CSC 735 – Regression

Introduction

A logical extension of classification:

- Yields a continuous variable from a set of features
- Optimize some metric of error

Use Cases

- Movie Viewership
  - Given info for a certain movie, such as how many people have watched/shared the trailer, predict how many people are likely to watch it
- Company Revenue
  - Given growth trajectory, the market, and the time of year, predict a company's future revenue
- Crop Yield
  - Given info about the area in which a crop is grown and a recent history of weather conditions, predict total crop yield for a certain plot of land

The Data Set

```
1  val df = spark.read.load("/databricks-datasets/definitive-guide/data/regression")
2
3  display(df)
4
5
```

▸ (4) Spark Jobs

▸ df: org.apache.spark.sql.DataFrame = [features: udt, label: double]

features	label
▸ [1,3,[],[3,10.1,3]]	2
▸ [1,3,[],[2,1.1,1]]	1
▸ [1,3,[],[1,0.1,-1]]	0
▸ [1,3,[],[1,0.1,-1]]	0
▸ [1,3,[],[2,4.1,1]]	2

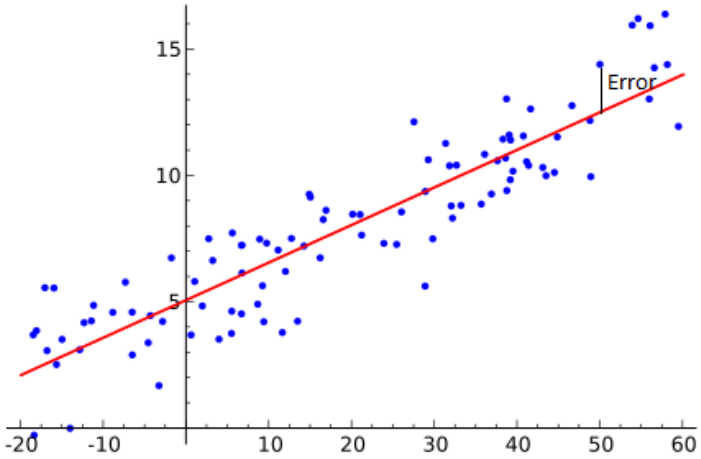
Command took 2.51 seconds -- by aub8665@live.missouristate.edu at 11/14/2018, 9:07:09 PM on My Cluster

Linear Regression

Overview

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

Assumes that a linear combination of your input features yields a prediction within a Gaussian amount of error of the true value.



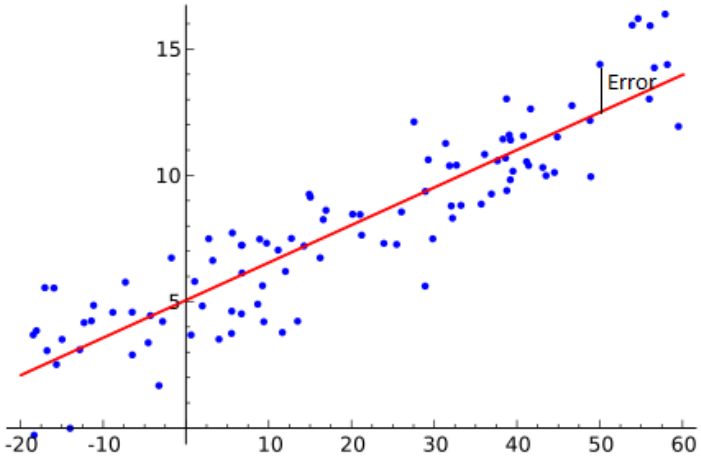
Linear Regression

Overview

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

Assumes that a linear combination of your input features yields a prediction within a Gaussian amount of error of the true value.

Does not always hold true, but does provide a simple model that is easy to interpret and hard to overfit.



Linear Regression

Hyperparameters

Params	Values	Description
family	Multinomial or binary	Multiclass vs binary classification
elasticNetParam	Floating point $\in$ [0, 1]	Specifies mix between L1, L2 regularization: Mix = r*L1 + (1-r)*L2
fitIntercept	Boolean	If true- fits the intercept of response variable
regParam	r >= 0	How much weight to five reg term
standardization	Boolean	If true- standardize inputs

Linear Regression

Training Parameters

Params	Values	Description
maxIter	Default = 100	Upper bound on number passes
tol	Default = 10^-6	Allows algorithm to stop before maxIter
weightCol	Default = None (weight column name)	Used to weight certain rows more than others **
regParam	r >= 0	How much weight to five reg term
standardization	Binary	If true- standardize inputs

Linear Regression

Example

```
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)
```

Linear Regression

Example

```
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)


val summary = lrModel.summary
summary.residuals.show()
```

Linear Regression

Example

```
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)


val summary = lrModel.summary
summary.residuals.show()
println(summary.objectiveHistory.toSeq.toDF.show())
```

Linear Regression

Example

```
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)


val summary = lrModel.summary
summary.residuals.show()
println(summary.objectiveHistory.toSeq.toDF.show())
println(summary.rootMeanSquaredError)
```

Linear Regression

Example

```
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)


val summary = lrModel.summary
summary.residuals.show()
println(summary.objectiveHistory.toSeq.toDF.show())
println(summary.rootMeanSquaredError)
println(summary.r2)
```

Generalized Linear Regression

Overview

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

A larger, more abstract family of algorithms -- simple linear regression included

Generalized Linear Regression

Overview

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

A larger, more abstract family of algorithms -- simple linear regression included

Spark has different implementations for simple vs the rest of generalized:

- Simple LR model optimized for very large sets of features
- Generalized LR supports more algorithms (e.g. error distr & link function)

Generalized Linear Regression

More control over what kind of regression model you use.

You may select an error distribution from a variety of families:

- Gaussian (linear)
- Binomial (logistic)
- Poisson (poisson)

Generalized Linear Regression

Overview

A **generalized linear model** (**GLM**) is a flexible generalization of ordinary linear regression.

The GLM generalizes linear regression by allowing the linear model to be related to the response variable via a ***link function*** and by allowing the magnitude of the variance of each measurement to be a function of its predicted value.

Generalized Linear Regression

Overview

The GLM consists of three elements:

1. A particular **distribution** for modeling
2. A **linear predictor**
3. A **link function** *η or g(μ)*

$$\mathrm{E}(\mathbf{Y} \mid \mathbf{X}) = \boldsymbol{\mu} = g^{-1}(\mathbf{X}\boldsymbol{\beta})$$

where E(**Y** | **X**) is the expected value of **Y** conditional on **X**; **Xβ** is the *linear predictor*, a linear combination of unknown parameters **β**; *g* is the link function.

the mean, ***μ***, of the distribution

the **independent** variables, **X**

Generalized Linear Regression

Overview

Can also set a link function which specifies a relationship between linear predictor (independent variables) and expected value of response (dependent) variable

η *or* $g(\mu)$

$\eta = g(E(Y_i))$

Where $E(Y_i)$ is expected value of response (dependent) variable for some X_i

Asterisks represent the typical link function for each error distr 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

Generalized Linear Regression

Hyperparameters

Params	**Values**	**Description**
family	Poisson, binomial, gamma, Gaussian, tweedie	Describes error distribution to be used

Generalized Linear Regression

Hyperparameters

Params	**Values**	**Description**
family	Poisson, binomial, gamma, Gaussian, tweedie	Describes error distribution to be used
link	Default = identity *Options vary based on error distr family	Name of function which gives relationship between linear predictor and and the expected value of the response variable

Generalized Linear Regression

Hyperparameters

Params	**Values**	**Description**
family	Poisson, binomial, gamma, Gaussian, tweedie	Describes error distribution to be used
link	Default = identify *Options vary based on error distr family	Name of function which gives relationship between linear predictor and and the expected value of the response variable
solver	irls	Solver algorithm used for optimization
variancePower	x ∈ 0, [1, inf)	Power in variance func of Tweedie distr
linkPower	integer	Index in power link function for Tweedie Family

Generalized Linear Regression

Parameters

Training parameters are the same as Linear and Logistic regression

One additional Prediction Parameter:

linkPredictionCol -- the name of a column which will hold output of the link function for each prediction

Generalized Linear Regression

Example

Generalized Linear Regression

```
1  import org.apache.spark.ml.regression.GeneralizedLinearRegression
2
3  val glr = new GeneralizedLinearRegression()
4    .setFamily("gaussian")
5    .setLink("identity")
6    .setMaxIter(10)
7    .setRegParam(0.3)
8    .setLinkPredictionCol("linkOut")
9
10 println(glr.explainParams())
11
12 val glrModel= glr.fit(df)
13
```

Simple & Gen. Linear Regression

Training Summary

Training summary provided by Spark can help us decide if our model is a good fit for the data (for both simple and generalized linear regression).

Includes many metrics for analysis:

- R squared
- Residuals
- Objective History
- Full list- org.apache.spark.ml.regression.LinearRegressionSummary, or inspect summary object.

Simple & Gen. Linear Regression

Training Summary

Training Summary

```
1  val lrModel = lr.fit(df)
2  val summary = lrModel.summary
3
4  summary.residuals.show
5  println(summary.objectiveHistory.toSeq.toDF.show)
6  println(summary.rootMeanSquaredError)
7  println(summary.r2)
```

```
RMSE
0.47308424392175985

R squared
0.720239122691221
```

```
+--------------------+
|           residuals|
+--------------------+
| 0.12805046585610147|
| -0.1446826926157201|
|-0.41903832622420606|
|-0.41903832622420606|
|  0.8547088792080306|
+--------------------+
```

```
objective History:
+--------------------+
|               value|
+--------------------+
|  0.5000000000000001|
| 0.43152958103627864|
|   0.313233593388102|
|   0.312256926665541|
| 0.30915060819830303|
| 0.30915058933480266|
+--------------------+
```

Decision Trees

Overview

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

Similar to DT for classification -- output a single number at a leaf node as opposed to a label.

Creates a tree to predict numerical outputs

Useful -- can predict nonlinear functions

Significant risk of overfitting

Decision Trees

Hyperparameters

Params	Value	Description
impurity	**variance**	**The metric for whether the model should split at a particular leaf node**
maxDepth	**Default = 5**	**Can specify maxDepth to avoid overfitting**
maxBins	**Default = 32**	**How many bins to be created from continuous features**
minInfoGain	**Default = 0**	**Min info gain that can be used for a split**
minInstancePerNode	**Default = 1**	**Min number samples that end up in a given leaf node**

Decision Trees

Training Parameters

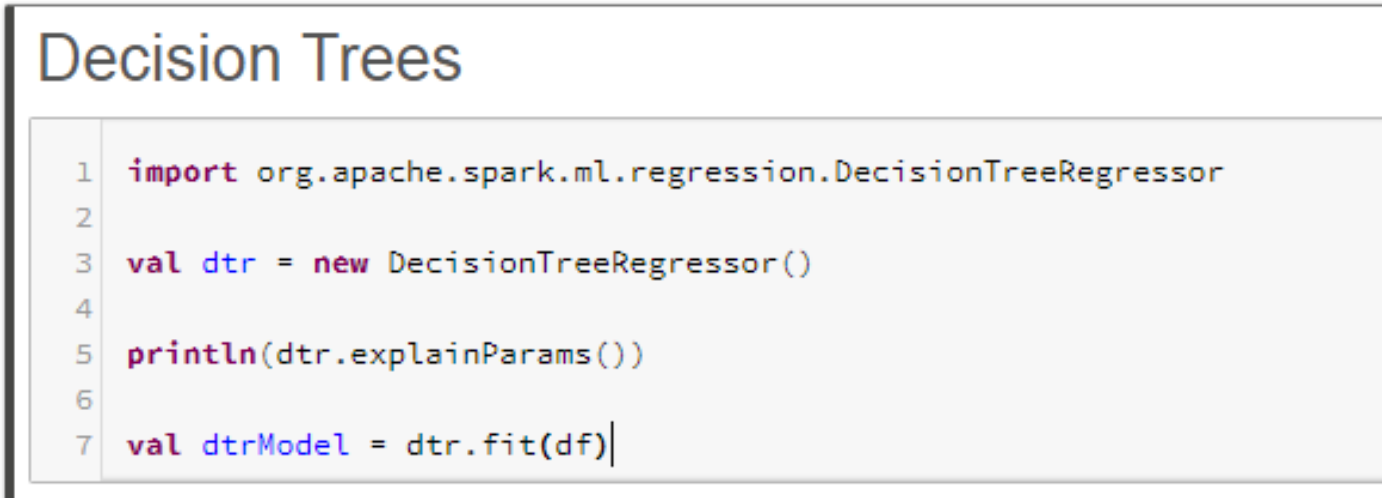
checkpointInterval

A way to save the model's work while training so that if nodes in the cluster crash, you don't lose your work.

If set to 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 useNodeIdCache=true.

Decision Trees

Example



Random Forest

Overview

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

An extension of Decision Trees that makes a prediction using an additive model by combining decisions from a sequence of base models (an average)

Random Forest

Overview

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

An extension of Decision Trees that makes a prediction using an additive model by combining decisions from a sequence of base models (an average).

This broad technique of using multiple models to obtain better predictive performance is called **model ensembling**

Very good at handling tabular data with numerical features, or categorical features with fewer than hundreds of categories.

Unlike linear models, random forests are able to capture non-linear interaction between the features and the target.

Gradient-Boosted Trees

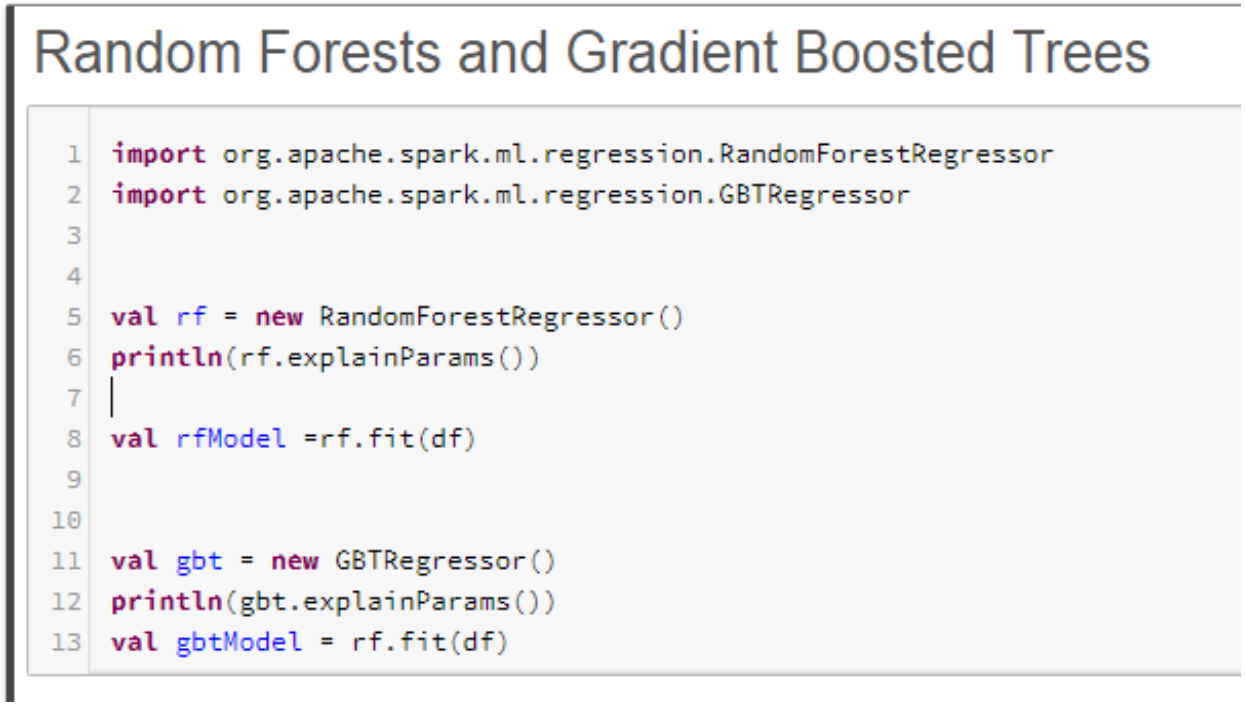
Overview

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

The gradient-boosted trees technique makes a **weighted** prediction by computing a sequence of very simple trees that each are formed from the residuals of the preceding tree

Random Forests and GBTs

Example



Random Forests & GBTs

Hyperparameters

Model	Parameter	Value	Description
Random Forest	numTrees	Int	Total number trees to train
	featureSubsetStrategy	auto, all, sqrt, log2, n	How many features should be considered for splits
Gradient-Boosted Trees	lossType	logistic	The loss func for gbt to minimize
	maxIter	Default = 100	Total num iterations before stopping
	stepSize	Default = 0.1 x ∈[0,1]	The learning rate; larger step size => larger jump between iterations

Advanced Methods - Survival Regression

import org.apache.spark.ml.regression.AFTSurvivalRegression

Survival analysis - understanding the survival of individuals

Advanced Methods - Survival Regression

import org.apache.spark.ml.regression.AFTSurvivalRegression

Survival analysis - understanding the survival of individuals

Survival Regression (Accelerated Failure Time) model - used by spark and stores the log of the survival time as oppose to the exact survival time

- This variation was chosen since it is scalable to large datasets. Each row contributes to the model independently and therefore can be parallelized

Advanced Methods - Survival Regression

Data frame should contain the following columns:

- featuresCol: The feature vectors.
- labelCol: The survival times (time to event).
- censorCol: An indicator that specifies whether the event of interest has occurred or not. A test subject censors during a scientific study when that individual drops out of a study.

Specify Quantile Probabilities: When you want to make predictions or estimate survival times at specific quantiles of the survival time distribution, you specify quantile probabilities as a list of values.

Advanced Methods - Survival Regression

Example

```scala
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.ml.regression.AFTSurvivalRegression

val training = spark.createDataFrame(Seq(
  (1.218, 1.0, Vectors.dense(1.560, -0.605)),
  (2.949, 0.0, Vectors.dense(0.346, 2.158)),
  (3.627, 0.0, Vectors.dense(1.380, 0.231)),
  (0.273, 1.0, Vectors.dense(0.520, 1.151)),
  (4.199, 0.0, Vectors.dense(0.795, -0.226))
)).toDF("label", "censor", "features")
val quantileProbabilities = Array(0.1, 0.2)
val aft = new AFTSurvivalRegression()
  .setQuantileProbabilities(quantileProbabilities)
  .setQuantilesCol("quantiles")

val model = aft.fit(training)

// Print the coefficients, intercept and scale parameter for AFT survival regression
println(s"Coefficients: ${model.coefficients}")
println(s"Intercept: ${model.intercept}")
println(s"Scale: ${model.scale}")
model.transform(training).show(false)
```

▸ (19) Spark Jobs

▸ training: org.apache.spark.sql.DataFrame = [label: double, censor: double ... 1 more fields]

```
Coefficients: [-0.4963111466650667,0.19844437699934134]
Intercept: 2.6380946151040034
Scale: 1.547234557436469
+-----+------+--------------+-----------------+----------------------------------------+
|label|censor|features      |prediction       |quantiles                               |
+-----+------+--------------+-----------------+----------------------------------------+
|1.218|1.0   |[1.56,-0.605] |5.718979487634969 |[0.17586199382570866,0.5615982157419642]|
|2.949|0.0   |[0.346,2.158] |18.076521181495636|[0.5558636926891851,1.7750967745729636] |
|3.627|0.0   |[1.38,0.231]  |7.381861804239106 |[0.22699660627322626,0.7248916396846232]|
|0.273|1.0   |[0.52,1.151]  |13.57761250142538 |[0.41751959611958356,1.3333083238578982]|
```

Advanced Methods - Isotonic Regression

import org.apache.spark.ml.regression.IsotonicRegression

Isotonic Regression

Specifies a piecewise linear function that is monotonically increasing, and cannot decrease (works best for data that is going up and to the right)

Advanced Methods - Isotonic Regression

import org.apache.spark.ml.regression.IsotonicRegression

Isotonic Regression

Specifies a **piecewise** linear function that is monotonically increasing, and cannot decrease (works best for data that is going up and to the right)

Gets a better fit than simple linear regression due to it's flexibility since it is defined piecewise

Includes information on the boundaries and predictions of the model

Advanced Methods - Isotonic Regression

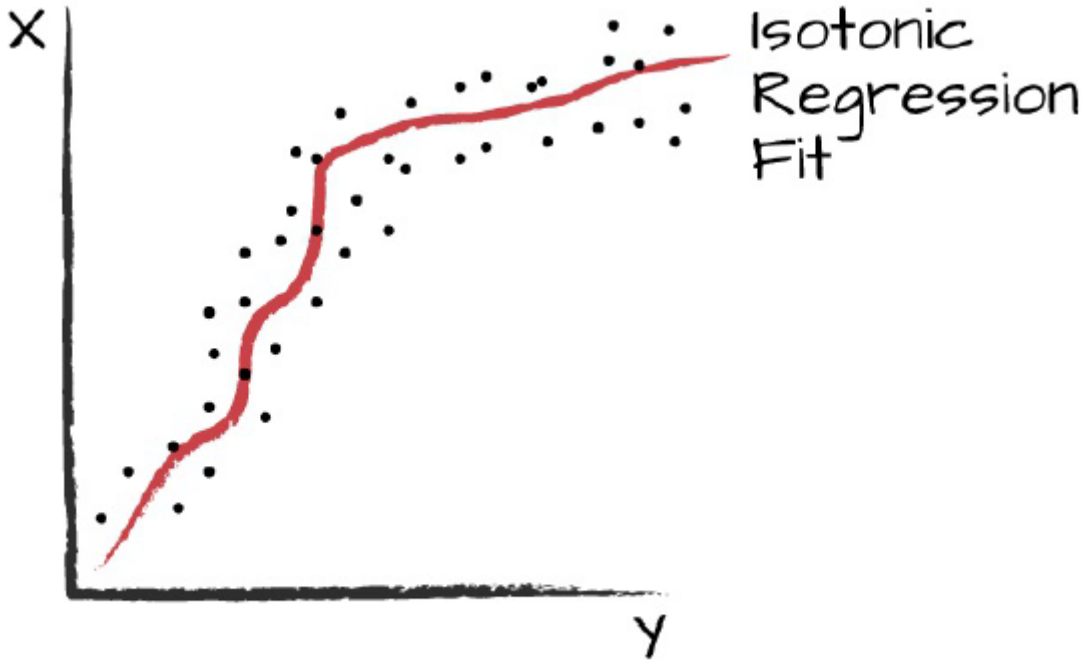


Figure 27-1. Isotonic regression line

Model Scalability

Model	Number Features	Training Examples
Linear Regression	1 to 10 million	No limit
Generalized Linear Regression	4096	No limit
Isotonic Regression	N/A	Millions
Decision Trees	1,000s	No limit
Random Forest	10,000s	No limit
Gradient-Boosted Trees	1000s	No limit
Survival Regression	1 to 10 million	No limit

Evaluators and Automating Model Tuning

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

Same core model tuning as classification

Steps:

1. Specify an Evaluator
2. Pick a metric to optimize
3. Train pipeline to perform parameter tuning

Evaluators and Automating Model Tuning

Step 1 Build the RegressionEvaluator object and specify two columns (prediction, label)

Step 2 There are four supported metrics to optimize for:

Root Mean Squared Error, Mean Squared Error, R2, and Mean Absolute Error

Step 3 Create a Pipeline, ParamGridBuilder, then pass those two and the RegressionEvaluator to build a CrossValidator object

Evaluators and Automating Model Tuning

```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)
```

Metrics

org.apache.spark.mllib.evaluation.RegressionMetrics

First map the model into this format: (prediction, label) pairs

Then create a RegressionMetrics object with that formatted model as a parameter

The object contains methods that will return statistical values of the data set including

- Mean Squared Error
- Root Mean Squared Error
- R Squared
- Mean Absolute Error
- Explained Variance

Metrics

```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}")

```

▸ (2) Spark Jobs

```
MSE = 0.21432448963739983
RMSE = 0.4629519301584127
R-squared = 0.7320943879532502
MAE = 0.34692281471051273
Explained variance = 0.32923447021104896
```

Conclusion

Regression can be thought of as an abstraction of classification -- where we wish our model to predict a continuous variable according to a linear combination of input features (the coefficients of which are chosen by regression to minimize error).

Conclusion

Simple and Generalized Linear Regression are useful when we have some idea of data distribution, but are interpretable and hard to overfit

Trees are useful in predicting nonlinear data, but at a higher risk of overfitting

Spark also includes more context-specific regression models, such as Survival & Isotonic Regression

References

https://turi.com/learn/userguide/supervised-learning/random_forest_regression.html

http://spark.apache.org/docs/latest/ml-classification-regression.html#isotonic-regression

http://enhancedatascience.com/2017/07/04/machine-learning-explained-regularization/

https://onlinecourses.science.psu.edu/stat504/node/216/

Generalized Linear Regression

Overview

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

A larger, more abstract family of algorithms -- simple linear regression included