Project2 Report——Logistic Regression with Jackknife

Siwei Wang

N10385479

1 Description of Logistic Regression

Logistic regression is a regression model where the dependent variables are categorical. In the titanic example here, the outcome takes 2 values, 1 and 0, which represents survived and not survived. Such model with only 2 types of output is also called a binary logistic regression model. Logistic regression model is used in many fields such as machine learning and social science as well. In this example, it has 10 predictors and I choose 6 from them: Pclass, Sex, Age, SibSp, Parch and Fare. By using these predictors in the logistic regression model, it's possible to predict whether one is survived or not.

2 Mathematical Formula of the Algorithm

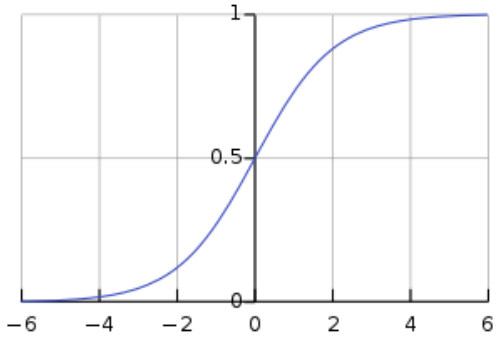
Logistic regression model uses the following probabilistic functions:

$$P(y = 1 \mid x) = \frac{1}{1 + e^{-z}}, \qquad P(y = 0 \mid x) = \frac{e^{-z}}{1 + e^{-z}}$$

The logistic function is

$$f(z) = \frac{1}{1 + e^{-z}}$$

And the corresponding plot of $f(z)$ is



Several important concepts are also used to describe how good the prediction is. Two of those concepts are confusion matrix and TPR or FPR.

Confusion matrix is a table often used to describe the performance of the classification model. It is able to visualize the result and is straightforward.

3 Description of Jackknife

The jackknife technique was developed by Maurice Quenouille in 1949. It is a resampling tool especially useful for variance and bias estimation. The jackknife estimator of a parameter is found by

systematically leaving out each observation from a dataset and calculating the estimate and then finding the average of these calculations. Given a sample of size n, the jackknife estimate is found by aggregating the estimates of each n-1 sized sub-sample. The ith and jth jackknife sample thus only have 2 different data: the ith and the jth sample.

The jackknife technique requires that the observations are independent of each other. The jackknife if not suitable for a time-series dataset. When the samples are not independent of each other, the result of the estimation will be more reliable than they actually are. It is also not appropriate to estimate some statistic values by using jackknife. For instance, the median of the following array [3, 20, 45, 85, 109, 151, 200]. The median after jackknife is [97, 97, 97, 77, 60, 60, 60] which is not close to the real result 85.

4 Description of the libsvm format

Libsvm is a format of dataset especially for machine learning which can be stored in a text file. The format of a libsvm data file is as follow:

<label 1> <index1.1>:<value1.1> <index1.2>:<value1.2>……

<label 2> <index2.1>:<value2.1> <index2.2>:<value2.2>……

……..

<label N> <indexN.1>:<valueN.1> <indexN.2>:<valueN.2>……

<label i> is the notation of the class, it can be 0, 1, or any other value. <value i.j> is the data of training or test part. In the titanic dataset, one example of the libsvm file is:

1 1:2 2:1 3:34 4:0 5:0 6:13

The first '1' notates the class, which represents 'survived' here. '1:2' means the Pclass of this person is 2. '3:34' means the age of this person is 34 and so on.

5 Two ways to measure the accuracy of Spark Java and R

I have done two ways to give measurement to decide which method is better.

The first one is to split the original dataset into a training data and a test data in advance and to do the algorithm in Spark Java and R. This method ensures that the training dataset and the test dataset are exactly same in both Spark Java and R.

The second way I try is to split the original dataset in a certain ratio randomly in both Spark Java and R, and run the algorithm repeatedly. Then I get the average of the metrics and make the comparison. The reason I try this is because the evaluation metrics is different if the training and test dataset are split in different ways.

I will show the result the of the second way first and then the first way.

6 Spark Java Logistic Regression Code and Corresponding Results

Spark Java needs a XML code part to download needed APIs.

General configuration code:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>spark-ml-second-app</groupId>
    <artifactId>spark-java-ml</artifactId>
    <version>1.0-SNAPSHOT</version>
```

Spark Java needs compiler version 1.8:

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Spark version 2.1.0 and Spark-milib are used:

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.apache.spark/spark-core_2.11 -->
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
        <version>2.1.0</version>
        <scope>provided</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.spark/spark-mllib_2.11 -->
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-mllib_2.11</artifactId>
        <version>2.1.0</version>
    </dependency>

</dependencies>

</project>
```

All the libraries are downloaded and need to be imported before use. These libraries provide a variety of functions including reading the file, loading the data into datasets, make logistic regression and predictions, getting properties of the predictions and so on:

```
import scala.Tuple2;

import org.apache.spark.api.java.*;
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics;
import org.apache.spark.mllib.classification.LogisticRegressionModel;
import org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS;
import org.apache.spark.mllib.evaluation.MulticlassMetrics;
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.util.MLUtils;
import org.apache.spark.mllib.linalg.Matrix;

import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;

import java.sql.Timestamp;
```

SparkConf is used to configure a new Spark application and set various Spark parameters as key-value parts:

```
SparkConf conf = new SparkConf().setAppName("LogisticRegression3");
SparkContext sc = new SparkContext(conf);
```

MLUtils.loadLibSVMFile() is used to Loads binary labeled data in the LIBSVM format into an RDD<LabeledPoint>, with number of features determined automatically and the default number of partitions:

```
String path = "titanic-ml.txt";
JavaRDD<LabeledPoint> data = MLUtils.loadLibSVMFile(sc, path).toJavaRDD();
```

In order to choose the ith jackknife sample and split the data into training and test part randomly, I used 2 timestamps to get a random number every time the program is executed:

```
Timestamp timestamp1 = new Timestamp(System.currentTimeMillis());
Timestamp timestamp2 = new Timestamp(System.currentTimeMillis());

JavaRDD<LabeledPoint>[] split1 = data.randomSplit(new double[]{713, 1}, timestamp1.getTime());
JavaRDD<LabeledPoint> data2 = split1[0].cache();
// Split the dataset into two parts 70% training and 30% test
JavaRDD<LabeledPoint>[] splits = data2.randomSplit(new double[]{0.7, 0.3}, timestamp2.getTime());
JavaRDD<LabeledPoint> training = splits[0].cache();
JavaRDD<LabeledPoint> test = splits[1];
```

LogisticRegressionWithLBFGS() is a method to rain a classification model for multinomial or binary logistic regression using limited-memory BFGS. Because the result is only survived or not survived, the number of the class is 2. Run the logistic regression on the training data and make prediction using the test data.

```
LogisticRegressionModel model = new LogisticRegressionWithLBFGS()
        .setNumClasses(2)
        .run(training.rdd());
JavaPairRDD<Object, Object> predictionAndLabels = test.mapToPair(p ->
        new Tuple2<>(model.predict(p.features()), p.label()));
```

In order to get the evaluation data of the prediction, Spark Java also provide methods to do so:

```
MulticlassMetrics metrics = new MulticlassMetrics(predictionAndLabels.rdd());

// Confusion matrix
Matrix confusion = metrics.confusionMatrix();
System.out.println("Confusion matrix: \n" + confusion);

// Accuracy
System.out.println("Accuracy = " + metrics.accuracy());

// True positive rate
System.out.format("True positive rate = %f\n", metrics.truePositiveRate(metrics.labels()[0]));

BinaryClassificationMetrics metrics1 = new BinaryClassificationMetrics(predictionAndLabels.rdd());

// AUPRC
System.out.println("Area under precision-recall curve = " + metrics1.areaUnderPR());

// AUROC
System.out.println("Area under ROC = " + metrics1.areaUnderROC());
```

Four metrics are measured: confusion matrix, accuracy, true positive rate and area under ROC. Experiments are repeated seven times and average value is used to present the overall evaluation.

Confusion matrix:

96.85714	20.28571
26.85714	55.42857

Accuracy: 0.764

True positive rate: 0.827

Area under ROC: 0.750

7 R Logistic Regression Code and Corresponding Results

Several libraries are also needed to be imported to do the logistic regression and get the evaluations of the regression:

```
library(caret)
library(ggplot2)
library(lattice)
library(ROCR)
library(AUC)
```

There are a lot of columns in the original dataset and not all the columns are needed:

```
training.data.raw <- read.csv('titanic-train.csv',header=T,na.strings=c(""))
data <- subset(training.data.raw,select=c(2,3,5,6,7,8,10))
```

Create a random number and delete the ith row in the dataset, remaining 713 rows:

```
newtimestamp <- Sys.time()
set.seed(as.numeric(newtimestamp))
data <- data[-c(as.numeric(newtimestamp)%%714)]
```

Split the data into 2 parts, training and test part:

```
tstidx <- sample(713, 200, replace = FALSE)
test <- data[tstidx,]
training <- data[-tstidx,]
```

Do the logistic regression and make the prediction. Since the result of the prediction is the probability of being 1, which is 'survived', one more procedure is needed to convert the probability into only one and zero. If the probability of survived is more than 0.5, it is converted to 1, and 0 otherwise:

```
model <- glm(Survived ~.,family=binomial(link='logit'),data=training)
summary(model)
model_predict <- predict(model,test[, 2:7],type='response')
model_predict1 <- ifelse(model_predict > 0.5,1,0)
```

Get the evaluation of the regression, accuracy, confusion matrix and area under ROC:

```
misClasificError <- mean(model_predict1 != test$Survived)
print(paste('Accuracy',1-misClasificError))
confusionMatrix(model_predict1,test[, 1])
pred <- prediction(model_predict, test$Survived)
auc <- performance(pred, measure = "auc")
auc <- auc@y.values[[1]]
```

Four metrics are measured: confusion matrix, accuracy, true positive rate and area under ROC. Experiments are repeated seven times and average value is used to present the overall evaluation.

Confusion matrix:

99.57143	24.14286
18.28571	58

Accuracy: 0.788

True positive rate: 0.806

Area under ROC: 0.848

8 Result Comparison between Spark Java and R

The accuracy of Spark Java and R are very close to each other. They are slightly different in the confusion matrix.

The true positive number and true negative number of Spark Java is smaller than that of R, which means that there are less survived and predicted survived, not survived and predicted not survived cases in Spark Java than R. However, the false positive number and the false negative number swap in these 2 methods. From the confusion matrix, 2 methods have similar performance. The AUROC is 0.750 in Spark Java and 0.848 in R. R is 10% higher than Spark Java. Using R is more precious.

Moreover, I split the dataset in advance and redo the test as well.

Result in Spark Java:

Confusion Matrix:

108	16
23	53

Accuracy: 0.805

True positive rate: 0.871

Area under ROC: 0.784

Result in R:

Confusion Matrix:

107	20
17	56

Accuracy: 0.815

True positive rate: 0.842

Area under ROC: 0.877

The true positive number and the true negative number are very close to each other but the false positive number and false negative number swap again. The accuracy is 1%different but the AUROC of Spark Java is 10.6% lower than R. So, using R is a better way.

8 Refinements

From the library and API documents given by Spark Java, there are still many things needed to be perfect. In the evaluation part, different evaluation indicators are located in different libraries and it is a little bit difficult to find the all the correct method. At first, when the code are run repeatedly, the output are the same, because the program created a pseudo random number to split the dataset into training and test part. So I use a timestamp to get a random number instead of setting a seed. While using the method in Spark Java to split the data into 2 parts, there is a slight difference in the total number of the test data every time so I normalize the confusion matrix instead of directly using it.

9 Reference

https://en.wikipedia.org/wiki/Logistic_regression

https://en.wikipedia.org/wiki/Jackknife_resampling

https://www.douban.com/note/265508213/

http://blog.csdn.net/hqh45/article/details/42914945

http://blog.csdn.net/yujunbeta/article/details/24141553

https://spark.apache.org/docs/1.6.2/api/java/org/apache/spark/SparkConf.html

https://spark.apache.org/docs/2.2.0/api/java/org/apache/spark/mllib/util/MLUtils.html

https://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html#/Q01:_Some_sample_uses_of_libsvm

https://stats.stackexchange.com/questions/61328/libsvm-data-format

http://blog.csdn.net/kobesdu/article/details/8944851

https://www.r-bloggers.com/how-to-perform-a-logistic-regression-in-r/

https://www.statmethods.net/advstats/glm.html

https://www.utdallas.edu/~herve/abdi-Jackknife2010-pretty.pdf

10 Output screenshots

Spark Java:

```
Confusion matrix:
108.0  16.0
23.0   53.0
Accuracy = 0.805
True positive rate = 0.870968
Area under precision-recall curve = 0.7902421815408085
Area under ROC = 0.7841680814940577
```

R:

```
[1] "Accuracy 0.815"
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 107  20
         1  17  56

               Accuracy : 0.815
                 95% CI : (0.7541, 0.8663)
    No Information Rate : 0.62
    P-Value [Acc > NIR] : 1.832e-09

                  Kappa : 0.6044
 Mcnemar's Test P-Value : 0.7423

            Sensitivity : 0.8629
            Specificity : 0.7368
         Pos Pred Value : 0.8425
         Neg Pred Value : 0.7671
             Prevalence : 0.6200
         Detection Rate : 0.5350
   Detection Prevalence : 0.6350
      Balanced Accuracy : 0.7999

       'Positive' Class : 0

[1] "AUC 0.876697792869269"
```