**CSYE7374 52001 ST: Big-Data Sys & Int Anltcs SEC 01 - Summer Full 2015**

**Team 4**

Samir Sharan, Harshit Shah, Jeevan Reddy

**Assignment 1**

1. **Summarizing implementation**

The algorithms which we are implementing comes under the **“Supervised Learning”** category which has the data with additional attributes that we want to predict. In the wine dataset “quality” is the output attribute. For implementing it as a binary class we have converted the quality column to 0’s and 1’s based on the following criteria:

0 - low quality wines: values <= 7

1 - high quality wines: values >7

We have implemented the various algorithms in plain python, PySpark and Scala

* 1. **Plain Python:**

We have used Spyder to run the code in python.

**Steps:**

1. Install Anaconda
2. After Anaconda is installed run “spyder” command on the command line
3. This will launch the spyder console where the code provided can be just copied on the IDE and run by clicking the run button.

**Classification:**

1. **sklearn.linear\_model.SGDClassifier (SVM => Hinge loss)**: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule. **For best results using the default learning rate schedule, the data should have zero mean and unit variance**.

**Loss:** The model it fits can be controlled with the loss parameter.

**Penalty:** The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1.

**Parameters used:**

* **loss:** Defaults to ‘hinge’, which gives a linear SVM
* **penalty:** Defaults to ‘l2’. Implemented ‘l1’ and ‘l2’ both
* **alpha:** Constant that multiplies the regularization term
* **n\_iter:** number of iterations

**Confusion Matrix:**

**L1:**

0 1

0[[1887    4]

1 [  68    0]]

**L2:**

0 1

0 [[1889    2]

1 [  68    0]]

       2. **sklearn.linear\_model.SGDClassifier (Logarithmic regression => log loss):**

**Parameters used:**

* **loss:** The ‘log’ loss gives logistic regression
* **penalty:** Defaults to ‘l2’. Implemented ‘l1’ and ‘l2’ both
* **alpha:** Constant that multiplies the regularization term
* **n\_iter:** number of iterations

**Confusion Matrix:**

**L1:**

0 1

0[[1880    7]

1 [  72    0]]

**L2:**

0 1

0 [[1894    1]

1 [  64    0]]

**3. sklearn.linear\_model.LogisticRegression (LBFGS version) :**  This class implements regularized logistic regression using the liblinear library, newton-cg and lbfgs solvers. We are implementing ‘lbfgs’. The lbfgs solvers support only L2 regularization.

**Parameters used:**

* **penalty:** Defaults to ‘l2’. Implemented ‘l1’ and ‘l2’ both
* **solver** : Algorithm to use in the optimization problem - ‘lbfgs’

**Confusion Matrix:**

**L1:**

0 1

0[[1886    0]

1 [  73    0]]

**L2:**

0 1

0 [[1878    0]

1 [  81    0]]

**4. sklearn.linear\_model.LinearRegression:** This algorithm doesn’t have a penalty parameter. So running it with the default parameters.

**5. sklearn.linear\_model.SGDRegressor:**

**Parameters used:**

* **random\_state:** The seed of the pseudo random number generator to use when shuffling the data
* **penalty:** Defaults to ‘l2’. Implemented ‘l1’ and ‘l2’ both
* **alpha:** Constant that multiplies the regularization term
* **n\_iter:** number of iterations

**6. sklearn.linear\_model.Ridge:** This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm.

**Parameters used:**

* **alpha** : Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates
* **max\_iter:** Maximum number of iterations for conjugate gradient solve

**7. sklearn.linear\_model.Lasso:** Linear Model trained with L1 prior as regularizer

**Parameters used:**

* **alpha:** Defaults to 1.0. alpha = 0 is equivalent to an ordinary least square, solved by the LinearRegression object. For numerical reasons, using alpha = 0 is with the Lasso object is not advised
* **random\_state** : The seed of the pseudo random number generator that selects a random feature to update
* **normalize: I**f True, the regressors X will be normalized before regression

The L1 and L2 values in the tables below are score values: **Returns the mean accuracy on the given test data and labels**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Algorithms | Regularization | | |
|  | L1 | L2 |
| Classification | sklearn.linear\_model.SGDClassifier (SVM => Hinge loss) |  | 0.955079122001 | 0.965288412455 |
| sklearn.linear\_model.SGDClassifier (Logarithmic regression => log loss) |  | 0.960694231751 | 0.96426748341 |
| sklearn.linear\_model.LogisticRegression (LBFGS version) |  |  | 0.966309341501 |
| Regression | sklearn.linear\_model.LinearRegression | 0.289496074035 |  |  |
| sklearn.linear\_model.SGDRegressor |  | 0.28 | 0.25 |
| sklearn.linear\_model.Ridge |  |  | 0.26694099672 |
| sklearn.linear\_model.Lasso |  | 0.27 |  |

* 1. **Pyspark**

We have used iPython Notebook to run Pyspark code.

**Steps:**

1. Install Spark
   1. Download the [source for the latest Spark release](http://spark.apache.org/downloads.html)
   2. Unzip source to ~/spark-1.4.0 (or wherever you wish to install Spark)
   3. From the Terminal/Command Prompt, type: cd ~/spark-1.4.0/
   4. Build Spark: sbt assembly (Takes a while)

**\Users\...\sbt-0.13.7\sbt\bin\sbt**

1. Create PySpark profile for IPython
   1. Type the following command in Terminal/Command Prompt to create a profile for PySpark

**ipython profile create pyspark**

1. Set the environment variable wherever you installed Spark

**export SPARK\_HOME="$HOME/spark-1.4.0"**

1. Run the following command to open iPython Notebook

ipython notebook --profile=**pyspark**

1. Copy the provided code and run it.

We have 2 categories:

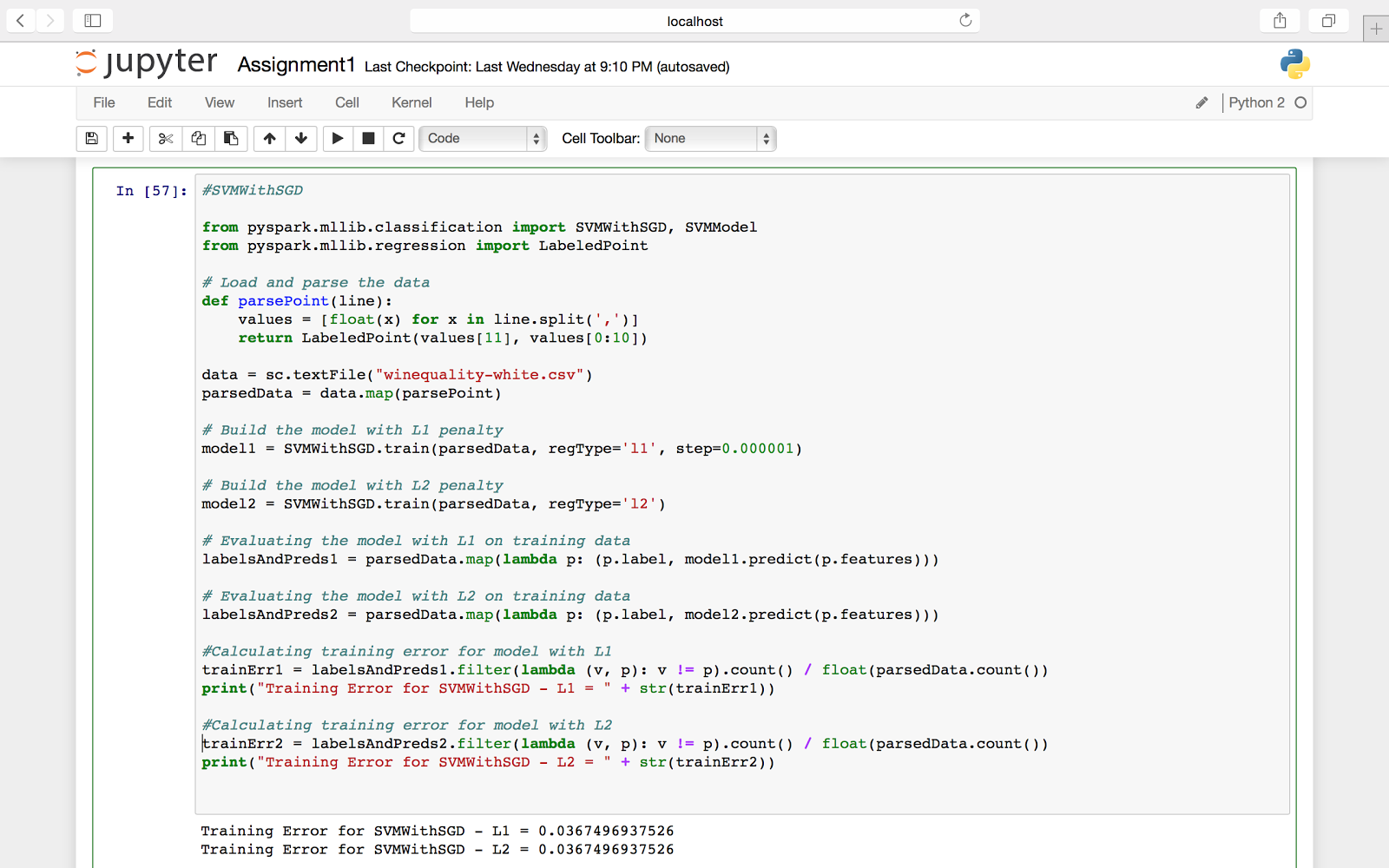
**Classification:**

1. **SVMWithSGD**: The linear SVM is a standard method for large-scale classification tasks. It is a linear method with the loss function in the formulation given by the **hinge** loss. Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to discriminative learning of linear classifiers under convex loss functions.

**Penalty:** The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1.

* **Parameters**
  + **data** – The training data, an RDD of LabeledPoint
  + **iterations** – The number of iterations (default: 100).
  + **step** – The step parameter used in SGD (default: 1.0).
  + **regParam** – The regularizer parameter (default: 0.01)
  + **miniBatchFraction** – Fraction of data to be used for each SGD iteration.
  + **initialWeights** – The initial weights (default: None)
  + **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data
  + **regType** - “l1” for L1 regularization

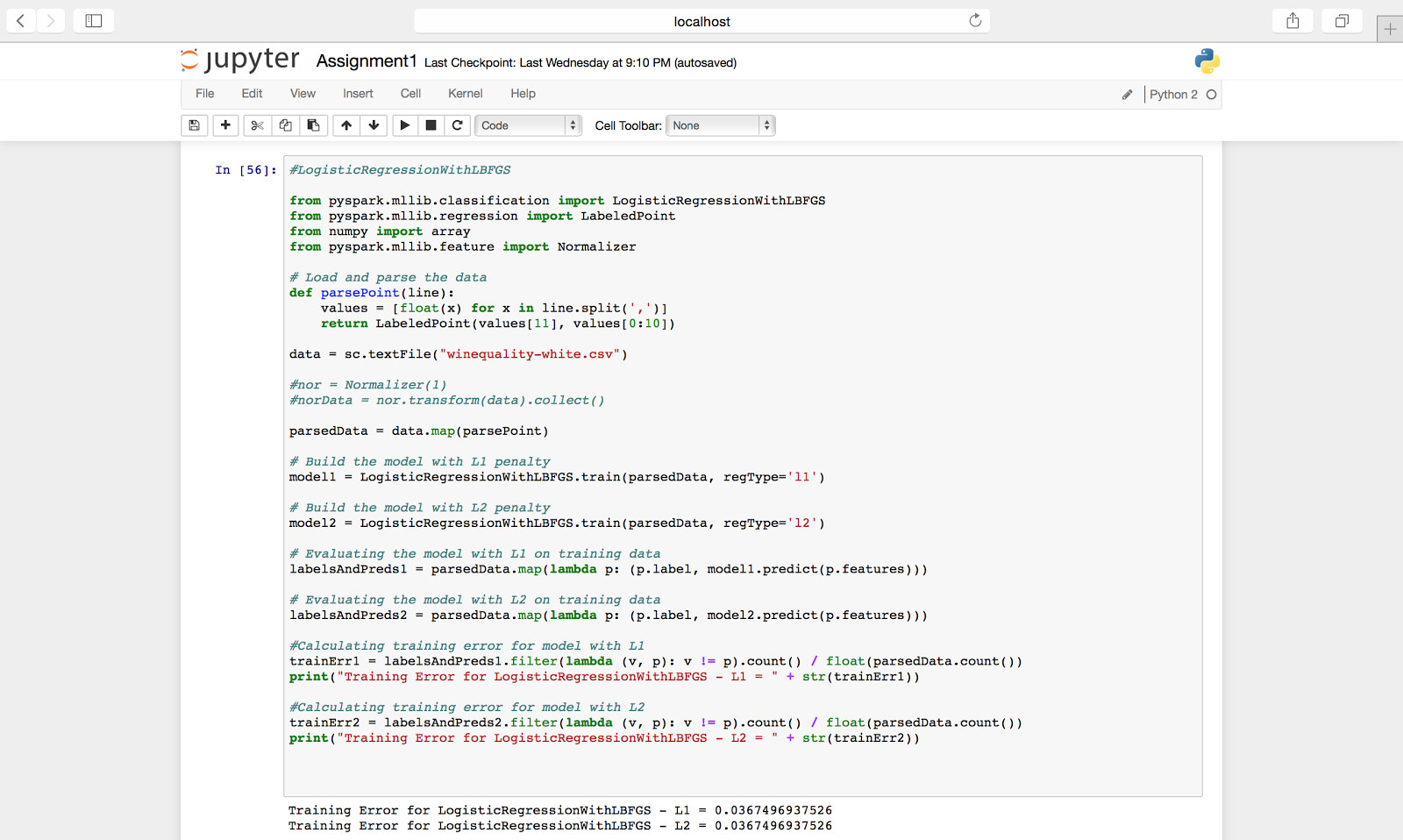
                                  - “l2” for L2 regularization

****

1. **LogisticRegressionWithLBFGS:** Logistic regression is widely used to predict a binary response. It is a linear method with the loss function given by the logistic loss

* **Parameters**
  + **data** – The training data, an RDD of LabeledPoint
  + **iterations** – The number of iterations (default: 100).
  + **regParam** – The regularizer parameter (default: 0.01)
  + **initialWeights** – The initial weights (default: None)
  + **corrections** – The number of corrections used in the LBFGS update (default: 10)
  + **tolerance** – The convergence tolerance of iterations for L-BFGS (default: 1e-4).
  + **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e. whether bias features are activated or not).
  + **regType** - “l1” for L1 regularization

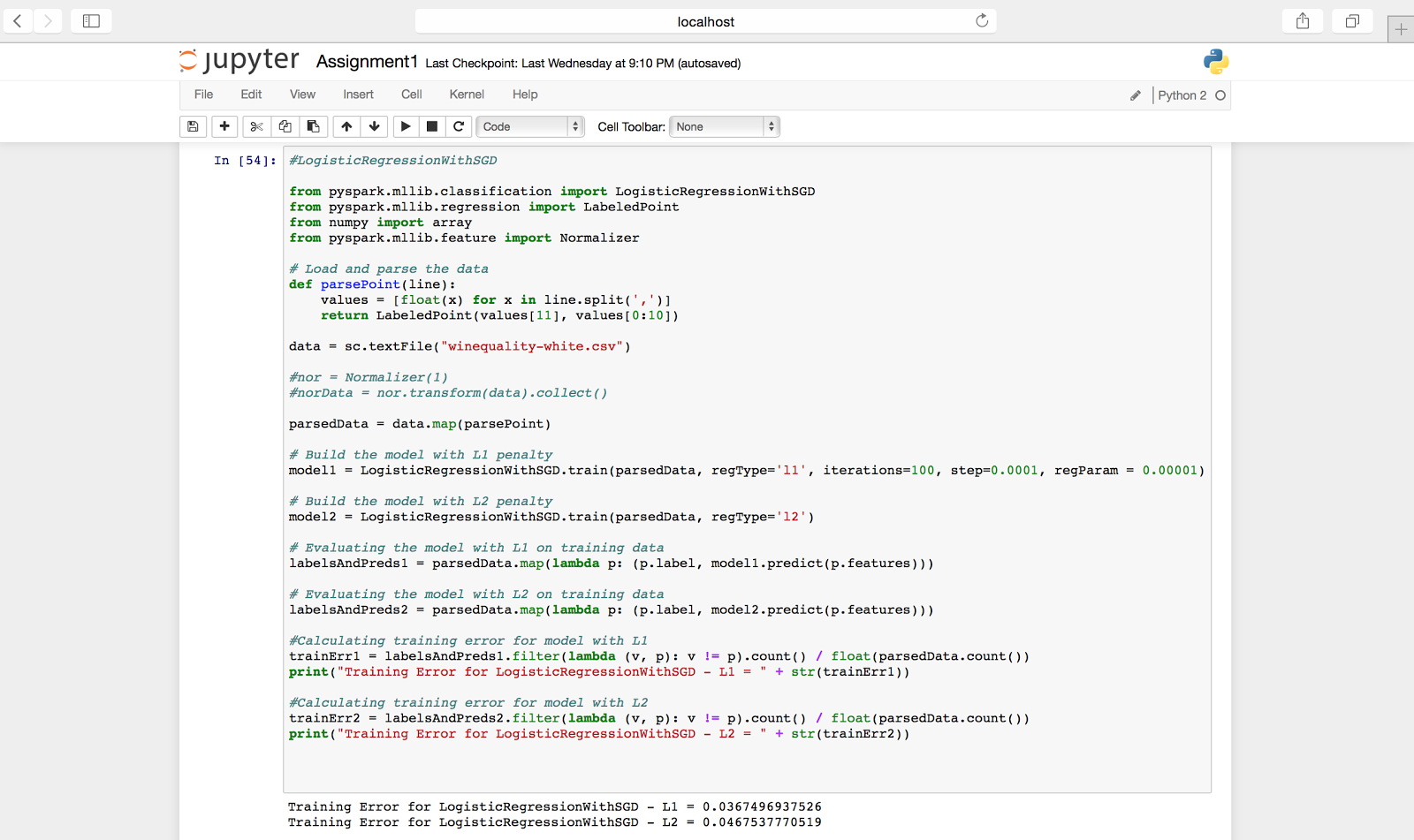
                 - “l2” for L2 regularization

****

1. **LogisticRegressionWithSGD:** Logistic regression is widely used to predict a binary response. It is a linear method with the loss function given by the logistic loss. Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to discriminative learning of linear classifiers under convex loss functions.

* **Parameters**
  + **data** – The training data, an RDD of LabeledPoint
  + **iterations** – The number of iterations (default: 100).
  + **regParam** – The regularizer parameter (default: 0.01)
  + **initialWeights** – The initial weights (default: None)
  + **step** – The step parameter used in SGD (default: 1.0)
  + **miniBatchFraction** – Fraction of data to be used for each SGD iteration.
  + **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e. whether bias features are activated or not).
  + **regType** - “l1” for L1 regularization

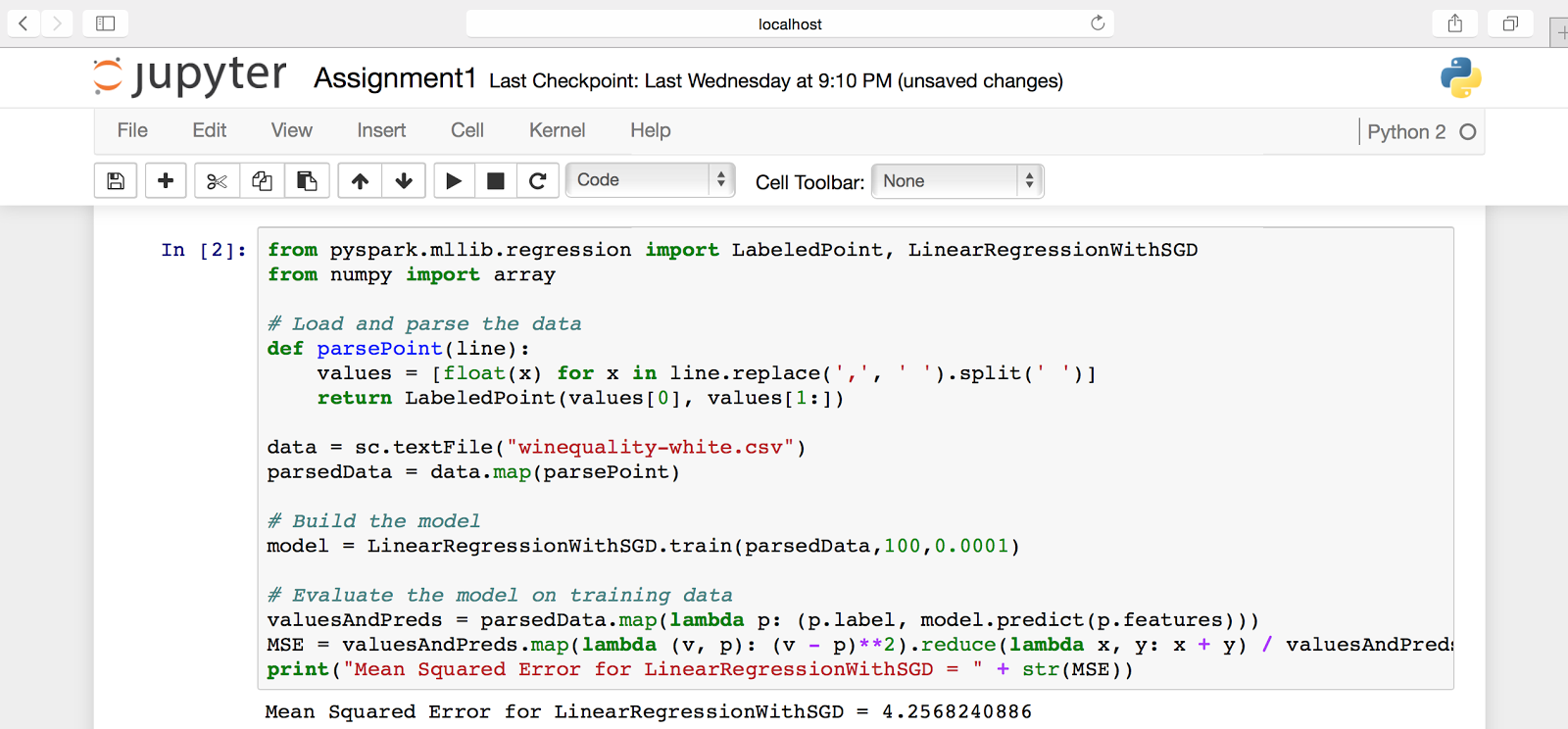
                 - “l2” for L2 regularization



**Regression:**

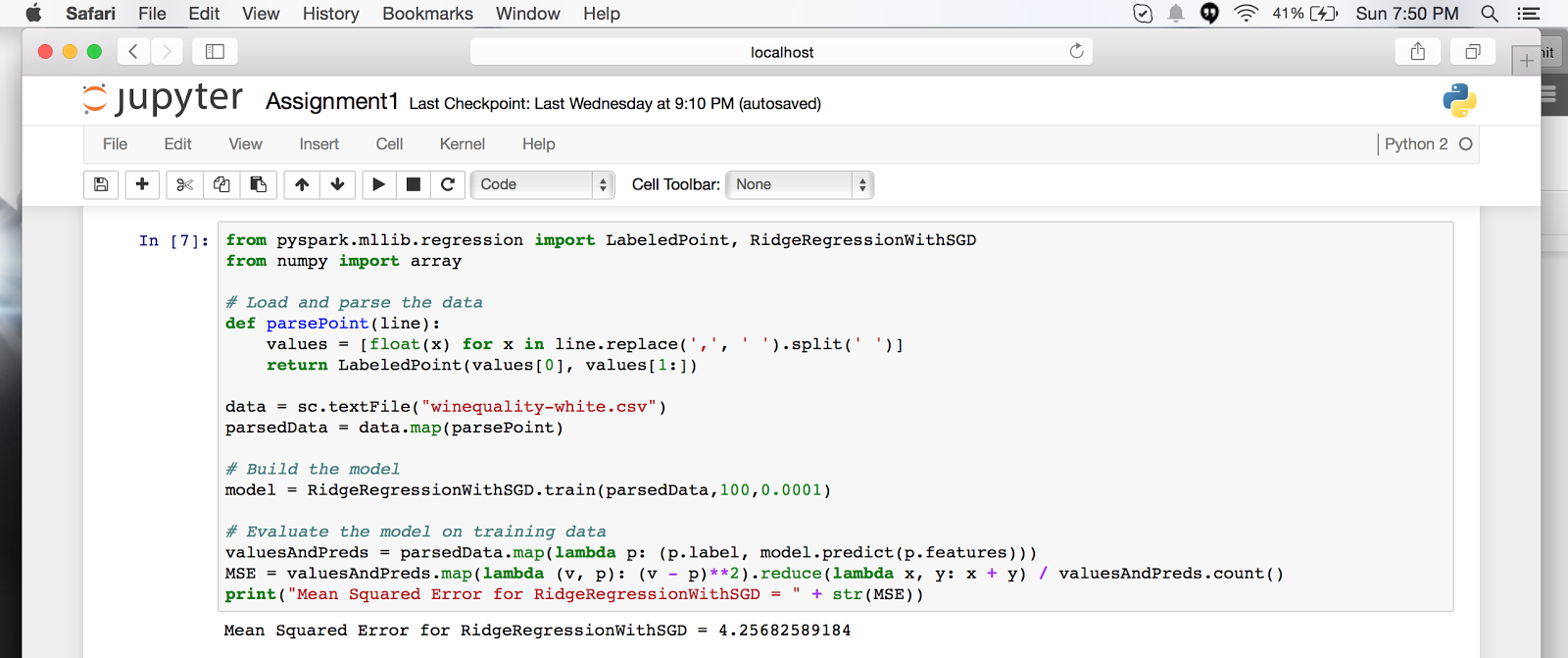
1. **LinearRegressionWithSGD**: Linear least squares is the most common formulation for regression problems. It is a linear method with the loss function in given by the squared loss. Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to discriminative learning of linear classifiers under convex loss functions.

* **Parameters**
  + **data** – The training data, an RDD of LabeledPoint
  + **iterations** – The number of iterations (default: 100).
  + **step** – The step parameter used in SGD (default: 1.0).
  + **regParam** – The regularizer parameter (default: 0.01)
  + **miniBatchFraction** – Fraction of data to be used for each SGD iteration.
  + **initialWeights** – The initial weights (default: None)
  + **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e. whether bias features are activated or not).
  + **regType** - None



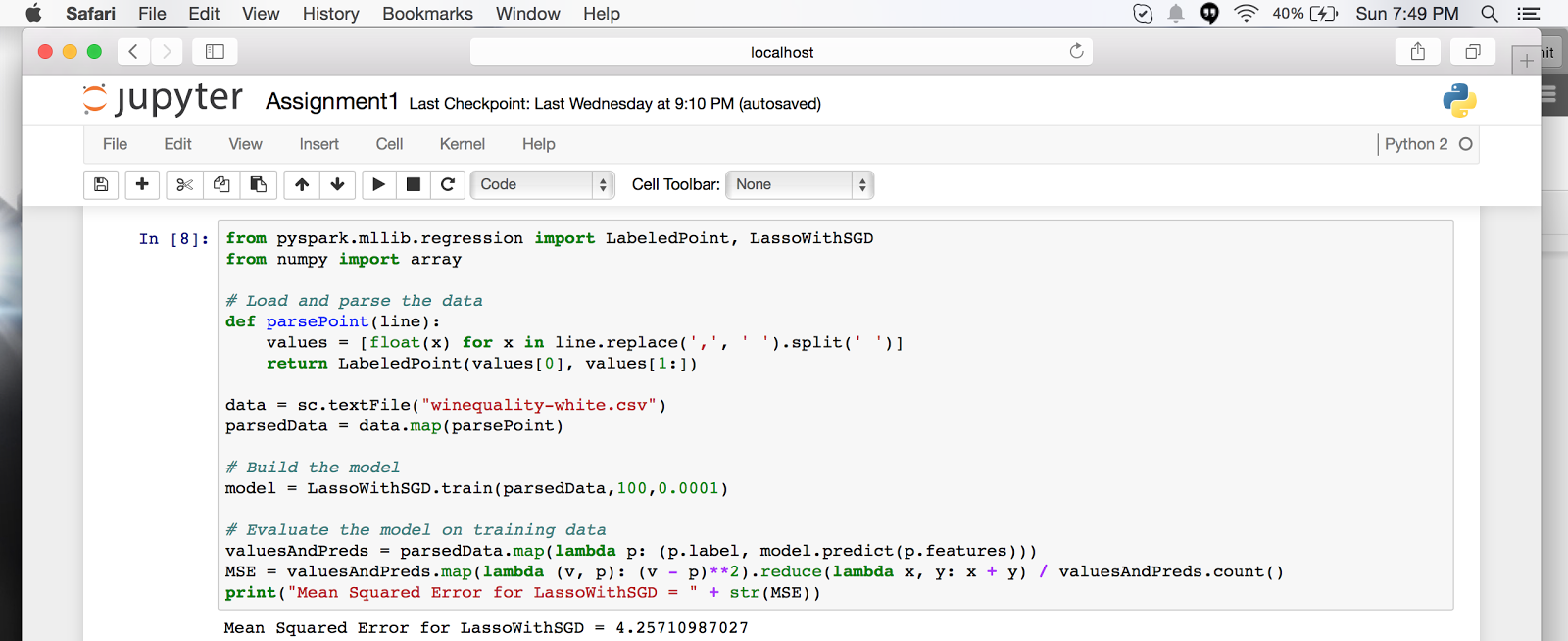
1. **RidgeRegressionWithSGD:**

* **Parameters**
  + **data** – The training data, an RDD of LabeledPoint
  + **iterations** – The number of iterations (default: 100).
  + **step** – The step parameter used in SGD (default: 1.0).
  + **regParam** – The regularizer parameter (default: 0.01)
  + **miniBatchFraction** – Fraction of data to be used for each SGD iteration.
  + **initialWeights** – The initial weights (default: None)
  + **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e. whether bias features are activated or not).
  + **regType** - “l2” for L2 regularization

****

1. **LassoWithSGD:**

* **Parameters**
  + **data** – The training data, an RDD of LabeledPoint
  + **iterations** – The number of iterations (default: 100).
  + **regParam** – The regularizer parameter (default: 0.01)
  + **initialWeights** – The initial weights (default: None)
  + **step** – The step parameter used in SGD (default: 1.0)
  + **miniBatchFraction** – Fraction of data to be used for each SGD iteration.
  + **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e. whether bias features are activated or not).
  + **regType** - “l1” for L1 regularization

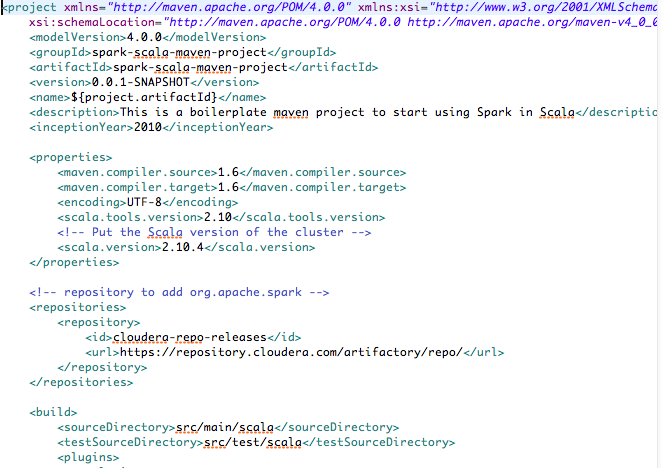
****

**Error Values using PySpark**

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Algorithms | Regularization | |
| L1 | L2 |
| Classification | SVMWithSGD | 0.0367496937526 | 0.0367496937526 |
| LogisticRegressionWithLBFGS | 0.0367496937526 | 0.0367496937526 |
| LogisticRegressionWithSGD | 0.0367496937526 | 0.0467537770519 |
| Regression | LinearRegressionWithSGD | 4.2568240886 | |
| RidgeRegressionWithSGD | N/A | 4.25682589184 |
| LassoWithSGD | 4.25710987027 | N/A |

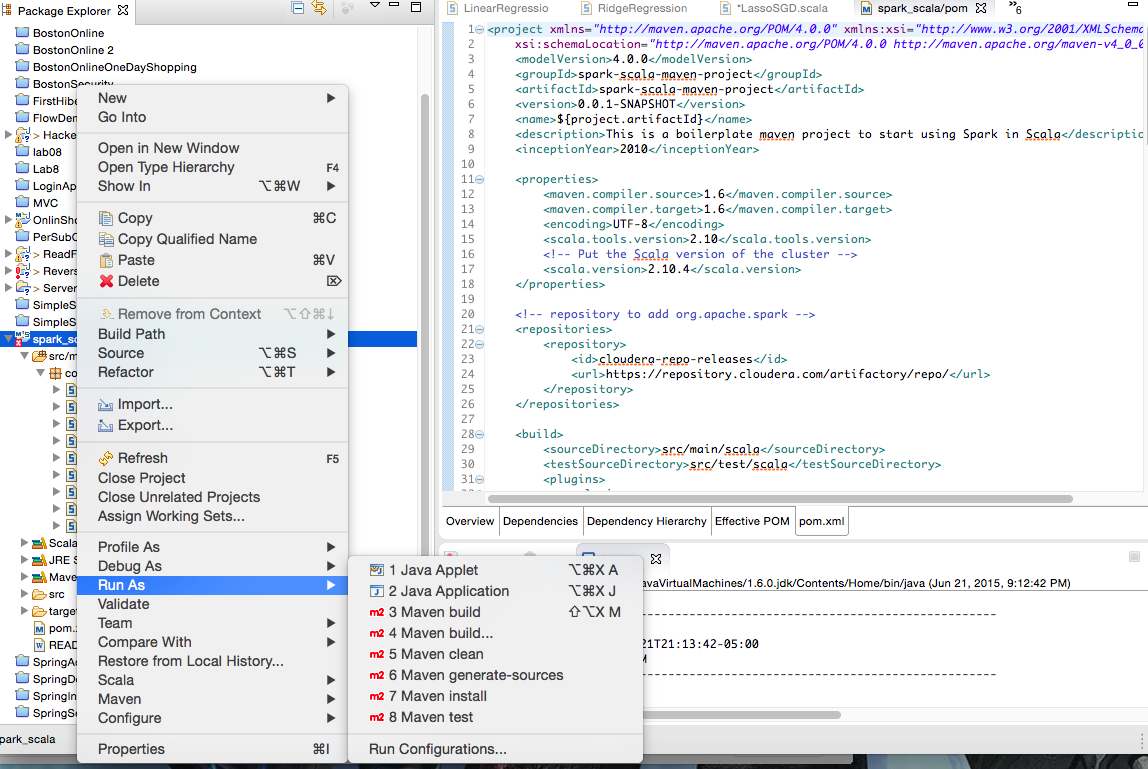
**c. Scala:**

1. Install Scala IDE in eclipse.
2. Configure the Maven dependencies.



4. Create Scala Object by right clicking the project and write the algorithm.

5. After the algorithm is complete run it by right clicking the on the project and click on Run as → Maven install.



6.  This will create the jar file under the project folder → Target folder

7. Go to the same location(project folder) in the Terminal :

**/Users/insignia/Downloads/spark-scala-maven-boilerplate-project-master**

**https://lh6.googleusercontent.com/3AGry4C7ZdP5fLvvhyJENvjYS_DTV_r6eQBHZjhVgpKEt578ZIG43YZGrqKc9lctJbOwvEqJ1im8Hrt_40-aFS8PVdzRJUWGEZvfm3DJX-fbKr2mSkWiLunLumBmXpr28FGy9w**

8. Run the following codes to get the output of the algorithms:

* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class “com.neu.algo.SVM\_L2”--master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar
* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class "com.neu.algo.SVM\_L1" --master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar
* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class "com.neu.algo.LogisticRegressionLBFGS\_L2" --master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar
* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class "com.neu.algo.LogisticRegressionLBFGS\_L1" --master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar
* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class "com.neu.algo.LogisticRegressionSGD\_L2" --master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar
* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class "com.neu.algo.LogisticRegressionSGD\_L1" --master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar
* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class "com.neu.algo.LinearRegressionSGD" --master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar
* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class "com.neu.algo.RidgeRegressionSGD" --master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar
* /Users/insignia/Desktop/Big\_Data\_Analytics/spark-1.4.0/bin/spark-submit --class "com.neu.algo.LassoSGD" --master local[1] target/spark-scala-maven-project-0.0.1-SNAPSHOT.jar

**https://lh3.googleusercontent.com/Tb4olT91xwDQb4SfY3EMNPZf4tpulWOfVIAYPmJgMZ9ocxb7ANxeBpxBbgqWqsHrqERMotz6DEwxkrTH2imJ23Krh9yWqyxOGdIgzIxksPKKVcmt1YI1t8ORQALXwZ3qCzG1jw**

**Error Values using Scala**

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Algorithms | Regularization | |
| L1 | L2 |
| Classification | SVMWithSGD | Area under ROC = 0.5970003978629032 | Area under ROC= 0.6260088666590823 |
| LogisticRegressionWithLBFGS | Precision = 0.9605809128630706 | Precision = 0.9605809128630706 |
| LogisticRegressionWithSGD | Precision = 0.9605809128630706 | Precision = 0.9605809128630706 |
| Regression | LinearRegressionWithSGD | training Mean Squared Error = 0.03405595905229357 | |
| RidgeRegressionWithSGD | N/A | training Mean Squared Error = 0.03405595906711153 |
| LassoWithSGD | training Mean Squared Error = 1.5173895481788833E76 | N/A |

**Interpretation:**

**Area under ROC :** The closer the value the 1 the better is the model

**Precision:** is the fraction of retrieved instances that are relevant

1. **Performance Metrics for Classification Algorithms**

* Number of Iterations (iterations) - Higher number of iterations is longer run time. Since our dataset is comparatively very small, the number of iterations don’t make a significant impact on the result of the algorithm.
* Regularization Type (regType) - To minimize overfitting, regularizers are introduced in the algorithm. L1 and L2 are two regularizers. Not every algorithm supports both regularizer. We saw a difference of 10% in error in LogisticRegressionWithSGD when using L2 and compared to L1, but for the rest of the classification algorithms it is not making any difference to the error.
* Alpha (alpha) - It is a constant that multiplies with the regularization term. We experienced better results when using lower values (tending to 0) of alpha.

|  |  |
| --- | --- |
| **Alpha** | **Mean Accuracy** |
| 1 | 0.9617 |
| 0.1 | 0.9637 |
| 0.001 | 0.9652 |

1. **Performance Metrics for Regression Algorithms**

* Number of Iterations (iterations) - Higher number of iterations is longer run time. Since our dataset is comparatively very small, the number of iterations don’t make a significant impact on the result of the algorithm.
* Alpha (alpha) - It is a constant that multiplies with the regularization term. We experienced better results when using lower values (tending to 0) of alpha.
* Step Size (step) - 0.0001 step size gave us the optimum Mean Squared Error (mse). Increasing the step size gives infinite value of mse and decreasing the step size is increasing the mse.

|  |  |
| --- | --- |
| **Step Size** | **Mean Squared Error** |
| 1 | Nan |
| 0.1 | inf |
| 0.001 | 1.71171872242e+82 |
| 0.0001 | 4.25710987027 |

**4. Comparison between SciPy and Apache Spark**

To compare the libraries, we will check the precision using both the libraries and then will jump to a conclusion.

**Classification**

|  |  |  |
| --- | --- | --- |
|  | **SciPy** | **Apache Spark** |
| SVMWithSGD | 0.965288412455 | 0.9632503062474 |
| LogisticRegressionWithSGD | 0.96426748341 | 0.9532462229481 |
| LogisticRegressionWithLBFGS | 0.966309341501 | 0.9632503062474 |

**Regression**

|  |  |  |
| --- | --- | --- |
|  | **SciPy** | **Apache Spark** |
| LinearRegressionWithSGD | 0.289496074035 | 0.97936792766443 |
| RidgeRegressionWithSGD | 0.26694099672 | 0.97936792329444 |
| LassoWithSGD | 0.27 | 0.97936723510949 |

**Apache Spark**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Performance in terms of time is better | Very sparse community help and documentation |
| Has better algorithms and libraries for Big Data |  |

**SciPy**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Is nearly 10 times slower than Apache Spark | Good community support and documentation |
|  | Has better ML libraries but not suitable for bigdata tasks |

After analysing the pros and cons of Apache Spark and SciPy and also looking at the precision rates using both the libraries, we can say that in this case, **Apache Spark has won the race**.