Machine Learning using PySpark

April 16, 2020

0.0.1 Spark:

is a powerful, general purpose tool for working with Big Data. Spark transparently handles the distribution of compute tasks across a cluster. This means that operations are fast, but it also allows us to focus on the analysis rather than worrying about technical details. In this document it is demonstrated how to get data into Spark and then using the three fundamental Spark Machine Learning algorithms: Linear Regression, Logistic Regression/Classifiers, and creating pipelines.

Spark helps to build machine learning models on large datasets using distribued computing techniques.

Spark is much faster than Hadoop because it does most processing in ram.

A cluster manager allocates resources and coordinates activities across the cluster.

Every application running on the spark cluster has a driver program.

[ ]:
```
import pyspark
```

[2]:
```
# Importing the PySpark module
from pyspark.sql import SparkSession

# Create SparkSession object
spark = SparkSession.builder \
                    .master('local') \
                    .appName('test') \
                    .getOrCreate()

# What version of Spark?
print(spark.version)
```

```
2.4.5
```

[237]:
```
# Reading data from CSV file
flights = spark.read.csv('flights.csv',
                         sep=',',
                         header=True,
                         inferSchema=True,
                         nullValue='NA')

# Getting number of records
```

```
print("The data contain %d records." % flights.count())

# View the first five records
flights.show(5)

# Check column data types
flights.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
```

[237]:
```
[('mon', 'int'),
 ('dom', 'int'),
 ('dow', 'int'),
 ('carrier', 'string'),
 ('flight', 'int'),
 ('org', 'string'),
 ('mile', 'int'),
 ('depart', 'double'),
 ('duration', 'int'),
 ('delay', 'int')]
```

The correct data types have been inferred for all of the columns.

[48]:
```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

# Specifying column names and types
schema = StructType([
    StructField("id", IntegerType()),
    StructField("text", StringType()),
    StructField("label", IntegerType())
])

# Loading sms data from a delimited file
sms = spark.read.csv('sms.csv', sep=';', header=False, schema=schema)

# Print schema of DataFrame
```

```
sms.printSchema()
```

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

[146]:
```
sms.show()
```

```
+----+--------------------+-----+
| id|                text|label|
+----+--------------------+-----+
|   1|Sorry, I'll call ...|    0|
|   2|Dont worry. I gue...|    0|
|   3|Call FREEPHONE 08...|    1|
|   4|Win a 1000 cash p...|    1|
|   5|Go until jurong p...|    0|
|   6|Ok lar... Joking ...|    0|
|   7|Free entry in 2 a...|    1|
|   8|U dun say so earl...|    0|
|   9|Nah I don't think...|    0|
|  10|FreeMsg Hey there...|    1|
|  11|Even my brother i...|    0|
|  12|As per your reque...|    0|
|  13|WINNER!! As a val...|    1|
|  14|Had your mobile 1...|    1|
|  15|I'm gonna be home...|    0|
|  16|SIX chances to wi...|    1|
|  17|URGENT! You have ...|    1|
|  18|I've been searchi...|    0|
|  19|I HAVE A DATE ON ...|    0|
|  20|XXXMobileMovieClu...|    1|
+----+--------------------+-----+
only showing top 20 rows
```

0.1 Preparing Data For Machine Learning Models

In above codes, Spark session is initiated and data is loaded. In the following codes data will be used to build a classification model.

[ ]:
```
# Removing an uninformative column and
# Removing rows which do not have information about whether or not a flight was␣
↪delayed.
```

[238]:
```
# Removing the 'flight' column
flights = flights.drop('flight')
```

[239]:
```
# Number of records with missing 'delay' values
flights.filter('delay IS NULL').count()
```

[239]:
```
2978
```

[240]:
```
# Removing the 'flight' column
flights = flights.dropna(subset=['delay'])
display(flights)
flights.show(20)
```

```
DataFrame[mon: int, dom: int, dow: int, carrier: string, org: string, mile: int, depart: doubl


+---+---+---+-------+---+----+------+--------+-----+
|mon|dom|dow|carrier|org|mile|depart|duration|delay|
+---+---+---+-------+---+----+------+--------+-----+
|  0| 22|  2|     UA|ORD| 316| 16.33|      82|   30|
|  2| 20|  4|     UA|SFO| 337|  6.17|      82|   -8|
|  9| 13|  1|     AA|ORD|1236| 10.33|     195|   -5|
|  5|  2|  1|     UA|SFO| 550|  7.98|     102|    2|
|  7|  2|  6|     AA|ORD| 733| 10.83|     135|   54|
|  1| 16|  6|     UA|ORD|1440|   8.0|     232|   -7|
|  1| 22|  5|     UA|SJC|1829|  7.98|     250|  -13|
| 11|  8|  1|     OO|SFO| 158|  7.77|      60|   88|
|  4| 26|  1|     AA|SFO|1464| 13.25|     210|  -10|
|  4| 25|  0|     AA|ORD| 978| 13.75|     160|   31|
|  8| 30|  2|     UA|ORD| 719| 13.28|     151|   16|
|  3| 16|  3|     UA|ORD|1745|   9.0|     264|    3|
|  0|  3|  4|     AA|LGA|1097| 17.08|     190|   32|
|  5|  9|  1|     UA|SFO| 967|  12.7|     158|   20|
|  3| 10|  4|     B6|ORD|1735| 17.58|     265|  155|
| 11| 15|  1|     AA|ORD| 802|  6.75|     160|   23|
|  8| 18|  4|     UA|SJC| 948|  6.33|     160|   17|
|  2| 14|  5|     B6|JFK| 944|  6.17|     166|    0|
|  7| 21|  4|     OO|ORD| 607|  19.0|     110|   21|
| 11|  6|  6|     OO|SFO| 316|  8.75|      82|   40|
+---+---+---+-------+---+----+------+--------+-----+
only showing top 20 rows
```

[241]:
```
# Number of complete records
flights.filter('delay IS NOT NULL').count()
```

[241]:
```
47022
```

[242]:
```
# Removing records with missing values in any column and getting the number of␣
↪remaining rows
flights = flights.dropna()
```

```
print(flights.count())
```

```
47022
```

The columns and rows are discarded which will certainly not contribute to a model.

Column manipulation

The Federal Aviation Administration (FAA) considers a flight to be “delayed” when it arrives 15 minutes or more after its scheduled time.

The next step of preparing the flight data has two parts:

1. converting the units of distance, replacing the mile column with a kmcolumn; and
2. creating a Boolean column indicating whether or not a flight was delayed.

[243]:
```
# Importing the required function
from pyspark.sql.functions import round

# Converting 'mile' to 'km' and dropping 'mile' column
flights_km = flights.withColumn('km', round(flights.mile * 1.60934, 0)) \
                    .drop('mile')

# Createing 'label' column indicating whether flight delayed (1) or not (0)
flights_km = flights_km.withColumn('label', (flights_km.delay >= 15).
 ↪cast('integer'))

# Checking first five records
flights_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
```

Categorical columns In the flights data there are two columns, carrier and org, which hold categorical data. Those columns need to be transformed into indexed numerical values.

[253]:
```
from pyspark.ml.feature import StringIndexer

# Create an indexer
indexer = StringIndexer(inputCol='carrier', outputCol='carrier_idx')
```

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

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

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

flights_indexed.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|
|  1| 16|  6|     UA|ORD|   8.0|     232|   -7|2317.0|    0|        0.0|    0.0|
|  1| 22|  5|     UA|SJC|  7.98|     250|  -13|2943.0|    0|        0.0|    5.0|
| 11|  8|  1|     OO|SFO|  7.77|      60|   88| 254.0|    1|        2.0|    1.0|
|  4| 26|  1|     AA|SFO| 13.25|     210|  -10|2356.0|    0|        1.0|    1.0|
|  4| 25|  0|     AA|ORD| 13.75|     160|   31|1574.0|    1|        1.0|    0.0|
|  8| 30|  2|     UA|ORD| 13.28|     151|   16|1157.0|    1|        0.0|    0.0|
|  3| 16|  3|     UA|ORD|   9.0|     264|    3|2808.0|    0|        0.0|    0.0|
|  0|  3|  4|     AA|LGA| 17.08|     190|   32|1765.0|    1|        1.0|    3.0|
|  5|  9|  1|     UA|SFO|  12.7|     158|   20|1556.0|    1|        0.0|    1.0|
|  3| 10|  4|     B6|ORD| 17.58|     265|  155|2792.0|    1|        4.0|    0.0|
| 11| 15|  1|     AA|ORD|  6.75|     160|   23|1291.0|    1|        1.0|    0.0|
|  8| 18|  4|     UA|SJC|  6.33|     160|   17|1526.0|    1|        0.0|    5.0|
|  2| 14|  5|     B6|JFK|  6.17|     166|    0|1519.0|    0|        4.0|    2.0|
|  7| 21|  4|     OO|ORD|  19.0|     110|   21| 977.0|    1|        2.0|    0.0|
| 11|  6|  6|     OO|SFO|  8.75|      82|   40| 509.0|    1|        2.0|    1.0|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
only showing top 20 rows
```

Machine Learning model needs numbers not strings, so these transformations are vital!

Before applying machine learning algorithms in PySpark, the data has to be prepared and various input columns are consolidated into a single column. This is necessary because the machine learning algorithms in spark operate on a single vector of predictors, although each element in that vector may consist of multiple values. Ultimately all the predictors are assembled into a single column.

[254]:
```
# Importing the necessary class
from pyspark.ml.feature import VectorAssembler
```

```
# Creating an assembler object
assembler =␣
↪VectorAssembler(inputCols=['mon','dom','dow','carrier_idx','org_idx','depart','duration',␣
↪'km'], outputCol='features')

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

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

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

[246]:
```
# Splitting into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_assembled.randomSplit([0.8, 0.2], seed=17)

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

```
0.7980732423121092
```

Building a Decision Tree Now that flights data is split into training and testing sets, we can use the training set to fit a Decision Tree model.

The data are available as flights_train and flights_test

[99]:
```
# Importing the Decision Tree Classifier class
from pyspark.ml.classification import DecisionTreeClassifier

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

# Creating predictions for the testing data and take a look at the predictions
prediction = tree_model.transform(flights_test)
prediction.select('label', 'prediction', 'probability').show(20, False)
```

```
+-----+----------+----------------------------------------+
|label|prediction|probability                             |
+-----+----------+----------------------------------------+
|1    |1.0       |[0.4911924119241192,0.5088075880758808] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|0    |1.0       |[0.3551949240436825,0.6448050759563175] |
|0    |1.0       |[0.45071868583162217,0.5492813141683778]|
|1    |0.0       |[0.6467889908256881,0.3532110091743119] |
|1    |0.0       |[0.8383233532934131,0.16167664670658682]|
|0    |1.0       |[0.4911924119241192,0.5088075880758808] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|0    |0.0       |[0.8383233532934131,0.16167664670658682]|
|1    |0.0       |[0.6876876876876877,0.3123123123123123] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|0    |1.0       |[0.3551949240436825,0.6448050759563175] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|0    |0.0       |[0.7533112582781457,0.24668874172185432]|
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
|1    |1.0       |[0.3551949240436825,0.6448050759563175] |
+-----+----------+----------------------------------------+
only showing top 20 rows
```

Evaluating the Decision Tree The quality of the model can be assessed by evaluating how well it performs on the testing data. Because the model was not trained on these data, this represents an objective assessment of the model.

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.

[100]:
```
# Creating a confusion matrix
prediction.groupBy('label', 'prediction').count().show()

# Calculating the elements of the confusion matrix
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
accuracy = (TN + TP) / (TN + TP + FN + FP)
print(accuracy)
```

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    1|       0.0| 1211|
|    0|       0.0| 2420|
|    1|       1.0| 3613|
|    0|       1.0| 2251|
+-----+----------+-----+

0.6353870458135861
```

Building a Logistic Regression model I have already built a Decision Tree model using the flights data. Now I am going to create a Logistic Regression model on the same data.

The objective is to predict whether a flight is likely to be delayed by at least 15 minutes (label 1) or not (label 0).

Although there are a variety of predictors at the disposal, I will only use the mon, depart and duration columns for the moment. These are numerical features which can immediately be used for a Logistic Regression model.

[280]:
```
# Creating an assembler object
assembler = VectorAssembler(inputCols=['mon','depart','duration'],␣
 ↪outputCol='features')

# Consolidating predictor columns
flights_LR = assembler.transform(flights_indexed)

# Checking the resulting column
flights_LR.select('mon', 'depart', 'duration','features', 'label' ).show(5,␣
 ↪truncate=False)
```

```
+---+------+--------+-----------------+-----+
|mon|depart|duration|features         |label|
+---+------+--------+-----------------+-----+
|0  |16.33 |82      |[0.0,16.33,82.0] |1    |
|2  |6.17  |82      |[2.0,6.17,82.0]  |0    |
|9  |10.33 |195     |[9.0,10.33,195.0]|0    |
|5  |7.98  |102     |[5.0,7.98,102.0] |0    |
|7  |10.83 |135     |[7.0,10.83,135.0]|1    |
+---+------+--------+-----------------+-----+
only showing top 5 rows
```

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

```
[282]: # Importing the logistic regression class
       from pyspark.ml.classification import LogisticRegression

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

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

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    1|       0.0| 1725|
|    0|       0.0| 2590|
|    1|       1.0| 3099|
|    0|       1.0| 2081|
+-----+----------+-----+
```

```
[283]: # Calculating the elements of the confusion matrix
       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()
```

```
[284]: from pyspark.ml.evaluation import MulticlassClassificationEvaluator,␣
        ↪BinaryClassificationEvaluator

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

       # Finding weighted precision
       multi_evaluator = MulticlassClassificationEvaluator()
       weighted_precision = multi_evaluator.evaluate(prediction, {multi_evaluator.
        ↪metricName: "weightedPrecision"})

       # Finding AUC
       binary_evaluator = BinaryClassificationEvaluator()
       auc = binary_evaluator.evaluate(prediction, {binary_evaluator.metricName:␣
        ↪"areaUnderROC"})
```

```
precision = 0.60
recall    = 0.64
```

[285]:
```
# Creating predictions for the testing data and take a look at the predictions
prediction = logistic.transform(flights_LR)
prediction.select('label', 'prediction', 'probability').show(20, False)
```

```
+-----+----------+----------------------------------------+
|label|prediction|probability                             |
+-----+----------+----------------------------------------+
|1    |1.0       |[0.4094652930693248,0.5905347069306752] |
|0    |0.0       |[0.6278090529736562,0.37219094702634375]|
|0    |0.0       |[0.5874081260366599,0.41259187396334013]|
|0    |0.0       |[0.62393041804841,0.37606958195159]     |
|1    |0.0       |[0.5814641321803106,0.41853586781968943]|
|0    |0.0       |[0.5052063260574969,0.49479367394250295]|
|0    |1.0       |[0.4963792308199202,0.5036207691800798] |
|1    |0.0       |[0.7181008461124411,0.281899153887559]  |
|0    |1.0       |[0.4566350417973066,0.5433649582026934] |
|1    |1.0       |[0.4725425397162709,0.5274574602837291] |
|1    |0.0       |[0.540503457041672,0.4594965429583279]  |
|0    |1.0       |[0.496860482755149,0.5031395172448511]  |
|1    |1.0       |[0.3441548650249355,0.6558451349750646] |
|1    |0.0       |[0.5073056216787756,0.49269437832122437]|
|1    |1.0       |[0.33773708322451607,0.6622629167754839]|
|1    |0.0       |[0.6918289336252362,0.3081710663747637] |
|1    |0.0       |[0.6631988405564212,0.3368011594435788] |
|0    |0.0       |[0.5868323140474861,0.4131676859525139] |
|1    |1.0       |[0.4384761701096394,0.5615238298903605] |
|1    |0.0       |[0.6931274605186096,0.30687253948139037]|
+-----+----------+----------------------------------------+
only showing top 20 rows
```

[142]:
```
# Features selection for the model
Flights = flights_assembled[['features', 'label']]

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

[143]:
```
# Importing the logistic regression class
from pyspark.ml.classification import LogisticRegression

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

# Creating predictions for the testing data and show confusion matrix
```

```
prediction = logistic.transform(flights_test)
prediction.groupBy('label', 'prediction').count().show()
```

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    1|       0.0| 1768|
|    0|       0.0| 2601|
|    1|       1.0| 3126|
|    0|       1.0| 2000|
+-----+----------+-----+
```

Evaluating 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 and
- recall. ###### Precision is the proportion of positive predictions which are correct. For all flights which are predicted to be delayed, what proportion is actually delayed? ###### Recall is the proportion of positives outcomes which are correctly predicted. For all delayed flights, what proportion is correctly predicted by the model? ###### The precision and recall are generally formulated in terms of the positive target class. But it's also possible to calculate weighted versions of these metrics which look at both target classes.

[144]:
```
# Calculating the elements of the confusion matrix
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()
```

[145]:
```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator,␣
 ↪BinaryClassificationEvaluator

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

# Finding weighted precision
multi_evaluator = MulticlassClassificationEvaluator()
weighted_precision = multi_evaluator.evaluate(prediction, {multi_evaluator.
 ↪metricName: "weightedPrecision"})

# Finding AUC
binary_evaluator = BinaryClassificationEvaluator()
```

```
auc = binary_evaluator.evaluate(prediction, {binary_evaluator.metricName:␣
↪"areaUnderROC"})
```

```
precision = 0.61
recall    = 0.64
```

The weighted precision indicates what proportion of predictions (positive and negative) are correct.

0.2 Punctuation, numbers and tokens

In the start of this document a dataset of SMS messages which had been labeled as either "spam" (label 1) or "ham" (label 0) was loaded. I am now going to use those data to build a classifier model.

But first the SMS messages needs to be prepared as follows:

removing punctuation and numbers

tokenizing (split into individual words)

removing stop words

applying the hashing trick

converting to TF-IDF representation.

In the following codes I will remove punctuation and numbers, then tokenizing the messages.

The SMS data are available as sms.

Importing the function to replace regular expressions and the feature to tokenize.

Replacing all punctuation characters from the text column with a space. Doing the same for all numbers in the text column.

Splitting the text column into tokens. Naming the output column words.

[147]:
```
# Importing the necessary functions
from pyspark.sql.functions import regexp_replace
from pyspark.ml.feature import Tokenizer

# Removing punctuation (REGEX provided) and numbers
wrangled = sms.withColumn('text', regexp_replace(sms.text, '[_():;,.!?\\-]', '␣
↪'))
wrangled = wrangled.withColumn('text', regexp_replace(wrangled.text, '[0-9]', '␣
↪'))

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

# Splitting the text into words
wrangled = Tokenizer(inputCol='text', outputCol='words').transform(wrangled)
```

```
wrangled.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
```

Next, stop words will be removed and hashing trick will be applied.

Stopping words and hashing

The next steps will be to remove stop words and then applying the hashing trick, converting the results into a TF-IDF.

A quick reminder about these concepts:

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.

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.

The tokenized SMS data are stored in sms in a column named words. I've cleaned up the handling of spaces in the data so that the tokenized text is neater.

[151]:
```
from pyspark.ml.feature import StopWordsRemover, HashingTF, IDF

# Remove stop words.
wrangled = StopWordsRemover(inputCol='words', outputCol='terms')\
      .transform(wrangled)

# Apply the hashing trick
wrangled = HashingTF(inputCol='terms', outputCol='hash', numFeatures=1024)\
      .transform(wrangled)
```

```
# Convert hashed symbols to TF-IDF
tf_idf = IDF(inputCol='hash', outputCol='features')\
      .fit(wrangled).transform(wrangled)

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

```
+--------------------------------+------------------------------------------------
-------------------------------------------------------+
|terms                           |features
|
+--------------------------------+------------------------------------------------
-------------------------------------------------------+
|[sorry, call, later, meeting]   |(1024,[138,344,378,1006],[2.2391682769656747,2
.892706319430574,3.684405173719015,4.244020961654438])|
|[dont, worry, guess, busy]      |(1024,[53,233,329,858],[4.618714411095849,3.55
7143394108088,4.618714411095849,4.937168142214383])   |
|[call, freephone]
|(1024,[138,396],[2.2391682769656747,3.3843005812686773])
|
|[win, cash, prize, prize, worth]|(1024,[31,69,387,428],[3.7897656893768414,7.28
4881949239966,4.4671645129686475,3.898659777615979])  |
+--------------------------------+------------------------------------------------
-------------------------------------------------------+
only showing top 4 rows
```

The data is ready to build a spam classifier.

[152]:
```
tf_idf.show()
```

```
+---+--------------------+-----+--------------------+--------------------+------
--------------+--------------------+
| id|                text|label|               words|               terms|
hash|            features|
+---+--------------------+-----+--------------------+--------------------+------
--------------+--------------------+
|  1|Sorry I'll call l…|    0|[sorry, i'll, cal…|[sorry, call,
lat…|(1024,[138,344,37…|(1024,[138,344,37…|
|  2|Dont worry I gues…|    0|[dont, worry, i, …|[dont, worry,
gue…|(1024,[53,233,329…|(1024,[53,233,329…|
|  3| Call FREEPHONE now |    1|[call, freephone,…|   [call,
freephone]|(1024,[138,396],[…|(1024,[138,396],[…|
|  4|Win a cash prize …|    1|[win, a, cash, pr…|[win, cash,
prize…|(1024,[31,69,387,…|(1024,[31,69,387,…|
|  5|Go until jurong p…|    0|[go, until, juron…|[go, jurong,
poin…|(1024,[116,262,33…|(1024,[116,262,33…|
|  6|Ok lar Joking wif…|    0|[ok, lar, joking,…|[ok, lar,
joking,…|(1024,[449,572,66…|(1024,[449,572,66…|
|  7|Free entry in a w…|    1|[free, entry, in,…|[free, entry,
wkl…|(1024,[16,24,77,1…|(1024,[16,24,77,1…|
|  8|U dun say so earl…|    0|[u, dun, say, so,…|[u, dun, say,
ear…|(1024,[26,212,249…|(1024,[26,212,249…|
|  9|Nah I don't think…|    0|[nah, i, don't, t…|[nah, think,
goes…|(1024,[364,396,50…|(1024,[364,396,50…|
| 10|FreeMsg Hey there…|    1|[freemsg, hey, th…|[freemsg, hey,
da…|(1024,[112,163,17…|(1024,[112,163,17…|
| 11|Even my brother i…|    0|[even, my, brothe…|[even, brother,
l…|(1024,[41,319,367…|(1024,[41,319,367…|
| 12|As per your reque…|    0|[as, per, your, r…|[per, request,
me…|(1024,[8,16,60,18…|(1024,[8,16,60,18…|
| 13|WINNER As a value…|    1|[winner, as, a, v…|[winner, valued,
…|(1024,[37,69,88,1…|(1024,[37,69,88,1…|
| 14|Had your mobile m…|    1|[had, your, mobil…|[mobile, months,
…|(1024,[119,138,15…|(1024,[119,138,15…|
| 15|I'm gonna be home…|    0|[i'm, gonna, be, …|[gonna, home,
soo…|(1024,[114,192,44…|(1024,[114,192,44…|
| 16|SIX chances to wi…|    1|[six, chances, to…|[six, chances,
wi…|(1024,[31,122,163…|(1024,[31,122,163…|
| 17|URGENT You have w…|    1|[urgent, you, hav…|[urgent, won,
wee…|(1024,[69,193,197…|(1024,[69,193,197…|
| 18|I've been searchi…|    0|[i've, been, sear…|[searching,
right…|(1024,[181,210,22…|(1024,[181,210,22…|
| 19|I HAVE A DATE ON …|    0|[i, have, a, date…|      [date,
sunday]|(1024,[28,360],[1…|(1024,[28,360],[5…|
| 20|XXXMobileMovieClu…|    1|[xxxmobilemoviecl…|[xxxmobilemoviecl…|(1024,
[9,45,87,13…|(1024,[9,45,87,13…|
+---+--------------------+-----+--------------------+--------------------+------
--------------+--------------------+
only showing top 20 rows
```

0.3 Training a spam classifier

The SMS data have now been prepared for building a classifier. Following procedures are done:

- removed numbers and punctuation
- split the messages into words (or "tokens")
- removed stop words
- applied the hashing trick and
- converted to a TF-IDF representation. ###### Next I'll need to split the TF-IDF data into training and testing sets. Then I'll use the training data to fit a Logistic Regression model and finally evaluate the performance of that model on the testing data.

[155]:
```
# Split the data into training and testing sets
sms_train, sms_test = tf_idf.randomSplit([0.8, 0.2], seed=13)
```

```python
# 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|   47|
|    0|       0.0|  987|
|    1|       1.0|  124|
|    0|       1.0|    3|
+-----+----------+-----+
```

0.4 Applying Linear Regression Model on the Flights Data

Before applying the regression model we need to convert categorical variables into numerical values.

One Hot Encoding: The process of creating dummy variables. Because only one of the columns created is ever active or hot.

Indexed values can be converted into a format in which we can perform meaningful mathematical operations. The first step is to create a column for each of the levels effectively we then can place a check in the column corresponding to the value in each row. These new columns are known as dummy variables. However, rather than having checks in the dummy variable columns it makes more sense to use binary values, where a one indicates the presense of the corresponding level.

Encoding flight origin

The org column in the flights data is a categorical variable giving the airport from which a flight departs.

- ORD — O'Hare International Airport (Chicago)
- SFO — San Francisco International Airport
- JFK — John F Kennedy International Airport (New York)
- LGA — La Guardia Airport (New York)
- SMF — Sacramento
- SJC — San Jose
- TUS — Tucson International Airport
- OGG — Kahului (Hawaii) ###### Obviously this is only a small subset of airports. Nevertheless, since this is a categorical variable, it needs to be one-hot encoded before it can be used in a regression model.

0.4.1 Flight duration model: Just distance

In the codes below a regression model will be built to predict flight duration (the duration column).

For the moment the model will be kept simple, including only the distance of the flight (the km column) as a predictor.

[261]:
```
flights_onehot.show(5)
```

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

[257]:
```
# Creating an assembler object
assembler = VectorAssembler(inputCols=['km'], outputCol='features')

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

# Checking the resulting column
flights_assembled.select('km', 'features', 'duration').show(5, truncate=False)
```

```
+------+--------+--------+
|km    |features|duration|
+------+--------+--------+
|509.0 |[509.0] |82      |
|542.0 |[542.0] |82      |
|1989.0|[1989.0]|195     |
|885.0 |[885.0] |102     |
|1180.0|[1180.0]|135     |
+------+--------+--------+
only showing top 5 rows
```

[258]:
```
# Features selection for the linear regression model
Flights = flights_assembled[['km','features', 'duration']]

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

[259]:
```
# Checking the data before implementing the model
Flights.show(4)
```

```
+------+--------+--------+
|    km|features|duration|
+------+--------+--------+
| 509.0| [509.0]|      82|
| 542.0| [542.0]|      82|
|1989.0|[1989.0]|     195|
| 885.0| [885.0]|     102|
+------+--------+--------+
only showing top 4 rows
```

[260]:
```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

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

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

# Calculating the RMSE
RegressionEvaluator(labelCol='duration').evaluate(predictions)
```

```
+--------+-----------------+
|duration|prediction       |
+--------+-----------------+
|44      |52.31997678812364|
|46      |52.31997678812364|
|47      |52.31997678812364|
|47      |52.31997678812364|
|47      |52.31997678812364|
+--------+-----------------+
only showing top 5 rows
```

[260]:
```
17.102928777865053
```

A simple regression model is built. Time to make sense of the coefficients!

0.5 Interpreting the coefficients

The linear regression model for flight duration as a function of distance takes the form duration= + ×distance

where

- — intercept (component of duration which does not depend on distance) and
- — coefficient (rate at which duration increases as a function of distance; also called the slope). ###### By looking at the coefficients of the model it can be inferred
- how much of the average flight duration is actually spent on the ground and
- what the average speed is during a flight.

[191]:
```
# Intercept (average minutes on ground)
inter = regression.intercept
print(inter)

# Coefficients
coefs = regression.coefficients
print(coefs)

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

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

```
44.151201083106216
[0.0756368120834947]
0.0756368120834947
793.2645275129605
```

Flight duration model: Adding origin airport

[269]:
```
# Importing the one hot encoder class
from pyspark.ml.feature import OneHotEncoderEstimator

# Creating an instance of the one hot encoder
onehot = OneHotEncoderEstimator(inputCols=['org_idx'], outputCols=['org_dummy'])

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

# Checking the results
flights_onehot.select('org', 'org_idx', 'org_dummy').distinct().sort('org_idx').
 ↪show()
```

```
+---+-------+-------------+
|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])|
|TUS|    6.0|(7,[6],[1.0])|
|OGG|    7.0|    (7,[],[])|
+---+-------+-------------+
```

[345]:
```
flights_onehot.show(5)
```

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

[387]:
```
# Creating an assembler object
assembler = VectorAssembler(inputCols=['km', 'org_dummy'], outputCol='features')

# Consolidating predictor columns
Flights_assembled = assembler.transform(flights_onehot)

# Checking the resulting column
Flights_assembled.select('km', 'org_idx', 'org_dummy', 'features').show(5,␣
↪truncate=False)
```

```
+------+-------+-------------+---------------------+
|km    |org_idx|org_dummy    |features             |
+------+-------+-------------+---------------------+
|509.0 |0.0    |(7,[0],[1.0])|(8,[0,1],[509.0,1.0]) |
|542.0 |1.0    |(7,[1],[1.0])|(8,[0,2],[542.0,1.0]) |
|1989.0|0.0    |(7,[0],[1.0])|(8,[0,1],[1989.0,1.0])|
|885.0 |1.0    |(7,[1],[1.0])|(8,[0,2],[885.0,1.0]) |
|1180.0|0.0    |(7,[0],[1.0])|(8,[0,1],[1180.0,1.0])|
+------+-------+-------------+---------------------+
only showing top 5 rows
```

[388]:
```python
# Splitting into training and testing sets in a 80:20 ratio
flights_train, flights_test = Flights_assembled.randomSplit([0.8, 0.2], seed=17)
```

[389]:
```python
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

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

# Creating predictions for the testing data
predictions = regression.transform(flights_test)

# Calculating the RMSE on testing data
RegressionEvaluator(labelCol='duration').evaluate(predictions)
```

[389]:
```
11.054736661521357
```

[279]:
```python
# Averaging speed in km per hour
avg_speed_hour = 60 / regression.coefficients[0]
print(avg_speed_hour)

# Averaging minutes on ground at OGG
inter = regression.intercept
print(inter)

# Averaging minutes on ground at JFK
avg_ground_jfk = inter + regression.coefficients[3]
print(avg_ground_jfk)

# Averaging minutes on ground at LGA
avg_ground_lga = inter + regression.coefficients[4]
print(avg_ground_lga)
```

```
807.9946998977274
15.536278501461982
68.2929992642743
62.37189890382314
```

Average speed in km per hour is 807.99 and you're going to spend over an hour on the ground at JFK or LGA but only around 15 minutes at OGG.

Adding more features to the model.

[416]:
```python
from pyspark.ml.feature import StringIndexer

# Creating an indexer
indexer = StringIndexer(inputCol='dow', outputCol='dow_idx')

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

# Indexer creates a new column with numeric index values
flights_indexed_1 = indexer_model.transform(Flights_Onehot)

# Repeating the process for the other categorical feature
flights_indexed_2 = StringIndexer(inputCol='mon', outputCol='mon_idx').
 ↪fit(flights_indexed_1).transform(flights_indexed_1)

flights_indexed_2.show(5)
```

```
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
-------------+-------------+-------------+-------+-------+
|mon|dom|dow|carrier|org|depart|duration|delay|    km|label|carrier_idx|org_idx|
org_dummy|depart_bucket| depart_dummy|dow_idx|mon_idx|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
-------------+-------------+-------------+-------+-------+
|  0| 22|  2|     UA|ORD| 16.33|      82|   30| 509.0|    1|        0.0|
0.0|(7,[0],[1.0])|          5.0|(7,[5],[1.0])|    3.0|    2.0|
|  2| 20|  4|     UA|SFO|  6.17|      82|   -8| 542.0|    0|        0.0|
1.0|(7,[1],[1.0])|          2.0|(7,[2],[1.0])|    2.0|    3.0|
|  9| 13|  1|     AA|ORD| 10.33|     195|   -5|1989.0|    0|        1.0|
0.0|(7,[0],[1.0])|          3.0|(7,[3],[1.0])|    1.0|   10.0|
|  5|  2|  1|     UA|SFO|  7.98|     102|    2| 885.0|    0|        0.0|
1.0|(7,[1],[1.0])|          2.0|(7,[2],[1.0])|    1.0|    0.0|
|  7|  2|  6|     AA|ORD| 10.83|     135|   54|1180.0|    1|        1.0|
0.0|(7,[0],[1.0])|          3.0|(7,[3],[1.0])|    6.0|    5.0|
+---+---+---+-------+---+------+--------+-----+------+-----+-----------+-------+
-------------+-------------+-------------+-------+-------+
only showing top 5 rows
```

[418]:
```python
# Creating an instance of the one hot encoder
onehot = OneHotEncoderEstimator(inputCols=['dow_idx'], outputCols=['dow_dummy'])

# Applying the one hot encoder to the flights data
onehot = onehot.fit(flights_indexed_2)
```

```
flights_onehot_1 = onehot.transform(flights_indexed_2)
```

[419]:
```
# Creating an instance of the one hot encoder
onehot = OneHotEncoderEstimator(inputCols=['mon_idx'], outputCols=['mon_dummy'])

# Applying the one hot encoder to the flights data
onehot = onehot.fit(flights_onehot_1)
flights_onehot_2 = onehot.transform(flights_onehot_1)
```

[420]:
```
# Creating an assembler object
assembler = VectorAssembler(inputCols=['km', 'org_dummy', 'depart_dummy',␣
 ↪'dow_dummy', 'mon_dummy'], outputCol='features')

# Consolidating predictor columns
Flights_assembled_2 = assembler.transform(flights_onehot_2)

# Checking the resulting column
Flights_assembled_2.select('features', 'delay').show(5, truncate=False)
```

```
+---------------------------------------------+-----+
|features                                     |delay|
+---------------------------------------------+-----+
|(32,[0,1,13,18,23],[509.0,1.0,1.0,1.0,1.0])  |30   |
|(32,[0,2,10,17,24],[542.0,1.0,1.0,1.0,1.0])  |-8   |
|(32,[0,1,11,16,31],[1989.0,1.0,1.0,1.0,1.0])|-5   |
|(32,[0,2,10,16,21],[885.0,1.0,1.0,1.0,1.0])  |2    |
|(32,[0,1,11,26],[1180.0,1.0,1.0,1.0])        |54   |
+---------------------------------------------+-----+
only showing top 5 rows
```

[421]:
```
# Splitting into training and testing sets in a 80:20 ratio
flights_train, flights_test = Flights_assembled_2.randomSplit([0.8, 0.2],␣
 ↪seed=17)
```

[422]:
```
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)
```

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

```
The test RMSE is 10.654636698370071
[0.07437053614755808,27.68596220025394,20.416195296423815,51.94941661211876,45.8
92641529964656,15.264827728130735,17.875603118834825,17.67202227282828 6,-13.7867
00781627408,1.100531233976254,4.040123230594686,6.923514338325492,4.618958027583
4,8.816029823713295,8.793023843706338,0.27650488910820925,0.25166618768575866,0.
40710366961217587,-0.020808916015339983,0.5593437454250453,0.20731758714474077,-
3.4617425722623896,-3.6145570485604535,-1.2445439267483436,-1.5169516812863153,-
1.415110333477263,-3.4085225815753186,0.9512780285894189,-3.4066856294989245,-2.
534162773012359,-3.085129805594701,-2.0085277342892898]
```

With all those non-zero coefficients the model is a little hard to interpret!

The way to minimize the residuals is through loss function. Which is an equation that describes how well the model fits the data. MSE = ‘Mean Squared Error’ Loss Function

It is good to have a single number which summarizes the performance of a model. The Root Mean Squared Error is often used for regression models. It’s the square root of the Mean Squared Error. Values of RMSE are relative to the scale of the value that you are aiming to predict. A smaller RMSE, always indicates better predictions.

The intercept is the value predicted by the model when all predictors are zero. This value can be found for the model using the intercept attribute.

There is a slope associated with each of the predictors too which represents how rapidly the model changes when the predictor changes. The coefficients attributes gives you access to those values. There is a coefficient for each of the predictors.

Regularization: Selecting only the useful features. A linear Regression model attempts to derive a coefficient for each feature in the data. The coefficients quantify the effect of the corresponding features. More features imply more coefficients. This works well when the dataset has a few columns and many rows.

You need to derive a few coefficients and you have plenty of data. The converse situation, with many columns and few rows is much more challenging. You need to calculate values for numerous coefficients but you don’t have much data to do it. Even if you do manage to derive values for all of those coefficients, your model will end up being very complicated and difficult to interpret. Ideally you want to create a parsimonious model: one that has just the minimum required number of predictors. It will be as simple as possible, yet still able to make robust predictions. The obvious solution is to simply select the best subset of columns. There are variety of approaches to the ‘feature selection’ problem. One such approach is ‘Penalized Regression’. The idea is that the model is penalized for having too many coefficients.

The conventional regression algorithm chooses coefficients to minimize the loss function, which is average of the squared residuals. A good model will result in low MSE because its predictions will be close to the observed values. With penalized regression an additional ‘regularization’ or ‘shrinkage’ term is added to the loss function. Rather than depending on the data, this term is a function of the model coefficients.

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 is 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 force them to zero precisely. Its possible to have a mix of Lasso and Ridge.

The strength of the regularization is determined by a parameter which is generally denoted by the Greek symbol lambda. When lambda is 0 thee is no regularization and when lambda is large regularization completely dominates. Ideally the lambda should be between these two extremes.

After calculating RMSE, if the model coefficients suggest that all predictors have been assigned non-zero values which means that every predictor is contributing to the model. This is possible but unlikely that all of the features are equally important for predicting consumption.

When we calculate RMSE on Ridge regression if the coefficients are smaller than the coefficients of the standard linear regression model.

After applying Lasso model if RMSE is increased but not by significant degree, turning to the coefficients if not all of them are 0, suggest that non zero coefficients are the right predictors.

[423]:
```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Fit Lasso model ( = 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)
```

```
The test RMSE is 11.658525377902118
[0.073477813981869,5.705477962636444,0.0,28.95807228294463,21.756742669403714,-2
.342239207839284,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0653078088579282,1.20515161209012
98,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
Number of coefficients equal to 0: 25
```

Regularisation produced a far simpler model with even better test performance.

0.6 Creating Pipeline

Pipelines seriously streamline the workflow. They also help to ensure that training and testing data are treated consistently and that no leakage of information between these two sets takes place.

Leakage?

Most of the actions used in this document involve both a fit() and a transform() method. Those methods are usually applied in a relaxed way. But to get really robust results one needs to be careful only to apply the fit() method to training data. Because if a fit() method is applied to any of the testing data then the model will effectively have seen those data during the training phase, so the results of testing will no longer be objective. The transform() method on the other hand can be applied to both training and testing data since it does not result in any changes in the underlying model. Leakage occurs whenever a fit() method is applied to testing data. the model would then already have seen the testing data, so using those data to test the model would not be fair. The model will perform well on data which has been used for training. If the fit model is applied to testing data in any stage of building a model then the model is compromised. However, if the fit method is only applied to the training data then the model will be in good shape. A pipeline makes it easier to avoid leakage because it simplifies the training and testing process. A pipeline 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 string indexer is used to convert the type column to indexed values. Applying a one-hot encoder to convert those indexed values into dummy variables. Then assembling a set of predictors into a single features column And finally building a regression model.

First the indexer is fit to the training data, then transform() is called on the training data to add the indexed column, Then you call the transform() method on the testing data to add the indexed column there too. The testing data was not used to fit the indexer. Next the same thing is done for the one-hot-encoder, fitting to the training. And then using the fitted encoder to update the training and testing data sets. Assembler is next. In this case there is not fit() method, so the transform method is simply applied to the training and testing data. Finally the model is ready. Throughout the process the testing data has been out of the training process. A pipeline makes a training and testing a complicated model a lot easier. The pipeline class lives in the ml sub module. You create a pipeline by specifying a sequence of stages where each stage corresponds to a step in the model building process. The stages are executed in order. Now rather than calling the fit() and transform() methods for each stage, you simply call the fit () method for the pipeline on the training data. each of the stages in the pipeline is the automatically applied to the training data in turn. This will systematically apply the fit() and transform() methods for each stage in the pipeline. The trained pipeline can then be used to make predictions on the testing data by calling its transform() method. The pipeline transform() method will only call the transform() method for each of the stages in the pipeline. The stages in the pipeline can be accessed by using the stages attribute, which is a list. You pick out individual stages by indexing into the list. For example, to access the regression component of the pipeline you'd use an index of 3. Having access to the component makes it possible to get the intercept and coefficients for the trained LinearRegression model.

[ ]:
```python
# Converting categorical strings to index values
indexer = StringIndexer(inputCol='org', outputCol='org_idx')

# One-hot encode index values
onehot = OneHotEncoderEstimator(
    inputCols=['org_idx','dow'],
    outputCols=['org_dummy', 'dow_dummy']
)

# Assembleing predictors into a single column
assembler = VectorAssembler(inputCols=['km', 'org_dummy', 'dow_dummy'],␣
 ↪outputCol= 'features')

# A linear regression object
regression = LinearRegression(labelCol='duration')
```

[427]:
```python
# Splitting into training and testing sets in a 80:20 ratio
flights_train, flights_test = flights_km.randomSplit([0.8, 0.2], seed=17)
```

[428]:
```python
# Importing class for creating a pipeline
from pyspark.ml import Pipeline

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

# Training the pipeline on the training data
pipeline = pipeline.fit(flights_train)

# Making predictions on the testing data
predictions = pipeline.transform(flights_test)
```

SMS Spam Pipeline

[430]:
```python
from pyspark.ml.feature import Tokenizer, StopWordsRemover, HashingTF, IDF

# Break text into tokens at non-word characters
tokenizer = Tokenizer(inputCol='text', outputCol='words')

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

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

# Create a logistic regression object and add everything to a pipeline
logistic = LogisticRegression()
```

```
pipeline = Pipeline(stages=[tokenizer, remover, hasher, idf, logistic])
```

Applying Pipeline is a lot simpler than applying each stage separately.

0.6.1 Cross validating simple flight duration model.

Until now I have been testing models using a rather simple technique: randomly splitting the data into training and testing sets. Training a model on the training data and then evaluating its performance on the testing set. There is one major drawback to this approach: you only get one estimate of the model performance. You would have a more robust idea of how well a model works if you were able to test it multiple times. This is the idea behind cross validation.

You start out with the full set of data. You still split the data into a training set and a testing set. Remember that before splitting its important to first randomize the data so that the distributions in the training and testing data are similar. You then split the training data into a number of partitions or folds. The number of folds normally factors into the name of techniques. For example, if you split into five folds then you would talk about 5 fold cross validation. Once the training data have been split into folds you can start cross validating. First keep aside the data in the first fold, train the model on the remaining 4 folds. Then evaluate the 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.

The classes needed are CrossValidator and ParamGridBuilder, both from the tuning sub-module. It's a more robust measure of model performance because it is based on multiple train/test splits. The average metric is returned as a list.

If we evaluate the predictions on the original testing data then we find 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.

[434]:
```
# Features selection for the linear regression model
Flights = flights_assembled[['km','features', 'duration']]

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

[435]:
```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

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

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

```
# Create a cross validator
cv = CrossValidator(estimator=regression, estimatorParamMaps=params,␣
 ↪evaluator=evaluator, numFolds=5)

# Train and test model on multiple folds of the training data
cv = cv.fit(flights_train)
```

Since cross-validation builds multiple models, the fit() method can take a little while to complete.

Now is the time to cross validate a model pipeline.

Cross validating flight duration model pipeline.

[457]:
```
# Creating an indexer for the org field
indexer = StringIndexer(inputCol='org', outputCol='org_idx')

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

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

# Creating a pipeline and cross-validator.
pipeline = Pipeline(stages=[indexer, onehot, assembler, regression])
cv = CrossValidator(estimator=pipeline,
                    estimatorParamMaps=params,
                    evaluator=evaluator)
```

Optimizing flights linear regression

[458]:
```
# Creating parameter grid
params = ParamGridBuilder()

# Adding 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])

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

# Creating cross-validator
cv = CrossValidator(estimator=pipeline, estimatorParamMaps=params,␣
 ↪evaluator=evaluator, numFolds=5)
```

```
Number of models to be tested:  12
```

Multiple models are built effortlessly using grid search.

Comparing a linear regression model with an intercept to one that passes through the origin. By default a linear regression model will always fit an intercept but we can be explicit and specify the fitIntercept parameter as True. You fit the model to the training data and then calculate the RMSE for the testing data. Next you repeat the process, but specify False for fitIntercept parameter. Now you are creating a model which passes through the origin. When you evaluate this model you find that the RMSE is higher. So, comparing the two models you will choose the first one that has a lower RMSE. However, there is a problem with this approach. Just getting a single estimate of RMSE is not very robust. It would be better to make this comparison using cross-validation. You also have to manually build the models for the two different parameter values. It would be great if that could be automated. 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. First an empty grid is created and points are added. First you create a grid builder and then you add one or more grids.

At present there is just one grid, which takes two values for the fitIntercept parameter. Call the build method to construct the grid. A separate model will be built for each point in the grid.

After that you create a cross-validator object and fit it to the training data. This builds a bunch of models: one model for each fold and point in the parameter grid. If there are two points in the grid and ten folds, this translates into twenty models. The cross validator is going to loop through each of the points in the parameter grid and for each point it will create a cross-validated model using the corresponding parameter values. When you take a look at the average metrics attribute, you can see why the matric is given as a list: you get one average value for each point in the grid. The value confirms what you observed before: the model that includes an intercept is superior to the model without intercept. Our goal is to get the best model for the data. That can be retrieved by using the appropriately named bestModel attribute. But its not actually necessary to work with this directly because the cross-validator object will behave like the best model. So you can use it directly to make predictions on the testing data. of course, you want to know what the best parameter value is and that can be retrieved using the explainParam() method. As expected the best value for the fitintercept parameter is True. It can be seen after the word current in the output. Its possible to add more parameters to the grid. If more parameters and values are added to the grid, the more models there are to be evaluated. Because each of these models will be evaluated using cross validation, this might take a little while.

Dissecting the best flight duration model.

[460]:
```
# Using the model with more features

# Splitting into training and testing sets in a 80:20 ratio
flights_train, flights_test = Flights_assembled_2.randomSplit([0.8, 0.2],␣
 ↪seed=17)
```

[461]:
```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Fit Lasso model ( = 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)
```

```
The test RMSE is 11.658525377902118
[0.073477813981869,5.705477962636444,0.0,28.95807228294463,21.756742669403714,-2
.342239207839284,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0653078088579282,1.20515161209012
98,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
Number of coefficients equal to 0: 25
```

SMS spam optimised

[465]:
```
# Create parameter grid
params = ParamGridBuilder()

# Add grid for hashing trick parameters
params = params.addGrid(hasher.numFeatures, [1024, 4096, 16384]) \
               .addGrid(hasher.binary, [True, False])

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

# Build parameter grid
params = params.build()
```

Using cross-validation on a pipeline makes it possible to optimise each stage in the workflow.

[468]:
```
# Assemble predictors into a single column
assembler = VectorAssembler(inputCols=['mon', 'depart', 'duration'], outputCol=␣
 ↪'features')

# Consolidating predictor columns
Flights_RM = assembler.transform(flights_km)

# Checking the resulting column
Flights_RM.select('mon', 'depart', 'duration', 'features', 'label').show(5,␣
 ↪truncate=False)
```

```
+---+------+--------+----------------+-----+
|mon|depart|duration|features        |label|
+---+------+--------+----------------+-----+
|0  |16.33 |82      |[0.0,16.33,82.0] |1    |
|2  |6.17  |82      |[2.0,6.17,82.0]  |0    |
|9  |10.33 |195     |[9.0,10.33,195.0]|0    |
|5  |7.98  |102     |[5.0,7.98,102.0] |0    |
|7  |10.83 |135     |[7.0,10.83,135.0]|1    |
+---+------+--------+----------------+-----+
only showing top 5 rows
```

[533]:
```
# Splitting into training and testing sets in a 80:20 ratio
flights_train, flights_test = Flights_RM.randomSplit([0.8, 0.2], seed=17)
```

Ensemble:

How models can be combined to form a collection or "ensemble" which is more powerful than each of the individual models alone. An ensemble is just a collection of models. An ensemble combines the results from multiple models to produce better predictions than any one of the models acting alone. The concept is based on the idea of the Wisdom of the Crowd which implies that the aggregated opinion of a group is better than the opinions of the individuals in that group, even if the individuals are experts. There must be diversity and independence in the crowd. This applies to models too: a successful 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.

A Random Forest, as the name implies, is a collection of trees. To ensure that each of the trees is different, the Decision Tree algorithm is modified slightly. Each tree is trained on a different random subset of the data and within each tree a random subset of features is used for splitting at each node. The result is a collection of trees where no trees are the same. Within the Random Forest model, all of the trees operate in parallel. By default the number is trees is 20, but they should be dropped to 5 to make the model easier to interpret. Once the model is trained its possible to access the individual trees in the forest using the trees attribute. Each tree is different by having varying number of nodes. We can then make predictions using each tree individually. The predictions of the individual trees can be collected in a subset of the testing data. Different trees can make different predictions for the same observations, the random forest creates the consensus prediction by aggregating the predictions across all of the individual trees. The transform() method automatically generates a consensus prediction column. It also creates a probability column which assigns aggregate probabilities to each of the outcomes.

Its possible to get an idea of the relative importance of the features in the model by looking at the featureImportances attribute. An importance is assigned to each feature, where a larger importance indicates a feature which makes a larger contribution to the model.

The second ensembled model is the Gradient Boosted Trees. Again the aim is to build a collection of diverse models, but the approach is slightly different. Rather than building a set of trees that operate in parallel, now we build trees which work in series. The boosting algorithm works 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. You can make an objective comparison between a plain Decision Tree and the two ensemble models by looking at the values of AUC obtained by each of them on the testing data. Ensemble models are more powerful than decision trees. The results can be further improved by using cross validation to tune the parameters.

Cross Validation and Grid Search help to choose a good set of parameters for any model.

0.6.2 Delayed flights with Gradient-Boosted Trees

[471]:
```python
from pyspark.ml.classification import DecisionTreeClassifier, GBTClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# 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()
evaluator.evaluate(tree.transform(flights_test))
evaluator.evaluate(gbt.transform(flights_test))

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

```
20
(3,[0,1,2],[0.33140587299252644,0.3362350235651107,0.33235910344236297])
```

A Gradient-Boosted Tree almost always provides better performance than a plain Decision Tree.

0.6.3 Delayed flights with a Random Forest

[490]:
```python
from pyspark.ml.classification import RandomForestClassifier



# Create a random forest classifier
forest = RandomForestClassifier()

# Create a parameter grid
params = ParamGridBuilder() \
            .addGrid(forest.featureSubsetStrategy, ['all', 'onethird', 'sqrt',␣
 ↪'log2']) \
            .addGrid(forest.maxDepth, [2, 5, 10]) \
            .build()

# Create a binary classification evaluator
evaluator = BinaryClassificationEvaluator()

# Create a cross-validator
cv = CrossValidator(estimator=forest, estimatorParamMaps=params,␣
↪evaluator=evaluator, numFolds=5)

cvModel = cv.fit(flights_train)
cvModel.avgMetrics[0]
```

[490]:
```
0.6140396536679908
```

[534]:
```
# Making predictions.
predictions = cvModel.transform(flights_test)

prediction.select('label', 'prediction', 'probability').show(20, False)
```

```
+-----+----------+----------------------------------------+
|label|prediction|probability                             |
+-----+----------+----------------------------------------+
|1    |1.0       |[0.4094652930693248,0.5905347069306752] |
|0    |0.0       |[0.6278090529736562,0.37219094702634375]|
|0    |0.0       |[0.5874081260366599,0.41259187396334013]|
|0    |0.0       |[0.62393041804841,0.37606958195159]     |
|1    |0.0       |[0.5814641321803106,0.41853586781968943]|
|0    |0.0       |[0.5052063260574969,0.49479367394250295]|
|0    |1.0       |[0.4963792308199202,0.5036207691800798] |
|1    |0.0       |[0.7181008461124411,0.281899153887559]  |
|0    |1.0       |[0.4566350417973066,0.5433649582026934] |
|1    |1.0       |[0.4725425397162709,0.5274574602837291] |
|1    |0.0       |[0.540503457041672,0.4594965429583279]  |
|0    |1.0       |[0.496860482755149,0.5031395172448511]  |
|1    |1.0       |[0.3441548650249355,0.6558451349750646] |
|1    |0.0       |[0.5073056216787756,0.49269437832122437]|
|1    |1.0       |[0.33773708322451607,0.6622629167754839]|
|1    |0.0       |[0.6918289336252362,0.3081710663747637] |
|1    |0.0       |[0.6631988405564212,0.3368011594435788] |
|0    |0.0       |[0.5868323140474861,0.4131676859525139] |
|1    |1.0       |[0.4384761701096394,0.5615238298903605] |
|1    |0.0       |[0.6931274605186096,0.30687253948139037]|
+-----+----------+----------------------------------------+
only showing top 20 rows
```

0.6.4 Evaluating Random Forest

[519]:
```
# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```

```
Test Error = 0.369142
```

[525]:
```
# AUC for best model on testing data
best_auc = evaluator.evaluate(cvModel.transform(flights_test))
print(best_auc)
```

```
0.630858346498157
```

[526]:
```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.linalg import Vectors

pyspark.ml.tuning.TrainValidationSplitModel(bestModel=None,␣
 ↪validationMetrics=[], subModels=None)

# Average AUC for each parameter combination in grid
avg_auc = cvModel.avgMetrics
print(avg_auc)
```

```
[0.6140396536679908, 0.6597444649920785, 0.6696930452057217, 0.6442375963770994,
0.662966189012898, 0.6737446602362016, 0.6390990816959313, 0.6615734602456644,
0.6707680494493815, 0.6390990816959313, 0.6615734602456644, 0.6707680494493815]
```

[527]:
```
# Average AUC for the best model
best_model_auc = max(cvModel.avgMetrics)
print(best_model_auc)
```

```
0.6737446602362016
```

[528]:
```
# What's the optimal parameter value?
opt_max_depth = cvModel.bestModel.explainParam('maxDepth')
opt_feat_substrat = cvModel.bestModel.explainParam('featureSubsetStrategy')
print(opt_max_depth)
print(opt_feat_substrat)
```

```
maxDepth: Maximum depth of the tree. (Nonnegative) E.g., depth 0 means 1 leaf
node; depth 1 means 1 internal node + 2 leaf nodes. (default: 5, current: 10)
featureSubsetStrategy: The number of features to consider for splits at each
tree node. Supported options: auto, all, onethird, sqrt, log2, (0.0-1.0], [1-n].
(default: auto, current: onethird)
```

```
[ ]:
```