

# ***Assignment 1 : Classification and Regression Algorithms***

Name – Tanmay Ingle, Bhaskar Akula

Dataset : white vinho verde wine samples data, from the north of Portugal. Dataset is used as it is for regression where the value of target variable (quality goes from 1-10). However quality variable is replaced by a binary categorical for classification where  $\text{quality} \geq 7 = 1$  and  $\text{quality} < 7 = 0$ .

Thus we have 1080 1s and 3800 0s. We haven't done any feature selection in the beginning to get insights into how our algorithms behave for such datasets.

We have made new datasets called Wine-Classification.csv and Wine\_Regression.csv

### **Performance metrics used for classification:-**

We have used three metrics for classification – Accuracy, ROC-Area Under curve and Area under Precision-Recall curve.

Accuracy is the number of correctly classified examples divided by the total examples. Since the distribution of categories in target variable is uneven, accuracy is not the only metric to be looked at. (E.g. if there are 100 records and 80 of them are 1s and 20 are 0s. Then even if our model has 80% accuracy, it is still a useless model since it is predicting 1 for all records). Thus to calculate how good a classifier a model is, we use the other two metrics.

The area under the ROC curve (commonly referred to as AUC) represents an average value. An AUC of 1.0 will represent a perfect classifier. An area of 0.5 is referred to as the random score. Thus, a model that achieves an AUC of 0.5 is no better than randomly guessing. Thus we tune our models to have maximum AUC even though we have to compromise on accuracy.

The area under PR curve is referred to as the average precision. An area under the PR curve of 1.0 will equate to a perfect classifier that will achieve 100 percent in both precision and recall. Thus it should as close to 1 as possible.

### **Performance metrics for regression:-**

We have majorly used Root Mean Square error as our metric for regression algorithms. As it states, it is the square root of mean of squared errors. This metric is used because it is easier to interpret coz it has the same unit as that of target variable.

In SKlearn we have also used RSquared as the metric to compare the models.

### **Regularization:-**

Regularization is required to prevent over-fitting by penalizing the model complexity. This is done by adding a term to the loss function that acts to increase the loss as a function of the model weight vector. Vice versa is also true. Since regularization makes the model simple, it can also hamper the performance

of the model and lead to under fitting. It is important when we have large number of features or if we have features which aren't significant. We have used two types of regularisers – L1 and L2.

L2 reduces the weights of less significant features however L1 directly removes the features by making the weights zero. Higher the reg parameter, higher the penalty on the model.

It is not advisable to use models without regularization. Thus L0 is not recommended at all.

## Summary of three implementations: Python(sklearn), Scala, Pyspark

### a) Python(sklearn)

**Configuration and Implementation:** We used sklearn package in python. The sklearn which comes with Anaconda has earlier version of sklearn. Updating this sklearn using 'conda update sklearn' will create duplicates and cause errors. Thus we recommend removing sklearn from anaconda by the command 'conda remove sklearn' and then installing it again by the command 'conda install sklearn'. Certain features like selecting solver = lbfgs in logistic regression is not there in earlier version of sklearn. We used ipython notebook for this assignment. All the ipython notebooks have been attached in with the assignment. Anyone who wishes to run the code can simply copy these notebooks in their root directory of ipython notebook and run the code.

### Results:

#### *Classification*

	Accuracy(Test Data) %		
	No regularisor	L1 regularisor	L2 Regulariser
SGD Classifier(Hinge loss=SVM)	64	78	69
SGD Classifier(log loss=Logistic Regression)	62	78	75
Logistic Regression(LBFGS)	79		

Loss Function: - loss function is basically the function which SGD (stochastic gradient descent tried to minimize). Log-loss minimization may sacrifice classification accuracy if it allows it to model probabilities better.

Regularisors will be explained in the next segment. We have used two regularisors above for SGD and as it can be seen, L1 has done a better job at being accurate.

#### *Regression*

We have done 4 types of regression on the data. Their RMSE errors are as follows:-

1) Linear Regression: RMSE = 0.77, Rsquared = 0.27

2) SGD Regressor: RMSE = 0.76 , Rsquared = 0.29

3) Ridge Regressor: RMSE = 0.76, Rsquared = 0.29

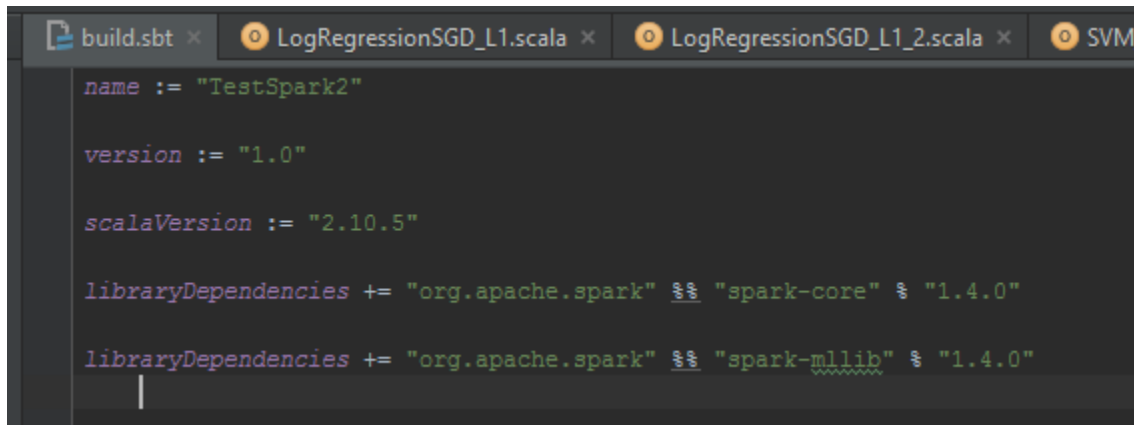
4) Lasso Regression: RMSE = 0.78, Rsquared = 0.28

We haven't used regularizer because it doesn't make sense in this case. Ridge is L2 regularization and Lasso is L1.

The RMSE is very less in this case given the range of the target variable. Scikit is very easy to implement and is perfectly suitable for doing data science.

#### b) Scala

We have used IntelliJ as our IDE for writing scala code in Apache Spark. We created a build.sbt file having all the dependencies of spark.

A screenshot of the IntelliJ IDEA IDE showing the 'build.sbt' file. The file contains the following code:

```
name := "TestSpark2"

version := "1.0"

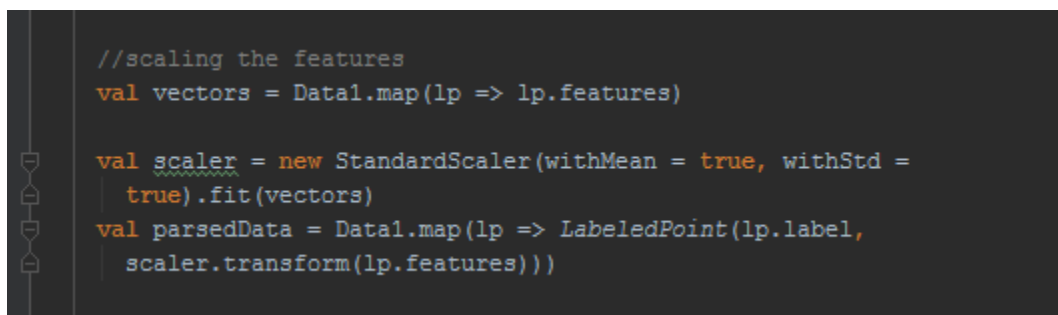
scalaVersion := "2.10.5"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.0"

libraryDependencies += "org.apache.spark" %% "spark-mllib" % "1.4.0"
```

The IDE tabs at the top show 'build.sbt', 'LogRegressionSGD\_L1.scala', 'LogRegressionSGD\_L1\_2.scala', and 'SVM'.

As mentioned before, our focus while tuning the algorithms was to increase the AUC and Area under PR curve. The major factor which resulted in a higher AUC was scaling the features before modelling.

A screenshot of Scala code in an IDE, showing feature scaling. The code is as follows:

```
//scaling the features
val vectors = Data1.map(lp => lp.features)

val scaler = new StandardScaler(withMean = true, withStd = true).fit(vectors)

val parsedData = Data1.map(lp => LabeledPoint(lp.label, scaler.transform(lp.features)))
```

Due to uneven range of features, scaling was of importance. However standardizing can be expensive when there is large amount of data.

Then we split the data into 70% training and 30% test data.

```
//splitting into training and test dataset
val splits = parsedData.randomSplit(Array(0.7, 0.3), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
```

Next step was to model the data.

### ***Classification***

Screenshots of modelling two of the classification algorithms are as follows (we have used mllib library in spark):-

Logistic Regression with L1

```
//Building the Model

val model = new LogisticRegressionWithSGD()
model.optimizer.setRegParam(0.1).setNumIterations(10).setUpdater(new L1Updater).setStepSize(1)

val lrModelSGD = model.run(training)
```

Logistic Regression with LBFGS

```
//Building the Model

val numFeatures = Data1.take(1)(0).features.size

val numCorrections = 10
val convergenceTol = 1e-4
val maxNumIterations = 20
val regParam = 0.01
val initialWeightsWithIntercept = Vectors.dense(new Array[Double](numFeatures + 1))

val (weightsWithIntercept, loss) = LBFGS.runLBFGS(
  training1,
  new LogisticGradient(),
  new SquaredL2Updater(),
  numCorrections,
  convergenceTol,
  maxNumIterations,
  regParam,
  initialWeightsWithIntercept)

val lrModel = new LogisticRegressionModel(
  Vectors.dense(weightsWithIntercept.toArray.slice(0, weightsWithIntercept.size - 1)),
  weightsWithIntercept(weightsWithIntercept.size - 1))
```

Next we computed Accuracy for training and test data for each model

```
//Accuracy test
val lrTotalCorrect1 = test.map { point =>
  if (lrModel.predict(point.features) == point.label) 1 else 0
}.sum
val lrAccuracy1 = lrTotalCorrect1/test.count
```

Then we cleared the threshold of the model to get the probability scores so that we can get Area under ROC curve and Area under PR curve.

```

//Area under ROC training
lrModel.clearThreshold()

val scoreAndLabels = training.map { point =>
    val score = lrModel.predict(point.features)
    (score, point.label)
}

// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()

//Area under PR

val auPR = metrics.areaUnderPR()

```

Thus we computed these three metrics for training and test dataset for each of the classification model. Few of the screenshots are as follows:-

For Logistic Regression with L1

```

15/06/21 22:44:44 INFO TaskSetManager: Finished task 0.0 in stage 51.0 (TID 74) in 9 ms on
15/06/21 22:44:44 INFO DAGScheduler: ResultStage 51 (aggregate at AreaUnderCurve.scala:45)
15/06/21 22:44:44 INFO TaskSchedulerImpl: Removed TaskSet 51.0, whose tasks have all compl
15/06/21 22:44:44 INFO DAGScheduler: Job 29 finished: aggregate at AreaUnderCurve.scala:45
Area under ROC training= 0.7568256265902671
Area under ROC test= 0.7416515899122806
Area under PR training= 0.486715286342039
Area under PR test= 0.45022557352004944
Accuracy training= 0.6733566026759744 Accuracy test= 0.669863013698630115/06/21 22:44:44
15/06/21 22:44:44 INFO DAGScheduler: Stopping DAGScheduler
15/06/21 22:44:44 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stop

```

For SVM with L2

```

15/06/21 22:46:35 INFO TaskSetManager: Finished task 0.0 in stage 71.0 (TID 114) in 0.0
15/06/21 22:46:35 INFO DAGScheduler: ResultStage 71 (aggregate at AreaUnderCurve.scala
15/06/21 22:46:35 INFO TaskSchedulerImpl: Removed TaskSet 71.0, whose tasks have all c
15/06/21 22:46:35 INFO DAGScheduler: Job 49 finished: aggregate at AreaUnderCurve.scal
Area under ROC training= 0.7739862360507237
Area under ROC test= 0.7545888157894745
Area under PR training= 0.5119555590059552
Area under PR test= 0.48378449482343994
Accuracy training= 0.6660849331006399 Accuracy test= 0.671232876712328815/06/21 22:46
15/06/21 22:46:35 INFO BlockManagerInfo: Removed broadcast_84_piece0 on localhost:5357

```

We tuned Number of Iterations, step Size and regParam for every model to get maximum AUC. After a point number of iterations have very less effect on the model and it just increases the cost. We varied step Size from 0.0001 to 10 in the multiples of 10 to see the change in AUC and tuned it for the highest AUC.

We observed that when we train and evaluate our model on the same dataset, we generally achieve the highest performance when regularization is lower. This is because our model has seen all the data points, and with low levels of regularization, it can over-fit the data set and achieve higher performance. In contrast, when we train on one dataset and test on another, we see that generally a slightly higher level of regularization results in better test set performance.

Results:-

	AUC-ROC(test)	AUC-PR(test)	Accuracy(test)
LogReg(L1)	0.74	0.45	0.669
LogReg(L2)	0.77	0.5	0.672
LogReg(LBFGS)	0.77	0.51	0.8
SVM(L1)	0.75	0.48	0.67
SVM(L2)	0.75	0.48	0.67

Thus as we can see Logistic Regression with LBFGS is best model for this dataset. It has high AUC as well as high Accuracy.

Regression:-

Linear Regression Model is unregularized. However Ridge is L2 and Lasso is L1.

Thus Ridge Regression with L1 doesn't make sense and vice versa. Thus we have created only three models.

E.g.

```
// Building the model  
  
val model1 = new RidgeRegressionWithSGD()  
  
model1.optimizer.setStepSize(0.0005).setNumIterations(250).setRegParam(0)  
  
val model = model1.run(training)
```



The to evaluate the model we find RMSE for training and test data

```
// Evaluate model on training examples and compute training error
val valuesAndPreds = training.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}

//testing the model

val valuesAndPreds1 = test.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}

println(model.weights)

val MSE = valuesAndPreds.map{case(v, p) => math.pow((v - p), 2)}.mean()
println("training Root Mean Squared Error = " + math.sqrt(MSE))

val MSE1 = valuesAndPreds1.map{case(v, p) => math.pow((v - p), 2)}.mean()
println("test Root Mean Squared Error = " + math.sqrt(MSE1))
```

We tune number of iterations, stepsize and regParameter in the same way as in classification. We find the model with lowest RMSE.

Results:

	RMSE(training)	RMSE(test)
LinearRegression	1.62	1.64
RidgeRegression	1.62	1.64
LassoRegression	1.67	1.68

Following are the weights of features in Ridge Regression:-

[0.047905757827890395, 0.001677780443146662, 0.0021650226103928512,-  
0.0016863429288773673, 1.9082867381981124E-4,0.011360789832666439,  
0.026413566613464112,0.007293403468208688, 0.023695144053257407, 0.003422645185906912 ,  
0.0953109198472794]

Following are the weights of features in LassoRegression:-

```

15/06/21 23:14:57 INFO BlockManagerInfo: Removed broadcast_502_piece0 on localhost:34672 in memory (size: 3.3 KB, free: 132.3 MB)
[0.04088965592267554,0.0,0.0,-0.0,0.0,0.00852516231321282,0.027979373435430933,0.0,0.01625762449458032,0.0,0.08890639392973286]
15/06/21 23:14:57 INFO SparkContext: Starting job: mean at Lasso_regression.scala:60
15/06/21 23:14:57 INFO DAGScheduler: Got job 252 (mean at Lasso_regression.scala:60) with 2 output partitions (allowLocal=false)

```

As you can see Lasso actually assigns zero to weights of less significant features. Number of zero weights go on increasing as we increase the regParam. In this case 2<sup>nd</sup>, 3<sup>rd</sup>, 4<sup>th</sup>, 5<sup>th</sup>, 8<sup>th</sup> and 10<sup>th</sup> feature are not considered. These features are volatile acidity, citric acid, residual sugar, free sulfurdioxide, sulphates, and density and not considered by lasso and in spite of that, the RMSE is off by a small margin.

This L1 i.e. lasso can be used for feature selection.

### c) PySpark

We have used ipython notebook to have interactive scripting with Apache Spark in python. All the notebook files can be imported into an ipython notebook which is configured with spark and can be used.

The flow of the code is similar to the one explained above for Scala. However there exists certain limitations in pyspark. E.g. Binary classification matrix module in mllib used for calculating AUC in scala is not present in pyspark. So to compute performance metrics we have converted the RDD to an array using numpy.array and used sklearn modules to compute these metrics.

Also we haven't standardized the features before modelling and thus the metrics will differ significantly from Scala. This is done to understand the importance of standardization.

All the steps remain the same as done in Scala. Following are few screenshots of modelling and metrics which are calculated for classification and regression.

```

In [1]: from pyspark import SparkConf, SparkContext
        from pyspark.mllib.regression import LabeledPoint
        from sklearn import metrics
        from numpy import array
        import numpy as np
        from sklearn.metrics import roc_auc_score
        from sklearn.metrics import average_precision_score
        from sklearn.metrics import confusion_matrix
        from pyspark.mllib.classification import LogisticRegressionWithSGD, LogisticRegressionModel
        sc = SparkContext(appName="ClassificationOfWineDataset")

In [2]: #getting data
        data = sc.textFile("D://NEU - Big Data and Intelligent Analytics/Assignment 1/winequality-white-classification-parsed format.csv")

```

```
In [65]: #clearing the threshold for getting scores as prediction.
model.clearThreshold()
```

```
In [66]: #making an array of true lables and its predicted scores for training data
scoreandlabels = np.array(training.map(lambda p: (p.label, model.predict(p.features)),
print "Training Area under curve = " + str(roc_auc_score(scoreandlabels[:,0],scoreandlabels[:,1]))
print "Training Area under Precision-Recall = " + str(average_precision_score(scoreandlabels[:,0],scoreandlabels[:,1]))

Training Area under curve = 0.67741902931
Training Area under Precision-Recall = 0.326743663734
```

```
In [67]: #making an array of true lables and its predicted scores for test data
scoreandlabels1 = np.array(test.map(lambda p: (p.label, model.predict(p.features)),
print "Test Area under curve = " + str(roc_auc_score(scoreandlabels1[:,0],scoreandlabels1[:,1]))
print "Test Area under Precision-Recall = " + str(average_precision_score(scoreandlabels1[:,0],scoreandlabels1[:,1]))

Test Area under curve = 0.690089434107
Test Area under Precision-Recall = 0.382073277092
```

```
[63]: # Evaluating the model on training data
labelsAndPreds = training.map(lambda p: (p.label, model.predict(p.features)),
accuracy_training = labelsAndPreds.filter(lambda (v, p): v == p).count() / labelsAndPreds.count()
print("Accuracy(training) = " + str(accuracy_training))
x = np.array(labelsAndPreds.collect())

Accuracy(training) = 0.563311220075
```

```
[64]: # Evaluating the model on test data
labelsAndPreds1 = test.map(lambda p: (p.label, model.predict(p.features)),
accuracy_test = labelsAndPreds1.filter(lambda (v, p): v == p).count() / labelsAndPreds1.count()
print("Accuracy(test) = " + str(accuracy_test))
y = np.array(labelsAndPreds1.collect())

Accuracy(test) = 0.589797344514
```

Above screenshots are just e.g. of our code. Entire code can be found in the assignment folder.

Classification:-

	AUC-ROC(test)	AUC-PR(test)	Accuracy(test)
LogReg(L1)	0.61	0.39	0.7
LogReg(L2)	0.69	0.38	0.58
LogReg(LBFGS)	0.77	0.5	0.79
SVM(L1)	0.66	0.32	0.77

SVM(L2)	0.64	0.31	0.69
---------	------	------	------

As we can see, due to lack of standardization, performance of the model has been affected a lot. Also we hope that with more features being added to mllib of python support pyspark will get much richer in future.

#### **Q4. Compare and contrast using Just Scipy libraries vs using Apache Spark. How did the results vary? When would you use just Python libraries and when would you use Apache Spark?**

Libraries in Scipy as of now are very rich as compared to Apache Spark mllib. However with the kind of pace at which databricks is adding to the mllib, it won't take long for it to reach a level close to Scipy.

Results varies quite a lot because Apache Spark mllib library consist of algorithms which are specifically developed for parallel computation and in this assignment we did not use that aspect of spark.

Also there was huge difference between the times required for computation. Apache Spark took more time for computation. But this will reverse as the size of data we deal with goes on increasing.

We would recommend using Scipy for local computation since it is rich and faster for small dataset. However if we need to compute on large datasets, Apache Spark with other cloud services will be an evident choice.