MLlib: Classification with Logistic Regression

From Distributed Data Mining Winter Semester 2015-2016

Contents

MLlib: Classification with Logistic Regression

we will use Spark's machine learning library MLlib to build a Logistic Regression classifier for network attack detection. We will use the KDD Cup 1999 (http://kdd.ics.uci.edu/databases/kddcup99/) complete datasets in order to test Spark capabilities with large datasets.Simple mathematical explanation of logistic regression can be found here (https://spark.apache.org/docs/latest/mllib-linear-methods.html#logistic-regression/) . There are two ways for model selection: by using **Corelation matrix** and by **hypothesis testing**

Getting the data and creating the RDD

We will use urllib to get the data directly and use it for implementation.

```
import urllib
f = urllib.urlretrieve ("http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data.gz", "kddcup.data.gz")
data_file = "./kddcup.data.gz"
raw_data = sc.textFile(data_file)
print "Train data size is {}".format(raw_data.count())
```

We will also use 10% of the sample data for testing and load it in separate RDD.

```
ft = urllib.urlretrieve("http://kdd.ics.uci.edu/databases/kddcup99/corrected.gz", "corrected.gz")
test_data_file = "./corrected.gz"
test_raw_data = sc.textFile(test_data_file)
print "Test data size is {}".format(test_raw_data.count())
```

Preparing the training data

For this data we are interested in detecting the networking attacks, hence we label each network interaction as non attack (i.e. 'normal' tag) or attack (i.e. anything else but 'normal').

```
from pyspark.mllib.regression import LabeledPoint
from numpy import array
def parse_interaction(line):
   line_split = line.split(",")
   # leave_out = [1,2,3,41]
   clean_line_split = line_split[0:1]+line_split[4:41]
   attack = 1.0
   if line_split[41]=='normal.':
       attack = 0.0
   return LabeledPoint(attack, array([float(x) for x in clean_line_split]))
```

```
training_data = raw_data.map(parse_interaction)
```

Similarly we prepare the test data:

```
test_data = test_raw_data.map(parse_interaction)
```

Detecting network attacks using Logistic Regression

We use Logistic Regression to predict the binary response. Spark has two algorithms to implement logistic regression: mini-batch gradient descent and L-BFGS. L-BFGS is recommended over mini-batch gradient descent for faster convergence.

Training a classifier

```
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from time import time
```

```
# Build the model
t0 = time()
logit_model = LogisticRegressionWithLBFGS.train(training_data)
tt = time() - t0
```

```
print "Classifier trained in {} seconds".format(round(tt,3))
```

Evaluating the model on new data

In order to measure the classification error on our test data, we use map on the test_data RDD and the model to predict each test point class.

```
labels_and_preds = test_data.map(lambda p: (p.label, logit_model.predict(p.features)))
```

Classification results are returned in pars, with the actual test label and the predicted one. This is used to calculate the classification error by using filter and count as follows.

```
t0 = time()
test_accuracy = labels_and_preds.filter(lambda (v, p): v == p).count() / float(test_data.count())
tt = time() - t0
```

```
print "Prediction made in {} seconds. Test accuracy is {}".format(round(tt,3), round(test_accuracy,4))
```

Model selection

Model or feature selection helps us building more interpretable and efficient models (or a classifier in this case). For illustrative purposes, we will follow two different approaches, hypothesis testing.

Using hypothesis testing

Hypothesis testing is a powerful tool in statistical inference and learning to determine whether a result is statistically significant. We use RDD[LabeledPoint] to enable feature selection via chi-squared independence tests.We want to perform some sort of feature selection.Features need to be categorical. Real-valued features will be treated as categorical in each of its different values.we will consider just features that either take boolean values or just a few different numeric values in our dataset. We could overcome this limitation by defining a more complex parse_interaction function that categorises each feature properly.

```
feature_names = ["land","wrong_fragment",
            "urgent","hot","num_failed_logins","logged_in","num_compromised",
            "root_shell","su_attempted","num_root","num_file_creations",
            "num_shells","num_access_files","num_outbound_cmds",
            "is_hot_login","is_guest_login","count","srv_count","serror_rate",
            "srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate",
            "diff_srv_rate","srv_diff_host_rate","dst_host_count","dst_host_srv_count",
            "dst_host_same_srv_rate","dst_host_diff_srv_rate","dst_host_same_src_port_rate",
            "dst_host_srv_diff_host_rate","dst_host_serror_rate","dst_host_srv_serror_rate",
            "dst_host_rerror_rate","dst_host_srv_rerror_rate"]
```

```
def parse_interaction_categorical(line):
   line_split = line.split(",")
   clean_line_split = line_split[6:41]
   attack = 1.0
   if line_split[41]=='normal.':
       attack = 0.0
   return LabeledPoint(attack, array([float(x) for x in clean_line_split]))
```

```
training_data_categorical = raw_data.map(parse_interaction_categorical)
```

```
from pyspark.mllib.stat import Statistics
```

```
chi = Statistics.chiSqTest(training_data_categorical)
```

Now we can check the resulting values after putting them into a Pandas data frame.

```
import pandas as pd
pd.set_option('display.max_colwidth', 30)
records = [(result.statistic, result.pValue) for result in chi]
chi_df = pd.DataFrame(data=records, index= feature_names, columns=["Statistic","p-value"])
```

```
chi_df
```