DATA WRANGLING WITH PYSPARK FOR DATA SCIENTISTS WHO KNOW PANDAS

# LOAD CSV

Pandas

df= pd.read\_csv("mtcars.csv")

PySpark

df = spark.read \

.options(header=True, inferSchema=True) \

.csv("mtcars.csv " )

# VIEW DATAFRAME

Pandas

df

df.head(10)

PySpark

df . show( )

df.show(10)

df.select('var1','var2').show(10)

df.groupBy("varNum").count().sort("VarNum",ascending=False).show()

# COLUMNS AND DATA TYPES

Pandas

df.columns

df.dtypes

PySpark

df.columns

df.dtypes

type(df[‘age’]) #sql.column

type(df.select(‘age’))#sql.dataframe

type(df.select([v1,v2]))#sql.dataframe

type(df.head(2)[0])#row object

# RENAME COLUMNS

Pandas

df .columns = [‘a', ‘b',’c’]

df.rename(columns {'old': ‘new’ })

PySpark

Df= Df.toDF (‘a', ‘b',’c’)

Df= df.withColumnRenamed('old',‘new’ )

# DROP COLUMN

Pandas

df.drop('mpg', axis=1)

PySpark

df.drop('mpg')

# FILTERING

Pandas

df[df.mpg < 20]

df[(df.mpg < 20) & (df.cyl == 6)]

PySpark

df[df.mpg < 20]

df[(df.mpg < 20) & (df.cyl == 6)]

# AGGREGATION

Pandas

df.groupby([ ‘cyl’,’gear’])\

.agg({ ‘mpg’: ‘mean’, ‘disp’:‘min' })

PySpark

df.groupby([ ‘cyl’,’gear’])\

.agg({ ‘mpg’: ‘mean’, ‘disp’:‘min' })

# Groupby.

Df.groupBy(‘var’).count()

Df.groupBy(‘var’).mean()

# aggregate by group by

# Aggregate method

Df.agg( {‘var2’: ‘sum’ }).show()

#aggregate across all the data

# Groupby & agg()

Df.groupBy(‘var1’).agg( {‘var2’: ‘max’ }).show()

# orderBy

Df.orderBy(‘var1’)

Df.orderBy(df[‘var1’].desc())

# ADD COLUMN

Pandas

df[‘new' |] = 1 / df.mpg

PySpark

df.withColumn('new', 1 / df.mpg)

df.withColumn('new', 1 / df[‘mpg’])

# FILL NULLS

Pandas

df.fillna(0) + Many more options

PySpark

df.fillna(0)

# STANDARD TRANSFORMATIONS

Pandas

Import numpy as **np**

df['logVar1'] = **np.**log(df.var1)

PySpark

import pyspark.sgl.functions as **F** df.withColumn( 'logVar1', **F.**log(df.var1))

# ROW CONDITIONAL STATEMENTS

Pandas

df['cond'] =df.apply(lambda r:

1 if r.mpg > 20 else 2 if r.cyl == 6 else 3, axis=1)

PySpark

You have to indicate that you’re working with columns by adding the col() function to your code. Otherwise, you won’t be able to do element-wise operations.

Import pyspark.sql.functions as F

from pyspark.sql.functions import \*

df.withColumn( ‘cond’, \

F.when(df.mpg > 20, 1) \

.when(df.cyl == 6, 2)\

.otherwise(3))

# SUMMARY STATISTICS

# Pandas

df.describe()

# PySpark

df.describe().show()

(only count, mean, stddev, min, max)

df.**selectExpr** (

"percentile approx(mpg, array(.25, .5, .75)) as mpg"

). show() #column and array of percentile

# PYTHON WHEN REQUIRED

Pandas

df[‘disp1'] = df.disp.apply(lambda x: x+1)

PySpark

Import pyspark.sql.functions as F

from pyspark.sgl.types import DoubleType

**fn** = **F.udf**(lambda x: x+1, **DoubleType**()) #return type

df.withColumn('disp1', **fn**(df.disp) )

# MERGE/JOIN DATAFRAMES

# Pandas

left.merge(right, on='key' ) left.merge(right, left\_on='a', right\_on="b')

# PySpark

left.join(right, on='key' )

left. join(right, left.a == right.b)

# PIVOT TABLE

# Pandas

pd.pivot\_table(df, values='D', \ index=[‘A’, °B’'], columns=['C'], \ age func=np.sum)

# PySpark

df.groupBy("A ,“B”}.pivot("C" ).sum("D")

# HISTOGRAMS

# Pandas

df.hist()

# PySpark

Df.sample(False,0.1).toPandas().hist()

# SQL

# Pandas

# n/a

# PySpark

df.createOrReplaceTempView('foo')

df2 = spark.sql('select \* from foo')

# SPARK FUNCTIONS

from pyspark.sql.functions import \*

df.select(countDistinct(‘Sales’)).show()

# Alias

df.select(countDistinct(‘Sales’).alias(‘Levels’).show() # Alias in display

# format number (round)

var1\_std =df.select(stddev(‘var1).alias(‘std’))

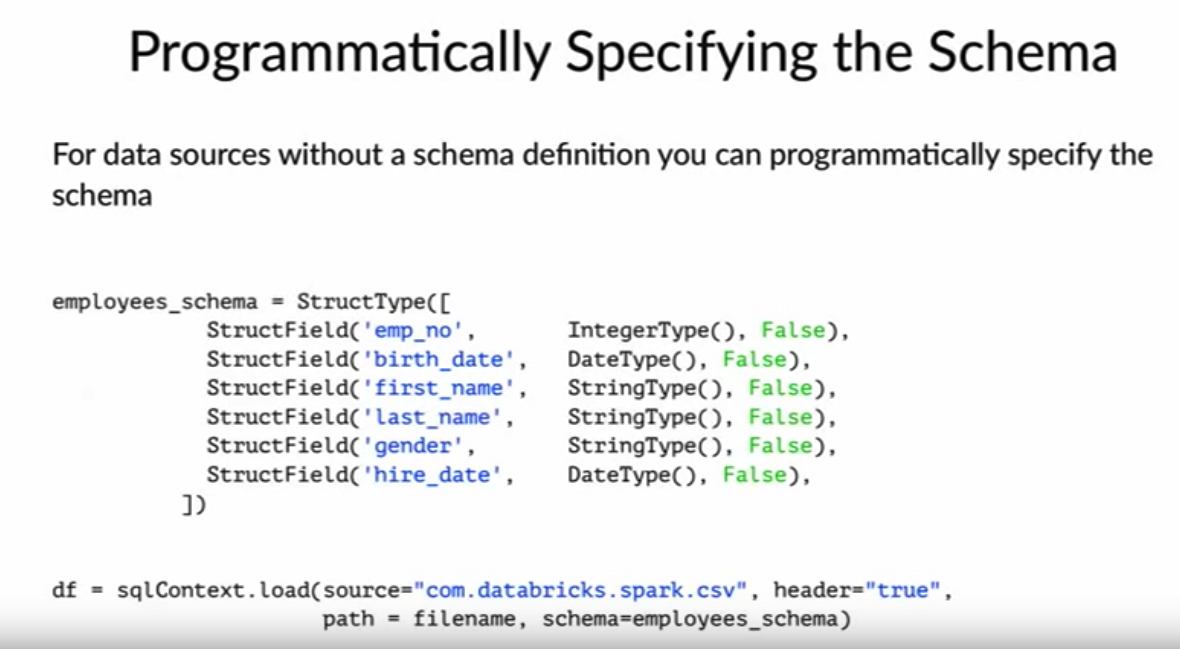
var1\_std.select(format\_number(‘std’,2))#2 decimals

# Schema:

For .csv files you need to provide a schema

Or: inferSchema=true





# Collect Data

Result= df.filter(df.low==5).collect #save as row list object

row = result[0]

row.asDict()[‘var3’] #extract value

feature eng

https://www.datacamp.com/community/tutorials/apache-spark-tutorial-machine-learning

# What is Spark,

Spark is a platform for cluster computing. Spark lets you spread data and computations over **clusters**with multiple **nodes** (think of each node as a separate computer). Splitting up your data makes it easier to work with very large datasets because each node only works with a small amount of data.

As each node works on its own subset of the total data, it also carries out a part of the total calculations required, so that both data processing and computation are performed in **parallel** over the nodes in the cluster. It is a fact that parallel computation can make certain types of programming tasks much faster.

**The first step in using Spark is connecting to a cluster.**

In practice, the cluster will be hosted on a remote machine that's connected to all other nodes.

There will be one computer, called the **master**that manages splitting up the data and the computations. The master is connected to the rest of the computers in the cluster, which are called **slaves.** The master sends the slaves data and calculations to run, and they send their results back to the master.

Creating the connection is as simple as creating an instance of the SparkContext class called sc. The class constructor takes a few optional arguments that allow you to specify the attributes of the cluster you're connecting to.

An object holding all these attributes can be created with the SparkConf() constructor.

print(sc.version) # 2.1.0

**Second, Creating a spark Session.**

You can think of the SparkContext as your connection to the cluster and the SparkSession as your interface with that connection.

from pyspark.sql import SparkSession

# Create an SparkSession or returns an existing if there's already one in the environment.

spark = SparkSession.builder.getOrCreate()

# Print spark session

print(spark)

# Print the tables/data in in your cluster as a list.

print(spark.**catalog**.listTables())

Spark's core data structure is the Resilient Distributed Dataset (RDD). This is a low level object that lets Spark work its magic by splitting data across multiple nodes in the cluster. However, RDDs are hard to work with directly, so you'll be using the Spark DataFrame abstraction built on top of RDDs.

As you saw in the last exercise, one of the tables in your cluster is the flights table. Running a query on this table is as easy as using the .sql() method on your SparkSession. This method takes a string containing the query and returns a DataFrame with the results.  you look closely, you'll notice that the table flights is only mentioned in the query, not as an argument to any of the methods. This is because there isn't a local object in your environment that holds that data, so it wouldn't make sense to pass the table as an argument.

query = "FROM flights SELECT \* LIMIT 10"

# Get the first 10 rows of flights

flights10 = **spark.sql(query)**

# Show the results

flights10.show()

**Pandafy a Spark DataFrame**

Sometimes it makes sense to then take that table and work with it locally using a tool like pandas. Spark DataFrames make that easy with the .toPandas() method. Calling this method on a Spark DataFrame returns the corresponding pandas DataFrame.

query = "SELECT origin, dest, COUNT(\*) as N FROM flights GROUP BY origin, dest"

# Run the query

flight\_counts = spark.sql(query)

# Convert the results to a pandas DataFrame

pd\_counts = flight\_counts.**toPandas()**

# Print the head of pd\_counts

print(pd\_counts.head())

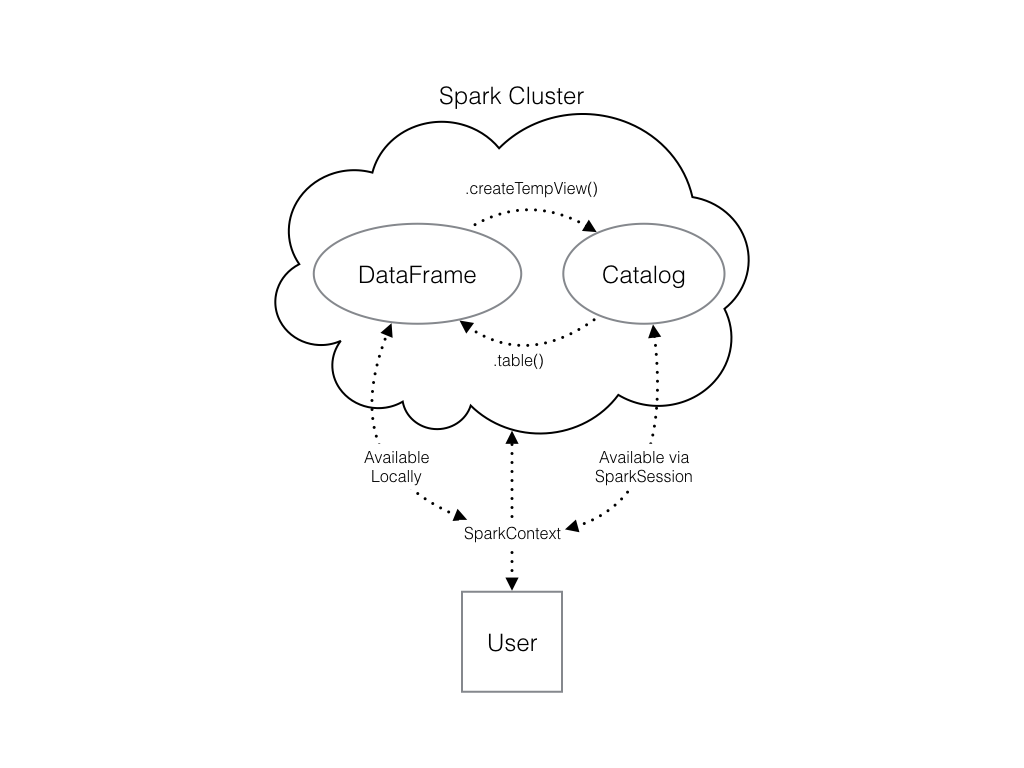
## Put a pandasDataFrame into a Spark cluster

The .createDataFrame() method takes a pandas DataFrame and returns a Spark DataFrame.

The output of this method is stored locally, not in the SparkSession catalog. This means that you can use all the Spark DataFrame methods on it, but you can't access the data in other contexts.

For example, a SQL query (using the .sql() method) that references your DataFrame will throw an error. To access the data in this way, you have to save it as a temporary table.

You can do this using the .createTempView() Spark DataFrame method, which takes as its only argument the name of the temporary table you'd like to register. This method registers the DataFrame as a table in the catalog, but as this table is temporary, it can only be accessed from the specific SparkSession used to create the Spark DataFrame. There is also the method .createOrReplaceTempView(). This safely creates a new temporary table if nothing was there before, or updates an existing table if one was already defined.



# Create pandas df pd\_temp

pd\_temp = pd.DataFrame(np.random.random(10))

# Create spark\_temp from pd\_temp

spark\_temp = spark.createDataFrame(pd\_temp)

# Add spark\_temp to the catalog

spark\_temp.createOrReplaceTempView("temp ")

# Examine the tables in the catalog again

print(spark.catalog.listTables())

**Create a SparkDataFrame from a file**

Use the .read.csv() method to create a Spark DataFrame. Dataframe will not be added yet to the catalog.

file\_path = "/usr/local/share/datasets/airports.csv"

# Read in the airports data

airports = spark.read.csv(file\_path,header=True)

airports.show()

Updating a Spark DataFrame is somewhat different than working in pandasbecause the Spark DataFrame is *immutable*. This means that it can't be changed, and so columns can't be updated in place.

## Creating Columns

 In Spark you can do this using the .withColumn() method, which takes two arguments. First, a string with the name of your new column, and second the new column itself.

Thus, all these methods return a new DataFrame. To overwrite the original DataFrame you must reassign the returned DataFrame using the method like so:

df = df.withColumn("newCol", df.oldCol + 1)

The above code creates a DataFrame with the same columns as df plus a new column, newCol, where every entry is equal to the corresponding entry from oldCol, plus one.

To overwrite an existing column, just pass the name of the column as the first argument!

**Selecting**

The Spark variant of SQL's SELECT is the .select() method. This method takes multiple arguments - one for each column you want to select. These arguments can either be the column name as a string (one for each column) or a column object (using the df.colName syntax).

The two expressions will produce the same output:

s1 =df.select('tailnum','origin', 'dest')

s2 = df.select(df.tailnum, df.origin, df.dest)

**Selecting and performing operations, alias()**

 When you pass a column object, you can perform operations like addition or subtraction on the column to change the data contained in it, much like inside .withColumn().

The difference between .select() and .withColumn() methods is that .select() returns only the columns you specify, while .withColumn()returns all the columns of the DataFrame in addition to the one you defined.

It's often a good idea to drop columns you don't need at the beginning of an operation so that you're not dragging around extra data as you're wrangling. In this case, you would use .select() and not .withColumn().

When you're selecting a column using the df.colName notation, you can perform any column operation and the .select() method will return the transformed column.

flights.select(flights.air\_time/60)

returns a column of flight durations in hours instead of minutes. You can also use the .alias() method to rename a column you're selecting.

So if you wanted to .select() the column duration\_hrs (which isn't in your DataFrame) you could do

flights.select((flights.air\_time/60).alias("duration\_hrs"))

**# select with operation in a col**

# Define avg\_speed variable

avg\_speed = (df.distance/( df.air\_time/60)).alias("avg\_speed")

# Select the correct columns

speed1 = df.select("origin", "dest", "tailnum", avg\_speed)

## Select using SQL notation

.selectExpr() takes SQL expressions as a string, with the SQL as keyword being equivalent to the .alias()method. To select multiple columns, you can pass multiple strings:

df.selectExpr("air\_time/60 **as** duration\_hrs")

# Create the same table using a SQL expression

speed2 = df.selectExpr("origin", "dest", "tailnum", "distance/(air\_time/60) as avg\_speed")

**Filtering Data**

the .filter() method. As you might suspect, this is the Spark counterpart of SQL's WHERE clause. The .filter() method takes either a Spark Column of boolean (True/False) values as a string. The two expressions will produce the same output:

df.filter(df.air\_time > 120).show()

df.filter("air\_time > 120").show()

df3= df.filter(df[‘var’]>1).select(‘var1’

**Applying two filters:**

filterA = df.origin == "SEA"

filterB = df.dest == "PDX"

df2= df.filter(filterA).filter(filterB)

df3= df.filter( (df[‘var’]>1) & (..) )

## Aggregating with groupBy().method()

All of the common aggregation methods, like .min(), .max(), and .count() are GroupedData methods. These are created by calling the .groupBy() DataFrame method.

For example, to find the minimum value of a column, col, in a DataFrame, df, you could do

df.groupBy().min("col").show()

This creates a GroupedData object (so you can use the .min()method), then finds the minimum value in col, and returns it as a DataFrame.

# Find the shortest flight from PDX

flights.filter(flights.origin == "PDX").groupBy().min("distance").show()

# Average duration of Delta flights

flights.filter(flights.carrier=='DL').filter(flights.origin=='SEA').groupBy().avg('air\_time').show()

# Total hours in the air by creating a column called duration\_hrs

flights.withColumn("duration\_hrs", flights.air\_time/60).groupBy().sum('duration\_hrs').show()

**Grouping and Aggregating**: **groupBy(‘argument)**

when you pass the name of one or more columns in your DataFrame to the .groupBy() method, the aggregation methods behave like when you use a GROUP BY statement in a SQL query!

# Number of flights each plane made: Group by tailnum

by\_plane = flights.groupBy("tailnum")

by\_plane.count().show()

# Average duration of flights: Group by origin

by\_origin = flights.groupBy("origin")

by\_origin.avg("air\_time").show()

**Aggregating with agg(F.function())**

In addition to the GroupedData methods you've already seen, there is also the .agg() method. This method lets you pass an aggregate column expression that uses any of the aggregate functions from the pyspark.sql.functions submodule.

# Import pyspark.sql.functions as F

import pyspark.sql.functions as F

# Create a GroupedData table: Group by month and dest

by\_month\_dest = flights.groupBy('month', 'dest')

# Average departure delay by month and destination

by\_month\_dest.avg('dep\_delay').show()

# Standard deviation

by\_month\_dest.agg(F.stddev('dep\_delay')).show()

## JOIN

In PySpark, joins are performed using the DataFrame method .join(). This method takes three arguments. The first is the second DataFrame that you want to join with the first one. The second argument, on, is the name of the key column(s) as a string. The names of the key column(s) must be the same in each table. The third argument, how, specifies the kind of join to perform.

twodf = df1.join(df2, on="dest", how="leftouter")

## Machine Learning Pipelines

At the core of the pyspark.ml module are the Transformer and Estimator classes. Almost every other class in the module behaves similarly to these two basic classes.

Transformer classes have a .transform() method that takes a DataFrame and returns a new DataFrame; usually the original one with a new column appended. For example, you might use the class Bucketizer to create discrete bins from a continuous feature or the class PCA to reduce the dimensionality of your dataset using principal component analysis.

Estimator classes all implement a .fit() method. These methods also take a DataFrame, but instead of returning another DataFrame they return a model object. This can be something like a StringIndexerModel for including categorical data saved as strings in your models, or a RandomForestModel that uses the random forest algorithm for classification or regression.

## Data types

 Spark only handles numeric data for modeling.

To remedy this, you can use the .cast() method in combination with the .withColumn() method. It's important to note that .cast() works on columns, while .withColumn() works on DataFrames.

You can put this call to .cast() inside a call to .withColumn() to overwrite the already existing column

# String to integer

dataframe = dataframe.withColumn("colName", dataframe.colName.cast("new\_type")

df = df.withColumn("arr\_delay", df.arr\_delay.cast("integer"))

df= df.withColumn("var1", df[ ‘var1].cast(FloatType())) \

.withcolumn(var2… )

# Function for Data types

*# Import all from `sql.types`*

from pyspark.sql.types import \*

*# Write a custom function to convert the data type of DataFrame columns*

def convertColumn(df, names, newType):

for name in names:

df = df.withColumn(name, df[name].cast(newType))

return df

*# Running: Create a list to Assign all column names to `columns`*

columns = ['households', 'housingMedianAge', 'latitude', 'longitude', 'medianHouseValue', 'medianIncome', 'population', 'totalBedRooms', 'totalRooms']

# Call function & Conver the `df` columns to `FloatType()`

df = convertColumn(df, columns, FloatType())

**Making a Boolean & transform to integer**

# Create is\_late

model\_data = model\_data.withColumn("is\_late", model\_data.arr\_delay > 0)

# Convert to an integer: target VAR

model\_data = model\_data.withColumn("label", model\_data.is\_late.cast("integer"))

**Remove missing values**

# Remove missing values

model\_data = model\_data.filter("arr\_delay is not NULL and dep\_delay is not NULL and air\_time is not NULL and plane\_year is not NULL")

**Change units of target**

# Import all from `sql.functions`

from pyspark.sql.functions import \*

# Adjust the values of `var1`

df = df.withColumn("var1", col("var1")/100000)

## Strings and factors: one-hot vector

A one-hot vector is a way of representing a categorical feature where every observation has a vector in which all elements are zero except for at most one element, which has a value of one (1).

Each element in the vector corresponds to a level of the feature, so it's possible to tell what the right level is by seeing which element of the vector is equal to one (1).

The first step to encoding your categorical feature is to create a StringIndexer. Members of this class are Estimators that take a DataFrame with a column of strings and map each unique string to a number. Then, the Estimator returns a Transformer that takes a DataFrame, attaches the mapping to it as metadata, and returns a new DataFrame with a numeric column corresponding to the string column.

The second step is to encode this numeric column as a one-hot vector using a OneHotEncoder. This works exactly the same way as the StringIndexer by creating an Estimator and then a Transformer. The end result is a column that encodes your categorical feature as a vector that's suitable for machine learning routines!

Import pyspark.ml.features

# Create a StringIndexer

dest\_indexer = StringIndexer(inputCol="dest", outputCol="dest\_index")

# Create a OneHotEncoder

dest\_encoder = OneHotEncoder(inputCol="dest\_index", outputCol="dest\_fact")

**Assemble a vector**

The last step in the Pipeline is to combine all of the **columns** containing our features into a single column. This has to be done before modeling can take place because every Spark modeling routine expects the data to be in this form. You can do this by storing each of the values from a column as an entry in a vector. Then, from the model's point of view, every observation is a vector that contains all of the information about it and a label that tells the modeler what value that observation corresponds to.

VectorAssembler. This Transformer takes all of the columns you specify and combines them into a new vector.

vec\_assembler = VectorAssembler(inputCols=["month", "air\_time", "carrier\_fact", "dest\_fact", "plane\_age"], outputCol="features")

## Create the pipeline

Pipeline is a class in the pyspark.ml module that combines all the Estimators and Transformers that you've already created. This lets you reuse the same modeling process over and over again by wrapping it up in one simple object.

stages should be a list holding all the stages you want your data to go through in the pipeline. Here this is just

# Import Pipeline

from pyspark.ml import Pipeline

# Make the pipeline

flights\_pipe = Pipeline(stages= [dest\_indexer, dest\_encoder, carr\_indexer, carr\_encoder, vec\_assembler])

## Test vs Train

After you've cleaned your data and gotten it ready for modeling, one of the most important steps is to split the data into a test set and a train set. After that, don't touch your test data until you think you have a good model! As you're building models and forming hypotheses, you can test them on your training data to get an idea of their performance. By evaluating your model with a test set you can get a good idea of performance on new data.

In Spark it's important to make sure you split the data **after** all the transformations. This is because operations like StringIndexer don't always produce the same index even when given the same list of strings.

# Fit and transform the data:

#Create the DataFrame by calling the Pipeline methods .fit() and .transform() in a chain

piped\_data = flights\_pipe.fit(model\_data).transform(model\_data)

Split the data

# Split the data into training and test sets. training= 60% & test with 40% of the data.

training, test = piped\_data.randomSplit([.6, .4])

Model to Fit

You'll tune the model by testing different values for several *hyperparameters*.

A *hyperparameter* is just a value in the model that's not estimated from the data, but rather is supplied by the user to maximize performance.

# Create the modeler

The Estimator you'll be using is a LogisticRegression from the pyspark.ml.classification submodule.

# Import LogisticRegression

from pyspark.ml.classification import LogisticRegression

# Create a LogisticRegression Estimator

lr = LogisticRegression()

# Cross validation

k-fold cross validation. This is a method of estimating the model's performance on unseen data (like your test DataFrame).

It works by splitting the training data into a few different partitions. Once the data is split up, one of the partitions is set aside, and the model is fit to the others. Then the error is measured against the held out partition. This is repeated for each of the partitions, so that every block of data is held out and used as a test set exactly once. Then the error on each of the partitions is averaged. This is called the cross validation error of the model, and is a good estimate of the actual error on the held out data.

Cross validation helps to choose the hyperparameters by creating a grid of the possible pairs of values for the two hyperparameters, elasticNetParam and regParam, and using the cross validation error to compare all the different models so you can choose the best one!

# Create the evaluator

The first thing you need when doing cross validation for model selection is a way to compare different models.

For instance, the area under the ROC. This is a metric that combines the two kinds of errors a binary classifier can make (false positives and false negatives) into a simple number.

*# Import the evaluation submodule*

import pyspark.ml.evaluation as evals

*# Create a BinaryClassificationEvaluator*

evaluator = evals.BinaryClassificationEvaluator(metricName="areaUnderROC")

# Make a grid

Next, you need to create a grid of values to search over when looking for the optimal hyperparameters.

The .addGrid()method takes a model parameter (an attribute of the model Estimator, lr, that you created a few exercises ago) and a list of values that you want to try. The .build() method takes no arguments, it just returns the grid that you'll use later.

*# Import the tuning submodule*

import pyspark.ml.tuning as tune

import numpy as np

*# Create the parameter grid*

grid = tune.ParamGridBuilder()

*# Add the hyperparameter*

grid = grid.addGrid(lr.regParam, np.arange(0, .1, .01))

grid = grid.addGrid(lr.elasticNetParam, [0, 1])

# np arange: creates a list of numbers from 0 to .1, incrementing by .01.

*# Build the grid*

grid = grid.build()

# Make the validator

CrossValidator takes the modeler you want to fit, the grid of hyperparameters you created, and the evaluator you want to use to compare your models.

# Create the CrossValidator

cv = tune.CrossValidator(estimator=lr,

estimatorParamMaps=grid,

evaluator=evaluator

)

# Fit the model(s)

Unfortunately, cross validation is a very computationally intensive procedure. Fitting all the models would take too long. Remember, the training data is called training and you're using lr to fit a logistic regression model. Cross validation selected the parameter values regParam=0 and elasticNetParam=0 as being the best.

# Call lr.fit() on training data

best\_lr = lr.fit(training)

# Print best\_lr

print(best\_lr) # lr object

# Evaluate the model

AUC, or area under the curve. In this case, the curve is the ROC, or receiver operating curve. The details of what these things actually measure isn't important for this course. All you need to know is that for our purposes, the closer the AUC is to one (1), the better the model is!

*# Use the model to predict the test set*

test\_results = best\_lr.transform(test)

*# Evaluate the predictions*

print(evaluator.evaluate(test\_results))