Spark Program

Chapter 7: Regression Analysis

Chapter Objectives

In this chapter, we will:

- Introduce Linear Regression
- Explore data preparation
- Train and test regression model

Chapter Concepts

Regression Analysis

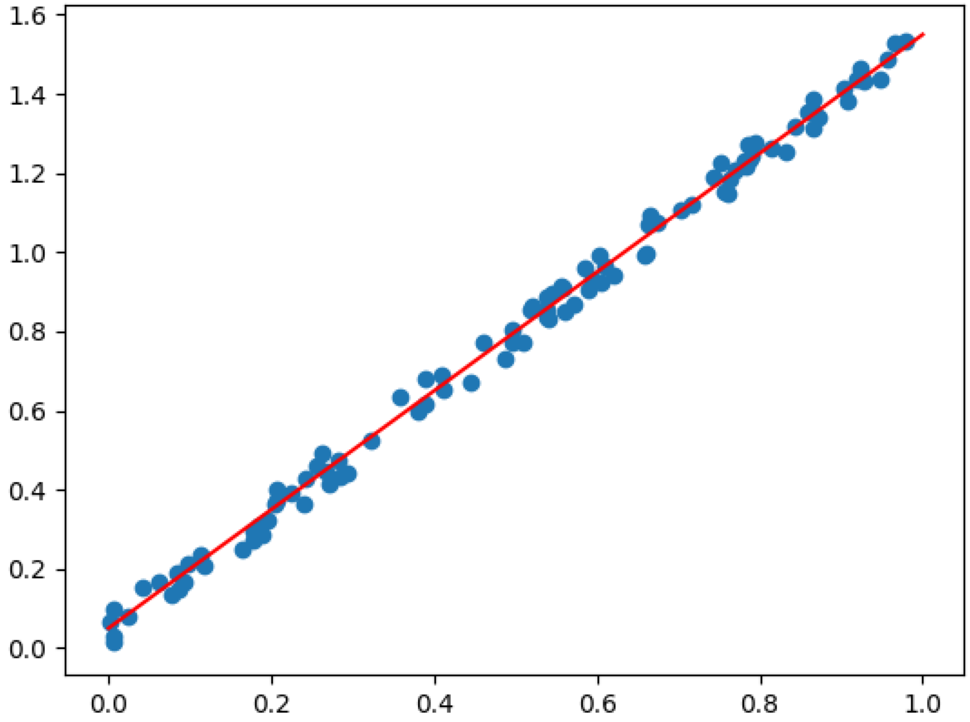
Data Preparation

Algorithms

Chapter Summary

Linear Regression

- Given a collection of X, Y points, you could easily see there is a pattern
- If you remember enough algebra, you could describe the pattern of dots as roughly following the red line, which could be described with the formula y = 1.5x + .01



Linear Regression (continued)

- The idea is that the line that best describes the pattern of dots is the one that has the least distances of the dots from the line
- The formula that describes the line could then be used to predict a value that we have not observed
  - The better the line and formula are at describing that pattern of dots, the more accurate that prediction should be
- Extrapolate this idea onto more than just two axes and instead try to find a line that goes through many different dimensions and you have the idea of multiple linear regression
  - $y = \alpha + \beta 1 x1 + \beta 2 x2 + \ldots + \beta i xi + \varepsilon$
- Has many use cases
  - Predicting a stock or commodity price
  - Predicting election results
  - Predicting crime rate

Linear Regression (continued)

- Is a supervised model that requires training from a known set of data and testing to see how good it is at predicting before using it for real predictions
- Only works with numeric values
  - Categorical data needs to be dummy encoded
- Does not deal well with missing data, so must be fixed by removing or replacing with central tendency
- There are many algorithms to do this, each with its own pros and cons

Chapter Concepts

Regression Analysis

Data Preparation

Algorithms

Chapter Summary

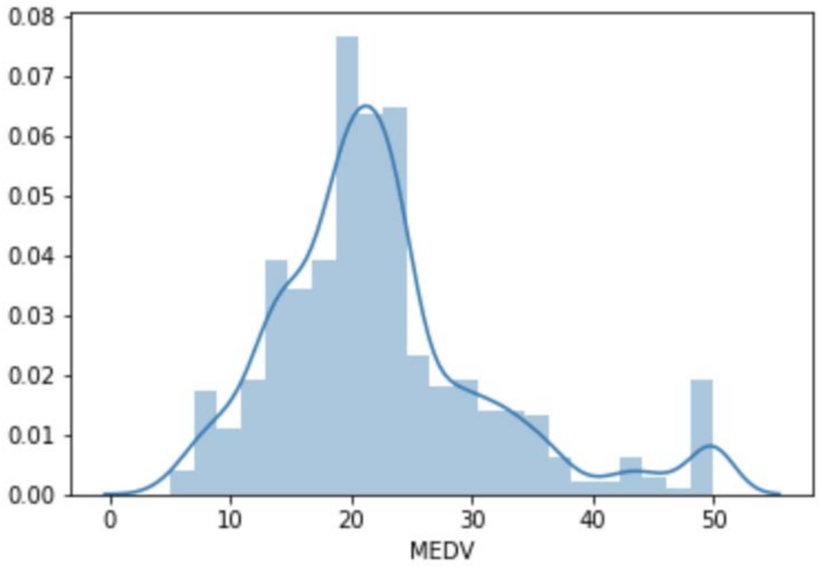
Dataset

- For our examples, let's use a public data set of housing data

```
import pandas as pd
import seaborn as sns
sns.distplot(df.toPandas()['MEDV'])
```

- Plotting the distribution of Prices shows that they are normally distributed, except for some outliers, so let's try comparing the model with them and then later filter out

```
dfRaw = dfRaw.where('MEDV < 48')
```



Convert Categorical Features

- Categorical data cannot stay as string, so it must be converted to a numeric format and then into a vector format
- `pyspark` has a class which will transform a column into indexed numbers for each unique string value

```
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCol = col, outputCol = col+'_Index')
x = indexer.fit(df).transform(df).select(col, col+'_Index').distinct()
display(x.orderBy(col))
display(x.orderBy(col+'_Index'))
```

- For convenience, use this helper function we made:

  `display(pyh.StringIndexEncode(df, ['TOWN', 'TRACT']))`

	TOWN	TOWN_Index
0	Arlington	23.0
1	Ashland	64.0
2	Bedford	61.0
3	Belmont	17.0
4	Beverly	27.0
5	Boston Allston-Brighton	18.0
6	Boston Back Bay	32.0
7	Boston Beacon Hill	54.0
8	Boston Charlestown	31.0
9	Boston Dorchester	12.0

	TOWN	TOWN_Index
0	Cambridge	0.0
1	Boston Savin Hill	1.0
2	Lynn	2.0
3	Boston Roxbury	3.0
4	Newton	4.0
5	Somerville	5.0
6	Boston South Boston	6.0
7	Quincy	7.0
8	Boston East Boston	8.0
9	Brookline	9.0

One-Hot Encoding

- Numerical indexes are good for some algorithms such as Naive Bayes and Decision Trees, but ones that use distance calculations would get distorted
- Need to re-encode this as One-Hot Encoding which creates a separate column for each unique value and fills the columns with zeros and ones
- In Spark, this column needs to be a single Vector column, unlike Pandas which makes a lot of unique columns
- Sparse vectors are hard to interpret visually, but they are not meant for human eyes
- Must first re-encode data with `StringIndexer`

```
from pyspark.ml.feature import OneHotEncoderEstimator
encoder = OneHotEncoderEstimator(inputCols=[col + '_Index'],
outputCols=[col+'_Vector'])
display(encoder.fit(df).transform(df))
```

One-Hot Encoding (continued)

- `SparseVector` version

	TOWN	TOWN_Index	TOWN_Vector
0	Cambridge	0.0	(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
1	Boston Savin Hill	1.0	(0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
2	Lynn	2.0	(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
3	Boston Roxbury	3.0	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
4	Newton	4.0	(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...
5	Somerville	5.0	(0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, ...
6	Boston South Boston	6.0	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, ...
7	Quincy	7.0	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, ...
8	Boston East Boston	8.0	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, ...
9	Brookline	9.0	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...

- Helper function to call `StringIndexer`, then `OneHotEncoder`

```
display(pyh.OneHotEncode(df, ['TOWN', 'TRACT']))
```

Putting It All Together

- You have to `OneHotEncode` all categorical data, then assemble all the features into one vector and the target variable into another
- Spark provides the VectorAssembler class to do this
- Our helper function makes the whole process more convenient
- Just pass in a DataFrame, list of categorical, numeric, and target columns and it returns a DataFrame with the two columns needed for machine learning algorithms

```
dfML = pyh.AssembleFeatures(df, categorical_features,
numeric_features, target_label = 'target', target_is_categorical
= False))
```

Explore Numerical Features

- Generally, you want to take a look at the numerical features and get standard measurements like min, max, mean, std
  - DataFrames have a `describe` method which makes that easy
- The provided helper functions make that easier

```
numeric_features = ['totalvolume','PLU4046', 'PLU4225',
'PLU4770', 'smallbags', 'largebags', 'xlargebags']
display(df.select(numeric)describe())
```

	summary	CRIM	ZN	INDUS	CHAS	NOX
0	count	487	487	487	487	487
1	mean	3.663863696098563	10.944558521560575	11.155215605749499	0.059548254620123205	0.5544979466119098
2	stddev	8.745039991517844	22.587028902677194	6.820162970724796	0.23689130625554344	0.11678383814441988
3	min	0.00632	0.0	0.74	0	0.385
4	max	88.9762	100.0	27.74	1	0.871

Prepare the Data

- Data must be in a DataFrame of two vectorized objects
  - Features will contain all the independent variables
  - Target will be the dependent variable we are trying to predict
- The provided helper functions make that easier
- Then split the data into a train and test set with the `randomSplit` function

```
import pyspark_helpers as pyh

numeric_features = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', \
             'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO']
categorical_features = ['TOWN', 'TRACT']
target_label = 'MEDV'
df = dfRaw.select(categorical_features + numeric_features +
[target_label])
dfML = pyh.MakeMLDataFrame(df, categorical_features, \
       numeric_features, target_label, False)

train, test = dfML.randomSplit([.7,.3], seed = 1000)
```

Chapter Concepts

Regression Analysis

Data Preparation

Algorithms

Chapter Summary

Run the Model

- Create and instance of the regression class
- There are several to choose from
  - `LinearRegression`
  - `GeneralizedLinearRegression`
  - `DecisionTreeRegressor`
  - `RandomForestRegressor`
  - `GBTRegressor`
  - `AFTSurvivalRegression`
  - `IsotonicRegression`

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol = 'features', labelCol='target', \
     maxIter=10, regParam=0.3, elasticNetParam=0.8)
lrModel = lr.fit(train)
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
print("Root Mean Squared Error: {}\nR Squared (R2) {}" \
  .format(lrModel.summary.rootMeanSquaredError, lrModel.summary.r2))
```

Run the Test

```
lrPredictions = lrModel.transform(test)
display(lrPredictions.select("prediction","target","features"), 30)
from pyspark.ml.evaluation import RegressionEvaluator
lrEvaluator = RegressionEvaluator(predictionCol="prediction", \
             labelCol="target",metricName="r2")
testResult = lrModel.evaluate(test)
print("Root Mean Squared Error on Test set: {}" \
             .format(testResult.rootMeanSquaredError))
```

```
Coefficients: [-0.09835471749252538,0.005332754299371162,-0.10096929421151506,0.0,-5.571116
7,-0.04213205450570569 5,-0.42612539816791184,0.0,-0.004595425300618804,-0.5955258325154016]
Intercept: 9.933834380915346
Root Mean Squared Error: 4.054438240179903
R Squared (R2) 0.6990028588000552
```

25	5.084273	13.8	[18.4982, 0.0, 18.1, 0.0, 0.668, 4.138, 100.0,...
26	18.237093	14.0	[0.2909, 0.0, 21.89, 0.0, 0.624, 6.174, 93.6, ...
27	15.081058	14.3	[0.88125, 0.0, 21.89, 0.0, 0.624, 5.637, 94.7,...
28	18.499367	14.3	[5.58107, 0.0, 18.1, 0.0, 0.713, 6.436, 87.9, ...
29	16.793500	14.8	[5.66637, 0.0, 18.1, 0.0, 0.74, 6.219, 100.0, ...

```
Root Mean Squared Error on Test set: 4.370645041749265
```

Chapter Concepts

Regression Analysis

Data Preparation

Algorithms

Chapter Summary

Next Steps

- Regression has a lot more complexity to it once you master the basics
- Use Pipelines to create more complex transformations in a more efficient way
- Some subjects to explore in this area:
  - Under- and over-fitting a model
  - Correlation between the independent variables

Chapter Summary

In this chapter, we have:

- Introduced Linear Regression
- Explored data preparation
- Trained and test regression model