Machine Learning and Applications (MLAP)

Open Examination - Report

Exam Number: Y6189686

**This report accompanies my code submission. For both parts, Python 2 was used.**

# **Part 1 – Linear and Logistic Regression**

The code for this part is located in a folder called Regression, in a file called *regression.py*. The folder also includes all the data that accompanies this part. My submission makes use of some basic functions and data structures from the SciPy [2] Python library, and NumPy, which is part of SciPy.

## **Task 1**

To accommodate for this task, I have implemented linear regression in the Python function called *linear*. My algorithm starts by converting the input data file into a NumPy ndarray, starting from the 11th row, and adding the stock price and stock volume information from the previous 10 rows (accomplished in the *read\_data\_file* function). This creates a matrix with the size N - 10 by 21 (where N is the total number of rows). The first 10 columns contain the stock volume for each of the previous 10 days, while the next 10 – the stock price. The 21st column of this matrix contains 1’s only, and represents **x0** (easier to handle as last location). In parallel to this, a NumPy vector containing the actual price values for each day is also generated (also of length N - 10).

The stock price and volume matrix (labelled as **X** in the code) is then passed to the *feature\_selection\_financial\_data* function, which performs basis expansion, incorporating various features into the matrix. Data standardization is performed next, in the *standardize\_data* function. The latter performs location and scale transform on each feature (apart from the final one, which contains **x0**, as mentioned previously). Just location transform is also applied on the actual price values (labelled as **y** in the code).

Next, **X** and **y** are split into two randomly selected halves, implemented in the *split\_data\_random* function. Thus, we have two folds to use in cross-validation – two pairs of X and y: **fold\_1\_x, fold\_2\_x, fold\_1\_y, fold\_2\_y**. The algorithm performs cross-validation by using one pair of **X** and **y** for training, and the other for validation, and vice-versa.

The functionality for training and validation is implemented in a separate function, called *find\_theta\_linear*. The function first applies the L-BFGS-B minimization algorithm[[1]](#footnote-1) (part of SciPy [2], executed through the minimize function), to find optimal values for theta, through minimizing the *squared\_loss* function (defining the loss function for linear regression) with the training fold for x and y. Then, theta is used to calculate the mean squared loss for the validate dataset. This procedure is applied for the other combination of training and validate data.

Finally, the algorithm calculates the average mean squared loss, by averaging the validate mean squared loss for the two folds.

I will now discuss the results from executing the algorithm with the sample dataset of stock information. In order to better present my results, I introduce the following notation. Referring to **pi** throughout this part of the assessment would be equivalent to referring to the price for day **i**, where **i** is the day number, with **p0** representing the stock price in the previous day, and **p9** – in the 10th previous. Equivalently, **vi** will represent the volume for day **i**.

My results from feature experimentation with the algorithm on the stock data are outlined in Table 1 below, with 9 feature sets presented. The features that were used with each execution are enlisted in the middle column (the relevant theta and x0 for each feature is omitted for brevity). The average mean squared error that was achieved (from the cross-validation of the randomly split data) is enlisted in the right column. The table is sorted by the MSE error. Figure 1 also contains a bar chart of the MSE values of the target features (with the feature number on the y axis, as in the table).

|  |  |  |
| --- | --- | --- |
| **№** | **Features** | **Average MSE** |
| 1 |  | 6.34 |
| 2 |  | 2.11 |
| 3 |  | 1.051 |
| 4 | 79 features in total | 0.681 |
| 5 |  | 0.4809 |
| 6 |  | 0.416 |
| 7 |  | 0.386 |
| 8 |  | 0.385 |
| 9 |  | 0.384 |

Table 1: Linear Regression Feature Experimentation

The results achieved indicate that complex combinations of features (feature set 1 for example) perform much worse compared to simple feature sets. The 3 best results that minimize the MSE (sets 7, 8 and 9) were achieved with only linear features. The best result (set 9) only contains information of the stock price for the previous 3 days – the stock volume is not used. This leads to suggest that the stock price data provides sufficient guidance for the price for the current day – and, that the price of the previous day is very important in modelling this.

It is important to note that due to the fact that my algorithm shuffles the data, there would be some small variations in MSE with multiple executions. Hence, it is inconclusive which of the last three models is most appropriate, but all of them should achieve a similar performance. According to Occam’s razor, the 8th feature set should be preferred, however intuitively the 9th seems to be a better choice since it encapsulates more knowledge about the price.

Figure 1: Linear Regression MSE Bar Chart

## Task 2

To accomplish this task, I have implemented a multi-class logistic regression classifier in the Python function *logistic*. My algorithm starts similarly to linear regression. The data file is read through the *read\_data\_file* function, generating the same matrix for **X** as in the previous task. For **y**, however, instead of using the stock price for the current day, the class for the day is computed through measuring the difference in price between the current and previous days. This is accomplished in the *compute\_class* function, which returns an integer between 0 and 4, representing the computed class. There are 5 classes as defined in the assessment document.

Similar to the linear regression algorithm, basis expansion is performed (controlled also by the *feature\_selection\_financial\_data* function). The data in **X** is standardized, and split into two randomly selected folds (**fold\_1\_x, fold\_2\_x, fold\_1\_y, fold\_2\_y**), after which cross-validation is performed.

As for the previous task, the functionality for performing training and validation is realized in a separate function - *find\_theta\_logistic* in this case. This function begins by initializing an empty theta for each of our 5 classes (creating a matrix with the size 5 by M, where M is the number of features). Next, the L-BFGS-B minimization algorithm is applied with our training data to find optimal values for theta – by making a call to the SciPy *minimize* function (similar to the previous task). L-BFGS-B minimizes the logistic cost function, implemented in *compute\_cost\_logistic*. The latter computes and returns negated values for the logistic cost and gradient, since SciPy’s *minimize* is a minimizing function.

The output values for theta are used to compute the accuracy of my classifier on the validate dataset. This is accomplished in the function *logistic\_accuracy*, which reports the hard classification accuracy on the target validate dataset. It compares computed class estimates with the actual classes of the fold. Finally, the algorithm finishes by reporting the average validate accuracy for the 2-fold cross validation, by averaging the accuracies achieved for the two datasets. The accuracy reported is between 0 and 1, with 1 being 100% accurate.

My results from feature experimentation with the algorithm on the stock data are outlined in Table 2 below, with the same 9 feature sets presented, as for Task 1. Figure 2 displays a bar chart with the average accuracy achieved by the feature sets.

|  |  |  |
| --- | --- | --- |
| **№** | **Features** | **Average Accuracy** |
| 1 |  | 0.86001 |
| 2 |  | 0.856 |
| 3 |  | 0.859 |
| 4 | 79 features in total | 0.852 |
| 5 |  | 0.8525 |
| 6 |  | 0.8564 |
| 7 |  | 0.8604 |
| 8 |  | 0.861 |
| 9 |  | 0.8604 |

Table 2: Logistic Regression Feature Experimentation

Figure 2: Logistic Regression Accuracy Bar Chart

Although the feature set performance seems to be in a similar ordering as for Task 1, all of the accuracies presented are very similar, with the difference between the best and the worst accuracy being less than 0.01. In fact, extensive experimentation with various other feature sets did not yield any success in increasing the accuracy beyond 0.861. In my opinion, this is due to two factors. The first is that, looking at the class spread for our data, one can identify that about 87% of the data is class 0 (or the price difference between current and previous days is within ±5%). This leaves the other 4 classes greatly underrepresented, and limits our capacity to learn accurate predictors for them.

The second is that the current dataset on stock price and volume isn’t sufficient to reach to accurate predictors about the last 4 classes – and more information other than stock price and volume may be required in order to effectively differentiate between them. This was indicated by examining sample computed classes (or, computed **y**) with different feature sets. Even when a prediction was made for a class different to class 0, it was often a wrong prediction.

Hence, the classifiers presented have been successful in predicting class 0, but are usually unsuccessful in predicting any other class.

## Task 3

To accommodate for this task, I extended my linear and logistic models to implement ridge (L2) regularisation. This was realized in two Python functions – *reglieanr* and *reglogistic.* Regularisation code was added to the relevant cost functions – they accept a Boolean parameter, which indicates whether regularisation should be applied. The procedure for executing the algorithms is identical as in Task 1 and Task 2. Further to that, a function called *lambda\_test­* was implemented to support the process of discovering suitable lambda values. It executes a number of repeating experiments with different lambda values and averages over each MSE or Accuracy value that was achieved with every Lambda, for a specific set of features. The averaging was implemented in order to get a more accurate estimate – as mentioned earlier, the data is shuffled on each algorithm execution, and hence there are small variations in MSE/Accuracy values with each execution.

A plot for the progression of λ for *regularised linear regression* on feature set 9 (see Table 1) is presented in Figure 3. This was produced from executing 100 experiments, with λ varying from 0.05 to 1, and then averaging on the results (as mentioned above). The best result was achieved with λ=0.65, which is also the closest to the 0.5 balance between model complexity and fit. The MSE error that was achieved at this point is ~0.396.

Figure 3: Progression of Lambda for Feature set 9 – Regularised linear regression

As a further example for regularised linear regression, consider Figure 4, which depicts a similar progression of λ for feature set 4 (which has 79 features and is very complex). For it, a very small value of λ such as 0.05 or smaller would be preferred, which would greatly simplify our model, and reduce the MSE achieved.

Figure 4: Progression of Lambda for Feature set 9 – Regularised linear regression

As for *regularised logistic regression*, the results were much more ambiguous. Figure 5 below depicts the accuracy change for different λ values, for the 9th feature set. This result was produced from averaging 5 identical accuracy progressions, to accommodate for the shuffled data. The accuracies that were achieved are quite similar to each other, and although there are spikes such as for λ=0.75, it is inconclusive whether this is due to chance or to an actual improvement. As mentioned previously in Task 2, there seem to be some problems with the data, which prevent from achieving a substantial improvement in the accuracy.

Figure 5: Progression of Lambda for Feature set 9 - Regularised logistic regression

As an additional example for regularised logistic regression, consider Figure 6, displaying a similar progression as in Figure 5. In this figure, the accuracy steadily decreases as λ is increasing, with the minimum achieved when only the fit of the data is considered (λ=1). My interpretation of this result is that, when λ is small, the class predictors that are trained tend to predict class 0 almost 100% of the time. As λ increases and the complex model begins to be given preference, a loss in accuracy is observed due to an increase in class predictions that are incorrect.

Figure 6: Progression of Lambda for Feature set 9 - Regularised logistic regression

# **Part 2 – Bayesian networks**

The code for this part is located in a folder called Bayesian\_networks, in a file called *bn.py*. The folder also includes all the data that accompanies this part. My submission makes use of some basic functions and data structures from the popular NumPy Python library, which is part of SciPy [2].

## Task 4

To accomplish this task, I have implemented the function *bnbayesfit* to estimate the parameters of a Bayesian network. The function starts by reading the two input files, *structure\_file\_name* and *data\_file\_name*, which contain the structure of the Bayesian network, and the sample data, respectively. Reading is accomplished in a separate function, called *read\_data\_file*, which returns a NumPy ndarray of integers (containing 0s and 1s) from the two CSV files.

The *bnbayesfit* has actually been developed with flexibility in mind, thus it also accepts a NumPy ndarray as data, instead of a data file. This has been useful, as I’ll discuss later.

The function then proceeds with estimating each parameter of the Bayesian network independently. The functionality to estimate a parameter is implemented in the *estimate\_parameter* function. The latter uses a Bayesian approach to estimate each parameter, with a uniform prior of α=1 and β=1. This was implemented as described in Section 9.4 of [1] (page 199). The *estimate\_parameter* function simply counts the data, similar to MLE, and adds the uniform Beta prior. Counting conditional probabilities is more difficult, hence it has been implemented in a separate function *calculate\_conditional\_prob.*

The output of *bnbayesfit* is a dictionary, with each variable index as the key, and its probabilities as the value. Because there are multiple values for conditional probabilities, their values are organised through enclosing them in a OrderedDict (an ordered dictionary structure, part of the core Python library). Each item in this OrderedDict is a pair of conditions on the variable, along with the actual probability which was estimated with this combination of conditions.

The output from *bnbayesfit*, applied on the structure and data files (bnstruct.csv and bndata.csv) is shown below (the function formats all the calculated probabilities for display).

*Figure 7: Output from bnbayesfit, containing all probabilities for bstruct.csv.*

The final output structure from bnbayesfit, which contains the fitted Bayesian network, is shown in Figure 2.

*Figure 8: Output structure from bnbayesft – fitted Bayesian network.*

## Task 5

To accommodate for this task, the required *bnsample* function was implemented, to sample from a probability distribution, as defined by the *fittedbn* input parameter. My implementation uses Ancestral sampling (as defined in slide 8 of the MLAP Markov Chain Monte Carlo lecture and 27.2 of [1]) to generate *nsamples* number of samples (*nsamples* is the second parameter to *bnsample*).

Ancestral sampling is implemented in the *ancestral\_sampling* function, which returns a single sample. The procedure begins by sorting the input Bayesian network by length (in this way, placing variables with fewer conditions in the beginning). It maintains a dictionary of already sampled variables with their value, which is used in estimating conditional probabilities for other variables (the function *conditional\_sample* deals with this). In addition, a list of variables which couldn’t be estimated (because of unknown values for variables) is also maintained. In the end, if the sample doesn’t contain estimates for all variables, a recursive call is made to ancestral*\_sampling* such as to go over the BN again, and attempt to estimate any outstanding variables.

A sample output for *nsamples*=10 is presented in Figure 3 below. The output is a

*Figure 9: Output from bnsample with nsamples=10.*

One can use this approach to produce estimates for any conditional probability defined by the Bayesian network – the functionality for this is in the *conditional\_sample* function. The necessary prerequisite is to already have estimated the values of the parents – or to have their values as evidence. Then, one can use the already estimated probabilities for variables from data to draw a sample for the target conditional probability (or, rewrite the probability in terms of the known probabilities using Bayes rule).

Consider estimating the following probability from the Bayesian network, defined in *bnstruct.csv*: (0 and 1 are variables). The sampler would first need to estimate a value for – this can be accomplished by drawing a random number between 0 and 1, and comparing its value to the probability for 1 (which is ~0.94 for our BN). If the random value is smaller than the probability, we set the value of 1 to True, and use this to estimate the value of – by using the relevant probability estimate. This gives us a sampled value for 0.

This can be done efficiently, when values for all parents can easily be estimated. It is inefficient, when estimating probabilities for multiple variables, which are dependent (as explained in section 27.2.1 in [1]). The latter case means that forward sampling would get very complex, having to find alternative representations.

# References

|  |  |
| --- | --- |
| [1] | D. Barber, Bayesian reasoning and machine learning, Cambridge University Press, 2012. |
| [2] | E. Jones, T. Oliphant and P. Peterson, “SciPy: Open source scientific tools for Python,” 2001--. [Online]. Available: http://www.scipy.org/. |

1. <http://docs.scipy.org/doc/scipy-0.13.0/reference/generated/scipy.optimize.minimize.html> [↑](#footnote-ref-1)