Reference

DataCamp course

Getting to know PySpark

In this chapter, you'll learn how Spark manages data and how can you read and write tables from Python.

What is Spark, anyway?

Spark is a platform for cluster computing. Spark lets you **spread data and computations over clusters with multiple nodes**

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**.

Deciding whether or not Spark is the best solution for your problem takes some experience, but you can consider questions like:

Is my data too big to work with on a single machine?
Can my calculations be easily parallelized?

Using Spark in Python

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.

When you're just getting started with Spark it's simpler to just run a cluster locally. Thus, for this course, instead of connecting to another computer, all computations will be run on DataCamp's servers in a simulated cluster.

Creating the connection is as simple as creating an instance of the SparkContext class. 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. Take a look at the documentation for all the details!

For the rest of this course you'll have a SparkContext called sc already available in your workspace.

How do you connect to a Spark cluster from PySpark?

Answer: Create an instance of the SparkContext class.

Examining The SparkContext

You'll probably notice that code takes longer to run than you might expect. This is because Spark is some serious software. It takes more time to start up than you might be used to. You may also find that running simpler computations might take longer than expected. That's because all the optimizations that Spark has under its hood are designed for complicated operations with big data sets. **That means that for simple or small problems Spark may actually perform worse than some other solutions!**

In [ ]:
```python
# Verify SparkContext
print(sc)

# Print Spark version
print(sc.version)
```

Using DataFrames

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 in this course you'll be using the Spark DataFrame abstraction built on top of RDDs**.

The Spark DataFrame was designed to behave a lot like a SQL table (a table with variables in the columns and observations in the rows). Not only are they easier to understand, DataFrames are also more optimized for complicated operations than RDDs.

When you start modifying and combining columns and rows of data, there are many ways to arrive at the same result, but some often take much longer than others. **When using RDDs, it's up to the data scientist to figure out the right way to optimize the query, but the DataFrame implementation has much of this optimization built in!**

To start working with Spark DataFrames, you first have to create a SparkSession object from your SparkContext. You can think of the SparkContext as your connection to the cluster and the SparkSession as your interface with that connection.

Remember, for the rest of this course you'll have a SparkSession called spark available in your workspace!

Which of the following is an advantage of Spark DataFrames over RDDs?
Answer: Operations using DataFrames are automatically optimized.

Creating a SparkSession

We've already created a SparkSession for you called spark, but what if you're not sure there already is one? Creating multiple SparkSessions and SparkContexts can cause issues, so it's best practice to use the SparkSession.builder.getOrCreate() method. This returns an existing SparkSession if there's already one in the environment, or creates a new one if necessary!

In [ ]:
```
# Import SparkSession from pyspark.sql
from pyspark.sql import SparkSession

# Create my_spark
my_spark = SparkSession.builder.getOrCreate()

# Print my_spark
print(my_spark)
```

Viewing tables

Your SparkSession has an attribute called catalog which lists all the data inside the cluster. This attribute has a few methods for extracting different pieces of information.

One of the most useful is the .listTables() method, which returns the names of all the tables in your cluster as a list.

In [ ]:
```
# Print the tables in the catalog
print(spark.catalog.listTables())
```

Output:
[Table(name='flights', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]

Are you query-ious?

One of the advantages of the DataFrame interface is that you can run SQL queries on the tables in your Spark cluster.

As you saw in the last exercise, one of the tables in your cluster is the flights table. This table contains a row for every flight that left Portland International Airport (PDX) or Seattle-Tacoma International Airport (SEA) in 2014 and 2015.

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!

The table flights is only mentioned in the query, but 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.

Remember, we've already created a SparkSession called spark in your workspace.

In [ ]:
```
# Don't change this query
query = "FROM flights SELECT * LIMIT 10"
# Comments: This is a little different from usual SQL select statement.

# Get the first 10 rows of flights
flights10 = spark.sql(query)

# Show the results
flights10.show()
```

Pandafy a Spark DataFrame

Suppose you've run a query on your huge dataset and aggregated it down to something a little more manageable.

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. It's as simple as that!

This time the query counts the number of flights to each airport from SEA and PDX.

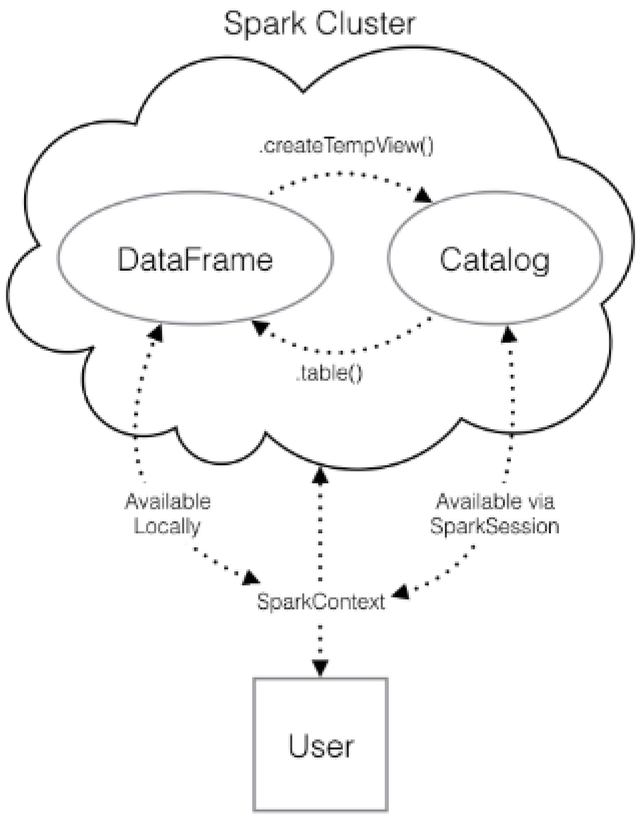
```
In [ ]: # Don't change this query
query = "SELECT origin, dest, COUNT(*) as N FROM flights GROUP BY origin, dest"

# Run the query
flight_counts = spark.sql(query)

# Convert the results to a pandas DataFrame
pd_counts = flight_counts.toPandas()
## This pd_counts will be local, right?

# Print the head of pd_counts
print(pd_counts.head())
```

```
  origin dest    N
0    SEA  RNO    8
1    SEA  DTW   98
2    SEA  CLE    2
3    SEA  LAX  450
4    PDX  SEA  144
```

Put some Spark in your data

- In the last exercise, you saw how to move data from Spark to pandas. However, maybe you want to go the other direction, and put a pandas DataFrame into a Spark cluster! The SparkSession class has a method for this as well. 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. Comments: **The created spark DataFrame is different from the original Pandas DataFrame.**
- For example, a SQL query (using the .sql() method) that references your DataFrame will throw an error (Because we cannot access a Spark DataFrame if it is only stored locally?). To access the data in this way (e.g. using .sql), 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. **Comments: we can use .sql() method to access the table only when we register the DataFrame as a table in the catalog.**

- 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. You'll use this method to avoid running into problems with duplicate tables.



```
In [ ]: # Create pd_temp
        pd_temp = pd.DataFrame(np.random.random(10))

        # Create spark_temp from pd_temp
        spark_temp = spark.createDataFrame(pd_temp)
        # takes a pandas DataFrame and returns a Spark DataFrame. But this is stored locally.

        # Examine the tables in the catalog
        print(spark.catalog.listTables())

        # Add spark_temp to the catalog
        spark_temp.createOrReplaceTempView("temp")

        # Examine the tables in the catalog again
        print(spark.catalog.listTables())
```

```
[Table(name='flights', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]
[Table(name='flights', database=None, description=None, tableType='TEMPORARY', isTemporary=True), Table
(name='temp', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]
```

Dropping the middle man

Now you know how to put data into Spark via pandas, but you're probably wondering why deal with pandas at all? Wouldn't it be easier to just read a text file straight into Spark? Of course it would!

Luckily, your SparkSession has a .read attribute which has several methods for reading different data sources into Spark DataFrames. Using these you can create a DataFrame from a .csv file just like with regular pandas DataFrames!

The variable file_path is a string with the path to the file airports.csv. This file contains information about different airports all over the world.

A SparkSession named spark is available in your workspace.

In [ ]:
```python
# Don't change this file path
file_path = "/usr/local/share/datasets/airports.csv"

# Read in the airports data
airports = spark.read.csv(file_path, header=True)

# Show the data
airports.show()
```

Manipulating data

In this chapter, you'll learn about the pyspark.sql module, which provides optimized data queries to your Spark session.

Comments. This seems treating the Spark session as a SQL server session.

Creating columns

- How to use the methods defined by Spark's DataFrame class to perform common data operations.
- Performing column-wise operations: 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.
- The new column must be an object of class Column. Creating one of these is as easy as extracting a column from your DataFrame using df.colName.
- Updating a Spark DataFrame is somewhat different than working in pandas because the **Spark DataFrame is immutable. This means that it can't be changed, and so columns can't be updated in place**.
- 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!

Remember, a SparkSession called spark is already in your workspace.

In [ ]:
```
# Create the DataFrame flights
flights = spark.table("flights")

# Show the head
print(flights.show())

# Add duration_hrs
flights = flights.withColumn("duration_hrs", flights.air_time/60)
```

Now you can make new columns derived from the old ones!

Filtering Data

Let's take a look at 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 or the WHERE clause of a SQL expression as a string.

For example, the following two expressions will produce the same output:

flights.filter(flights.air_time > 120).show()
flights.filter("air_time > 120").show()
Remember, a SparkSession called spark is already in your workspace, along with the Spark DataFrame flights.

In [ ]:
```
# Filter flights with a SQL string
long_flights1 = flights.filter("distance > 1000")
#flights is a Spark DataFrame

# Filter flights with a boolean column
long_flights2 = flights.filter(flights.distance > 1000)

# Examine the data to check they're equal
print(long_flights1.show())
print(long_flights2.show())
```

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). 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().

Comments:

- SqlAlchemy has its own version of SQL like syntax which can be used with many flavors of SQL.
- Here Spark also has its own version of SQL like syntax.

In [ ]:
```python
# Select the first set of columns
selected1 = flights.select("tailnum", "origin", "dest")

# Select the second set of columns
temp = flights.select(flights.origin, flights.dest, flights.carrier)

# Define first filter
filterA = flights.origin == "SEA"

# Define second filter
filterB = flights.dest == "PDX"

# Filter the data, first by filterA then by filterB
selected2 = temp.filter(filterA).filter(filterB)
```

Selecting II

- Similar to SQL, you can also use the .select() method to perform column-wise operations. 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. For example,

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"))

The equivalent Spark DataFrame method .selectExpr() takes SQL expressions as a string:

flights.selectExpr("air_time/60 as duration_hrs")

with the SQL `as` keyword being equivalent to the .alias() method. To select multiple columns, you can pass multiple strings.

Remember, a SparkSession called spark is already in your workspace, along with the Spark DataFrame flights.

```
In [ ]: # Define avg_speed
        avg_speed = (flights.distance/(flights.air_time/60)).alias("avg_speed")

        # Select the correct columns
        speed1 = flights.select("origin", "dest", "tailnum", avg_speed)
        # Unlike the .withColumn method before, this should not have created a new column avg_speed, right? Just output

        # Create the same table using a SQL expression
        speed2 = flights.selectExpr("origin", "dest", "tailnum", "distance/(air_time/60) as avg_speed")
```

Aggregating

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

```
In [ ]: # Find the shortest flight from PDX in terms of distance
        flights.filter(flights.origin == "PDX").groupBy().min("distance").show()

        # Find the longest flight from SEA in terms of duration
        flights.filter(flights.origin == "SEA").groupBy().max("air_time").show()
```

Aggregating II

To get you familiar with more of the built in aggregation methods, here's a few more exercises involving the flights table!

Remember, a SparkSession called spark is already in your workspace, along with the Spark DataFrame flights.

In [ ]:
```
# Average duration of Delta flights
flights.filter(flights.carrier == "DL").filter(flights.origin == "SEA").groupBy().avg("air_time").show()

# Total hours in the air
flights.withColumn("duration_hrs", flights.air_time/60).groupBy().sum("duration_hrs").show()
```

Grouping and Aggregating I

In [ ]:
```
# Group by tailnum
by_plane = flights.groupBy("tailnum")

# Number of flights each plane made
by_plane.count().show()

# Group by origin
by_origin = flights.groupBy("origin")

# Average duration of flights from PDX and SEA
by_origin.avg("air_time").show()
```

Grouping and Aggregating II

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.

This submodule contains many useful functions for computing things like standard deviations. All the aggregation functions in this submodule take the name of a column in a GroupedData table.

Remember, a SparkSession called spark is already in your workspace, along with the Spark DataFrame flights. The grouped DataFrames you created in the last exercise are also in your workspace.

In [ ]:
```python
# Import pyspark.sql.functions as F
import pyspark.sql.functions as F

# 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()
```

Joining II

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. In this course we'll always use the value how="leftouter".

The flights dataset and a new dataset called airports are already in your workspace.

In [ ]:
```python
# Examine the data
print(airports.show())

# Rename the faa column
airports = airports.withColumnRenamed("faa", "dest")

# Join the DataFrames
flights_with_airports = flights.join(airports, on="dest", how="leftouter")

# Examine the data again
print(flights_with_airports.show())
```

Getting started with machine learning pipelines

PySpark provides cutting edge machine learning routines built in, along with utilities for creating full machine learning pipelines.

Machine Learning Pipelines

In the next two chapters you'll step through every stage of the machine learning pipeline, from data intake to model evaluation.

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.

Which of the following is not true about machine learning in Spark?

1. Spark's algorithms give better results than other algorithms.
2. Working in Spark allows you to create reproducible machine learning pipelines.
3. Machine learning pipelines in Spark are made up of Transformers and Estimators.
4. PySpark uses the pyspark.ml submodule to interface with Spark's machine learning routines.

Answer 1

Join the DataFrames

In the next two chapters you'll be working to build a model that predicts whether or not a flight will be delayed based on the flights data we've been working with. This model will also include information about the plane that flew the route, so the first step is to join the two tables: flights and planes!

In [ ]:
```
# Rename year column
planes = planes.withColumnRenamed("year", "plane_year")

# Join the DataFrames
model_data = flights.join(planes, on="tailnum", how="leftouter")
```

Data types

- It's important to know that Spark only handles numeric data. That means all of the columns in your DataFrame must be either integers or decimals (called 'doubles' in Spark).
- When we imported our data, we let Spark guess what kind of information each column held. Unfortunately, Spark doesn't always guess right and you can see that some of the columns in our DataFrame are strings containing numbers as opposed to actual numeric values.
- 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.
- The only argument you need to pass to .cast() is the kind of value you want to create, in string form. For example, to create integers, you'll pass the argument "integer" and for decimal numbers you'll use "double".
- You can put this call to .cast() inside a call to .withColumn() to overwrite the already existing column, just like you did in the previous chapter!

What kind of data does Spark need for modeling?

1. Doubles
2. Integers
3. Decimals
4. Numeric
5. Strings

Answer: Numeric (integer or double)

String to integer

In [ ]:
```
# Cast the columns to integers
model_data = model_data.withColumn("arr_delay", model_data.arr_delay.cast("integer"))
model_data = model_data.withColumn("air_time", model_data.air_time.cast("integer"))
model_data = model_data.withColumn("month", model_data.month.cast("integer"))
model_data = model_data.withColumn("plane_year", model_data.plane_year.cast("integer"))
```

Create a new column

```
In [ ]: # Create the column plane_age
        model_data = model_data.withColumn("plane_age", model_data.year - model_data.plane_year)
```

Making a Boolean

```
In [ ]: # Create is_late
        model_data = model_data.withColumn("is_late", model_data.arr_delay > 0)

        # Convert to an integer
        model_data = model_data.withColumn("label", model_data.is_late.cast("integer"))

        # 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")
```

Strings and factors

Create what are called 'one-hot vectors' to represent the carrier and the destination of each flight.

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!

Carrier

In this exercise you'll create a StringIndexer and a OneHotEncoder to code the carrier column. To do this, you'll call the class constructors with the arguments inputCol and outputCol.

The inputCol is the name of the column you want to index or encode, and the outputCol is the name of the new column that the Transformer should create.

In [ ]:
```python
# Create a StringIndexer
carr_indexer = StringIndexer(inputCol="carrier", outputCol="carrier_index")

# Create a OneHotEncoder
carr_encoder = OneHotEncoder(inputCol="carrier_index", outputCol="carrier_fact")
```

Destination

Now you'll encode the dest column just like you did in the previous exercise.

In [ ]:
```python
# 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.

Because of this, the pyspark.ml.feature submodule contains a class called VectorAssembler. This Transformer takes all of the columns you specify and combines them into a new vector column.

In [ ]:
```python
# Make a VectorAssembler
vec_assembler = VectorAssembler(inputCols=["month", "air_time", "carrier_fact", "dest_fact", "plane_age"], outpu
```

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.

In [ ]:
```
# 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

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.

Transform the data

Hooray, now you're finally ready to pass your data through the Pipeline you created!

In [ ]:
```
# Fit and transform the data
piped_data = flights_pipe.fit(model_data).transform(model_data)
```

Split the data

Now that you've done all your manipulations, the last step before modeling is to split the data!

In [ ]:
```
# Split the data into training and test sets
training, test = piped_data.randomSplit([.6, .4])
```

Model tuning and selection

In this last chapter, you'll apply what you've learned to create a model that predicts what flights will be delayed!

Create the modeler

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

In [ ]:
```python
# Import LogisticRegression
from pyspark.ml.classification import LogisticRegression

# Create a LogisticRegression Estimator
lr = LogisticRegression()
```

Create the evaluator

This evaluator calculates 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. You'll learn more about this towards the end of the chapter!

In [ ]:
```python
# 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 submodule pyspark.ml.tuning includes a class called ParamGridBuilder that does just that (maybe you're starting to notice a pattern here; PySpark has a submodule for just about everything!).

You'll need to use the .addGrid() and .build() methods to create a grid that you can use for cross validation. 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.

In [ ]:
```python
# Import the tuning submodule
import pyspark.ml.tuning as tune

# 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])

# Build the grid
grid = grid.build()
```

Make the validator

The submodule pyspark.ml.tuning also has a class called CrossValidator for performing cross validation. This Estimator takes the modeler you want to fit, the grid of hyperparameters you created, and the evaluator you want to use to compare your models.

The submodule pyspark.ml.tune has already been imported as tune. You'll create the CrossValidator by passing it the logistic regression Estimator lr, the parameter grid, and the evaluator you created in the previous exercises.

In [ ]:
```python
# Create the CrossValidator
cv = tune.CrossValidator(estimator=lr,
                         estimatorParamMaps=grid,
                         evaluator=evaluator
                         )
```

Fit the model(s)

You're finally ready to fit the models and select the best one!

Unfortunately, cross validation is a very computationally intensive procedure. Fitting all the models would take too long on DataCamp.

To do this locally you would use the code

#Fit cross validation models
models = cv.fit(training)

#Extract the best model
best_lr = models.bestModel

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. These are the default values, so you don't need to do anything else with lr before fitting the model.

In [ ]:
```
# Call lr.fit()
best_lr = lr.fit(training)

# Print best_lr
print(best_lr)
```

Evaluating binary classifiers

For this course we'll be using a common metric for binary classification algorithms call the 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!

If you've created a perfect binary classification model, what would the AUC be?

Answer 1

Evaluate the model

Remember the test data that you set aside waaaaaay back in chapter 3? It's finally time to test your model on it! You can use the same evaluator you made to fit the model.

In [ ]:
```
# Use the model to predict the test set
test_results = best_lr.transform(test)

# Evaluate the predictions
print(evaluator.evaluate(test_results))
```

```
0.7125950520013005
```