**Chapter 27. Regression**

Regression is a logical extension of classification. Rather than just predicting a single value from a set of values, *regression* is the act of predicting a real number (or continuous variable) from a set of features (represented as numbers).

Regression can be harder than classification because, from a mathematical perspective, there are an infinite number of possible output values. Furthermore, we aim to optimize some metric of error between the predicted and true value, as opposed to an accuracy rate. Aside from that, regression and classification are fairly similar. For this reason, we will see a lot of the same underlying concepts applied to regression as we did with classification.

**Use Cases**

The following is a small set of regression use cases that can get you thinking about potential regression problems in your own domain:

Predicting movie viewership

Given information about a movie and the movie-going public, such as how many people have watched the trailer or shared it on social media, you might want to predict how many people are likely to watch the movie when it comes out.

Predicting company revenue

Given a current growth trajectory, the market, and seasonality, you might want to predict how much revenue a company will gain in the future.

Predicting crop yield

Given information about the particular area in which a crop is grown, as well as the current weather throughout the year, you might want to predict the total crop yield for a particular plot of land.

**Regression Models in MLlib**

There are several fundamental regression models in MLlib. Some of these models are carryovers from [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification). Others are only relevant to the regression problem domain. This list is current as of Spark 2.2 but will grow:

* Linear regression
* Generalized linear regression
* Isotonic regression
* Decision trees
* Random forest
* Gradient-boosted trees
* Survival regression

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

* A simple explanation of the model and the intuition behind the algorithm
* Model hyperparameters (the different ways that 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 search over the hyperparameters and training parameters using 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**

The regression models in MLlib all scale to large datasets. Table 27-1 is a simple model scalability scorecard that will help you in choosing the best model for your particular task (if scalability is your core consideration). These will depend on your configuration, machine size, and other factors.

*Table 27-1. Regression scalability reference*

|  |  |  |
| --- | --- | --- |
| **Model** | **Number features** | **Training examples** |
| Linear regression | 1 to 10 million | No limit |
| Generalized linear regression | 4,096 | No limit |
| Isotonic regression | N/A | Millions |
| Decision trees | 1,000s | No limit |
| Random forest | 10,000s | No limit |
| Gradient-boosted trees | 1,000s | No limit |
| Survival regression | 1 to 10 million | No limit |

**NOTE**

Like our other advanced analytics chapters, this one cannot teach you the mathematical underpinnings of every model. See Chapter 3 in [ISL](http://www-bcf.usc.edu/~gareth/ISL/) and [ESL](http://statweb.stanford.edu/~tibs/ElemStatLearn/) for a review of regression.

Let’s read in some sample data that we will use throughout the chapter:

*// in Scala*

**val** df **=** spark.read.load("/data/regression")

*# in Python*

df = spark.read.load("/data/regression")

**Linear Regression**

*Linear regression* assumes that a linear combination of your input features (the sum of each feature multiplied by a weight) results along with an amount of Gaussian error in the output. This linear assumption (along with Gaussian error) does not always hold true, but it does make for a simple, interpretable model that’s hard to overfit. Like logistic regression, Spark implements ElasticNet regularization for this, allowing you to mix L1 and L2 regularization.

See [ISL](http://www-bcf.usc.edu/~gareth/ISL/) 3.2 and [ESL](http://statweb.stanford.edu/~tibs/ElemStatLearn/) 3.2 for more information.

**Model Hyperparameters**

Linear regression has the same model hyperparameters as logistic regression. See [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification) for more information.

**Training Parameters**

Linear regression also shares all of the same training parameters from logistic regression. Refer back to [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification) for more on this topic.

**Example**

Here’s a short example of using linear regression on our sample dataset:

*// in Scala*

**import** **org.apache.spark.ml.regression.LinearRegression**

**val** lr **=** **new** **LinearRegression**().setMaxIter(10).setRegParam(0.3)\

.setElasticNetParam(0.8)

println(lr.explainParams())

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

*# in Python*

**from** **pyspark.ml.regression** **import** LinearRegression

lr = LinearRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)

**print** lr.explainParams()

lrModel = lr.fit(df)

**Training Summary**

Just as in logistic regression, we get detailed training information back from our model. The code font method is a simple shorthand for accessing these metrics. It reports several conventional metrics for measuring the success of a regression model, allowing you to see how well your model is actually fitting the line.

The summary method returns a summary object with several fields. Let’s go through these in turn. The residuals are simply the weights for each of the features that we input into the model. The objective history shows how our training is going at every iteration. The root mean squared error is a measure of how well our line is fitting the data, determined by looking at the distance between each predicted value and the actual value in the data. The R-squared variable is a measure of the proportion of the variance of the predicted variable that is captured by the model.

There are a number of metrics and summary information that may be relevant to your use case. This section demonstrates the API, but does not comprehensively cover every metric (consult the API documentation for more information).

Here are some of the attributes of the model summary for linear regression:

*// in Scala*

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

summary.residuals.show()

println(summary.objectiveHistory.toSeq.toDF.show())

println(summary.rootMeanSquaredError)

println(summary.r2)

*# in Python*

summary = lrModel.summary

summary.residuals.show()

**print** summary.totalIterations

**print** summary.objectiveHistory

**print** summary.rootMeanSquaredError

**print** summary.r2

**Generalized Linear Regression**

The standard linear regression that we saw in this chapter is actually a part of a family of algorithms called *generalized linear regression*. Spark has two implementations of this algorithm. One is optimized for working with very large sets of features (the simple linear regression covered previously in this chapter), while the other is more general, includes support for more algorithms, and doesn’t currently scale to large numbers of features.

The generalized form of linear regression gives you more fine-grained control over what kind of regression model you use. For instance, these allow you to select the expected noise distribution from a variety of families, including Gaussian (linear regression), binomial (logistic regression), poisson (poisson regression), and gamma (gamma regression). The generalized models also support setting a link function that specifies the relationship between the linear predictor and the mean of the distribution function. Table 27-2 shows the available link functions for each family.

*Table 27-2. Regression families, response types, and link functions*

|  |  |  |
| --- | --- | --- |
| **Family** | **Response type** | **Supported links** |
| Gaussian | Continuous | Identity\*, Log, Inverse |
| Binomial | Binary | Logit\*, Probit, CLogLog |
| Poisson | Count | Log\*, Identity, Sqrt |
| Gamma | Continuous | Inverse\*, Idenity, Log |
| Tweedie | Zero-inflated continuous | Power link function |

The asterisk signifies the canonical link function for each family.

See [ISL 3.2](http://www-bcf.usc.edu/~gareth/ISL/) and [ESL 3.2](http://statweb.stanford.edu/~tibs/ElemStatLearn/) for more information on generalized linear models.

**WARNING**

A fundamental limitation as of Spark 2.2 is that generalized linear regression only accepts a maximum of 4,096 features for inputs. This will likely change for later versions of Spark, so be sure to refer to the documentation.

**Model Hyperparameters**

These are configurations that we specify to determine the basic structure of the model itself. In addition to fitIntercept and regParam (mentioned in [“Regression”](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch24.html#regression)), generalized linear regression includes several other hyperparameters:

family

A description of the error distribution to be used in the model. Supported options are Poisson, binomial, gamma, Gaussian, and tweedie.

link

The name of link function which provides the relationship between the linear predictor and the mean of the distribution function. Supported options are cloglog, probit, logit, inverse, sqrt, identity, and log (default: identity).

solver

The solver algorithm to be used for optimization. The only currently supported solver is irls (iteratively reweighted least squares).

variancePower

The power in the variance function of the Tweedie distribution, which characterizes the relationship between the variance and mean of the distribution. Only applicable to the Tweedie family. Supported values are 0 and [1, Infinity). The default is 0.

linkPower

The index in the power link function for the Tweedie family.

**Training Parameters**

The training parameters are the same that you will find for logistic regression. Consult [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification) for more information.

**Prediction Parameters**

This model adds one prediction parameter:

linkPredictionCol

A column name that will hold the output of our link function for each prediction.

**Example**

Here’s an example of using GeneralizedLinearRegression:

*// in Scala*

**import** **org.apache.spark.ml.regression.GeneralizedLinearRegression**

**val** glr **=** **new** **GeneralizedLinearRegression**()

.setFamily("gaussian")

.setLink("identity")

.setMaxIter(10)

.setRegParam(0.3)

.setLinkPredictionCol("linkOut")

println(glr.explainParams())

**val** glrModel **=** glr.fit(df)

*# in Python*

**from** **pyspark.ml.regression** **import** GeneralizedLinearRegression

glr = GeneralizedLinearRegression()\

.setFamily("gaussian")\

.setLink("identity")\

.setMaxIter(10)\

.setRegParam(0.3)\

.setLinkPredictionCol("linkOut")

**print** glr.explainParams()

glrModel = glr.fit(df)

**Training Summary**

As for the simple linear model in the previous section, the training summary provided by Spark for the generalized linear model can help you ensure that your model is a good fit for the data that you used as the training set. It is important to note that this does not replace running your algorithm against a proper test set, but it can provide more information. This information includes a number of different potential metrics for analyzing the fit of your algorithm, including some of the most common success metrics:

R squared

The coefficient of determination; a measure of fit.

The residuals

The difference between the label and the predicted value.

Be sure to inspect the summary object on the model to see all the available methods.

**Decision Trees**

Decision trees as applied to regression work fairly similarly to decision trees applied to classification. The main difference is that decision trees for regression output a single number per leaf node instead of a label (as we saw with classification). The same interpretability properties and model structure still apply. In short, rather than trying to train coeffiecients to model a function, decision tree regression simply creates a tree to predict the numerical outputs. This is of significant consequence because unlike generalized linear regression, we can predict nonlinear functions in the input data. This also creates a significant risk of overfitting the data, so we need to be careful when tuning and evaluating these models.

We also covered decision trees in [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification) (refer to [“Decision Trees”](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#decision-trees)). For more information on this topic, consult [ISL 8.1](http://www-bcf.usc.edu/~gareth/ISL) and [ESL 9.2](http://statweb.stanford.edu/~tibs/ElemStatLearn/).

**Model Hyperparameters**

The model hyperparameters that apply decision trees for regression are the same as those for classification except for a slight change to the impurity parameter. See [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification) for more information on the other hyperparameters:

impurity

The impurity parameter represents the metric (information gain) for whether or not the model should split at a particular leaf node with a particular value or keep it as is. The only metric currently supported for regression trees is “variance.”

**Training Parameters**

In addition to hyperparameters, classification and regression trees also share the same training parameters. See [“Training Parameters”](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#training-parameters-a) for these parameters.

**Example**

Here’s a short example of using a decision tree regressor:

*// in Scala*

**import** **org.apache.spark.ml.regression.DecisionTreeRegressor**

**val** dtr **=** **new** **DecisionTreeRegressor**()

println(dtr.explainParams())

**val** dtrModel **=** dtr.fit(df)

*# in Python*

**from** **pyspark.ml.regression** **import** DecisionTreeRegressor

dtr = DecisionTreeRegressor()

**print** dtr.explainParams()

dtrModel = dtr.fit(df)

**Random Forests and Gradient-Boosted Trees**

The random forest and gradient-boosted tree models can be applied to both classification and regression. As a review, these both follow the same basic concept as the decision tree, except rather than training one tree, many trees are trained to perform a regression. In the random forest model, many de-correlated trees are trained and then averaged. With gradient-boosted trees, each tree makes a weighted prediction (such that some trees have more predictive power for some classes over others). Random forest and gradient-boosted tree regression have the same model hyperparameters and training parameters as the corresponding classification models, except for the purity measure (as is the case with DecisionTreeRegressor).

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

**Model Hyperparameters**

These models share many of the same parameters as we saw in the previous chapter as well as for regression decision trees. Refer back to [“Model Hyperparameters”](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#model-hyperparameters-a) for a thorough explanation of these parameters. As for a single regression tree, however, the only impurity metric currently supported is variance.

**Training Parameters**

These models support the same checkpointInterval parameter as classification trees, as described in [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification).

**Example**

Here’s a small example of how to use these two models to perform a regression:

*// in Scala*

**import** **org.apache.spark.ml.regression.RandomForestRegressor**

**import** **org.apache.spark.ml.regression.GBTRegressor**

**val** rf **=** **new** **RandomForestRegressor**()

println(rf.explainParams())

**val** rfModel **=** rf.fit(df)

**val** gbt **=** **new** **GBTRegressor**()

println(gbt.explainParams())

**val** gbtModel **=** gbt.fit(df)

*# in Python*

**from** **pyspark.ml.regression** **import** RandomForestRegressor

**from** **pyspark.ml.regression** **import** GBTRegressor

rf = RandomForestRegressor()

**print** rf.explainParams()

rfModel = rf.fit(df)

gbt = GBTRegressor()

**print** gbt.explainParams()

gbtModel = gbt.fit(df)

**Advanced Methods**

The preceding methods are highly general methods for performing a regression. The models are by no means exhaustive, but do provide the essential regression types that many folks use. This next section will cover some of the more specialized regression models that Spark includes. We omit code examples simply because they follow the same patterns as the other algorithms.

**Survival Regression (Accelerated Failure Time)**

Statisticians use survival analysis to understand the survival rate of individuals, typically in controlled experiments. Spark implements the accelerated failure time model, which, rather than describing the actual survival time, models the log of the survival time. This variation of survival regression is implemented in Spark because the more well-known Cox Proportional Hazard’s model is semi-parametric and does not scale well to large datasets. By contrast, accelerated failure time does because each instance (row) contributes to the resulting model independently. Accelerated failure time does have different assumptions than the Cox survival model and therefore one is not necessarily a drop-in replacement for the other. Covering these differing assumptions is outside of the scope of this book. See [L. J. Wei’s paper](http://bit.ly/2rKxqcW) on accelerated failure time for more information.

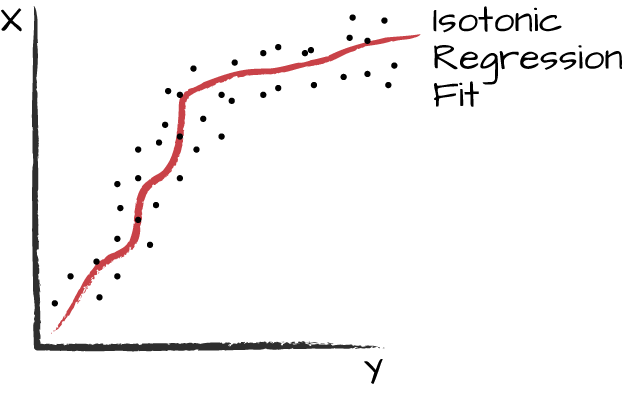
The requirement for input is quite similar to that of other regressions. We will tune coefficients according to feature values. However, there is one departure, and that is the introduction of a censor variable column. A test subject *censors* during a scientific study when that individual drops out of a study, since their state at the end of the experiment may be unknown. This is important because we cannot assume an outcome for an individual that censors (doesn’t report that state to the researchers) at some intermediate point in a study.

See more about survival regression with AFT [in the documentation](http://bit.ly/2nht2wD).

**Isotonic Regression**

Isotonic regression is another specialized regression model, with some unique requirements. Essentially, *isotonic regression* specifies a piecewise linear function that is always monotonically increasing. It cannot decrease. This means that if your data is going up and to the right in a given plot, this is an appropriate model. If it varies over the course of input values, then this is not appropriate.

The illustration of isotonic regression’s behavior in Figure 27-1 makes it much easier to understand.



*Figure 27-1. Isotonic regression line*

Notice how this gets a better fit than the simple linear regression. See [more about how to use this model in the Spark documentation](http://spark.apache.org/docs/latest/ml-classification-regression.html#isotonic-regression).

**Evaluators and Automating Model Tuning**

Regression has the same core model tuning functionality that we saw with classification. We can specify an evaluator, pick a metric to optimize for, and then train our pipeline to perform that parameter tuning on our part. The evaluator for regression, unsurprisingly, is called the RegressionEvaluator and allows us to optimize for a number of common regression success metrics. Just like the classification evaluator, RegressionEvaluator expects two columns, a column representing the prediction and another representing the true label. The supported metrics to optimize for are the root mean squared error (“rmse”), the mean squared error (“mse”), the r2 metric (“r2”), and the mean absolute error (“mae”).

To use RegressionEvaluator, we build up our pipeline, specify the parameters we would like to test, and then run it. Spark will automatically select the model that performs best and return this to us:

*// in Scala*

**import** **org.apache.spark.ml.evaluation.RegressionEvaluator**

**import** **org.apache.spark.ml.regression.GeneralizedLinearRegression**

**import** **org.apache.spark.ml.Pipeline**

**import** **org.apache.spark.ml.tuning.**{**CrossValidator**, **ParamGridBuilder**}

**val** glr **=** **new** **GeneralizedLinearRegression**()

.setFamily("gaussian")

.setLink("identity")

**val** pipeline **=** **new** **Pipeline**().setStages(**Array**(glr))

**val** params **=** **new** **ParamGridBuilder**().addGrid(glr.regParam, **Array**(0, 0.5, 1))

.build()

**val** evaluator **=** **new** **RegressionEvaluator**()

.setMetricName("rmse")

.setPredictionCol("prediction")

.setLabelCol("label")

**val** cv **=** **new** **CrossValidator**()

.setEstimator(pipeline)

.setEvaluator(evaluator)

.setEstimatorParamMaps(params)

.setNumFolds(2) *// should always be 3 or more but this dataset is small*

**val** model **=** cv.fit(df)

*# in Python*

**from** **pyspark.ml.evaluation** **import** RegressionEvaluator

**from** **pyspark.ml.regression** **import** GeneralizedLinearRegression

**from** **pyspark.ml** **import** Pipeline

**from** **pyspark.ml.tuning** **import** CrossValidator, ParamGridBuilder

glr = GeneralizedLinearRegression().setFamily("gaussian").setLink("identity")

pipeline = Pipeline().setStages([glr])

params = ParamGridBuilder().addGrid(glr.regParam, [0, 0.5, 1]).build()

evaluator = RegressionEvaluator()\

.setMetricName("rmse")\

.setPredictionCol("prediction")\

.setLabelCol("label")

cv = CrossValidator()\

.setEstimator(pipeline)\

.setEvaluator(evaluator)\

.setEstimatorParamMaps(params)\

.setNumFolds(2) *# should always be 3 or more but this dataset is small*

model = cv.fit(df)

**Metrics**

Evaluators allow us to evaluate and fit a model according to one specific metric, but we can also access a number of regression metrics via the RegressionMetrics object. As for the classification metrics in the previous chapter, RegressionMetrics operates on RDDs of (prediction, label) pairs. For instance, let’s see how we can inspect the results of the previously trained model.

*// in Scala*

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

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

.select("prediction", "label")

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

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

println(s"MSE = ${metrics.meanSquaredError}")

println(s"RMSE = ${metrics.rootMeanSquaredError}")

println(s"R-squared = ${metrics.r2}")

println(s"MAE = ${metrics.meanAbsoluteError}")

println(s"Explained variance = ${metrics.explainedVariance}")

*# in Python*

**from** **pyspark.mllib.evaluation** **import** RegressionMetrics

out = model.transform(df)\

.select("prediction", "label").rdd.map(**lambda** x: (float(x[0]), float(x[1])))

metrics = RegressionMetrics(out)

**print** "MSE: " + str(metrics.meanSquaredError)

**print** "RMSE: " + str(metrics.rootMeanSquaredError)

**print** "R-squared: " + str(metrics.r2)

**print** "MAE: " + str(metrics.meanAbsoluteError)

**print** "Explained variance: " + str(metrics.explainedVariance)

Consult [the Spark documentation](http://bit.ly/2rFTbef) for the latest methods.

**Conclusion**

In this chapter, we covered the basics of regression in Spark, including how we train models and how we measure success. In the next chapter, we’ll take a look at recommendation engines, one of the more popular applications of MLlib.