Predictive Analysis on Adult Income Census using Machine Learning Techniques

Tan Yu Hoe
Diploma in Applied Artificial Intelligence and Analytics, School of Computing
Singapore Polytechnic

This research paper attempts to conduct predictive analytics on adult income census data leveraging over supervised learning techniques. The Adult Income dataset originated from a US Census database in 1994, with 32561 records. The engaged prediction task aims to predict whether an individual's income will be greater than $50,000 per year based on the varying demographic characteristics from the data. This study will be making use of SMOTE to generate synthetic data for class balancing.

Keywords—adult, census, income, prediction, machine learning, classification, python

I. Introduction

With the rise of the word "artificial intelligence" and "machine learning" trending the past decade, we can see more and more people joining the field. But what exactly are these words coined here? By my definition, Artificial Intelligence is where machines being able to replicate human intelligence to do an intensive and strenuous task. In this research paper, I will be demonstrating an example of this Artificial Intelligence, where the machine will be using a learning algorithm to predict whether a person's income will be greater than $50,000 per year based on attributes that came along with the data.

II. Related Works

Generally, this dataset is used as a dummy dataset to test various data science application, such as SMOTE [4] and Bayesian Networks [5]. This dataset can be found in UC Irvine Machine Learning Data Repository, being the second most popular dataset in between Iris and Wine dataset. The dataset can also be found on Kaggle [6], with over 300 notebooks related to the dataset.

III. Dataset

The dataset used here is provided by UC Irvine Machine Learning Data Repository, originating from Barry Becker who extracted it from the 1994 US Census database. The dataset consists of 32,561 rows and 15 columns with each row containing attributes of a single individual. For each person, we are provided with age, work class, final weight, education number, marital status, occupation, relationship, race, sex, capital gain, capital loss, working hours per week, native country, and class. Class, in this case, refers to whether a person earns greater or lesser than 50k per year.

Table 1: Attributes of the data

Attribute	Description
age	*the age of an individual*
workclass	*a general term to represent the employment status of an individual*
fnlwgt	*Final weight: a specified weight based socio-economic characteristics of the population*
education	*The highest level of education achieved by an individual*
education_num	*The highest level of education represented in a discrete form*
marital_status	*Martial status of an individual*
occupation	*General type of occupation of an individual*
relationship	*The relationship of this individual generally perceived to others*
race	*Description of an individual race*
sex	*The biological sex of an individual*
capital_gain	*Capital gain of an individual*
capital_loss	*Capital loss of an individual*
hours_per_week	*Total working hour the individual reported per week*

Before doing any of the fancy stuff, we explore our data to get a better understanding of the dataset first. The main goal is to determine the issue within the dataset that requires a solution to ensure that our model will make better and accurate predictions.

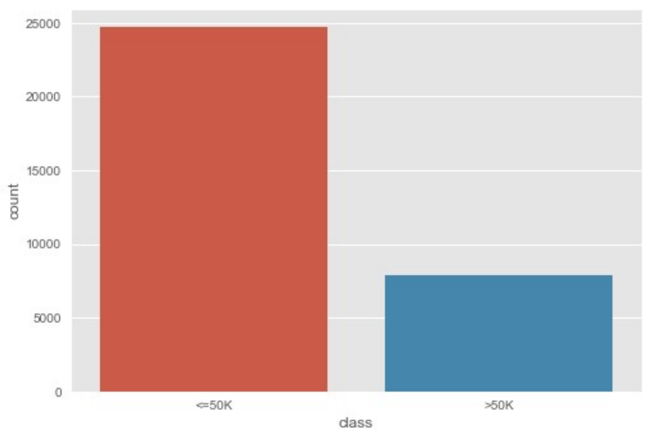


Fig. 1. Frequency Plot on Class

Figure 1 shows there are significantly more people in the data earning lower than $50,000 a year compared to people earning more than $50,000 a year. Furthermore, the classes are quite unevenly distributed, meaning the classes are quite unbalanced. Unbalanced classes are bad as the model might have more bias towards the class with more samples, resulting in bad classification on the other class. As such, I have to either do upsampling the minority or downsampling the majority to balance the classes.

Table 2: Descriptive Statistics on Categorical Columns

	count	unique
workclass	*30725*	*8*
education	*32561*	*16*
marital_status	*32561*	*7*
occupation	*30718*	*14*
relationship	*32561*	*6*
race	*32561*	*5*
sex	*32561*	*2*
native_country	*31978*	*41*

Here the most significant thing we can see is that there are 41 unique values in native_country. We also see occupation and education has a relatively high number of unique values. Here, in this case, I would drop native_country, occupation, relationship, race, and education. The reason being, native_country has too many unique values, occupation and relationship can be represented as work class and marital_status, the race is a sensitive topic in America as such I rather not use it, and there is an education_num in the dataset to represent education.

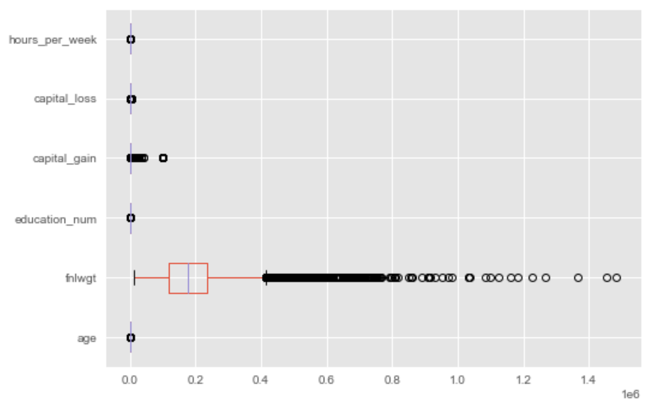


Fig. 2. Boxplot of continuous columns

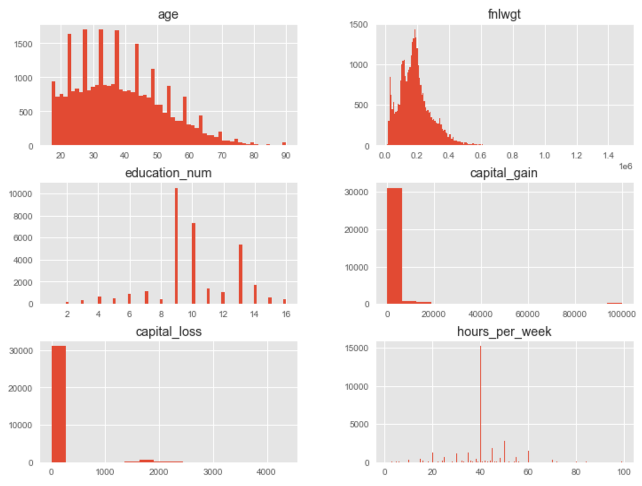


Fig. 3. Histogram of continuous columns

On figure 2 the scale of every column is different as such feature scaling has to be done. Feature Scaling is important as it distributes the weight of each column equally such that there would be no such precedence or hierarchy for one variable over the other. In figure 3, we can see that the continuous attributes, age and fnlwgt, does not have a very skewed distribution as such a simple MinMax normalisation would do. However, capital_gain and capital_loss has quite many "zeros", which I might need to do feature engineering and do normalisation to ensure even distribution. The rest of the discrete variables does not look like they have any abnormalities.

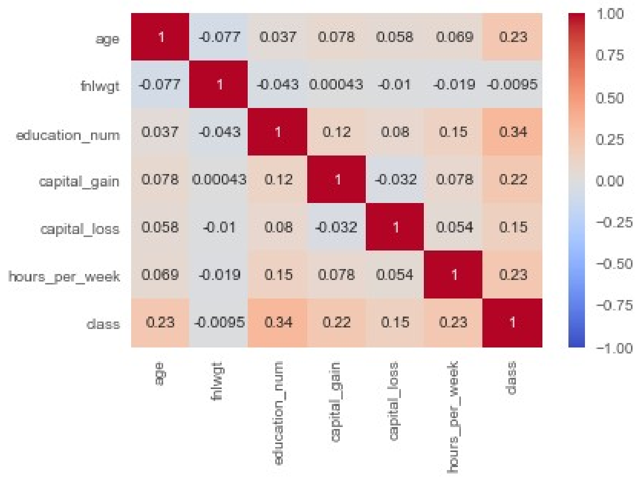


Fig. 4. Pearson Correlation Matrix

Every time I check a correlation matrix, there are two things I will look out for – multicollinearity and weak correlation with the target variable. Multicollinearity is an issue in machine learning as it might reduce statistical significance on another variable, resulting in less significance in the machine learning model. Weak correlation with the target variable just means that the feature has little to no effect in predicting the target variable. In figure 4, it looks like there is no occurrence of multicollinearity; however "fnlwgt" has a very weak correlation with "class". Usually, I would drop "fnlwgt" due to the weak correlation. However, there is very little continuous variable in this dataset, so I would not want to waste one column.

IV. Feature Engineering

For Feature Engineering, I plan to apply these techniques before modelling – Manual Encoding, Data Partition, One-Hot Encoding, Class Balancing and Min-Max Scaling.

Firstly for Manual Encoding, I want to manually encode marital_status and sex into binary form. For marital_status, there are many classes such as 'Married-spouse-absent', 'Seperated' and many more. I want to encode it in such a way that it represents 0 (not married) and 1 (married). For 'sex', it is pretty straightforward since there is only two genders.

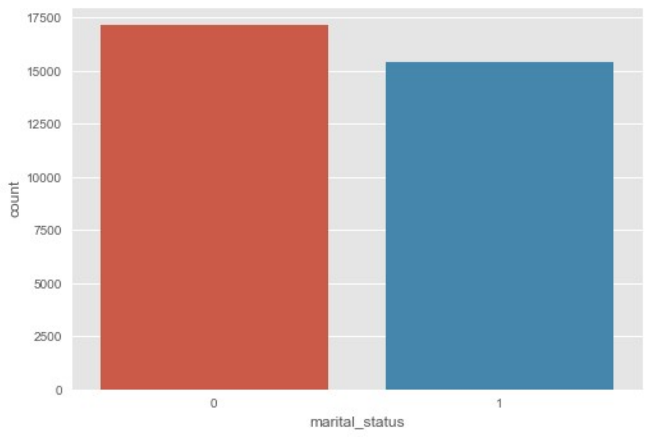


Fig. 5. Frequency plot on marital_status

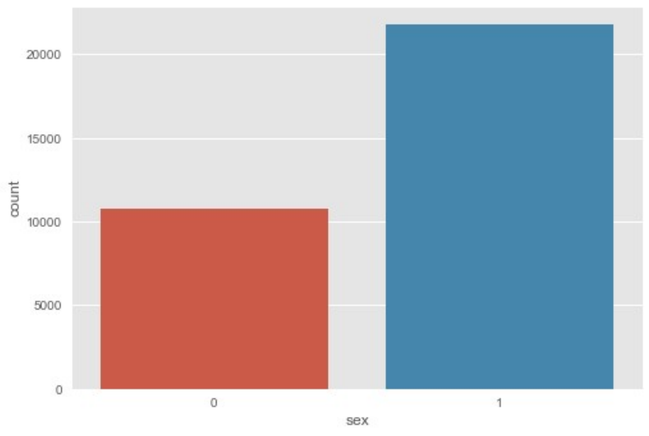
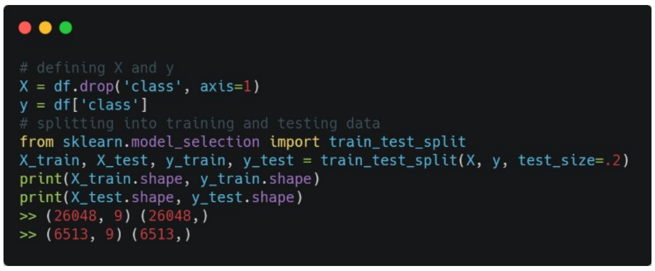


Fig. 6. Frequency Plot on Sex

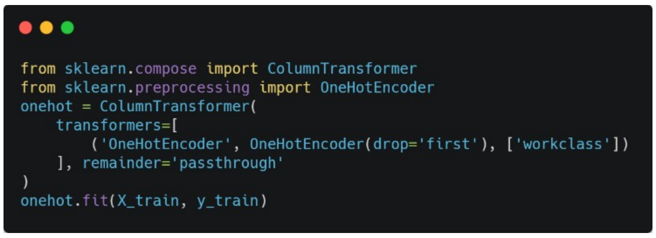
Now for data partition, I want to split my data into a training set (80%) and testing set (20%). The reason being that this is a relatively big dataset, as such I would like to have more data fork-fold cross-validation, later on, that way I can lower k. After splitting, there are 24,580 rows in the training set and 6,145 rows in the testing set.



```
# defining X and y
X = df.drop('class', axis=1)
y = df['class']
# splitting into training and testing data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2)
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
>> (26048, 9) (26048,)
>> (6513, 9) (6513,)
```

Fig. 7. Code Overview on Data Partitioning

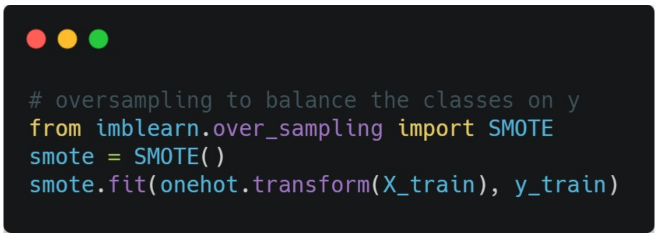
Moving on to One-Hot Encoding, the previous ‘workclass’ was noted to have 8 unique values. Since ‘workclass’ has no inherent ranking or order to it, One-Hot Encoding would be the most suitable technique for encoding nominal data. Since I am going to encode one column, I will be using *sklearn.preprocessing.OneHotEncoder* and *sklearn.compose.ColumnTransformer* method to encode only ‘workclass’.



```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
onehot = ColumnTransformer(
    transformers=[
        ('OneHotEncoder', OneHotEncoder(drop='first'), ['workclass'])
    ], remainder='passthrough'
)
onehot.fit(X_train, y_train)
```

Fig. 8. Code Overview of One Hot Encoding

As mentioned previously, imbalanced classes could lead to the model misclassify the minority class due to the overwhelming amount of the majority class. The solution to this is either upsampling, creating synthetic data on the minority class, or downsampling, delete data from the majority class. I am not a big fan of dropping rows brought another library called “imbalanced-learn”, an open-source library relying on scikit-learn which provided tools when dealing with classification with imbalanced class.[2] Imbalanced-learn has a signature algorithm called SMOTE (Synthetic Minority Oversampling Technique). SMOTE works in a similar concept to k-nearest neighbours, it first finds n-nearest neighbours in the minority class for each of the samples in the class. Then it draws a line between neighbours to generate synthetic data along the line.



```
# oversampling to balance the classes on y
from imblearn.over_sampling import SMOTE
smote = SMOTE()
smote.fit(onehot.transform(X_train), y_train)
```

Fig. 9. Code Overview of SMOTE

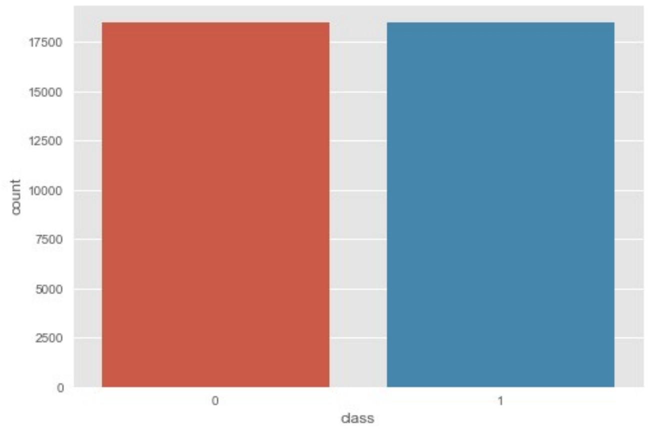
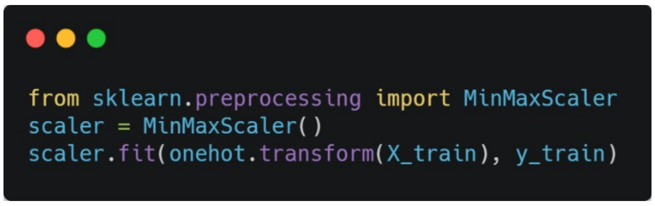


Fig. 10. Frequency Plot of *y_train* after SMOTE upsampling

Comparing between figure 1 and figure 10, we can see that SMOTE has oversampled the minority class successfully.

The last step, Min-Max Scaling, is to reduce the scale of a feature to (0, 1), this is to mitigate any creation of bias towards a single feature. The reason why I chose this method is due to using one-hot encoding previously, thus it would be easier to scale my features down to a range of (0, 1).



```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(onehot.transform(X_train), y_train)
```

Fig. 11. Code Overview of MinMaxScaler

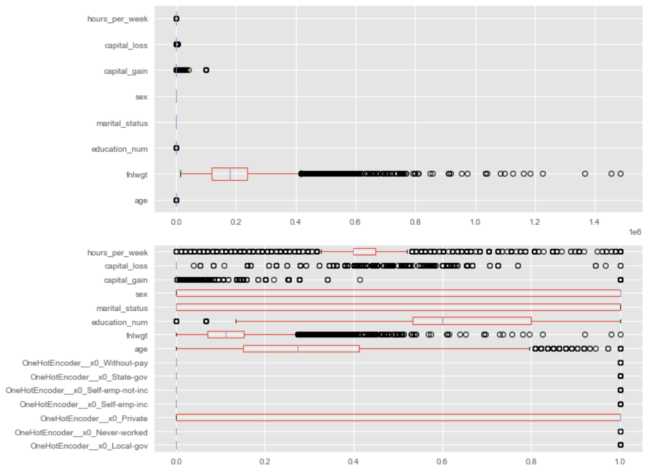
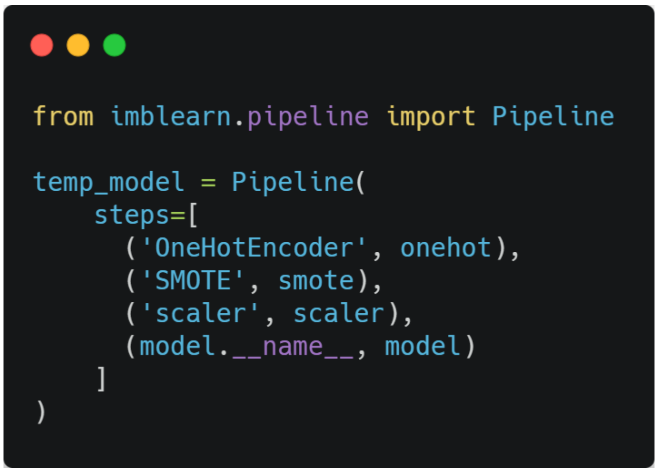


Fig. 12. Boxplots before and after Min-Max Normalization

V. Model Selection



```
from imblearn.pipeline import Pipeline

temp_model = Pipeline(
    steps=[
      ('OneHotEncoder', onehot),
      ('SMOTE', smote),
      ('scaler', scaler),
      (model.__name__, model)
    ]
)
```

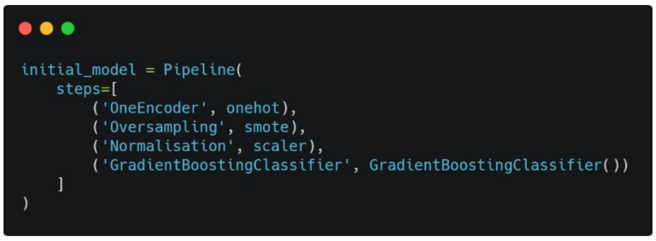
Fig. 13. Code Overview of Pipeline

As shown in figure 13, I prepared a pipeline to ensure all the data preprocessing are applied properly. Another reason for this is to prevent any data/information leakage to the testing set. In model selection, I would want to do k-fold cross validation across multiple models with default hyperparameters, such as ensemble algorithms and logistic regression, and extract the mean accuracy, balanced accuracy, and F1 score.

Table 3: Mean Cross Validation Scores

	F1 Score	*Accuracy*	*Balanced Accuracy*
Gradient Boosting	**0.693306083**	**0.84544345**	**0.7962968**
SVC	**0.679122667**	**0.824572823**	**0.796882281**
Adaptive Boosting	**0.678614795**	**0.836289666**	**0.787702009**
Logistic Regression	**0.669874697**	**0.797030106**	**0.80557602**
Random Forest	**0.653052358**	**0.832058584**	**0.765341478**
Ridge	**0.639234274**	**0.760903173**	**0.789329303**
Extra Trees	**0.637973519**	**0.822335232**	**0.756798575**
K-NN	**0.630265627**	**0.811757526**	**0.754940299**
Decision Trees	**0.596141175**	**0.793531326**	**0.732065752**
Naïve Bayes	**0.408963091**	**0.276566314**	**0.517447182**

From table 3, generally, some indications boosting learning algorithms, Gradient Boosting and Adaptive Boosting, tend to perform better in this dataset compared to other models, especially bagging algorithms. Gradient Boosting happens to be the best performing model in terms of 0.693 F1-Score and 0.846 accuracies. However, the large deviation between F1-score and accuracy could indicate that the false prediction rate in the model is quite high. This is further supported by the deviation between accuracy and balanced accuracy, meaning the recall is low. I decided to go with Gradient Boosting as it has more potential to achieve better results after hyperparameter tuning.



```
initial_model = Pipeline(
    steps=[
        ('OneEncoder', onehot),
        ('Oversampling', smote),
        ('Normalisation', scaler),
        ('GradientBoostingClassifier', GradientBoostingClassifier())
    ]
)
```

Fig. 14. Code Overview of implementing Gradient Boosting

Table 4: Cross Validation Scores on Gradient Boosting (cv=10)

	F1-Score	*Accuracy*	*Balanced Accuracy*
0	**0.696825397**	**0.844589097**	**0.801023429**
1	**0.693459417**	**0.841741253**	**0.799666062**

2	**0.697975709**	**0.84825061**	**0.799131409**
3	**0.690879742**	**0.844182262**	**0.794793307**
4	**0.71086262**	**0.852725793**	**0.809699633**
5	**0.680129241**	**0.838893409**	**0.787474469**
6	**0.673650282**	**0.835231896**	**0.783407664**
7	**0.700162075**	**0.849471115**	**0.800160751**
8	**0.701444623**	**0.848657445**	**0.802319084**
9	**0.708674304**	**0.855166802**	**0.804501213**
Mean	**0.695406341**	**0.845890968**	**0.798217702**

From Table 4, it seems that the standard deviation in testing is low, indicating that the results from table 3 are reliable. At this point, I am going to do an overall evaluation of the model's performance, utilising Learning Curve, Confusion Matric, and ROC (Receiving Operating Curve).

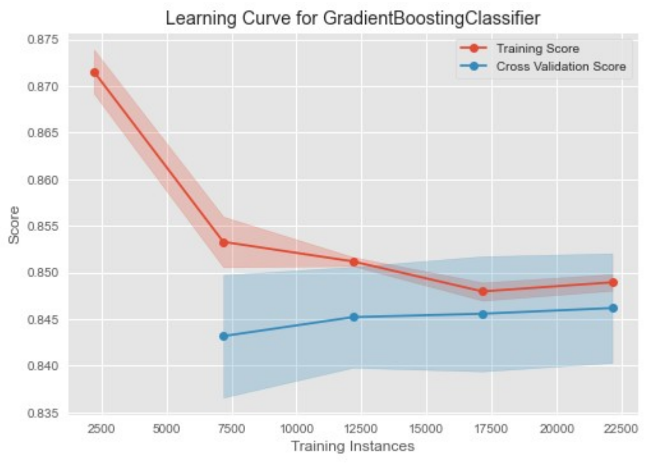


Fig. 15. Learning Curve for Gradient Boosting

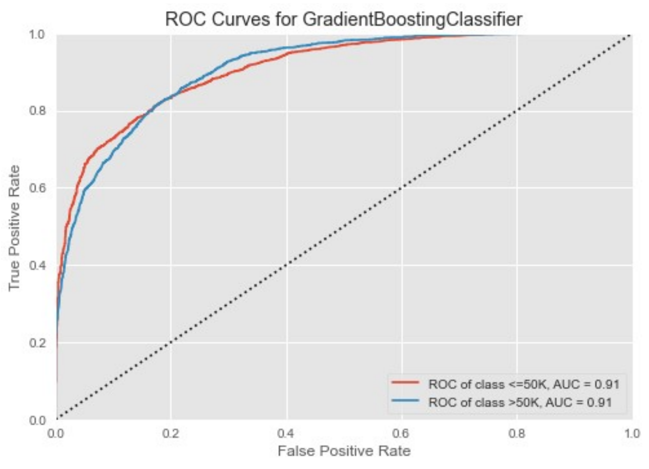


Fig. 16. ROC Curve for Gradient Boosting

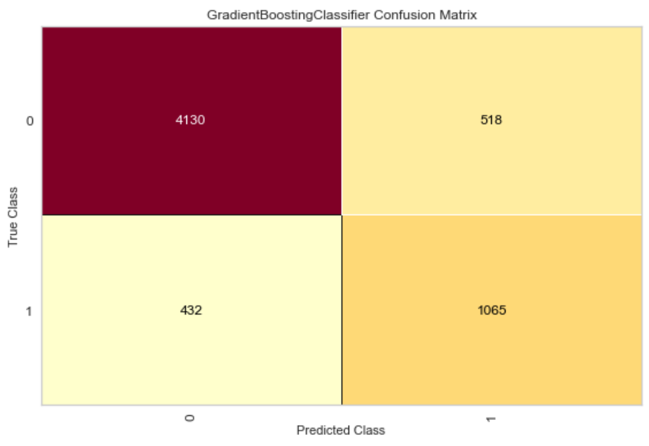


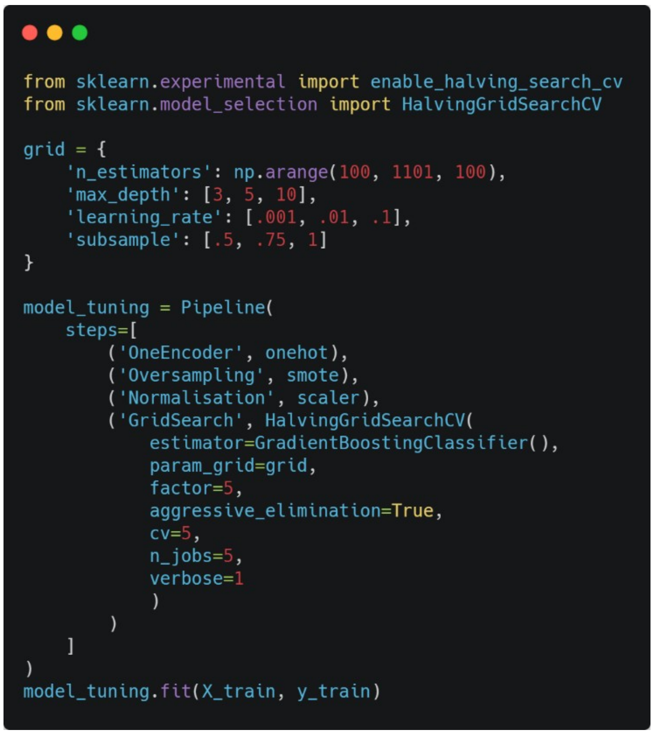
Fig. 17. Confusion Matrix for Gradient Boosting

Figure 15 shows the training score decreasing to match the validation score, reducing overfitting. However, at training instance 15,000 onwards the training score and validation start to move apart, even though the margin is really small. This suggests some overfitting in the model. Figure 15 introduces type 1 and type 2 errors. When AUC (area under the curve) is 0.91, it means there is a 91% chance that the model will be able to distinguish between positive class and negative class. Figure 17 confusion matrix shows the model can classify accurately the individual's income. However, we can see that the model does still make a similar amount of false positive and false negative predictions. Seeing that the model is making false predictions at default hyperparameters, I want to do hyperparameter tuning to improve the model's performance.

VI. Hyperparameter Tuning

Hyperparameter tuning is the process of finding the right combination of hyperparameters to optimise a model's complexity to find the right fit for the dataset. A little background behind Hyperparameter Tuning, initially there are two main methods procided by scikit-learn for tuning Hyperparameter [7] , 'GridSearchCV` and `Randomized SearchCV`. `GridSearchCV` is an exhaustive search function that compares every Hyperparameters using brute force, while RandomizedSearchCV has selects the Hyperparameters based on sampling methods.

Earlier this year, scikit-learn 0.24 update came with ‘HalvingGridSearchCV’, a new way to do Hyperparameter Tuning. HalvingGridSearchCV works by giving every combination a small number of resources and see how well it fare, those who have done well will go through the next iteration. Online articles have shown that HalvingGridSearchCV is apparently 11 times faster than ‘GridSearchCV’. [3] As such, I will be using this method to run Hyperparameter Tuning. I will be tuning these hyperparameters for gradient boosting – n_esimators, max_depth, learning_rate, and subsample.

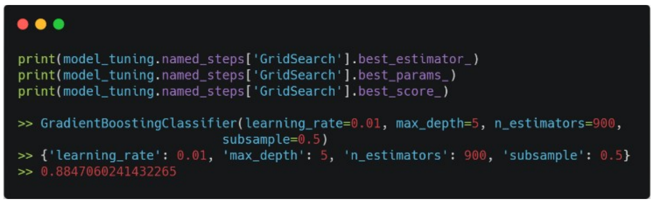


```
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingGridSearchCV

grid = {
    'n_estimators': np.arange(100, 1101, 100),
    'max_depth': [3, 5, 10],
    'learning_rate': [.001, .01, .1],
    'subsample': [.5, .75, 1]
}

model_tuning = Pipeline(
    steps=[
        ('OneEncoder', onehot),
        ('Oversampling', smote),
        ('Normalisation', scaler),
        ('GridSearch', HalvingGridSearchCV(
            estimator=GradientBoostingClassifier(),
            param_grid=grid,
            factor=5,
            aggressive_elimination=True,
            cv=5,
            n_jobs=5,
            verbose=1
            )
        )
    ]
)
model_tuning.fit(X_train, y_train)
```

Fig. 18. Code Overview of HalvingGridSearchCV

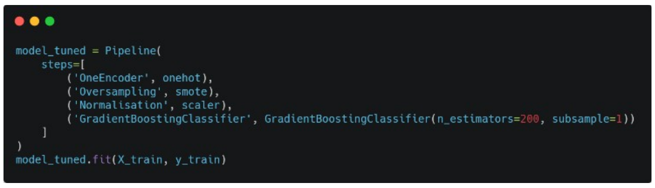


```
print(model_tuning.named_steps['GridSearch'].best_estimator_)
print(model_tuning.named_steps['GridSearch'].best_params_)
print(model_tuning.named_steps['GridSearch'].best_score_)

>> GradientBoostingClassifier(learning_rate=0.01, max_depth=5, n_estimators=900,
                           subsample=0.5)
>> {'learning_rate': 0.01, 'max_depth': 5, 'n_estimators': 900, 'subsample': 0.5}
>> 0.8847060241432265
```

Fig. 19. Results after Hyperparameter Tuning

After Hyperparameter Tuning, we can see that the accuracy score has improved. Here we can see that max depth and n-estimators has increased while subsample has decreased. Likely the model was overfitting and requires subsample to control the bias-variance balance.



```
model_tuned = Pipeline(
    steps=[
        ('OneEncoder', onehot),
        ('Oversampling', smote),
        ('Normalisation', scaler),
        ('GradientBoostingClassifier', GradientBoostingClassifier(n_estimators=200, subsample=1))
    ]
)
model_tuned.fit(X_train, y_train)
```

Fig. 20. Code Overview of Tuned Gradient Boosting

Table 5: Cross Validation Scores on Tuned Gradient Boosting

	F1-Score	*Accuracy*	***Balanced Accuracy***
0	***0.708367181***	***0.853946298***	***0.805638121***
1	***0.705693665***	***0.850691619***	***0.805634592***
2	***0.709359606***	***0.855980472***	***0.80482776***
3	***0.696369637***	***0.850284784***	***0.795612491***
4	***0.712739384***	***0.859641985***	***0.805102585***
5	***0.699507389***	***0.851098454***	***0.79832193***
6	***0.677284158***	***0.843368592***	***0.781249642***
7	***0.71641791***	***0.86086249***	***0.807761186***
8	***0.709942482***	***0.856387307***	***0.804775301***
9	***0.718120805***	***0.863303499***	***0.807228873***
Mean	***0.705380222***	***0.85455655***	***0.801615248***

Overall, the scores in table 5 have improved by abit, with no big changes in scores. The most important part is to see whether the model complexity has been optimised to better fit the dataset.

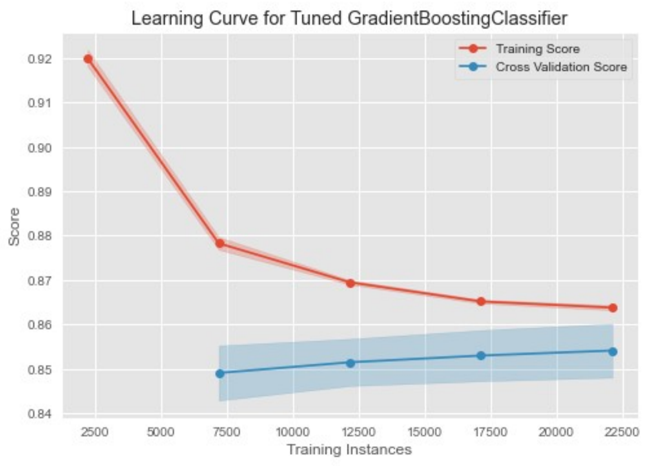


Fig. 21. Learning Curve for Tuned Gradient Boosting

In figure 21, the training score and validation score are approaching each other, variance gap is getting close. This indicates that the model has reduced overfitting. Furthermore, the overall scores has improved compared to the learning curve in figure 15.

VII. Model Evaluation

Now I want to generate predictions and evaluate the model performance by comparing the predictions and the ground truth.

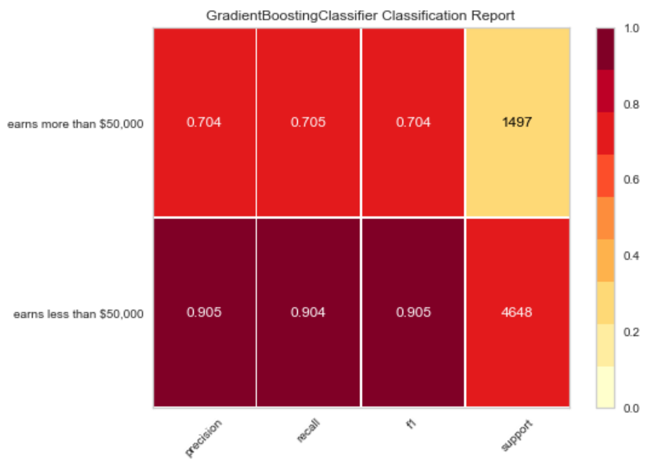


Fig. 22. Classification Report on Tuned Gradient Boosting

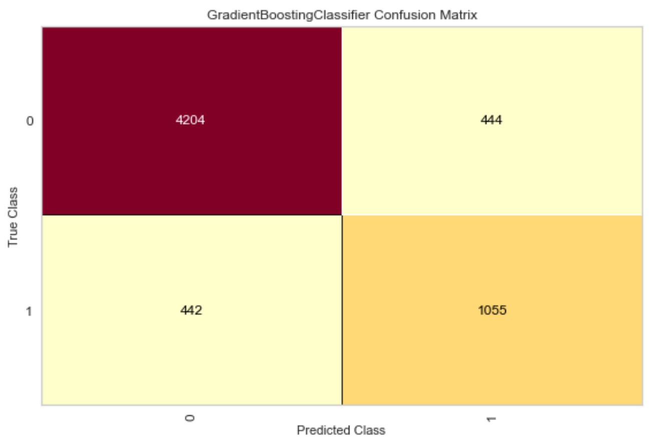


Fig. 23. Confusion Matrix on Tuned Gradient Boosting

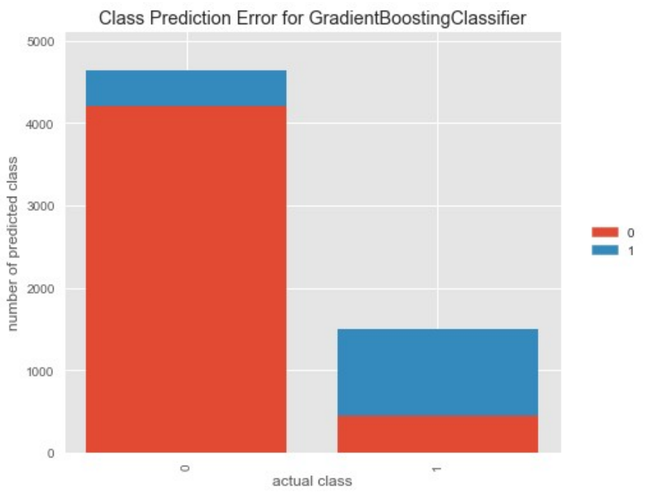


Fig. 24. Class Prediction error for Tuned Gradient Boosting

The classification report shown in figure 22, shows that the model is more biased towards predicting an individual earning less than $50,000. This is further supported by the figure 22 confusion matrix and figures 23 prediction error class, showing there is a proportion of false prediction earning more than $50,000. A possible explanation for this is due to the synthetic data, thus lacking in variability for the minority class in the training data.

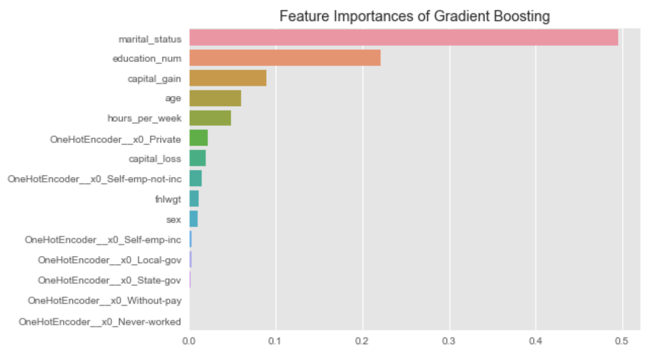


Fig. 25. Feature Importances

Figure 25, the feature importance shows a surprising outcome. The number one feature in terms of feature importance was marital status, for predicting yearly income. We could also see other features such as education number and capital gain being important in making a prediction, which is expected based on my general knowledge. Surprisingly enough, work class, in which I used One-Hot Encoding, was not so important in class prediction.

VIII. Conclusion

In conclusion, we have used Gradient Boosting as our go-to model to attempt this prediction task. I would say, the main thing I learn from this paper was learning and applying class balanced on the imbalanced target variable. The classifier used has shown great results, with an approximate 85% accuracy rate.

Fair to say, the findings of this mini project were quite interesting. Adult Income Classification is just one instance to demonstrate the use of Artificial Intelligence and Machine Learning; in the future, I hope to attempt more of these such mini projects to gain more relevant experiences concerning this field of study.

Acknowledgment

I thank Dr Wilson and Dr Peter, who implemented this technical paper section into the assignment. I would also like to thank Zhao Wu and Tien Cheng for teaching me the fundamentals of machine learning.

References

[1] Dheeru , D., &amp; Casey, G. (2017). Adult Income Census. Irvine; University of California, Irvine, School of Information and Computer Sciences.

[2] Lemaître, G., Nogueira , F., & Aridas, C. K. (2017). Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research*, *18*, 1–5.

[3] T., B. (2021, April 9). *11 Times Faster Hyperparameter Tuning with HalvingGridSearch*. Medium. https://towardsdatascience.com/11-times-faster-hyperparameter-tuning-with-halvinggridsearch-232ed0160155.

[4] Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, *16*, 321–357. https://doi.org/10.1613/jair.953

[5] Cheng, J., Greiner, R., Kelly, J., Bell, D., & Liu, W. (2002). Learning Bayesian networks from data: An information-theory based approach. *Artificial Intelligence*, *137*(1-2), 43–90. https://doi.org/10.1016/s0004-3702(02)00191-1

[6] Learning, U. C. I. M. (2016, October 7). *Adult Census Income*. Kaggle. https://www.kaggle.com/uciml/adult-census-income.

[7] *3.2. Tuning the hyper-parameters of an estimator*¶. scikit. (n.d.). https://scikit-learn.org/stable/modules/grid_search.html.