## Introduction

I chose to do the kaggle competition rather than the exploratory project. That being said, I'm assuming you don't need me to recap what the dataset is like I would normally do if writing a small research paper.

I used python's sklearn library to implement my machine learning since, as far as I know, nothing said this wasn't allowed. It definitely took a bit of learning to figure out how it worked and the format that they wanted things in. However, once I figured it out the process of trying out different models or tweaking the hyperparamaters was really easy and significantly better than if I was trying to make these adjustments to the library we built via the homework.

The github for my attempts on this project can be found here. `https://github.com/Tweal/CS6350FinalProject`

## Data Cleaning and Preprocessing

I did initially try running without doing any data preprocessing other than one hot encoding the categorical variables. The results weren't great but they weren't nearly as bad as I expected, I saw about a 5% increase after cleaning the data. I'm assuming this is because only about 1% of the data has missing values.

As far as cleaning the data I took it in two steps, first was replacing the missing values and second was encoding the categorical variables. To replace the missing values I used sklearn's SimpleImputer and simply replaced missing values, denoted as '?' in this dataset, with the most frequent variable in the training dataset. I did experiment with instead replacing with the most common matching label for the training data but did not see a noticeable improvement in the results doing so.

I used two approaches to encode the categorical variables. I started out using panda's get_dummies to one hot encode them. This had the advantage of being extremely trivial to implement but I quickly discovered that it was less than ideal because I couldn't tell it what the values should be for each of the features so I encountered a problem where there was a value in the test dataset, specifically the *native.country* value of *Holand-Netherlands*, that was never seen in the training data and so sklearn gets a little unhappy when evaluating the test data. There was easy solution here to just hard code that column into the encoded data as 0 but I elected to go with the more robust solution of implementing it using sklearn's OHE module. This method has the advantage that you can pass it an array of categories and it will generate the dummy variables for values even if they are not found in the data. This meant that I could create a list of all of the values for each feature from both the training set and the test set and have them end up with the same columns after encoding. This did take quite a bit of finagling because I wanted the encoded data in a pandas dataframe for ease of viewing and manipulating. The problem with that is that sklearn returns a list of lists that I then had to manhandle back into a dataframe. However, once I figured it out I got it working and got basically the same results as when using pandas which is definitely good since it means that I did it correctly.

The other method of encoding that I tried was sklearn's OrdinalEncoder. This works as you'd expect and just assigns a distinct number to each option. The obvious advantages to this being that you don't end up with a hundred features and you can treat each feature as a numeric value. I implemented both of these methods because I wanted to see the difference they made in this dataset. It turns out it didn't make that big of a difference from OHE, the time it takes was lower but not enough for me to justify using one more than the other. It ended up being less than a 2% difference in each instance.

I also split the data 75:25 for training and test data once I realized that I couldn't use the kaggle submissions to test the provided test data. For the remainder of this paper I will refer to this split data as train and test with the dataset submitted to kaggle as final since that is how I referenced it in code.

## Further Preprocessing

All of the above section was what was done before the midterm, this section will contain the improvements implemented since then.

The first thing that I did was decided to just use the ordinal encoding. As mentioned previously the performance of it was similar to the OHE methods but it is more robust because it does not limit the data to just binary labels and thus gives a bit more freedom to the learning.

After that I implemented the KNNImputer from sklearn that I mentioned in my midterm report. As mentioned in the previous section the simple imputer just replaces missing values with the most common value in the dataset, which is easy but not good. By using the KNNImputer it replaces it with the value of the nearest neighbors to that dataset. I used the 20 nearest neighbors to preform the replacements.

## Feature Importance

One thing that I said that I was going to work on in the midterm report w3as that I was going to take a look at feature importance to see what was contributing and could be dropped from my data. To accomplish this I used sklearn's KNeighborsRegressor to model the training data and then ran their permutation_importance to get values for how much each feature contributes for both the training and test data. The reason that I included the test importance as well is to see how big of a roll the features play in generalization of the dataset. The results of these importance tests can be seen in the table below.

| Feature | Training Importance | Testing Importance |
|---|---|---|
| age | 0.00248 | 0.00235 |
| workclass | 0.00000 | 0.00001 |
| fnlwgt | 0.06755 | 0.00987 |
| education | 0.00005 | -0.00001 |
| education.num | 0.00014 | 0.00017 |
| marital.status | 0.00001 | 0.00002 |
| occupation | 0.00010 | -0.00002 |
| relationship | 0.00006 | 0.00006 |
| race | -0.00002 | 0.00005 |
| sex | 0.00000 | -0.00000 |
| capital.gain | 0.05697 | 0.05010 |
| capital.loss | 0.01554 | 0.00743 |
| hours.per.week | 0.00225 | 0.00125 |
| native.country | 0.00025 | -0.00001 |

From this I decided that anything that had an importance for both the training and test dataset of less than 0.0001 was not important and excluded it from my models going forward. This means that the following features were excluded; *workclass*, *education* (but notably not *education.num*), *marital.status*, *occupation*, *relationship*, *race*, *sex*, and *native.country*. You will see in the neural network section that I did confirm that the removal of these features did not play a significant roll in the model.

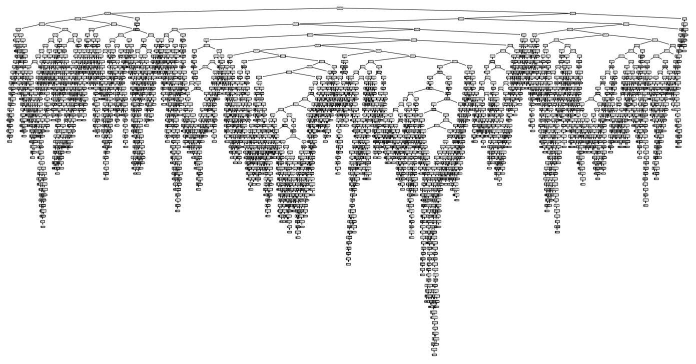
### Data Regularization

Since I know that regularizing data can improve performance I decided to do so to my data. I used two methods, the first is sklearn's StandardScalar which scales the data such that it is a normal distribution with a mean of 0 and a variance of 1. The other method I attemped used sklearn's MinMaxScaler which simply transforms the data to a [0,1] scale. The results of both of these normalization methods is discussed in the neural network section.

## Methods Implemented

### Linear Models

Despite knowing that it was going to be bad I started out with a regular decision tree. I tried it without depth limits, and as you'd expect it completely overfit the training data and got completely out of hand. That being said, it did make a cool picture when visualized, as you can see below. The no max depth tree ended up with a test accuracy of 80.8% but for some reason this did not quite translate to the final dataset as it ended up of 74.5%. I further tried a regular decision tree with max depths of 10 and 5. Those ended up with test accuracies of 85.6% and 85.3% respectively, again these did not translate to the final dataset and ended up about 5% lower when submitted.



From there I moved on to the ensemble methods of random forest, adaboost, and bagged decision trees. For the random forest implementation I left it with the default of 100 trees and tried depths of 10, 5, and 1. The test accuracy were 86.1%, 85.3%, and 86.1% respectively. Depth 10 trees had a final accuracy of 75.0% and the depth 5 trees had 69.6%. The stumps preformed so poorly on the test data that I did not bother submitting to kaggle.

I ran AdaBoost with the default parameters of 10 learners with a learning rate of 1 as well as with 500 learners with a learning rate of 0.01. The default parameters resulted in a training accuracy

86.6% and a final accuracy of 77.9%, while the 500 learners and .01 learning rate had a test accuracy of 84.8% and final of 70.5%.

I ran the bagged decision trees with the default parameters, 10 trees with 100% sampling and all features, as well as with 500 trees and 50% feature sampling. The test results for those were 84.7% for default and 87.0% for the modified parameters. That ended up with final results of about 76% for both.

## Neural Network

The final models that I used to classify the data was sklearn's neural_network package MLPClassifier. This package implements a multi-layer Perceptron using backpropagation to train. I started with the default parameters to see how it would preform out of the box. I ran it with all features and the default parameters and then ran it with the reduced features from the previous sections and got the following results.

| | All Features | Reduced Features |
|---|---|---|
| Training Accuracy | 0.896 | 0.892 |
| Test Accuracy | 0.893 | 0.886 |

These results are better than what I was getting with ADABoost but not by as much as I was hoping, clearly I needed to do more tweaking and manipulation. The bright side is that the reduced feature set is preforming basically the same as the full features so I can just use that going forward and save myself a little bit of running time. That being said assume that for all further data the reduced feature set is used.

After that I tested the regularization of my data using the two methods mentioned above, the results of which can be seen below. The bolded column being the unmodified (at this step) results.

| | **Unscaled** | StandardScaler | MinMaxScaler |
|---|---|---|---|
| Training Accuracy | **0.892** | 0.932 | 0.864 |
| Test Accuracy | **0.886** | 0.928 | 0.878 |

This is obviously an improvement over the un-regularized data and the performance of each is essentially the same as such I will simply use the StandardScaler. Similar to the feature reduction test I will simply use this data going forward.

The next steps that I took were to vary the parameters, starting with the layer count. The model defaults to 100 layers so I decided to try 3, 10, 50, 100, and 500 to see how they preformed. The following table outlines the results of those parameters. As I mentioned the default is 100 layers so this column is bolded as a reminder.

| | 3 layers | 10 layers | 50 layers | **100 layers** | 500 layers | 1000 layers |
|---|---|---|---|---|---|---|
| Training Accuracy | 0.926 | 0.927 | 0.931 | **0.931** | 0.935 | 0.935 |
| Test Accuracy | 0.924 | 0.924 | 0.927 | **0.925** | 0.927 | 0.927 |

It doesn't seem that layer amount is doing much as such I will simply keep with the default 100 layers. At this point my results are good enough to get me into the top 10 as such I am satisfied with this model and going to use it as my final model.

## Final Results

So the final model that I ended up using was a 500 hidden layer neural network using Adam as the optimizer and ReLU as the activation function. Additionally the features were pruned down to just the 8 contributing features (probably could have gone to even more extremes based on the importance results) with those columns normalized to a standard normal Gaussian distribution. This resulted in a training accuracy of 93.5% with a test accuracy of 92.7% accuracy. This translated to a final accuracy (ie the results from kaggle) of also 92.7%.

## Going Forward

As I mentioned previously I am quite satisfied with my results however if I were to continue with this I would streamline my process of iterative modifications by using sklearn's pipeline tools to automatically set up pipelines to preform the steps repeatedly and use grid search to vary the hyperparameters to see how they affect it. As is I did it all by hand and it was a bit of a pain.

Additionally I would look into trying more hyperparameters, I didn't play around with learning rate, the paramaters of Adam, or any other optimiziation or activation models. I'm not sure how much more these results can be improved by someone at my level given that an accuracy of over 90% is pretty good in my opinion.