. Lamentablemente, ninguno de estos dos tipos de clustering se pudo realizar ya que se presento el mismo problema que con T-SNE anteriormente, los algoritmos no eran escalables.

Debido a estos inconvenientes, se llegó a la conclusión de que la plataforma elegida hasta el momento para trabajar, no era la correcta así también como las librerías utilizadas.

En resumen, este primer camino por el cual se opto tomar tuvo el inconveniente de que resultó no ser escalable. Por lo que si bien, la utilización de dichas librerías es la más recomendable y utilizada para el tipo de trabajo que se quería realizar, cuando se trata de grandes volúmenes de datos, la implementación de algoritmos y el uso de recursos de estas librerías no es la más eficiente. Como resultado de este problema, el consumo de memoria era excesivo provocando en muchos casos el mal funcionamiento y limitando el uso del ordenador.

Otro punto a tener en cuenta, es que hasta el momento solo se trabajaba con los datos pertenecientes a un día de la realidad. Inevitablemente cuando se procediera a procesar una semana, un mes o un periodo de tiempo grande de datos, el problema sería mucho mayor e imposible de realizar con las herramientas que se estaban utilizando.

Es en este punto que se decide cambiar el enfoque del proyecto y no solo afrontarlo como un problema de análisis de datos sino que ahora se presentaba el problema de manejar estos grandes volúmenes de información. Por lo que se necesitaba una plataforma capaz de soportar grandes volúmenes de datos y a su vez poder aplicar

algoritmos de minería de datos de forma eficiente para poder obtener resultados en un tiempo considerablemente pequeño.

Por estas razones es que se decidió por la utilización de Hortonworks Data Plataform, la cual tiene todo un ecosistema de herramientas para trabajar con volúmenes de datos a escala Big Data y que además posee una herramienta como Spark (ver capítulo 3 para obtener más información referente a la plataforma y sus componentes) que cuenta con una librería (ML[9]) que hace posible aplicar ciertos algoritmos de learning de forma eficiente sobre estos grandes volúmenes de datos.

Como punto en contra que se tuvo de este cambio de plataforma, es que la librería antes mencionada no cuenta con ciertos algoritmos que se tenía pensado utilizar, como T-SNE y clustering jerárquico, no obstante no hubo ningún problema en la ejecución de los otros algoritmos.

1.3. Recopilación de Datos

Para este estudió se utilizaron datos, pertenecientes a las consultas DNS, obtenidas de los distintos puntos de acceso de la red. Las consultas DNS quedan registradas en el programa UMBRELLA[10] perteneciente a la empresa CISCO[11]

Para cada consulta que queda registrada, se almacenan ciertos datos de interés:

- **Timestamp** - Indica el momento en que se realizo la consulta DNS.
- **Most Granular Identity** - Si se tiene en cuenta que una consulta puede pasar por más de un dispositivo hasta llegar al DNS, este campo indica el identificador del primer dispositivo que recibió la consulta.
- **Identities** - Muestra el identificador de todos los dispositivos por los cuales la consulta fue pasada.
- **Action** - Campo que indica si la consulta DNS fue permitida o bloqueada.
- **Query Type** - Campo que indica si la consulta DNS fue permitida o rechazada.
- **Response Code** - Campo que indica el código de retorno de DNS para la consulta.
- **Domain** - El dominio que se solicitó.
- **Categories** - Las categorías de seguridad o contenido con las que coincide el destino.

Este software otorga a cada consulta una o varias categorías a la cual pertenece, dependiendo de la url consultada. Existen aproximadamente 150 categorías diferentes, a modo de ejemplo se detallan las categorías que más se repiten para la ventada de tiempo utilizada.

- **Software/Technology** - Sitios sobre informática, hardware y tecnología, incluidas noticias, información, código e información del proveedor.
- **Search Engines** - Sitios que ofrecen listados de resultados basados en palabras clave.

- **Educational Institutions** - Sitios para escuelas que cubren todos los niveles y tipos de edad.
- **Infrastructure** - Infraestructura de entrega de contenido y contenido generado dinámicamente, sitios web que no pueden clasificarse más específicamente porque son seguros o de otra manera difíciles de clasificar.
- **Social Networking** - Sitios que promueven la interacción y la creación de redes entre las personas.
- **Video Sharing** - Sitios para compartir contenido de video.
- **Chat** - Sitios donde puede chatear en tiempo real con grupos de personas. Incluye IRC y sitios de video chat.
- **Instant Messaging** - Sitios que ofrecen acceso o software para comunicarse en tiempo real con otras personas.
- **Business Services** - Sitios para corporaciones y negocios de todos los tamaños, especialmente sitios web de empresas.

2 Algoritmos

2.1. Principal Component Analysis (PCA)

La idea central del análisis de componentes principales (Principal Component Analysis en ingles) es reducir la dimensionalidad de un conjunto de datos que contiene una gran cantidad de variables interrelacionadas, al tiempo que conserva la mayor cantidad posible de la variabilidad presente en el conjunto de datos. Esto se logra mediante la transformación a un nuevo conjunto de variables, los componentes principales (PC), que no están correlacionados, y que están ordenados para que los primeros retengan la mayor parte de la variación presente en todas las variables originales[12][1].

En otras palabras "conservar la mayor variabilidad posible" se transforma en encontrar nuevas variables que sean funciones lineales de las del conjunto de datos original, que maximicen sucesivamente la varianza y que no estén correlacionadas entre sí. Encontrar tales nuevas variables (los componentes principales) se reduce a resolver un problema de valor propio/vector propio (eigenvalues y eigenvectors en ingles).

2.1.1. Eigenvectors

Los eigenvectors son un caso particular de multiplicación entre una matriz y un vector. Los eigenvectors de una matriz son todos aquellos vectores que, al multiplicarlos por dicha matriz, resultan en el mismo vector o en un múltiplo entero del mismo. Los eigenvectors tienen una serie de propiedades matemáticas específicas:

- Los eigenvectors solo existen para matrices cuadradas y no para todas. En el caso de que una matriz n x n tenga eigenvectors, el número de ellos es n.
- Si se escala un eigenvector antes de multiplicarlo por la matriz, se obtiene un múltiplo del mismo eigenvector. Esto se debe a que si se escala un vector multiplicándolo por cierta cantidad, lo único que se consigue es cambiar su longitud pero la dirección es la misma.
- Todos los eigenvectors de una matriz son perpendiculares (ortogonales) entre ellos, independientemente de las dimensiones que tengan.

2.1.2. Eigenvalue

Cuando se multiplica una matriz por alguno de sus eigenvectors se obtiene un múltiplo del vector original, es decir, el resultado es ese mismo vector multiplicado por un número. Al valor por el que se multiplica el eigenvector resultante se le conoce como eigenvalue. A todo eigenvector le corresponde un eigenvalue y viceversa.

En el método PCA, cada una de las componentes se corresponde con un eigenvector,

y el orden de componente se establece por orden decreciente de eigenvalue. Así, la primera componente es el eigenvector con el eigenvalue asociado más alto.

2.1.3. Matriz de covarianza

Una matriz de varianzas-covarianzas es una matriz cuadrada que contiene las varianzas y covarianzas asociadas con diferentes variables. Los elementos de la diagonal de la matriz contienen las varianzas de las variables, mientras que los elementos que se encuentran fuera de la diagonal contienen las covarianzas entre todos los pares posibles de variables.

La covarianza es un valor que indica el grado de variación conjunta de dos variables aleatorias respecto a sus medias. Es el dato básico para determinar si existe una dependencia entre ambas variables.

2.1.4. Cálculo de las componentes principales

Cada componente principal (Z_i) se obtiene por combinación lineal de las variables originales[13]. Se pueden entender como nuevas variables obtenidas al combinar de una determinada forma las variables originales. La primera componente principal de un grupo de variables ($X_1, X_2, \ldots, X_p$) es la combinación lineal normalizada de dichas variables que tiene mayor varianza:

$$Z_1 = \phi_{11}X_1 + \phi_{21}X_2 + ... + \phi_{p1}X_p \tag{2.1}$$

Que la combinación lineal sea normalizada implica que:

$$\sum_{j=1}^{p} \phi_{j1}^2 = 1 \tag{2.2}$$

Los términos $\phi_{11}, \ldots, \phi_{1p}$ reciben el nombre de *loadings* y son los que definen a la componente. ϕ_{11} es el loading de la variable X_1 de la primera componente principal. Los *loadings* pueden interpretarse como el peso/importancia que tiene cada variable en cada componente y, por lo tanto, ayudan a conocer que tipo de información recoge cada una de las componentes.

Dado un set de datos X con n observaciones y p variables, el proceso a seguir para calcular la primera componente principal es:

- Centralización de las variables: se resta a cada valor la media de la variable a la que pertenece. Con esto se consigue que todas las variables tengan media cero.
- Se resuelve un problema de optimización para encontrar el valor de los *loadings* con los que se maximiza la varianza. Una forma de resolver esta optimización es mediante el cálculo de eigenvector/eigenvalue de la matriz de covarianza.

Una vez calculada la primera componente (Z_1) se calcula la segunda (Z_2) repitiendo el mismo proceso, pero añadiendo la condición de que la combinación lineal no puede estar correlacionada con la primera componente. Esto equivale a decir que Z_1 y Z_2 tienen que ser perpendiculares. El proceso se repite de forma iterativa hasta calcular todas las posibles componentes ($min(n-1, p)$) o hasta que se decida detener

el proceso. El orden de importancia de las componentes viene dado por la magnitud del eigenvalue asociado a cada eigenvector.

2.2. K-Means

Los algoritmos de clustering son considerados de aprendizaje no supervisado. El clustering es una técnica para encontrar y clasificar K grupos de datos (clusters). Así, los elementos que comparten características semejantes estarán juntos en un mismo grupo, separados de los otros grupos con los que no comparten características[3].

El método K-means agrupa las observaciones en K clusters distintos, donde el número K lo determina explícitamente antes de ejecutar del algoritmo. K-means encuentra los K mejores clusters, entendiendo como mejor cluster aquel cuya varianza interna sea lo más pequeña posible. Se trata por lo tanto de un problema de optimización, en el que se reparten las observaciones en K clusters de forma que la suma de las varianzas internas de todos ellos sea lo menor posible[3]. Dos de las medidas más comúnmente empleadas definen la varianza interna de un cluster ($W(C_k)$) como:

- La suma de las distancias euclídeas al cuadrado entre cada observación (x_i) y el centroide (μ) de su cluster. Esto equivale a la suma de cuadrados internos del cluster.
- La suma de las distancias euclídeas al cuadrado entre todos los pares de observaciones que forman el cluster, dividida entre el número de observaciones del cluster.

Minimizar la suma total de varianza interna $\sum_{k=1}^{k} W(C_k)$ de forma exacta (encontrar el mínimo global) es un proceso muy complejo debido a la inmensa cantidad de formas en las que n observaciones se pueden dividir en K grupos. Sin embargo, es posible obtener una solución que, aun no siendo la mejor de entre todas las posibles, es muy buena (óptimo local). El algoritmo empleado para ello es:

1. Asignar aleatoriamente un número entre 1 y K a cada observación. Esto sirve como asignación inicial aleatoria de las observaciones a los clusters.
2. Iterar los siguientes pasos hasta que la asignación de las observaciones a los clusters no cambie o se alcance un número máximo de iteraciones establecido por el usuario.
    - Para cada uno de los clusters calcular su centroide. Entendiendo por centroide la posición definida por la media de cada una de las variables de las observaciones que forman el cluster.
    - Asignar cada observación al cluster cuyo centroide está más próximo.

2.2.1. Ventajas y desventajas

Requiere que se indique de antemano el número de clusters que se van a crear. Esto puede ser complicado si no se dispone de información adicional sobre los datos con los que se trabaja. Se han desarrollado varias estrategias para ayudar a identificar potenciales valores óptimos de K, aunque todas ellas son orientativas.

Las agrupaciones resultantes pueden variar dependiendo de la asignación aleatoria inicial de los centroides. Para minimizar este problema se recomienda repetir el

proceso de clustering entre 25-50 veces y seleccionar como resultado definitivo el que tenga menor suma total de varianza interna. Aun así, solo se puede garantizar la reproducibilidad de los resultados si se emplean semillas.

2.2.2. Silhouette

Silhouette es una forma de medir qué tan cerca está cada punto de un cluster a los puntos en sus clusters vecinos. Es una forma ordenada de encontrar el valor óptimo para k para algoritmos de clustering como k-means. Los valores de silhouette se encuentran en el rango de [-1, 1]. Un valor cercano a 1 indica que la muestra está muy lejos de su cluster vecino y muy cerca del cluster asignado. De manera similar, un valor cercano a -1 indica que el punto está más cerca de su cluster vecino que del cluster asignado. Por lo tanto, cuanto mayor sea el valor, mejor es la configuración del cluster.

Se define[14] el valor de silhouette para un punto i perteneciente a un cluster, como *s(i)*, tal que:

$$s(i) = \frac{b(i) - a(i)}{max(b(i), a(i))} \tag{2.3}$$

Se define *a(i)* como la distancia media del punto i a todos los otros puntos en el cluster asignado A. Se puede interpretar *a(i)* como qué tan bien se asigna el punto al cluster, cuanto menor sea el valor, mejor será la asignación.

Del mismo modo, se define *b(i)* como la distancia media del punto i a otros puntos de su cluster vecino más cercano B. El punto i no pertenece al cluster B, pero la distancia a este es la más cercano entre todos los demás clusters.

Para que *s(i)* sea cercano a 1, *a(i)* debe ser muy pequeño en comparación con *b(i)*. Esto sucede cuando i está muy cerca de los demás puntos asignados a su mismo cluster. Un gran valor de *b(i)* implica que está muy lejos de su próximo cluster más cercano. En resumen, cuando *s(i)* es cercano a 1, indica que el conjunto de datos i está bien definido como cluster. Para obtener el valor de silhouette de todo el cluster, alcanza con la media de los valores de silhouette de todo el conjunto de datos para un cluster.

2.3. Random Forest

Random Forest, como su nombre lo indica, consiste en una gran cantidad de árboles de decisión individuales que operan como en conjunto. Cada árbol individual en random forest realiza una predicción de clase y la clase con más votos se convierte en la predicción de nuestro modelo [4].

El concepto fundamental detrás de random forest es simple pero poderoso: la sabiduría de las multitudes. En ciencia de datos, la razón por la que el modelo de bosque aleatorio funciona tan bien es que una gran cantidad de modelos (árboles) relativamente no correlacionados que operan como comité superará a cualquiera de los modelos constituyentes individuales.

La baja correlación entre modelos es la clave. Los modelos no correlacionados pueden producir predicciones de conjunto que son más precisas que cualquiera de las

predicciones individuales. La razón de este efecto es que los árboles se protegen entre sí de sus errores individuales (siempre que no se equivoquen constantemente en la misma dirección). Si bien algunos árboles pueden estar equivocados, muchos otros árboles estarán en lo correcto, por lo que, como grupo, los árboles pueden moverse en la dirección correcta. Los requisitos previos para que un bosque aleatorio funcione bien son:

- Es necesario que haya alguna señal real en nuestras funciones para que los modelos creados con esas funciones funcionen mejor que las conjeturas aleatorias.
- Las predicciones (y, por lo tanto, los errores) hechas por los árboles individuales deben tener bajas correlaciones entre sí.

Para asegurarse la baja correlación entre los árboles, random forest utiliza la técnica de *bagging*. Los árboles de decisión son muy sensibles a los datos utilizados en el entrenamiento del modelo. Pequeños cambios en el conjunto de entrenamiento pueden dar como resultado estructuras de árboles significativamente diferentes. Random forest hace uso de esto al permitir que cada árbol individual utilice para el entrenamiento muestras obtenidas aleatoriamente del conjunto de datos, lo que da como resultado diferentes árboles. Este es el proceso que se conoce como *bagging*.

Otra técnica que utiliza para obtener baja correlación la *aleatoriedad de features*.En un árbol de decisión normal, cuando es el momento de dividir un nodo, consideramos todas las características posibles y elegimos la que produce la mayor separación entre las observaciones en el nodo izquierdo y las del nodo derecho. Por el contrario, cada árbol en random forest puede elegir solo de un subconjunto aleatorio de características. Esto obliga a una variación aún mayor entre los árboles en el modelo y, en última instancia, da como resultado una menor correlación entre los árboles y una mayor diversificación.

3 Plataforma

Para el desarrollo de este proyecto se decidió trabajar con la plataforma Hortonworks Data Plataform perteneciente a Cloudera[15]. No obstante, en un principio está no fue la primera opción con la cual se trabajó. Las primeras implementación fueron hechas utilizando, simplemente, las librerías Pandas[6] y Scikit-Learn[8] pertenecientes al lenguaje Python, como se explico en el capítulo anterior.

En este capítulo se detallaran algunos de los principales componentes de la plataforma, que fueron claves para la realización de este proyecto. No obstante, vale aclarar, que este es un modo de uso que se le dio a la plataforma, por lo que existen muchas herramientas pertenecientes a esta, que no fueron exploradas ni utilizadas.

3.1. Hortonwork Data Plataform

Hortonworks Data Platform (HDP) es un framework de código abierto para el almacenamiento distribuido y el procesamiento de grandes conjuntos de datos de múltiples fuentes[16]. Que tiene como objetivo facilitar la implementación y administración de clústeres de Hadoop[17]. En comparación con la simple descarga de las diversas bases de código de Apache y tratar de ejecutarlas juntas en un sistema, HDP simplifica enormemente el uso de Hadoop. HDP permite la implementación ágil de aplicaciones, cargas de trabajo de aprendizaje automático y aprendizaje profundo, almacenamiento de datos en tiempo real y seguridad.

Hortonworks Sandbox[17] es una implementación de un solo nodo de HDP. Se empaqueta como una máquina virtual para que la evaluación y experimentación con HDP sea rápida y fácil. Las funciones de Sandbox están orientados a explorar cómo HDP puede ayudar a resolver problemas de big data.

En la figura 3.1 se puede apreciar, a grandes rasgos, como es la arquitectura de HDP. Donde se muestran algunos de los componentes principales que posee la plataforma dependiendo del uso que se le quiera dar.

3.1.1. Instalación y configuración

HDP se puede descargar y usar de forma gratuita[18]. Para este proyecto se utilizó la versión Sandbox VirtualBox v3.0, pero también se puede descargar en versión Docker y VMWARE. Cabe destacar que se recomienda disponer de al menos 10 GB en memoria RAM libre para el uso exclusivo de HDP.

La versión Sandbox de HDP utiliza un sistema operativo Red Hat de 64 bits, por lo que desde la consola del sistema se puede utilizar el comando *yum install <componente>* para instalar cualquier componente adicional que se desee como se haría en cualquier otro sistema Red Hat.

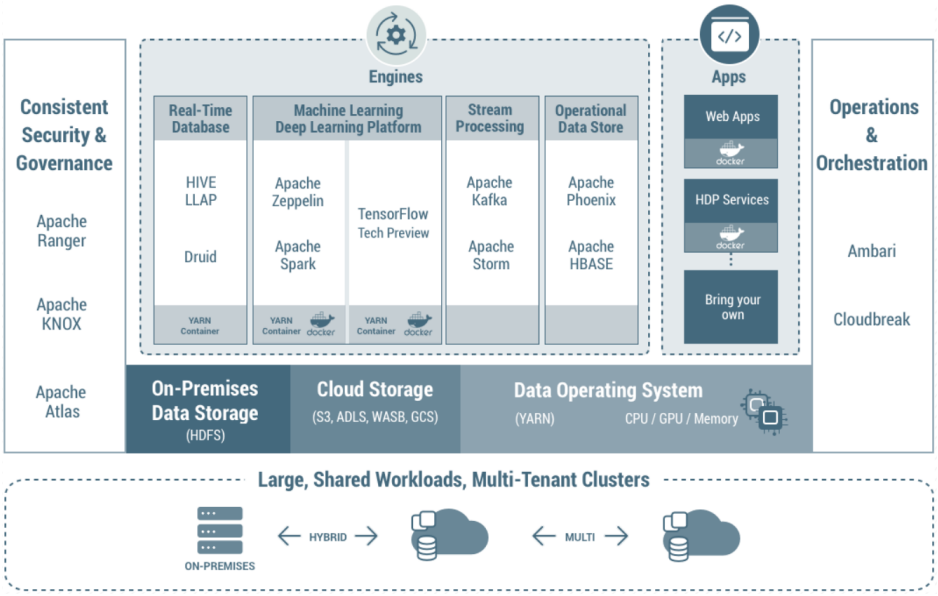

FIGURA 3.1: Arquitectura de HDP v3.1. Fuente: HDP Reference Architecture [16]

Esta versión Sandbox de HDP levanta los servicios en una determinada *ip*[19], que puede variar dependiendo de la versión descargada:

Docker: IP Address = 127.0.0.1
VirtualBox: IP Address = 127.0.0.1
VMWare: IP Address = 192.168.x.x

Una vez obtenida esta *ip* se recomienda modificar el archivo de hosts, ya que una vez configurado, permite que la dirección IP del Sandbox se asigne a un nombre de host que sea más fácil de recordar que un número. Esta acción depende del sistema operativo que se utilice:

- Mac:
  echo '{IP-Address} sandbox-hdp.hortonworks.com sandbox-hdf.hortonworks.com' | sudo tee -a /private/etc/hosts
- Linux:
  echo '{IP-Address} sandbox-hdp.hortonworks.com sandbox-hdf.hortonworks.com' | sudo tee -a /etc/hosts
- Windows:
  - Abrir el notepad como administrador
  - Abrir el archivo *c:\Windows\System32\drivers\etc\hosts*
  - Agregar la siguiente linea:
    {IP-Address} localhost sandbox-hdp.hortonworks.com sandbox-hdf.hortonworks.com
  - Guardar el archivo

Reemplazar *{IP-Address}* por la *ip* que se especifico anteriormente, en donde se levantan los servicios.

Una vez iniciada la máquina virtual, lo más recomendable es cambiar la contraseña

del usuario *admin*. Para poder realizar esto, lo primero es abrir el *Shell Web Client* el cual por defecto se encuentra en la dirección *sandbox-hdp.hortonworks.com:4200*, para poder logearse por primera vez utilizar usuario *root* y contraseña *hadoop*, luego de esto ejecutar en consola el comando *ambari-admin-password-reset*. También se puede utilizar una conexión SSH, accediendo de la siguiente forma:

ssh root@sandbox-hdp.hortonworks.com -p 2222

Esta versión Sandbox de HDP ya cuenta con ciertos usuarios predefinidos con ciertos roles y permisos, mostrando un ejemplo de la variedad de roles que se puede tener y como administrarlos según la funcionalidad que desempeñen. La tabla 3.1 muestran estos usuarios predefinidos, con sus respectivas contraseñas, roles y servicios a los que tienen acceso.

Usuario	**Contraseña**	**Rol**	**Servicios**
admin	Es necesario un reset de la contraseña	Ambari Admin	Ambari
maria_dev	maria_dev	Spark and SQL Developer	Hive, Zeppelin, MapReduce/Tez/Spark, Pig, Solr, HBase/Phoenix, Sqoop, NiFi, Storm, Kafka, Flume
raj_ops	raj_ops	Hadoop Warehouse Operator	Hive/Tez, Ranger, Falcon, Knox, Sqoop, Oozie, Flume, Zookeeper
holger_gov	holger_gov	Data Steward	Atlas
amy_ds	amy_ds	Data Scientist	Spark, Hive, R, Python, Scala

TABLA 3.1: Distintos usuarios predefinidos en HDP

3.2. Apache Hadoop

Apache Hadoop es un framework de código abierto para el almacenamiento distribuido y el procesamiento de grandes conjuntos de datos en hardware básico (no potente). Hadoop permite obtener rápidamente información de grandes cantidades de datos estructurados y no estructurados. La base de Apache Hadoop puede decirse que la componen los siguientes módulos/proyectos:

- Hadoop Common: contiene bibliotecas y utilidades que necesitan otros módulos de Hadoop.
- Hadoop Distributed File System (HDFS): un sistema de archivos distribuido que almacena datos en máquinas no necesariamente muy potentes (de uso comercial), proporcionando un ancho de banda agregado muy alto en todo el cluster.
- Hadoop YARN: una plataforma de administración de recursos responsable de administrar los recursos informáticos en clusteres y usarlos para programar las aplicaciones de los usuarios.
- Hadoop MapReduce: Un modelo de programación para el procesamiento de datos a gran escala.

Cada proyecto ha sido desarrollado para ofrecer una función explícita y cada uno tiene su propia comunidad de desarrolladores y ciclos de lanzamiento individuales. Existen cinco pilares principales en Hadoop:

1. Gestión de datos: es posible almacenar y procesar grandes volúmenes de datos en una capa de almacenamiento que escala linealmente. El Sistema de archivos

distribuidos de Hadoop (HDFS) es la tecnología central para la capa de almacenamiento de escalamiento horizontal eficiente, y está diseñado para ejecutarse en hardware de bajo costo. Apache Hadoop YARN es el requisito previo para Hadoop, ya que proporciona la gestión de recursos y la arquitectura necesaria para permitir que una amplia variedad de métodos de acceso a datos operen en los datos almacenados en Hadoop con un rendimiento y niveles de servicio predecibles.

   - Apache Hadoop YARN: YARN es un framework de trabajo para el procesamiento de datos de Hadoop que extiende las capacidades de MapReduce al admitir cargas de trabajo, que no son de MapReduce, asociadas con otros modelos de programación.
   - HDFS: Hadoop Distributed File System (HDFS) es un sistema de archivos basado en Java que proporciona almacenamiento de datos escalable y confiable que está diseñado para abarcar grandes grupos de servidores.

2. Acceso a datos: Permite trabajar con datos en una amplia variedad de formas, desde lotes hasta en tiempo real. Apache Hive es la tecnología de acceso a datos más utilizada, aunque hay muchos motores especializados. Por ejemplo, Apache Pig ofrece capacidades de secuencias de comandos, Apache Storm ofrece procesamiento en tiempo real, Apache HBase ofrece almacenamiento NoSQL en columnas y Apache Accumulo ofrece control de acceso a nivel de celda. Todos estos motores pueden funcionar en un conjunto de datos y recursos gracias a YARN y motores intermedios como Apache Tez para acceso interactivo y Apache Slider para aplicaciones de larga duración.

   - Apache Hive: Hive es un data warehouse que permite un fácil resumen de datos y consultas ad-hoc a través de una interfaz similar a SQL para grandes conjuntos de datos almacenados en HDFS.

3. Gobernanza e integración de datos: Carga de datos de forma rápida y sencilla. Workflow Manager proporciona flujos de trabajo para el gobierno de datos, mientras que Apache Flume y Sqoop permiten una fácil ingesta de datos, al igual que las interfaces NFS y WebHDFS para HDFS.

4. Seguridad: Se proporciona seguridad en cada capa de la pila de Hadoop, desde HDFS y YARN hasta Hive y los demás componentes de acceso a datos en todo el perímetro del clúster a través de Apache Knox.

5. Operaciones: Administrar, supervisar y operar clústeres de Hadoop a escala (Apache Ambari).

Apache Hadoop es usado para resolver problemas en donde se necesite almacenar, procesar y analizar grandes volúmenes de datos. Ejemplos de estos casos puede ser: la automatización del marketing digital, la detección y prevención de fraudes, el análisis de redes sociales, etc.

3.3. HDFS

HDFS[20] es un sistema de archivos distribuido que está diseñado para almacenar grandes archivos de datos. HDFS es un sistema de archivos basado en Java que proporciona almacenamiento de datos escalable y confiable, y fue diseñado para

abarcar grandes grupos de servidores básicos. HDFS ha demostrado una escalabilidad de producción de hasta 200 PB de almacenamiento, y un solo cluster de 4500 servidores, soportando cerca de mil millones de archivos y bloques. HDFS es un sistema de almacenamiento distribuido, escalable y tolerante a fallas que trabaja en estrecha colaboración con una amplia variedad de aplicaciones concurrentes de acceso a datos, coordinadas por YARN.

Un clúster HDFS se compone de un NameNode, que administra los metadatos del cluster y DataNodes que almacenan los datos. Los archivos y directorios están representados en el NameNode por inodes. Inodes registra atributos como permisos, tiempos de modificación, acceso y espacio en disco.

El contenido del archivo se divide en bloques grandes (típicamente 128 megabytes), y cada bloque del archivo se replica independientemente en múltiples DataNodes (Generalmente el factor de replica es tres pero puede ser modificable). Los bloques se almacenan en el sistema de archivos local en los DataNodes.

Namenode supervisa activamente el número de réplicas de un bloque. Cuando se pierde una réplica de un bloque debido a una falla de DataNode o falla de disco, NameNode crea otra réplica del bloque. NameNode mantiene el árbol de espacio de nombres y la asignación de bloques a DataNodes, manteniendo toda la imagen del espacio de nombres en la RAM.

3.4. Apache MapReduce

MapReduce[21] es el algoritmo clave que utiliza el motor de procesamiento de datos Hadoop para distribuir el trabajo en un clúster. Un trabajo de MapReduce divide un gran conjunto de datos en fragmentos independientes y los organiza en pares clave y de valor para el procesamiento paralelo. Este procesamiento paralelo mejora la velocidad y la confiabilidad del clúster, devolviendo soluciones más rápidamente y con mayor confiabilidad.

La función Map divide la entrada en rangos por InputFormat y crea una tarea de mapa para cada rango en la entrada. JobTracker distribuye esas tareas a los nodos de trabajo. El resultado de cada tarea de mapeo se divide en un grupo de pares clave-valor para cada reducción.

La función Reduce luego recopila los diversos resultados y los combina para responder al problema mayor que el nodo maestro necesita resolver. Por lo tanto, la función Reduce puede recopilar los datos de todos los mapas para las claves y combinarlos para resolver el problema.

El sistema actual de Apache Hadoop MapReduce está compuesto por JobTracker, que es el maestro, y los esclavos por nodo llamados TaskTrackers. JobTracker es responsable de la gestión de recursos (gestión de los nodos de los trabajadores, es decir, TaskTrackers), el seguimiento del consumo de recursos/disponibilidad y también la gestión del ciclo de vida del trabajo (programación de tareas individuales del trabajo, seguimiento del progreso, proporcionar tolerancia a fallas para tareas, etc.).

El TaskTracker tiene la responsibilidad de iniciar/desmontar tareas en pedidos del

JobTracker y proporcionar información del estado de la tarea al JobTracker periódicamente.

3.5. Apache YARN

Hadoop HDFS es la capa de almacenamiento de datos para Hadoop y MapReduce era la capa de procesamiento de datos en Hadoop 1x. Sin embargo, el algoritmo MapReduce, por sí solo, no es suficiente para la gran variedad de casos de uso que se quería resolver. YARN[22] se introdujo en Hadoop 2.0, como un marco genérico de administración de recursos y aplicaciones distribuidas, mediante el cual se pueden implementar múltiples aplicaciones de procesamiento de datos personalizadas para la tarea en cuestión. La idea fundamental de YARN es dividir las dos responsabilidades principales del JobTracker, es decir, la gestión de recursos y la programación/supervisión del trabajo, en demonios separados: un ResourceManager global y un ApplicationMaster (AM) por aplicación.

ResourceManager es la máxima autoridad que arbitra recursos entre todas las aplicaciones en el sistema. El ApplicationMaster por aplicación tiene la tarea de negociar recursos desde el ResourceManager y trabajar con los NodeManager para ejecutar y monitorear las tareas componentes.

NodeManager es el esclavo por máquina, que es responsable de iniciar los contenedores de las aplicaciones, monitorear el uso de recursos (CPU, memoria, disco, red) e informarlo al ResourceManager.

3.6. Apache Spark

Apache Spark es un framework de programación para procesamiento de datos distribuidos diseñado para ser rápido y de propósito general. Como su propio nombre indica, ha sido desarrollada en el marco del proyecto Apache, lo que garantiza su licencia Open Source.

Apache Spark se basa en Hadoop MapReduce y amplía el modelo de MapReduce para usarlo de manera eficiente para más tipos de cálculos, que incluyen consultas interactivas y procesamiento de flujo. Consta de diferentes APIs (desarrollo en Scala, Java, Python y R) y módulos que permiten que sea utilizado con diferentes propósitos. Desde soporte para análisis interactivo de datos con SQL a la creación de complejos pipelines de machine learning y procesamiento en streaming, todo usando el mismo motor de procesamiento y las mismas APIs.

Spark[23] no es una versión modificada de Hadoop y, en realidad, no depende de Hadoop porque tiene su propia administración de clusteres. Hadoop es solo una de las formas de implementar Spark. Spark usa Hadoop de dos maneras: una es el almacenamiento y la segunda es el procesamiento. Dado que Spark tiene su propio cómputo de administración de clúster, usa Hadoop solo para fines de almacenamiento.

El la figura 3.2 se muestra tres formas de cómo se puede construir Spark con componentes Hadoop.

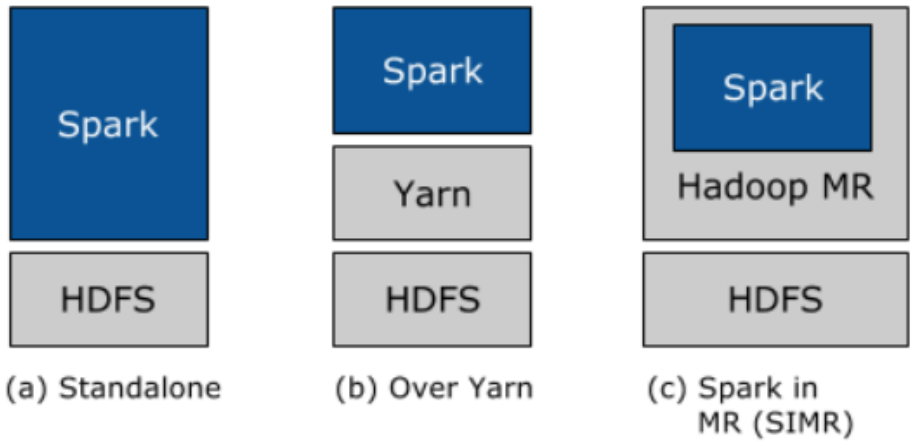

FIGURA 3.2: Tres formas de implementar Spark en un cluster de Hadoop. Fuente: Databricks [24]

A Standalone: se puede asignar estáticamente recursos en todas o un subconjunto de máquinas en un cluster Hadoop y ejecutar Spark junto con Hadoop MapReduce. El usuario puede ejecutar trabajos arbitrarios de Spark en sus datos HDFS. Su simplicidad hace que esta sea la implementación elegida por muchos usuarios de Hadoop 1.x.

B Over Hadoop Yarn: los usuarios de Hadoop que ya han implementado o planean implementar Hadoop Yarn pueden simplemente ejecutar Spark en YARN sin necesidad de preinstalación o acceso administrativo requerido. Esto permite a los usuarios integrar fácilmente Spark en su pila Hadoop y aprovechar toda la potencia de Spark, así como de otros componentes que se ejecutan sobre Spark.

C Spark In MapReduce (SIMR): para los usuarios de Hadoop que aún no ejecutan YARN, otra opción, además de la implementación Standalone, es utilizar SIMR para iniciar trabajos de Spark dentro de MapReduce.

3.6.1. Modelo de ejecución de Spark

La ejecución de la aplicación Spark[25] implica conceptos de tiempo de ejecución como controlador (driver), ejecutor (executor), tarea (task), trabajo (job) y etapa (stage). Ver figura 3.3. En tiempo de ejecución, una aplicación Spark se asigna a un proceso de controlador único y a un conjunto de procesos de ejecución distribuidos entre los hosts en un clúster.

El proceso del controlador gestiona el flujo de trabajo, programa las tareas y está disponible todo el tiempo que se ejecuta la aplicación. Normalmente, este proceso del controlador es el mismo que el proceso del cliente utilizado para iniciar el trabajo, aunque cuando se ejecuta en YARN, el controlador puede ejecutarse en el cluster.

Los ejecutores son responsables de realizar el trabajo, en forma de tareas, así como de almacenar información en caché. Un ejecutor tiene varios slots para ejecutar tareas.

Invocar una acción dentro de una aplicación Spark desencadena el inicio de un trabajo para cumplirlo. Spark examina el conjunto de datos del que depende esa acción y formula un plan de ejecución. El plan de ejecución ensambla las transformaciones del conjunto de datos en etapas. Una etapa es una colección de tareas que ejecutan el mismo código, cada una en un subconjunto diferente de datos.

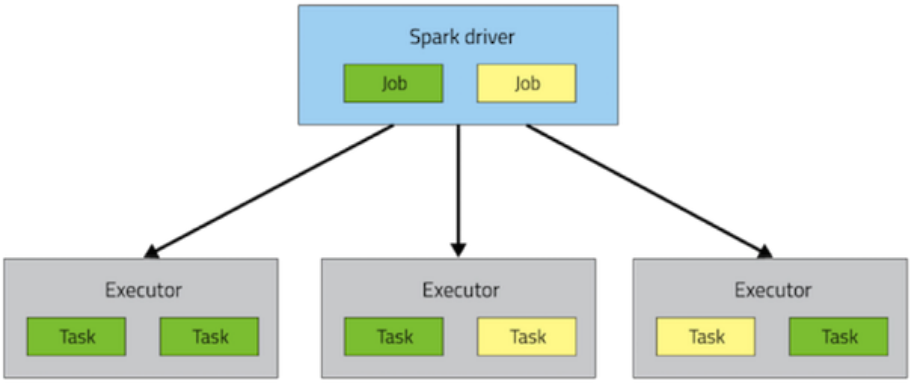

FIGURA 3.3: Modelo de ejecución Spark. Fuente: Clouder Documentation [25]

4 Aplicaciones Spark

Como ya se mencionó en el capitulo 1.2 se decidió trabajar con la plataforma HDP como medio para procesar y almacenar los datos provistos, así como también para realizar un estudio analítico de estos teniendo como objetivo el poder aplicar las pautas de trabajo descriptas en el capitulo 1.1. Para poder cumplir con esto, es que se decide trabajar con el componente Spark para el desarrollo de las aplicaciones y uso de algoritmos de learning, en especial se utiliza la API para Python de nombre Pyspark.

En un principio se utilizó Zeppelin para la implementación y ejecución de las aplicaciones. Apache Zeppelin es una implementación del concepto de web notebook, centrado en la analítica de datos interactivo mediante lenguajes y tecnologías como Shell, Spark, SparkSQL, Hive, R, etc. En una etapa posterior, luego de implementados y probados los algoritmos en Zeppelin, se decidió transcribirlos a archivos *.py* para poder ejecutarlos mediante *spark-submit* y de esta manera también se conoce otra herramienta y forma de trabajar con Spark.

4.1. Configuración de parámetros

Los trabajos (jobs) de Spark utilizan ejecutores (executors), que son aplicaciones que ejecutan tareas (task), que se ejecutan en un nodo del cluster. Los trabajos de Spark se subdividen en tareas que se distribuyen a los ejecutores de acuerdo con el tipo de operaciones y la estructura subyacente de los datos.

Los ejecutores tienen la capacidad de ejecutar múltiples tareas simultáneamente y usar cualquier cantidad de RAM física disponible en un solo nodo. Un ejecutor solo puede ejecutarse en un solo nodo (generalmente una sola máquina o VM). La configuración principal está determinada por un conjunto de parámetros:

- *spark.executors.instances*: define el número total de ejecutores disponibles para Spark.
- *spark.executors.cores*: define cuántas CPUs tiene permitido usar cada ejecutor. Esto afecta directamente su capacidad multitarea.
- *spark.executors.memory*: define cuánta RAM puede usar cada ejecutor.

En el caso de usar *Zepellin* los parámetros se pueden agregar y/o modificar en la definición del interprete Spark, utilizando los mismos nombres definidos anteriormente. En el caso de usar *spark-submit* es necesario especificar los parámetros en la línea de comando de la ejecución como se muestra en el ejemplo 4.1.

```
spark-submit prueba.py \
 --num-executors ? \
 --executor-cores ? \
```

```
--executor-memory ? [...]
```

EJEMPLO 4.1: Spark-submit definición de parámtros

Estos parámetros están relacionados directamente con la capacidad de hardware disponible para el uso de Spark. En la practica, se podrían definir valores de los parámtros que no se corresponda con la realidad de recursos que se dispone, aunque obviamente, esto producirá errores por parte del administrador de recursos (YARN) cuando intente acceder a recursos (por ejemplo memoria) no existentes.

Existen diferentes combinación de valores para estos parámetros. Una gran cantidad de ejecutores en general es bueno ya que se puede hacer más tareas en paralelo. A su vez, tener más memoria disponible también es algo positivo al igual que tener más de un núcleo (core) por ejecutador, en particular para el caso del número de núcleos, el número máximo de núcleos que se puede tener es 5. Ya que HDFS tiene problemas para el procesamiento de muchos hilos concurrentes, por lo que, para lograr un rendimiento de escritura máximo se recomienda no superar la cantidad de 5 núcleos por ejecutor[26].

El objetivo es encontrar una configuración de parámetros que mejor se adapte a los recursos de hardware disponibles. Y no caer en los extremos (en la jerga se les conoce como *tiny vs fat* ejecutores). Ejecutores *tiny* hace referencia a tener un solo núcleo por ejecutor, lo que significa tener tantos ejecutores por nodo como núcleos disponibles. Los ejecutores *fat*, hace referencia a usar todos los núcleos en un solo nodo para que solo haya un ejecutor por nodo del cluster.

Los pasos[27][28] principales para configurar los parámetros en un clúster de Spark dado el número de nodos del clúster, el número de núcleos por nodo y la cantidad de RAM por nodo:

1 Reservar un núcleo por nodo para YARN/Hadoop.

2 Usar los núcleos restantes como total de núcleos disponibles en el cluster.

3 Establecer *spark.executor.cores* = 5

4 Dividir el total de núcleos disponibles por *spark.executor.cores* para encontrar el número total de ejecutores en el cluster.

5 Reservar un ejecutor para el administrador de la aplicación (reducir el número de ejecutores en uno). Usar el valor resultante para establecer *spark.executor.instances*.

6 Calcular el número de ejecutores por nodo dividiendo el número de ejecutores por el número de nodos en el cluster (redondeando al entero más cercano).

7 Calcular la memoria por ejecutor dividiendo la RAM total del nodo por los ejecutores por nodo.

8 Reducir en un 7 % la memoria del ejecutor para tener en cuenta el *heap overhead* para YARN/Hadoop. Utilizar el número resultante como *spark.executor.memory*.

En el paso 3 se define *spark.executor.cores* = 5, que es el número ideal de núcleos como ya se mencionó anteriormente. Sin embargo, esta definición podría ocasionar que no se utilicen ciertos núcleos en cada nodo, dependiendo de la configuración de hardware que se tenga. En general, las únicas soluciones que nunca dejan ningún núcleo

sin usar son tener ejecutores *tiny* o ejecutores *fat*, pero ambas opciones tienen inconvenientes, como ya se mencionó.

En la referencia [29] se presenta un algoritmo (ejemplo 4.2) en donde el objetivo es encontrar el mejor valor para *spark.executor.cores*:

```
def calc_executor_cores(available_cores):
    executor_cores_max = 5
    if available_cores >= executor_cores_max:
        executor_cores = min(executor_cores_max, available_cores // 2)
    else:
        executor_cores = max(1, available_cores // 2)
    remainder_cores = available_cores % executor_cores
    while remainder_cores > 1 and executor_cores > 2:
        executor_cores -= 1
        remainder_cores = available_cores % executor_cores
    return executor_cores
```

EJEMPLO 4.2: Definición de *spark.executor.cores*

La función toma el número de núcleos disponibles por nodo (después de eliminar el núcleo que se le asigna a YARN) como entrada y devuelve el valor calculado para *spark.executor.cores*. Sus principales pasos son:

- Inicializar *executor_cores* a un número que pueda ajustarse a los núcleos disponibles.
- Comenzar a reducir *executor_cores* en una unidad hasta que no haya núcleos no utilizados o *executor_cores* = 2.
- Devolver el *executor_cores* resultante.

De esta manera se tiene las herramientas necesarias para poder estimar los tres parámetros necesarios para optimizar el uso de Spark. Por ejemplo, en el marco de este proyecto, se tiene como recursos de hardware, un único nodo que cuenta con 25 núcleos y 77 Gb de memoria RAM disponibles para usar por parte de la plataforma. Al aplicar los pasos anteriores para encontrar los parámetros optimos, tenemos como resultado que:

- *spark.executors.instances* = 6
- *spark.executors.cores* = 4
- *spark.executors.memory* = 11 Gb

No obstante, la configuración de estas variables no garantiza el correcto funcionamiento de Spark. Según las referencias [30][31] existen problemas al ejecutar aplicaciones *Pyspark*, en el log aparecerá el error *Container killed by YARN for exceeding memory limits..*, por ejemplo.

Cuando se ejecuta *Pyspark*, existen dos procesos: un proceso JVM "on-heap" y un proceso Python "off-heap". El parámetro *spark.executors.memory* define el tamaño del "heap" y ambos procesos están limitados por la memoria del contenedor. Por lo tanto, cuanto más grande sea el "heap", menos memoria tendrá el proceso de Python, y puede alcanzar el límite del contenedor más rápido.

En estos casos se recomienda disminuir el tamaño de *spark.executors.memory*. Otra alternativa es reducir la cantidad de núcleos, para que se ejecuten menos tareas en paralelo dentro del mismo ejecutor. Si bien esto reduce el paralelismo, aumentó la cantidad de memoria disponible para cada tarea lo suficiente como para que sea posible completar el trabajo.

No obstante, en la mayoría de las ocasiones, la configuración de los parámetros depende fuertemente del tipo de aplicación que se está ejecutando y que no existe una única configuración para todas las aplicaciones. A modo de ejemplo de lo dicho anteriormente, para poder realizar una ejecución completa del algoritmo *k-means* se probaron varias configuraciones hasta llegar a una que completara la ejecución. Como recursos se contaba con la mitad aproximadamente de los recursos del nodo (12 núcleos y 40 Gb de RAM), la configuración de los parámetros que mejor se adapto en este caso fue:

- *spark.executors.instances* = 2
- *spark.executors.cores* = 3
- *spark.executors.memory* = 18 Gb
- *spark.yarn.am.memory* = 2 Gb

4.2. Ejecución de aplicaciones Python en Spark

Administrar dependencias y hacerlas disponibles para trabajos de Python en un cluster puede ser difícil. Para determinar qué dependencias son necesarias en el cluster, debe comprender que las aplicaciones de código de Spark se ejecutan en ejecutores de Spark distribuidos en todo el cluster. Si las transformaciones de Python que define utilizan bibliotecas de terceros, como NumPy, los ejecutores de Spark requieren acceso a esas bibliotecas cuando se ejecutan en ejecutores remotos.

Después de que los paquetes de Python que se desea utilizar están en una ubicación coherente en su clúster, hay que establecer las variables de entorno apropiadas[32] en la ruta de los ejecutables de Python. Para esto es necesario especificar la dirección del archivo binario de Python que se desea utilizar, configurando la variable de entorno *PYSPARK_PYTHON* en *spark-env.sh*. También hay que sobreescribir la ruta del binario de Python del controlador (driver) utilizando la variable de entorno *PYSPARK_DRIVER_PYTHON*. Esto es necesario independientemente si se está usando el modo yarn-client o yarn-cluster.

En este proyecto se decidió trabajar con ambientes *conda* para tener una mejor administración del trabajo, ya que se tiene distintos tipos de usuarios trabajando sobre la plataforma, por lo que se crea una ambiente especifico dependiendo del rol del usuario y dentro de cada ambiente se instalan los paquetes Python requeridos por el usuario. Para distribuir los entornos a la hora de ejecutar una aplicación Spark, se utiliza *conda-pack*.

Conda-pack se puede utilizar[33] para distribuir entornos de trabajos *conda*, para usar con trabajos de Spark cuando se utiliza YARN como administrador de recursos. Al agrupar el entorno de trabajo para su uso en Spark, se puede hacer uso de todas las librerías proporcionadas por *conda* y asegurarse de que se proporcionen a cada

nodo del cluster. Esto hace uso de la localización de recursos de YARN mediante la distribución de entornos como archivos, que luego se disponen automáticamente en cada nodo. Se deben usar los formatos *tar.gz* o *zip*.

Los pasos a seguir desde crear el ambiente hasta su ejecución con *spark-submit* son los siguientes:

1 conda create -y -n example python=3.5 numpy pandas scikit-learn

2 conda activate example

3 conda pack -o environment.tar.gz

Primero se crea el ambiente *conda*, en ese ejemplo el nombre del ambiente será *example*, el cual tendra Python 3.5 y una serie de paquetes a instalar. En segundo lugar se activa el ambiente, una vez activado se estará trabajando dentro de ese entorno, teniendo disponibles para uso los paquetes previamente instalados en el ambiente. Por último se empaqueta todo el entorno en un archivo de nombre *environment.tar.gz* y es este archvio el posteriormente será distribuido por los nodos del cluster cuando se desee ejecutar una aplicación Spark.

El comando para ejecutar *spark-submit*, depednerá si es en modo yarn-client o yarn-cluster, esto se puede apreciar en el ejemplo 4.3, en donde se muestra el uso tanto en modo yarn-client o yarn-cluster.

```
#Modo yarn-cluster
PYSPARK_PYTHON=./environment/bin/python \
spark-submit \
--conf spark.yarn.appMasterEnv.PYSPARK_PYTHON=./environment/bin/python \
--master yarn \
--deploy-mode cluster \
--archives environment.tar.gz#environment \
prueba.py

#Modo yarn-client
PYSPARK_DRIVER_PYTHON=`which python` \
PYSPARK_PYTHON=./environment/bin/python \
spark-submit \
--conf spark.yarn.appMasterEnv.PYSPARK_PYTHON=./environment/bin/python \
--master yarn \
--deploy-mode client \
--archives environment.tar.gz#environment \
prueba.py
```

EJEMPLO 4.3: spark-submit utilizando ambientes *conda*

En caso de utilizar *Zepellin*, se setean las variables de entorno antes descritas (*PYSPARK_PYTHON* y *PYSPARK_DRIVER_PYTHON*) en la definición del interprete y además, es necesario cambiar la variable *zeppelin.pyspark.python* por la ruta al binario de python que se desea utilizar.

4.3. Conceptos generales de implementación

Debido a que queremos trabajar con datos en columnas, utilizaremos DataFrames que son parte de Spark SQL. El punto de entrada para usar Spark SQL es un

objeto llamado SparkSession. Inicia una aplicación Spark en la que se ejecutará todo el código de esa sesión[34]. Para trabajar con datos almacenados en tablas Hive desde aplicaciones Spark, se debe crear un HiveContext[35], explícitamente como en el ejemplo 4.4 siempre y cuando la aplicación se ejecute con *spark-submit*, de lo contrario no es necesario ya que con Zepellin se crea automáticamente con SQLContext como se muestra en la linea comentada.

- builder: proporciona acceso a la API Builder, que se usa para configurar la sesión.
- master(): determina dónde se ejecutará el programa; "local [*]" lo configura para ejecutarse localmente en todos los cores, pero puede usar "local [1]" para ejecutarse en un core, por ejemplo.
- appName(): método opcional para nombrar la aplicación Spark.
- getOrCreate(): obtiene una SparkSession existente o crea una nueva si no existe.

```
from pyspark import SparkContext
from pyspark.sql import SQLContext, SparkSession, HiveContext

spark = SparkSession.builder \
    .master("local[*]") \
    .appName("Learning_Spark") \
    .getOrCreate()

sc = spark.sparkContext
sqlContext = HiveContext(sc)
#sqlContext = SQLContext(sc)
```

EJEMPLO 4.4: Configurando SparkSession

La carga y almacenamiento de datos utilizando Hive es bastante simple de usar, como se muestra en el ejemplo 4.5. En donde en primera instancia se cargan los datos en un dataframe Spark desde una tabla de nombre *datos_procesados_ceibal* en Hive (Spark también soporta otros tipos de datos como csv, parquet, etc) y la forma de guardado es como se muestra en la segunda línea. En este ejemplo se utiliza el modo *"overwrite"*, el cual si existe la tabla, la elimina y crea otra nueva, también existe otros modos como *"append"* el cual agrega los datos a una tabla ya existente[36].

```
#Carga los datos de la tabla hive en un dataframe spark
df = spark.sql('select␣*␣from␣datos_procesados_ceibal')

#Guarda el dataframe en una nueva tabla hive
df.write.mode('overwrite').saveAsTable('datos_procesados_ceibal')
```

EJEMPLO 4.5: Carga y guardado en Hive

4.4. Procesamiento de CSVs

Para poder utilizar los algoritmos de learning de manera apropiada, hay que crear una matriz de entrada con los datos correspondientes, esta matriz se crea a partir del procesamiento de los archivos CSV.

Los archivos CSV se encuentran almacenados dentro de carpetas organizadas por día, en donde para un mismo día se tiene varios archivos. Se procesan cada uno de estos archivos para luego poder crear la matriz de entrada para los algoritmo.

Como primer paso, se crea el rango de fechas con el que se quiere trabajar y se procesan los CSV pertenecientes a esas fechas solamente. De esta manera se le permite al usuario decidir si se van a analizar los datos correspondientes a un día, una semana, un mes o el rango de fechas que se desee. Ver ejemplo 4.6.

```
#Se crea el rango de fechas que se desea utilizar para cargar los datos
start = datetime.datetime.strptime("01-03-2019", "%d-%m-%Y")
end = datetime.datetime.strptime("07-03-2019", "%d-%m-%Y")
date_generated = [start + datetime.timedelta(days=x)
                  for x in range(0, (end-start).days)]
```

EJEMPLO 4.6: Rango de fechas a utilizar

Una vez definido el rango de fecha a utilizar, se recorre mediante un *for*, cada una de esas fechas y para cada fecha se carga en memoria los archivos CSV correspondientes para realizar su procesamiento. Ver ejemplo 4.7.

```
#Para cada fecha del rango anterior generado
#Se buscan todos los csv de datos para ese dia y se cargan en un dataframe
for auxDate in date_generated:
    auxdf = spark.read.load("file:///opt/datos/dnslogs/" +
                            auxDate.strftime("%Y-%m-%d") + "/*.csv",
                            format="csv", sep=",", inferSchema="true",
                            header="false")
    procesarCSV(auxdf)
```

EJEMPLO 4.7: Se procesa cada CSV

Otra alternativa que en un principio se utilizó fue la de cargar todos los CSV, del rango de fechas, y luego realizar el procesamiento. Pero esta solución tiene el inconveniente de que se malgastan recursos, ya que es necesario almacenar todos los datos en memoria para luego procesarlos. Al procesar de un día por vez, solo se tiene en memoria los datos pertenecientes a ese día únicamente.

La función *procesarCSV* recibe como parámetro un dataframe y es la encargada de realizar las modificaciones, agregar o eliminar columnas y por útlimo, guarda el dataframe en una base hive.

4.5. Aprendizaje automático en Spark (MLlib)

Spark cuenta con una librería propia dedicada al aprendizaje automático. Ofrece algoritmos de alta calidad (por ejemplo, múltiples iteraciones para aumentar la precisión) y una alta velocidad de ejecución (hasta 100 veces más rápido que MapReduce). La biblioteca se puede usar en Java, Scala y Python como parte de las aplicaciones de Spark. Spark cuenta con dos versiones de está librería, una orientada a Dataframes (que es la utilizada en este proyecto) y otra basada en RDD (Resilient Distributed Datasets), la cual actualmente está en mantenimiento por lo que puede tener bugs[9].

Muchos de los algoritmos de learning implementados en la librería MLlib, no aceptan una matriz de datos de entrada tal cual, sino que es necesario vectorizar las columnas con las que se desea trabajar, es decir, se crea una nueva columna en donde cada fila contiene un vector con los valores de las columnas con las que se quiere trabajar para esa fila (ver ejemplo 4.8), y es esta columna la que se usa como entrada para los algoritmos.

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

#Carga las features (columnas) que se usaran como entrada en los algoritmos
#Se sacan las dos primeras columnas
#Que se corresponden con el time y el identificador
features = df.schema.names
features = features[2:]

#Transforma los valores de las columnas, en el array features, en un vector
#Lo guarda en una nueva columna de nombre "features"
assembler = VectorAssembler(inputCols=features,outputCol='features')
assembled_df  = assembler.transform(df)
```

EJEMPLO 4.8: Vectorización de features

A su vez en algoritmos como PCA y K-Means, es necesario que los datos de entrada estén normalizados. Para esto se utiliza la función *StandardScaler* que toma como entrada un conjunto de datos de filas de vectores, normaliza cada feature para que tenga una desviación estándar de la unidad y/o media cero (ver ejemplo 4.9). Para esto se configura los parámetros:

- *withStd*: True por defecto. Escala los datos a la desviación estándar de la unidad.
- *withMean*: False por defecto. Centra los datos con la media antes de escalar.

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import StandardScaler

#Estandariza las features al eliminar la media
#Y escalar a la varianza de la unidad
#La salida se guarda en la columna "scaledFeatures"
standardScaler = StandardScaler(inputCol="features", \
                 outputCol="scaledFeatures", withStd=True, withMean=True)
scaled_df = standardScaler.fit(assembled_df).transform(assembled_df)
```

EJEMPLO 4.9: Estandarización de features

4.5.1. PCA

Como se mencionó anteriormente, la mayoría de los algoritmos de learning en la librería usada, trabajan con vectores de datos. PCA no es la excepción, es bastante intuitivo y fácil de usar. Como primer paso, se crea el modelo de PCA[37], para esto hay que definir los parámetros mínimos que contiene el modelo, estos son:

- *inputCol*: que al igual que en los algoritmos anteriores y en muchos otros más pertenecientes a esta librería, se utiliza para indicar el nombre de la columna con los datos de entrada. Recordar que los datos deben estar vectorizados.

- *outputCol*: este parámetro se utiliza para indicar el nombre de la columna de salida en donde se almacenará los resultados. Los resultados como es de esperar también se encuentran vectorizados.
- *k*: indica el número de componentes que se desea obtener.

Una vez seteados esto parámetros, se procede a hacer el "fit" (entrenamiento) del modelo, tomando como entrada el dataframe que contiene la columna indicada en el parámetro inputCol del modelo, ver ejemplo 4.10. Una vez terminado el "fit" se puede aplicar la función "transform" del modelo. La cual permite obtener el transformado de los datos de entrada en función de los componentes principales que se obtuvieron.

```
from pyspark.ml.feature import PCA

#Crea el modelo PCA tomando como entrada la columna "scaledFeatures"
#con el "fit" realiza en "entrenamiento" del modelo
#para aplicarlo en los datos hacer pca.transform(dataframe)

pca = PCA(k=20, inputCol="scaledFeatures", outputCol="pcaFeatures")
model = pca.fit(scaled_df)
```

EJEMPLO 4.10: PCA en Spark

Del modelo, es posible obtener otros datos de relevancia. Uno de estos datos es la varianza de cada componente principal, lo que permite identificar las componentes con más peso en el nuevo sistema, se obtiene invocando dicha propiedad del modelo (*model.explainedVariance*). Otro dato de importancia es la matriz con los vectores correspondiente a cada componente principal, cada columna de dicha matriz se corresponde con un componente principal, para obtener dicha matriz se invoca la propiedad *model.pc*.

```
#Guarda el modelo de PCA
model.save('models_pca_ceibal')
```

EJEMPLO 4.11: Guardado del modelo

Otro punto importante a tener en cuenta y que facilita la implementación, es el concepto de modelos. La mayoría de los algoritmos que brinda la librería de learning permite crear modelos de los algoritmos, esto es que, se carga primero los parámetros correspondiente al algoritmo en un modelo, se hace el entrenamiento del modelo con los datos que se desee y luego es este modelo que se utiliza como herramienta, permitiendo además el guardado de este, ver ejemplo 4.11.

Esto nos brinda la facilidad de que una vez definido y entrenado el modelo, se pueda utilizar en el momento que uno desee, de esta manera facilita la implementación y agiliza los tiempo, ya que el entrenamiento de los modelos es lo que requiere más tiempo y esfuerzo. Esto se puede ver en el ejemplo 4.12, en una primera instancia se carga el modelo de PCA (previamente implementado y entrenado), se lo aplica a los datos deseados y luego sobre este resultado es que se utiliza k-means.

4.5.2. K-Means

Con el algoritmo k-means[38], se tiene algunos parámetros similares a los de PCA. También posee un parámetro para indicar la columna que será la entrada de

datos, en este caso se recibe el nombre de *featuresCol* que al igual que en PCA, los datos deben estar vectorizados. Y tiene una columna *predictionCol* en la cual se indicará a cual cluster pertenece la tupla. Además de estos parámetros posee otros como:

- *k*: para indicar el número de cluster que se desea crear.
- *maxIter*: para indicar el número máximo de iteraciones que se desea.
- *tol*: para indicar la tolerancia de convergencia del algoritmo.
- *initMode*: para indicar el algoritmo de inicialización. Puede ser "random" para elegir puntos aleatorios como centros de las agrupaciones iniciales o "k-means||" para usar una variante paralela de k-means++.
- *initSteps*: el número de pasos para el modo de inicialización "k-means||".
- *seed*: para elegir una semilla random de inicialización del algoritmo.
- *distanceMeasure*: indica la medida de distancia que se utilizará en el algoritmo. Opciones admitidas: *euclidiana* y *coseno*.

En este proyecto se utilizaron los parámetros: *seed*, eligiendo un número primo mayor a mil como semilla, *distanceMeasure*, seleccionando la distancia euclidiana, se utiliza la opción "random" para el parámetro *initMode*, ver ejemplo 4.12. Y para elegir el número de cluster a crear, se opto por utilizar en un paso previo el estimador de clustering, Silhouette.

```
from pyspark.ml.feature import PCA, PCAModel
from pyspark.ml.clustering import KMeans, KMeansModel

#Se ejecuta PCA para luego aplicar kmeans sonre los resultados del PCA
model_pca = PCAModel.load('models_pca')
resultDFpca = model_pca.transform(scaled_df)

#Se ejecuta kmean
kmeans = KMeans(k=15, seed=5043, featuresCol='pcaFeatures',  \
                initMode='random', distanceMeasure='euclidian')
model = kmeans.fit(resultDFpca)
```

EJEMPLO 4.12: k-means en Spark

Silhouette es una medida para la validación de la coherencia dentro de los clusters. Varía entre 1 y -1, donde un valor cercano a 1 significa que los puntos en un cluster están cerca de los otros puntos en el mismo cluster y lejos de los puntos de los otros cluster.

El procedimiento es evaluar Silhouette para los distintos k clusters posibles. Luego de obtenido el valor de Silhouette para cada división en k clusters, se toma como válidos el k cuyo valor de Silhouette se aproxime más a 1. Una vez conocido el número de clusters k, se vuelve a ejecutar k-means con el k encontrado anteriormente. Al igual que en PCA primero se hace el "fit" del modelo y luego se procede a ejecutar la función "transform", creando la columna indicada en *predictionCol*, indicando para cada tupla a cual cluster pertenece.

```
from pyspark.ml.evaluation import ClusteringEvaluator

silh_lst = []
```

```
k_lst = [2, 6, 10, 15, 20]
evaluator = ClusteringEvaluator()

for kCluster in k_lst:
    # Entrenamiento de k-means model.
    kmeans = KMeans(k=15, seed=5043, featuresCol='pcaFeatures',  \
                initMode='random', distanceMeasure='euclidian')
    model = kmeans.fit(resultDFpca)

    # Se hacen las predicciones de los clusters
    predictions = model.transform(resultDFpca)

    # Se evalua el cluster calculando el valor de silhouette
    silhouette = evaluator.evaluate(predictions)

    print("K:␣", kCluster)
    print("silhouette:␣", silhouette)

    silh_lst.append({"K":kCluster,"silhouette":silhouette})

print(silh_lst)
```

EJEMPLO 4.13: Estimador k-means silhouette

Como se puede ver en el ejemplo 4.13, solo se toman algunos valores de k cluster y no todos los valores posibles. Esto se debe a que hacer el "fit" para k-means, teniendo grandes volúmenes de datos como los que se trabajaron y con los recursos disponibles, la demora y consumo de recursos puede llegar a ser excesiva. Es por esta razón que se elige tomar ciertos valores de k. Por otra parte lo ideal sería ejecutar k-means más de una vez (se recomienda entre 25-30 veces) para un mismo k, pero por las mismas razones esto no pudo ser viable.

Otro dato importante que se puede obtener del modelo, es el centroide de los clusters. Una vez terminado el "fit" del modelo simplemente se utiliza la función *model.clusterCenters()*.

4.5.3. Random Forest

Random Forest es el modelo (de los trabajados en este proyecto) que más parámetros tiene para configurar. Esto tiene como consecuencia que no es una tarea fácil encontrar la combinación de parámetros que mejor se adapte a la realidad del problema planteado.

Otro punto a recordar, es la forma en que se decidió utilizar random forest, la idea es tener dos grandes grupos de sets de datos, los datos reales y los datos sintéticos (ficticios), los cuales se diferencian por la columna "etiqueta" la cual adopta el valor 1 para los datos reales y 0 para los datos sintéticos. Y es esta columna la que se desea "predecir" con random forest.

EL objetivo de utilizar random forest no es le resultado final en si, ya que ya se tiene conocimiento de cuales son los datos reales y cuales sintéticos. El objetivo es poder identificar que features son las más relevantes, según random forest, a la hora de decidir si el dato es real o no.

A continuación se describen los parámetro que fueron utilizados en la implementación y que, según nuestro criterio, son los que más influencia podrían tener en el resultado. Por más información sobre los otros parámetros, referirse a [39].

- *featuresCol*: nombre de la columna con los datos vectorizados con las features a tener en cuenta.
- *labelCol*: nombre de la columna que se desea predecir.
- *predictionCol*: nombre de la columna en donde se colocarán los resultados.
- *maxDepth*: profundidad máxima del árbol. Por ejemplo, profundidad 0 significa 1 nodo de hoja, profundidad 1 significa 1 nodo interno más 2 nodos hoja.
- *numTrees*: número de árboles (forest) deseados para el "fit" (entrenamiento).
- *seed*: para elegir una semilla random de inicialización del algoritmo.
- *impurity*: criterio utilizado para el cálculo de ganancia de información. Opciones admitidas: "entropy" y "gini".

Los valores elegidos fueron: para *seed*, al igual que en random forest se eligió un número primo mayor a mil como semilla. En el caso de *maxDepth* se le asigno el valor 5, ya que se entiende que con esa profundidad cada árbol de decisión tiene en cuenta todas las features del modelo. Por último se eligió tener 200 (*numTrees*) árboles de decisión como bosque (forest).

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

df_FAKE = spark.sql("select␣*␣from␣datos_procesados_ceibal_FAKE")
df = spark.sql("select␣*␣from␣datos_procesados_ceibal")

#ELIMINO DOS COLUMNAS DEL ORIGINAL
#PARA HACER LA UNION TIENEN QUE TENER LAS MIMSAS COLUMNAS
df = df.drop("Most_Granular_Identity")
df = df.drop("Timestamp")

# UNION DE LOS DOS DATAFRAMES
dfUnion = df.union(df_FAKE)

#Se hace un shuffle de los datos
shuffledDF = dfUnion.orderBy(rand())

#Carga las features que se usaran como entrada en los algoritmos
#no se cuenta la  ltima  que se corresponda con la columna "Etiqueta"
features = shuffledDF.schema.names
features = features[:-1]

#Transforma los valores de las columnas, en el array features
#en un vector y lo guarda en una nueva columna de nombre "features"
assembler = VectorAssembler(inputCols=features,outputCol='features')
shuffledDF  = assembler.transform(shuffledDF)
```

EJEMPLO 4.14: RF - preparación de los datos

Como se visualiza en el ejemplo 4.14, el primer paso que se hace es cargar los datos, tantos los reales como los ficticios, se eliminan las columnas que no se utilizan y se hace la unión de los dos dataframes. Se extraen las features con las que se quiere trabajar y se vectorizan los datos al igual que en los algoritmos anteriores.

```
from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer

# Etiquetas de  ndice , agregando metadatos a la columna de etiqueta.
# Ajuste en todo el conjunto de datos para incluir todas las etiquetas en el indice.
labelIndexer = StringIndexer(inputCol="Etiqueta", \
                             outputCol="indexedLabel").fit(shuffledDF)

# Identifica autom ticamente las categ ricas y crea un indice con ellas.
featureIndexer =  VectorIndexer(inputCol="features", \
                                outputCol="indexedFeatures").fit(shuffledDF)

# Dividir los datos en conjuntos de entrenamiento y prueba (30 % para pruebas)
(trainingData, testData) = shuffledDF.randomSplit([0.7, 0.3], seed=5043)
```

EJEMPLO 4.15: RF - indexado y separación de los datos

En un segundo paso, se realiza el entrenamiento y creación de los modelos *StringIndexer* y *VectorIndexer*. Estos modelos son utilizados para crea indices, por la columna que se desea predecir (*StringIndexer*) y por las features que se desean utilizar (*VectorIndexer*). Esto se hace para tener una mejor performace al utilizar luego random forest. Por ultimo se divide, de manera random, todo el conjunto de datos en dos grupos *trainingData* y *testData*, el primer grupo es utilizado para entrenar el modelo de random forest mientras que el segundo grupo se utiliza para validar las predicciones del modelo. Ver ejemplo 4.15.

```
from pyspark.ml.classification import RandomForestClassifier

# Entrena un modelo RandomForest.
rf = RandomForestClassifier(featuresCol='indexedFeatures', \
                            labelCol="indexedLabel" , \
                            predictionCol='prediction', \
                            probabilityCol='probability', \
                            rawPredictionCol='rawPrediction', maxDepth=5, \
                            impurity='gini', numTrees=200, \
                            seed=5043)

# Convierta las etiquetas indexadas a etiquetas originales.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel", \
                               labels=labelIndexer.labels)

# Se guarda la cadena de modelos en un Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf, labelConverter])

# Se entrenan los modelos con el conjuto de datos de entrenamiento
model = pipeline.fit(trainingData)

# Se utiliza el modelo para hacer las predicciones sobre los datos de testeo.
predictions = model.transform(testData)

#SAVE PIPELINE MODEL
```

```
model.save('pipeline_model')

#SAVE RANDOM FOREST MODEL
rfModel = model.stages[2]
rfModel.save('random_forest_model')

importances = rfModel.featureImportances
```

EJEMPLO 4.16: Creación de modelo RF

Luego de definido el modelo de random forest, se procede a utilizar *Pipeline*. Un *pipeline* es una secuencia de etapas, y cada etapa puede ser un "Transformer" o un "Estimator". Las etapas se ejecutan en orden y el dataframe de entrada se modifica a medida que pasa por cada etapa. Para las etapas "Transformer", el método *transform()* se llama en el dataframe. Para las etapas "Estimator", se llama al método *fit()* para producir un "Transformer", que se convierte en parte del *PipelineModel* y luego si poder invocar al método *fit()*.

El uso de *pipeline* permite la combinación (encadenamiento) de varios algoritmos en un mismo flujo de trabajo. *Pipeline*, también posee la opción se ser guardado como cualquier otro modelo, lo que simplifica de gran manera el posterior uso de un mismo flujo de trabajo. También permite acceder a cualquiera de los modelos que pertenecen a cada etapa, pudiendo acceder a la información particular de cada uno.

En el ejemplo 4.16, en el *pipeline* se agregan (en orden) todas las etapas por las que tiene que pasar el dataframe y tener al final las predicciones del random forest. Luego para hacer el entrenamiento se utiliza el set de datos *trainingData*, una vez entrenado el modelo, se lo utiliza contra el set de datos *testData* para verificar los resultados.

Por último se guarda en una primera instancia todo el *pipeline* y luego, por separado, se guarda el modelo de random forest, en caso de que, por ejemplo, solo se quiera trabajar con alguna información especial del modelo, como lo es *featureImportances*.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Selecciona (prediction, true label) y computa el error
evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", \
                                              predictionCol="prediction", \
                                              metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```

EJEMPLO 4.17: Evaluación del modelo

Para la evaluación del modelo se utiliza el método *MulticlassClassificationEvaluator*, el cual recibe como parámetro la columna con los datos reales y la columna con la predicción que realizó random forest, compara para cada tupla los valores de las columnas y devuelve el nivel de acierto. Ver ejemplo 4.17.

5 Resultados

Luego de muchos intento se puedo obtener una configuración que permita el procesamiento de los tres meses de datos que se tenía. A raíz de esto se pudo hacer un primer análisis estadístico de la información proveniente de las consultas DNS.

Al realizar un conteo de la cantidad total de consultas por hora y mostrar los resultados en forma de gráficas de barras (ver figura 5.1), claramente se puede apreciar la distinción de la actividad y flujo de datos durante el día y la noche. En donde existe un gran contraste en cuanto a la cantidad de consultas, durante el día en horas del mediodía, se alcanzan picos de hasta más de 800 millones de consultas mientras que en horas de la noche esta cantidad empieza a descender drásticamente hasta estar por debajo de las 100 millones durante la madrugada.

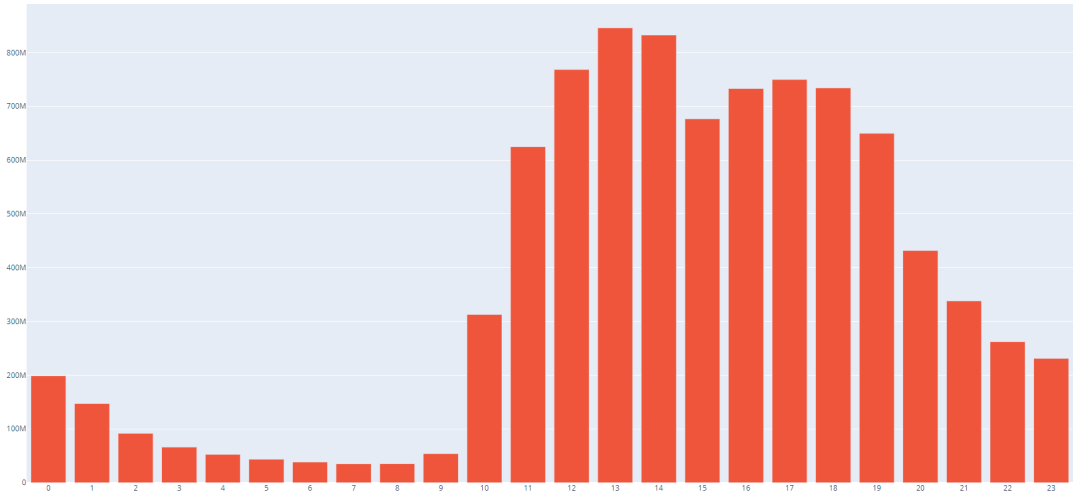

FIGURA 5.1: Total de consultas DNS por hora.

En un segundo paso, se procedió a realizar el mismo análisis pero esta vez separando las consultas pertenecientes a la capital (Montevideo, ver figura 5.2) del interior del país (ver figura 5.3), con la intención de comprobar si existía algún comportamiento diferente. Como resultado se obtuvo que no hubo diferencia alguno y el comportamiento fue el mismo en ambos casos, a excepción de la cantidad de consultas que se hicieron.

Luego se procedió a identificar la categoría a la cual pertenecía la consulta y agruparlas por hora (ver figura 5.4). En un principio se hizo de manera total y luego se procedió a hacer la diferenciación entre Montevideo y el interior. Al igual que el caso anterior, el comportamiento fue muy similar entre el interior y Montevideo. En cuanto a las categorías, era de esperar que hubiera un crecimiento repentino en las horas de pasaje entre la noche y el día, en especial entre las 9 y 11 A.M y en la tarde en las horas 19 y 20 P.M, en donde las cantidades aumentan en algunos casos a más

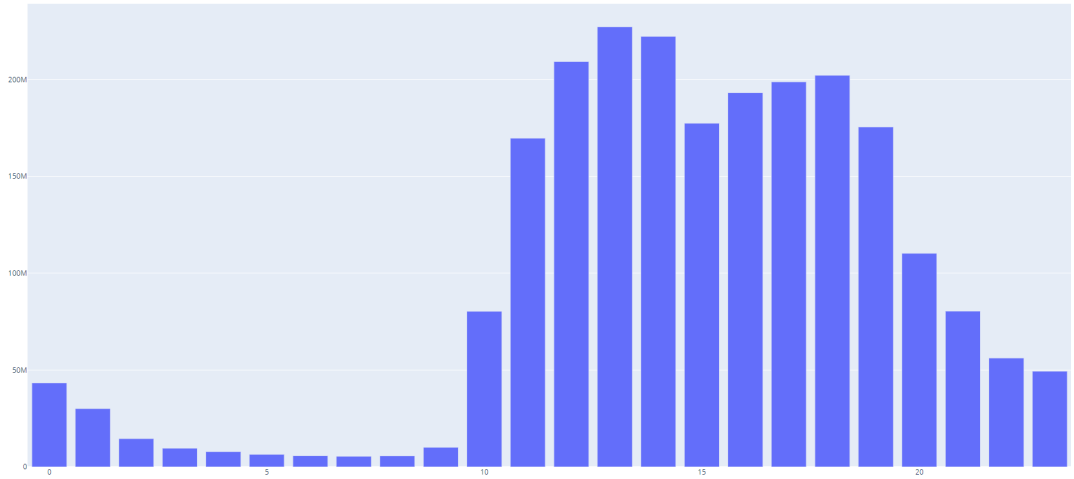

FIGURA 5.2: Total de consultas DNS por hora en Montevideo.

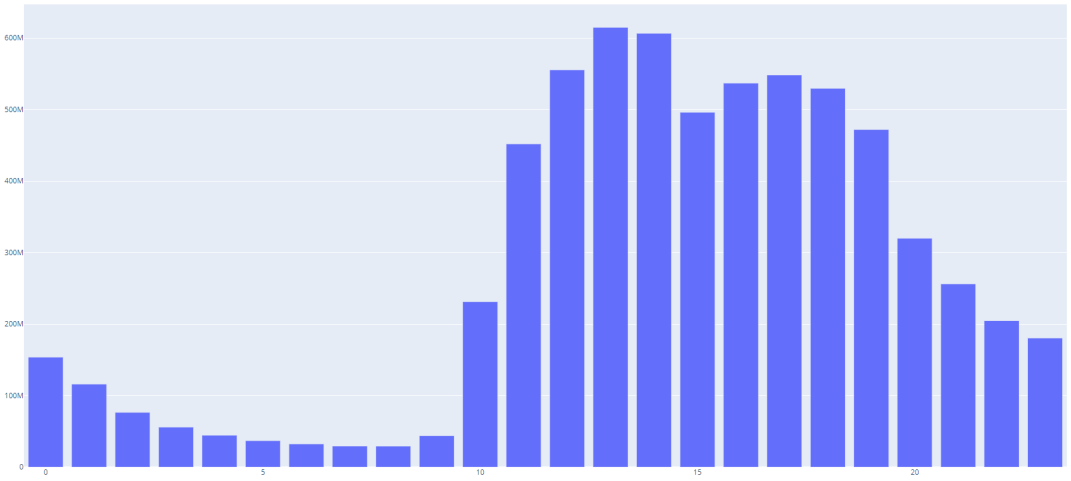

FIGURA 5.3: Total de consultas DNS por hora en el Interior.

del doble en la mañana y disminuyen, de forma más paulatina, en la tarde.

Por otro lado, se llegó a poder ejecutar un análisis de componentes principales de los tres meses almacenados. Al analizar la varianza de cada componente (ver figura 5.5), se puede ver que las dos primeras componentes solo abarcan, aproximadamente, un 30 % del total de la varianza, a partir de incluir la quinta componente se podría pasar el 50 % de la varianza observada, lo cual todavía sigue siendo muy poco, posiblemente el conjunto de features que se eligió no ayuda a representar la realidad.

De todas maneras y continuando con el análisis de PCA, al representar en forma de vectores, el peso que tiene cada feature en los componentes principales (ver figura 5.6), se puede apreciar que la primera componente recoge mayoritariamente la información correspondiente a la mayoría de las categorías (social networking, photo sharing, chat) mientras que la segunda componente recoge la información de las categorías restantes como business services y infrastrucutre. Además existen otras features que no son relevantes, o por lo menos para estas componentes, como lo son el departamento y si la consulta es permitida o bloqueada (action).

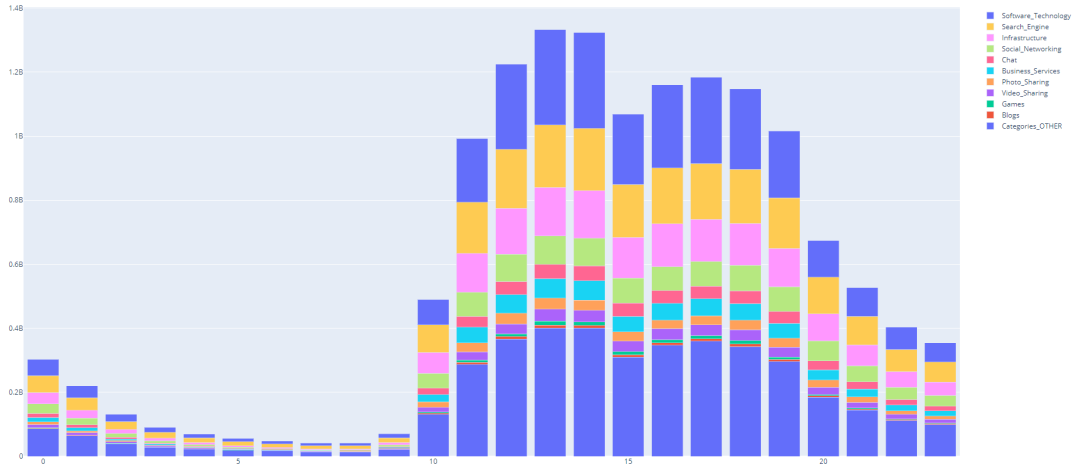

FIGURA 5.4: Total de categorías por hora.

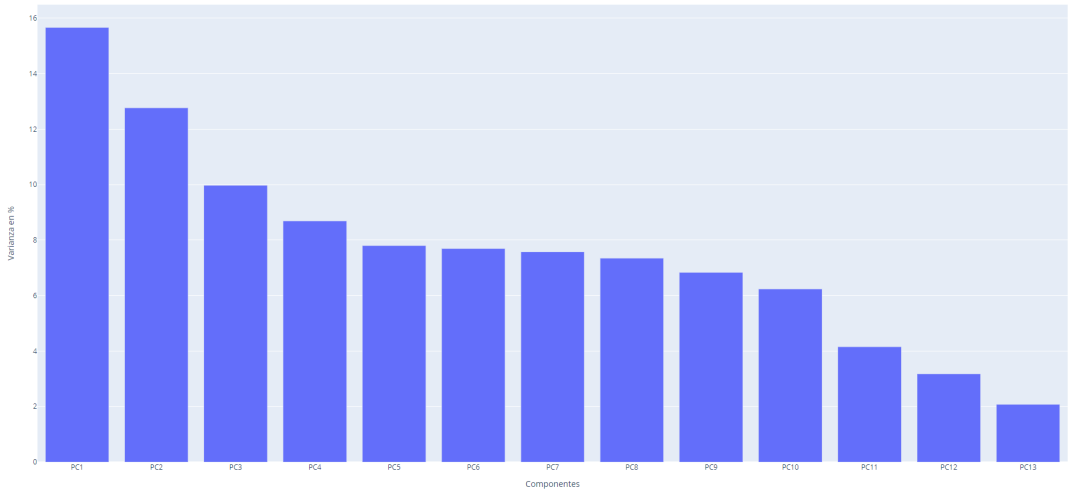

FIGURA 5.5: Porcentaje de varianza de cada componente PCA.

No obstante, y como ya se mencionó anteriormente estos resultados están muy ligados a las features que se seleccionaron como entrada para el algoritmo. Por lo que evidentemente, si mejoramos esa elección de features es muy probable que los resultados sean mejores.

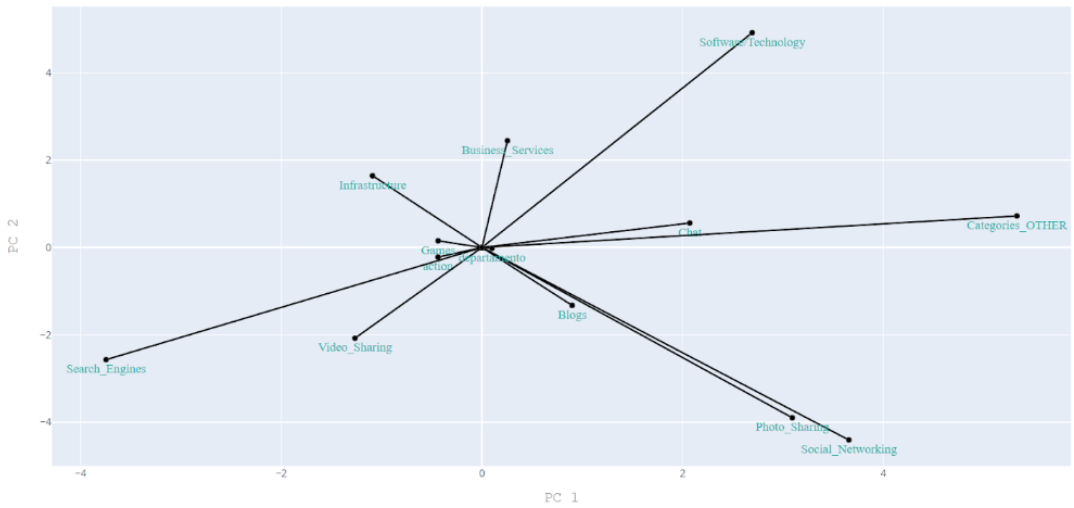

FIGURA 5.6: Representación de vectores de las features en componentes principales.

6 Trabajo futuro

En vista que se pudo encontrar una configuración de parámetros para Spark, que hasta el momento no ha dado problemas y permite procesar los tres meses de datos que se tiene, sería deseable poder cumplir con los objetivos planteados desde un principio.

Empezar con un set de datos de desde cero, seleccionando las features que mejor describan la realidad planteada, para de esta manera poder tener un modelo de datos que pueda ser usado como entrada para los algoritmos correspondientes. Así como también seguir mejorando en la calidad de los datos que se tiene, por ejemplo, completando los datos del contexto socio-cultural, ya que solo se cuenta con esta información solo para una parte de los datos.

Otro punto a trabajar, es el de tener presente la temporalidad de los datos que hasta el momento es una variable que no ha sido tenido en cuenta pero que puede influir en los resultados. Para esto se puede incurrir en el estudio de algoritmos de clustering para series temporales. Así como también hacer agrupaciones de los datos ya sea por hora, minutos, días, etc y ver si esto influye de alguna manera en los resultados.

En cuanto a la plataforma, seguir investigando para poder mejorar los tiempos de ejecución de los algoritmos, probar otras formas de almacenar los datos para también tener una mejor performance, ver como se desempeña Spark si se utiliza librerías que no son propias del componente. Ver como se comporta la versión sandbox de HDP si se agrega otro nodo al cluster.

Bibliografía

[1] Jolliffe IT, Cadima J. *Principal component analysis: a review and recent developments*. Última visita: noviembre 2019. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4792409/.

[2] *T-SNE*. Última visita: diciembre 2019. URL: https://lvdmaaten.github.io/tsne/.

[3] K-Means. *K-Means Clustering: Agrupamiento con Minería de datos*. Última visita: noviembre 2019. URL: https://estrategiastrading.com/k-means/.

[4] *Random Forest*. Última visita: diciembre 2019. URL: https://towardsdatascience.com/understanding-random-forest-58381e0602d2.

[5] Tao SHI and Steve HORVATH. *Unsupervised Learning With Random Forest Predictors*. Última visita: diciembre 2019. URL: https://horvath.genetics.ucla.edu/html/RFclustering/RFclustering/RandomForestHorvath.pdf.

[6] Pandas. *Pandas página principal*. Última visita: noviembre 2019. URL: https://pandas.pydata.org/.

[7] Parquet. *Parquet file página principal*. Última visita: noviembre 2019. URL: http://parquet.apache.org/.

[8] Scikit-Learn. *Scikit-Learn página principal*. Última visita: noviembre 2019. URL: https://scikit-learn.org/.

[9] MLlib. *Machine Learning Library (MLlib) Guide*. Última visita: noviembre 2019. URL: https://spark.apache.org/docs/2.3.4/ml-guide.html.

[10] *Umbrella - CISCO*. Última visita: diciembre 2019. URL: https://umbrella.cisco.com/.

[11] *CISCO*. Última visita: diciembre 2019. URL: https://www.cisco.com/.

[12] Mishra, S., Sarkar, U., Taraphder, S., Datta, S., Swain, D., Saikhom, R. et al. (2017). *Multivariate Statistical Data Analysis- Principal Component Analysis (PCA). International Journal of Livestock Research*. Última visita: noviembre 2019. URL: https://www.researchgate.net/publication/316652806_Principal_Component_Analysis.

[13] Joaquín Amat Rodrigo. *Análisis de Componentes Principales (Principal Component Analysis, PCA) y t-SNE*. Última visita: noviembre 2019. URL: https://www.cienciadedatos.net/documentos/35_principal_component_analysis.

[14] *Silhouette*. Última visita: noviembre 2019. URL: https://kapilddatascience.wordpress.com/2015/11/10/using-silhouette-analysis-for-selecting-the-number-of-cluster-for-k-means-clustering/.

[15] Cloudera. *Cloudera página principal*. Última visita: noviembre 2019. URL: https://www.cloudera.com/.

[16] Hortonworks Data Platform. Última visita: octubre 2019. URL: https://www.cloudera.com/products/hdp.html.

[17] Hortonworks Data Platform. *Getting Started with HDP Sandbox*. Última visita: octubre 2019. URL: https://www.cloudera.com/tutorials/getting-started-with-hdp-sandbox/1.html.

[18] Download Hortonworks Data Platform. Última visita: octubre 2019. URL: https://www.cloudera.com/downloads/hortonworks-sandbox/hdp.html.

[19] Hortonworks Data Platform. *Learning the Ropes of the HDP Sandbox*. Última visita: octubre 2019. URL: https://www.cloudera.com/tutorials/learning-the-ropes-of-the-hdp-sandbox.html.

[20] HDFS. *Hadoop Distributed File System Introduction*. Última visita: noviembre 2019. URL: https://youtu.be/1_ly9dZnmWc.

[21] MapReduce. *Introduction to MapReduce*. Última visita: noviembre 2019. URL: https://youtu.be/ht3dNvdNDzI.

[22] YARN. *Hadoop YARN: Past, present and future*. Última visita: noviembre 2019. URL: https://youtu.be/wlouNFscZS0.

[23] Tutorialspoint. *Apache Spark - Introduction*. Última visita: noviembre 2019. URL: https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm.

[24] Databricks. *Apache Spark and Hadoop: Working Together*. Última visita: noviembre 2019. URL: https://databricks.com/blog/2014/01/21/spark-and-hadoop.html.

[25] Clouder documentation. *Spark Execution Model*. Última visita: noviembre 2019. URL: https://docs.cloudera.com/documentation/enterprise/5-8-x/topics/cdh_ig_spark_apps.html#spark_exec_model.

[26] Cloudera. *Tuning Spark Applications*. Última visita: noviembre 2019. URL: https://docs.cloudera.com/documentation/enterprise/5-8-x/topics/admin_spark_tuning.html#spark_tuning__spark_tuning_resource_allocation.

[27] *Calculating executor memory, number of Executors Cores per executor for a Spark Application*. Última visita: noviembre 2019. URL: http://www.mycloudplace.com/calculating-executor-memory-number-of-executors-cores-per-executor-for-a-spark-application/.

[28] *Distribution of Executors, Cores and Memory for a Spark Application running in Yarn*. Última visita: noviembre 2019. URL: https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html.

[29] Matteo Guzzo. *How to Optimize your Spark Jobs*. Última visita: noviembre 2019. URL: https://matteoguzzo.com/blog/spark-configurator/.

[30] Stefano Meschiari. *Debugging Apache Spark pipelines*. Última visita: noviembre 2019. URL: https://duo.com/labs/tech-notes/debugging-apache-spark-pipelines.

[31] Adir Mashiach. *Apache Spark: 5 Performance Optimization Tips*. Última visita: noviembre 2019. URL: https://medium.com/@adirmashiach/apache-spark-5-performance-optimization-tips-4d85ae7ac0e3.

[32] Cloudera. *Spark setting the Python Path*. Última visita: noviembre 2019. URL: https://docs.cloudera.com/documentation/enterprise/5-9-x/topics/spark_python.html#spark_python__section_ark_lkn_25.

[33] Conda. *Usage with Apache Spark on YARN*. Última visita: noviembre 2019. URL: https://conda.github.io/conda-pack/spark.html.

[34] Spark. *A Neanderthal's Guide to Apache Spark in Python*. Última visita: noviembre 2019. URL: https://towardsdatascience.com/a-neanderthals-guide-to-apache-spark-in-python-9ef1f156d427.

[35] Cloudera. *Using Spark SQL*. Última visita: noviembre 2019. URL: https://docs.cloudera.com/documentation/enterprise/5-5-x/topics/spark_sparksql.html.

[36] Spark. *Spark - Load/Save functions*. Última visita: noviembre 2019. URL: https://spark.apache.org/docs/2.3.4/sql-programming-guide.html#generic-loadsave-functions.

[37] Spark. *Spark PCA documentation*. Última visita: noviembre 2019. URL: https://spark.apache.org/docs/2.4.3/api/python/pyspark.ml.html#pyspark.ml.feature.PCA.

[38] Spark. *Spark k-means documentation*. Última visita: noviembre 2019. URL: https://spark.apache.org/docs/2.4.3/api/python/pyspark.ml.html#pyspark.ml.clustering.KMeans.

[39] Spark. *Spark random forest documentation*. Última visita: noviembre 2019. URL: https://spark.apache.org/docs/2.4.3/api/python/pyspark.ml.html#pyspark.ml.classification.RandomForestClassifier.