Spark for Machine Learning

The PySpark has two sub-packages for ML:

- pyspark.mlib: use the unstructured representation of data RDDs and has been deprecated.
- pyspark.ml: based on a structured, tabular representation of data in DataFrames

```
### Create a Spark session on a Spark cluster

# Import the PySpark module
from pyspark.sql import SparkSession

# Create SparkSession object

spark = SparkSession.builder \
                    .master('local[*]') \
                    .appName('test') \
                    .getOrCreate()

# What version of Spark?
# (Might be different to what you saw in the presentation!)
print(spark.version)

# Get the spark context
sc=spark.sparkContext
print(sc.version)
```

```
3.0.1
3.0.1
```

1. Load datasets

```
# Load some airline flight data from a CSV file 'flights.csv'
# The file needs to be uploaded from your local machine to Databricks File Store in advance
# In the dataset, fields are separated by a comma and missing data are denoted by the string 'NA'.

# Data dictionary:
# mon – month (integer between 1 and 12)
# dom – day of month (integer between 1 and 31)
# dow – day of week (integer; 1 = Monday and 7 = Sunday)
# org – origin airport (IATA code)
# mile – distance (miles)
# carrier – carrier (IATA code)
# depart – departure time (decimal hour)
# duration – expected duration (minutes)
# delay – delay (minutes)

# Read the flights dataset
# inferSchema: Infer data types of columns automatically
# nullValue: Deal with missing data
flights_original = spark.read.csv('/FileStore/tables/flights.csv',
                                  sep=',',
                                  header=True,
                                  inferSchema=True,
                                  nullValue='NA')

# Get number of records. The count() method gives the number of records.
print("The data contain %d records." % flights_original.count())

# View the first five records. The show() method displays the first few records.
flights_original.show(5)

# Check column data types. The dtypes attribute gives the column types
print(flights_original.dtypes)
```

```
The data contain 50000 records.
+---+---+---+-------+------+---+----+------+--------+-----+
|mon|dom|dow|carrier|flight|org|mile|depart|duration|delay|
+---+---+---+-------+------+---+----+------+--------+-----+
| 11| 20|  6|     US|    19|JFK|2153|  9.48|     351| null|
|  0| 22|  2|     UA|  1107|ORD| 316| 16.33|      82|   30|
|  2| 20|  4|     UA|   226|SFO| 337|  6.17|      82|   -8|
|  9| 13|  1|     AA|   419|ORD|1236| 10.33|     195|   -5|
|  4|  2|  5|     AA|   325|ORD| 258|  8.92|      65| null|
+---+---+---+-------+------+---+----+------+--------+-----+
only showing top 5 rows

[('mon', 'int'), ('dom', 'int'), ('dow', 'int'), ('carrier', 'string'), ('flight', 'int'), ('org', 'string'), ('mile', 'int'), ('depart', 'double'), ('duration', 'i
nt'), ('delay', 'int')]
```

```
# The sms.csv dataset contains a selection of SMS messages which have been classified as
# either 'spam' (1) or 'ham' (0)
# Notes on CSV format:
# no header record and
# fields are separated by a semicolon (this is not the default separator of ',')

# Data dictionary:
# id – record identifier
# text – content of SMS message
# label – spam or ham (integer; 0 = ham and 1 = spam)

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

# Specify column names and types
# Specify the data schema, giving columns names ("id", "text", and "label") and column types.
schema = StructType([
    StructField("id", IntegerType()),
    StructField("text", StringType()),
    StructField("label", IntegerType())
])

# Load data from a delimited file
sms_original = spark.read.csv("/FileStore/tables/sms.csv", sep=';', header=False, schema=schema)

# Print schema of DataFrame
sms_original.printSchema()
```

```
root
 |-- id: integer (nullable = true)
 |-- text: string (nullable = true)
 |-- label: integer (nullable = true)
```

2. Data pre-processing

```
# We are going to develop a model which will predict whether or not a given flight will be delayed.

# Firstly we need to trim those data down by:
# - removing an uninformative column and
# - removing rows which do not have information about whether or not a flight was delayed.

# Remove the 'flight' column which is irrelevant for prediction
# The drop() method applies to columns only.
flights_drop_column = flights_original.drop('flight')

# Number of records with missing 'delay' values
# Use the filter() method to choose specific rows and
# the count() method to find the number of rows in the result.
flights_drop_column.filter('delay IS NULL').count()

# Remove records with missing 'delay' values
flights_valid_delay = flights_drop_column.filter('delay IS NOT NULL')

# Remove records with missing values in any column and get the number of remaining rows
# The dropna() method will discard all records with any missing fields.
flights_none_missing = flights_valid_delay.dropna()
print("Number of records without any missing values: ", flights_none_missing.count())
```

```
Number of records without any missing values:  47022
```

```
# The next step of preparing the flight data has two parts:
# - convert the units of distance, replacing the mile column with a kmcolumn; and
# - create a Boolean column indicating whether or not a flight was delayed.

# Import the required function
from pyspark.sql.functions import round

# Convert 'mile' to 'km' and drop 'mile' column
# Use the withColumn() method to manipulate columns.
flights_adding_km = flights_none_missing.withColumn('km',
                                             round(flights_original.mile * 1.60934, 0)).drop('mile')

# Create 'label' column indicating whether flight delayed (1) or not (0)
# a flight to be "delayed" when it arrives 15 minutes or more after its scheduled time.
flights_adding_km = flights_adding_km.withColumn('label', (flights_adding_km.delay >= 15).cast('integer'))

# Check first five records
flights_adding_km.show(5)
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|
+---+---+---+-------+---+------+--------+-----+------+-----+
|  0| 22|  2|     UA|ORD| 16.33|      82|   30| 509.0|    1|
|  2| 20|  4|     UA|SFO|  6.17|      82|   -8| 542.0|    0|
|  9| 13|  1|     AA|ORD| 10.33|     195|   -5|1989.0|    0|
|  5|  2|  1|     UA|SFO|  7.98|     102|    2| 885.0|    0|
|  7|  2|  6|     AA|ORD| 10.83|     135|   54|1180.0|    1|
+---+---+---+-------+---+------+--------+-----+------+-----+
only showing top 5 rows
```

```python
# In the flights data there are two columns, carrier and org, which hold categorical data.
# You need to transform those columns into indexed numerical values.
# Machine Learning model needs numbers not strings, so these transformations are vital!

# Import the appropriate class and create an indexer object to
# transform the carrier column from a string to an numeric index
from pyspark.ml.feature import StringIndexer

# Create an indexer; Prepare the indexer object on the flight data.
indexer = StringIndexer(inputCol='carrier', outputCol='carrier_idx')

# Indexer identifies categories in the data
indexer_model = indexer.fit(flights_adding_km)

# Indexer creates a new column with numeric index values
flights_indexed = indexer_model.transform(flights_adding_km)

# Repeat the process for the other categorical feature
flights_indexed = StringIndexer(inputCol='org', outputCol='org_idx').fit(flights_indexed).transform(flights_indexed)

print(flights_indexed.show(5))
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
|  0| 22|  2|     UA|ORD| 16.33|      82|   30| 509.0|    1|        0.0|    0.0|
|  2| 20|  4|     UA|SFO|  6.17|      82|   -8| 542.0|    0|        0.0|    1.0|
|  9| 13|  1|     AA|ORD| 10.33|     195|   -5|1989.0|    0|        1.0|    0.0|
|  5|  2|  1|     UA|SFO|  7.98|     102|    2| 885.0|    0|        0.0|    1.0|
|  7|  2|  6|     AA|ORD| 10.83|     135|   54|1180.0|    1|        1.0|    0.0|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
only showing top 5 rows

None
```

```python
# The final stage of data preparation is to consolidate all of the predictor columns into a single column.
# An updated version of the flights data, which takes into account all of the changes,
# has the following predictor columns:
# - mon, dom and dow
# - carrier_idx (indexed value from carrier)
# - org_idx (indexed value from org)
# - km
# - depart
# - duration

# Import the class which will assemble the predictors.
from pyspark.ml.feature import VectorAssembler

# Create an assembler object that can merge the predictors columns into a single column.
assembler = VectorAssembler(inputCols=[
    'mon', 'dom', 'dow', 'carrier_idx', 'org_idx', 'km', 'depart', 'duration'
], outputCol='features')

# Consolidate predictor columns
flights_assembled = assembler.transform(flights_indexed)

# Check the resulting column
flights_assembled.select('features', 'delay').show(5, truncate=False)

flights_assembled.show(5)
```

```
+-----------------------------------------+-----+
|features                                 |delay|
+-----------------------------------------+-----+
|[0.0,22.0,2.0,0.0,0.0,509.0,16.33,82.0]  |30   |
|[2.0,20.0,4.0,0.0,1.0,542.0,6.17,82.0]   |-8   |
|[9.0,13.0,1.0,1.0,0.0,1989.0,10.33,195.0]|-5   |
|[5.0,2.0,1.0,0.0,1.0,885.0,7.98,102.0]   |2    |
|[7.0,2.0,6.0,1.0,0.0,1180.0,10.83,135.0] |54   |
+-----------------------------------------+-----+
only showing top 5 rows

+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|            features|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
|  0| 22|  2|     UA|ORD| 16.33|      82|   30| 509.0|    1|        0.0|    0.0|[0.0,22.0,2.0,0.0...|
|  2| 20|  4|     UA|SFO|  6.17|      82|   -8| 542.0|    0|        0.0|    1.0|[2.0,20.0,4.0,0.0...|
|  9| 13|  1|     AA|ORD| 10.33|     195|   -5|1989.0|    0|        1.0|    0.0|[9.0,13.0,1.0,1.0...|
|  5|  2|  1|     UA|SFO|  7.98|     102|    2| 885.0|    0|        0.0|    1.0|[5.0,2.0,1.0,0.0,...|
|  7|  2|  6|     AA|ORD| 10.83|     135|   54|1180.0|    1|        1.0|    0.0|[7.0,2.0,6.0,1.0,...|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
only showing top 5 rows
```

3. Classification using Decision Tree

```
# We will split the data into two components:
# - training data (used to train the model) and
# - testing data (used to test the model).

# Split into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_assembled.randomSplit([0.8, 0.2], seed=17)

print(flights_train.show(5))

# Check that training set has around 80% of records
training_ratio = flights_train.count() / flights_assembled.count()
print(training_ratio)
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|            features|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
|  0|  1|  2|     AA|JFK|   7.0|     385|  -16|4162.0|    0|        1.0|    2.0|[0.0,1.0,2.0,1.0,...|
|  0|  1|  2|     AA|JFK|  12.0|     370|   11|3983.0|    0|        1.0|    2.0|[0.0,1.0,2.0,1.0,...|
|  0|  1|  2|     AA|JFK|  17.0|     379|  -10|3983.0|    0|        1.0|    2.0|[0.0,1.0,2.0,1.0,...|
|  0|  1|  2|     AA|LGA|   6.5|     240|   40|2235.0|    1|        1.0|    3.0|[0.0,1.0,2.0,1.0,...|
|  0|  1|  2|     AA|LGA|  8.25|     250|   27|2235.0|    1|        1.0|    3.0|[0.0,1.0,2.0,1.0,...|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
only showing top 5 rows

None
0.796967376972481
```

```
# Import the Decision Tree Classifier class
from pyspark.ml.classification import DecisionTreeClassifier

# Create a classifier object and fit to the training data
tree = DecisionTreeClassifier()
tree_model = tree.fit(flights_train)

# Create predictions for the testing data and take a look at the predictions
prediction = tree_model.transform(flights_test)
prediction.select('label', 'prediction', 'probability').show(5, False)
```

```
+-----+----------+----------------------------------------+
|label|prediction|probability                             |
+-----+----------+----------------------------------------+
|1    |0.0       |[0.53474762253109,0.46525237746891]     |
|0    |1.0       |[0.3528352276256133,0.6471647723743866]|
|0    |0.0       |[0.53474762253109,0.46525237746891]     |
|1    |1.0       |[0.3528352276256133,0.6471647723743866]|
|1    |1.0       |[0.3528352276256133,0.6471647723743866]|
+-----+----------+----------------------------------------+
only showing top 5 rows
```

```
# Evaluate the decision tree using confusion matrix
# A confusion matrix gives a useful breakdown of predictions versus known values.
# It has four cells which represent the counts of:
# - True Negatives (TN): model predicts negative outcome & known outcome is negative
# - True Positives (TP): model predicts positive outcome & known outcome is positive
# - False Negatives (FN): model predicts negative outcome but known outcome is positive
# - False Positives (FP): model predicts positive outcome but known outcome is negative.


# Create a confusion matrix by counting the combinations of label and prediction. Display the result.
prediction.groupBy('label', 'prediction').count().show()

# Count # of True Negatives, True Positives, False Negatives and False Positives in confusion matrix
# Use the predicatea:
# - prediction = 0 AND label = prediction (TF)
# - prediction = 1 AND label = prediction (TP)
# - prediction = 0 AND label != prediction (FN)
# - prediction = 1 AND label != prediction (FP)
TN = prediction.filter('prediction = 0 AND label = prediction').count()
TP = prediction.filter('prediction = 1 AND label = prediction').count()
FN = prediction.filter('prediction = 0 AND label != prediction').count()
FP = prediction.filter('prediction = 1 AND label != prediction').count()

# Accuracy measures the proportion of correct predictions
# The accuracy is the ratio of correct predictions (TP and TN) to all predictions (TP, TN, FP and FN)
accuracy = (TN + TP) / (TN + TP + FN + FP)
print(accuracy)
```

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    1|       0.0| 1487|
|    0|       0.0| 2622|
|    1|       1.0| 3423|
|    0|       1.0| 2015|
+-----+----------+-----+

0.6331831989106526
```

4. Classification using Logistic regression

```
### Use logistic regression to predict whether a flight is likely to be delayed by
# at least 15 minutes (label 1) or not (label 0).
# Import the logistic regression class
from pyspark.ml.classification import LogisticRegression

# Create a classifier object and train on training data
logistic = LogisticRegression().fit(flights_train)

print(flights_train.show(5))

# Create predictions for the testing data and show confusion matrix
prediction = logistic.transform(flights_test)
prediction.groupBy('label', 'prediction').count().show()
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|            features|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
|  0|  1|  2|     AA|JFK|   7.0|     385|  -16|4162.0|    0|        1.0|    2.0|[0.0,1.0,2.0,1.0,...|
|  0|  1|  2|     AA|JFK|  12.0|     370|   11|3983.0|    0|        1.0|    2.0|[0.0,1.0,2.0,1.0,...|
|  0|  1|  2|     AA|JFK|  17.0|     379|  -10|3983.0|    0|        1.0|    2.0|[0.0,1.0,2.0,1.0,...|
|  0|  1|  2|     AA|LGA|   6.5|     240|   40|2235.0|    1|        1.0|    3.0|[0.0,1.0,2.0,1.0,...|
|  0|  1|  2|     AA|LGA|  8.25|     250|   27|2235.0|    1|        1.0|    3.0|[0.0,1.0,2.0,1.0,...|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------------------+
only showing top 5 rows

None
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    1|       0.0| 1671|
|    0|       0.0| 2522|
|    1|       1.0| 3239|
|    0|       1.0| 2115|
+-----+----------+-----+
```

```
## Evaluate the logistic regression model.
# Accuracy is generally not a very reliable metric because it can be biased by the most common target class
# There are two other useful metrics:
# - Precision is the proportion of positive predictions which are correct, ie. TP/(TP+FP)
# - Recall is the proportion of positives outcomes which are correctly predicted, ie. TP/(TP+FN)

from pyspark.ml.evaluation import MulticlassClassificationEvaluator, BinaryClassificationEvaluator

# Calculate precision and recall
precision = TP / (TP + FP)
recall = TP / (TP + FN)
print('precision = {:.2f}\nrecall    = {:.2f}'.format(precision, recall))

# Find weighted precision.
# The weighted precision indicates what proportion of predictions (positive and negative) are correct.
# Create a multi-class evaluator and evaluate weighted precision.
# The metric name is "weightedPrecision".
multi_evaluator = MulticlassClassificationEvaluator()
weighted_precision = multi_evaluator.evaluate(prediction, {multi_evaluator.metricName: "weightedPrecision"})

# Find AUC
# Create a binary evaluator and evaluate AUC using the "areaUnderROC" metric.
binary_evaluator = BinaryClassificationEvaluator()
auc = binary_evaluator.evaluate(prediction, {binary_evaluator.metricName: "areaUnderROC"})
```

```
precision = 0.63
recall    = 0.70
```

```
## Another example of classification using logistic regression for the sms dataset

# Firstly to repare the SMS messages as follows:
# - remove punctuation and numbers
# - tokenize (split into individual words)
# - remove stop words
# - apply the hashing trick
# - convert to TF-IDF representation.

# Import the necessary functions
from pyspark.sql.functions import regexp_replace
from pyspark.ml.feature import Tokenizer

# Use regular expressions (or REGEX) to remove the punctuation symbols.
# Replace all punctuation characters from the text column with a space.
# Do the same for all numbers in the text column.
wrangled = sms_original.withColumn('text', regexp_replace(sms_original.text, '[_():;,.!?\\-]', ' '))
wrangled = wrangled.withColumn('text', regexp_replace(wrangled.text, '[0-9]', ' '))

# Merge multiple spaces
sms_cleaned = wrangled.withColumn('text', regexp_replace(wrangled.text, ' +', ' '))

# Split the text into words
# Split the 'text' column into tokens. Name the output column 'words'
sms_tokenized = Tokenizer(inputCol='text', outputCol='words').transform(sms_cleaned)

sms_tokenized.show(4, truncate=False)
```

```
+---+----------------------------------+-----+------------------------------------------+
|id |text                              |label|words                                     |
```

```
+---+----------------------------------+-----+------------------------------------------+
|1  |Sorry I'll call later in meeting  |0    |[sorry, i'll, call, later, in, meeting]   |
|2  |Dont worry I guess he's busy      |0    |[dont, worry, i, guess, he's, busy]       |
|3  |Call FREEPHONE now                |1    |[call, freephone, now]                    |
|4  |Win a cash prize or a prize worth |1    |[win, a, cash, prize, or, a, prize, worth]|
+---+----------------------------------+-----+------------------------------------------+
only showing top 4 rows
```

```
from pyspark.ml.feature import StopWordsRemover, HashingTF, IDF

# Remove stop words - to eliminate so commonly used words that carry very little useful info.
# StopWordsRemover class contains a list of stop words which can be customized if necessary.
sms_without_stop = StopWordsRemover(inputCol='words', outputCol='terms').transform(sms_tokenized)

# Apply the hashing trick
# The hashing trick provides a fast and space-efficient way to
# map a very large (possibly infinite) set of items (in this case, all words contained in the SMS messages)
# onto a smaller, finite number of values.
sms_hashed = HashingTF(inputCol='terms', outputCol='hash', numFeatures=1024).transform(sms_without_stop)

# Convert hashed symbols to TF-IDF representation
# The TF-IDF matrix reflects how important a word is to each document.
# It takes into account both the frequency of the word within each document but also
# the frequency of the word across all of the documents in the collection.
# ie. Weight the number of counts for a word in a particular document against
# how frequently that word occurs across all documents

sms_tfidf = IDF(inputCol='hash', outputCol='features').fit(sms_hashed).transform(sms_hashed)

sms_tfidf.select('terms', 'features').show(4, truncate=False)
```

```
+--------------------------------+----------------------------------------------------------------------------------------------------+
|terms                           |features                                                                                            |
+--------------------------------+----------------------------------------------------------------------------------------------------+
|[sorry, call, later, meeting]   |(1024,[138,384,577,996],[2.273418200008753,3.6288353225642043,3.5890949939146903,4.104259019279279])|
|[dont, worry, guess, busy]      |(1024,[215,233,276,329],[3.9913186080986836,3.3790235241678332,4.734227298217693,4.58299632849377]) |
|[call, freephone]               |(1024,[133,138],[5.367951058306837,2.273418200008753])                                              |
|[win, cash, prize, prize, worth]|(1024,[31,47,62,389],[3.6632029660684124,4.754846585420428,4.072170704727778,7.064594791043114])    |
+--------------------------------+----------------------------------------------------------------------------------------------------+
only showing top 4 rows
```

```
# Split the tf_idf data into training and testing sets in a 4:1 ratio
sms_train, sms_test = sms_tfidf.randomSplit([0.8, 0.2], seed=13)

# Fit a Logistic Regression model to the training data
logistic = LogisticRegression(regParam=0.2).fit(sms_train)

# Make predictions on the testing data
prediction = logistic.transform(sms_test)

# Create a confusion matrix, comparing predictions to known labels
prediction.groupBy('label', 'prediction').count().show()
```

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    1|       0.0|   41|
|    0|       0.0|  948|
|    1|       1.0|  105|
|    0|       1.0|    2|
+-----+----------+-----+
```

5. Regression

```python
# One-hot encoding is a common way of preprocessing categorical variable for machine learning models
# In one-hot encoding, each category value is converted into a new column and
# assigned a 1 or 0 (notation for true/false) value to the column

# Import the one hot encoder class
from pyspark.ml.feature import OneHotEncoder

# The 'org' column in the flights data is a categorical variable giving the airport from which a flight departs.
# Since this is a categorical variable, it needs to be one-hot encoded before it can be used in a regression model.
# Create an one-hot encoder instance, naming the output column 'org_dummy'.
onehot = OneHotEncoder(inputCols=['org_idx'], outputCols=['org_dummy'])

print(flights_indexed.show(5))

# Apply the one hot encoder to the flights data
onehot = onehot.fit(flights_indexed)
flights_onehot = onehot.transform(flights_indexed)

# Generate a summary of the mapping from categorical values to binary encoded dummy variables
flights_onehot.select('org', 'org_idx', 'org_dummy').distinct().sort('org_idx').show()
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
|  0| 22|  2|     UA|ORD| 16.33|      82|   30| 509.0|    1|        0.0|    0.0|
|  2| 20|  4|     UA|SFO|  6.17|      82|   -8| 542.0|    0|        0.0|    1.0|
|  9| 13|  1|     AA|ORD| 10.33|     195|   -5|1989.0|    0|        1.0|    0.0|
|  5|  2|  1|     UA|SFO|  7.98|     102|    2| 885.0|    0|        0.0|    1.0|
|  7|  2|  6|     AA|ORD| 10.83|     135|   54|1180.0|    1|        1.0|    0.0|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
only showing top 5 rows

None
+---+-------+-------------+
|org|org_idx|    org_dummy|
+---+-------+-------------+
|ORD|    0.0|(7,[0],[1.0])|
|SFO|    1.0|(7,[1],[1.0])|
|JFK|    2.0|(7,[2],[1.0])|
|LGA|    3.0|(7,[3],[1.0])|
|SMF|    4.0|(7,[4],[1.0])|
|SJC|    5.0|(7,[5],[1.0])|
```

```python
# Build a regression model to predict flight duration (the 'duration' column).
# To keep the model simple, we include only the distance of the flight (the 'km' column) as a predictor.

# Import the class which will assemble the predictors.
from pyspark.ml.feature import VectorAssembler

# Create an assembler object that can merge the predictors columns into a single column.
assembler = VectorAssembler(inputCols=['km'], outputCol='features')

# Consolidate predictor columns
flights_assembled = assembler.transform(flights_indexed)

# Split into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_assembled.randomSplit([0.8, 0.2], seed=17)

print(flights_train.show(5))

from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Create a regression object and train on training data (by calling 'fit' function)
regression = LinearRegression(labelCol='duration').fit(flights_train)

# Create predictions for the testing data (by calling the 'transform' function)
# and take a look at the predictions
predictions = regression.transform(flights_test)
predictions.select('km', 'duration', 'prediction').show(5, False)

# Calculate the RMSE (Root Mean Squared Error) which corresponds to the standard deviation of the residuals
RMSE = RegressionEvaluator(labelCol='duration').evaluate(predictions)
print('RMSE: ', RMSE)

# Intercept (average minutes on ground)
inter = regression.intercept
print('Intercept: ', inter)

# Coefficients
coefs = regression.coefficients
print('Coefficient: ', coefs[0])

# Average minutes per km
minutes_per_km = regression.coefficients[0]
print('Average minutes per km: ', minutes_per_km)

# Average speed in km per hour
avg_speed = 60 / minutes_per_km
print('Average speed in km/hour: ', avg_speed)
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|features|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------+
```

```
|  0|  1|  2|     AA|JFK|   7.0|     385|  -16|4162.0|    0|        1.0|    2.0|[4162.0]|
|  0|  1|  2|     AA|JFK|  12.0|     370|   11|3983.0|    0|        1.0|    2.0|[3983.0]|
|  0|  1|  2|     AA|JFK|  17.0|     379|  -10|3983.0|    0|        1.0|    2.0|[3983.0]|
|  0|  1|  2|     AA|LGA|   6.5|     240|   40|2235.0|    1|        1.0|    3.0|[2235.0]|
|  0|  1|  2|     AA|LGA|  8.25|     250|   27|2235.0|    1|        1.0|    3.0|[2235.0]|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+--------+
only showing top 5 rows

None
+------+--------+------------------+
|km    |duration|prediction        |
+------+--------+------------------+
|2570.0|230     |238.54045166717412|
|1180.0|170     |133.41009501509737|
|1180.0|120     |133.41009501509737|
|1180.0|135     |133.41009501509737|
|415.0 |70      |75.55058218140046 |
+------+--------+------------------+
```

```python
# This time the regression model the duration of a flight is extended - it depends
# not only on the distance being covered ('km') but also the airport from which the flight departs ('org_dummy').

# Import the class which will assemble the predictors.
from pyspark.ml.feature import VectorAssembler

# Create an assembler object that can merge the predictors columns into a single column.
assembler = VectorAssembler(inputCols=['km', 'org_dummy'], outputCol='features')

# Consolidate predictor columns
flights_assembled = assembler.transform(flights_onehot)

# Split into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_assembled.randomSplit([0.8, 0.2], seed=17)

print(flights_train.show(5))


from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Create a regression object and train on training data
regression = LinearRegression(labelCol='duration').fit(flights_train)

# Create predictions for the testing data
predictions = regression.transform(flights_test)
predictions.select('km', 'org_dummy','duration', 'prediction').show(5, False)

# Calculate the RMSE (Root Mean Squared Error) which corresponds to the standard deviation of the residuals
RMSE = RegressionEvaluator(labelCol='duration').evaluate(predictions)
print('RMSE: ', RMSE)

# Calculate average speed in km per hour
# The first coefficient, regression.coefficients[0], is in minutes per km
avg_speed_hour = 60 / regression.coefficients[0]
print('Agerage speed in km/hour: ', avg_speed_hour)

# Average minutes on ground at OGG
inter = regression.intercept
print('Intercept: ', inter)

# Average minutes on ground at JFK
avg_ground_jfk = inter + regression.coefficients[3]
print('Average minutes on groud at JFK airport: ', avg_ground_jfk)

# Average minutes on ground at LGA
avg_ground_lga = inter + regression.coefficients[4]
print('Average minutes on groud at LGA airport: ', avg_ground_lga)
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+-------------+--------------------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|    org_dummy|            features|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+-------------+--------------------+
|  0|  1|  2|     AA|JFK|   7.0|     385|  -16|4162.0|    0|        1.0|    2.0|(7,[2],[1.0])|(8,[0,3],[4162.0,...|
|  0|  1|  2|     AA|JFK|  12.0|     370|   11|3983.0|    0|        1.0|    2.0|(7,[2],[1.0])|(8,[0,3],[3983.0,...|
|  0|  1|  2|     AA|JFK|  17.0|     379|  -10|3983.0|    0|        1.0|    2.0|(7,[2],[1.0])|(8,[0,3],[3983.0,...|
|  0|  1|  2|     AA|LGA|   6.5|     240|   40|2235.0|    1|        1.0|    3.0|(7,[3],[1.0])|(8,[0,4],[2235.0,...|
|  0|  1|  2|     AA|LGA|  8.25|     250|   27|2235.0|    1|        1.0|    3.0|(7,[3],[1.0])|(8,[0,4],[2235.0,...|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+-------------+--------------------+
only showing top 5 rows

None
+------+-------------+--------+-----------------+
|km    |org_dummy    |duration|prediction       |
+------+-------------+--------+-----------------+
|2570.0|(7,[2],[1.0])|230     |259.1362558120262|
|1180.0|(7,[3],[1.0])|170     |150.1497766375601|
|1180.0|(7,[0],[1.0])|120     |132.0591167025192|
|1180.0|(7,[0],[1.0])|135     |132.0591167025192|
|415.0 |(7,[0],[1.0])|70      |75.2332350687274 |
+------+-------------+--------+-----------------+
```

```python
# This time the regression model the duration of a flight is extended again -
# On the basis of the distance being covered ('km') and the airport from which the flight departs ('org_dummy'),
# a third feature departure time ('depart') is added.
# The departure time feature is continous variable, and it can be bucketed into discrete variable.

# Bucketing is to convert a continuous variable, like age or height, into discrete values.
# This can be done by assigning values to buckets or bins with well defined boundaries.
# The buckets might have uniform or variable width.

from pyspark.ml.feature import Bucketizer, OneHotEncoder

# Create buckets at 3 hour intervals through the day
buckets = Bucketizer(splits=[0, 3, 6, 9, 12, 15, 18, 21, 24], inputCol='depart', outputCol='depart_bucket')

# Bucket the departure times
bucketed = buckets.transform(flights_onehot)
bucketed.select('depart', 'depart_bucket').show(5)

# Create a one-hot encoder
onehot = OneHotEncoder(inputCols=['depart_bucket'], outputCols=['depart_dummy'])

# One-hot encode the bucketed departure times
flights_onehot_bucketed_departure = onehot.fit(bucketed).transform(bucketed)
flights_onehot_bucketed_departure.select('depart', 'depart_bucket', 'depart_dummy').show(5)



# Import the class which will assemble the predictors.
from pyspark.ml.feature import VectorAssembler

# Create an assembler object that can merge the predictors columns into a single column.
assembler = VectorAssembler(inputCols=['km', 'org_dummy', 'depart_dummy'], outputCol='features')

# Consolidate predictor columns
flights_assembled = assembler.transform(flights_onehot_bucketed_departure)

# Split into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_assembled.randomSplit([0.8, 0.2], seed=17)

print(flights_train.show(5))


from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Create a regression object and train on training data
regression = LinearRegression(labelCol='duration').fit(flights_train)

# Create predictions for the testing data
predictions = regression.transform(flights_test)
predictions.select('km', 'org_dummy', 'depart_dummy', 'duration', 'prediction').show(5, False)



# Find the RMSE on testing data
from pyspark.ml.evaluation import RegressionEvaluator
RMSE = RegressionEvaluator(labelCol='duration').evaluate(predictions)
print('RMSE: ', RMSE)

# Average minutes on ground at OGG for flights departing between 21:00 and 24:00
avg_eve_ogg = regression.intercept
print('Average minutes on ground at OGG for flights departing between 21:00 and 24:00: ', avg_eve_ogg)

# Average minutes on ground at OGG for flights departing between 00:00 and 03:00
avg_night_ogg = regression.intercept + regression.coefficients[8]
print('Average minutes on ground at OGG for flights departing between 00:00 and 03:00: ', avg_night_ogg)

# Average minutes on ground at JFK for flights departing between 00:00 and 03:00
avg_night_jfk = regression.intercept + regression.coefficients[8] + regression.coefficients[3]
print('Average minutes on ground at JFK for flights departing between 00:00 and 03:00: ', avg_night_jfk)


# It looks adding adding the 3rd feature 'departure time' resulted in a smaller RMSE, which is good!
```

```
+------+-------------+
|depart|depart_bucket|
+------+-------------+
| 16.33|          5.0|
|  6.17|          2.0|
| 10.33|          3.0|
|  7.98|          2.0|
| 10.83|          3.0|
+------+-------------+
only showing top 5 rows

+------+-------------+-------------+
|depart|depart_bucket| depart_dummy|
+------+-------------+-------------+
| 16.33|          5.0|(7,[5],[1.0])|
|  6.17|          2.0|(7,[2],[1.0])|
| 10.33|          3.0|(7,[3],[1.0])|
|  7.98|          2.0|(7,[2],[1.0])|
| 10.83|          3.0|(7,[3],[1.0])|
+------+-------------+-------------+
```

```
only showing top 5 rows
```

```
### Regularization in regression:
# It is a form of regression, that constrains/ regularizes or shrinks the coefficient estimates towards zero.
# In other words, this technique discourages learning a more complex or flexible model,
# so as to avoid the risk of overfitting.
# There are two standard forms for the regularization term.
# Lasso regression uses a term which is proportional to the absolute value of the coefficients,
# while Ridge regression uses the square of the coefficients.
# In both cases this extra term in the loss function penalizes models with too many coefficients.
# There's a subtle distinction between Lasso and Ridge regression.
# Both will shrink the coefficients of unimportant predictors.
# However, whereas Ridge will result in those coefficients being close to zero, Lasso will actually force them to zero precisely.


## This time, we include more features in the next model:
# - km
# - org (origin airport, one-hot encoded, 8 levels)
# - depart (departure time, binned in 3 hour intervals, one-hot encoded, 8 levels)
# - dow (departure day of week, one-hot encoded, 7 levels) and
# - mon (departure month, one-hot encoded, 12 levels).


from pyspark.ml.feature import OneHotEncoder

# Create the one-hot encoding for 'mon' and 'dow' feature
onehot = OneHotEncoder(inputCols=['dow', 'mon'], outputCols=['dow_dummy', 'mon_dummy'])

# Apply the one hot encoder to the flights data
onehot = onehot.fit(flights_onehot_bucketed_departure)
flights_5_features = onehot.transform(flights_onehot_bucketed_departure)


# Import the class which will assemble the predictors.
from pyspark.ml.feature import VectorAssembler

# Create an assembler object that can merge the predictors columns into a single column.
assembler = VectorAssembler(inputCols=['km', 'org_dummy', 'depart_dummy', 'dow_dummy', 'mon_dummy'], \
                            outputCol='features')

# Consolidate predictor columns
flights_assembled = assembler.transform(flights_5_features)

# Split into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_assembled.randomSplit([0.8, 0.2], seed=17)

print(flights_train.show(5))



from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Fit linear regression model to training data
regression = LinearRegression(labelCol='duration').fit(flights_train)

# Make predictions on testing data
predictions = regression.transform(flights_test)

# Calculate the RMSE on testing data
rmse = RegressionEvaluator(labelCol='duration').evaluate(predictions)
print("The test RMSE is", rmse)

# Look at the model coefficients
coeffs = regression.coefficients
print(coeffs)

# We can see more features will not necessarily result in a better model.
# Adding some features might improve the model. Adding other features might make it worse.
# More features will always make the model more complicated and difficult to interpret.





# Now we use Lasso regression (regularized with a L1 penalty) to create a more parsimonious model.
# Many of the coefficients in the resulting model will be set to zero.
# This means that only a subset of the predictors actually contribute to the model.
# Despite the simpler model, it still produces a good RMSE on the testing data.


# Fit Lasso model (α = 1) to training data
regression = LinearRegression(labelCol='duration', regParam=1, elasticNetParam=1).fit(flights_train)

# Calculate the RMSE on testing data
rmse = RegressionEvaluator(labelCol='duration').evaluate(regression.transform(flights_test))
print("The test RMSE is", rmse)

# Look at the model coefficients
coeffs = regression.coefficients
print(coeffs)

# Number of zero coefficients
zero_coeff = sum([beta == 0 for beta in regression.coefficients])
print("Number of coefficients equal to 0:", zero_coeff)

# Regularisation produced a far simpler model with similar test performance.
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+-------------+-------------+-------------+-------------+--------------+-------------
-------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|    org_dummy|depart_bucket| depart_dummy|    dow_dummy|     mon_dummy|            f
eatures|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+-------------+-------------+-------------+-------------+--------------+-------------
-------+
|  0|  1|  2|     AA|JFK|   7.0|     385|  -16|4162.0|    0|        1.0|    2.0|(7,[2],[1.0])|          2.0|(7,[2],[1.0])|(6,[2],[1.0])|(11,[0],[1.0])|(32,[0,3,10,1
7,21...|
|  0|  1|  2|     AA|JFK|  12.0|     370|   11|3983.0|    0|        1.0|    2.0|(7,[2],[1.0])|          4.0|(7,[4],[1.0])|(6,[2],[1.0])|(11,[0],[1.0])|(32,[0,3,12,1
7,21...|
|  0|  1|  2|     AA|JFK|  17.0|     379|  -10|3983.0|    0|        1.0|    2.0|(7,[2],[1.0])|          5.0|(7,[5],[1.0])|(6,[2],[1.0])|(11,[0],[1.0])|(32,[0,3,13,1
7,21...|
|  0|  1|  2|     AA|LGA|   6.5|     240|   40|2235.0|    1|        1.0|    3.0|(7,[3],[1.0])|          2.0|(7,[2],[1.0])|(6,[2],[1.0])|(11,[0],[1.0])|(32,[0,4,10,1
7,21...|
|  0|  1|  2|     AA|LGA|  8.25|     250|   27|2235.0|    1|        1.0|    3.0|(7,[3],[1.0])|          2.0|(7,[2],[1.0])|(6,[2],[1.0])|(11,[0],[1.0])|(32,[0,4,10,1
7,21...|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+-------------+-------------+-------------+-------------+--------------+-------------
-------+
only showing top 5 rows

None
```

6. Pipeline

```python
# A pipeline in machine learning is a mechanism to combine a series of steps.
# Rather than applying each of the steps individually, they are all grouped together and applied as a single unit.
# A pipeline makes your code easier to read and maintain.

from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator
# Import class for creating a pipeline
from pyspark.ml import Pipeline

# Split into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_adding_km.randomSplit([0.8, 0.2], seed=17)

# Create the stages of pipeline
# 1. Create an indexer to convert 'org' column into an indexed column 'org_idx'.
indexer = StringIndexer(inputCol='org', outputCol='org_idx')

# 2. Create a one-hot encoder to convert the 'org_idx' and 'dow' columns into
# dummy variable columns called 'org_dummy' and 'dow_dummy'.
onehot = OneHotEncoder(inputCols=['org_idx', 'dow'],
                       outputCols=['org_dummy', 'dow_dummy'])

# 3. Create an assembler to combine the 'km' column with two dummy variable columns
# The output column should be called 'features'.
assembler = VectorAssembler(inputCols=['km', 'org_dummy', 'dow_dummy'],
                            outputCol='features')

# 4. Create a linear regression object to predict flight duration.
regression = LinearRegression(labelCol='duration')


# Construct a pipeline
pipeline = Pipeline(stages=[indexer, onehot, assembler, regression])

# Train the pipeline on the training data
# This will apply each of the individual stages in the pipeline to  training data in turn.
# None of the stages will be exposed to the testing data at all: there will be no leakage!
pipeline = pipeline.fit(flights_train)

# Make predictions on the testing data
# Once entire pipeline has been trained it will then be used to make predictions on  testing data
predictions = pipeline.transform(flights_test)

RMSE = RegressionEvaluator(labelCol='duration').evaluate(predictions)
print('RMSE: ', RMSE)
```

```
RMSE:  10.843041010712316
```

```python
## Pipleline example for the sms dataset

from pyspark.ml.feature import Tokenizer, StopWordsRemover, HashingTF, IDF
from pyspark.ml import Pipeline

from pyspark.sql.functions import regexp_replace
from pyspark.ml.feature import Tokenizer

# Use regular expressions (or REGEX) to remove the punctuation symbols.
# Replace all punctuation characters from the text column with a space.
# Do the same for all numbers in the text column.
wrangled = sms_original.withColumn('text', regexp_replace(sms_original.text, '[_():;,.!?\\-]', ' '))
wrangled = wrangled.withColumn('text', regexp_replace(wrangled.text, '[0-9]', ' '))

# Merge multiple spaces
sms_cleaned = wrangled.withColumn('text', regexp_replace(wrangled.text, ' +', ' '))

# Split into training and testing sets in a 80:20 ratio
sms_train, sms_test = sms_cleaned.randomSplit([0.8, 0.2], seed=17)


# Create the stages of pipleline
# 1. Break text into tokens at non-word characters
tokenizer = Tokenizer(inputCol='text', outputCol='words')

# 2. Remove stop words
remover = StopWordsRemover(inputCol=tokenizer.getOutputCol(),
                           outputCol='terms')

# 3. Apply the hashing trick and
# 4. transform to TF-IDF
hasher = HashingTF(inputCol=remover.getOutputCol(),
                   outputCol="hash")
idf = IDF(inputCol=hasher.getOutputCol(),
          outputCol="features")

# Create a pipeline which wraps all of the above steps as well as
# an object to create a Logistic Regression model.
logistic = LogisticRegression()
pipeline = Pipeline(stages=[tokenizer, remover, hasher, idf, logistic])

sms_pipeline = pipeline.fit(sms_train)

# Make predictions on the testing data
# Once the entire pipeline has been trained it will then be used to make predictions on the testing data.
predictions = sms_pipeline.transform(sms_test)

# Create a confusion matrix, comparing predictions to known labels
prediction.groupBy('label', 'prediction').count().show()
```

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    1|       0.0|   41|
|    0|       0.0|  948|
|    1|       1.0|  105|
|    0|       1.0|    2|
+-----+----------+-----+
```

7. Cross-validation

```
# Cross-validation is to split the training data into a number of partitions or "folds".
# Once the training data have been split into folds you can start cross-validating.
# First keep aside the data in the first fold. Train a model on the remaining four folds.
# Then evaluate that model on the data from the first fold.
# This will give the first value for the evaluation metric.
# Next you move onto the second fold, where the same process is repeated:
# data in the second fold are set aside for testing while the remaining four folds are used to train a model.
# That model is tested on the second fold data, yielding the second value for the evaluation metric.
# You repeat the process for the remaining folds.
# Each of the folds is used in turn as testing data and
# you end up with as many values for the evaluation metric as there are folds.
# At this point you are in a position to calculate the average of the evaluation metric over all folds,
# which is a much more robust measure of model performance than a single value.


from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Split into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_adding_km.randomSplit([0.8, 0.2], seed=17)

# Create an empty parameter grid for the estimator
params = ParamGridBuilder().build()

# Create objects for building and evaluating a regression model
regression = LinearRegression(labelCol='duration')
evaluator = RegressionEvaluator(labelCol='duration')

# Create an indexer for the org field
indexer = StringIndexer(inputCol='org', outputCol='org_idx')

# Create an one-hot encoder for the indexed org field
onehot = OneHotEncoder(inputCols=['org_idx'], outputCols=['org_dummy'])

# Assemble the km and one-hot encoded fields
assembler = VectorAssembler(inputCols=['km', 'org_dummy'], outputCol='features')

# Create a pipeline and cross-validator.
pipeline = Pipeline(stages=[indexer, onehot, assembler, regression])


# ingredients to perform cross-validation:
# - an estimator: build the model and is often a pipeline;
# - an evaluator: quantify how well a model works on testing data (in this case, an evluator to calculate RMSE)
cv = CrossValidator(estimator=pipeline,
                    estimatorParamMaps=params,
                    evaluator=evaluator,
                    numFolds=5)

# Train and test model on multiple folds of the training data
# NOTE: Since cross-validation builds multiple models, the fit() method can take a little while to complete.
cv = cv.fit(flights_train)

# Average RMSE across the folds
print('RMSE from cross-validation: ', cv.avgMetrics)

# Making predictions on the original test data (calling 'transform' function )
print('RMSE on testing data: ', evaluator.evaluate(cv.transform(flights_test)))

# The original testing data gives a smaller value for the RMSE than we obtained using cross-validation
# This means that a simple train-test split would have given an overly optimistic view on model performance.
```

```
/databricks/spark/python/pyspark/ml/util.py:762: UserWarning: Cannot find mlflow module. To enable MLflow logging, install mlflow from PyPI.
  warnings.warn(_MLflowInstrumentation._NO_MLFLOW_WARNING)
RMSE from cross-validation:  [11.08787714827372]
RMSE on testing data:  10.841774345692713
```

8. Grid search

```
# There is no universal "best" set of parameters for a particular model.
# The optimal choice of parameters will depend on the data and the modeling goal.
# The idea is relatively simple, you build a selection of models, one for each set of model parameters.
# Then you evaluate those models and choose the best one.

# You can systematically evaluate a model across a grid of parameter values
# using a technique known as grid search.
# To do this you need to set up a parameter grid.



# Create parameter grid
params = ParamGridBuilder()

# Add grids for two parameters
params = params.addGrid(regression.regParam, [0.01, 0.1, 1.0, 10.0]) \
               .addGrid(regression.elasticNetParam, [0.0, 0.5, 1.0])

# Build the parameter grid
params = params.build()
print('Number of models to be tested: ', len(params))

# Create cross-validator
cv = CrossValidator(estimator=pipeline,
                    estimatorParamMaps=params,
                    evaluator=evaluator,
                    numFolds=5)

cv = cv.fit(flights_train)

# Get the best model from cross validation
best_model = cv.bestModel

# Look at the stages in the best model
print(best_model.stages)

# Get the parameters for the LinearRegression object in the best model
# The stages attribute of best_model is a list of components in the model pipeline.
# The linear regression model is at index 3 in the list.
# Once you've got the model, call its extractParamMap() method.
best_model.stages[3].extractParamMap()

# Generate predictions on testing data using the best model then calculate RMSE
# Apply the transform() method to make predictions.
# Calculate the RMSE with the evaluate() method on evaluator.
predictions = best_model.transform(flights_test)
print("RMSE on testing data based on best model: ", evaluator.evaluate(predictions))
```

```
Number of models to be tested:  12
[StringIndexerModel: uid=StringIndexer_f7eef3be0cb5, handleInvalid=error, OneHotEncoderModel: uid=OneHotEncoder_058c766be225, dropLast=true, handleInvalid=error, nu
mInputCols=1, numOutputCols=1, VectorAssembler_3e9679360dbe, LinearRegressionModel: uid=LinearRegression_7c2748a48b74, numFeatures=8]
RMSE on testing data based on best model:  10.841775956248467
```

9. Ensemble

```
# An ensemble model is just a collection of models.
# An ensemble model combines the results from multiple models to
# produce better predictions than any one of those models acting alone.
# Asuccessful ensemble requires diverse models.
# It does not help if all of the models in the ensemble are similar or exactly the same.
# Ideally each of the models in the ensemble should be different.

# Gradient-Boosted Trees is an ensemble model working iteratively:
# First build a decision tree and add to the ensemble.
# Then use the ensemble to make predictions on the training data.
# Compare the predicted labels to the known labels.
# Now identify training instances where predictions were incorrect.
# Return to the start and train another tree which focuses on improving the incorrect predictions.
# As trees are added to the ensemble its predictions improve
# because each new tree focuses on correcting the shortcomings of the preceding trees.

# Import the classes required
from pyspark.ml.classification import DecisionTreeClassifier, GBTClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator


# Import the class which will assemble the predictors.
from pyspark.ml.feature import VectorAssembler

# Create an assembler object that can merge the predictors columns into a single column.
assembler = VectorAssembler(inputCols=['mon', 'depart', 'duration'],
                            outputCol='features')

# Consolidate predictor columns
flights_assembled = assembler.transform(flights_adding_km)

# Split into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_assembled.randomSplit([0.8, 0.2], seed=17)


# Create model objects and train on training data
tree = DecisionTreeClassifier().fit(flights_train)
gbt = GBTClassifier().fit(flights_train)

# Compare AUC on testing data
evaluator = BinaryClassificationEvaluator()
print('AUC for decision tree: ', evaluator.evaluate(tree.transform(flights_test)))
print('AUC for GBT: ', evaluator.evaluate(gbt.transform(flights_test)))

# Find the number of trees and the relative importance of features
print(gbt.getNumTrees)
print(gbt.featureImportances)

# Comparing the AUC, we can see
# A Gradient-Boosted Tree almost always provides better performance than a plain Decision Tree.
```

```
AUC for decision tree:  0.6252793544530469
AUC for GBT:  0.6709233092362981
20
(3,[0,1,2],[0.33420416013955273,0.34020941202937444,0.3255864278310728])

Largest AUC:  0.6762084268930573
maxDepth =  maxDepth: Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2 leaf nodes. (default: 5, current: 10)
featureSbusetStrategy =  featureSubsetStrategy: The number of features to consider for splits at each tree node. Supported options: 'auto' (choose automatically for
task: If numTrees == 1, set to 'all'. If numTrees > 1 (forest), set to 'sqrt' for classification and to 'onethird' for regression), 'all' (use all features), 'oneth
ird' (use 1/3 of the features), 'sqrt' (use sqrt(number of features)), 'log2' (use log2(number of features)), 'n' (when n is in the range (0, 1.0], use n * number o
f features. When n is in the range (1, number of features), use n features). default = 'auto' (default: auto, current: onethird)
AUC for the best model:  0.6755402726761235
```