Machine Learning COSC2673

Assignment Two

Team Member	Student ID
Chao Geng	s3512592
Aedan Pragnell	s3657788

Decision Tree Learning Task

Definition

Task Overview

The Decision Tree Learning task was created to investigate the prices of properties sold in Melbourne based on data from the last two years. The main goal was to pre-process the given data, apply feature selection and feature engineering and to create a hypothesis that assisted in predicting the price band for a given property.

Problem Statement

The goal is to create a hypothesis, or model, which can be used to efficiently predict the price band for a Melbourne property from the supplied dataset. The tasks involved are:

1. Import the supplied datasets
2. Pre-process the data
3. Engineer and select features
4. Train a model to classify the data
5. Evaluate and refine model

Metrics

- Precision:
  TP / (TP + FP)
- Recall:
  TP / (TP + FN)
- F1 Score:
  2 * (Precision * Recall) / (Precision + Recall)
- Accuracy Score:
  (TP + TN) / (TP + TN + FP + FN)

We took advantage of Scikit-learn's classification_report function which provided us with Precision, Recall and F1 Score. However, we also used accuracy score, to express how many examples were correctly predicted.

Findings

Surprisingly, a Decision Tree model which utilised a tweaked implementation of forward feature selection was more accurate than models with less features. However, the singular tree was hardly comparable to more advanced models such as the Random Forest Classifier, which utilises the average of multiple Decision Tree models.

Analysis

Exploratory Data Analysis

The supplied data set had a total of 27244 entries, 20 features and the target price_bands. Although there were a decent amount of entries, the majority had at least one missing attribute which could either be filled with the mean, median or most frequent value, or could be removed entirely. With simple domain knowledge, we could see there were attributes which are likely to have little statistical significance, such as the lattitude and longtitude features. The address feature would also be insignificant as there were not many options for us to process the string into a meaningful feature. Postcode/suburb would be sufficient enough to drop the feature entirely. Other features such as building_area and year_built had 16588 and 15160 missing values respectively. Handling building_area would be crucial to creating a decent classifier as it is an important aspect of properties.

Methodology

Data Preprocessing

Upon importing the data, we converted the date to UNIX format so it was on a continuous scale. This allows the Decision Tree to utilise the feature, as it was not an object. We also had to deal with the other features with a type of object, to allow the model to work. This was done simply by using the Pandas get_dummies function.

The main priority was dealing with the large amount of null values. We reduced the amount features by getting rid of the ones which we felt were insignificant. These were: address, year_built, lattitude and longtitude. This left us with 5 features which contained null values; bedrooms, bathrooms, car_parks, landsize and building_area. We felt like all of these features were important for the model's classification so used the Imputer function to fill in the missing values with the mean, median and most frequent values. We also removed all of the null values for comparison. This resulted in the following scores:

Null Value Treatment	Precision	Recall	F1 Score
Entry Removed	**0.61**	**0.60**	**0.61**
Mean	0.58	0.59	0.58
Median	**0.58**	**0.58**	**0.58**
Most Frequent	0.58	0.59	0.59

Removing all the entries with null values produced the highest Precision, Recall and F1 Score. Since an extensive amount of entries were missing at least one attribute, the simple Decision Tree model slightly struggled to learn the relationships between

features, due to them being filled in. However, with the entries removed, the data was more accurate, resulting in the better scores. After continuing with the DataFrame with the rows with null values dropped, we explored the following scalers: StandardScaler, QuantileTransformer, Normalizer and RobustScaler. This resulted in the following scores:

Scaler	Precision	Recall	F1 Score
StandardScaler	0.61	0.60	0.61
QuantileTransformer	0.61	0.60	0.61
Normalizer	**0.35**	**0.35**	**0.24**
RobustScaler	0.61	0.60	0.61

It was clear that scaling did not improve the data, however this may be due to the data being in the format of “dummy” values, from the get_dummies function. Therefore, we moved on to feature engineering.

Feature Engineering

During further research into the property market, we discovered that realestate.com.au displays the “Average Demand Market” for a given property, and compares it with the average of Victoria. For example, the Abbotsford suburb has an average of 492 visits per property while the average in Victoria is 1023. After using Chrome Developer Tools, we found the API backend and took advantage of it to scrape the data and store as a csv. (See demand_ratio_scraper.ipynb). This resulted in the demand_ratio feature for each suburb, was applied to the dataset. We divided the Victorian average by the suburb average to find the ratio.

Feature Selection

We utilised a tweaked version of forward selection to select our features. Our version differed to the normal version by not stopping once the addition of a new feature didn’t improve the score. This was due to some manual tests that expressed the score improving later on. To start of the forward selection, we first had to find the singular feature that was the most accurate at classifying the properties. The best five in order were: suburb, postcode, suburb_property_count, demand_ratio and council_area. All 5 features were related to the location of the property, rather than its features, such as bedrooms and bathrooms. This also expresses that the model may benefit from the engineered demand_ratio feature. After the loop was done, the results were sorted and the best model ended up using all of the 18 features.

Implementation
The data was split into 75% training set and 25% testing set. We chose this ratio over an alternative such as 80% training and 20% testing to minimise the chance of overfitting.

We created 4 different models in total, with two being Decision Tree Classifiers and the other two being Random Forest Classifiers for comparison. Model v1 used all 18 features as mentioned above. Model v2 used only 6 basic features for a property: suburb, type, bedrooms, bathrooms, car_parks and building_area. This is usually the basic data for a given property so we felt like it would be a good comparison with v1. The two Random Forest Classifiers both used the same features as v1 and v2 respectively.

Refinement
With decision trees, the main parameter that influenced the classification accuracy was max_depth. Using GridSearchCV, we tested different max_depth, min_samples_split and min_samples_leaf which resulted in a poorer score compared to an unrefined model. After removing min_samples_split and min_samples_leaf from the search, the models improved.

To illustrate the power of the Random Forest Classifier, there wasn't any parameter tuning. The models simply had n_estimators set to 1000.

Model	Accuracy Score
Decision Tree v1 (18 features)	0.49
Decision Tree v1.1	0.54
Decision Tree v1.2	0.62
Decision Tree v2 (6 features)	**0.41**
Decision Tree v2.1	0.45
Decision Tree v2.2	0.53
Decision Tree v2.3	0.56
Random Forest v1 (18 features)	**0.67**
Random Forest v2 (6 features)	0.58

Results
Model Evaluation and Results
Decision Tree v1.2 had the following 18 features: suburb, rooms, type, postcode, bedrooms, council_area, region_name, distance, suburb_property_count,

demand_ratio, car_parks, bathrooms, landsize, building_area, method, date and realestate_agent. The only parameter changed was: max_depth = 10.

Decision Tree v2.3 had the following 5 features: suburb, type, bedrooms, bathrooms, car_parks, and building_area. The only parameter changed was: max_depth = 64.

Random Forest v1 had the same parameters as Decision Tree v1.2 and had n_estimators set to 1000.

Random Forest v2 had the same parameters as Decision Tree v2.3 and had n_estimators set to 1000.

The best estimator was the Random Forest v1 with an accuracy of 0.67. The second best was the Decision Tree v1.2 with an accuracy of 0.62. Next was the Random Forest v2 with a score of 0.58 and lastly was the Decision Tree v2.3 with a score of 0.56.

Conclusion

It was clear that the Random Forest Classifier was no match for the Decision Tree Classifier. Utilising multiple decision trees and bagging, to create an average prediction, the Random Forest v1 model was 5% more accurate than the Decision Tree v1.2 model. Getting rid of the data entries with null values proved to create the most realistic representation of the data, rather than filling with the mean, median and most frequent value. Both the Decision Tree and Random Forest classifiers benefitted from the large amount of features, including the engineered demand_ratio feature. However, the final result is not too accurate which may due to the target being in price_bands.

Multi-Layer Perceptron Task

Definition

Task Overview

The Multi-Layer Perceptron task was created to use deep learning to correctly label images from the MNIST fashion dataset. These images included different a number of different garment types, such as trousers and sneakers. The main goal was to design the perceptron, experiment with the number of hidden nodes and to create a hypothesis that assisted in classifying fashion-related images.

Problem Statement

The goal is to create a hypothesis, or model, which can be used to efficiently the category of a given from the supplied dataset. The tasks involved are:

1. Import the supplied datasets
2. Train a model to classify the data
3. Evaluate and refine model

Metrics

- Accuracy Score:
  (TP + TN) / (TP + TN + FP + FN)

Throughout the Decision Tree Learning task, we gravitated towards using the Accuracy Score as the primary metric for assessing performance. Therefore, we decided to only use Accuracy Score for this task.

Findings

Using 431 hidden nodes in our model produced the best test accuracy due to it being in the sweet spot between underfitting and overfitting. Cross validation wasn't effective in our implementation.

Methodology

Data Preprocessing

We loaded all 70000 data examples from the training and testing data folders into our notebook in the form of numpy arrays. After reshaping the arrays into a form suitable for our deep learning model, we then normalised the data. Due to all images being on the same scale from 0 to 255, denoting the grayscale pixel colour, we simply had to divide all the values by 255. All the labels were converted to a binary

class matrix, or one hot encoded, to ensure the labels were not on a continuous scale.

Implementation

The data was split into 90% training set and 10% validation set. Furthermore, we had 54000 training examples and 6000 validation examples.
We created a few helper methods to create the model based on the number of hidden nodes, score the model and print out graphs related to the model. We then tested the following hidden node amounts: 78, 254, 431, 607 and 784. Each amount had an almost equal increment of 176 or 177. This resulted in the following tables:

Run No.	Epochs	No. of Hidden Nodes	Training Accuracy %	Validation Accuracy %
1	10	78	**88.18**	**87.59**
2	10	254	88.78	88.11
3	10	431	88.87	88.32
4	10	607	88.92	**88.40**
5	10	784	**88.95**	88.21

No. of Hidden Nodes	Test Accuracy %
78	**87.76**
254	88.30
431	**88.94**
607	88.29
784	88.08

As expected, run number 1, which had 78 hidden nodes, was the poorest performing model. With the little amount of nodes, the model was underfitting the data. On the other hand with run number 5, which had 784 hidden nodes, the training accuracy was the highest, but the test accuracy was the 2nd worst. This clearly expresses overfitting. The best performing run based on test accuracy was run number 3, which was near the sweet spot, between underfitting and overfitting. It was also in the middle of the two for the amount of nodes. Therefore, we used 431 nodes for cross validation.

Cross Validation

Using StratifiedKFold, the training data was split 5 different times to cross validate our Multi Layer Perceptron.
The same preprocessing was applied to the data. This resulted in the following table:

Fold	Epochs	No. of Hidden Nodes	Cross Validation Accuracy %
1	10	431	88.52
2	10	431	88.49
3	10	431	87.60
4	10	431	87.71
5	10	431	88.26

Mean	88.12
Standard Deviation	0.39

The StratifiedKFold cross validation ensured that the model performed well on different sets of training and validation data. Rather than focusing on one split, having multiple splits and recording the mean accuracy score expresses a more reliable score. The cross validation resulted in a mean of 88.12%. This was slightly lower than the test accuracy of 88.94%, but this can most likely be attributed to the different training/validation sample amounts. StratifiedKFold used 48000 training samples and 12000 validation samples while we used 54000 training samples and 6000 validation samples. Being significantly less than the test accuracy, we can conclude that the cross validation wasn't effective for our model.

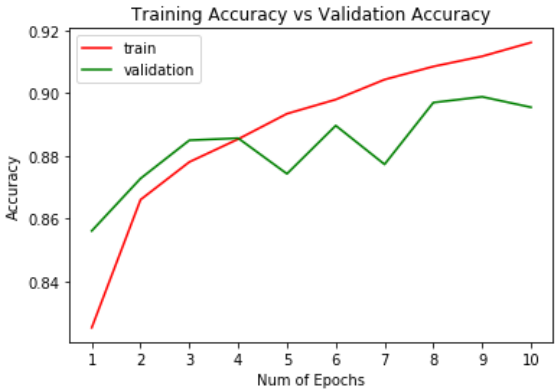
Results

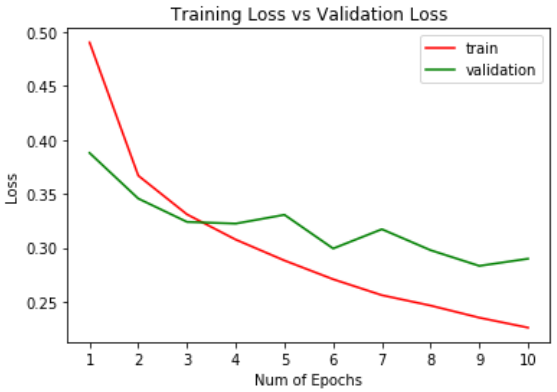
Model Evaluation and Results

Our best performing Multi Layer Perceptron had 431 hidden nodes. Having less hidden nodes resulted in a poorer performance, attributed to underfitting and having more nodes also resulted in a poorer performance, attributed to overfitting. Although the training and validation accuracy percentage wasn't the best out of the 5 runs, the test accuracy was the best. Therefore, the assumption can be made that the model can predict well on unseen data.

No. of Hidden Nodes	Training Accuracy %	Validation Accuracy %	Test Accuracy %	Cross Validation Accuracy Mean %
431	88.87	88.32	88.94	88.12

The cross validation accuracy mean was slightly lower than the other 3 accuracy results, but this was likely caused by the different training/validation sample amounts.





As depicted in the above graphs, the validation score experienced heavy fluctuations. However, it flattens out at the 8-10 epoch mark. Therefore, for the best classification performance, the maximum validation accuracy score should be used, which would 9 epochs with an accuracy of 89.88%. Any more epochs and the model is likely to overfit the data. Using too many epochs for training is not always a good approach due to the tendency to overfit as there are more epochs. For the graphs above, the error decreases on the 10th epoch and would likely decrease further, if the amount of epochs was increased. However, this was not something we needed to examine.

Conclusion

The training process for the Multi Layer Perceptron task mainly involved a lot of minor adjusting, such as the trial of 5 different hidden nodes amounts. It was clear that having too little hidden nodes resulted in underfitting the data and having too many hidden nodes resulted in overfitting the data. Additionally, having too many epochs would result in overfitting the data but this wasn't obvious in our evaluation.

It is clearly know we can get better result from single split instead of k-fold. The different the number of hidden nodes provide different result. We need to figure out which one is more suitable for our dataset. We find 607 hidden nodes give us best result on test is 88.6% accuracy and the k-fold give us 87.84% accuracy. The difference between the two is small, but k-fold is more overfitting than single-split and this is not a good thing.

Compare and Contrast

The decision tree is an easily-understandable model that is easy to use. It is extremely easy to just plug in a given dataset into the model, compared to the multi-layer perceptron that you have to configure yourself. Therefore, the perceptron is more complex and has a greater learning curve. However, that can be said for most deep learning models compared to machine learning models due to jump in complexity.

Both decision trees and multi-layer perceptrons are viable solutions to classification problems. However, given there is enough data, the deep learning route is likely to outperform the basic tree model. This was evident as the decision tree's performance was fairly poor with the best iteration only scoring a test accuracy of 62%, while the best iteration of the perceptron scored a test accuracy of 88.94%. Although the datasets and tasks were different in nature, the decision tree is a fairly weak learner and benefits from bagging, as shown by the Random Forest scoring a test accuracy of 67%. More advanced ensemble methods like the Gradient Boosting classifier would likely perform as good as the Random Forest or better.

The decision tree had a lot more parameters to tune once the model had been created. On the other hand, the multi-layer perceptron allows for you to build the perceptron however you like. With only having one hidden layer, this wasn't a factor in the task. The perceptron was much easier to refine since we only had to find the number of hidden nodes that resulted in the highest accuracy. However, the decision tree benefited from feature engineering and selection. This wasn't needed in for the multi-layer perceptron task although we could have experimented with down-scaling the images.

The decision tree could be used for the MNIST fashion dataset as the image is formatted into numpy arrays. However, since the decision tree performed poorly on the housing dataset, we assume it would perform poorly on the fashion dataset too. The multi-layer perceptron could also be used for the housing dataset, but may perform poorly due to the lack of data. Deep learning models often benefit from a larger scale of data and after all the data cleaning, we weren't left with many examples. In conclusion, the decision tree was more suited towards the housing price band classification task, albeit performing poorly, and the multi-layer perceptron was more suited towards the fashion image classification task.

Overall, it is clear there is a separation of complexity between the two models. The decision tree model is a lot simpler and easier to configure, while the multi-layer

perceptron is substantially more advanced and involves configuring the model yourself to get it to run.