Introduction to PySpark

1. Getting to know PySpark

1.1 What is Spark, anyway?

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.

However, with greater computing power comes greater complexity.

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?

1.2 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 *worker*. The master sends the workers 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!

1.3 Examining The SparkContext

In this exercise you'll get familiar with 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!

Instructions

Get to know the `SparkContext`.

- Call `print()` on `sc` to verify there's a `SparkContext` in your environment.
- `print()` `sc.version` to see what version of Spark is running on your cluster.

```
# Verify SparkContext
print(sc)

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

1.4 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.

1.5 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 `SparkSession`s and `SparkContext`s 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!

Instructions

- Import `SparkSession` from `pyspark.sql`.
- Make a new `SparkSession` called `my_spark` using `SparkSession.builder.getOrCreate()`.
- Print `my_spark` to the console to verify it's a `SparkSession`.

```
# Import SparkSession from pyspark.sql
from pyspark.sql import SparkSession

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

# Print my_spark
print(my_spark)
```

1.6 Viewing tables

Once you've created a `SparkSession`, you can start poking around to see what data is in your cluster!

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.

Instructions

- See what tables are in your cluster by calling `spark.catalog.listTables()` and printing the result!

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

1.7 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. If you don't have any experience with SQL, don't worry, we'll provide you with queries! (To learn more SQL, start with our **Introduction to SQL** course.)

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!

If 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.

Remember, we've already created a `SparkSession` called `spark` in your workspace. (It's no longer called `my_spark` because we created it for you!)

Instructions

- Use the `.sql()` method to get the first 10 rows of the `flights` table and save the result to `flights10`. The variable `query` contains the appropriate SQL query.
- Use the DataFrame method `.show()` to print `flights10`.

```
# Don't change this query
query = "FROM flights SELECT * LIMIT 10"

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

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

1.8 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.

Remember, there's already a `SparkSession` called `spark` in your workspace!

Instructions

- Run the query using the `.sql()` method. Save the result in `flight_counts`.
- Use the `.toPandas()` method on `flight_counts` to create a `pandas` DataFrame called `pd_counts`.
- Print the `.head()` of `pd_counts` to the console.

```
# Don't change this query
query = "SELECT origin, dest, COUNT(*) as N FROM flights GROUP B

# 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())
```

1.9 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.

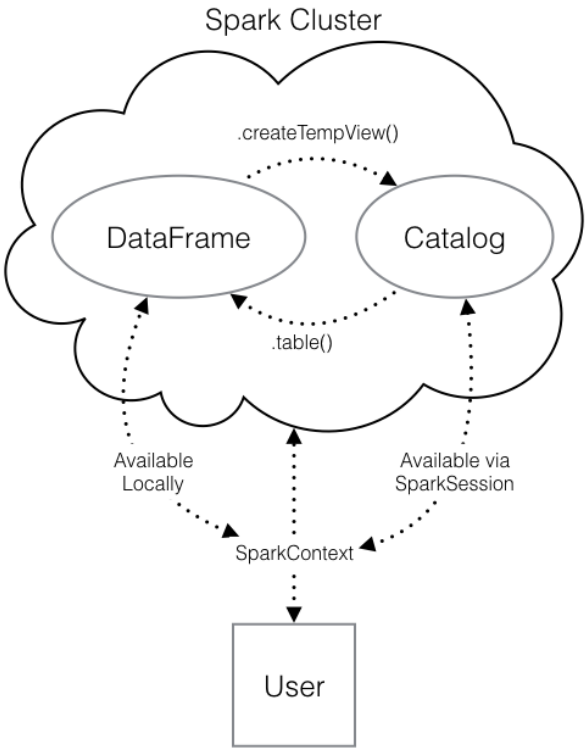
The output of this method is stored locally, not in the `SparkSession` catalog. This means that you can use all the Spark DataFrame methods on it, but you can't access the data in other contexts.

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

Check out the diagram to see all the different ways your Spark data structures interact with each other.



Instructions

- The code to create a `pandas` DataFrame of random numbers has already been provided and saved under `pd_temp`.
- Create a Spark DataFrame called `spark_temp` by calling the Spark method `.createDataFrame()` with `pd_temp` as the argument.
- Examine the list of tables in your Spark cluster and verify that the new DataFrame is *not* present. Remember you can use `spark.catalog.listTables()` to do so.
- Register the `spark_temp` DataFrame you just created as a temporary table using the `.createOrReplaceTempView()` method. THe temporary table should be named `"temp"`. Remember that the table name is set including it as the only argument to your method!
- Examine the list of tables again.

```
# Create pd_temp
pd_temp = pd.DataFrame(np.random.random(10))

# Create spark_temp from pd_temp
spark_temp = spark.createDataFrame(pd_temp)

# 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())
```

1.10 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.

Instructions

- Use the `.read.csv()` method to create a Spark DataFrame called `airports`
  - The first argument is `file_path`
  - Pass the argument `header=True` so that Spark knows to take the column names from the first line of the file.
- Print out this DataFrame by calling `.show()`.

```
# 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()
```

2. Manipulating Data

2.1 Creating columns

In this chapter, you'll learn how to use the methods defined by Spark's `DataFrame` class to perform common data operations.

Let's look at 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.

Instructions

- Use the `spark.table()` method with the argument `"flights"` to create a DataFrame containing the values of the `flights` table in the `.catalog`. Save it as `flights`.
- Show the head of `flights` using `flights.show()`. Check the output: the column `air_time` contains the duration of the flight in minutes.

- Update `flights` to include a new column called `duration_hrs`, that contains the duration of each flight in hours (you'll need to divide `air_time` by the number of minutes in an hour).

```
# Create the DataFrame flights
flights = spark.table("flights")

# Show the head
flights.show()

# Add duration_hrs
flights = flights.withColumn('duration_hrs' , flights.air_time /
```

2.2 SQL in a nutshell

As you move forward, it will help to have a basic understanding of SQL. A more in depth look can be found **here**.

A SQL query returns a table derived from one or more tables contained in a database.

Every SQL query is made up of commands that tell the database what you want to do with the data. The two commands that every query has to contain are `SELECT` and `FROM`.

The `SELECT` command is followed by the *columns* you want in the resulting table.

The `FROM` command is followed by the name of the table that contains those columns. The minimal SQL query is:

```
SELECT * FROM my_table;
```

The `*` selects all columns, so this returns the entire table named `my_table`.

Similar to `.withColumn()`, you can do column-wise computations within a `SELECT` statement. For example,

```
SELECT origin, dest, air_time / 60 FROM flights;
```

returns a table with the origin, destination, and duration in hours for each flight.

Another commonly used command is `WHERE`. This command filters the rows of the table based on some logical condition you specify. The resulting table contains the rows where your condition is true. For example, if you had a table of students and grades you could do:

```
SELECT * FROM students
WHERE grade = 'A';
```

to select all the columns and the rows containing information about students who got As.

2.3 SQL in a nutshell (2)

Another common database task is aggregation. That is, reducing your data by breaking it into chunks and summarizing each chunk.

This is done in SQL using the `GROUP BY` command. This command breaks your data into groups and applies a function from your `SELECT` statement to each group.

For example, if you wanted to count the number of flights from each of two origin destinations, you could use the query

```
SELECT COUNT(*) FROM flights
GROUP BY origin;
```

`GROUP BY origin` tells SQL that you want the output to have a row for each unique value of the `origin` column. The `SELECT` statement selects the values you want to populate each of the columns. Here, we want to `COUNT()` every row in each of the groups.

It's possible to `GROUP BY` more than one column. When you do this, the resulting table has a row for every combination of the unique values in each column. The following query counts the number of flights from SEA and PDX to every destination airport:

```
SELECT origin, dest, COUNT(*) FROM flights
GROUP BY origin, dest;
```

The output will have a row for every combination of the values in `origin` and `dest` (i.e. a row listing each origin and destination that a flight flew to). There will also be a column with the `COUNT()` of all the rows in each group.

Remember, a more in depth look at SQL can be found **here**.

2.4 Filtering Data

Now that you have a bit of SQL know-how under your belt, it's easier to talk about the analogous operations using Spark DataFrames.

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 an expression that would follow the `WHERE` clause of a SQL expression as a string, or a Spark Column of boolean (`True`/`False`) values.

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

```
flights.filter("air_time > 120").show()
flights.filter(flights.air_time > 120).show()
```

Notice that in the first case, we pass a *string* to `.filter()`. In SQL, we would write this filtering task as `SELECT * FROM flights WHERE air_time > 120`. Spark's `.filter()` can accept any expression that could go in the `WHERE` clause of a SQL query (in this case, `"air_time > 120"`), as long as it is passed as a string. Notice that in this case, we do not reference the name of the table in the string -- as we wouldn't in the SQL request.

In the second case, we actually pass a *column of boolean values* to `.filter()`. Remember that `flights.air_time > 120` returns a column of boolean values that has `True` in place of those records in `flights.air_time` that are over 120, and `False` otherwise.

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

Instructions

- Use the `.filter()` method to find all the flights that flew over 1000 miles two ways:
  - First, pass a SQL **string** to `.filter()` that checks whether the distance is greater than 1000. Save this as `long_flights1`.
  - Then pass a column of boolean values to `.filter()` that checks the same thing. Save this as `long_flights2`.
- Use `.show()` to print heads of both DataFrames and make sure they're actually equal!

```
# Filter flights by passing a string
long_flights1 = flights.filter("distance > 1000")

# Filter flights by passing a column of boolean values
long_flights2 = flights.filter(flights.distance > 1000)

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

2.5 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()`.

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

Instructions

- Select the columns `"tailnum"`, `"origin"`, and `"dest"` from `flights` by passing the column names as strings. Save this as `selected1`.
- Select the columns `"origin"`, `"dest"`, and `"carrier"` using the `df.colName` syntax and then filter the result using both of the filters already defined for you (`filterA` and `filterB`) to only keep flights from SEA to PDX. Save this as `selected2`.

```
# 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.carr

# 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)
```

2.6 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`.

Instructions

Create a table of the average speed of each flight both ways.

- Calculate average speed by dividing the `distance` by the `air_time` (converted to hours). Use the `.alias()` method name this column `"avg_speed"`. Save the output as the variable `avg_speed`.
- Select the columns `"origin", "dest", "tailnum"`, and `avg_speed` (without quotes!). Save this as `speed1`.
- Create the same table using `.selectExpr()` and a string containing a SQL expression. Save this as `speed2`.

```
# Define avg_speed
avg_speed = (flights.distance/(flights.air_time/60)).alias("avg_

# Select the correct columns
speed1 = flights.select("origin", "dest", "tailnum", avg_speed)

# Create the same table using a SQL expression
speed2 = flights.selectExpr("origin", "dest", "tailnum", "distar
```

2.7 Aggregating

All of the common aggregation methods, like `.min()`, `.max()`, and `.count()` are `GroupedData` methods. These are created by calling the `.groupBy()` DataFrame method. You'll learn exactly what that means in a few exercises. For now, all you have to do to use these functions is call that method on your DataFrame. 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.

Now you're ready to do some aggregating of your own!

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

Instructions

- Find the length of the shortest (in terms of distance) flight that left PDX by first `.filter()`ing and using the `.min()` method. Perform the filtering by referencing the column directly, not passing a SQL string.
- Find the length of the longest (in terms of time) flight that left SEA by `filter()`ing and using the `.max()` method. Perform the filtering by referencing the column directly, not passing a SQL string.

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

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

2.8 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`.

Instructions

- Use the `.avg()` method to get the average air time of Delta Airlines flights (where the `carrier` column has the value `"DL"`) that left SEA. The place of departure is stored in the column `origin`. `show()` the result.
- Use the `.sum()` method to get the total number of hours all planes in this dataset spent in the air by creating a column called `duration_hrs` from the column `air_time`. `show()` the result.

```
# Average duration of Delta flights
flights.filter(flights.carrier == "DL").filter(flights.origin ==

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

2.9 Grouping and Aggregating I

Part of what makes aggregating so powerful is the addition of groups. PySpark has a whole class devoted to grouped data frames: `pyspark.sql.GroupedData`, which you saw in the last two exercises.

You've learned how to create a grouped DataFrame by calling the `.groupBy()` method on a DataFrame with no arguments.

Now you'll see that 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!

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

Instructions

- Create a DataFrame called `by_plane` that is grouped by the column `tailnum`.

- Use the `.count()` method with no arguments to count the number of flights each plane made.
- Create a DataFrame called `by_origin` that is grouped by the column `origin`.
- Find the `.avg()` of the `air_time` column to find average duration of flights from PDX and SEA.

```
# 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()
```

2.10 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.

Instructions

- Import the submodule `pyspark.sql.functions` as `F`.
- Create a `GroupedData` table called `by_month_dest` that's grouped by both the `month` and `dest` columns. Refer to the two columns by passing both strings as separate arguments.
- Use the `.avg()` method on the `by_month_dest` DataFrame to get the average `dep_delay` in each month for each destination.
- Find the standard deviation of `dep_delay` by using the `.agg()` method with the function `F.stddev()`.

```
# 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 of departure delay
by_month_dest.agg(F.stddev("dep_delay")).show()
```

2.11 Joining

Another very common data operation is the *join*. Joins are a whole topic unto themselves, so in this course we'll just look at simple joins. If you'd like to learn more about joins, you can take a look **here**.

A join will combine two different tables along a column that they share. This column is called the *key*. Examples of keys here include the `tailnum` and `carrier` columns from the `flights` table.

For example, suppose that you want to know more information about the plane that flew a flight than just the tail number. This information isn't in the `flights` table because the same plane flies many different flights over the course of two years, so including this information in every row would result in a lot of duplication. To avoid this, you'd have a second table that has only one row for each plane and whose columns list all the information about the plane, including its tail number. You could call this table `planes`

When you join the `flights` table to this table of airplane information, you're adding all the columns from the `planes` table to the `flights` table. To fill these columns with information, you'll look at the tail number from the `flights` table and find the matching one in the `planes` table, and then use that row to fill out all the new columns.

2.12 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.

Instructions

- Examine the `airports` DataFrame by calling `.show()`. Note which key column will let you join `airports` to the `flights` table.
- Rename the `faa` column in `airports` to `dest` by re-assigning the result of `airports.withColumnRenamed("faa", "dest")` to `airports`.

- Join the `flights` with the `airports` DataFrame on the `dest` column by calling the `.join()` method on `flights`. Save the result as `flights_with_airports`.
  - The first argument should be the other DataFrame, `airports`.
  - The argument `on` should be the key column.
  - The argument `how` should be `"leftouter"`.
- Call `.show()` on `flights_with_airports` to examine the data again. Note the new information that has been added.

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

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

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

# Examine the new DataFrame
flights_with_airports.show()
```

3. Getting Started with Machine Learning Pipelines

3.1 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`!

Instructions

- First, rename the `year` column of `planes` to `plane_year` to avoid duplicate column names.
- Create a new DataFrame called `model_data` by joining the `flights` table with `planes` using the `tailnum` column as the key.

```
# Rename year column
planes = planes.withColumnRenamed("year","plane_year")

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

3.2 Data types

Good work! Before you get started modeling, 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.

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.

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!

3.3 String to integer

Now you'll use the `.cast()` method you learned in the previous exercise to convert all the appropriate columns from your DataFrame `model_data` to integers!

To convert the type of a column using the `.cast()` method, you can write code like this:

```
dataframe = dataframe.withColumn("col", dataframe.col.cast("n
ew_type"))
```

Instructions

- Use the method `.withColumn()` to `.cast()` the following columns to type `"integer"`. Access the columns using the `df.col` notation:
  - `model_data.arr_delay`
  - `model_data.air_time`
  - `model_data.month`
  - `model_data.plane_year`

```
# Cast the columns to integers
model_data = model_data.withColumn("arr_delay", model_data.arr_
model_data = model_data.withColumn("air_time", model_data.air_t
model_data = model_data.withColumn("month", model_data.month.cas
model_data = model_data.withColumn("plane_year", model_data.plan
```

3.4 Create a new column

In the last exercise, you converted the column `plane_year` to an integer. This column holds the year each plane was manufactured. However, your model will use the planes' *age*, which is slightly different from the year it was made!

Instructions

- Create the column `plane_age` using the `.withColumn()` method and subtracting the year of manufacture (column `plane_year`) from the year (column `year`) of the flight.

```
# Create the column plane_age
model_data = model_data.withColumn("plane_age", model_data.year
```

3.5 Making a Boolean

Consider that you're modeling a yes or no question: is the flight late? However, your data contains the arrival delay in minutes for each flight. Thus, you'll need to create a boolean column which indicates whether the flight was late or not!

Instructions

- Use the `.withColumn()` method to create the column `is_late`. This column is equal to `model_data.arr_delay > 0`.
- Convert this column to an integer column so that you can use it in your model and name it `label` (this is the default name for the response variable in Spark's machine learning routines).
- Filter out missing values (this has been done for you).

```
# Create is_late
model_data = model_data.withColumn("is_late", model_data.arr_del

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

# Remove missing values
model_data = model_data.filter("arr_delay is not NULL and dep_de
```

3.6 Strings and factors

As you know, Spark requires numeric data for modeling. So far this hasn't been an issue; even boolean columns can easily be converted to integers without any trouble. But you'll also be using the airline and the plane's destination as features in your model. These are coded as strings and there isn't any obvious way to convert them to a numeric data type.

Fortunately, PySpark has functions for handling this built into the `pyspark.ml.features` submodule. You can create what are called 'one-hot vectors' to represent the carrier and the destination of each flight. 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 `Estimator`s 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!

This may seem complicated, but don't worry! All you have to remember is that you need to create a `StringIndexer` and a `OneHotEncoder`, and the `Pipeline` will take care of the rest.

3.7 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.

Instructions

- Create a `StringIndexer` called `carr_indexer` by calling `StringIndexer()` with `inputCol="carrier"` and `outputCol="carrier_index"`.
- Create a `OneHotEncoder` called `carr_encoder` by calling `OneHotEncoder()` with `inputCol="carrier_index"` and `outputCol="carrier_fact"`

```
# Create a StringIndexer
carr_indexer = StringIndexer(inputCol="carrier",outputCol="carri

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

3.8 Destination

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

Instructions

- Create a `StringIndexer` called `dest_indexer` by calling `StringIndexer()` with `inputCol="dest"` and `outputCol="dest_index"`.
- Create a `OneHotEncoder` called `dest_encoder` by calling `OneHotEncoder()` with `inputCol="dest_index"` and `outputCol="dest_fact"`.

```
# Create a StringIndexer
dest_indexer = StringIndexer(inputCol="dest",outputCol="dest_ind

# Create a OneHotEncoder
dest_encoder = OneHotEncoder(inputCol="dest_index",outputCol="de
```

3.9 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.

Instructions

- Create a `VectorAssembler` by calling `VectorAssembler()` with the `inputCols` names as a list and the `outputCol` name `"features"`.
  - The list of columns should be `["month", "air_time", "carrier_fact", "dest_fact", "plane_age"]`.

```
# Make a VectorAssembler
vec_assembler = VectorAssembler(inputCols=["month", "air_time",
```

3.10 Create the pipeline

You're finally ready to create a `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. Neat, right?

Instructions

- Import `Pipeline` from `pyspark.ml`.
- Call the `Pipeline()` constructor with the keyword argument `stages` to create a `Pipeline` called `flights_pipe`.
  - `stages` should be a list holding all the stages you want your data to go through in the pipeline. Here this is just: `[dest_indexer, dest_encoder, carr_indexer, carr_encoder, vec_assembler]`

```
# Import Pipeline
from pyspark.ml import Pipeline
```

```
# Make the pipeline
flights_pipe = Pipeline(stages=[dest_indexer, dest_encoder, carr
```

3.11 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.

Once you've got your favorite model, you can see how well it predicts the new data in your test set. This never-before-seen data will give you a much more realistic idea of your model's performance in the real world when you're trying to predict or classify 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.

3.12 Transform the data

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

Instructions

- Create the DataFrame `piped_data` by calling the `Pipeline` methods `.fit()` and `.transform()` in a chain. Both of these methods take `model_data` as their only argument.

```
# Fit and transform the data
piped_data = flights_pipe.fit(model_data).transform(model_data)
```

3.13 Split the data

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

Instructions

- Use the DataFrame method `.randomSplit()` to split `piped_data` into two pieces, `training` with 60% of the data, and `test` with 40% of the data by passing the list `[.6, .4]` to the `.randomSplit()` method.

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

4. Model tuning and selection

4.1 What is logistic regression?

The model you'll be fitting in this chapter is called a *logistic regression*. This model is very similar to a linear regression, but instead of predicting a numeric variable, it predicts the probability (between 0 and 1) of an event.

To use this as a classification algorithm, all you have to do is assign a cutoff point to these probabilities. If the predicted probability is above the cutoff point, you

classify that observation as a 'yes' (in this case, the flight being late), if it's below, you classify it as a 'no'!

You'll tune this 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. For this course it's not necessary to understand the mathematics behind all of these values - what's important is that you'll try out a few different choices and pick the best one.

4.2 Create the modeler

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

Instructions

- Import the `LogisticRegression` class from `pyspark.ml.classification`.
- Create a `LogisticRegression` called `lr` by calling `LogisticRegression()` with no arguments.

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

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

4.3 Cross validation

In the next few exercises you'll be tuning your logistic regression model using a procedure called *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. The exact number is up to you, but in this course you'll be using PySpark's default value of three. 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.

You'll be using cross validation 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!

4.5 Create the evaluator

The first thing you need when doing cross validation for model selection is a way to compare different models. Luckily, the `pyspark.ml.evaluation` submodule has classes for evaluating different kinds of models. Your model is a binary classification model, so you'll be using the `BinaryClassificationEvaluator` from the `pyspark.ml.evaluation` module.

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!

Instructions

- Import the submodule `pyspark.ml.evaluation` as `evals`.
- Create `evaluator` by calling `evals.BinaryClassificationEvaluator()` with the argument `metricName="areaUnderROC"`.

```
# Import the evaluation submodule
import pyspark.ml.evaluation as evals

# Create a BinaryClassificationEvaluator
evaluator = evals.BinaryClassificationEvaluator(metricName="area
```

4.6 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.

Instructions

- Import the submodule `pyspark.ml.tuning` under the alias `tune`.
- Call the class constructor `ParamGridBuilder()` with no arguments. Save this as `grid`.
- Call the `.addGrid()` method on `grid` with `lr.regParam` as the first argument and `np.arange(0, .1, .01)` as the second argument. This second call is a function from the `numpy` module (imported `as np`) that creates a list of numbers from 0 to .1, incrementing by .01. Overwrite `grid` with the result.
- Update `grid` again by calling the `.addGrid()` method a second time create a grid for `lr.elasticNetParam` that includes only the values `[0, 1]`.

- Call the `.build()` method on `grid` and overwrite it with the output.

```
# 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()
```

4.7 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.

Instructions

- Create a `CrossValidator` by calling `tune.CrossValidator()` with the arguments:
  - `estimator=lr`
  - `estimatorParamMaps=grid`

    - `evaluator=evaluator`
- Name this object `cv`.

```
# Create the CrossValidator
cv = tune.CrossValidator(estimator=lr,
               estimatorParamMaps=grid,
               evaluator=evaluator
               )
```

4.8 Fit the model(s)

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

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.

Instructions

- Create `best_lr` by calling `lr.fit()` on the `training` data.
- Print `best_lr` to verify that it's an object of the `LogisticRegressionModel` class.

```
# Call lr.fit()
best_lr = lr.fit(training)

# Print best_lr
print(best_lr)
```

4.9 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!

4.10 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.

Instructions

- Use your model to generate predictions by applying `best_lr.transform()` to the `test` data. Save this as `test_results`.
- Call `evaluator.evaluate()` on `test_results` to compute the AUC. Print the output.

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

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