Machine Learning Pipelines

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

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

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

Join the DataFrames

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

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

Data types

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"`.

String to integer

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

```
dataframe = dataframe.withColumn("col", dataframe.col.cast("new_type")
```

# 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

This model will use the planes' *age*, which is slightly different from the year it was made!

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

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!

```
# 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

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.

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.

```
# 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.

```
# 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.

```
# Make a VectorAssembler
vec_assembler =
VectorAssembler(inputCols=['month','air_time','carrier_fact','dest_f
act','plane_age'], outputCol='features')
```

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.

> `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_indexer, carr_encoder, vec_assembler])
```

Test vs Train

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

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.

Transform the data

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

Split the data

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

#Use the DataFrame method `.randomSplit()` to split `model_data`into two pieces, `training` with 60% of the data, and `test` with 40%

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

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!

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!

```
# 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.

```
# Import the tuning submodule
import pyspark.ml.tuning as tune

# Create the parameter grid
grid = tune.ParamsGridBuilder()

# 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()

# 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()

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

Fit the model(s)

```
# 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
print(best_lr)

Evaluating binary classifiers

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!

Evaluate the model

Remember the test data that you set aside

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

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