Backlinks (1)

Spark

1. Filter by fraudulent transactions
2. What is most common category in all the transactions and just in the fraudulent ones?
3. What is the average amount for fraudulent and non-fraudulent transactions?
4. What is the maximum amount for fraudulent and non-fradulent transactions?
5. Which state has the highest rate of frajal transactions) ? - then filter by this state to investigate these transactions further
6. Create a table of people that have been affected by fraudulent transactions. Compare the number of people affected to the total number of fraudulent transactions.

Spark

- a platform for **cluster computing**.
- Purpose: 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?

Using Spark in Python

Step1: Connecting to a cluster by creating an instance of SparkContext class

Where is the cluster?

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

What is a master?

- 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.**

Why does the code take a long time to run?

- all the optimizations that Spark has under its hood are designed for complicated operations with big data sets.

In practice

- 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.

`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)
>  <SparkContext master=local[*] appName=pyspark-shell>

# Print Spark version
print(sc.version)
>   3.2.0
```

Using DataFrames

Step2: Creating a SparkSession object from your SparkContext

What is Spark's data structure?

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!

In practice

- 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!
- 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!

```
# Import SparkSession from pyspark.sql
from pyspark.sql import SparkSession
# Create my_spark
my_spark = SparkSession.builder.getOrCreate()
# Print my_spark
print(my_spark)
>  <pyspark.sql.session.SparkSession object at 0x7f8087ab44c0>
```

Step3: Creating a SparkSession object from your SparkContext

- 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.

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

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

Step4: Running SQL queries with the DataFrame interface

- You can run SQL queries on the tables in your Spark cluster with the DataFrame interface
- 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, 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.
- we've already created a `SparkSession` called `spark` in your workspace. (It's no longer called `my_spark` because we created it for you!)

```
# 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()
```

Spark to Pandas

Step5: Pandafy a Spark DataFrame using .toPandas( ) method

- Calling this method on a Spark DataFrame returns the corresponding `pandas` DataFrame.
- there's already a `SparkSession` called `spark` in your workspace!

```
# 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()

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

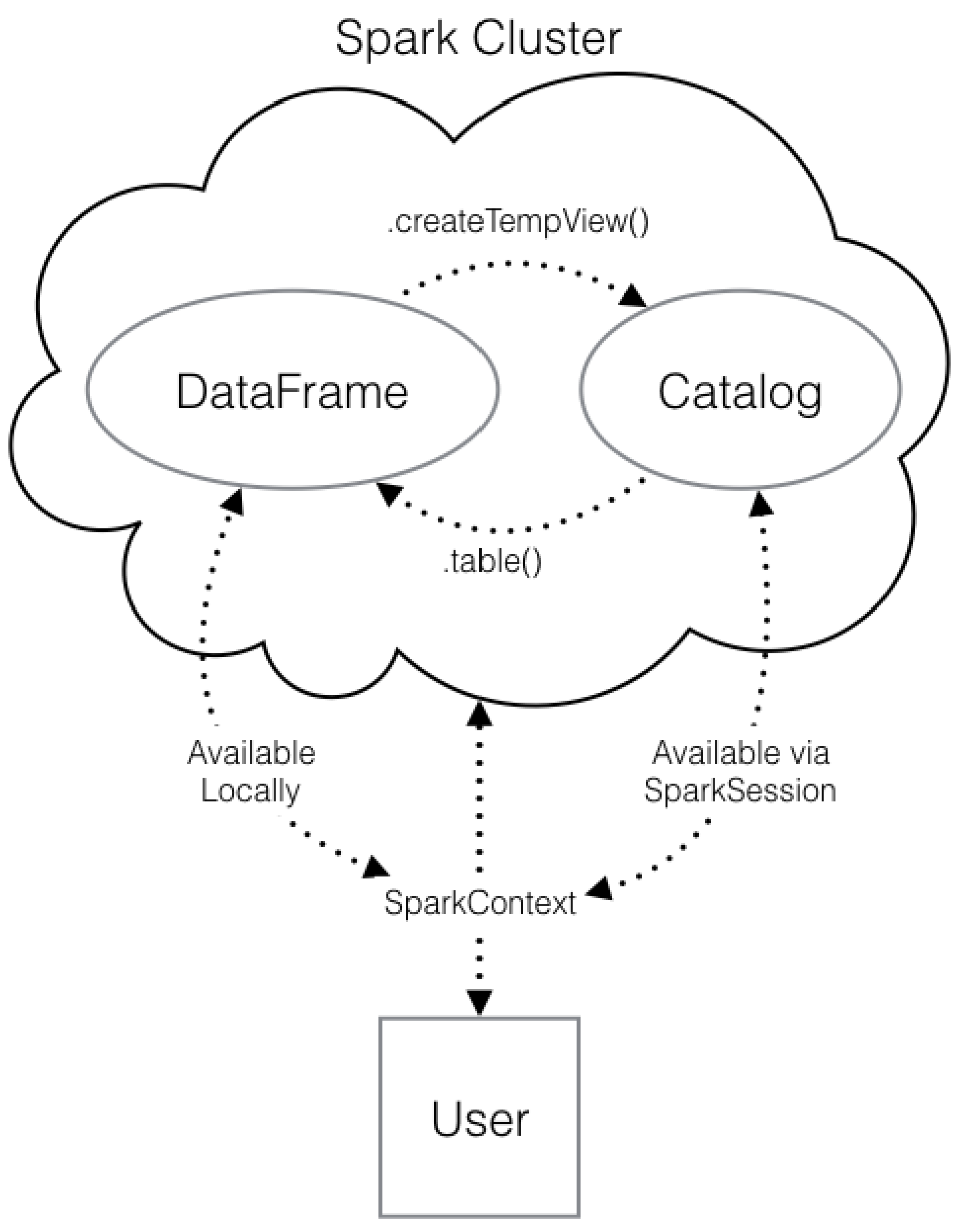
Pandas to Spark

Step6: Put Pandas DataFrame into a Spark cluster using .createDataFrame( )

- this 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 via SparkSession, **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.**
- There's already a `SparkSession` called `spark` in your workspace, `numpy` has been imported as `np`, and `pandas` as `pd`.

```
# 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())
```

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



Dropping the middle man

Read a text file straight into Spark

- 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.

```
# 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()
```

##Connection	# Verify SparkContext
	print(sc)
	
	# Print Spark version
	print(sc.version)

##Start session	# Import SparkSession from pyspark.sql
	from pyspark.sql import SparkSession
	
	# Create my_spark
	my_spark = SparkSession.builder.getOrCreate()
	
	# Print my_spark
	print(my_spark)

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

##SQL query	# 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()

## Pandafy a Spark DataFrame	# 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()
	
	# Print the head of pd_counts
	print(pd_counts.head())

##Put pandas DF into spark cluster	# 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())

##Read csv file	# 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()

Methods in Spark's `DataFrame` class to perform common data operations

Creating columns
.withColumn("name of column", df.colName)
• *immutable*.
• To overwrite an existing column, just pass the name of the column as the first argument
	# Create the DataFrame flights
	flights = spark.table("flights")
	
	# Show the head
	flights.show()
	
	# Add duration_hrs
	flights = flights.withColumn("duration_hrs", flights.air_time/60)

Filtering Data
`.filter()` method
• Spark counterpart of SQL's `WHERE` clause
• takes `WHERE` clause of a SQL expression as a string OR
• a Spark Column of boolean (`True` / `False`) values, like python
	# 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()

Selecting
• to drop columns
• can either be the column name as a string (one for each column) or a column object (using the `df.colName` syntax).
• can perform operations like addition or subtraction on the column to change the data contained in it, much like inside `.withColumn()`
• `.select()` returns only the columns you specify, while `.withColumn()` returns all the columns of the DataFrame in addition to the one you defined.
	# 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)

Select to perform column-wise operations
• `.alias()` method to rename a column you're selecting
• equivalent Spark DataFrame method `.selectExpr()` takes SQL expressions as a string
	# 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)
	
	# Create the same table using a SQL expression
	speed2 = flights.selectExpr("origin", "dest", "tailnum", "distance/(air_time/60) as avg_speed")

Aggregating
• .min(), .max(), and .count() are GroupedData
	# Find the shortest flight from PDX in terms of distance
	flights.filter(flights.origin == "PDX").groupBy().min("distance").show()

methods.
• created by calling the `.groupBy()` DataFrame method
• 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 longest flight from SEA in terms of air time
	flights.filter(flights.origin == "SEA").groupBy().max("air_time").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
	flights.withColumn("duration_hrs", flights.air_time/60).groupBy().sum("duration_hrs").show()

Grouping and Aggregating
• the aggregation methods behave like when you use a `GROUP BY` statement in a SQL query
• `.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
	# 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()
	
	# 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()

Joining	# 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 new DataFrame
	print(flights_with_airports.show())

Machine Learning Pipelines

Modeule/classes	Description
pyspark.ml module	has two classes: `Transformer` and `Estimator`
`Transformer` classes	• `.transform()` method that takes a DataFrame and returns a new DataFrame; usually the original one with a new column appended. • 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	Description
Spark only handles numeric data - integers or decimals (doubles)	
Change data types `.cast()` method in combination with the `.withColumn()` method	`.cast()` works on columns, while `.withColumn()` works on DataFrames • "integer", "double"

Cast to integer	# 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"))

Making a Boolean	# 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")

pyspark.ml.features handles strings and factors • *one-hot vector* can be created to represent 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).	**Step 1: 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. **Step 2: 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 **Step 3: Assemble a vector, i.e. combine all of the columns containing our features into a single column, using the `VectorAssembler`** because every Spark modeling routine expects the data to be in this form. This is done by storing each of the values from a column as an entry in a vector so 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` is a `Transformer` takes all of the columns you specify and combines them into a new vector column.

Dealing with categorical variables	# Create a StringIndexer
	carr_indexer = StringIndexer(inputCol="carrier",outputCol="carrier_index")
	
	# Create a OneHotEncoder
	carr_encoder = OneHotEncoder(inputCol="carrier_index",outputCol="carrier_fact")

Assemble a vector	# Make a VectorAssembler
	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.**

Make the pipeline
• stages should be a list holding all the stages you want your data to go through in the pipeline
	# Import Pipeline
	from pyspark.ml import Pipeline
	
	# Make the pipeline
	flights_pipe = Pipeline(stages=[dest_indexer, dest_encoder, carr_indexer, carr_encoder, vec_assembler])

Transform the data
• reuse the same process on different data by changing the argument inside the method
	# Fit and transform the data
	piped_data = flights_pipe.fit(model_data).transform(model_data)

Test vs. Train	• split the data into a *test set* and a *train set* • 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. ◦ 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.**
Split the data	# Split the data into training and test sets training, test = piped_data.randomSplit([.6, .4])

Linear Regression	MLlib library is a wrapper over PySpark that supports many machine learning algorithms for classification

	many machine learning algorithms for classification, regression, clustering, dimensionality reduction, and more

Data Pre-Processing	df = spark.read.csv("/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv", header="true", inferSchema="true")
	
	display(df)

Preprocess Categorical Features
1. StringIndexer: This is essentially assigning a numeric value to each category (i.e.; Fair: 0, Ideal: 1, Good: 2, Very Good: 3, Premium: 4)
2. OneHotEncoder: This converts categories into binary vectors. The result is a SparseVector that indicates which index from StringIndexer has the one-hot value of 1.
	from pyspark.ml import Pipeline
	
	from pyspark.ml.feature import StringIndexer, OneHotEncoder
	
	cat_cols= ["cut", "color", "clarity"]
	stages = [] # Stages in Pipeline
	
	for c in cat_cols:
	    stringIndexer = StringIndexer(inputCol=c, outputCol=c + "_index")
	
	    encoder = OneHotEncoder(inputCols=[**stringIndexer.getOutputCol()**], \\
	    outputCols=[c + "_vec"])
	
	    stages += [stringIndexer, encoder] # Stages will be run later on

Assemble Feature Vector
• combine numerical features and the categorical sparse vector features from before into one vector.
• run the stages as a pipeline. This runs the data through all the feature transformations we've defined so far.
• flights_pipe = Pipeline(stages=[dest_indexer, dest_encoder, carr_indexer, carr_encoder, vec_assembler])
	from pyspark.ml.feature import VectorAssembler
	
	# Transform all features into a vector
	
	num_cols = ["carat", "depth", "table", "x", "y", "z"]
	
	assemblerInputs = [c + "_vec" for c in cat_cols] + num_cols
	
	assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
	
	stages += [assembler]
	
	# Create pipeline and use on dataset
	
	pipeline = Pipeline(stages=stages)
	
	df = pipeline.fit(df).transform(df)

Train-test Split	train, test = df.randomSplit([0.90, 0.1], seed=123)
	
	print('Train dataset count:', train.count())
	
	print('Test dataset count:', test.count())

Post-split Data Processing
• it is often recommended to **standardize your features.** PySpark's StandardScaler achieves this by removing the mean (set to zero) and scaling to unit variance.
• First**, fit StandardScaler onto the training data.** Then, use the scaler to **transform the train and test data.** The reason why you should fit on the train data is to avoid data leakage —
	from pyspark.ml.feature import StandardScaler
	
	# Fit scaler to train dataset
	
	scaler = StandardScaler().setInputCol('features') \\
	
	.setOutputCol('scaled_features')
	
	scaler_model = scaler.fit(train)

when applying your model to the real world, you won't know the distribution of your test data yet.

```
# Scale train and test features

train = scaler_model.transform(train)

test = scaler_model.transform(test)
```

Build and Train Model

- We can start with the default parameter values and adjust these during model tuning.
- The default for some parameters to pay attention to are: `maxIter=100`, `regParam=0.0`, `elasticNetParam=0.0`, `loss='squaredError'`. Then call `fit()` to fit the model to the training data.

```
from pyspark.ml.regression import LinearRegression


lr = LinearRegression(featuresCol='scaled_features',
labelCol='price')

lr_model = lr.fit(train)
```

Evaluate Model

- develop predictions from our data using our new model. To get predictions, call `transform()`.
- help identify if you're overfitting or underfitting on your train data.
- If the evaluations for train and test are both similar, you may be underfitting. If your test evaluation is much lower than your train evaluation, you may be overfitting.

```
train_predictions = lr_model.transform(train)

test_predictions = lr_model.transform(test)
```

To evaluate your model, PySpark has RegressionEvaluator at your service. You can choose whichever evaluation metric is most appropriate for your use case:

- RMSE — Root mean squared error (Default)
- MSE — Mean squared error
- R2 — R-squared
- MAE — Mean absolute error
- Var — Explained variance

```
from pyspark.ml.evaluation import RegressionEvaluator


evaluator =
RegressionEvaluator(predictionCol="prediction", \

 labelCol="price", metricName="r2")


print("Train R2:", evaluator.evaluate(train_predictions))

print("Test R2:", evaluator.evaluate(test_predictions))
```

Analyze Feature Weights

- To pull out the linear regression weights and coefficients, run the following code.

```
print("Coefficients: " + str(lr_model.coefficients))

print("Intercept: " + str(lr_model.intercept))
```

Match the weights to the feature names and plot them.

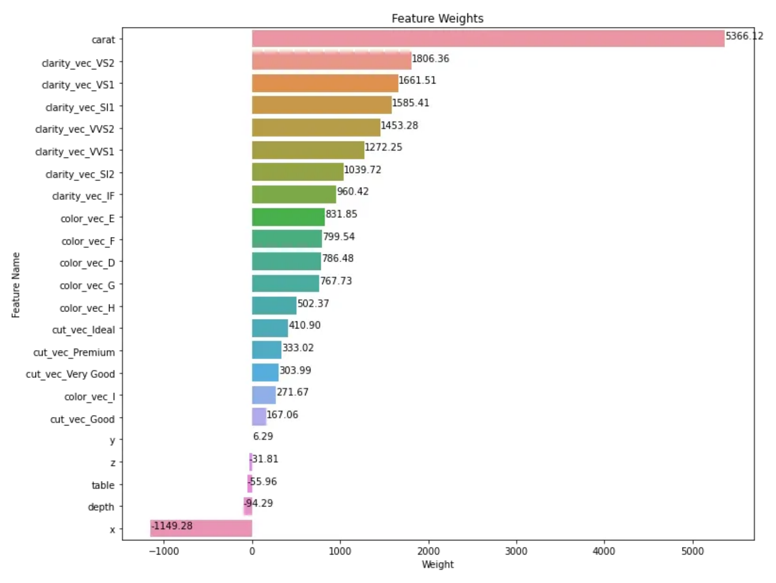
- This view is especially of high interest to key stakeholders who want to understand key drivers of the model.
- This code produces a Pandas dataframe containing the feature names and weights.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

list_extract = []
for i in df.schema['features'].metadata["ml_attr"]["attrs"]:
    list_extract = list_extract + df.schema['features'] \
    .metadata["ml_attr"]["attrs"][i]

varlist = pd.DataFrame(list_extract)
varlist['weight'] = varlist['idx'].apply(lambda x: coef[x])
weights = varlist.sort_values('weight', ascending = False)
```

Plot the weights

```
def show_values(axs, space=.01):

    def _single(ax):
        for p in ax.patches:
            x = p.get_x() + p.get_width() + float(space)
```

```
                _y = p.get_y() + p.get_height() -
               (p.get_height()*0.5)
                value = '{:.2f}'.format(p.get_width())
                ax.text(_x, _y, value, ha="left")

        if isinstance(axs, np.ndarray):

            for idx, ax in np.ndenumerate(axs):
                _single(ax)
                else:  _single(axs)


def plot_feature_weights(df):

        plt.figure(figsize=(10, 8))
        p = sns.barplot(x=df['weight'], y=df['name'])
        show_values(p, space=0)

        plt.title('Feature Weights')
        plt.xlabel('Weight')
        plt.ylabel('Feature Name')


plot_feature_weights(weights)
```



Summary

Steps for implementing linear regression with PySpark:

1. One-hot encode categorical features using StringIndexer and OneHotEncoder
2. Create input feature vector column using VectorAssembler
3. Split data into train and test
4. Scale data using StandardScaler
5. Initialize and fit LinearRegression model on train data
6. Transform model on test data to make predictions
7. Evaluate model with RegressionEvaluator
8. Analyze feature weights to understand and improve model

Logistic regression

- 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 can finetune the model by supplying and testing different values for several hyperparameters to improve model performance. • 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	• 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**. • cv can be used 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
• pyspark.ml.evaluation submodule has classes for evaluating different kinds of models.
• `BinaryClassificationEvaluator` from the pyspark.ml.evaluation module for logit model
• 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.**
	# Import the evaluation submodule
	import pyspark.ml.evaluation as evals
	
	# Create a BinaryClassificationEvaluator
	evaluator = evals.BinaryClassificationEvaluator(metricName="areaUnderROC")

Make a grid
• create a grid of values to search over when looking for the optimal hyperparameters.
• np.arange(0, .1, .01) -> a list of numbers from 0 to .1, incrementing by .01.
• includes only the values `[0, 1]`
	# 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
• `CrossValidator` is an estimator that takes
	# Create the CrossValidator
	cv = tune.CrossValidator(estimator=lr,

the modeler you want to fit, the grid of hyperparameters you created, and the evaluator you want to use to compare your models.	estimatorParamMaps=grid, evaluator=evaluator)
Fit the model(s) • cross validation is a very computationally intensive procedure. • 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.	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 # Call lr.fit() best_lr = lr.fit(training) # Print best_lr to verify that it's an object of the LogisticRegressionModel class. print(best_lr)
Evaluating binary classifiers • AUC, or area under the curve. In this case, the curve is the ROC, or receiver operating curve. • 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))

More on ROC and AUC

ROC (Receiver Operating Characteristic) and AUC (Area Under the Curve) are commonly used evaluation metrics in binary classification problems like logistic regression.

ROC is a graphical representation of the tradeoff between the **true positive rate (TPR) and false positive rate (FPR) for different threshold values of a binary classifier**. TPR is the proportion of positive cases that are correctly identified as positive, while FPR is the proportion of negative cases that are incorrectly classified as positive.

A ROC curve is plotted with TPR on the y-axis and FPR on the x-axis. The curve shows how the TPR and FPR trade off as the threshold for classifying a sample as positive changes. The curve starts at (0, 0) (all samples are classified as negative) and ends at (1, 1) (all samples are classified as positive). A perfect classifier has an ROC curve that passes through the top-left corner, indicating that it has high TPR and low FPR.

AUC, on the other hand, is a scalar value that summarizes the ROC curve. AUC is the area under the ROC curve, which ranges from 0 to 1. AUC of 0.5 indicates that the classifier is no better than random, while an AUC of 1 indicates that the classifier is perfect. AUC can be interpreted as the probability that a randomly chosen positive sample will be ranked higher than a randomly chosen negative sample by the classifier.

In summary, ROC and AUC are important evaluation metrics in binary classification, as they provide a comprehensive view of the classifier's performance across different threshold values. Higher AUC values indicate better performance of the classifier.