# Machine Learning with Spark

## Introduction

Now that we've performed some data manipulation and aggregation with Spark SQL DataFrames, let's get to the really cool stuff: machine learning!

## Objectives

You will be able to:

- Define estimators and transformers in Spark ML
- Create a Spark ML pipeline that transforms data and runs over a grid of hyperparameters

## A Tale of Two Libraries

If you look at the PySpark documentation, you'll notice that there are two different libraries for machine learning, MLlib (RDD-based) (https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html) and MLlib (DataFrame-based) (https://spark.apache.org/docs/latest/api/python/pyspark.ml.html).

These libraries are extremely similar to one another, the only difference being the kinds of data structures they are designed to operate on. The Spark maintainers have announced that the RDD-based library is now in "maintenance mode" (https://spark.apache.org/docs/latest/ml-guide.html) and that the DataFrame-based library is the primary API.

## Loading the Forest Fire Dataset

Once again we'll use the Forest Fire dataset (https://archive.ics.uci.edu/ml/datasets/Forest+Fires) from UCI, which contains data about the area burned by wildfires in the Northeast region of Portugal in relation to numerous other factors.

Because we are using the DataFrame-based Spark MLlib, we'll load it as a Spark SQL DataFrame.

In [1]:
```
# Create a SparkSession to connect to Spark local cluster
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local').getOrCreate()
```

In [2]:
```
# Read in local CSV file to a Spark SQL DataFrame
fire_df = spark.read.csv('forestfires.csv', header='true', inferSchema='true')
fire_df.show(5)
```

```
+---+---+-----+---+----+----+-----+---+----+---+----+----+----+
|  X|  Y|month|day|FFMC| DMC|   DC|ISI|temp| RH|wind|rain|area|
+---+---+-----+---+----+----+-----+---+----+---+----+----+----+
|  7|  5|  mar|fri|86.2|26.2| 94.3|5.1| 8.2| 51| 6.7| 0.0| 0.0|
|  7|  4|  oct|tue|90.6|35.4|669.1|6.7|18.0| 33| 0.9| 0.0| 0.0|
|  7|  4|  oct|sat|90.6|43.7|686.9|6.7|14.6| 33| 1.3| 0.0| 0.0|
|  8|  6|  mar|fri|91.7|33.3| 77.5|9.0| 8.3| 97| 4.0| 0.2| 0.0|
|  8|  6|  mar|sun|89.3|51.3|102.2|9.6|11.4| 99| 1.8| 0.0| 0.0|
+---+---+-----+---+----+----+-----+---+----+---+----+----+----+
only showing top 5 rows
```

## Machine Learning with Spark

PySpark states that they've used scikit-learn as an inspiration for their implementation of a machine learning library. As a result, many of the methods and functionalities look similar, but there are some crucial distinctions. There are three main concepts found within the ML library:

`Transformer` : An algorithm that transforms one PySpark DataFrame into another DataFrame. Just like in scikit-learn, a Transformer is a class that implements the `transform()` method. Transformers include both feature transformers and `Model` s.

`Estimator` : An algorithm that can be fit onto a PySpark DataFrame that can then be used as a Transformer. Just like in scikit-learn, an Estimator is a class that implements the `fit()` method.

`Pipeline` : A pipeline very similar to a scikit-learn pipeline that chains together different actions. Typically Transformers and Estimators are components of a Pipeline.

The reasoning behind this separation of the fitting and transforming step is because Spark is lazily evaluated, so the 'fitting' of a model does not actually take place until the Transformation action is called. Let's examine what this actually looks like by performing a regression on the Forest Fire dataset. To start off with, we'll import the necessary machine learning classes for our tasks.

In [3]:
```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
```

## ML Preprocessing with Spark

### Encoding Categorical Data

In [4]:
```
fire_df.printSchema()
```

```
root
 |-- X: integer (nullable = true)
 |-- Y: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- day: string (nullable = true)
 |-- FFMC: double (nullable = true)
 |-- DMC: double (nullable = true)
 |-- DC: double (nullable = true)
 |-- ISI: double (nullable = true)
 |-- temp: double (nullable = true)
 |-- RH: integer (nullable = true)
 |-- wind: double (nullable = true)
 |-- rain: double (nullable = true)
 |-- area: double (nullable = true)
```

Looking at our data, one can see that all the categories are numeric except for `month` and `day`. We saw some correlation between the month and area burned in a fire during our previous EDA process, so we will include that in our model. The day of the week, however, is highly unlikely to have any effect on fire, so we will drop it from the DataFrame.

In [5]:
```
df_without_day = fire_df.drop('day')
df_without_day.show(5)
```

```
+---+---+-----+----+----+-----+---+----+---+----+----+----+
|  X|  Y|month|FFMC| DMC|   DC|ISI|temp| RH|wind|rain|area|
+---+---+-----+----+----+-----+---+----+---+----+----+----+
|  7|  5|  mar|86.2|26.2| 94.3|5.1| 8.2| 51| 6.7| 0.0| 0.0|
|  7|  4|  oct|90.6|35.4|669.1|6.7|18.0| 33| 0.9| 0.0| 0.0|
|  7|  4|  oct|90.6|43.7|686.9|6.7|14.6| 33| 1.3| 0.0| 0.0|
|  8|  6|  mar|91.7|33.3| 77.5|9.0| 8.3| 97| 4.0| 0.2| 0.0|
|  8|  6|  mar|89.3|51.3|102.2|9.6|11.4| 99| 1.8| 0.0| 0.0|
+---+---+-----+----+----+-----+---+----+---+----+----+----+
only showing top 5 rows
```

In [6]:
```
df_without_day.printSchema()
```

```
root
 |-- X: integer (nullable = true)
 |-- Y: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- FFMC: double (nullable = true)
 |-- DMC: double (nullable = true)
 |-- DC: double (nullable = true)
 |-- ISI: double (nullable = true)
 |-- temp: double (nullable = true)
 |-- RH: integer (nullable = true)
 |-- wind: double (nullable = true)
 |-- rain: double (nullable = true)
 |-- area: double (nullable = true)
```

In order for us to run our model, we need to turn the months variable into a dummy variable. In MLlib this is a 2-step process that first requires turning the categorical variable into a numerical index ( `StringIndexer` ). Only after the variable is an integer can PySpark create dummy variable columns related to each category ( `OneHotEncoder` ).

Your key parameters when using these transformers are:

- `inputCol` (the column you want to change), and
- `outputCol` (where you will store the changed column)

Here it is in action:

In [7]:
```
# Create a StringIndexer (Estimator) that takes month as the input column
# and outputs month_num
si_untrained = StringIndexer(inputCol='month', outputCol='month_num', handleInvalid='keep')

# Create a trained Model (Transformer) by fitting the StringIndexer on the data
si_trained = si_untrained.fit(df_without_day)

# Create a new DataFrame using the trained Model (Transformer) and data
df_month_num = si_trained.transform(df_without_day)

df_month_num.show(5)
```

```
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+
|  X|  Y|month|FFMC| DMC|   DC|ISI|temp| RH|wind|rain|area|month_num|
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+
|  7|  5|  mar|86.2|26.2| 94.3|5.1| 8.2| 51| 6.7| 0.0| 0.0|      2.0|
|  7|  4|  oct|90.6|35.4|669.1|6.7|18.0| 33| 0.9| 0.0| 0.0|      6.0|
|  7|  4|  oct|90.6|43.7|686.9|6.7|14.6| 33| 1.3| 0.0| 0.0|      6.0|
|  8|  6|  mar|91.7|33.3| 77.5|9.0| 8.3| 97| 4.0| 0.2| 0.0|      2.0|
|  8|  6|  mar|89.3|51.3|102.2|9.6|11.4| 99| 1.8| 0.0| 0.0|      2.0|
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+
only showing top 5 rows
```

In [8]:
```
# Double-checking that we have 12 different numbers as expected
df_month_num.select('month_num').distinct().orderBy('month_num').show()
```

```
+---------+
|month_num|
+---------+
|      0.0|
|      1.0|
|      2.0|
|      3.0|
|      4.0|
|      5.0|
|      6.0|
|      7.0|
|      8.0|
|      9.0|
|     10.0|
|     11.0|
+---------+
```

As you can see, we have created a new column called `month_num` that represents the month by a number.

Note the small, but critical distinction between `sklearn` 's implementation of a transformer and PySpark's implementation. `sklearn` is more object oriented and Spark is more functional oriented.

Specifically, calling `fit()` on a Spark transformer does not cause that transformer to become trained, but rather returns a trained transformer.

In [9]:
```
# this is an Estimator, which has a fit() method
type(si_untrained)
```

Out[9]:
```
pyspark.ml.feature.StringIndexer
```

In [10]:
```
# this is a Model (Transformer), the result of the fit() method
type(si_trained)
```

Out[10]:
```
pyspark.ml.feature.StringIndexerModel
```

Only the trained Model has fitted attributes. In this case, the `StringIndexerModel` has a `labels` attribute, which explain the relationship between `month_num` and `month` :

In [11]:
```
si_trained.labels
```

Out[11]:
```
['aug',
 'sep',
 'mar',
 'jul',
 'feb',
 'jun',
 'oct',
 'apr',
 'dec',
 'jan',
 'may',
 'nov']
```

In [12]:
```
try:
    si_untrained.labels
except Exception as e:
    print(type(e))
    print(e)
```

```
<class 'AttributeError'>
'StringIndexer' object has no attribute 'labels'
```

Now that we have `month_num`, we can use Spark's version of `OneHotEncoder()`. This time rather than creating an intermediate variable for the model, we'll chain together the `fit()` and `tranform()` method calls.

In [13]:
```
# Instantiating, fitting, and transforming month_num with OneHotEncoder
ohe = OneHotEncoder(inputCols=['month_num'], outputCols=['month_vec'], dropLast=True)
df_month_vec = ohe.fit(df_month_num).transform(df_month_num)
df_month_vec.show(5)
```

```
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+
|  X|  Y|month|FFMC| DMC|   DC|ISI|temp| RH|wind|rain|area|month_num|    month_vec|
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+
|  7|  5|  mar|86.2|26.2| 94.3|5.1| 8.2| 51| 6.7| 0.0| 0.0|      2.0|(12,[2],[1.0])|
|  7|  4|  oct|90.6|35.4|669.1|6.7|18.0| 33| 0.9| 0.0| 0.0|      6.0|(12,[6],[1.0])|
|  7|  4|  oct|90.6|43.7|686.9|6.7|14.6| 33| 1.3| 0.0| 0.0|      6.0|(12,[6],[1.0])|
|  8|  6|  mar|91.7|33.3| 77.5|9.0| 8.3| 97| 4.0| 0.2| 0.0|      2.0|(12,[2],[1.0])|
|  8|  6|  mar|89.3|51.3|102.2|9.6|11.4| 99| 1.8| 0.0| 0.0|      2.0|(12,[2],[1.0])|
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+
only showing top 5 rows
```

Great, we now have a one-hot encoded sparse vector in the `month_vec` column! Because Spark is optimized for big data, sparse vectors are used rather than entirely new columns for dummy variables because it is more space efficient. You can see in this first row of the DataFrame:

```
(11,[2],[1.0])
```

This indicates that:

- We have a sparse vector of size `11` (because there are 12 unique month values, and we set `dropLast=True` when we instantiated the `OneHotEncoder`)
- This particular data point is the `2`-th index of our month labels (`'mar'`, based off the labels in the `StringIndexerModel`)

**One Sparse Vector**

The final requirement for all machine learning models in PySpark is to put all of the features of your model into one sparse vector. This is once again for efficiency's sake. Here, we are doing that with the `VectorAssembler()` estimator.

In [14]:
```
# Set name of target
target = "area"

# Get list of features, excluding area (the target) as well
# as month and month_num (since we are using month_vec)
features = df_month_vec.drop(target, "month", "month_num").columns
features
```

Out[14]:
```
['X',
 'Y',
 'FFMC',
 'DMC',
 'DC',
 'ISI',
 'temp',
 'RH',
 'wind',
 'rain',
 'month_vec']
```

In [15]:
```
# A VectorAssembler is a "pure" Transformer, not a Model that
# requires fitting. Use it to transform features.
va = VectorAssembler(inputCols=features, outputCol="features")
df_vec = va.transform(df_month_vec)

df_vec.show(5)
```

```
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+--------------------+
|  X|  Y|month|FFMC| DMC|   DC|ISI|temp| RH|wind|rain|area|month_num|     month_vec|            features|
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+--------------------+
|  7|  5|  mar|86.2|26.2| 94.3|5.1| 8.2| 51| 6.7| 0.0| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...|
|  7|  4|  oct|90.6|35.4|669.1|6.7|18.0| 33| 0.9| 0.0| 0.0|      6.0|(12,[6],[1.0])|(22,[0,1,2,3,4,5,...|
|  7|  4|  oct|90.6|43.7|686.9|6.7|14.6| 33| 1.3| 0.0| 0.0|      6.0|(12,[6],[1.0])|(22,[0,1,2,3,4,5,...|
|  8|  6|  mar|91.7|33.3| 77.5|9.0| 8.3| 97| 4.0| 0.2| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...|
|  8|  6|  mar|89.3|51.3|102.2|9.6|11.4| 99| 1.8| 0.0| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...|
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+--------------------+
only showing top 5 rows
```

This is definitely a bit weird to look at compared to data prepared in pandas for a scikit-learn model. What you need to know is that all of the feature data that the model will actually use is encoded in the `features` column.

### Fitting an ML Model with Spark

Great! We now have our data in a format that seems acceptable for the last step. It's time for us to actually fit our model to data! Let's fit a Random Forest Regression model to our data. Although there are still a bunch of other features in the DataFrame, it doesn't matter for the machine learning model API. All that needs to be specified are the names of the features column and the label (i.e. target) column.

In [16]:
```
# Instantiating and fitting the model
rf_model = RandomForestRegressor(featuresCol='features',
                                 labelCol=target,
                                 predictionCol='prediction'
                                ).fit(df_vec)
```

In [17]:
```
# Inspecting fitted model attributes
rf_model.featureImportances
```

Out[17]:
```
SparseVector(22, {0: 0.1139, 1: 0.0508, 2: 0.2052, 3: 0.1079, 4: 0.1395, 5: 0.0359, 6: 0.0752, 7: 0.1318, 8: 0.1093, 9: 0.002,
10: 0.0018, 11: 0.0142, 12: 0.0, 13: 0.0107, 14: 0.0002, 16: 0.001, 17: 0.0001, 18: 0.0001, 20: 0.0003})
```

In [18]:
```
# Transform the DataFrame using the fitted model
df_final = rf_model.transform(df_vec)
df_final.show(5)
```

```
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+--------------------+------------------+
|  X|  Y|month|FFMC| DMC|   DC|ISI|temp| RH|wind|rain|area|month_num|     month_vec|            features|        prediction|
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+--------------------+------------------+
|  7|  5|  mar|86.2|26.2| 94.3|5.1| 8.2| 51| 6.7| 0.0| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...| 5.898088759947248|
|  7|  4|  oct|90.6|35.4|669.1|6.7|18.0| 33| 0.9| 0.0| 0.0|      6.0|(12,[6],[1.0])|(22,[0,1,2,3,4,5,...| 5.005294139647212|
|  7|  4|  oct|90.6|43.7|686.9|6.7|14.6| 33| 1.3| 0.0| 0.0|      6.0|(12,[6],[1.0])|(22,[0,1,2,3,4,5,...|5.9012003896472125|
|  8|  6|  mar|91.7|33.3| 77.5|9.0| 8.3| 97| 4.0| 0.2| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...|7.7049237280786285|
|  8|  6|  mar|89.3|51.3|102.2|9.6|11.4| 99| 1.8| 0.0| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...|4.7609687837235715|
+---+---+-----+----+----+-----+---+----+---+----+----+----+---------+-------------+--------------------+------------------+
only showing top 5 rows
```

Again, this looks very different from our usual scikit-learn approach. Instead of generating a prediction as a separate vector, it was added as another column in the overall DataFrame.

### Model Evaluation with Spark

If we want to look at just the target vs. the prediction, we can select just those columns:

In [19]:
```
df_final.select(target, 'prediction').show(10)
```

```
+----+------------------+
|area|        prediction|
+----+------------------+
| 0.0| 5.898088759947248|
| 0.0| 5.005294139647212|
| 0.0|5.9012003896472125|
| 0.0|7.7049237280786285|
| 0.0|4.7609687837235715|
| 0.0| 7.910257549593487|
| 0.0| 7.146172361255255|
| 0.0|11.923624873519476|
| 0.0| 7.499414846964632|
| 0.0| 6.922794908436825|
+----+------------------+
only showing top 10 rows
```

Ok, this snapshot is showing the model consistently predicting too high of an area.

What if we look at the subset where the area was greater than zero?

In [20]:
```
df_final[df_final[target] > 0].select(target, 'prediction').show(10)
```

```
+----+------------------+
|area|        prediction|
+----+------------------+
|0.36|3.7585944317835063|
|0.43|15.022976888053885|
|0.47|11.840964939026374|
|0.55| 7.938726300414726|
|0.61| 3.992752285380399|
|0.71|13.697652912644354|
|0.77| 4.596214663344088|
| 0.9|11.419223850936309|
|0.95|4.4435624079080895|
|0.96| 18.59024169502403|
+----+------------------+
only showing top 10 rows
```

Still not so great. Let's use some metrics to evaluate formally.

PySpark has a class `RegressionEvaluator` that is designed for this purpose:

In [21]:
```
from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator(predictionCol='prediction', labelCol=target)
```

In [22]:
```
# Evaluating r^2
evaluator.evaluate(df_final, {evaluator.metricName: 'r2'})
```

Out[22]:
```
0.582721694127662
```

In [23]:
```
# Evaluating mean absolute error
evaluator.evaluate(df_final, {evaluator.metricName: 'mae'})
```

Out[23]:
```
13.464552164786383
```

In other words, our model is able to explain about 80% of the variance in the data, and on average its area prediction is off by about 13 hectares.

## Putting It All in a Pipeline

We just performed a whole lot of transformations to our data. Let's take a look at all the estimators we used to create this model:

- `StringIndexer()`
- `OneHotEncoder()`
- `VectorAssembler()`
- `RandomForestRegressor()`

Once we've fit our model in the Pipeline, we're then going to want to evaluate it to determine how well it performs. We can do this with:

- `RegressionEvaluator()`

We can streamline all of these transformations to make it much more efficient by chaining them together in a pipeline. The Pipeline object expects a list of the estimators to be set to the parameter `stages` .

In [24]:
```
# Recall that we have already established features and target variables
print(features)
print(target)
```

```
['X', 'Y', 'FFMC', 'DMC', 'DC', 'ISI', 'temp', 'RH', 'wind', 'rain', 'month_vec']
area
```

In [25]:
```
# Importing relevant class
from pyspark.ml import Pipeline

# Creating a variable for this one because we will need it later
# Using the features and target columns, create a column of predictions
random_forest = RandomForestRegressor(featuresCol='features', labelCol=target, predictionCol='prediction')

# Instantiating the pipeline with all the estimators
pipeline = Pipeline(stages=[
    # Convert string data in 'month' column to numeric data in 'month_num' column
    StringIndexer(inputCol='month', outputCol='month_num', handleInvalid='keep'),
    # Convert 'month_num' numbers to sparse vector in 'month_vec' column
    OneHotEncoder(inputCols=['month_num'], outputCols=['month_vec'], dropLast=True),
    # Convert all of the relevant features to sparse vector in 'features' column
    VectorAssembler(inputCols=features, outputCol='features'),
    random_forest
])
```

In [26]:
```
# Now we can perform all of the above preprocessing steps!
df_final_pipeline = pipeline.fit(fire_df).transform(fire_df)
df_final_pipeline.show(10)
```

```
+---+---+-----+---+----+-----+-----+----+----+---+----+----+----+---------+-------------+--------------------+----------
--+
|  X|  Y|month|day|FFMC|  DMC|   DC| ISI|temp|  RH|wind|rain|area|month_num|    month_vec|            features|        predicti
on|
+---+---+-----+---+----+-----+-----+----+----+---+----+----+----+---------+-------------+--------------------+----------
--+
|  7|  5|  mar|fri|86.2| 26.2| 94.3| 5.1| 8.2|  51| 6.7| 0.0| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...| 5.8980887599472
48|
|  7|  4|  oct|tue|90.6| 35.4|669.1| 6.7|18.0|  33| 0.9| 0.0| 0.0|      6.0|(12,[6],[1.0])|(22,[0,1,2,3,4,5,...| 5.0052941396472
12|
|  7|  4|  oct|sat|90.6| 43.7|686.9| 6.7|14.6|  33| 1.3| 0.0| 0.0|      6.0|(12,[6],[1.0])|(22,[0,1,2,3,4,5,...|5.90120038964721
25|
|  8|  6|  mar|fri|91.7| 33.3| 77.5| 9.0| 8.3|  97| 4.0| 0.2| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...|7.70492372807862
85|
|  8|  6|  mar|sun|89.3| 51.3|102.2| 9.6|11.4|  99| 1.8| 0.0| 0.0|      2.0|(12,[2],[1.0])|(22,[0,1,2,3,4,5,...|4.76096878372357
15|
|  8|  6|  aug|sun|92.3| 85.3|488.0|14.7|22.2|  29| 5.4| 0.0| 0.0|      0.0|(12,[0],[1.0])|(22,[0,1,2,3,4,5,...| 7.9102575495934
87|
|  8|  6|  aug|mon|92.3| 88.9|495.6| 8.5|24.1|  27| 3.1| 0.0| 0.0|      0.0|(12,[0],[1.0])|(22,[0,1,2,3,4,5,...| 7.1461723612552
55|
|  8|  6|  aug|mon|91.5|145.4|608.2|10.7| 8.0|  86| 2.2| 0.0| 0.0|      0.0|(12,[0],[1.0])|(22,[0,1,2,3,4,5,...|11.9236248735194
76|
|  8|  6|  sep|tue|91.0|129.5|692.6| 7.0|13.1|  63| 5.4| 0.0| 0.0|      1.0|(12,[1],[1.0])|(22,[0,1,2,3,4,5,...| 7.4994148469646
32|
|  7|  5|  sep|sat|92.5| 88.0|698.6| 7.1|22.8|  40| 4.0| 0.0| 0.0|      1.0|(12,[1],[1.0])|(22,[0,1,2,3,4,5,...| 6.9227949084368
25|
+---+---+-----+---+----+-----+-----+----+----+---+----+----+----+---------+-------------+--------------------+----------
--+
only showing top 10 rows
```

In [27]:
```
# Evaluating r^2
evaluator.evaluate(df_final_pipeline, {evaluator.metricName: 'r2'})
```

Out[27]:
```
0.582721694127662
```

In [28]:
```
# Evaluating mean absolute error
evaluator.evaluate(df_final_pipeline, {evaluator.metricName: 'mae'})
```

Out[28]:
```
13.464552164786383
```

And we're getting the same metrics as before!

### Model Tuning

You might have missed a critical step in the random forest regression above; we did not cross validate or perform a train/test split! This means that we don't have a good idea of how our model would perform on unseen data, or have a good setup for model tuning.

Fortunately pipelines in MLlib make this process fairly straightforward. Some components to be aware of are:

- `ParamGridBuilder` (documentation here (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.ParamGridBuilder.html)): This is similar to `GridSearchCV` in scikit-learn.

- It allows you to specify and test out various combinations of parameters.
- Also like `GridSearchCV`, it is easy to write code that will take a **very** long time to execute, so be cautious with the number of models you try to run at once!
- Unlike `GridSearchCV`, it isn't necessarily used with a cross-validation strategy. You can choose between using cross-validation or a train-validation split depending on the context.

- `CrossValidator` (documentation here (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.CrossValidator.html)): This is similar to `cross_val_score` in scikit-learn, although it's a class rather than a function.
  - It allows you to perform k-fold cross-validation, to get model metrics across several different splits of data.
- `TrainValidationSplit` (documentation here (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.TrainValidationSplit.html)): There is no direct parallel to this in scikit-learn, although it contains elements of `train_test_split`.
  - It allows you to perform a train-validation split, and to evaluate a model based on a validation set that the model was not trained on.

#### Comparing `CrossValidator` and `TrainValidationSplit`

`CrossValidator` is more computationally expensive but will likely give you a more realistic view of how your model will perform on unseen data. `TrainValidationSplit` is faster but potentially less realistic, because it is relying on a single split of the data. For this lesson we'll only use `CrossValidator`, and we expect the grid search to take up to a couple of minutes. If your computer is running very slowly, you might consider changing it to a `TrainValidationSplit`, although the numbers will come out a bit different.

#### Building Our Param Grid

Let's go ahead and import those classes, then build our param grid!

Note that we are specifying hyperparameters for the `RandomForestRegressor`, documentation for which can be found here (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.RandomForestRegressor.html).

In [29]:
```
# Model tuning imports
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
```

In [30]:
```
# Creating parameter grid

# (This style of multi-line code with backslashes imitates the
# style of Scala, the original Spark languge)

params = ParamGridBuilder()\
          .addGrid(random_forest.maxDepth, [5, 10, 15])\
          .addGrid(random_forest.numTrees, [20, 50, 100])\
          .build()
```

Let's take a look at the params variable we just built.

In [31]:
```
print('total combinations of parameters: ', len(params))

params[0]
```

```
total combinations of parameters:  9
```

Out[31]:
```
{Param(parent='RandomForestRegressor_bc564c30865e', name='maxDepth', doc='Maximum depth of the tree. (>= 0) E.g., depth 0 means
1 leaf node; depth 1 means 1 internal node + 2 leaf nodes. Must be in range [0, 30].'): 5,
 Param(parent='RandomForestRegressor_bc564c30865e', name='numTrees', doc='Number of trees to train (>= 1).'): 20}
```

#### Cross-Validation

Now it's time to combine all the steps we've created to work in a single line of code with the `CrossValidator()` estimator.

In [32]:
```
# Instantiating the evaluator by which we will measure our model's performance
reg_evaluator = RegressionEvaluator(
    # 'prediction' is where our model is adding the prediction to the df
    predictionCol='prediction',
    # The evaluator is comparing the prediction to this target
    labelCol=target,
    # The evaluator is evaluation the prediction based on this metric
    metricName = 'mae'
)

# Instantiating cross validator estimator
cv = CrossValidator(
    # Will call fit and transform using this
    estimator=pipeline,
    # Will iterate over these hyperparameter options
    estimatorParamMaps=params,
    # Will evaluate based on this evaluator
    evaluator=reg_evaluator,
    # Will use 4 threads when running parallel algorithms
    parallelism=4
)
```

In [33]:
```
# Fitting cross validator
cross_validated_model = cv.fit(fire_df)
```

**Model Evaluation with Cross-Validation**

Now, let's see how well the model performed! Let's take a look at the average performance for each one of our 9 models.

In [34]:
```
cross_validated_model.avgMetrics
```

Out[34]:
```
[22.790540868526982,
 20.69425857338582,
 21.054343931678073,
 23.376957675022418,
 21.56271738942795,
 22.00020398191387,
 23.556214611760687,
 21.67247430231654,
 22.088303223954767]
```

It looks like the optimal performance is an MAE around 20. Note that this is worse than our original model, but that's because our original model had substantial data leakage. We didn't do a train-test split!

Now, let's take a look at the optimal parameters of our best performing model. The `cross_validated_model` variable is now saved as the best performing model from the grid search just performed. Let's look to see how well the predictions performed.

In [35]:
```
predictions = cross_validated_model.transform(fire_df)
predictions.select('area', 'prediction').show(150)
```

```
+----+------------------+
|area|        prediction|
+----+------------------+
| 0.0|  5.46338663408609|
| 0.0| 5.060755772318182|
| 0.0| 5.737809143630919|
| 0.0| 4.748977483241895|
| 0.0| 5.332632130720389|
| 0.0|11.612369608132388|
| 0.0|19.185312316761088|
| 0.0| 6.216435834110016|
| 0.0| 7.379463598849831|
| 0.0|28.065495368422503|
| 0.0|27.812209937124873|
| 0.0| 5.794094576555795|
| 0.0| 6.037209276188649|
| 0.0| 9.290245863672006|
| 0.0| 47.24540019586294|
| 0.0| 6.795623362962512|
```

As you can see, this dataset has a large number of areas of "0.0" burned. Perhaps it would be better to investigate this problem as a classification task.

Now let's go ahead and take a look at the feature importances of our random forest model. In order to do this, we need to unroll our pipeline to access the random forest model. Let's start by first checking out the `.bestModel` attribute of our `cross_validated_model` .

In [36]:
```
type(cross_validated_model.bestModel)
```

Out[36]:
```
pyspark.ml.pipeline.PipelineModel
```

MLlib is treating the entire pipeline as the best performing model, so we need to go deeper into the pipeline to access the random forest model within it. Previously, we put the random forest model as the final "stage" in the stages variable list. Let's look at the `.stages` attribute of the `.bestModel`.

In [37]:
```
cross_validated_model.bestModel.stages
```

Out[37]:
```
[StringIndexerModel: uid=StringIndexer_c22019728c37, handleInvalid=keep,
 OneHotEncoderModel: uid=OneHotEncoder_21bc973c69a4, dropLast=true, handleInvalid=error, numInputCols=1, numOutputCols=1,
 VectorAssembler_2dc86ea63350,
 RandomForestRegressionModel: uid=RandomForestRegressor_bc564c30865e, numTrees=50, numFeatures=22]
```

Perfect! There's the RandomForestRegressionModel, represented by the last item in the stages list. Now, we should be able to access all the attributes of the random forest regressor.

In [38]:
```
optimal_rf_model = cross_validated_model.bestModel.stages[3]
```

In [39]:
```
optimal_rf_model.featureImportances
```

Out[39]:
```
SparseVector(22, {0: 0.0812, 1: 0.0675, 2: 0.1677, 3: 0.1252, 4: 0.1171, 5: 0.0689, 6: 0.1147, 7: 0.1173, 8: 0.0966, 9: 0.0, 1
0: 0.0092, 11: 0.011, 12: 0.0, 13: 0.0169, 14: 0.0004, 15: 0.0056, 16: 0.0001, 17: 0.0003, 18: 0.0002, 20: 0.0})
```

In [40]:
```
optimal_rf_model.getNumTrees
```

Out[40]:
```
50
```

## Summary

In this lesson, you learned about PySpark's machine learning models and pipelines. With the use of a pipeline, you can train a huge number of models simultaneously, saving you a substantial amount of time and effort. Up next, you will have a chance to build a PySpark machine learning pipeline of your own with a classification problem!