**Data Science Project Report**

**1. Principal Investigator**

Alec Lehmphul – [lehmphua2@hawkmail.newpaltz.edu](mailto:lehmphua2@hawkmail.newpaltz.edu)

* 1. Individual Contribution Breakdown (list the percentage)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task | Member 1 | Member 2 | Member 3 | Total |
| Introduction | 100% | -- | -- | 100% |
| Background | 100% | -- | -- | 100% |
| Implementation | 100% | -- | -- | 100% |
| Experiment Results and Discussion | 100% | -- | -- | 100% |
| Conclusion | 100% | -- | -- | 100% |
| Other contribution and explain | -- | -- | -- | 100% |

**2. Title of Project**

Predicting Rain with Logistic Regression

**3. Mentoring**

Professor Min Chen, Department of Computer Science, SUNY-New Paltz

[chenm@newpaltz.edu](mailto:chenm@newpaltz.edu)

**4. Introduction**

**4.1 Project Motivation**

The goal of this project is to try and emulate a meteorologist (on a small scale). It’s on a small scale because my model will only be capable of predicting if it will rain tomorrow. It unfortunately cannot yet give you a full 7-day forecast. The overall intention is to produce a model that can correctly predict with 80-90% accuracy, just like a meteorologist.

**4.2 Aims and Objectives**

The overall aim and objective of the project is to learn about the data science field as a whole and effectively apply data science ideas within the project. I learned a lot about R programming, data processing, Apache Hadoop, logistic regression, and confusion matrix metrics.

Here are the steps that I used to organize and complete my project.

1. Data Pre-processing in R
2. MapReduce research and Java implementation
3. Testing the model using a testing dataset
4. Analysis using confusion matrix formulas
5. Think about possible improvements by reflecting on the project and its results

**4.3 Report Structure**

The final project contains a single R project (code for data processing) and five separate Java classes (code for training and testing the model). Three of the five Java classes are used for the MapReduce Job where the model is being trained. More specifically, these three classes are used in tandem to calculate and update the weights of the model. These weights will later be used for classification and testing the model. The three java classes are the map-reduce driver, mapper, and reducer classes. The remaining two Java classes are used for testing the model. More specifically, summing the number of correct and incorrect predictions as well as determining if it was false positive, true positive, false negative, or true negative. This information is vital for testing the effectiveness of the model. This is further elaborated on in the **Approach and Implementation (6)** and **Experiment Results and Discussion (7)** sections of the report.

**5. Background/History of the Study**

It is no secret that weather can be predicted. There is an entire field of study that is obsessed with this, meteorology. Meteorology has been around for centuries; it’s even older if you include the Greek philosophers and thinkers observing the atmosphere and its patterns. Meteorology and predicting weather were not technically a science until 1600s, when Torricelli invented the barometer. This device accurately measured the pressure of air and is still a key instrument in understanding and forecasting weather systems[3]. From then to the present day, many other instruments were invented to aid in weather forecasting. The thermometer, anemometer, weather satellite, and weather radar are some of these key tools. I hope that by the end of this project, I can emulate a meteorologist by being able to accurately (80%-90%) predict if it will rain with my model.

**6. Approach and Implementation**

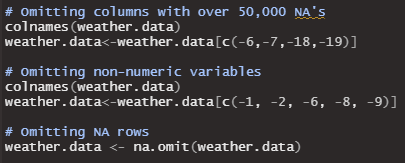
My approach was to divide the project into three main steps/tasks that must be completed: (1) Data Pre-processing, (2) Map-Reduce, and (3) Testing Model.

**Data Pre-Processing:**

This step was the most time consuming and important step to get right. The goal of data pre-processing is to properly clean the dataset so it can be usable and effective in modelling a model after it. With unclean data, your machine learning algorithm will never be able to produce an effective model.

Omitting Missing Values

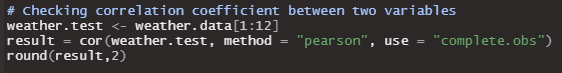
In order to clean the dataset, I first checked the number of missing values in each independent variable and omitted every independent variable containing over 50,000 missing entries. Next, I removed non-numeric independent variables to simply the model as well as the machine learning algorithm. Finally, any row containing a missing value was omitted from the dataset.



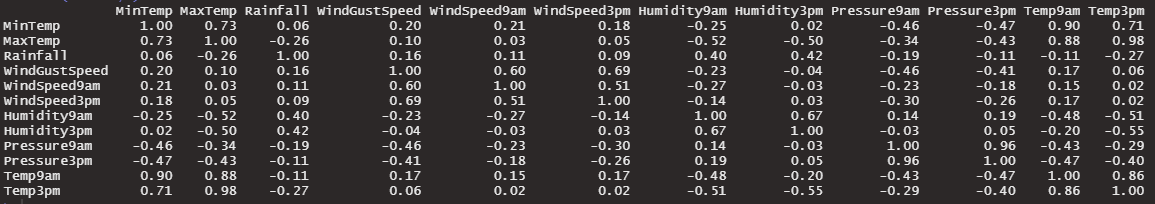
*Figure 1*: R code for omitting missing values

Correlation

The next step was to check for highly correlated variables and remove them from the dataset. For determining correlation, I used the Pearson method for finding the correlation coefficient. The larger the magnitude of the correlation coefficient, the higher the correlation between the two variables. In my case, there were a few highly correlated variables. Minimum temperature and temperature at 9am were highly correlated with a correlation coefficient of 0.90. Maximum temperature was also highly correlated with temperature at 3pm; their correlation coefficient was 0.98. Although these variables were highly correlated, I did not omit them from the dataset because I thought that they would be important for the model.



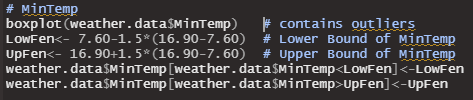
*Figure 2*: R code that creates correlation coefficient matrix



*Figure 3*: Correlation coefficient matrix

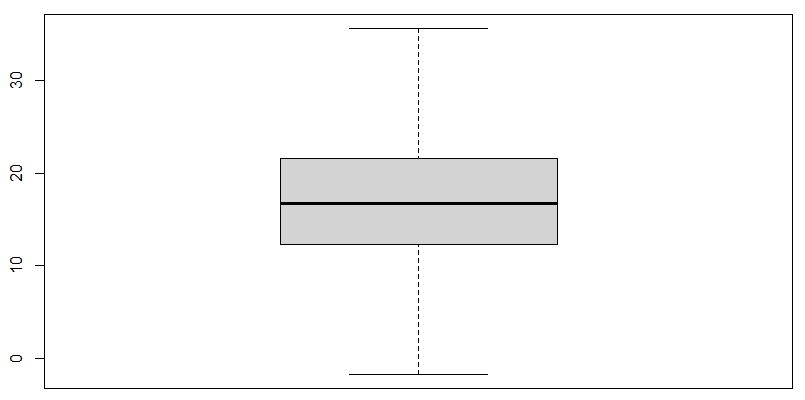
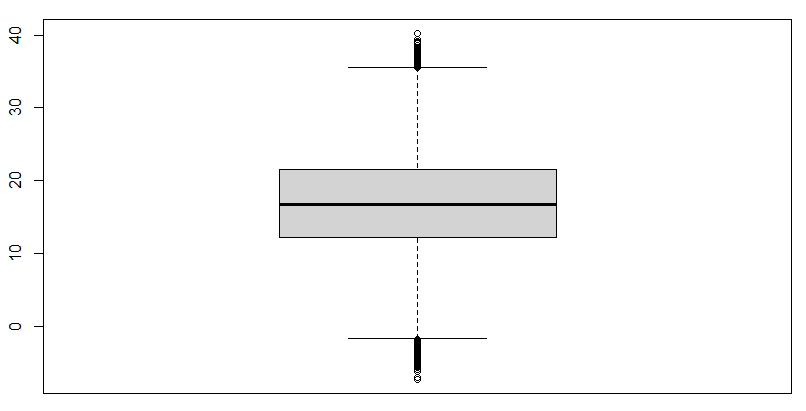
Winsorization

The next step was to figure out an effective way to deal with outliers. Because nearly a third of my data was outliers, simply omitting them did not seem like an option. I feared that I would not have enough data to properly train and test the model. So instead, I used the method of winsorization to limit the extreme values and mitigate the possibility of outliers skewing the model. Essentially, if an outlier is greater than the 3rd quartile, then the outlier’s value is set the value of the 3rd quartile. Likewise, if an outlier is less than the 1st quartile, then the outlier’s value is set to the value of the 1st quartile. *Figure 4,* which shows the implementation of winsorization in R, was repeated for every independent variable in the dataset containing outliers. For a graphical representation of winsorization, see *Figure 5*.



*Figure 4*: R code of winsorization for a single independent variable

Before After

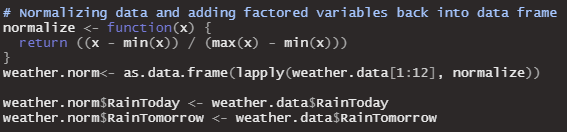


*Figure 5*: Graphical representation of winsorization

Normalization

The goal of normalization or normalizing the data is to change all of numeric columns in the data set to a common scale. Before I normalized my data, my logistic regression algorithm was inconsistent. This was due to the scale of “pressure at 9am” and “pressure at 3pm” were much higher than the rest of the dataset. When my logistic regression algorithm was calculating weights, both pressure variables dominated the model. The linear normalization formula was used to normalize each variable and solve this issue.

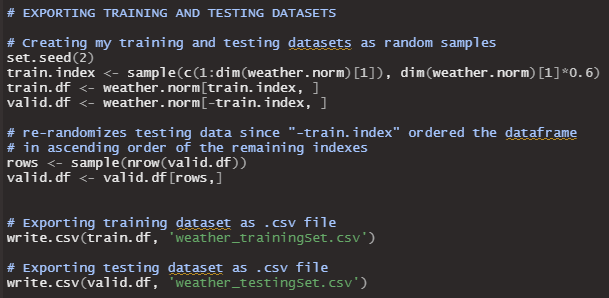
*Figure 6*: Linear normalization formula



*Figure 7*: R code of normalization

Creating Training and Testing Datasets

The final step of data pre-processing is to create usable training and testing datasets for the machine learning algorithm to use. To create these datasets, I used a R function to randomize the rows of my pre-processed dataset and assigned 60% of the randomized rows to a data frame called “train.df”, which will become the training dataset. I assigned the remaining 40% to another data frame called “valid.df”, which will become the testing dataset. Then, I exported both files with the comma-seperated values format so both datasets could then be read within a Java program.



*Figure 8*: Randomizing data and exporting as training and testing dataset

**Map-Reduce:**

Before I describe my approach and implementation of map-reduce within my model, I first need to describe logistic regression.

Logistic Regression

Logistic regression[5] is a machine learning algorithm that is perfect for classifying a binary outcome. This is due to the sigmoid function, which is the key behind the logistic regression algorithm. Essentially, the sigmoid function takes a linear model and maps it between the values of 0 and 1. It turns a linear function into probability. The formula for the sigmoid function is as follows:

where,

(linear function)

When researching logistic regression, many sources mentioned logit or log-odds. Logit is important because the sigmoid function can be derived from it. The formula for the logit function is as follows:

where,

The inverse of logit is equal to the sigmoid function, as shown below.

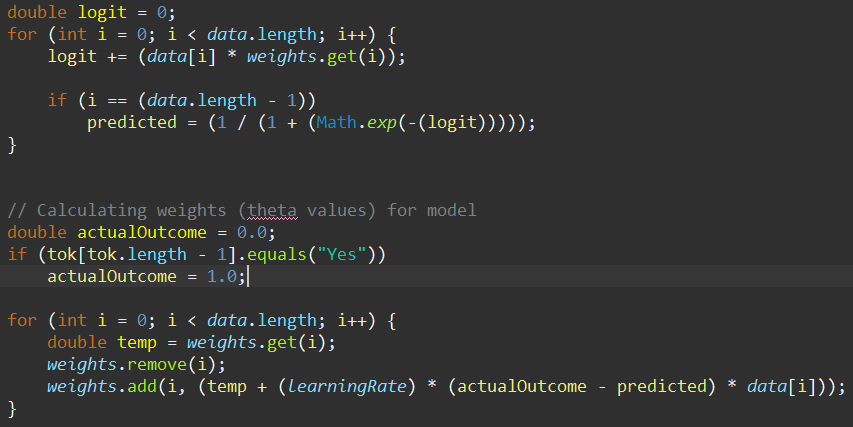
Map-Reduce Goal

The goal of map-reduce job is to train the model by using the training dataset to calculate the weights for the model. After the map-reduce job finishes, there will be a text file that contains the model’s weights. This text file will be used, along with the testing dataset, to test the model and its performance.

Mapper Class [6]

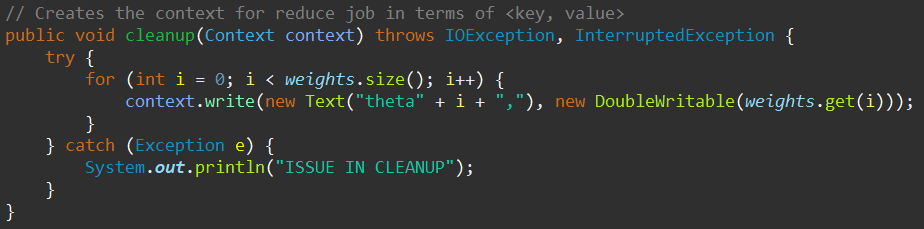
The mapper class parses through the training dataset file, which is determined in the configuration of the driver class. This is a super important step because the data needs to extracted properly so that the algorithm can correctly estimate and calculate the weights. The goal of the mapper class is to utilize the sigmoid function, as well as a learning rate, to update the weight of each independent variable within the model. The formula that algorithm uses for calculating new weights is shown below.[1]

Each weight value calculation is sent to the reducer class as a <key, value> pair of <(Text) theta, (doubleWritable) weight calculation>. *Figure 9* is a snippet of code from my Mapper class that shows the use of the new weight equation above as well as the use of the sigmoid function.



*Figure 9*: Calculation of weights in Mapper class

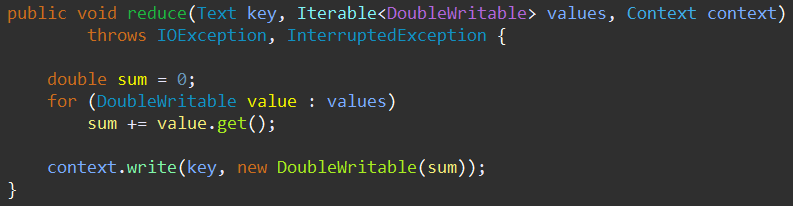
After all of the mapper jobs are fully completed, the clean-up function is then called which sends the intermediate <key, value> pairs to the reducer class to handle.



*Figure 10*: Clean-up function

Reducer Class

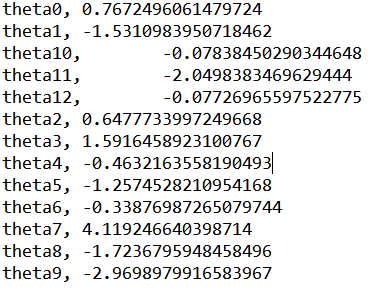
The goal of the reducer class is to output the proper weight value of each independent variable and write it to an output file. In my case, this was done simply through summing all of the mapper jobs’ values and using the context.write(key, value) function to write it to an output file.



*Figure 11*: Reduce function

Output of Map-Reduce Job

As mentioned before, the Map-Reduce job’s goal was to train the model. The output file contains the trained weights for the model which can be used for testing in the future.



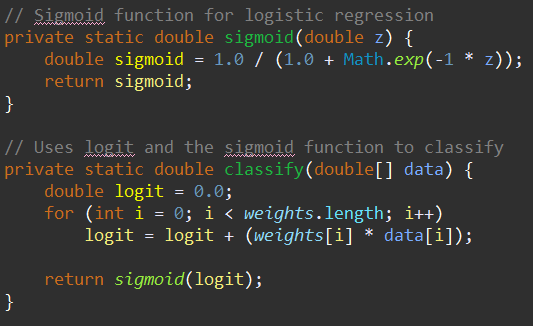
*Figure 12*: Output file from Map-Reduce

The weights seem to be weighted correctly because the theta value with the largest weight is theta7 which corresponds to the independent variable “humidity at 3pm”. If it will rain tomorrow, then the day before would most likely have a high humidity. Inversely, the theta value with the smallest weight is theta9 which corresponds to the independent variable “pressure at 3pm”. This makes sense because usually high air pressure means fairer weather.

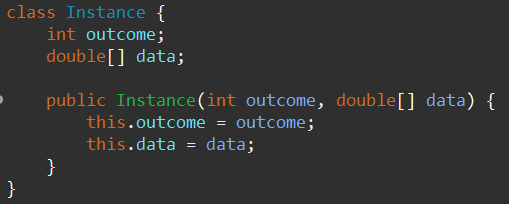
**Testing Model:**

To test the model, I created two separate Java classes. The first Java class, called “TestingHadoop.java”, parses both the testing dataset and Map-Reduce output file, as well as, classifies and predicts each instance of the testing dataset. The second Java class, called “Instance.java”, is a short class used for creating instances of the dataset (where each instance holds the outcome of whether it will rain tomorrow and an array of the data within that same row).

The following figures are snippets of code that show important functions used for testing the model.



*Figure 13*: TestingHadoop.java code that shows use of sigmoid function for classification



*Figure 14*: Instance.java

By testing the model in the way shown in *Figure 15* below*,* the program can determine the number of predictions that were:

1. Correct
2. Wrong
3. True Positive
4. True Negative
5. False Positive
6. False Negative

These metrics are important for testing the effectiveness of the model which will be shown in the next section.



Figure 15: Code of classifying and predicting each test instance

**7. Experiment Results and Discussion**

To test the effectiveness of the model, I will use the confusion matrix formulas of recall, precision, and accuracy. For each metric mentioned in the previous section, their totals were:

Correct = 40,014 Wrong = 7,822

True Positive = 4,373 True Negative = 35,641

False Positive = 6,066 False Negative = 1,756

Recall

Recall measures the model’s ability to find all of the relevant cases within a dataset In layman’s terms, “when it’s actually yes, how often does the model predict yes?”

Precision

Precision measures the model’s ability to return only relevant instances within a dataset. In layman’s terms, “when it predicts yes, how often is it correct?”

Accuracy

Accuracy measures the how often the prediction is correct.

All of these statistical measures were researched and taken from [4].

Of these three statistical measures, precision did the weakest. Although my model was quite accurate, it was not too great at predicting when it will rain tomorrow. Instead, my model was very good at predicting when it will not rain tomorrow. This is evident from the large number of true negatives and small number of false negatives.

**8. Conclusion**

Possible Improvements

No model is perfect and my model is no exception. Here are a list of possible changes and improvements that can increase the effectiveness of the model.

1. **Utilize dates to take into account seasons**. Currently my model does not take into account seasons. This is a big flaw because all of the seasons are different and have many different characteristics. If season was taken into account, then I believe the model would perform much better.
2. **Figure out a way to include the non-numeric variables such as wind direction and location.** These variables were removed to simplify the model. These variables could be valuable within the model. For example, maybe a front that causes a storm always comes from the western coast of Australia and causes the wind direction to always blew in from the west. Another example could be that locations close to the shores of Australia experience rain at a more frequent rate compared to the inland locations.
3. **There needs to be a greater balance in the number of rainTomorrow = “Yes” and rainTomorrow = “No” within the training dataset to properly expose the model to more instances of rainy conditions.** I fear that this happened in my model. It was not exposed to enough rainy days to learn what conditions caused rain.

Conclusion

This was a very fun project to work on. I learned so much in so many new areas, such as R, Apache Hadoop, data cleaning, and machine learning (with logistic regression) just to mention a few.

**9. References**

[1] Punit Naik. 2018. MLHadoop/LogisticRegression\_MapReduce. <https://github.com/punit-naik/MLHadoop/tree/master/LogisticRegression_MapReduce>. 2021

[2] Matthieu Labas. Terry. 2016. tpeng/logistic-regression. <https://github.com/tpeng/logistic-regression>. 2021.

[3] National Geographic Society. “Meteorology.” *National Geographic Society*, National Geographic, 9 Oct. 2012, [www.nationalgeographic.org/encyclopedia/meteorology/](http://www.nationalgeographic.org/encyclopedia/meteorology/) .

[4] *Narkhede, Sarang*. “Understanding Confusion Matrix.” *Medium*, Towards Data Science, 14 Jan. 2021, [www.towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62](http://www.towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62) .

[5] *Logistic Regression*, **faculty.cas.usf.edu/mbrannick/regression/Logistic.html**.

[6] *Mapper (Apache Hadoop Main 2.7.4 API)*, 29 July 2017, **hadoop.apache.org/docs/r2.7.4/api/org/apache/hadoop/mapreduce/Mapper.html**.