Spark Program

Chapter 8: Classification Models

Chapter Objectives

In this chapter, we will:

- Introduce Pipelines for better transformations
- Understand the use cases for Classification models
- Discuss and compare various algorithms
  - Naive Bayes
  - Decision Tree
  - Logistic Regression
  - Neural Network

Chapter Concepts

Problem with Existing Transformations

- Most transformation occurs on one column at a time in a DataFrame
- Because the data is partitioned among many nodes it means that wide operations have to shuffle around a lot of data to process that one column
- Then you do a transformation on another column and the whole process starts all over again
- If you have a lot of categorical columns to OneHotEncode, each one is done sequentially after the other, then you do the same thing again to scale the numeric features, and again to bundle them all up into a Vector
- Pipelines allow a more elegant and efficient mechanism to chain together many transformation steps so that they can be optimized and run together in parallel instead of sequentially
- Also, they allow you to save the entire chain of transformations so they can be reused when you need to encode new data to make predictions on

Pipeline Components

- Pipelines are composed of Transformers and Estimators
- Transformers
  - Implement a `transform()` method
  - Convert one DataFrame into another, generally by adding one or more columns
- Estimators
  - Implement as `fit()` methods
  - Take a DataFrame object and return a Model object

Steps to Pipeline

- Create a Python `list` of the steps you want to do in the order you want them done
- Create a pipeline object and pass it the list of stages
- Call `fit` on the pipeline object to train the model
- Call `transform` on the model object to get the predictions
- Watch the performance improvement
- As always, a custom helper function makes things easier to use instead of building it from scratch each time

Pipeline Example

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Prepare test documents, which are unlabeled (id, text) tuples.

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
```

- As usual, a good helper function makes all this easier so we will use the one we wrote in `pyspark_helpers` called `MakeMLPipeline`

Chapter Concepts

Classification

- There are many business use cases for wanting to predict whether a record will fall into one category or another
  - Is a credit card swipe fraudulent or not?
  - Does a patient have a disease?
  - Will an applicant be a profitable customer?
- Classification can make such predictions by using historical data to train the model to look for patterns
- To see how good a job the model does, you test it with another set of data that was not used to train the model
  - By comparing the known values to the predicted ones, you can judge how well the model performs
- If the model guesses better than a coin flip or random guess, it may not be perfect, but it's better than nothing and could be used to create actionable business decisions with a best guess
- There are many different algorithms that can do classification, so it's best to run the same data through many different models to see which works best for your data

Steps to Classification

- There are some consistent steps you need to do regardless of which algorithm you use, although some models have different requirements
- Usually, categorical data needs to be re-encoded as a vector that is OneHotEncoded
- All models should take the dataset and split it into a training and testing set
- First, you fit the model with the training set
  - Can take some time for large datasets
- Once the model is trained, you can see how good it is at making predictions by using a predict function and comparing those values to known values for the variable you are trying to predict
- After you've picked the model that does the best job, you can use it to predict values either individually or in a batch for new data as it comes in

Notes on Classification

- Often you are trying to predict an either/or value
  - Is a card swipe fraudulent or legitimate?
  - Does a patient have cancer or not?
- Not limited to just two choices, you could predict whether a record falls into a category with many different values
  - It just gets trickier sometimes to interpret the results
- The math and techniques behind the scenes can get complicated, but you don't really need to know any of it to use the algorithms
- Just knowing how to identify that you need a classification model and how to prep the data and interpret the results is often good enough to get started
- As you get more sophisticated, you can learn the math behind the scenes to tweak the results and try to get better results

Preparing the Data

- Let's look at some data first, and use this with several different algorithms
- The dataset is whether borrowers defaulted on a loan or not

```
filename = 'bank.csv'
df = spark.read.csv(f'/class/datasets/finance/{filename}', header =
True, inferSchema = True)
display(df)
```

- Let's keep the following numerical and categorical features and try to predict whether they default yes/no

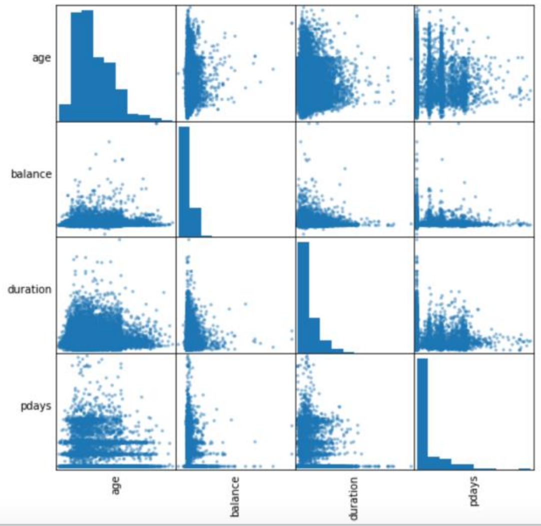
```
numeric_features = ['age','balance', 'duration', 'pdays']
categorical_features = ['job', 'marital', 'education', 'housing',
'loan', 'contact', 'campaign', 'poutcome', 'deposit']
target_label = 'default'

df = dfRawFile.select(numeric_features + categorical_features +
[target_label])
```

Explore Numerical Features

- Scatter plots are helpful to explore the numerical data

```
pyh.describe_numeric_features(df, numeric_features)
pyh.scatter_matrix(df, numeric_features)
```

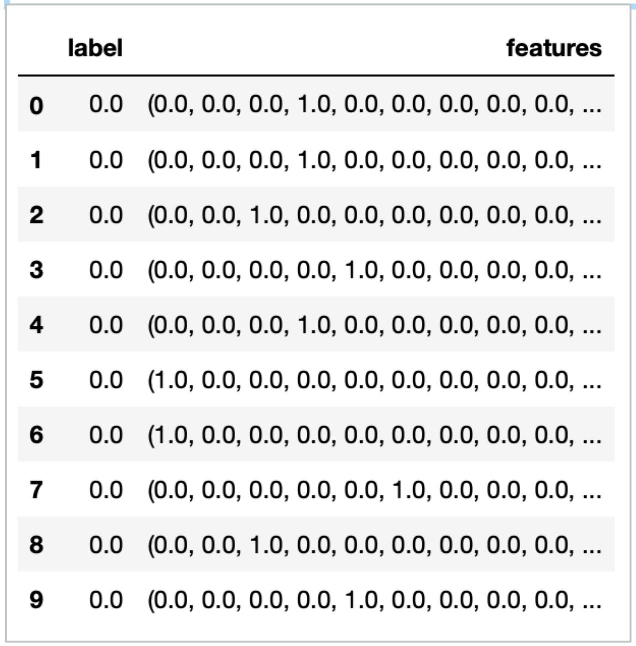
```
  summary                 age             balance            duration  \
0   count               11162               11162               11162
1    mean  41.231947679627304  1528.5385235620856  371.99381831213043
2  stddev  11.913369192215518   3225.413325946149  347.12838571630687
3     min                  18               -6847                   2
4     max                  95               81204                3881

               pdays
0              11162
1  51.33040673714388
2 108.75828197197717
3                 -1
4                854
```



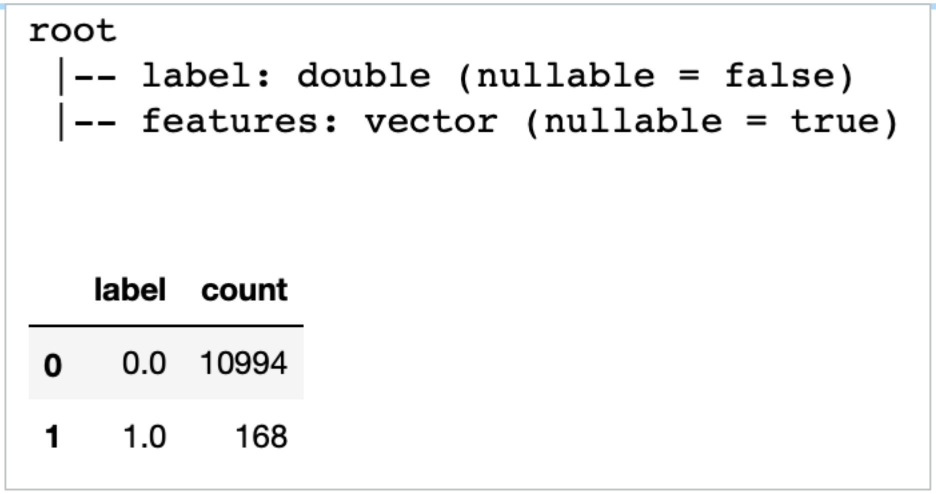
Change Categorical Column

- Use the helper function we saw earlier in the Regression chapter

```
pipeline = MakeMLPipeline(df, categorical_features,
numeric_features, target_label)
pipelineModel = pipeline.fit(df)
dfML = pipelineModel.transform(df)
display(dfML)
display(dfML.groupBy('label').count())
```



	label	features
0	0.0	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
1	0.0	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
2	0.0	(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
3	0.0	(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...
4	0.0	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
5	0.0	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
6	0.0	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
7	0.0	(0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, ...
8	0.0	(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
9	0.0	(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...



```
root
 |-- label: double (nullable = false)
 |-- features: vector (nullable = true)
```

	label	count
0	0.0	10994
1	1.0	168

Saving Processed Data

- If you want, you can save the newly structured DataFrame for future use
- `libsvm` is a good format choice for vectorized data

```
# write
dfML.write.format('libsvm').save('testsave')

# read
dfML = spark.read.format('libsvm').load('testsave')
x.printSchema()
display(x)
```

Splitting the Data

- Split the data into training and testing sets using the `randomSplit` function

```
train, test = dfML.randomSplit([.7,.3], seed = 1000)
print (f'Training set row count {train.count()}')
print (f'Testing set row count {test.count()}')
```

Chapter Concepts

Decision Trees

- Data is split up based on some factor among the independent variables, then evaluated for how good a job it did
- Recursively keeps applying this algorithm over and over until it comes up with a good set of rules
- Pros
  - It's easy
  - Performs well with categorical data and continuous data
  - Transparency lets you see how it made its choices
- Cons
  - Calculations take a lot longer as you add more and more columns
  - Becomes difficult to understand the decision tree as it gets larger

Apply Decision Tree

- Load the module, create a model and train it

```
from pyspark.ml.classification import
DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol = 'features', \
                            labelCol = 'label', maxDepth = 3)
dtModel = dt.fit(train)
```

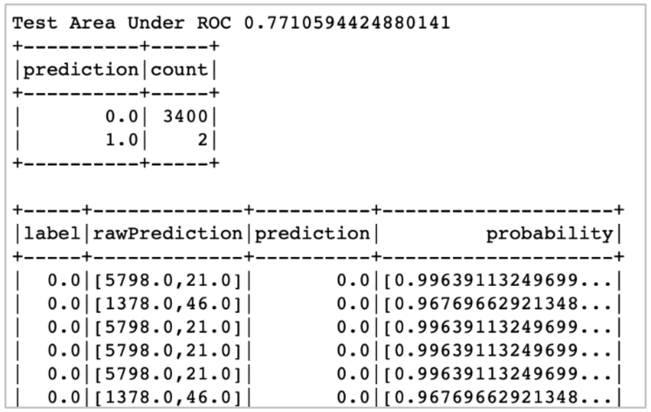
- You can save the results of the trained model for future use

```
filename1 = filename.replace('.','_') + '_DT_trainedModel'
dtModel.write().overwrite().save(filename1)

# load a saved trained model
dtModel2 = DecisionTreeClassifier.load(filename1)
```

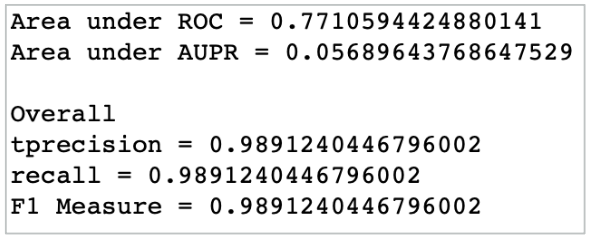
Interpret the Results

- After you train the model, you want to make predictions on the reserved test set and compare them to the known labels to see how well it did
  - `predictions = model.transform(test)`
- There are a lot of measures to see how good of a job it did
  - For convenience, we have wrapped them into a helper function in the `pyspark_helpers` package
  - `predict_and_evaluate` will return the predicted results and show how well it did

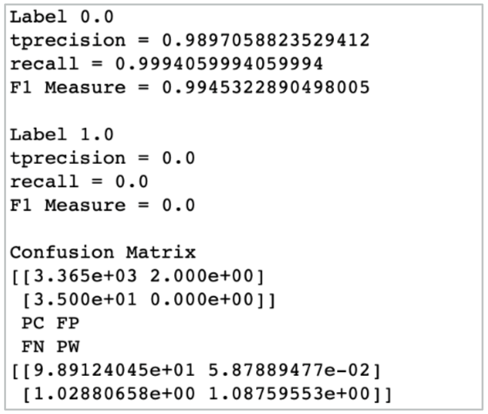
```
dtPredictions, dtLog = pyh.predict_and_evaluate(dtModel, test)
```



```
Test Area Under ROC 0.7710594424880141
+----------+-----+
|prediction|count|
+----------+-----+
|       0.0| 3400|
|       1.0|    2|
+----------+-----+

+-----+-------------+----------+--------------------+
|label|rawPrediction|prediction|         probability|
+-----+-------------+----------+--------------------+
|  0.0|[5798.0,21.0]|       0.0|[0.99639113249699...|
|  0.0|[1378.0,46.0]|       0.0|[0.96769662921348...|
|  0.0|[5798.0,21.0]|       0.0|[0.99639113249699...|
|  0.0|[5798.0,21.0]|       0.0|[0.99639113249699...|
|  0.0|[5798.0,21.0]|       0.0|[0.99639113249699...|
|  0.0|[1378.0,46.0]|       0.0|[0.96769662921348...|
```



```
Area under ROC = 0.7710594424880141
Area under AUPR = 0.05689643768647529

Overall
tprecision = 0.9891240446796002
recall = 0.9891240446796002
F1 Measure = 0.9891240446796002
```

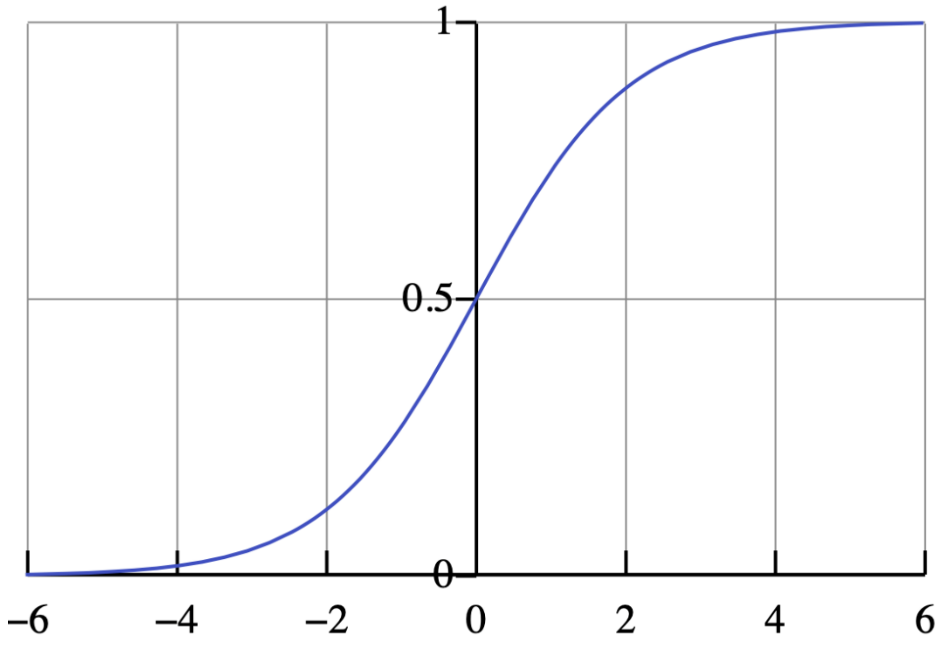


```
Label 0.0
tprecision = 0.9897058823529412
recall = 0.9994059994059994
F1 Measure = 0.9945322890498005

Label 1.0
tprecision = 0.0
recall = 0.0
F1 Measure = 0.0

Confusion Matrix
[[3.365e+03 2.000e+00]
 [3.500e+01 0.000e+00]]
 PC FP
 FN PW
[[9.89124045e+01 5.87889477e-02]
 [1.02880658e+00 1.08759553e+00]]
```

Logistic Regression

- Another alternative to categorizing, but with a twist
  - Not only predicts a value but predicts the probability of its occurrence
  - Allows you to change a probability threshold to favor false positives or false negatives
- The math behind it involves logarithms and finding coefficients of the independent variables

$$\ell = \log \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$



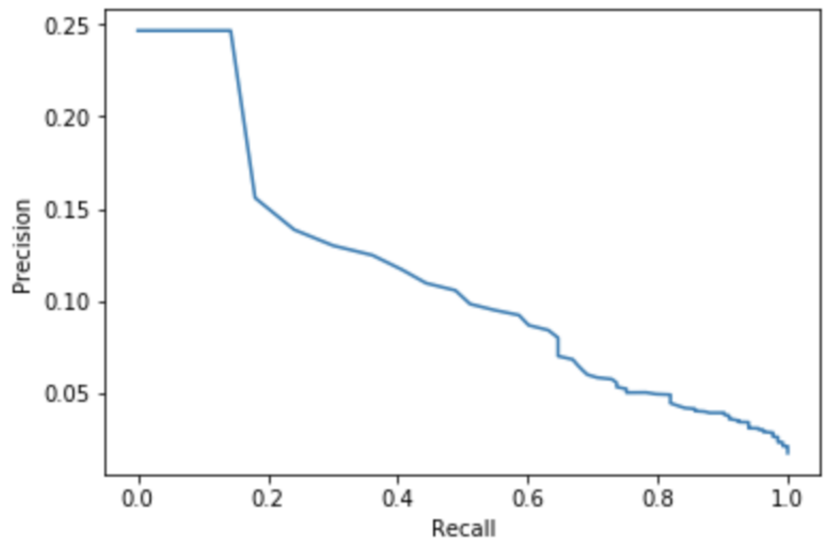
Logistic Regression (continued)

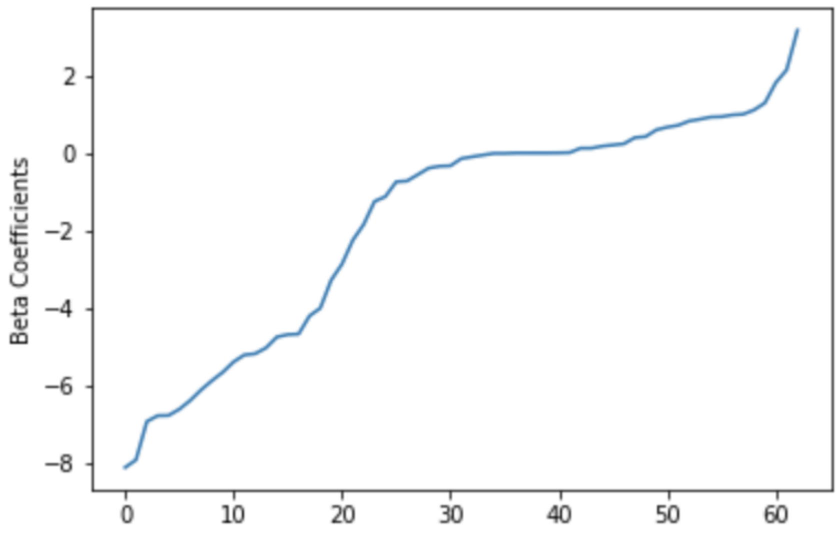
- Data needs to be dummy encoded skipping one value as a reference value
- Pros
  - Works better in cases with low signal to noise ratio
  - Allows for tweaking of false positives and false negatives
  - Transparency lets you see how it made its choices
- Cons
  - Does not perform well with too many features (independent variables)
  - Not good with large number of categorical values within a feature because of dummy encoding

Apply Logistic Regression

- Has some additional measurements to see how well it did

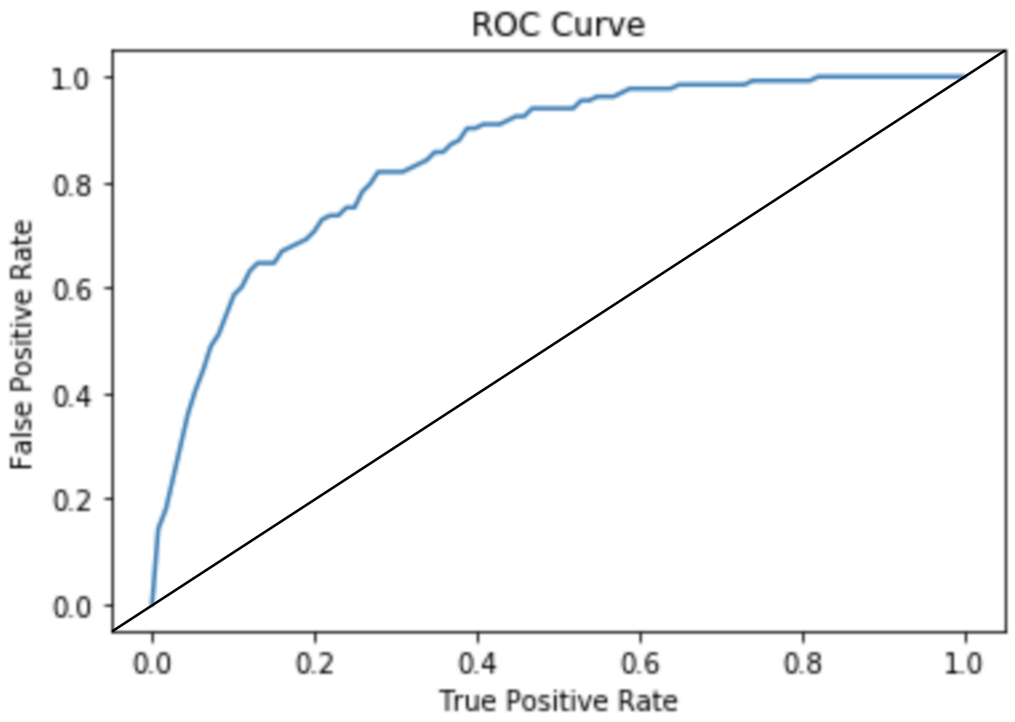
```
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(featuresCol = 'features', \
             labelCol = 'label', maxIter=10)
lrModel = lr.fit(train)
lrPredictions, lrLog = pyh.predict_and_evaluate(lrModel, test)
```





ROC Curve

- Shows the tradeoff between accuracy and sensitivity in adjusting the false positives
- The closer the curve is to the left or top border, the more accurate it is
- The closer to the 45-degree line, the less accurate it is



Random Forest

- Creates Decision Trees on randomly selected samples of the training set
- Performs multiple iterations and gets prediction results
- Votes on the best random sample
- Pros
  - Often highly accurate due to the strength of multiple predictions
  - Usually does not suffer from overfitting
  - Can see the relative feature importance which is useful in revising the model
- Cons
  - Slow to generate because of multiple iterations
  - Compared to a Decision Tree you cannot really see the path of the tree

Apply Random Forest

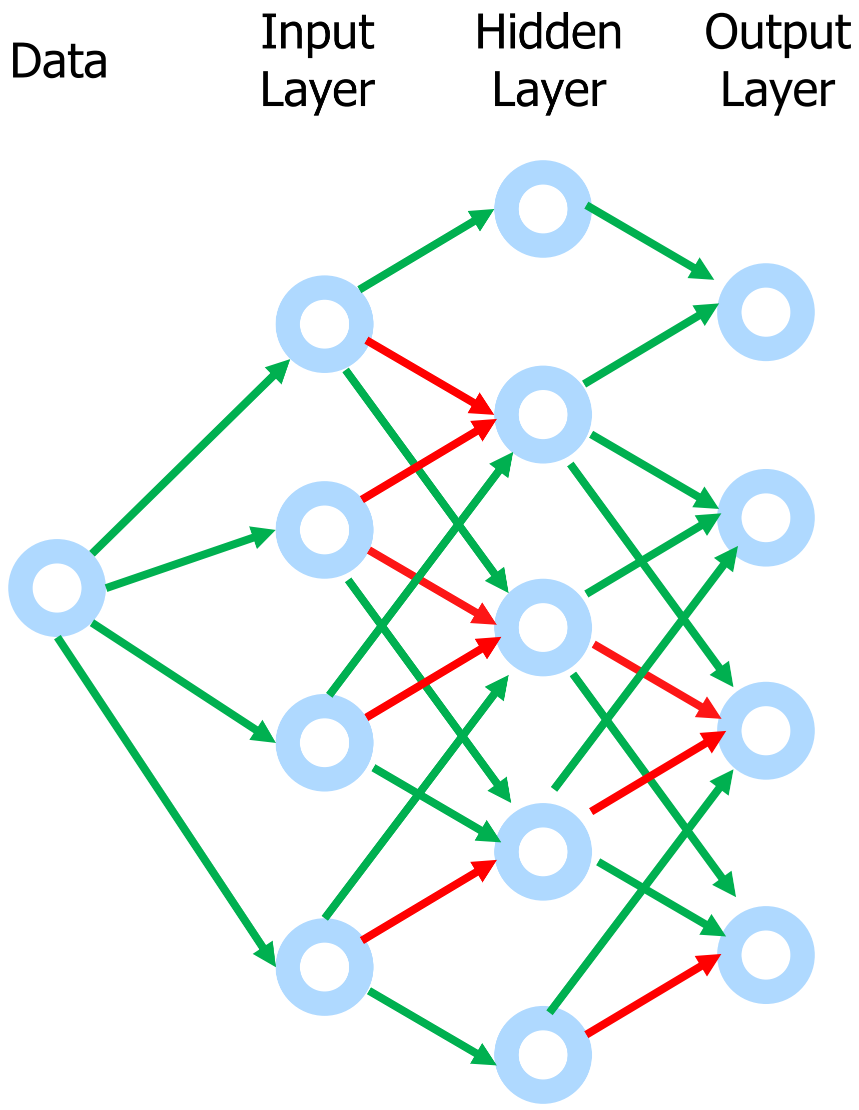
- Load the module, create a model, and train it

```
from pyspark.ml.classification import
RandomForestClassifier
rf = RandomForestClassifier(featuresCol = 'features', \
    labelCol = 'label')
rfModel = rf.fit(train)
```

Neural Networks

- Simulates the way the human brain solves
  - Uses a perceptron, algorithm, or function run in multiple layers
- Not only predicts a value but predicts the probability of its occurrence
  - Allows you to change a probability threshold to favor false positives or false negatives
- Pros
  - Often perform better than others which can be important where accuracy is desired (predicting cancer)
  - Good for unusual data like image, video, audio
- Cons
  - Black box, you don't know how it made its decision
  - Not appropriate in cases where transparency is important
  - Require a lot more data to train than other models
  - Computationally expensive
- Cool visualization of Neural Network from Google
  - https://playground.tensorflow.org/

Neural Network Visualized



Apply Neural Network

- Need to dummy encode categorical features

```
from pyspark.ml.classification import
MultilayerPerceptronClassifier
from pyspark.ml.evaluation import
MulticlassClassificationEvaluator

# specify layers for the neural network:
# input layer of size 13 (features), two intermediate of size
5 and 4
# and output of size 2 (classes)
layers = [13, 5, 4, 2]

nn = MultilayerPerceptronClassifier(maxIter=100,
layers=layers, blockSize=128, seed=1234)
nnModel = nn.fit(train)
```

Chapter Concepts

Classification Review

- Classification is one of the most widely used models
- It is supervised
- Good at predicting either/or or multiple-choice categories
- Lots of algorithms
- No one algorithm is best for all situations so often it involves running many of them, documenting the results, and choosing the best for your data and business case
- Can save the results of a lengthy training to a file and reload it for use with the predict function when needed

Next Steps

- Classification is one of the most useful models
- We have explored several different algorithms here and there are tons more, each with its own strengths and weaknesses

Chapter Summary

In this chapter, we have:

- Introduced Pipelines for efficient transformations
- Understood the use cases for Classification models
- Discussed and compared various algorithms
  - Naive Bayes
  - Decision Tree
  - Logistic Regression
  - Neural Network