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 popular 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*, which accepts a single parameter – the input data file. My algorithm starts by converting the 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 BFGS minimization algorithm[[1]](#footnote-1) (part of SciPy [2]), to find optimal values for theta, through minimizing the *mean\_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.

In order to better present the results, I will introduce the following notation. Referring to **pi** would be equivalent to the price for day **i**, where **i** is the day, 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**.

## 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 with the *read\_data\_file* function, generating the same matrix for **X**. For **y**, however, instead of the stock price for the day, the class of 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.

Similar to the linear regression algorithm, basis expansion is performed (controlled by the *feature\_selection\_financial\_data\_logistic* function). The data 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.

Again 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[[2]](#footnote-2)* function. 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 our 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.

## Task 3

To accommodate for this task, I extended my linear and logistic models to implement ridge regularisation.

# **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’ve 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 trickier, hence it has been implemented in a separate function *calculate\_conditional\_prob.*

The output of *bnbayesfit* is a dictionary, with each variable as the key, and the probabilities as the value. Because there are multiple values for conditional probabilities, their value corresponds to an OrderedDict (an ordered dictionary structure, part of the core Python library), which contains probabilities for each condition combination. This structure is used in the next task.

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

*Figure 1: Output from bnbayesfit, containing all probabilities for bstruct.csv.*

The final output structure from the function is shown in Figure 2 below.

*Figure 2: Output structure from bnbayesfit.*

## 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 3: 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.fmin_bfgs.html> [↑](#footnote-ref-1)
2. <http://docs.scipy.org/doc/scipy-0.13.0/reference/generated/scipy.optimize.minimize.html> [↑](#footnote-ref-2)