Apache Spark II - Machine Learning

Competencias

- Conocer los casos de uso de la librería MLlib y ML.
- Conocer los principales tipos de datos asociados para trabajar con Machine Learning en Spark.
- Implementar algoritmos de Machine Learning en Spark.
- Generar búsqueda de hiper parámetros vía grilla en Spark.

Caveat: *Esta lectura se realizó en modo local (y no desde una instancia de trabajo en AWS EMR) para poder implementar visualizaciones con matplotlib.pyplot y seaborn. Cabe destacar que el código pertinente a pyspark funcionará sin problemas si es que se ejecuta desde un JupyterNotebook en la instancia AWS EMR, la única limitante es que no será posible visualizar gráficos desde este notebook remoto.*

Motivación

Hasta el momento tenemos un conocimiento relativamente rudimentario sobre cómo se comporta Spark a grandes rasgos. También sabemos cómo trabajar con datos mediante RDD y DataFrame de Spark. Resulta que mediante otra de las API de Spark, podemos implementar algoritmos de Machine Learning bajo este paradigma. La API presenta dos modos de trabajo con Machine Learning:

- `pyspark.mllib`: Un framework analítico basado en las Resilient Distributed Datasets (RDD) como unidad de análisis.
- `pyspark.ml`: Un framework analítico basado en los DataFrame como unidad de análisis.

Cabe destacar que `pyspark.mllib` entró en modo de mantención por los desarrolladores de Spark, dado que buscarán priorizar `pyspark.ml` como first-class citizen para la implementación de mejoras posteriores.

Parte de las ventajas de la implementación basada en DataFrame `pyspark.ml` es la naturaleza intuitiva de implementar operaciones en tablas definidas en columnas y filas, por sobre representaciones abstractas basadas en RDD. Esta API también se encarga de proveer un estándar uniforme para todos los algoritmos. Por último, los DataFrame facilitan la implementación de Pipelines para poner en producción los modelos prototipados.

Si bien Apache Spark va hacia la adopción de `pyspark.ml` como norma, gran parte del código circulando en la actualidad está escrito en base a `pyspark.mllib`, por lo que es necesario entender cómo funciona.

Caveat: *Un aspecto importante a destacar es que en la documentación tanto `pyspark.mllib` como `pyspark.ml` se refieren como MLLib. La documentación específica de `pyspark.ml` está en la glosa* ***MLLib: Main Guide*** *y la documentación específica de `pyspark.mllib` está en la glosa* ***MLLib: RDD-Based API Guide****.*

Introducción a `pyspark.mllib` y `pyspark.ml`

`pyspark.mllib` es una librería con funciones orientadas a Machine Learning. Está diseñada para ejecutarse de forma paralela en clusters de trabajos. MLlib es accesible desde múltiples lenguajes y presenta una variedad de algoritmos de aprendizaje, dentro de los que se destacan:

- Algoritmos de clasificación.
- Modelos basados en árboles.
- Algoritmos de regresión.
- Algoritmos de recomendación.
- Algoritmos de clustering.

`pyspark.mllib` se guía por el principio de implementar algoritmos en conjuntos de datos distribuidos que representan todos los datos como RDD. Para operar desde un algoritmo de Machine Learning, se provee de un par de estructuras de datos específicas como `DenseVectors`, `SparseVectors` y `LabeledPoints` que son funciones orientadas a la transformación de los RDD.

Otro aspecto relevante a considerar en la implementación de `pyspark.mllib` es el hecho que **sólo contiene algoritmos paralelizados** optimizados para su ejecución en clusters. Hay algunos algoritmos específicos que no tienen una versión disponible en `pyspark.mllib` dado que no operan bien en plataformas paralelas.

Por lo general, el caso de uso de un algoritmo de `pyspark.mllib` como Gradient Boosting y KMeans es implementarlo en un gran conjunto de datos. Si estamos interesados en implementar algoritmos en muchos conjuntos medianos de datos, por lo general preferiremos utilizar la librería `scikit-learn`. Así, todo se destila al tipo de uso de los datos. Si estamos interesados en generar modelos prototípicos (mediante búsqueda de hiper parámetros vía grilla) para posteriormente llevar a producción, por lo general optaremos por utilizar librerías como `scikit-learn`. Una vez que el mejor modelo está calibrado, podremos implementarlo en `Spark`.

La principal diferencia de `pyspark.ml` respecto a `pyspark.mllib` es el hecho que permite implementar operaciones en objetos `DataFrame`. De manera adicional a la implementación en `DataFrame` para entrenar modelos, la librería moderna pyspark.ml hereda muchas de las buenas prácticas definidas por `scikit-learn` para la resolución de problemas. Así, mientras que `pyspark.mllib` está orientado a entregar un modelo dejando el preprocesamiento y conversión de datos al usuario, `pyspark.ml` se encarga de incorporar elementos de pipelining, limpieza, preprocesamiento, selección de atributos específica.

Tipos de datos en `pyspark.mllib`

Supongamos que estamos interesados en implementar un modelo predictivo de mensajes xenófobos en un portal de noticias. El flujo de trabajo con PySpark quedaría de la siguiente manera:

1. Recopilamos los datos mediante scrapers y generamos un conjunto etiquetado con validadores humanos.

2. Para transformar los comentarios, utilizaremos alguna técnica de extracción de atributos para generar una representación de los textos en una matriz dispersa. Este punto devolverá un RDD de vectores.

3. Implementamos un algoritmo de clasificación en el RDD de vectores, lo cual devolverá un objeto de modelo capaz de clasificar nuevos puntos entregados.

4. Evaluamos el comportamiento del modelo en un conjunto de datos previamente ignorados.

Quizás una de las peculiaridades en la implementación de un algoritmo de Machine Learning en `Spark` es cómo debemos declarar los datos a ingresar. A grandes rasgos, éstos deben ser concatenados. Así, los pasos de extracción de atributos y creación del RDD de vectores se ejemplifica en la siguiente imágen:

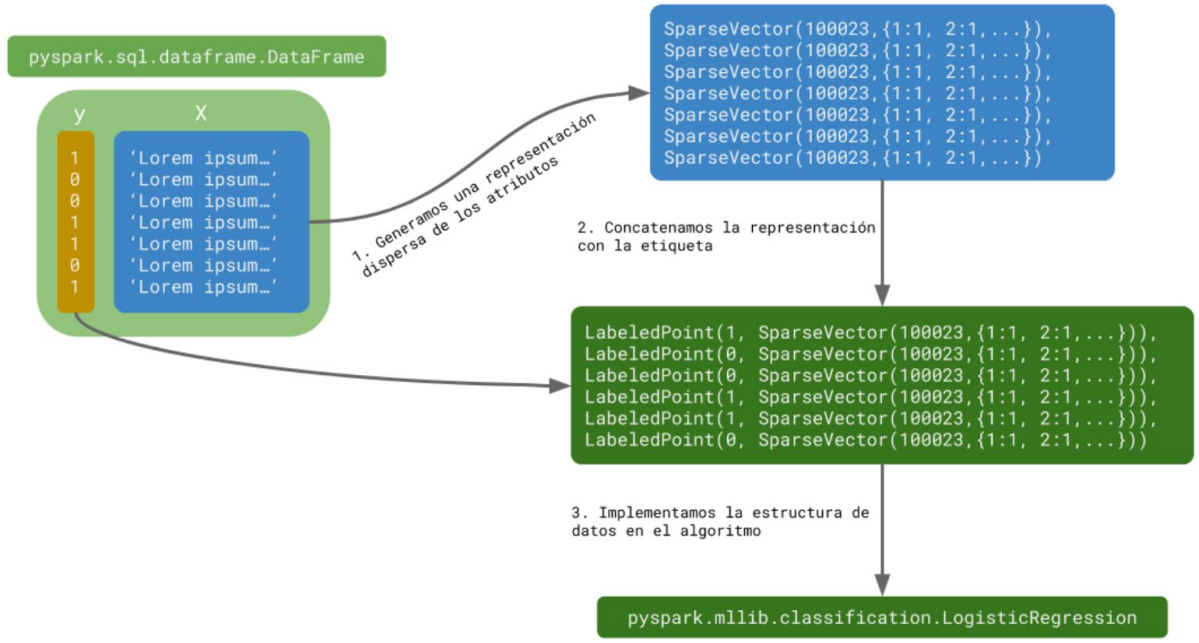

Imagen 1. Pasos de extracción de atributos y creación del RDD.

El primer paso es generar una representación dispersa de los atributos, ya sea mediante algún tokenizador como `CountVectorizer`, `TfIdf` o `HashingTf`. Esta representación dispersa de los atributos debe ser concatenada con su etiqueta asociada. Para ello vamos a utilizar el objeto `LabeledPoint`. Con los datos concatenados, podremos implementar nuestro modelo.

Los tipos de datos específicos para `pyspark.mllib` se pueden categorizar en tres:

1. **`Vector`:** Un objeto Vector es la representación de un vector matemático en un RDD. A grandes rasgos, pyspark.mllib ofrece dos tipos de vectores:
    - **Vectores densos:** Devuelve una lista con todos los elementos dentro del registro. Se considera denso dado que no ignora los datos como 0 o nulos.
    - **Vectores dispersos:** Devuelve un diccionario con todos los elementos dentro del registro que no sean 0. La llave de este diccionario indica la posición del elemento dentro del registro, y el valor asociado es el valor del registro en sí. Cabe destacar que esta representación de los datos tiende a ocupar menos espacio en memoria. Así, `pyspark.mllib` coerciona de manera automática el tipo de `Vector` en función a la eficiencia computacional.

2. **`LabeledPoint`:** Un objeto `LabeledPoint` es la representación de una etiqueta asociada a una serie de registros que tienen sentido a nivel de usuario. Por lo general se implementan en modelos de aprendizaje supervisados como clasificación o regresión. El objeto `LabeledPoint` siempre tendrá un vector de atributos y una etiqueta.

3. **`Rating`:** Para el caso específico de los modelos de recomendación implementados en `pyspark.mllib.recommendation`, existe el tipo de dato Rating. Este tipo de dato puede tomar un rango entre 1-5 o 1-10, donde el valor representa la puntuación asignada.

Encoding específico de los datos

La mayoría de los modelos implementados en Spark requieren que los datos se encuentren en un formato específico. Esto significa que los atributos deben estar representados en una columna del tipo `Vector`. Si estamos entrenando un modelo supervisado, también se requerirá de envolver este vector de atributos en un `LabeledPoint`, donde la etiqueta deberá ser un número flotante.

Spark ML presenta una serie de operaciones orientadas a la preparación de los datos, desde Tokenizadores de texto, Estandarizadores de Atributos y variados esquemas de encoding. Dentro de esta gama de procesadores de datos, probablemente el método `VectorAssembler` es el más común. El objetivo de este es ordenar cada registro con `LabeledPoints` dentro de un RDD de manera intuitiva y fácil.

Práctico: Implementación de un modelo con `pyspark.mllib` y `pyspark.ml`

Para este ejemplo trabajaremos con una base de datos sobre rotación de clientes en una compañía de telecomunicaciones. El archivo contiene 3333 registros y 20 atributos. El vector objetivo a modelar es la tasa de rotación entre los clientes de una compañía de telecomunicaciones `churn`. Los atributos existentes hacen referencia a características de la cuenta de cada cliente.

Lista de atributos

- **State:** Estado de Estados Unidos.
- **Account_Length:** Tiempo en que la cuenta ha sido activada.
- **Area_Code:** Código de área.
- **internationa_plan:** Plan internacional activado.
- **voice_mail_plan:** Plan de mensajes de voz activado.
- **number_vmail_messages:** Cantidad de mensajes de voz.
- **total_day_minutes:** Cantidad de minutos ocupados en la mañana.
- **total_day_calls:** Cantidad de llamadas realizadas en la mañana.
- **total_day_charge:** Cobros realizados en la mañana.
- **total_eve_minutes:** Cantidad de minutos ocupados en la tarde.
- **total_eve_calls:** Cantidad de llamadas realizadas en la tarde.
- **total_eve_charge:** Cobros realizados en la tarde.
- **total_night_calls:** Cantidad de llamadas realizadas en la noche.
- **tota_night_minutes:** Cantidad de minutos ocupados en la noche.
- **total_night_charge:** Cobros realizados en la noche.
- **total_intl_minutes:** Cantidad de minutos ocupados en llamadas internacionales.
- **total_intl_calls:** Cantidad de llamadas internacionales realizadas.
- **total_intl_charge:** Cobros realizados por llamadas internacionales.
- **churn:** 1 si el cliente se cambió de compañía, 0 de los contrario.

Nuestro objetivo va a ser implementar una aplicación de Spark que nos permita predecir la tasa de fuga. A grandes rasgos, podemos esquematizar el flujo de trabajo en los siguientes pasos:

- Generar un objeto `SparkSession` donde podamos importar nuestros datos a un `pyspark.sql.dataframe.DataFrame`.
- Definir el esquema de este conjunto de datos y adjuntarlo.
- Explorar los datos.
- Definir la estrategia de preprocesamiento.
- Implementar un algoritmo clasificador.
- Evaluar el comportamiento en un conjunto de Validación.

Paso 1: Preparación de los datos

Para este ejemplo, vamos a ingresar los datos existentes en el archivo `churn_train.csv`. Nuestro objetivo es generar un objeto `pyspark.sql.dataframe.DataFrame` para su posterior procesamiento. Partamos por realizar los imports básicos y declarar un `SparkSession`, habilitando el soporte para Hive.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from pyspark.sql import SparkSession

# crear objeto SparkSession
spark = SparkSession\
          .builder\
          .master('local[2]')\
          .appName('u5lec1')\
          .enableHiveSupport()\
          .getOrCreate()
```

Con nuestra conexión habilitada, podremos utilizar el método `spark.read.csv` para ingestar los datos. Una vez que los carguemos, verifiquemos el schema de los datos en tabla. Veremos que por defecto todos los atributos dentro del archivo adquirieron el tipo de dato `string`.

```
train_churn = spark\
  .read\
  .csv('churn_train.csv', header=True)
```

```
train_churn.printSchema()
```

```
root
|-- _c0: string (nullable = true)
|-- churn: string (nullable = true)
|-- state: string (nullable = true)
|-- account_length: string (nullable = true)
|-- area_code: string (nullable = true)
|-- international_plan: string (nullable = true)
|-- voice_mail_plan: string (nullable = true)
|-- number_vmail_messages: string (nullable = true)
|-- total_day_minutes: string (nullable = true)
|-- total_day_calls: string (nullable = true)
|-- total_day_charge: string (nullable = true)
|-- total_eve_minutes: string (nullable = true)
|-- total_eve_calls: string (nullable = true)
|-- total_eve_charge: string (nullable = true)
|-- total_night_minutes: string (nullable = true)
|-- total_night_calls: string (nullable = true)
|-- total_night_charge: string (nullable = true)
|-- total_intl_minutes: string (nullable = true)
|-- total_intl_calls: string (nullable = true)
|-- total_intl_charge: string (nullable = true)
|-- number_customer_service_calls: string (nullable = true)
```

Paso 2: Definición explícita del schema de datos

Lamentablemente, en esta situación el schema de los datos no se pudo inferir de una manera correcta, dado que está asumiendo que todos los datos son del tipo string. Nuestro siguiente paso va a ser declarar de manera explícita el schema de la tabla de datos. Para ello, partamos por visualizar cómo se comporta un registro específico.

```
train_churn.show(1)
```

```
+---+-----+-----+--------------+-------------+-----------------+-------
--------+---------------------+-----------------+--------------+-------
---------+-----------------+--------------+----------------+-----------
--------+-----------------+------------------+------------------+-------
---------+-----------------+-----------------------------+
|_c0|churn|state|account_length|
area_code|international_plan|voice_mail_plan|number_vmail_messages|total
_day_minutes|total_day_calls|total_day_charge|total_eve_minutes|total_ev
e_calls|total_eve_charge|total_night_minutes|total_night_calls|total_nig
ht_charge|total_intl_minutes|total_intl_calls|total_intl_charge|number_c
ustomer_service_calls|
+---+-----+-----+--------------+-------------+-----------------+-------
--------+---------------------+-----------------+--------------+-------
---------+-----------------+--------------+----------------+-----------
--------+-----------------+------------------+------------------+-------
---------+-----------------+-----------------------------+
| 0|   no|   KS|           128|area_code_415|               no|
yes|                  25|           265.1|           110|
45.07|           197.4|             99|           16.78|
244.7|               91|            11.01|               10|
3|             2.7|                          1|
+---+-----+-----+--------------+-------------+-----------------+-------
--------+---------------------+-----------------+--------------+-------
---------+-----------------+--------------+----------------+-----------
--------+-----------------+------------------+------------------+-------
---------+-----------------+-----------------------------+
only showing top 1 row
```

Se observa que existe una cantidad no despreciable de datos que deberían ser numéricos pero no fueron reconocidos de manera exitosa por `Spark`. Para ello, vamos a tener que generar el schema de manera explícita.

Para generar el schema de manera explícita, vamos a incorporar una serie de tipos de datos provenientes de `pyspark.sql.types`. La manera de definir el schema de los datos es definir el comportamiento de todos los campos dentro de un registro. Para ello, vamos a utilizar los siguientes tipos de datos:

- **`StructType`**: Permite definir un registro del tipo estructurado. Dentro de éste vamos a definir una serie de campos con `StructField`.
- **`StructField`**: Permite definir un campo específico. Dentro de éste vamos a definir el nombre de la columna y el tipo de dato asociado.
- **`IntegerType`**: Permite definir que el tipo de dato en el campo específico será del tipo entero.
- **`FloatType`**: Permite definir que el tipo de dato en el campo específico será del tipo flotante.
- **`StringType`**: Permite definir que el tipo de dato en el campo específico será del tipo string.

```
from pyspark.sql.types import StructType, StructField, StringType, FloatType,
IntegerType
```

```
schema = StructType([
   StructField('index', IntegerType(), False),
   StructField('churn', StringType(), False),
   StructField('state', StringType(), nullable= False),
   StructField('account_length', IntegerType(), False),
   StructField('area_code', StringType(), False),
   StructField('international_plan', StringType(), False),
   StructField('voice_mail_plan', StringType(), False),
   StructField('number_vmail_messages', IntegerType(), False),
   StructField('total_day_minutes', FloatType(), False),
   StructField('total_day_calls', IntegerType(), False),
   StructField('total_day_charge', FloatType(), False),
   StructField('total_eve_minutes', FloatType(), False),
   StructField('total_eve_calls', IntegerType(), False),
   StructField('total_eve_charge', FloatType(), False),
   StructField('total_night_minutes', FloatType(), False),
   StructField('total_night_calls', IntegerType(), False),
   StructField('total_night_charge',FloatType(), False),
   StructField('total_intl_minutes', FloatType(), False),
   StructField('total_intl_calls', IntegerType(), False),
   StructField('total_intl_charge', FloatType(), False),
   StructField('number_customer_service_calls', IntegerType(), False)
])
```

Volvamos a cargarlos datos, incorporando nuestro schema definido de forma explícita y volvamos a pedir por su representación con printSchema.

```
train_churn = spark\
    .read\
    .csv('churn_train.csv', schema = schema, header=True)\
    .drop('index')

train_churn.printSchema()
```

```
root
 |-- churn: string (nullable = true)
 |-- state: string (nullable = true)
 |-- account_length: integer (nullable = true)
 |-- area_code: string (nullable = true)
 |-- international_plan: string (nullable = true)
 |-- voice_mail_plan: string (nullable = true)
 |-- number_vmail_messages: integer (nullable = true)
 |-- total_day_minutes: float (nullable = true)
 |-- total_day_calls: integer (nullable = true)
 |-- total_day_charge: float (nullable = true)
 |-- total_eve_minutes: float (nullable = true)
 |-- total_eve_calls: integer (nullable = true)
 |-- total_eve_charge: float (nullable = true)
 |-- total_night_minutes: float (nullable = true)
 |-- total_night_calls: integer (nullable = true)
 |-- total_night_charge: float (nullable = true)
 |-- total_intl_minutes: float (nullable = true)
 |-- total_intl_calls: integer (nullable = true)
 |-- total_intl_charge: float (nullable = true)
 |-- number_customer_service_calls: integer (nullable = true)
```

```
train_churn.select('area_code').groupBy('area_code').count().show()
```

```
+-------------+-----+
|    area_code|count|
+-------------+-----+
|area_code_510|  840|
|area_code_415| 1655|
|area_code_408|  838|
+-------------+-----+
```

Paso 3: Exploración de datos

Ya con el schema de datos definido y correctamente inferido, podemos seguir con la exploración de datos. Vamos a generar un reporte gráfico para aproximarnos de forma intuitiva cómo se comportan las columnas componentes de la base de datos. Para aquellas variables que sean nominales, vamos a implementar gráficos de barras con `sns.countplot` y para aquellas variables que sean numéricas (tanto en enteros como flotantes), vamos a implementar gráficos de distribución con `sns.distplot`.

```
rows = 4
cols = np.ceil(len(train_churn.columns) / rows)
plt.rcParams['figure.figsize'] = (16, 10)
for index, (colname, serie) in
enumerate(train_churn.toPandas().iteritems()):
    plt.subplot(rows, cols, index + 1);
    if serie.dtype == 'object':
        sns.countplot(serie);
    else:
        sns.distplot(serie);
    plt.title(colname);
    plt.tight_layout()
```

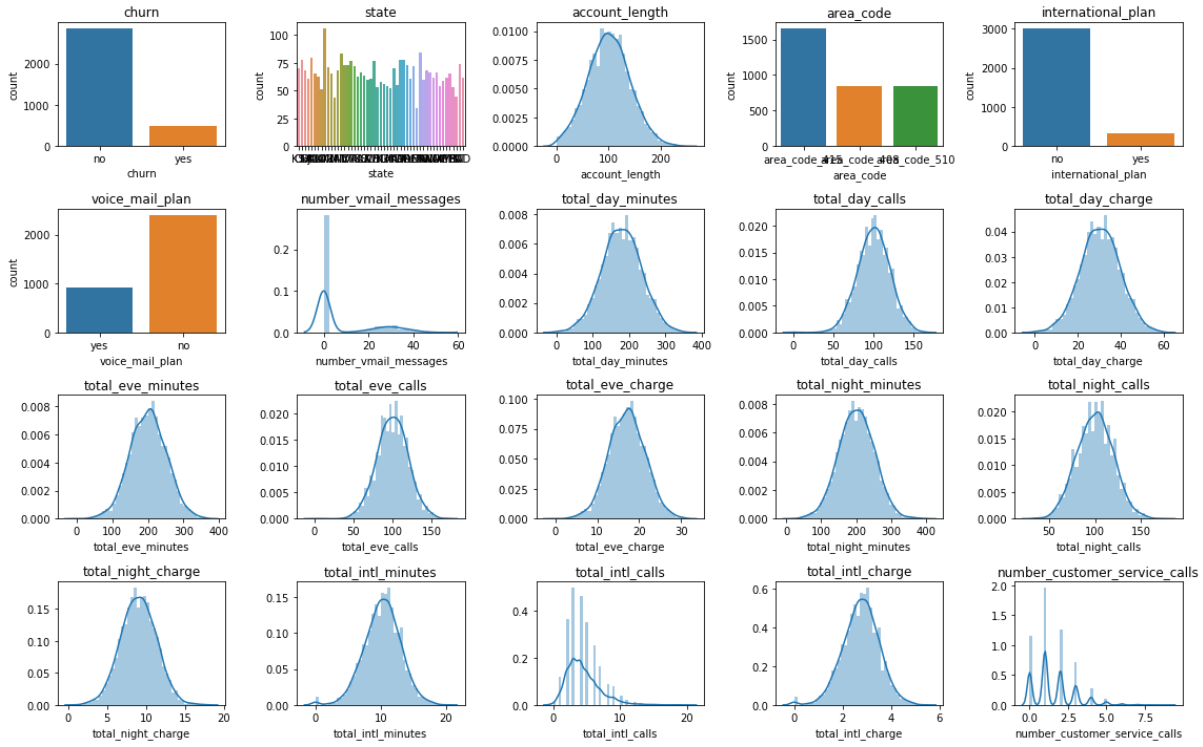

Imagen 2. Gráficos de distribución.

Respecto a la variable `state`, esta representa a cada estado de Estados Unidos, por lo cual deberá ser recodificada en múltiple columnas binarias. Observamos un dominio del `area_code_415` respecto a las demás categorías con las demás variables con aproximadamente 1600 casos. Algo similar encontramos respecto a la variable `international_plan`, donde aproximadamente 3000 individuos optaron por no subscribirse al plan internacional. Un comportamiento similar tiene nuestro vector objetivo `churn`, donde aproximadamente 2500 individuos optaron por migrar de compañía telefónica. Esta última información nos señala sobre el desbalance de clases existente en nuestro problema a resolver.

Las variables con rango contínuo suelen comportarse de forma relativamente normal, salvo con contadas excepciones como `total_intl_calls`, `number_customer_service_calls` y `number_vmail_messages`. Estas tres variables tienden a comportarse con sesgos marcados hacia valores bajos, por lo cual vamos a optar por tomar su logaritmo para re-expresarlo.

Así nuestra estrategia de preprocesamiento va a ser implementar la recodificación de las columnas categóricas `churn`, `state`, `area_code`, `international_plan` y `voice_mail_plan` dado que por defecto son variables con un tipo de dato cadena. De manera adicional, vamos a implementar transformaciones logarítmicas para los vectores `total_intl_calls`, `number_customer_service_calls` y `number_vmail_messages`.

Paso 4: Preprocesamiento

Partamos por recodificar de string a numérico los valores de nuestro vector objetivo `churn`. Para ello, vamos a utilizar la función `when` que se encuentra en el módulo `pyspark.sql.functions`. Esta función opera de una manera similar a como funciona `np.where`. Debemos identificar la condición cuando la columna `churn` tenga el valor de `yes` y asignarle un 1, y 0 para los demás casos. Dado que vamos a sobreescribir la columna `churn`, utilizaremos la función `withColumns` para afectar su comportamiento. Dentro del argumento `train_data.withColumns`, vamos a implementar `when(train_churn['churn'] == 'yes', 1).otherwise(0)`. Posterior a la redefinición de los elementos dentro del `DataFrame`, podemos volver a imprimir el schema, y observaremos que la columna `churn` cambió de string a integer.

```
from pyspark.sql.functions import when
train_churn = train_churn\
    .withColumn('churn', when(train_churn['churn'] == 'yes', 1)\
                .otherwise(0))
```

```
train_churn.printSchema()
```

```
root
 |-- churn: integer (nullable = false)
 |-- state: string (nullable = true)
 |-- account_length: integer (nullable = true)
 |-- area_code: string (nullable = true)
 |-- international_plan: string (nullable = true)
 |-- voice_mail_plan: string (nullable = true)
 |-- number_vmail_messages: integer (nullable = true)
 |-- total_day_minutes: float (nullable = true)
 |-- total_day_calls: integer (nullable = true)
 |-- total_day_charge: float (nullable = true)
 |-- total_eve_minutes: float (nullable = true)
 |-- total_eve_calls: integer (nullable = true)
 |-- total_eve_charge: float (nullable = true)
 |-- total_night_minutes: float (nullable = true)
 |-- total_night_calls: integer (nullable = true)
 |-- total_night_charge: float (nullable = true)
 |-- total_intl_minutes: float (nullable = true)
 |-- total_intl_calls: integer (nullable = true)
 |-- total_intl_charge: float (nullable = true)
 |-- number_customer_service_calls: integer (nullable = true)
```

Ya con nuestro vector objetivo reexpresado a números enteros, repitamos un procedimiento similar para las demás columnas. A grandes rasgos, lo que deseamos es tener una versión `get_dummies` de Pandas, pero para `Spark`. Sabiendo que podemos implementar una operación de recodificación con el procedimiento de arriba, lo que debemos hacer es generar una función que genere $k-1$ categorías. Esta función va a tener la siguiente estructura:

```
#get dummies
def get_dummies_pyspark(df, column):

    tmp_col_categories = df.select(column).distinct().rdd\
                           .flatMap(lambda x: x)\
                           .collect()
    tmp_dummies_expression = [when(col(column) == 1, 1).otherwise(0)\
                              .alias(f"{column}_{str(i)}") for i in
tmp_col_categories]

    return df.select(df.drop(column).columns + tmp_dummies_expression)
```

La primera expresión de la función `tmp_col_categories` se encarga de identificar los valores únicos de la columna. La segunda expresión de la función `tmp_dummies_expression` se encarga de implementar el procedimiento de `when` dentro de una comprensión de lista. Antes de devolver el nuevo dataframe, vamos a eliminar la columna originaria y concatenar ambas partes.Para efectos prácticos, una versión más completa se encuentra en el archivo de funciones auxiliares.

```
import lec5_graphs as afx

train_churn = afx.get_dummies_pyspark(train_churn, 'state')
train_churn = afx.get_dummies_pyspark(train_churn, 'area_code')
train_churn = afx.get_dummies_pyspark(train_churn, 'international_plan')
train_churn = afx.get_dummies_pyspark(train_churn, 'voice_mail_plan')
```

Si revisamos la conformación de columnas, obtendremos algo similar a esta lista.

```
print(train_churn.columns)
```

```
['churn', 'account_length', 'number_vmail_messages',
'total_day_minutes', 'total_day_calls', 'total_day_charge',
'total_eve_minutes', 'total_eve_calls', 'total_eve_charge',
'total_night_minutes', 'total_night_calls', 'total_night_charge',
'total_intl_minutes', 'total_intl_calls', 'total_intl_charge',
'number_customer_service_calls', 'state_SC', 'state_LA', 'state_MN',
'state_NJ', 'state_DC', 'state_OR', 'state_VA', 'state_RI', 'state_WY',
'state_KY', 'state_NH', 'state_MI', 'state_NV', 'state_WI', 'state_ID',
'state_CA', 'state_NE', 'state_CT', 'state_MT', 'state_NC', 'state_VT',
'state_MD', 'state_DE', 'state_MO', 'state_IL', 'state_ME', 'state_WA',
'state_ND', 'state_MS', 'state_AL', 'state_IN', 'state_OH', 'state_TN',
'state_IA', 'state_NM', 'state_PA', 'state_SD', 'state_NY', 'state_TX',
'state_WV', 'state_GA', 'state_MA', 'state_KS', 'state_FL', 'state_CO',
'state_AK', 'state_AR', 'state_OK', 'state_UT', 'state_HI',
'area_code_area_code_415', 'area_code_area_code_408',
'international_plan_yes', 'voice_mail_plan_yes']
```

Respecto a la transformación logarítmica de las columnas, implementaremos un procedimiento similar. Partiremos por sobreescribir los elementos dentro de ésta, donde los nuevos elementos serán la expresión logarítmica de los originales, más una constante de corrección de 0.01.

```
from pyspark.sql.functions import log

train_churn = train_churn.withColumn('number_customer_service_calls',

log(train_churn.number_customer_service_calls + 0.01))
train_churn = train_churn.withColumn('total_intl_calls',
                                     log(train_churn.total_intl_calls +
0.01))
train_churn = train_churn.withColumn('total_eve_charge',
                                     log(train_churn.total_eve_charge +
0.01))
```

Paso 5: Implementación de un algoritmo de clasificación

Ya tenemos nuestra tabla de trabajo lista para implementar un algoritmo. Resulta que vamos a implementar una regresión logística para predecir las tasas de recambio de clientes. En base a este modelo, podremos generar inferencias en nuevos conjuntos de datos.

Para poder implementar un algoritmo supervisado dentro de `Spark`, debemos coercionar los atributos a un tipo de dato `Vector` y asociarlo con una etiqueta específica dento de un tipo de dato `LabeledPoint`. Dado que `pyspark.ml` está orientado al pipelining de procesos, existe la función `pyspark.ml.feature.VectorAssembler` que nos sistematiza este proceso.

Una de las pocas limitantes de trabajar con el método `VectorAssembler` es el hecho que nuestra columna del vector objetivo debe llamarse `'label'`. Partamos por renombrarla.

```
train_churn = train_churn.withColumnRenamed('churn','label')
```

Posterior a este punto, debemos generar una lista con los nombres de todos los atributos que formarán parte de nuestro modelo. Para ello vamos a solicitar `train_churn.columns` y eliminar el nombre del vector objetivo que estará dentro de ésta.

```
feats = train_churn.columns
feats.remove('label')
```

```
print(feats)
```

```
['account_length', 'number_vmail_messages', 'total_day_minutes',
'total_day_calls', 'total_day_charge', 'total_eve_minutes',
'total_eve_calls', 'total_eve_charge', 'total_night_minutes',
'total_night_calls', 'total_night_charge', 'total_intl_minutes',
'total_intl_calls', 'total_intl_charge',
'number_customer_service_calls', 'state_SC', 'state_LA', 'state_MN',
'state_NJ', 'state_DC', 'state_OR', 'state_VA', 'state_RI', 'state_WY',
'state_KY', 'state_NH', 'state_MI', 'state_NV', 'state_WI', 'state_ID',
'state_CA', 'state_NE', 'state_CT', 'state_MT', 'state_NC', 'state_VT',
'state_MD', 'state_DE', 'state_MO', 'state_IL', 'state_ME', 'state_WA',
'state_ND', 'state_MS', 'state_AL', 'state_IN', 'state_OH', 'state_TN',
'state_IA', 'state_NM', 'state_PA', 'state_SD', 'state_NY', 'state_TX',
'state_WV', 'state_GA', 'state_MA', 'state_KS', 'state_FL', 'state_CO',
'state_AK', 'state_AR', 'state_OK', 'state_UT', 'state_HI',
'area_code_area_code_415', 'area_code_area_code_408',
'international_plan_yes', 'voice_mail_plan_yes']
```

Para generar el RDD con `VectorAssembler`, debemos declarar cuál va a ser el rango de atributos en el argumento `inputCols` y el nombre de este vector resultante con outputCol. Posterior a la declaración, debemos ejecutar el método `transform` con nuestro `DataFrame` y seleccionar sólo aquellos elementos de interés, en este caso, `'label'` y `'assembled_features'`.

```
# partamos por importar el método VectorAssembler
from pyspark.ml.feature import VectorAssembler
assemble_feats = VectorAssembler(inputCols = feats,
                                 outputCol = 'assembled_features')
assemble_feats = assemble_feats.transform(train_churn)
assemble_feats = assemble_feats.select('label', 'assembled_features')
# evaluemos la primera representación
assemble_feats.take(1)
```

```
[Row(label=0, assembled_features=SparseVector(69, {0: 128.0, 1: 25.0, 2:
265.1, 3: 110.0, 4: 45.07, 5: 197.4, 6: 99.0, 7: 2.8208, 8: 244.7, 9:
91.0, 10: 11.01, 11: 10.0, 12: 1.1019, 13: 2.7, 14: 0.01}))]
```

Observamos que un registro se va a componer de una etiqueta específica en conjunto a un vector de los atributos. En este ejemplo, los atributos se re-expresaron en un vector disperso `SparseVector` para hacer más eficiente el uso de memoria disponible.

Ya tenemos nuestros elementos de trabajo. El siguiente paso va a ser implementar una separación entre muestras de entrenamiento y validación. A diferencia de la implementación de `sklearn.model_selection.train_test_split`, acá sólo separamos los conjuntos por una razón, ignorando la nomenclatura de `X_train`, `X_test`, `y_train`, `y_test`.

```
train, test = assemble_feats.randomSplit([0.7, 0.3])
```

Con los datos separados entre entrenamiento y validación, el siguiente paso lógico es la implementación de nuestro modelo. Para este caso utilizaremos una regresión logística. Importaremos la regresión logística desde `pyspark.ml.classification.LogisticRegression`.

Resulta que la implementación en `pyspark` difiere a cómo funciona en `sklearn`. Esto se debe en parte al hecho que deberemos indicar de manera explícita por lo menos tres elementos:

- La columna de atributos (`featuresCol`), donde ingresamos el nombre del vector de atributos.
- La columna del vector objetivo (`labelCol`), donde ingresamos el nombre del vector objetivo.
- La columna de predicciones (`predictionCol`), donde ingresamos el nombre de la columna donde se guardarán las predicciones de clase realizadas por el modelo.

```
from pyspark.ml.classification import LogisticRegression
```

```
logistic_example = LogisticRegression(featuresCol='assembled_features',
                                      labelCol='label',
                                      predictionCol='churn_pred')
```

De manera adicional, es en esta etapa donde podemos incorporar hiper parámetros. Este punto lo visitaremos posteriormente. Por ahora, quedémonos en el proceso de entrenamiento. De manera similar a `sklearn`, una vez que instanciamos el objeto (en este caso, `logistic_example`), tendremos una serie de métodos disponibles, como `fit`. Dentro de éste vamos a incluir nuestro conjunto de entrenamiento.

```
logistic_example = logistic_example.fit(train)
```

Si imprimimos el resultado de este objeto, veremos que sólo nos informa el ID único del trabajo, la cantidad de clases a estimar y la cantidad de atributos implementados.

```
logistic_example
```

```
LogisticRegressionModel: uid = LogisticRegression_361e2a4a57c0,
numClasses = 2, numFeatures = 69
```

Para poder ingresar a las métricas existentes dentro del conjunto de entrenamiento, podemos hacer uso del método `summary` dentro del modelo ya entrenado. Para este caso, vamos a crear una función que reporte la exactitud, el área bajo la curva, así como el Recall y la Precisión a nivel de clase.

```
def report_metrics_on_pyspark_ml(model):
    print(f"Overall behavior")
    print(f"Area Under ROC: {model.areaUnderROC}")
    print(f"Accuracy: {model.accuracy}")
    for index, values in enumerate(model.labels):
        print(f"Behavior for class {values}:")
        print(f"\tPrecision {model.precisionByLabel[index]}")
        print(f"\tRecall {model.recallByLabel[index]}")
        print(f"\tTrue Positive Rate
{model.truePositiveRateByLabel[index]}")
        print(f"\tFalse Positive Rate
{model.falsePositiveRateByLabel[index]}")
        print("\n")
```

```
report_metrics_on_pyspark_ml(logistic_example.summary)
```

```
Overall behavior
Area Under ROC: 0.674655870445344
Accuracy: 0.8555460017196904
Behavior for class 0.0:
      Precision 0.8552859618717504
      Recall 0.9989878542510121
      True Positive Rate 0.9989878542510121
      False Positive Rate 0.9542857142857143

    Behavior for class 1.0:
      Precision 0.8888888888888888
      Recall 0.045714285714285714
      True Positive Rate 0.045714285714285714
      False Positive Rate 0.0010121457489878543
```

Observamos que nuestro conjunto de atributos tiene una exactitud general de un .85 en generar clasificaciones correctas, con un desempeño .17 superior a un clasificador aleatorio. A nivel de clases, se observa que la precisión (la tasa de predicciones `1` correctas por sobre todas observaciones etiquetadas como `1`) fue de un .85 y .88 respectivamente.

Ahora, sabemos que no podemos realizar inferencias a partir de este conjunto de datos, para lo cual vamos a evaluar el comportamiento del modelo en el test. Dentro del modelo entrenado se encuentra el método `evaluate`, que permite generar todas las métricas disponibles del método `summary`. Así, vamos a evaluar el desempeño del modelo en estos datos de validación.

```
model_on_test = logistic_example.evaluate(test)
report_metrics_on_pyspark_ml(model_on_test)
```

```
Overall behavior
Area Under ROC: 0.7176536879957334
Accuracy: 0.8768619662363456
Behavior for class 0.0:
      Precision 0.875751503006012
      Recall 1.0
      True Positive Rate 1.0
      False Positive Rate 0.9323308270676691
```

```
      Behavior for class 1.0:
      Precision 1.0
      Recall 0.06766917293233082
      True Positive Rate 0.06766917293233082
      False Positive Rate 0.0
```

Para generar las predicciones en el test, deberemos utilizar el método `transform` en el modelo entrenado, lo cual nos permitirá visualizar el comportamiento del modelo en cuanto a la probabilidad cruda (correspondiente a los log-odds del modelo), la normalizada entre 0 y 1 (mediante la función logística inversa) y la clase más probable.

```
logistic_example.transform(test).show(10)
```

```
+-----+--------------------+--------------------+--------------------+--
--------+
|label|  assembled_features|       rawPrediction|
probability|churn_pred|
+-----+--------------------+--------------------+--------------------+--
--------+
|    0|(69,[0,1,2,3,4,5,...|[2.59140721215207...|[0.93030651082840...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[2.76522670165614...|[0.94076755839399...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[1.88121106700504...|[0.86775017018907...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[2.41650493811128...|[0.91807725983525...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[2.82423667058721...|[0.94397156269610...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[2.34838428794127...|[0.91280571645127...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[2.63707894797509...|[0.93321012844807...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[2.428227945244,-...|[0.91895465340848...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[3.02269415404444...|[0.95358890675639...|
0.0|
|    0|(69,[0,1,2,3,4,5,...|[2.13933378328297...|[0.89466784452350...|
0.0|
+-----+--------------------+--------------------+--------------------+
only showing top 10 rows
```

Finalmente, podemos evaluar el comportamiento del modelo en el conjunto de datos de validación.

```
report_metrics_on_pyspark_ml(logistic_example.evaluate(test))
```

```
Overall behavior
Area Under ROC: 0.7176536879957334
Accuracy: 0.8768619662363456
Behavior for class 0.0:
      Precision 0.875751503006012
      Recall 1.0
      True Positive Rate 1.0
      False Positive Rate 0.9323308270676691

    Behavior for class 1.0:
      Precision 1.0
      Recall 0.06766917293233082
      True Positive Rate 0.06766917293233082
      False Positive Rate 0.0
```

Referencias

- Chambers, Bill; Zaharia, Matei. 2018. Spark, The Definitive Guide: Big Data Processing Made Simple. Sebastopol, CA: O'Reilly Media.

- Karau, Holden; Warren, R. 2017. High Performance Spark: Best practices for scaling and optimizing Apache Spark. Sebastopol, CA: O'Reilly Media.