Machine Learning and Applications (MLAP)

Open Examination - Report

Exam Number: Y6189686

**This report accompanies my code submission. For both parts of the assessment, Python 2 was used to implement the required solutions.**

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

The code for this part is located in the folder Regression, in a file called *regression.py*. The folder also includes the CSV data file, which accompanies this part - *stock\_price.csv*. My algorithm implementations make use of some basic functions and data structures from the SciPy [2] Python library, and NumPy, which is part of SciPy. Prior to getting to implementation details and results, I discuss the data preparation steps that have been performed to support each of the algorithms.

### **Data preparation and pre-processing**

All of the algorithms to be described begin their execution by reading the input data file, the functionality for which is accomplished in the *read\_data\_file* function. This function converts the stock data into a NumPy ndarray, starting from the 11th row, and adding the stock price and stock volume information from the previous 10 rows to each row, as specified in the assessment. This creates a matrix, labelled as **X** throughout the code, 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 (starting from earlier days), while the next 10 – the stock price. The 21st column of this matrix contains 1’s only, and represents **x0** (the bias, easier to handle as last location).

In parallel to this, a NumPy vector labelled **y**, containing the actual price values for each day is also generated, which is also of length N - 10. For logistic regression, instead of using the stock price, the class for the day is computed through measuring the difference in price between the current and the previous day. 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.

After this step, **X** is passed to the *feature\_selection\_financial\_data* function (the same for both linear and logistic regression), which performs basis expansion, incorporating various features into the matrix. Next, data standardisation is performed on each feature of the data – location and scale transform is applied on each column, apart from the final one, which contains **x0**. Standardising the data significantly speeds up algorithm executions by scaling up the stock price and volume, or any newly created features, to the same scales. This is actually also necessary for logistic regression, since experimentation with complex features without standardisation often resulted in a floating point underflow or overflow, even when using *logsumexp*. Data scaling also has some effect on the errors/accuracies that are achieved – they can potentially be scaled up themselves, although this shouldn’t have a major effect on the final results from experiments.

For linear regression, just location transform is also applied on the **y** vector, which contains the actual stock prices for the day.

A final step in data preparation is splitting the data into two halves, to be used in the 2-fold cross validation performed by the algorithms. The function *split\_data\_random* accomplishes this, by randomly splitting the data in two equally-sized halves through shuffling. Hence, in the end we have two folds to use in cross-validation – or two pairs of X and y – called: **fold\_1\_x, fold\_2\_x, fold\_1\_y, fold\_2\_y**. The algorithms perform cross-validation by using one pair of **X** and **y** for training, and the other for validation, and vice-versa.

## **Task 1**

To accommodate for this task, I have implemented linear regression in the Python function called *linear*. My algorithm starts by preparing the data and performing basis expansion, as described previously. Then it performs 2-fold cross validation using a pair of **X** and **y** for training, and a pair for validation, and then switching pairs.

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 (part of SciPy [2], executed through the *minimize*[[1]](#footnote-1) function), to find optimal values for theta, through minimizing the *squared\_loss* function (defining the squared loss function for linear regression, as in lecture slides) through using the training fold for x and y. Then, theta is used to calculate the mean squared loss (MSL or MSE – mean squared error) for the validate dataset. This procedure is also applied for the other combination of training and validate data. Finally, the algorithm calculates the average mean squared loss that was calculated for the two validate datasets, by averaging the MSE for the two validate folds. This gives us the cross-validation error.

I will now discuss the results achieved 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 relevant feature, which represents the price for the **i**th previous day, where **i** is the day number, with **p0** representing the stock price for the previous day, and **p9** – for the 10th previous. Equivalently, **vi** will represent the volume for the **i**th previous day. For each data point, only **p0** to **p9** and **v0** to **v9** are available.

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 second column, separated by semicolon (the relevant theta for each feature of the models and x0 are omitted for brevity). The average mean squared error that was achieved (from the cross-validation of the randomly split data) is enlisted in the final column. The table is sorted by the MSE. Figure 1 also contains a bar chart of the MSE values of the target features (with the feature set number on the y axis, as in the table).

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature set №** | **Features** | **#**  **Features[[2]](#footnote-2)** | **Average MSE** |
| 1 |  | 5 | 6.34 |
| 2 |  | 6 | 2.11 |
| 3 |  | 6 | 1.051 |
| 4 | ( and are excluded) | 78 | 0.681 |
| 5 |  | 20 | 0.4809 |
| 6 |  | 10 | 0.416 |
| 7 |  | 5 | 0.386 |
| 8 |  | 1 | 0.385 |
| 9 |  | 3 | 0.384 |

Table : Linear Regression Feature Experimentation

The results achieved indicate that complex combinations of features (feature set 4 for example) perform worse compared to much simpler feature sets. Adding higher order features such as seems to increase the MSE. The 3 best results that minimize the MSE (sets 7, 8 and 9) were achieved with only linear features. The best result in this table (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 of the current day.

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, hence my preference is to the 9th model.

Figure : 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, with the data read, prepared and standardised identically to linear regression. For the **y** vector, however, instead of the stock price for the current day, the class for the day is computed as an integer between 0 and 4. The data is split into two folds, and 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 a slice of 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, estimation of which is performed on each of the two validate folds. This is accomplished in the function *logistic\_accuracy*, which reports the hard classification accuracy by estimating the class probabilities for an input θ, and by picking the highest estimate for each data point in an input dataset. It compares the computed hard estimates with the actual class values of the validate fold, and calculates the success ratio (or accuracy). Finally, the algorithm finishes by reporting the average validate accuracy for the 2-fold cross validation, by averaging the accuracies achieved with the two validate folds. The accuracy value that is 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, and a similar representation, 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 | ( and are excluded) | 0.852 |
| 5 |  | 0.8525 |
| 6 |  | 0.8564 |
| 7 |  | 0.8604 |
| 8 |  | 0.861 |
| 9 |  | 0.8604 |

Table : Logistic Regression Feature Experimentation

Figure : 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 to each other, 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%) – see Figure 3. 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 perhaps 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 (which lead to a decrease in accuracy).

Hence, the classifiers presented have been successful in predicting class 0, but are usually unsuccessful in predicting the other 4 classes. Further data/more information may be required in order to get clearer predictions on the class of stock price.

Figure : Class spread as calculated from the stock data.

## **Task 3**

To accommodate for this task, I extended my linear and logistic regression algorithms 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 also accept a Boolean parameter, which indicates whether regularisation should be applied. The procedure for executing the algorithms and performing cross-validation 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 (for example, 10 experiments with λ=0.5, followed by 10 with λ=0.55) 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 4. This was produced from executing 100 identical 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. It is arguable whether regularisation is necessary for this particular model, which is very simple, having only 3 linear features.

Figure : Progression of Lambda for Feature set 9 – Regularised linear regression with 2-fold cross-validation

As a further example for regularised linear regression, consider Figure 5, 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. In this case, regularisation has a positive effect on the model and results in a decrease of MSE from ~0.84 to about ~0.54.

Figure : Progression of Lambda for Feature set 9 – Regularised linear regression with 2-fold cross-validation

As for *regularised logistic regression*, the results were much more ambiguous. Figure 6 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. Regularisation doesn’t seem to improve my logistic regression models.

Figure : Progression of Lambda for Feature set 9 - Regularised logistic regression with 2-fold cross-validation

As an additional example for regularised logistic regression, consider Figure 7, displaying a similar progression as in Figure 6. 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 : Progression of Lambda for Feature set 9 - Regularised logistic regression with 2-fold cross-validation

# **Part 2 – Bayesian networks**

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

## **Task 4**

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

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 probability values for each of the condition combinations of conditional probabilities, they are organised through enclosing them in an 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 final data structure for *fittedbn*, which contains the fitted Bayesian network, is presented in Figure 9. The output from applying *bnbayesfit* on the assessment files has also been formatted in Figure 8 below, which contains the full list of final probability estimates.

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

*Figure 9: Formatted output from bnbayesfit, containing all probabilities for bstruct.csv. 1=0 is equivalent to variable 1 is False.*

## **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*) and return them as a NumPy ndarray.

Ancestral sampling itself is implemented in the *ancestral\_sampling* function, and returns a single joint sample of the probability distribution of the fitted Bayesian network. The procedure for sample generation begins by sorting the input BN by length, in this way, placing variables with fewer or no conditions in the beginning, which is desirable. This allows for sampling some of the variables straight away.

To support sampling of more complex combinations of variables, the function maintains a dictionary of already sampled variables with their value, which is used in estimating conditional probabilities (implemented in the *conditional\_sample* function). In addition, a list of variables which couldn’t be estimated (because of unknown values for variables at the time of inspection) is also maintained. In the end, if the sample doesn’t contain estimates for all variables of the BN, a recursive call is made to ancestral*\_sampling* such as to go over the BN again, and attempt to estimate any outstanding variables. This approach should work well with any BN that is acyclic (cyclic networks are not supported).

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



Figure : Output from bnsample with nsamples=10.

Hence, one can use this sampling 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. Then, one can use the probability distribution of the BN to draw a sample for the target conditional probability.

As an example of the estimating process, consider the approach for estimating the probability of (0 and 1 are variables, 0=1 is identical to 0=True) as defined by the input Bayesian network (with probability distribution outlined in Figure 9). The sampler would first need to estimate a value for – this can be accomplished simply by drawing a random number between 0 and 1 (variable 1 has no conditions), and comparing its value to the probability for variable 1=1 (which is ~0.056 as specified by our BN probability distribution). Let’s assume that the random value drawn is smaller than the probability. Then, we can set the value of variable 1 to True, which can in turn be used to estimate our original target . This is similarly accomplished by drawing another random number and by comparing it to the relevant probability estimate for this scenario (which is ~0.93). If this newly drawn value is less than 0.93, we set the final sampled value for variable 0 to True. This in turn allows for estimating of other variables that are children of variable 0 (such as variable 2).

Ancestral sampling can be performed 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 of [1]). The latter case means that sampling would get very complex, having to find alternative representations.

With respect to the final question of this task, consider figures 11, 12, 10 and 13, which present sample outputs from *bnsample*, executed with *nsample* of 3, 5, 10 and 20. Increasing the number of samples generates a more accurate representation of the probability distribution of the Bayesian network, since there’s more information about each of the variables. For example, according to Figure 11 (variable 1 is the second column) would be 0 (all the samples given are *False*), however in Figure 12, one of the samples has value 1 for variable 1, which changes our expectation for variable 1. As the number of samples increases, we get closer and closer to the original estimate from the previous task ().



Figure : Output from bnsample with nsamples=3.



Figure : Output from bnsample with nsamples=5.



Figure : Output from bnsample with nsamples=20.

# 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)
2. Excluding x0 [↑](#footnote-ref-2)