CISC 372 Assignment 1

Rylen Sampson, 20051918

Introduction:

The problem to be solved for this assignment is to use data science techniques to predict the classification of an Airbnb listing as being in a low, medium or high price range (0, 1 or 2). To help with this task a dataset is provided which contains a large amount of data for each listing although some listings may be missing values.

My prediction for this assignment is that predicting the target label for over 7000 “unseen” listings will be rather difficult with the ambiguity of the characteristics in the dataset and the complexity of the problem’s nature. Thus only the best models will reach scores of 75% to 80%.

To tackle this problem I will implement both a Support Vector Machine (SVM) model and a Multilayer Perceptron (MLP) for classification. My prediction is that the MLP will perform better. This is because SVM’s rely on support vectors to construct decision boundaries and when we begin to use four or five plus features, constructing these boundaries may not be the most effective. MLP’s have the added bonus of tending to not become overfit too easily when the model is trained on many features.

Script a1_s1.ipynb Design Process:

Preliminary Steps: My first attempt at implementing my own model to solve the classification problem was very basic and more for my own understanding. I explored the capabilities and features of the different data science libraries available such as sklearn, pandas, matplotlib and seaborne. These libraries help with data preprocessing, data visualization, feature engineering and model construction. The first model I selected to attempt the problem at hand was a Support Vector Machine for multilabel classification. To get a sense of how this model performs I did no hyperparameter tuning, feature engineering, or data preprocessing. The features I chose to train the model were selected at random and were primarily based on what I thought made intuitive sense. This gave me an idea of what to expect in terms of accuracy by the SVM model and a benchmark to improve on with future versions of the model.

Feature Selection: No model can be successful in testing if it is trained on unexplanatory features. Thus my first idea in improving my model was feature engineering and selecting appropriate features to use for training my model.

To get an idea of the relationships between the different variables I used the correlation heatmap available in seaborne. This shows all possible correlations between numeric features in the dataset (Figure is found in the respective Jupyter notebook). A negative of the correlation heatmap is that it doesn't include categorical features. The numeric features I deemed worthwhile to include in the model training were:

- Bedrooms
- Beds
- Accommodates

The categorical features were chosen primarily based off intuition and a trial and error approach, gauging which categorical features resulted in an increase of accuracy. The categorical features in my final model for this submission were:

- Property_type
- Room_type

I tested versions of my SVM model with even more features, but this tended to cause a decrease in accuracy, likely due to overfitting.

While selecting features, an interesting insight I made was that the square_feet feature has 7521 missing values out of 7631 total values. My first thought was to implement a prediction method to fill missing values, but that would result in 98.5% of the values for this feature having the same value. If I was to use a prediction method such as linear interpolation or K nearest neighbours, the lack of known values could easily result in skewed predictions which in turn would negatively affect the model. I decided the best course of action was to drop this feature.

Data Preprocessing: Once I had selected features worth using in the training of my model, the next step was to process the data into a more model-friendly form. To accomplish this task I dove into the abundance of data processing methods available in sklearn. These included the Pipeline, ColumnTransformer, Simple and KNNImputer, StandardScaler and OneHotEncoder methods.

I used Pipeline to build a step-by-step data pipeline that allowed me to quickly and efficiently build and test my models. ColumnTransformer was just one step of said pipeline and provided an easy way to apply the different preprocessing methods I wanted to use. For numerical data I began by using the StandardScaler to scale the features to follow a standard normal distribution as well as using the SimpleImputer function to interpolate missing values based off a simple strategy such as filling using the mean or median. I then thought this may not effectively reflect the complexity of the dataset at hand and therefore used KNNImputer to interpolate missing values based on a K nearest neighbours' strategy. This resulted in an increase of accuracy. SimpleImputer was used to fill in missing categorical data with a simple "missing" string as the mean and median strategies are unusable with non-numeric data. OneHotEncoder was needed to transform the categorical data into a form readily accepted by sklearn models.

Furthermore, as mentioned above, the square_feet feature seems to be flawed and so it was dropped from the dataset. In terms of outlier detection, only one case was overly apparent. In both the beds and bedrooms features there was a data point that had a count of 20. This record, or records but I was under the assumption it belonged to the same data point, was removed from the training set.

Hyperparameter Tuning: Using an SVM model and the preprocessing techniques I did, there were numerous possible hyperparameters to tune but only so many had a noticeable effect. The ones which did were left within the script for my final model. These include.

- C, a regularization parameter
- Gamma, a kernel coefficient
- The number of neighbours to use in the KNNImputer
- The Kernel type

Note: In the script for the final model, the only kernel type checked is the radial basis function (rbf). This is solely to reduce runtime of the final model. Other kernels were checked such as polynomial and sigmoid.

To select hyperparameters GridSearchCV was imported from the model_selection sub library of sklearn. Passed to this function is a variable labeled param_grid which contains a range of values to test for each hyperparameter. GridSearchCV then takes every possible combination of these hyperparameter values and fits an SVM model with each, performing 5-fold validation using the training set further split up into mock train and test sets. An accuracy score for each fitted model is returned and finally GridSearchCV indicates which set of hyperparameter values provides the highest accuracy.

Testing: With hyperparameter values chosen, data preprocessed, and feature selection performed, a final model was constructed. I then used this model to predict on the test set, generating a csv file of the results to submit to Kaggle.

Script a1_s2.ipynb Design Process:

Preliminary Steps: After my first attempt I was much more familiarized with the data science libraries, allowing me to further expand on my previous attempt. This attempt utilized a Multilayer Perceptron for classification purposes. Similar to my previous attempt, I tested the model with no sort of model improvements to get an idea of the expected accuracy. This score would act as a benchmark in which I can improve upon.

Feature Selection: An issue which posed problems in my last model was the lack of insight into the categorical features. With this attempt I utilized the subplot ability of seaborne to plot numerous bar plots side by side for easy comparison of the categorical features. From these plots I identified further categorical features I thought would be worthwhile to try and train the model with. These plots can be found in the Jupyter notebook.

To gain a different perspective of the numeric features I also used the boxplot ability of seaborne. This shows how the data for each feature is distributed with respect to the target label. The boxplots weren't the clearest to interpret, but the ones which were, were easy choices for features to include in the training of our model. These boxplots are also found within the Jupyter notebook.

The features used in the final version of this model are:

Numeric

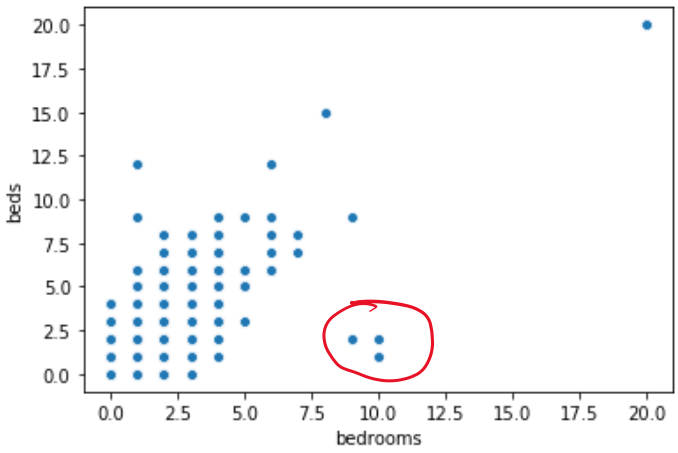
- Accommodates
- Bedrooms
- Extra_people
- Bathrooms

Categorical

- Room_type
- Property_type
- Instant_bookable
- Neighbourhood_cleansed

Data Preprocessing: After selecting the features they need to be processed. In this attempt I tried to better preprocess the data. This involved removing more outliers, and formatting features to be usable.

For outlier detection, I plotted a scatterplot comparing the number of bedrooms versus the number of beds within an Airbnb listing. An outlier could be identified as a listing that has a disproportionate number of beds to bedrooms. Below is the figure produced by the scatterplot.



The highlighted data points satisfy the definition of outliers. 10 bedrooms with less than 2.5 beds isn't indicative of a typical Airbnb listing. Removing these outliers will help our model become more generalizable.

From the boxplots, I concluded that the extra_people feature is worthwhile to train our mode with. The issue with using this feature is that it is formatted as a monetary value. Using pandas, removing the '$' sign was a simple task and the code for this step is found in the Jupyter notebook.

I thought the bedroom and review_scores_cleanliness features may have an interesting correlation. When I plotted these two features, the correlation was actually the opposite of what I expected. When the number of bedrooms increased, the cleanliness scores increased as well. One would think the opposite holds true. Because of this interesting relationship I tested using review_scores_cleanliness as a feature but it decreased the accuracy of the model, so I removed it in the end.

The rest of the data preprocessing methods used for this attempt are the same as the previous attempt and so will not be covered in further detail.

Hyperparameter tuning: Similar to the SVM model, the sklearn MLP classifier has numerous hyperparameter's that can be tuned. Some hyperparameters always took on the same value when tuning or had a negligible effect for the amount of time it takes to tune, these were taken out of the grid search to improve efficiency of the tuning. The hyperparameters that are tuned in my final model are:

- Size of Hidden Layers
- Number of Hidden Layers
- Alpha, a L2 penalty parameter
- Activation function
- Maximum number of calls to the solver function
- Number of neighbours for the KNNImputer

Note: In the final Jupyter notebook, only one activation function, logistic, is in the hyperparameter tuning to increase efficiency.

As was done in my first attempt, I used the sklearn implementation of grid search, GridSearchCV. Nothing substantial differed between this part of the hyperparameter tuning from my previous attempt.

To select the number and size of the hidden layers I followed the general consensus for hidden layer design [1]. This is demonstrated in the possible values I provided that this hyperparameter could take on as well as the one that continuously provided the best accuracy.

Testing: With a final model, I once again made my predictions on the test set of data. The results were then put into a csv satisfying the requirements as listed on Kaggle and uploaded for scoring.

Conclusion & Results:

As hypothesized, the MLP performed better than the SVM model in the test stage. As mentioned in the design processes for both models, the design process began with a completely raw model with just the sklearn model being used for classification. Immediately there was a noticeable difference between their scores.

Below are the public leaderboard Kaggle results for both final models:

Support Vector Machine	0.70492
Multilayer Perceptron	0.71776

Clearly the two final models do not largely differ in accuracy. From the first tests to the final model, the SVM model increased in score the most. Aside from the complexity of the problem at hand, higher scores may not have been achieved due to insufficient planning at any of the steps in the design process.

In conclusion, it is evident a standard implementation of a machine learning model with limited data preprocessing, feature engineering and hyperparameter tuning will not effectively solve this problem. I believe a more ingenious approach must be taken to reach higher scores, approaches that may not be well-known to me at this point due to a lack of experience.

Questions:

1) The reason why we might want to limit the public leaderboard to only three submissions per day is because it forces the student to be selective with their submissions. This makes the student more critical of their model design and how they have applied data science methods during the design process. Concepts such as data preprocessing, feature engineering, and more become necessary if the student wants to raise their chances of topping the leaderboard with the limit of three submissions per day.

2) If final results were displayed immediately after each submission it would encourage students to attempt to reverse engineer the “best” solutions rather than devise their own approaches to tackle the problem at hand. This is why there are two separate leaderboards, a public one so students can get a general idea of how their models perform and a private leaderboard to gauge final results and how a student’s model may be generalized to unseen data.

3) A model that I attempted to use, and which gave me my best score during the time of writing, was a Multilayer Perceptron Classifier. A few ways I controlled its flexibility were by changing the number of hidden layers, the size of those hidden layers, and using different sets of features to train the model.
   I started out with an inflexible model; this was done by letting all hyperparameters take on their default value provided by the sklearn library. Slowly, to increase the flexibility, I added different hyperparameters to my “param_grid” variable which allowed me to test the effect of manipulating the models hyperparameters.
   In this “param_grid” variable the hyperparameters used to introduce a more flexible model were the activation function, the size and number of hidden layers, a value alpha, the maximum number of calls to the solver function and the number of neighbours to use for the KNNImputer. Once I found a set of suitable hyperparameter values which consistently provided me with an accuracy higher than other tests, I used those as they seemed to provide the best balance between error bias and variance.

References:

[1] https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw