

CS5011 P4

160004864

20 December 2019

1 Introduction

For this assignment we were required to construct and use an artificial neural network applied in a help desk ticketing system, as described in the specification. For the requirements of the assignment I've used the Python programming language and the scikit-learn library.

2 Implementation Checklist

For the purposes of the assignment I've implemented the following:

1. 'Basic Agent': The purpose of this agent is to create and train a neural network to learn the classification patterns for the given dataset contained in a CSV file named 'tickets.csv' which holds sample tickets with the response teams to where the tickets was routed to.
2. 'Intermediate Agent': The purpose of this agent is to use the neural network mentioned above, to identify the appropriate response team given a new request by the user. This agent is also able to learn from user preferences by updating the model used in case the prediction made was not satisfactory.
3. Additional Requirements/'Advanced Agent': This agent implements all the above with the addition of several key features:
 - Several classification algorithms are used and their performance on the classification task at hand is compared.
 - Another neural network is created and trained on a different dataset, which is used to predict the number of days needed for the ticket to be dealt with, information which is given to the user upon a ticket's routing.
 - During training, the agent also plots graphs showing the learning curves of the two neural networks.
 - The agent carries out a grid search in order to find the neural network hyperparameters that perform the best at the ticket routing task.

3 Setup and Running Instructions

3.1 Setup

I have provided a virtual environment, `venv`, as well as a `requirements.txt` file in my submission to make setting up the program possible with the following commands:

```
source venv/bin/activate
pip3 install -r requirements.txt
```

At this point, the virtual environment is activated, and the program's dependencies should be installed. In order to run the program, use the following command:

```
python3 A4main.py <Bas|Int|Adv>
```

4 Design, Implementation and Evaluation

Note: Due to the word limit set in the specification, the reader is encouraged to inspect the heavily commented source code provided to gain a deeper understanding of the structure and implementation details of the program.

4.1 Basic Agent

The functionality of the basic agent can be found in the `train_agent.py` file. The different steps required for its implementation are discussed below:

4.1.1 Loading Data

The first step to implementing the basic agent, was to load the data found in the CSV file, into the program. Using functionality found in the `load_data.py` module, the csv file is read in and then it is encoded in a way that it will be understandable by the neural network, on the fly.

4.1.2 Data Encoding

The second step in implementing the basic agent, was encoding the data loaded from the CSV file in a way that the neural network can use. Firstly, it's obvious that the Yes, No answers can be converted to 1 and 0, respectively. The response team however can be encoded in two ways. One way is using 'one-hot' encoding [1] and the other one is using integer encoding, in which a response team is simply identified by an integer. For reasons discussed in the evaluation section, I've used the latter encoding.

4.1.3 Neural Network

Seeing that this is a logistical regression task, I used sklearn's `MLPClassifier`, a multi layer perceptron that uses either LBFGS or stochastic gradient decent to optimise the log-loss function[2]. As required, this neural network will have one hidden layer, input units equal to the number of questions about the nature of the ticket and output units equal to the number of response teams that the ticket can be routed to. For the basic agent, the values of the hyperparameters used such as the number of neurons in the hidden layer, momentum, the alpha (reguralisation) parameter and the activation function used by the neurons, were manually chosen according to commonly used values and general recommendations, such as the ones found in the lecture slides for the number of units in the hidden layer[3].

4.1.4 Training the Network

Before training the network, it is necessary to split the data and its labels into a training set and a testing set. The former is used to fine tune the network using backpropagation and the latter is used to validate its performance. For this, I used sklearn's `train_test_split` method[4]. This returns four lists containing the data of the training set, data of the testing set, the labels of the training set and the labels of the testing set. Then, the network is trained on the training set, and its performance can be evaluated by predicting the appropriate response team labels for the, unseen by the network, testing dataset, and comparing those to the 'true labels' (i.e. the expected labels in `y_test`).

4.1.5 Saving the network

The trained network is then saved in a file named `basic_agent.joblib` which can be loaded and used by the Intermediate agent.

4.2 Intermediate Agent

The functionality of the proof of concept system which shows the use of the ticket routing agent is implemented in the `route_ticket.py` file.

4.2.1 Routing tickets

The program prompts the user to answer Yes or No (y or n) on whether the different tags are related to their ticket. This builds up the query (using 1 for Yes and 0 for No) which can be used as input to the neural network (loaded from the file mentioned above) which determines the appropriate response team. After logging the ticket, the system is ready to accept a new one, until the user decides to quit.

4.2.2 Early prediction

Answering to all of the questions every single time can be time-consuming and tiring. For this reason, a prediction is made by the neural network before all the answers have been given. After each answer, the probabilities of the ticket belonging to each response team, returned by the `predict_proba` method, is inspected. This represents the certainty which with the neural network predicts the response team mapping of a ticket and if it's within a certain margin, controlled by the `ALLOWED_ERROR` variable in the `settings.py` file, then an early prediction is made to which the user is prompted to answer y/n, indicating whether the prediction is correct. The neural network can perform the prediction on an unfinished query as the system pads the unfinished query with 0s (for the unanswered question) so that its dimension matches the number of input units.

4.2.3 Retrain network

If the user is unhappy with the response team prediction of the neural network, the program prompts the user to choose one out of the available response teams and this new ticket is then appended to the `tickets.csv` file, after the query has been decoded from its binary form to a string containing Yes, No and the name of the response team. Then, the neural network is retrained as done for the basic agent, only by using the newly updated CSV file. The user can then continue logging new tickets, as required.

4.3 Additional Requirements/Advanced Agent

In the sections below I will be outlining the additional requirements I've implemented:

4.3.1 Grid Search

While in the basic agent I described how the hyperparameters used for the neural network were chosen manually, I've chosen to carry out a grid search, which allows us to evaluate all the possible combinations of potential hyperparameter values using cross validation[6]. The hyperparameters that perform the best are then used when creating the multi-layer perceptron used by the advanced agent. The network is then trained and it is saved in the `adv_agent.joblib` file and it can then be used for routing new tickets, as described in the section for the intermediate agent.

4.3.2 Regression Task

In order to predict the number of days need for a certain ticket to be dealt with by a response team, it was first necessary to create a new dataset. To do so, I took the data from the `tickets.csv` file and I removed the last column (the

response team column) and saved in a different CSV file named `tickets_no_y.csv`. In a different, standalone python module named `curate_csv.py`, I've implemented functionality which takes the data, and first adds a new column named 'Time'. For each row of the data, the time taken is calculated by summing the weight of the tags that the answer was 'Yes' for. I've introduced weights as I assumed that tickets with different tags need different times. For example tickets related to incidents take fewer days to be dealt with that tickets related to id cards. Furthermore, using this methodology, it is assumed that the more tags the ticket applies to, the more time the help desk needs to deal with it. After the times are calculated, the new data is written in a new CSV file named `tickets_time.csv`. The process of training this network is similar to the network carrying out the classification task, but seeing that this is a regression task it was necessary to use sklearn's `MLPRegressor`[7] which outputs a single time value predicting the number of days needed for the ticket will be with. This network is then saved in the `time_agent.joblib` file, and it is used by the ticket routing system, for informing the user its prediction when a new query is logged.

4.3.3 Plotting Learning Curves

A common way of checking whether models have converged is to use learning curves. Similar to the methodology found here[9], the networks are trained several times on different sized subsets of the training set and then a graph of performance against the size of the training set is plotted. The results are as follows:

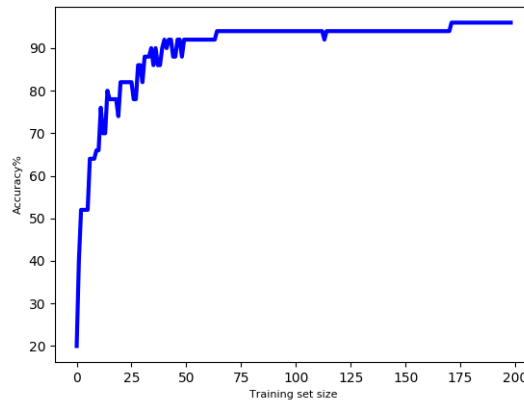


Figure 1: Learning curve of the MLPClassifier.

We can see that the MLPClassifier almost converges, which means that little could be gained if the training set contained more instances.

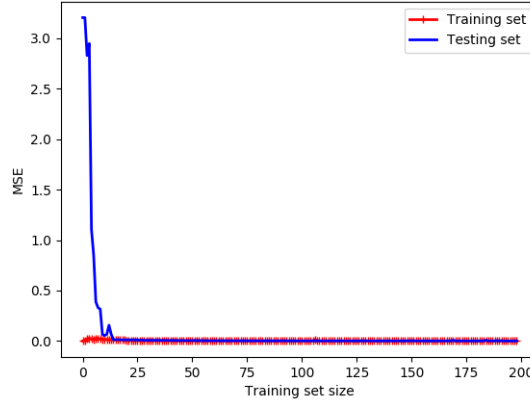


Figure 2: Learning curve of the MLPRegressor.

The performance metric used for this regression task is the mean squared error (MSE) and as we can see, this drops dramatically once the training set grows in size and it approaches zero very closely once the training set size reaches 20.

4.3.4 Classifier Comparison

For this functionality I've used a methodology similar to the one outlined by Jeff Delaney in one of his articles [8]. The comparison is made in the `classifier_comparison.py` module and it can be executed by running the `A4main.py` module with the `Comp` parameter. The module then instantiates a variety of classifiers including a Decision Tree, Random Forest classifier, K-Nearest Neighbours, SVM (Support Vector Machine) and the Gradient Boosting classifier. Details of each of the classifiers used can be found at sklearn's website[10]. These classifiers are then trained and tested using the same training and testing split and their performance is plotted and shown below:

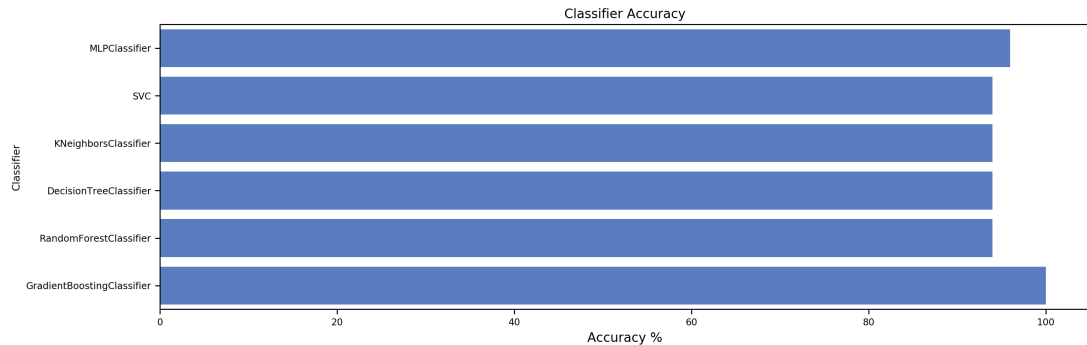


Figure 3: Accuracy scores of different classifiers.

As we can see, the MLPClassifier performs really well but it is outperformed by the Gradient Boosting Classifier which scores an astonishing 100% accuracy.

4.4 Evaluation

4.4.1 One Hot Encoding

We could have used a ‘one-hot’ encoding[1] in order to encode the dataset, with having each response team identified using a binary variable. However, this resulted in the MLPClassifier to carry out multi-label classification rather than multi-class classification and thus it sometimes identified two response teams for a ticket. This was obviously not what was required and for this reason I used integer encoding.

4.4.2 Retraining the Network

Instead of retraining the network from the beginning, I looked into whether the network could be trained ‘online’, that is the ability of the network to learn about new data on the fly[5]. This would be useful in cases where the data-set was very large and training the classification model would be time consuming. For the intents and purposes of this practical however, it was easier to use the existing infrastructure to train a new network with the updated dataset.

4.4.3 Learning Curves

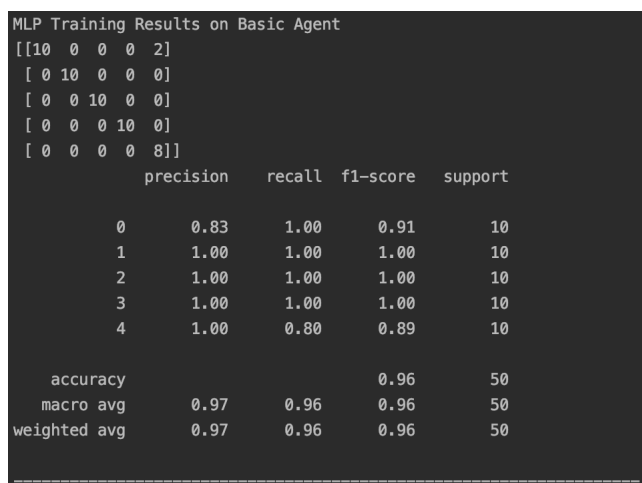
The reason as to why the MLPRegressor’s learning curve graph is this steep is probably due to the nature of the algorithmic way that the time column of the `tickets_time.csv` was populated and thus the neural network only needed a small training set to identify this. On the other hand, the MLPClassifier converges near a 100% accuracy but it’s not perfect, perhaps due to the fact that the data might be noisy and due to the non-linear nature of the model.

4.4.4 Grid Search

I have defined two parameter spaces for the grid search, a smaller one to satisfy timing constraints and a larger one which searches through far more combinations. The latter however is very time consuming so the smaller one is currently used for the intents and purposes of this assignment.

4.4.5 Performance

Some of the typical tools for evaluating how accurate a classifier is, is the use of classification reports and confusion matrices. These use the number of true and false positives and true and false negatives to calculate several performance metrics. The following is the confusion matrix and the classification report displayed when the basic agent is trained:



```
MLP Training Results on Basic Agent
[[10  0  0  0  2]
 [ 0 10  0  0  0]
 [ 0  0 10  0  0]
 [ 0  0  0 10  0]
 [ 0  0  0  0  8]]
```

	precision	recall	f1-score	support
0	0.83	1.00	0.91	10
1	1.00	1.00	1.00	10
2	1.00	1.00	1.00	10
3	1.00	1.00	1.00	10
4	1.00	0.80	0.89	10
accuracy			0.96	50
macro avg	0.97	0.96	0.96	50
weighted avg	0.97	0.96	0.96	50

Figure 4: Performance metrics of the basic agent.

As we can see, the model is very good with a very high 96% accuracy and it only misclassifies 2 tickets. This is evidence that the model generalises very well and it is very effective at the classification task it is intended to carry out. The following are the performance metrics displayed when the advanced agent is trained:


```

(venv) (base) Alexandross-MacBook-Pro:P4 alexandros michael$ python3 A4src/A4main.py Adv
Best parameters found:
{'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (7,), 'learning_rate': 'adaptive', 'momentum': 0.5, 'solver': 'lbfgs'}
MLP Training Results
<class 'sklearn.neural_network.multilayer_perceptron.MLPClassifier'>
[[10 0 0 0 2]
 [ 0 10 0 0 0]
 [ 0 0 10 0 0]
 [ 0 0 0 10 0]
 [ 0 0 0 0 8]]
      precision    recall  f1-score   support

         0         0.83      1.00      0.91         10
         1         1.00      1.00      1.00         10
         2         1.00      1.00      1.00         10
         3         1.00      1.00      1.00         10
         4         1.00      0.80      0.89         10

 accuracy          0.96         50
 macro avg          0.97      0.96      0.96         50
weighted avg          0.97      0.96      0.96         50

MLP Regression Training Results
<class 'sklearn.neural_network.multilayer_perceptron.MLPRegressor'>
Mean Squared Error: 4.621258098484139e-07

```

Figure 5: Performance metrics of the advanced agent.

As we can see the program first outputs the best hyperparameters resulting from the grid search and it then prints the performance metrics for the MLP-Classifer as well as the MSE of the MLPRegressor. Both networks seem to be performing extremely well, especially with the Regressor having a infinitesimal MSE.

5 Testing

Since testing a neural network is tied in with its performance metrics, in this section we'll only be concerned with the Intermediate agent and the correctness of its functionality. To convince the reader of the system's correctness I will be providing sample runs of the program with its outputs.

5.1 Logging a new ticket

This is the result of logging a new ticket to the system:

```
Welcome to the ticket routing agent.
Please enter y/n to answer the following questions.
Request?y
Incident?n
WebServices?y
Login?n
Wireless?n
Printing?y
IdCards?n
Staff?y
Students?n
Route the ticket to: Networking?y
Routing ticket to: Networking
Enter q to quit. Any other key to continue with an other key.q
Thank you for using the ticket routing service. Have a great day!
```

Figure 6: Logging a new ticket.

As we can see, the program correctly asks questions about the tags related to the ticket, and it makes a prediction and prompts the user to answer whether they're satisfied with the prediction. If yes, then it is routed to the predicted response team. If the user is not satisfied with the prediction, the following is displayed:

```

Welcome to the ticket routing agent.
Please enter y/n to answer the following questions.
Request?y
Incident?n
WebServices?y
Login?n
Wireless?n
Printing?y
IdCards?n
Staff?y
Students?n
Route the ticket to: Equipment?n
Where would you like the ticket to be routed to?
1 - Emergencies
2 - Networking
3 - Credentials
4 - Datawarehouse
5 - Equipment
Enter 1-5: 2
Yes,No,Yes,No,No,Yes,No,Yes,No,Networking
Retraining ANN
MLP Training Results on Basic Agent
[[10 0 0 0 2]
 [ 0 11 0 0 0]
 [ 0 0 10 0 0]
 [ 0 0 0 10 0]
 [ 0 0 0 0 8]]

```

	precision	recall	f1-score	support
0	0.83	1.00	0.91	10
1	1.00	1.00	1.00	11
2	1.00	1.00	1.00	10
3	1.00	1.00	1.00	10
4	1.00	0.80	0.89	10
accuracy			0.96	51
macro avg	0.97	0.96	0.96	51
weighted avg	0.97	0.96	0.96	51

```

-----
Enter q to quit. Any other key to continue with an other key.

```

Figure 7: Logging ticket and retraining model.

As we can see, the user is prompted to choose the team they wish the ticket to be sent to and then the network is retrained. After training it is ready to log new tickets.

Note: For these tests I have turned off early prediction by setting the allowed error setting to 0. I did so to show all the questions asked by the program.

5.2 Early prediction

This is the result when the allowed error is set to 0.05:

```

Welcome to the ticket routing agent.
Please enter y/n to answer the following questions.
Request?y
Incident?y
Early prediction: Datawarehouse? Enter y to route ticket to this response team.y
Routing ticket to: Datawarehouse
Enter q to quit. Any other key to continue with an other key.

```

Figure 8: Early prediction.

As we can see after 2 questions answered already, the model has made an early prediction with 95% certainty that the ticket is to be routed to the Datawarehouse team. At this point the user may enter 'n' and continue with building the query or 'y' if they are happy with routing the ticket to the team predicted by the model.

5.3 Regression testing

This is the result when the advanced agent is used to carry out the ticket routing process:

```

Welcome to the advanced ticket routing agent.
Please enter y/n to answer the following questions.
Request?y
Incident?n
WebServices?n
Login?n
Wireless?y
Printing?n
IdCards?n
Staff?y
Students?n
Route the ticket to: Equipment?y
Routing ticket to: Equipment. It will approximately take: 3 days
Enter q to quit. Any other key to continue with an other key.

```

Figure 9: Advanced agent ticket routing.

As we can see the user is informed on the approximated number of days the help desk will need to deal with the ticket, a value returned by the MLPRegressor implemented as part of the additional requirements.

Note: I have turned off early prediction for this test as well, as the purpose of this test was to show that a time prediction is made.

5.4 Comments on testing

The reader is encouraged to run the program several times, experimenting with different inputs in order to see the full functionality of the system.

6 Conclusion

Overall I'm very pleased with the result of my work as I've fully implemented the Basic and Intermediate agents and I have also implemented numerous additional features including the implementation of another neural networks carrying out a regression task, I've plotted several informative graphs and I have also found ways to automate the tedious process of hyperparameter tuning using a grid search. Furthermore I have also implemented a module which compares the performance of several classification algorithms, many of which I have never come across before. If I had more time I'd probably implement additional features such as providing the ability for the IT services to introduce new teams that deals with specific subset of tickets tagged in a certain way. In conclusion, this practical was very successful in achieving its objectives as even though I've used some machine learning algorithms in the past, I have gained a broader knowledge of the subject matter and acquired new skills and insights that will be very useful in the future.

References

- [1] Brownlee, J. (2019). Why One-Hot Encode Data in Machine Learning?. Retrieved 20 December 2019, from <https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/>
- [2] sklearn.neural_network.MLPClassifier — scikit-learn 0.22 documentation. (2019). Retrieved 20 December 2019, from https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
- [3] Toniolo, A. (2019). CS5011(L16 W9): CS5011 Learning - Part A
- [4] sklearn.model_selection.train_test_split — scikit-learn 0.22 documentation. (2019). Retrieved 20 December 2019, from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- [5] Geron, A. Hands-on machine learning with Scikit-Learn and TensorFlow (pp. 15-17).
- [6] Geron, A. Hands-on machine learning with Scikit-Learn and TensorFlow (pp. 73-75)
- [7] sklearn.neural_network.MLPRegressor — scikit-learn 0.22 documentation. (2019). Retrieved 20 December 2019, from https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html
- [8] Delaney, J. (2019). 10 Classifier Showdown in Scikit-Learn — Kaggle. Retrieved 20 December 2019, from <https://www.kaggle.com/jeffd23/10-classifier-showdown-in-scikit-learn>
- [9] Geron, A. Hands-on machine learning with Scikit-Learn and TensorFlow (pp. 125-127).
- [10] scikit-learn: machine learning in Python — scikit-learn 0.22 documentation. (2019). Retrieved 20 December 2019, from <https://scikit-learn.org/stable/>