**Chapter 26. Classification**

Classification is the task of predicting a label, category, class, or discrete variable given some input features. The key difference from other ML tasks, such as regression, is that the output label has a finite set of possible values (e.g., three classes).

**Use Cases**

Classification has many use cases, as we discussed in [Chapter 24](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch24.html#s6c1---advanced-analytics-and-machine-learning). Here are a few more to consider as a reinforcement of the multitude of ways classification can be used in the real world.

Predicting credit risk

A financing company might look at a number of variables before offering a loan to a company or individual. Whether or not to offer the loan is a binary classification problem.

News classification

An algorithm might be trained to predict the topic of a news article (sports, politics, business, etc.).

Classifying human activity

By collecting data from sensors such as a phone accelerometer or smart watch, you can predict the person’s activity. The output will be one of a finite set of classes (e.g., walking, sleeping, standing, or running).

**Types of Classification**

Before we continue, let’s review several different types of classification.

**Binary Classification**

The simplest example of classification is *binary classification*, where there are only two labels you can predict. One example is fraud analytics, where a given transaction can be classified as fraudulent or not; or email spam, where a given email can be classified as spam or not spam.

**Multiclass Classification**

Beyond binary classification lies *multiclass classification*, where one label is chosen from more than two distinct possible labels. A typical example is Facebook predicting the people in a given photo or a meterologist predicting the weather (rainy, sunny, cloudy, etc.). Note how there is always a finite set of classes to predict; it’s never unbounded. This is also called multinomial classification.

**Multilabel Classification**

Finally, there is *multilabel classification*, where a given input can produce multiple labels. For example, you might want to predict a book’s genre based on the text of the book itself. While this could be multiclass, it’s probably better suited for multilabel because a book may fall into multiple genres. Another example of multilabel classification is identifying the number of objects that appear in an image. Note that in this example, the number of output predictions is not necessarily fixed, and could vary from image to image.

**Classification Models in MLlib**

Spark has several models available for performing binary and multiclass classification out of the box. The following models are available for classification in Spark:

* Logistic regression
* Decision trees
* Random forests
* Gradient-boosted trees

Spark does not support making multilabel predictions natively. In order to train a multilabel model, you must train one model per label and combine them manually. Once manually constructed, there are built-in tools that support measuring these kinds of models (discussed at the end of the chapter).

This chapter will cover the basics of each of these models by providing:

* A simple explanation of the model and the intuition behind it
* Model hyperparameters (the different ways we can initialize the model)
* Training parameters (parameters that affect how the model is trained)
* Prediction parameters (parameters that affect how predictions are made)

You can set the hyperparameters and training parameters in a ParamGrid as we saw in [Chapter 24](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch24.html#s6c1---advanced-analytics-and-machine-learning).

**Model Scalability**

Model scalability is an important consideration when choosing your model. In general, Spark has great support for training large-scale machine learning models (note, these are *large scale*; on single-node workloads there are a number of other tools that also perform well). Table 26-1 is a simple model scalability scorecard to use to find the best model for your particular task (if scalability is your core consideration). The actual scalability will depend on your configuration, machine size, and other specifics but should make for a good heuristic.

*Table 26-1. Model scalability reference*

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Features count** | **Training examples** | **Output classes** |
| Logistic regression | 1 to 10 million | No limit | Features x Classes < 10 million |
| Decision trees | 1,000s | No limit | Features x Classes < 10,000s |
| Random forest | 10,000s | No limit | Features x Classes < 100,000s |
| Gradient-boosted trees | 1,000s | No limit | Features x Classes < 10,000s |

We can see that nearly all these models scale to large collections of input data and there is ongoing work to scale them even further. The reason *no limit* is in place for the number of training examples is because these are trained using methods like stochastic gradient descent and L-BFGS. These methods are optimized specifically for working with massive datasets and to remove any constraints that might exist on the number of training examples you would hope to learn on.

Let’s start looking at the classification models by loading in some data:

*// in Scala*

**val** bInput **=** spark.read.format("parquet").load("/data/binary-classification")

.selectExpr("features", "cast(label as double) as label")

*# in Python*

bInput = spark.read.format("parquet").load("/data/binary-classification")\

.selectExpr("features", "cast(label as double) as label")

**NOTE**

Like our other advanced analytics chapters, this one cannot teach you the mathematical underpinnings of every model. See [Chapter 4](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch04.html#s2c1---structured-api-overview) in [ISL](http://www-bcf.usc.edu/~gareth/ISL/) and [ESL](http://statweb.stanford.edu/~tibs/ElemStatLearn/) for a review of classification.

**Logistic Regression**

Logistic regression is one of the most popular methods of classification. It is a linear method that combines each of the individual inputs (or features) with specific weights (these weights are generated during the training process) that are then combined to get a probability of belonging to a particular class. These weights are helpful because they are good representations of feature importance; if you have a large weight, you can assume that variations in that feature have a significant effect on the outcome (assuming you performed normalization). A smaller weight means the feature is less likely to be important.

See [ISL 4.3](http://www-bcf.usc.edu/~gareth/ISL/) and [ESL 4.4](http://statweb.stanford.edu/~tibs/ElemStatLearn/) for more information.

**Model Hyperparameters**

Model hyperparameters are configurations that determine the basic structure of the model itself. The following hyperparameters are available for logistic regression:

family

Can be multinomial (two or more distinct labels; multiclass classification) or binary (only two distinct labels; binary classification).

elasticNetParam

A floating-point value from 0 to 1. This parameter specifies the mix of L1 and L2 regularization according to elastic net regularization (which is a linear combination of the two). Your choice of L1 or L2 depends a lot on your particular use case but the intuition is as follows: L1 regularization (a value of 1) will create sparsity in the model because certain feature weights will become zero (that are of little consequence to the output). For this reason, it can be used as a simple feature-selection method. On the other hand, L2 regularization (a value of 0) does not create sparsity because the corresponding weights for particular features will only be driven toward zero, but will never completely reach zero. ElasticNet gives us the best of both worlds—we can choose a value between 0 and 1 to specify a mix of L1 and L2 regularization. For the most part, you should be tuning this by testing different values.

fitIntercept

Can be true or false. This hyperparameter determines whether or not to fit the intercept or the arbitrary number that is added to the linear combination of inputs and weights of the model. Typically you will want to fit the intercept if we haven’t normalized our training data.

regParam

A value ≥ 0. that determines how much weight to give to the regularization term in the objective function. Choosing a value here is again going to be a function of noise and dimensionality in our dataset. In a pipeline, try a wide range of values (e.g., 0, 0.01, 0.1, 1).

standardization

Can be true or false, whether or not to standardize the inputs before passing them into the model. See [Chapter 25](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch25.html#s6c2---preprocessing-and-feature-engineering) for more information.

**Training Parameters**

Training parameters are used to specify how we perform our training. Here are the training parameters for logistic regression.

maxIter

Total number of iterations over the data before stopping. Changing this parameter probably won’t change your results a ton, so it shouldn’t be the first parameter you look to adjust. The default is 100.

tol

This value specifies a threshold by which changes in parameters show that we optimized our weights enough, and can stop iterating. It lets the algorithm stop before maxIter iterations. The default value is 1.0E-6. This also shouldn’t be the first parameter you look to tune.

weightCol

The name of a weight column used to weigh certain rows more than others. This can be a useful tool if you have some other measure of how important a particular training example is and have a weight associated with it. For example, you might have 10,000 examples where you know that some labels are more accurate than others. You can weigh the labels you know are correct more than the ones you don’t.

**Prediction Parameters**

These parameters help determine how the model should actually be making predictions at prediction time, but do not affect training. Here are the prediction parameters for logistic regression:

threshold

A Double in the range of 0 to 1. This parameter is the probability threshold for when a given class should be predicted. You can tune this parameter according to your requirements to balance between false positives and false negatives. For instance, if a mistaken prediction would be costly—you might want to make its prediction threshold very high.

thresholds

This parameter lets you specify an array of threshold values for each class when using multiclass classification. It works similarly to the single threshold parameter described previously.

**Example**

Here’s a simple example using the LogisticRegression model. Notice how we didn’t specify any parameters because we’ll leverage the defaults and our data conforms to the proper column naming. In practice, you probably won’t need to change many of the parameters:

*// in Scala*

**import** **org.apache.spark.ml.classification.LogisticRegression**

**val** lr **=** **new** **LogisticRegression**()

println(lr.explainParams()) *// see all parameters*

**val** lrModel **=** lr.fit(bInput)

*# in Python*

**from** **pyspark.ml.classification** **import** LogisticRegression

lr = LogisticRegression()

**print** lr.explainParams() *# see all parameters*

lrModel = lr.fit(bInput)

Once the model is trained you can get information about the model by taking a look at the coefficients and the intercept. The coefficients correspond to the individual feature weights (each feature weight is multiplied by each respective feature to compute the prediction) while the intercept is the value of the italics-intercept (if we chose to fit one when specifying the model). Seeing the coefficients can be helpful for inspecting the model that you built and comparing how features affect the prediction:

*// in Scala*

println(lrModel.coefficients)

println(lrModel.intercept)

*# in Python*

**print** lrModel.coefficients

**print** lrModel.intercept

For a multinomial model (the current one is binary), lrModel.coefficientMatrix and lrModel.interceptVector can be used to get the coefficients and intercept. These will return Matrix and Vector types representing the values or each of the given classes.

**Model Summary**

Logistic regression provides a model summary that gives you information about the final, trained model. This is analogous to the same types of summaries we see in many R language machine learning packages. The model summary is currently only available for binary logistic regression problems, but multiclass summaries will likely be added in the future. Using the binary summary, we can get all sorts of information about the model itself including the area under the ROC curve, the f measure by threshold, the precision, the recall, the recall by thresholds, and the ROC curve. Note that for the area under the curve, instance weighting is not taken into account, so if you wanted to see how you performed on the values you weighed more highly, you’d have to do that manually. This will probably change in future Spark versions. You can see the summary using the following APIs:

*// in Scala*

**import** **org.apache.spark.ml.classification.BinaryLogisticRegressionSummary**

**val** summary **=** lrModel.summary

**val** bSummary **=** summary.asInstanceOf[**BinaryLogisticRegressionSummary**]

println(bSummary.areaUnderROC)

bSummary.roc.show()

bSummary.pr.show()

*# in Python*

summary = lrModel.summary

**print** summary.areaUnderROC

summary.roc.show()

summary.pr.show()

The speed at which the model descends to the final result is shown in the objective history. We can access this through the objective history on the model summary:

summary.objectiveHistory

This is an array of doubles that specify how, over each training iteration, we are performing with respect to our objective function. This information is helpful to see if we have sufficient iterations or need to be tuning other parameters.

**Decision Trees**

Decision trees are one of the more friendly and interpretable models for performing classification because they’re similar to simple decision models that humans use quite often. For example, if you have to predict whether or not someone will eat ice cream when offered, a good feature might be whether or not that individual likes ice cream. In pseudocode, if person.likes(“ice\_cream”), they will eat ice cream; otherwise, they won’t eat ice cream. A decision tree creates this type of structure with all the inputs and follows a set of branches when it comes time to make a prediction. This makes it a great starting point model because it’s easy to reason about, easy to inspect, and makes very few assumptions about the structure of the data. In short, rather than trying to train coeffiecients in order to model a function, it simply creates a big tree of decisions to follow at prediction time. This model also supports multiclass classification and provides outputs as predictions and probabilities in two different columns.

While this model is usually a great start, it does come at a cost. It can overfit data *extremely* quickly. By that we mean that, unrestrained, the decision tree will create a pathway from the start based on every single training example. That means it encodes all of the information in the training set in the model. This is bad because then the model won’t generalize to new data (you will see poor test set prediction performance). However, there are a number of ways to try and rein in the model by limiting its branching structure (e.g., limiting its height) to get good predictive power.

See [ISL 8.1](http://www-bcf.usc.edu/~gareth/ISL/) and [ESL 9.2](http://statweb.stanford.edu/~tibs/ElemStatLearn/) for more information.

**Model Hyperparameters**

There are many different ways to configure and train decision trees. Here are the hyperparameters that Spark’s implementation supports:

maxDepth

Since we’re training a tree, it can be helpful to specify a max depth in order to avoid overfitting to the dataset (in the extreme, every row ends up as its own leaf node). The default is 5.

maxBins

In decision trees, continuous features are converted into categorical features and maxBins determines how many bins should be created from continous features. More bins gives a higher level of granularity. The value must be greater than or equal to 2 and greater than or equal to the number of categories in any categorical feature in your dataset. The default is 32.

impurity

To build up a “tree” you need to configure when the model should branch. Impurity represents the metric (information gain) to determine whether or not the model should split at a particular leaf node. This parameter can be set to either be “entropy” or “gini” (default), two commonly used impurity metrics.

minInfoGain

This parameter determines the minimum information gain that can be used for a split. A higher value can prevent overfitting. This is largely something that needs to be determined from testing out different variations of the decision tree model. The default is zero.

minInstancePerNode

This parameter determines the minimum number of training instances that need to end in a particular node. Think of this as another manner of controlling max depth. We can prevent overfitting by limiting depth or we can prevent it by specifying that at minimum a certain number of training values need to end up in a particular leaf node. If it’s not met we would “prune” the tree until that requirement is met. A higher value can prevent overfitting. The default is 1, but this can be any value greater than 1.

**Training Parameters**

These are configurations we specify in order to manipulate how we perform our training. Here is the training parameter for decision trees:

checkpointInterval

Checkpointing is a way to save the model’s work over the course of training so that if nodes in the cluster crash for some reason, you don’t lose your work. A value of 10 means the model will get checkpointed every 10 iterations. Set this to -1 to turn off checkpointing. This parameter needs to be set together with a checkpointDir (a directory to checkpoint to) and with useNodeIdCache=true. Consult the Spark documentation for more information on checkpointing.

**Prediction Parameters**

There is only one prediction parameter for decision trees: thresholds. Refer to the explanation for thresholds under “Logistic Regression”.

Here’s a minimal but complete example of using a decision tree classifier:

*// in Scala*

**import** **org.apache.spark.ml.classification.DecisionTreeClassifier**

**val** dt **=** **new** **DecisionTreeClassifier**()

println(dt.explainParams())

**val** dtModel **=** dt.fit(bInput)

*# in Python*

**from** **pyspark.ml.classification** **import** DecisionTreeClassifier

dt = DecisionTreeClassifier()

**print** dt.explainParams()

dtModel = dt.fit(bInput)

**Random Forest and Gradient-Boosted Trees**

These methods are extensions of the decision tree. Rather than training one tree on all of the data, you train multiple trees on varying subsets of the data. The intuition behind doing this is that various decision trees will become “experts” in that particular domain while others become experts in others. By combining these various experts, you then get a “wisdom of the crowds” effect, where the group’s performance exceeds any individual. In addition, these methods can help prevent overfitting.

Random forests and gradient-boosted trees are two distinct methods for combining decision trees. In random forests, we simply train a lot of trees and then average their response to make a prediction. With gradient-boosted trees, each tree makes a weighted prediction (such that some trees have more predictive power for some classes than others). They have largely the same parameters, which we note below. One current limitation is that gradient-boosted trees currently only support binary labels.

**NOTE**

There are several popular tools for learning tree-based models. For example, the [XGBoost](https://xgboost.readthedocs.io/en/latest/) library provides an integration package for Spark that can be used to run it on Spark.

See [ISL](http://www-bcf.usc.edu/~gareth/ISL/) 8.2 and [ESL](http://statweb.stanford.edu/~tibs/ElemStatLearn/) 10.1 for more information on these tree ensemble models.

**Model Hyperparameters**

Random forests and gradient-boosted trees provide all of the same model hyperparameters supported by decision trees. In addition, they add several of their own.

**RANDOM FOREST ONLY**

numTrees

The total number of trees to train.

featureSubsetStrategy

This parameter determines how many features should be considered for splits. This can be a variety of different values including “auto”, “all”, “sqrt”, “log2”, or a number “n.” When your input is “n” the model will use n \* number of features during training. When n is in the range (1, number of features), the model will use n features during training. There’s no one-size-fits-all solution here, so it’s worth experimenting with different values in your pipeline.

**GRADIENT-BOOSTED TREES (GBT) ONLY**

lossType

This is the loss function for gradient-boosted trees to minimize during training. Currently, only logistic loss is supported.

maxIter

Total number of iterations over the data before stopping. Changing this probably won’t change your results a ton, so it shouldn’t be the first parameter you look to adjust. The default is 100.

stepSize

This is the learning rate for the algorithm. A larger step size means that larger jumps are made between training iterations. This can help in the optimization process and is something that should be tested in training. The default is 0.1 and this can be any value from 0 to 1.

**Training Parameters**

There is only one training parameter for these models, checkpointInterval. Refer back to the explanation under “Decision Trees” for details on checkpointing.

**Prediction Parameters**

These models have the same prediction parameters as decision trees. Consult the prediction parameters under that model for more information.

Here’s a short code example of using each of these classifiers:

*// in Scala*

**import** **org.apache.spark.ml.classification.RandomForestClassifier**

**val** rfClassifier **=** **new** **RandomForestClassifier**()

println(rfClassifier.explainParams())

**val** trainedModel **=** rfClassifier.fit(bInput)

*// in Scala*

**import** **org.apache.spark.ml.classification.GBTClassifier**

**val** gbtClassifier **=** **new** **GBTClassifier**()

println(gbtClassifier.explainParams())

**val** trainedModel **=** gbtClassifier.fit(bInput)

*# in Python*

**from** **pyspark.ml.classification** **import** RandomForestClassifier

rfClassifier = RandomForestClassifier()

**print** rfClassifier.explainParams()

trainedModel = rfClassifier.fit(bInput)

*# in Python*

**from** **pyspark.ml.classification** **import** GBTClassifier

gbtClassifier = GBTClassifier()

**print** gbtClassifier.explainParams()

trainedModel = gbtClassifier.fit(bInput)

**Naive Bayes**

Naive Bayes classifiers are a collection of classifiers based on Bayes’ theorem. The core assumption behind the models is that all features in your data are independent of one another. Naturally, strict independence is a bit naive, but even if this is violated, useful models can still be produced. Naive Bayes classifiers are commonly used in text or document classification, although it can be used as a more general-purpose classifier as well. There are two different model types: either a *multivariate Bernoulli model*, where indicator variables represent the existence of a term in a document; or the *multinomial model*, where the total counts of terms are used.

One important note when it comes to Naive Bayes is that all input features must be non-negative.

See [ISL 4.4](http://www-bcf.usc.edu/~gareth/ISL/) and [ESL 6.6](http://statweb.stanford.edu/~tibs/ElemStatLearn/) for more background on these models.

**Model Hyperparameters**

These are configurations we specify to determine the basic structure of the models:

modelType

Either “bernoulli” or “multinomial.” See the previous section for more information on this choice.

weightCol

Allows weighing different data points differently. Refer back to “Training Parameters” for the explanation of this hyperparameter.

**Training Parameters**

These are configurations that specify how we perform our training:

smoothing

This determines the amount of regularization that should take place using [additive smoothing](https://en.wikipedia.org/wiki/Additive_smoothing). This helps smooth out categorical data and avoid overfitting on the training data by changing the expected probability for certain classes. The default value is 1.

**Prediction Parameters**

Naive Bayes shares the same prediction parameter, thresholds, as all of our other models. Refer back to the previous explanation for threshold to see how to use this.

Here’s an example of using a Naive Bayes classifier.

*// in Scala*

**import** **org.apache.spark.ml.classification.NaiveBayes**

**val** nb **=** **new** **NaiveBayes**()

println(nb.explainParams())

**val** trainedModel **=** nb.fit(bInput.where("label != 0"))

*# in Python*

**from** **pyspark.ml.classification** **import** NaiveBayes

nb = NaiveBayes()

**print** nb.explainParams()

trainedModel = nb.fit(bInput.where("label != 0"))

**WARNING**

Note that in this example dataset, we have features that have negative values. In this case, the rows with negative features correspond to rows with label “0”. Therefore we’re just going to filter them out (via the label) instead of processing them further to demonstrate the naive bayes API.

**Evaluators for Classification and Automating Model Tuning**

As we saw in [Chapter 24](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch24.html#s6c1---advanced-analytics-and-machine-learning), evaluators allow us to specify the metric of success for our model. An evaluator doesn’t help too much when it stands alone; however, when we use it in a pipeline, we can automate a grid search of our various parameters of the models and transformers—trying all combinations of the parameters to see which ones perform the best. Evaluators are most useful in this pipeline and parameter grid context. For classification, there are two evaluators, and they expect two columns: a predicted label from the model and a true label. For binary classification we use the BinaryClassificationEvaluator. This supports optimizing for two different metrics “areaUnderROC” and areaUnderPR.” For multiclass classification, we need to use the MulticlassClassificationEvaluator, which supports optimizing for “f1”, “weightedPrecision”, “weightedRecall”, and “accuracy”.

To use evaluators, we build up our pipeline, specify the parameters we would like to test, and then run it and see the results. See [Chapter 24](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch24.html#s6c1---advanced-analytics-and-machine-learning) for a code example.

**Detailed Evaluation Metrics**

MLlib also contains tools that let you evaluate multiple classification metrics at once. Unfortunately, these metrics classes have not been ported over to Spark’s DataFrame-based ML package from the underlying RDD framework. So, at the time of this writing, you still have to create an RDD to use these. In the future, this functionality will likely be ported to DataFrames and the following may no longer be the best way to see metrics (although you will still be able to use these APIs).

There are three different classification metrics we can use:

* Binary classification metrics
* Multiclass classification metrics
* Multilabel classification metrics

All of these measures follow the same approximate style. We’ll compare generated outputs with true values and the model calculates all of the relevant metrics for us. Then we can query the object for the values for each of the metrics:

*// in Scala*

**import** **org.apache.spark.mllib.evaluation.BinaryClassificationMetrics**

**val** out **=** model.transform(bInput)

.select("prediction", "label")

.rdd.map(x **=>** (x(0).asInstanceOf[**Double**], x(1).asInstanceOf[**Double**]))

**val** metrics **=** **new** **BinaryClassificationMetrics**(out)

*# in Python*

**from** **pyspark.mllib.evaluation** **import** BinaryClassificationMetrics

out = model.transform(bInput)\

.select("prediction", "label")\

.rdd.map(**lambda** x: (float(x[0]), float(x[1])))

metrics = BinaryClassificationMetrics(out)

Once we’ve done that, we can see typical classification success metrics on this metric’s object using a similar API to the one we saw with logistic regression:

*// in Scala*

metrics.areaUnderPR

metrics.areaUnderROC

println("Receiver Operating Characteristic")

metrics.roc.toDF().show()

*# in Python*

**print** metrics.areaUnderPR

**print** metrics.areaUnderROC

**print** "Receiver Operating Characteristic"

metrics.roc.toDF().show()

**One-vs-Rest Classifier**

There are some MLlib models that don’t support multiclass classification. In these cases, users can leverage a one-vs-rest classifier in order to perform multiclass classification given only a binary classifier. The intuition behind this is that for every class you hope to predict, the one-vs-rest classifier will turn the problem into a binary classification problem by isolating one class as the target class and grouping all of the other classes into one. Thus the prediction of the class becomes binary (is it this class or *not* this class?).

One-vs-rest is implemented as an estimator. For the base classifier it takes instances of the classifier and creates a binary classification problem for each of the 𝘬 classes. The classifier for class *i* is trained to predict whether the label is *i* or not, distinguishing class *i* from all other classes.

Predictions are done by evaluating each binary classifier and the index of the most confident classifier is output as the label.

See the Spark documentation for a nice example of the [use of one-vs-rest](http://bit.ly/2BxBwVI).

**Multilayer Perceptron**

The multilayer perceptron is a classifier based on neural networks with a configurable number of layers (and layer sizes). We will discuss it in [Chapter 31](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch31.html#s6c8---deep-learning).

**Conclusion**

In this chapter we covered the majority of tools Spark provides for classification: predicting one of a finite set of labels for each data point based on its features. In the next chapter, we’ll look at regression, where the required output is continuous instead of categorical.