# From Naïve to XGBoost and ANN: Adult Census Income

| NAME | SID |
| --- | --- |
| BAI JUNYAN | 12212108 |

> *The source code is in file `project3.ipynb`, where each step is clearly clarified in the markdown part. So `Readme.md` is unneccessary.*

## Problem Formulation

The Adult Census Income data was extracted from the 1994 Census bureau database by Ronny Kohavi and Barry Becker (Data Mining and Visualization, Silicon Graphics). In this project, we need to train a model on the training dataset and predict whether the income would exceed 50K a year for each line of data in test dataset. Therefore, this will be a binary classification problem where input data is both continuous and categorical. In order to make predictions we will develop the following models:

- Logistic Regression
- Categorical Naïve Bayes
- Gaussian Naïve Bayes
- K-Nearest Neighbors
- Support Vector Machines
- Decision Trees
- Random Forest
- XGBoost
- Artificial Neural Networks

As usual, first we will have a look at the data in order to understand the different variables, then we will clean it in order to use it for prediction purposes, and finally we will develop the different models. These models will be tested with cross validation in order to pick the best ones and use them to develop an ensemble model with the

purpose of achieving more accuracy than each member of it. Then we will use these models to make predictions on the test dataset.

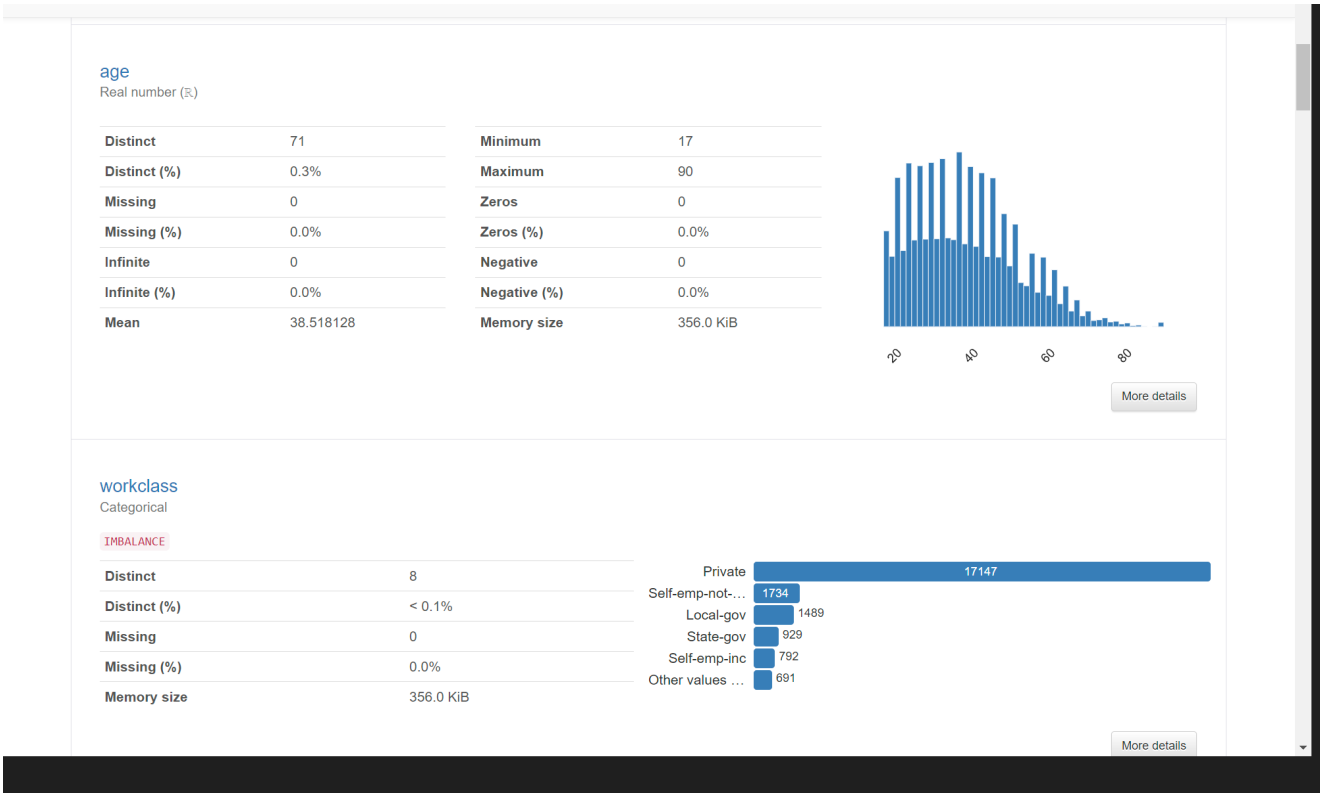
# Data Preprocessing

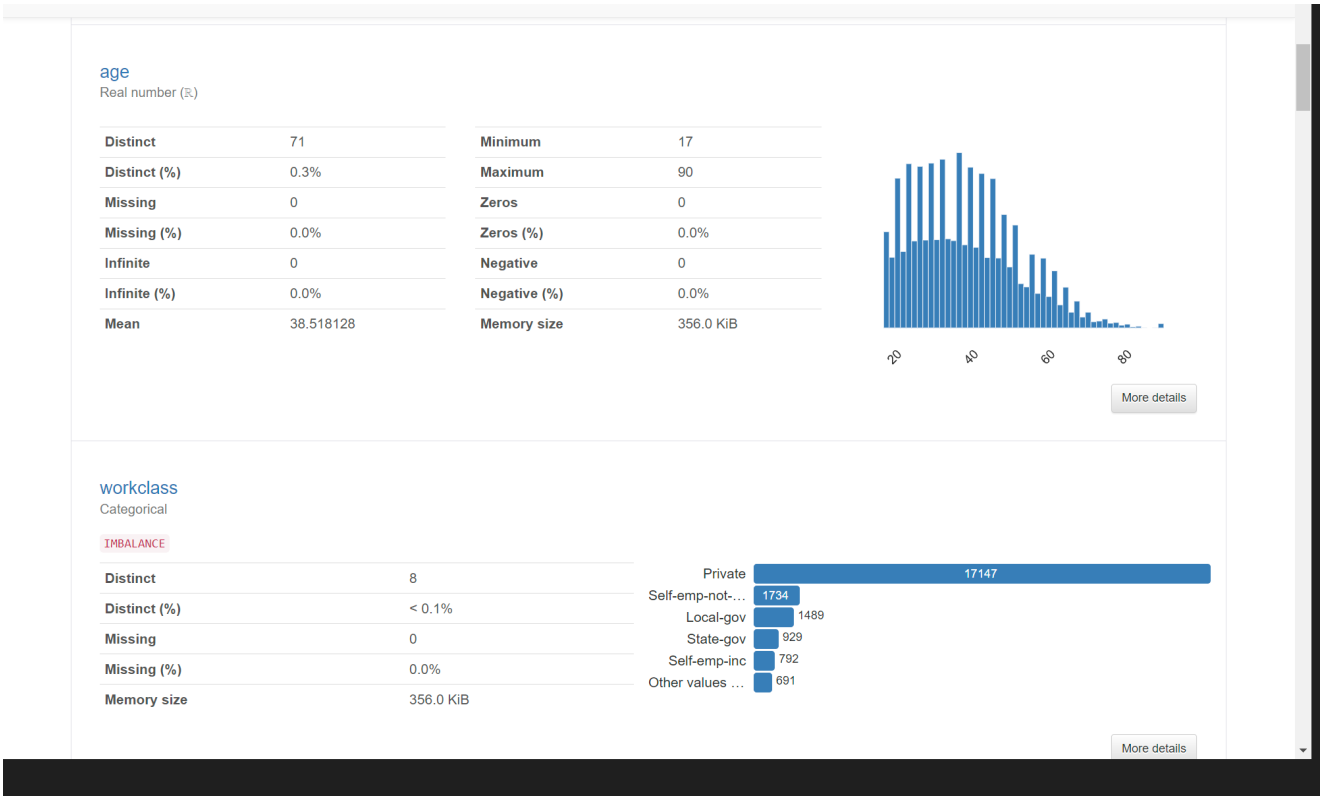
The data preprocessing step, as shown in the source code, could be divided into several stages:

- Data loading and overview
- Data analyzing and cleaning
- Data Visualization
- Prepare for the training and testing data

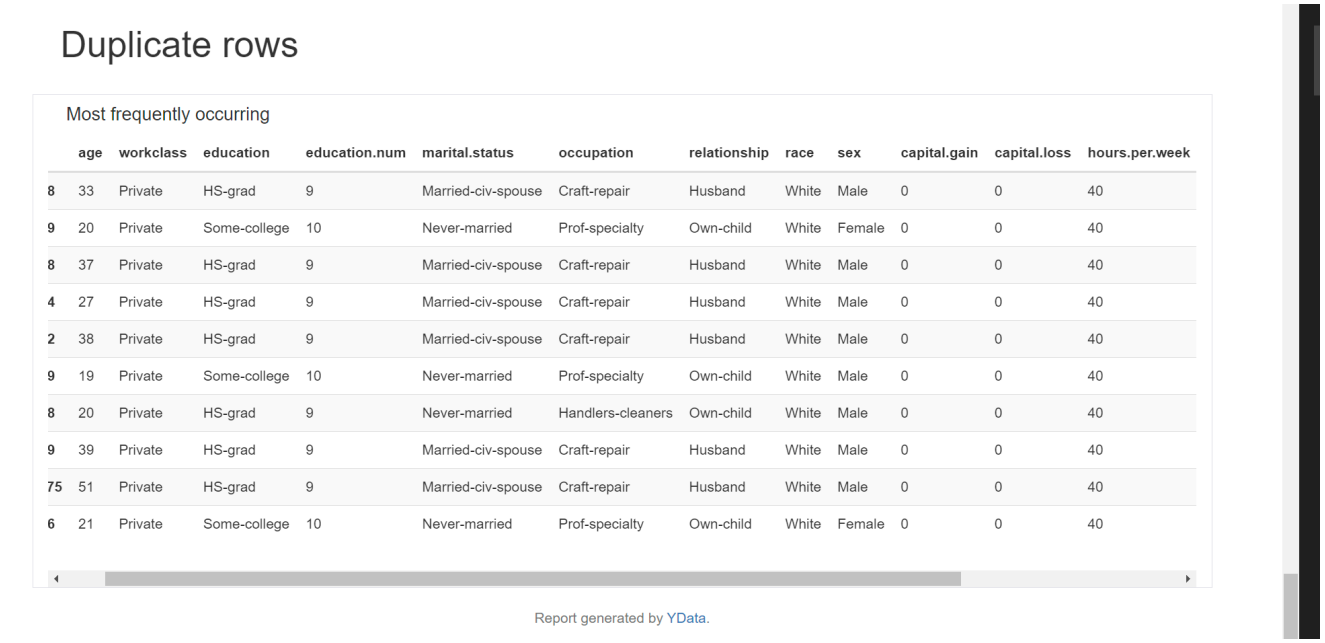
## Data Loading and Overview

Loading the training data using `panda` library, then using the method `ProfileReport` imported from `ydata_profiling` library, we can get an overview of the training dataset, which is highly detailed. Part of the result can be seen as below:






We can see from the "duplicated row" part that there are duplicated rows we need to deal with later:



Duplicate rows

Most frequently occurring

| | age | workclass | education | education.num | marital.status | occupation | relationship | race | sex | capital.gain | capital.loss | hours.per.week |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 33 | Private | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 |
| 9 | 20 | Private | Some-college | 10 | Never-married | Prof-specialty | Own-child | White | Female | 0 | 0 | 40 |
| 8 | 37 | Private | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 |
| 4 | 27 | Private | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 |
| 2 | 38 | Private | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 |
| 9 | 19 | Private | Some-college | 10 | Never-married | Prof-specialty | Own-child | White | Male | 0 | 0 | 40 |
| 8 | 20 | Private | HS-grad | 9 | Never-married | Handlers-cleaners | Own-child | White | Male | 0 | 0 | 40 |
| 9 | 39 | Private | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 |
| 75 | 51 | Private | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 |
| 6 | 21 | Private | Some-college | 10 | Never-married | Prof-specialty | Own-child | White | Female | 0 | 0 | 40 |

Report generated by YData.

## Data Analyzing and Cleaning

Running `train_data_set.info()`, we get:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22792 entries, 0 to 22791
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             22792 non-null  int64
```

```
 1   workclass       22792 non-null  object
 2   fnlwgt          22792 non-null  int64
 3   education       22792 non-null  object
 4   education.num   22792 non-null  int64
 5   marital.status  22792 non-null  object
 6   occupation      22792 non-null  object
 7   relationship    22792 non-null  object
 8   race            22792 non-null  object
 9   sex             22792 non-null  object
 10  capital.gain    22792 non-null  int64
 11  capital.loss    22792 non-null  int64
 12  hours.per.week  22792 non-null  int64
 13  native.country  22792 non-null  object
 14  income          22792 non-null  int64
dtypes: int64(7), object(8)
memory usage: 2.6+ MB
```

It seems that there is no null data, however, it can be easily seen that there are `?` data. We need to encode them as `NANS` first:

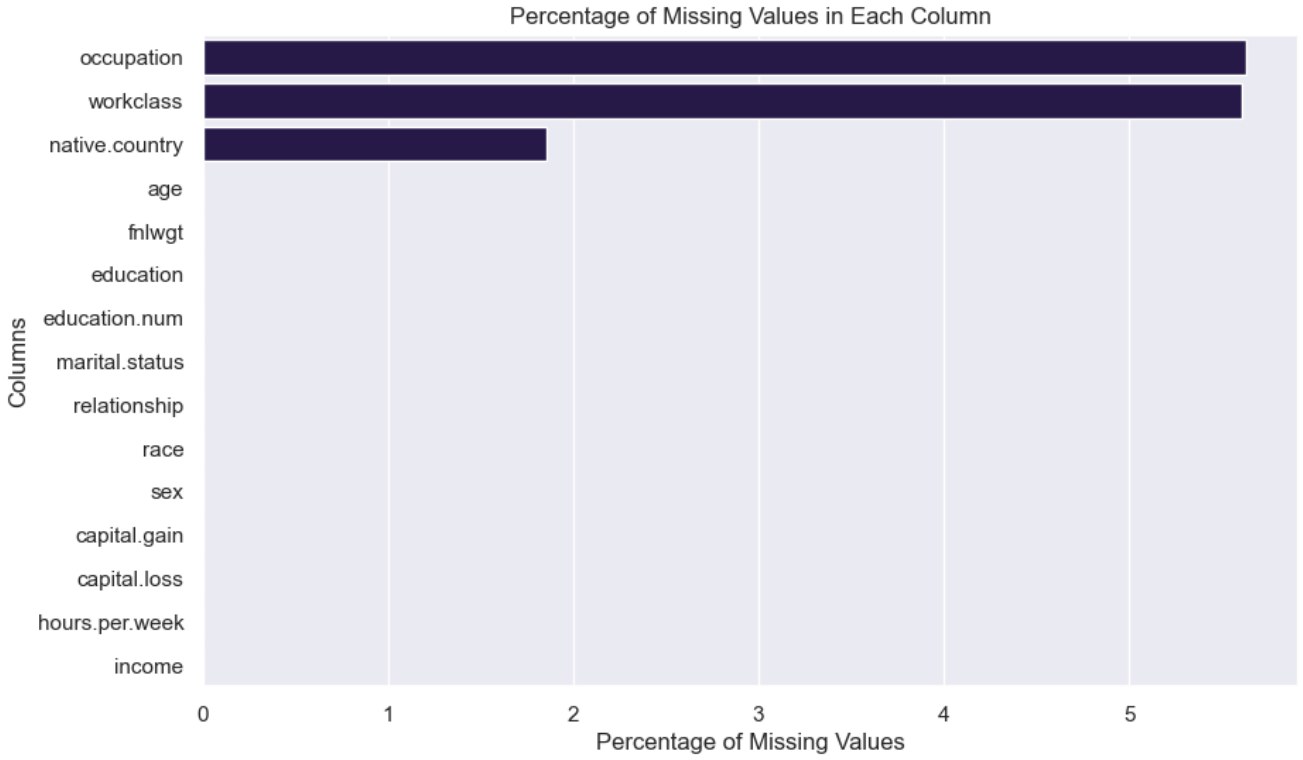
```
train_data_set[train_data_set == '?'] = np.nan
train_data_set.info()

--------------------------------------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22792 entries, 0 to 22791
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             22792 non-null  int64
 1   workclass       21514 non-null  object
 2   fnlwgt          22792 non-null  int64
 3   education       22792 non-null  object
 4   education.num   22792 non-null  int64
 5   marital.status  22792 non-null  object
 6   occupation      21509 non-null  object
 7   relationship    22792 non-null  object
 8   race            22792 non-null  object
 9   sex             22792 non-null  object
 10  capital.gain    22792 non-null  int64
 11  capital.loss    22792 non-null  int64
 12  hours.per.week  22792 non-null  int64
 13  native.country  22369 non-null  object
 14  income          22792 non-null  int64
dtypes: int64(7), object(8)
```

```
26  memory usage: 2.6+ MB
```

Then the result changes for some rows, to visualize the percentage of the missing data, we run

```
missing_value_percentage = train_data_set.isnull().mean()*100
missing_value_percentage_sorted =
missing_value_percentage.sort_values(ascending = False)
plt.figure(figsize=(10,6))
sns.barplot(x=missing_value_percentage_sorted,y=missing_value_perc
entage_sorted.index)
plt.title('Percentage of Missing Values in Each Column')
plt.xlabel('Percentage of Missing Values')
plt.ylabel('Columns')
plt.show()
```




Now, the summary shows that the variables - workclass, occupation and native.country contain missing values. All of these variables are categorical data type. So, I will impute the missing values with the most frequent value.

```
for col in ['workclass', 'occupation', 'native.country']:
    train_data_set[col].fillna(train_data_set[col].mode()[0],
inplace=True)
```

Then we need to deal with the duplicates, which are as follow. We run `train_data_set.duplicated().sum()` to delete the duplicated rows:

| | age | workclass | fnlwgt | education | education.num | marital.status | occupation | relationship | race | sex | capital.gain | capital.loss | hours.per.week | native.country | income |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 55 | 25 | Private | 195994 | 1st-4th | 2 | Never-married | Priv-house-serv | Not-in-family | White | Female | 0 | 0 | 40 | Guatemala | 0 |
| 704 | 23 | Private | 240137 | 5th-6th | 3 | Never-married | Handlers-cleaners | Not-in-family | White | Male | 0 | 0 | 55 | Mexico | 0 |
| 1961 | 46 | Private | 173243 | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 | United-States | 0 |
| 4108 | 44 | Private | 367749 | Bachelors | 13 | Never-married | Prof-specialty | Not-in-family | White | Female | 0 | 0 | 45 | Mexico | 0 |
| 5721 | 44 | Private | 367749 | Bachelors | 13 | Never-married | Prof-specialty | Not-in-family | White | Female | 0 | 0 | 45 | Mexico | 0 |
| 8088 | 25 | Private | 308144 | Bachelors | 13 | Never-married | Craft-repair | Not-in-family | White | Male | 0 | 0 | 40 | Mexico | 0 |
| 8254 | 21 | Private | 243368 | Preschool | 1 | Never-married | Farming-fishing | Not-in-family | White | Male | 0 | 0 | 50 | Mexico | 0 |
| 9196 | 42 | Private | 204235 | Some-college | 10 | Married-civ-spouse | Prof-specialty | Husband | White | Male | 0 | 0 | 40 | United-States | 1 |
| 9830 | 20 | Private | 107658 | Some-college | 10 | Never-married | Tech-support | Not-in-family | White | Female | 0 | 0 | 10 | United-States | 0 |
| 9853 | 25 | Private | 195994 | 1st-4th | 2 | Never-married | Priv-house-serv | Not-in-family | White | Female | 0 | 0 | 40 | Guatemala | 0 |
| 11332 | 39 | Private | 30916 | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 | United-States | 0 |
| 13676 | 27 | Private | 255582 | HS-grad | 9 | Never-married | Machine-op-inspct | Not-in-family | White | Female | 0 | 0 | 40 | United-States | 0 |
| 15108 | 20 | Private | 107658 | Some-college | 10 | Never-married | Tech-support | Not-in-family | White | Female | 0 | 0 | 10 | United-States | 0 |
| 16958 | 42 | Private | 204235 | Some-college | 10 | Married-civ-spouse | Prof-specialty | Husband | White | Male | 0 | 0 | 40 | United-States | 1 |
| 18372 | 25 | Private | 308144 | Bachelors | 13 | Never-married | Craft-repair | Not-in-family | White | Male | 0 | 0 | 40 | Mexico | 0 |
| 18382 | 23 | Private | 240137 | 5th-6th | 3 | Never-married | Handlers-cleaners | Not-in-family | White | Male | 0 | 0 | 55 | Mexico | 0 |
| 19488 | 46 | Private | 173243 | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 | United-States | 0 |
| 19733 | 39 | Private | 30916 | HS-grad | 9 | Married-civ-spouse | Craft-repair | Husband | White | Male | 0 | 0 | 40 | United-States | 0 |
| 20026 | 27 | Private | 255582 | HS-grad | 9 | Never-married | Machine-op-inspct | Not-in-family | White | Female | 0 | 0 | 40 | United-States | 0 |
| 20405 | 21 | Private | 243368 | Preschool | 1 | Never-married | Farming-fishing | Not-in-family | White | Male | 0 | 0 | 50 | Mexico | 0 |

To inspect the useless features, we count the number of each features that are not unique by

```
train_data_set.nunique().sort_values

--------------------------------------
<bound method Series.sort_values of
age                 71
workclass            8
fnlwgt           16645
education           16
education.num       16
marital.status       7
occupation          14
relationship         6
race                 5
sex                  2
capital.gain       117
capital.loss        86
hours.per.week      92
native.country      40
income               2
dtype: int64>
```

Then we may drop the row `fnlwgt` since the number of non-uniqueness is extremely huge.

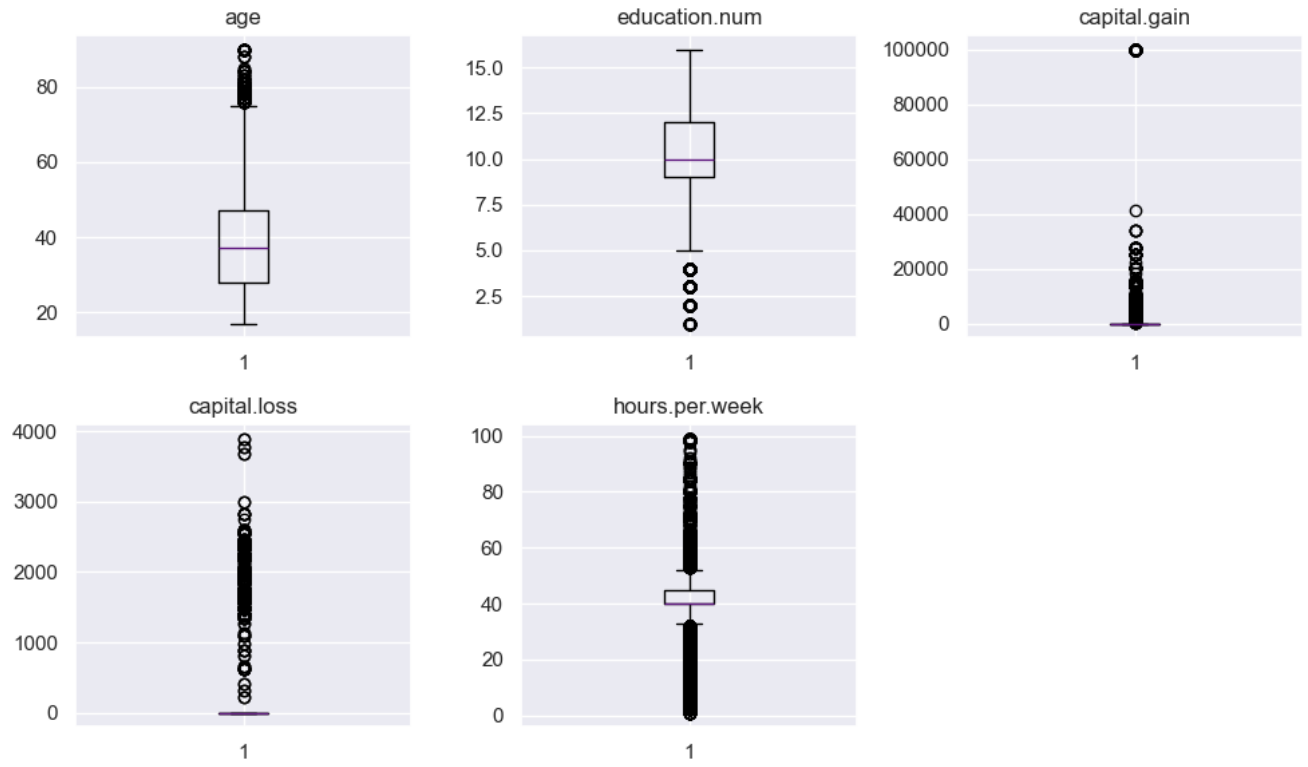
## Data Visualization

The remaining columns can be divided into two categories: **Numeric Values** and **Non-Numeric Values**. We firstly draw the boxplot for each columns to see the distributions of the specific datas:

```
numeric_columns = ['age', 'education.num', 'capital.gain',
'capital.loss', 'hours.per.week']

plt.figure(figsize=(10, 6))
for i, column in enumerate(numeric_columns, 1):
    plt.subplot(2, 3, i)
    plt.boxplot(train_data_set[column])
    plt.title(column)
    plt.grid(True)

plt.tight_layout()
plt.show()
```



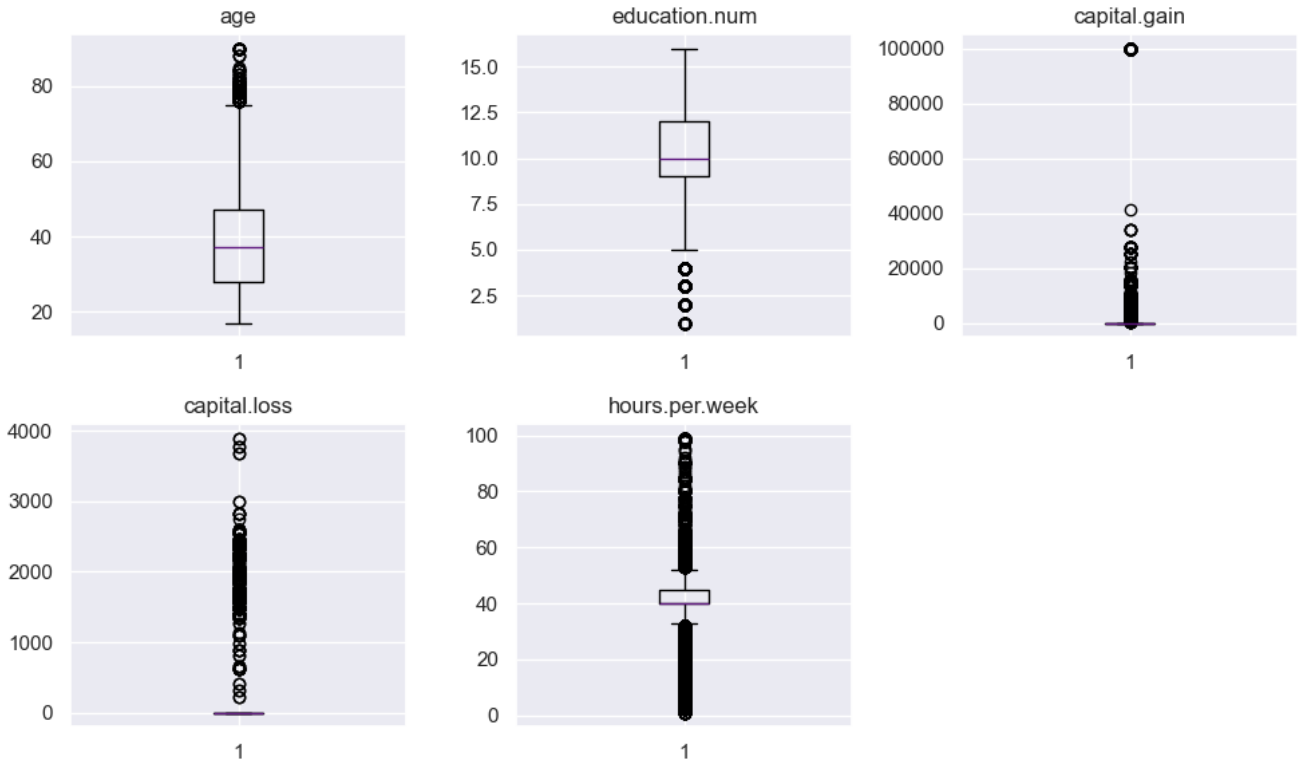

```
condition = (train_data_set['capital.gain'] > 40000) |
(train_data_set['capital.loss'] > 3500)
df = train_data_set[~condition]

numeric_columns = ['age', 'education.num', 'capital.gain',
'capital.loss', 'hours.per.week']

plt.figure(figsize=(10, 6))
for i, column in enumerate(numeric_columns, 1):
```

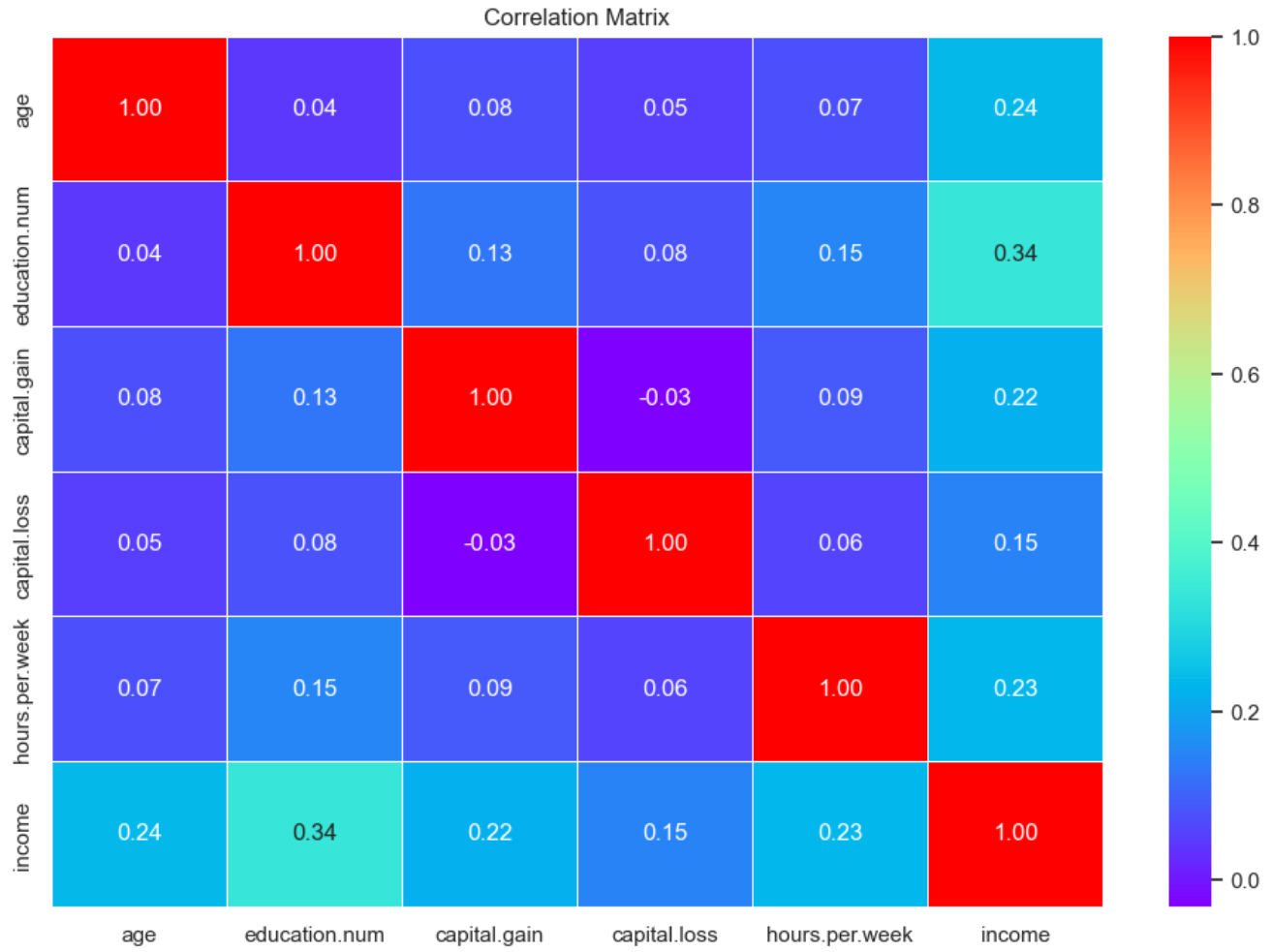
```
    plt.subplot(2, 3, i)
    plt.boxplot(train_data_set[column])
    plt.title(column)
    plt.grid(True)

plt.tight_layout()
plt.show()
```




Then we draw the correlation matrix for the numeric columns and add the column `income` into it.

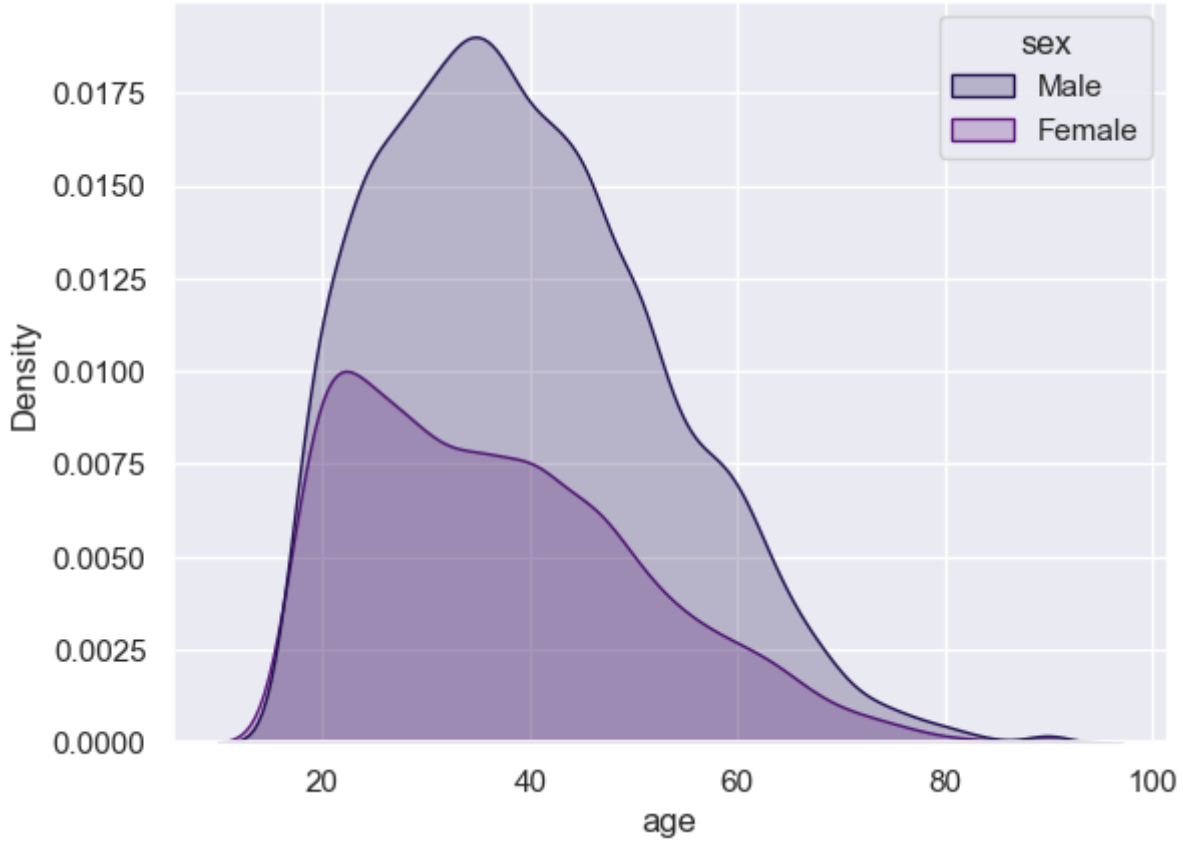
```
numeric_columns = train_data_set.select_dtypes(include=['number'])
correlation_matrix = numeric_columns.corr()
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='rainbow',
fmt=".2f", linewidths=0.5)
plt.title('Correlation Matrix')
```



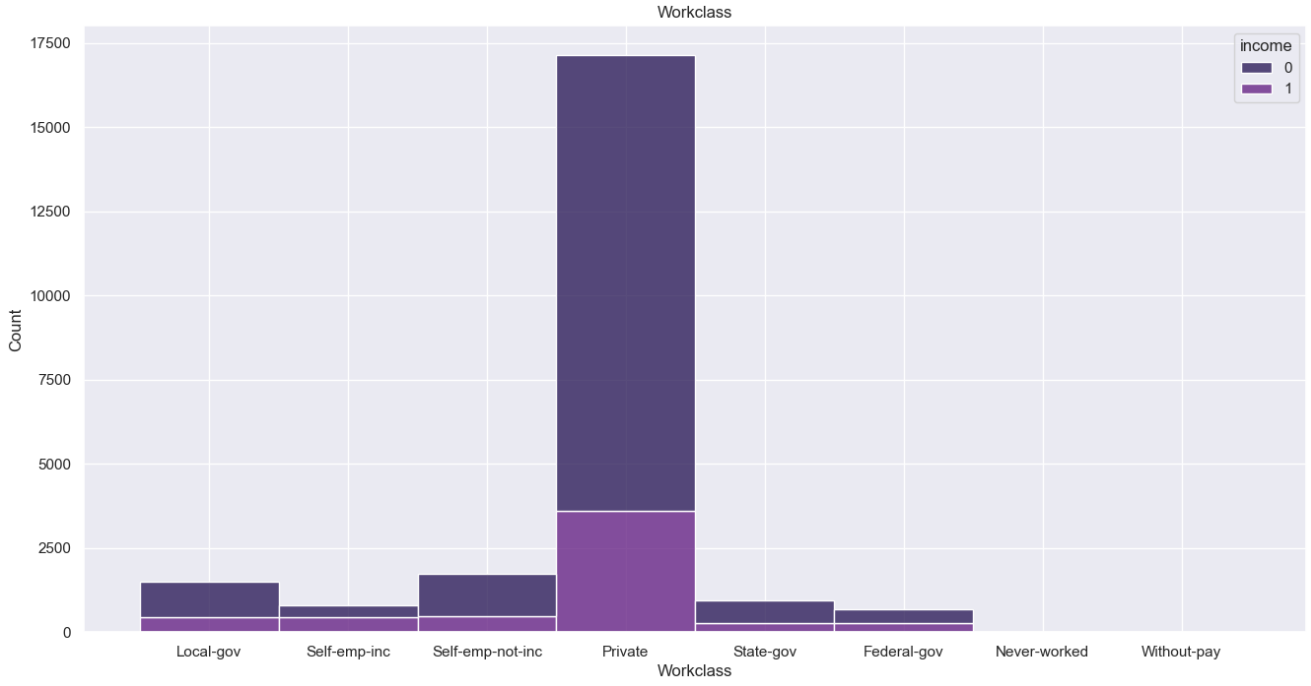

From the result, the correlation between each pair of columns is rather small, which shows that to make precise predictions, we need all of the numeric columns. And it can also tell from the graph that `education.num` has the strongest relation with `income`.

For each pair of numeric values, we also draw the curve graph to see more detailed relation between them:

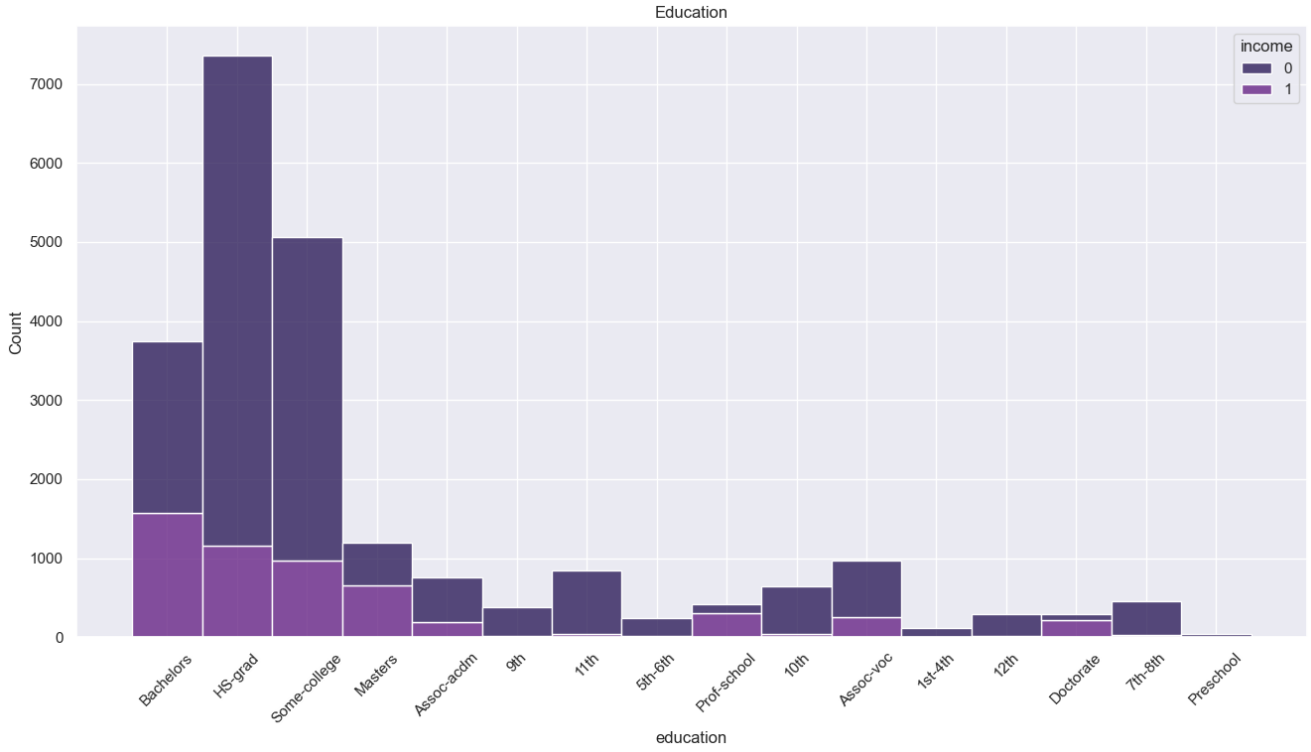
People of different ages are of normal distribution, and it seems that people around 50 years old have the highest income.



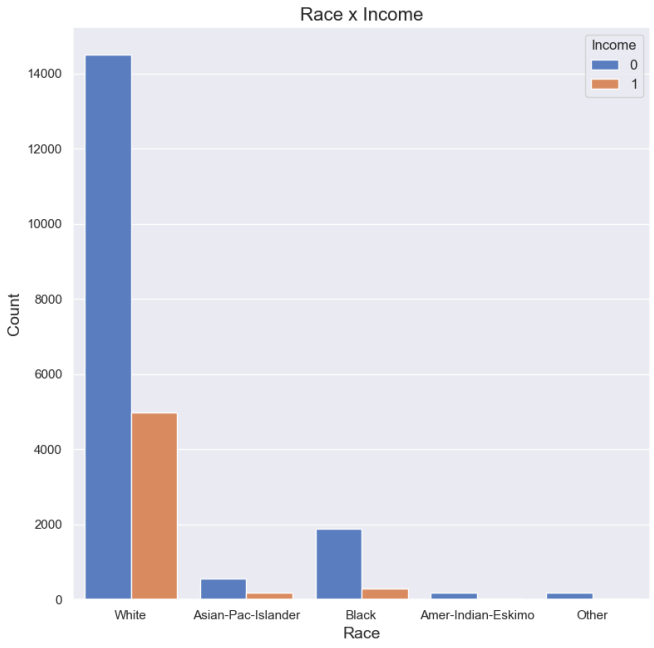

The relation between sex and age cannot tell us anything but fact that the participants are mostly men of middle ages.

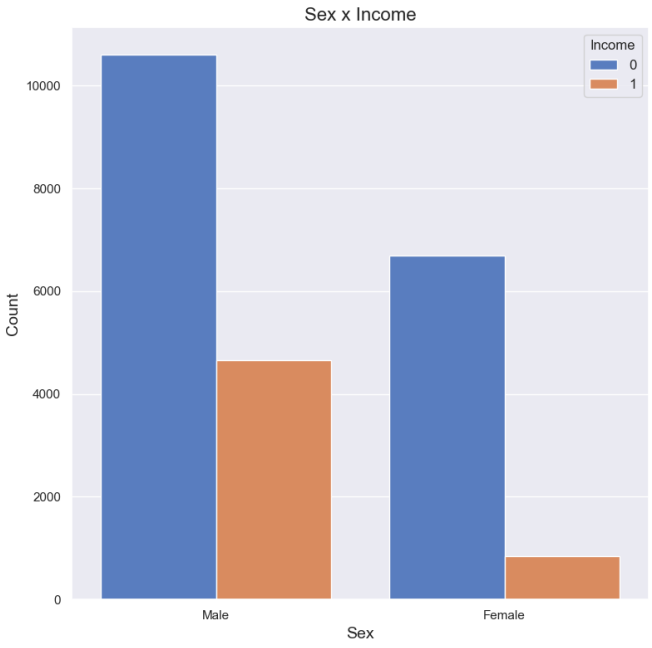


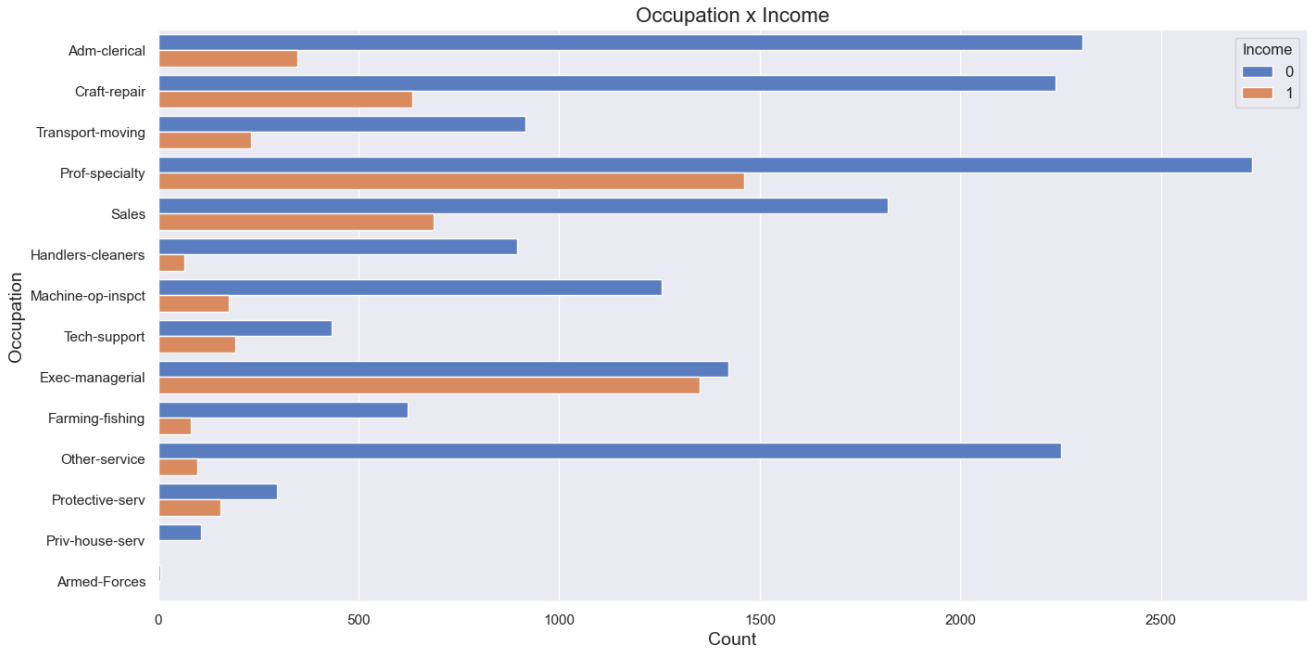

It seems that work class makes no contribution to the income, which is against the common senses. The cause is that most information is "private", which explains nothing.

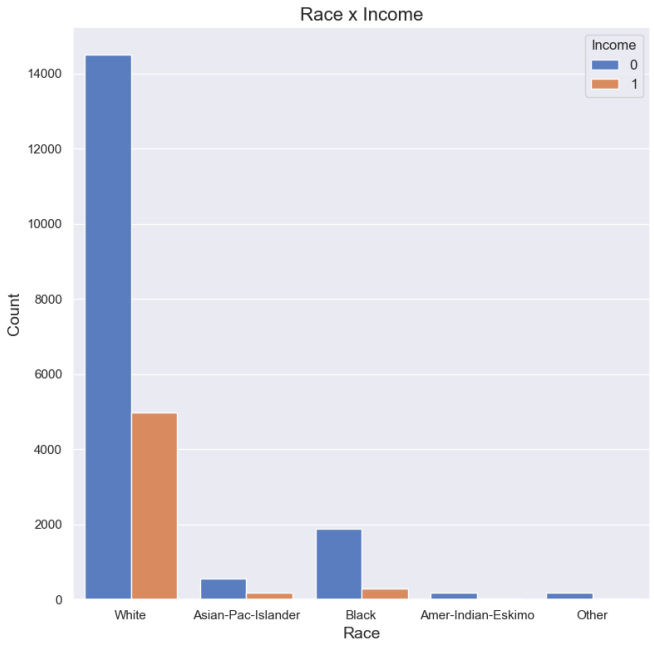


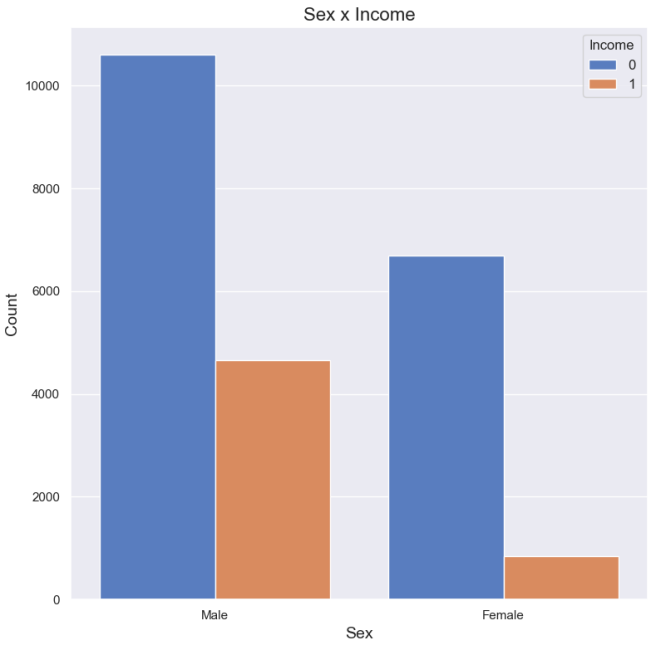

Obviously, people with higher education tends to have higher income.





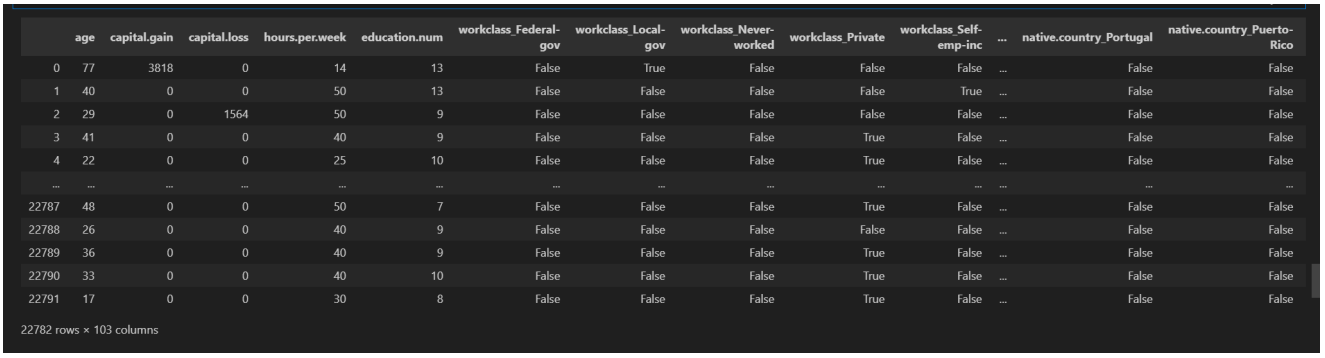




## Prepare for the training and testing data

To make predictions on the testing dataset, we need to conduct similar operations on testing dataset as we did to training dataset. Also, we split the columns into **continuous variables** and **categorical variable** and get the dummy data of categorical and change the original both datasets into new forms with 103 columns compared to the original 14 columns:

```
X_continous  = X[['age', 'capital.gain', 'capital.loss', 'hours.per.week','education.num']]

X_categorical = X[['workclass', 'education', 'marital.status', 'occupation', 'relationship', 'race',
                   'sex', 'native.country']]
# Get the dummies
X_encoded = pd.get_dummies(X_categorical)
# Concatenate both continous and encoded sets:
X = pd.concat([X_continous, X_encoded],axis=1)
X
```



| | age | capital.gain | capital.loss | hours.per.week | education.num | workclass_Federal-gov | workclass_Local-gov | workclass_Never-worked | workclass_Private | workclass_Self-emp-inc | ... | native.country_Portugal | native.country_Puerto-Rico |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 77 | 3818 | 0 | 14 | 13 | False | True | False | False | False | ... | False | False |
| 1 | 40 | 0 | 0 | 50 | 13 | False | False | False | False | True | ... | False | False |
| 2 | 29 | 0 | 1564 | 50 | 9 | False | False | False | False | False | ... | False | False |
| 3 | 41 | 0 | 0 | 40 | 9 | False | False | False | True | False | ... | False | False |
| 4 | 22 | 0 | 0 | 25 | 10 | False | False | False | True | False | ... | False | False |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 22787 | 48 | 0 | 0 | 50 | 7 | False | False | False | True | False | ... | False | False |
| 22788 | 26 | 0 | 0 | 40 | 9 | False | False | False | False | False | ... | False | False |
| 22789 | 36 | 0 | 0 | 40 | 9 | False | False | False | True | False | ... | False | False |
| 22790 | 33 | 0 | 0 | 40 | 10 | False | False | False | True | False | ... | False | False |
| 22791 | 17 | 0 | 0 | 30 | 8 | False | False | False | True | False | ... | False | False |

22782 rows × 103 columns

# Modeling

As mentioned at the beginning of the report, different models are chosen to predict the income of test dataset. For each model, the procedure is:

- Processing the data(scaling) to fit into the model
- For complex models with hyperparameters involved, optimization will be done using `GridSearchCV`, which would take much time to complete.
- Modeling on the training dataset
- Cross Validation of the model on the training dataset and calculated the accuracy as the criteria of evaluation of the model
- Make predictions on the testing dataset

## Logistic Regression

Use `LogisticRegression` method of `sklearn` library to model and `StratifiedKFold` to do cross validation:

```
#MODEL
logit = LogisticRegression(max_iter=10000)
logit = logit.fit(X,y)

#Cross Validation
cv = StratifiedKFold(n_splits=3)
val_logit = cross_val_score(logit,X,y,cv=cv).mean()
val_logit

------------------------------
0.8507593714335879
```

The result of cross validation is 0.85, which is rather good.

Then we use the model on the test dataset by calling `predict` and store the result into `LogicRegression.txt`:

```
# PREDICTIONS
logit_predictions = logit.predict(T)
with open('LogicRegression.txt', 'w') as f:
    for prediction in logit_predictions:
        f.write(f"{prediction}\n")

print("Predictions saved to LogicRegression.txt")
```

## Categorical Naive Bayes

To use Categorical Naive Bayes as the model, we should only look at the **categorical** columns of dataset. Then use `CategoricalNB` method of `sklearn` library to model and do cross validation:

```
# Prepare the test data, we only use categorical independent
variables
column_difference = set(T_encoded.columns) -
set(X_encoded.columns)
T_encoded.drop(columns=column_difference,inplace=True)

# Modeling
cnb = CategoricalNB()
cnb = cnb.fit(X_encoded,y)

#Cross Validation
cv = StratifiedKFold(n_splits=3)
```

```
val_logit = cross_val_score(cnb,X_encoded,y,cv=cv).mean()
val_logit


----------------------------------------
0.7501097357562988
```

The result of cross validation is 0.75, which indicates that it is much worse than Logistic Regression when modeling this dataset, since we only use the categorical columns to model.

Then we use the model on the test dataset by calling `predict` and store the result into `CNB.txt`:

```
# PREDICTOONS
cnb_predictions = cnb.predict(T_encoded)
with open('CNB.txt', 'w') as f:
    for prediction in cnb_predictions:
        f.write(f"{prediction}\n")

print("Predictions saved to CNB.txt")
```

## Gaussian Naive Bayes

To use Categorical Naive Bayes as the model, we should only look at the **numeric** columns of dataset. Then use `GaussianNB` method of `sklearn` library to model and do cross validation:

```
# Prepare the data. We only use continous independent variables
column_difference = set(T_continous.columns) -
set(X_continous.columns)
T_continous.drop(columns=column_difference,inplace=True)

#Modeling
gnb = GaussianNB()
gnb = gnb.fit(X_continous,y)

#Cross Validation
cv = StratifiedKFold(n_splits=3)
val_logit = cross_val_score(gnb,X_continous,y,cv=cv).mean()
val_logit


-------------------------------------
```

```
15  0.7949697129312615
```

The result of cross validation is 0.79, which is better than Categorical Naive Bayes. It shows that the numeric columns contribute more to the `income`.

Then we use the model on the test dataset by calling `predict` and store the result into `GNB.txt`:

```
# PREDICTIONS
gnb_predictions = gnb.predict(T_continous)
with open('GNB.txt', 'w') as f:
    for prediction in gnb_predictions:
        f.write(f"{prediction}\n")

print("Predictions saved to GNB.txt")
```

## K-Nearest Neighbors

KNN is a supervised learning model, to use this model on the training dataset, we should scale both the training and testing data in a range of (0,1) since this model is only distance-based.

```
# Prepare the data. We scale the data as this algorithm is
distance-based
# scale data in a range of (0,1)
scaler = MinMaxScaler(feature_range=(0, 1))
scaler = scaler.fit(X)
X_train = scaler.transform(X)
X_test = scaler.transform(T)
```

Then we should choose the hyper parameter `k` evaluated by cross validation. I choose 40, 50, 60, 70, 80 at first and it shows that 50 is the best parameter. Then choosing the parameter around 50 and narrowing the range gradually. After several rounds of testing, the best parameter is **47**. The process of hyper parameter tuning is done through `GridSearchCV` in `sklearn.model_selection`:

```
param_grid = {'n_neighbors' : [45,46,47,48]}
cv = StratifiedKFold(n_splits=3)

optimal_params = GridSearchCV(
    estimator = KNeighborsClassifier(),
    param_grid = param_grid,
    scoring = 'accuracy',
    verbose = 2,
    cv = cv,
)
optimal_params.fit(X_train,y)
optimal_params.best_estimator_
```

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits
[CV] END .....................................n_neighbors=45; total time=   0.4s
[CV] END .....................................n_neighbors=45; total time=   0.5s
[CV] END .....................................n_neighbors=45; total time=   0.6s
[CV] END .....................................n_neighbors=46; total time=   0.5s
[CV] END .....................................n_neighbors=46; total time=   0.5s
[CV] END .....................................n_neighbors=46; total time=   0.5s
[CV] END .....................................n_neighbors=47; total time=   0.4s
[CV] END .....................................n_neighbors=47; total time=   0.4s
[CV] END .....................................n_neighbors=47; total time=   0.5s
[CV] END .....................................n_neighbors=48; total time=   0.5s
[CV] END .....................................n_neighbors=48; total time=   0.4s
[CV] END .....................................n_neighbors=48; total time=   0.9s

▾      KNeighborsClassifier
KNeighborsClassifier(n_neighbors=47)
```

Then we call `KNeighborClassifier` with `n_neighbors = 47` and testing by cross validation:

```
# MODEL
knn = KNeighborsClassifier(n_neighbors=47)
knn = knn.fit(X_train,y)

# CROSS VALIDATION
cv = StratifiedKFold(n_splits=3)
val_knn = cross_val_score(knn, X_train, y, cv=cv).mean()
val_knn # validation score

---------------------------------
0.8362303572996225
```

The result shows that **KNN** is better than the two **Naive Bayes** models while slightly inferior to **Logic Regression**. After modeling, we use the model on the test dataset by calling `predict` and store the result into `KNN.txt`:

```
# PREDICTIONS
knn_predictions = knn.predict(X_test)

with open('KNN.txt', 'w') as f:
    for prediction in knn_predictions:
        f.write(f"{prediction}\n")

print("Predictions saved to KNN.txt")
```

## Support Vector Machines

SVM could also be used to model on this dataset. Since it's also distance-based, we should scale the dataset first:

```
# Scale the data(mean = 0 and sd = 1)
X_train = scale(X)
X_test = scale(T)
```

Then we should also do hyperparameters optimization, which is to choose the kernel function of SVM.

> ***Kernel Function*** *is a method used to take data as input and transform it into the required form of processing data. "Kernel" is used due to a set of mathematical functions used in Support Vector Machine providing the window to manipulate the data. So, Kernel Function generally transforms the training set of data so that a non-linear decision surface is able to transform to a linear equation in a higher number of dimension spaces. Basically, It returns the inner product between two points in a standard feature dimension. ----Geeksforgeek*

We use `GridSearchCV` to find the best parameter among **Linear Kernel**, **Polynomial Kernel**, **Sigmoid Kernel** and **Gaussian Kernel Radial Basis Function**.

```
# HyperParameters Optimization
#1 Round -- Choose the Kernals
param_grid = {
    'kernel' : ['linear','poly','rbf','sigmoid']
}
cv = StratifiedKFold(n_splits=3)

optimal_params = GridSearchCV(
    estimator = svm.SVC(),
    param_grid = param_grid,
```

```
    scoring = 'accuracy',
    verbose = 2,
    cv = cv
)
optimal_params.fit(X_train,y)
optimal_params.best_params_
```

```
Fitting 3 folds for each of 4 candidates, totalling 12 fits
[CV] END .....................................kernel=linear; total time=  11.5s
[CV] END .....................................kernel=linear; total time=  11.0s
[CV] END .....................................kernel=linear; total time=  10.6s
[CV] END .......................................kernel=poly; total time=   6.4s
[CV] END .......................................kernel=poly; total time=   6.5s
[CV] END .......................................kernel=poly; total time=   6.7s
[CV] END ........................................kernel=rbf; total time=   9.7s
[CV] END ........................................kernel=rbf; total time=   9.3s
[CV] END ........................................kernel=rbf; total time=  14.2s
[CV] END ....................................kernel=sigmoid; total time=  10.7s
[CV] END ....................................kernel=sigmoid; total time=  10.9s
[CV] END ....................................kernel=sigmoid; total time=  11.0s

{'kernel': 'linear'}
```

Surprisingly, the **Linear Function** is chosen, which is the simplest one. We can further investigate how much better it is with respect to other kernels:

| | params | mean_test_score |
|---|---|---|
| 0 | {'kernel': 'linear'} | 0.849223 |
| 1 | {'kernel': 'poly'} | 0.826047 |
| 2 | {'kernel': 'rbf'} | 0.846589 |
| 3 | {'kernel': 'sigmoid'} | 0.831183 |

So **RBF** is nearly as good as **Linear**, so we will try to improve it through tuning. We will try to tune by deciding on the hyperparameter C.

> *The C parameter trades off correct classification of training examples against maximization of the decision function's margin. For larger values of C, a smaller margin will be accepted if the decision function is better at classifying all training points correctly. A lower C will encourage a larger margin, therefore a simpler decision function, at the cost of training accuracy. In other words C behaves as a regularization parameter in the SVM. --- scikit learn*

```
param_grid = {
```

```
    'kernel': ['rbf'],
    'C' : [0,1,2,3,4,5],
}
cv = StratifiedKFold(n_splits=3)
optimal_params = GridSearchCV(
    estimator = svm.SVC(),
    param_grid = param_grid,
    scoring = 'accuracy',
    verbose = 2,
    cv = cv
)
optimal_params.fit(X_train,y)
print(
    "The best parameters are %s with a score of %0.2f"
    % (optimal_params.best_params_, optimal_params.best_score_)
)
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
[CV] END .....................................C=0, kernel=rbf; total time=   0.0s
[CV] END .....................................C=0, kernel=rbf; total time=   0.0s
[CV] END .....................................C=0, kernel=rbf; total time=   0.0s
[CV] END .....................................C=1, kernel=rbf; total time=   8.3s
[CV] END .....................................C=1, kernel=rbf; total time=   9.5s
[CV] END .....................................C=1, kernel=rbf; total time=  10.6s
[CV] END .....................................C=2, kernel=rbf; total time=  17.0s
[CV] END .....................................C=2, kernel=rbf; total time=  17.1s
[CV] END .....................................C=2, kernel=rbf; total time=  16.2s
[CV] END .....................................C=3, kernel=rbf; total time=  12.1s
[CV] END .....................................C=3, kernel=rbf; total time=  13.5s
[CV] END .....................................C=3, kernel=rbf; total time=   8.0s
[CV] END .....................................C=4, kernel=rbf; total time=   8.0s
[CV] END .....................................C=4, kernel=rbf; total time=   8.1s
[CV] END .....................................C=4, kernel=rbf; total time=   8.1s
[CV] END .....................................C=5, kernel=rbf; total time=   8.2s
[CV] END .....................................C=5, kernel=rbf; total time=   8.1s
[CV] END .....................................C=5, kernel=rbf; total time=   8.2s
The best parameters are {'C': 2, 'kernel': 'rbf'} with a score of 0.85
```

Still, the result shows that **RBF** is worse than that of the **Linear Kernel**, so we will continue to use linear kernel instead. Calling `svm.SVC()` to model on the training dataset and use cross validation to evaluate the model:

```
# MODEL
suppvm = svm.SVC(kernel='linear')
suppvm = suppvm.fit(X_train,y)
# CROSS VALIDATION
cv = StratifiedKFold(n_splits=3)
val_suppvm = cross_val_score(suppvm, X_train, y, cv=cv).mean()
val_suppvm # validation score

-----------------------------
0.8492230708454042
```

We have the score 0.849, which is as good as that of Logistic Regression. Then we call `predict()` and store the result into `SVM.txt`:

```
# PREDICTIONS
suppvm_predictions = suppvm.predict(X_test)
with open('SVM.txt', 'w') as f:
    for prediction in suppvm_predictions:
        f.write(f"{prediction}\n")

print("Predictions saved to SVM.txt")
```

## Decision Trees

First of all we need to tune the hyperparameter `max_depth` of Decision Trees classifier:

```
# HYPERPARAMETERS OPTIMIZATION
param_grid = {
'max_depth' : [2,4,6,7,8,9,10,11,12,16,20]
}

cv = StratifiedKFold(n_splits=3)

optimal_params = GridSearchCV(
    estimator = DecisionTreeClassifier(),
    param_grid = param_grid,
    scoring = 'accuracy',
    verbose = 2,
    cv = cv
)
optimal_params.fit(X,y)
optimal_params.best_params_
```

```
Fitting 3 folds for each of 11 candidates, totalling 33 fits
[CV] END .....................................max_depth=2; total time=   0.0s
[CV] END .....................................max_depth=2; total time=   0.0s
[CV] END .....................................max_depth=2; total time=   0.0s
[CV] END .....................................max_depth=4; total time=   0.0s
[CV] END .....................................max_depth=4; total time=   0.0s
[CV] END .....................................max_depth=4; total time=   0.0s
[CV] END .....................................max_depth=6; total time=   0.0s
[CV] END .....................................max_depth=6; total time=   0.0s
[CV] END .....................................max_depth=6; total time=   0.0s
[CV] END .....................................max_depth=7; total time=   0.0s
[CV] END .....................................max_depth=7; total time=   0.0s
[CV] END .....................................max_depth=7; total time=   0.0s
[CV] END .....................................max_depth=8; total time=   0.0s
[CV] END .....................................max_depth=8; total time=   0.0s
[CV] END .....................................max_depth=8; total time=   0.0s
[CV] END .....................................max_depth=9; total time=   0.0s
[CV] END .....................................max_depth=9; total time=   0.0s
[CV] END .....................................max_depth=9; total time=   0.0s
[CV] END ....................................max_depth=10; total time=   0.0s
[CV] END ....................................max_depth=10; total time=   0.0s
[CV] END ....................................max_depth=10; total time=   0.0s
[CV] END ....................................max_depth=11; total time=   0.0s
[CV] END ....................................max_depth=11; total time=   0.0s
[CV] END ....................................max_depth=11; total time=   0.0s
...
[CV] END ....................................max_depth=16; total time=   0.0s
[CV] END ....................................max_depth=20; total time=   0.0s
[CV] END ....................................max_depth=20; total time=   0.0s
[CV] END ....................................max_depth=20; total time=   0.0s
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

{'max_depth': 8}
```

After choosing the best `max_depth=8`, we can do modeling by calling `DecisionTreeClassifier`.

```
# MODEL
tree = DecisionTreeClassifier(max_depth=8)
tree = tree.fit(X,y)

# CROSS VALIDATION
cv = StratifiedKFold(n_splits=3)
val_tree = cross_val_score(tree, X, y, cv=cv).mean()
val_tree # validation score

--------------------
0.8546220700553068
```

Now **Decision Trees** defeat Logistic Regression to become the best model yet. Then we just need to make predictions:

```
tree_predicitons = tree.predict(T)
with open('DecisionTree.txt', 'w') as f:
    for prediction in tree_predicitons:
        f.write(f"{prediction}\n")

print("Predictions saved to DecisionTree.txt")
```

## Random Forest

There are three hyperparameters to tune: `max_depth`, `n_estimators` and `max_samples`.

- **max_depth**: This parameter controls the maximum depth of each decision tree in the forest. A higher depth allows the model to capture more details of the data but can also lead to overfitting. The values specified are 8, 10, 12, 16, 18, and 20.
- **n_estimators**: This parameter specifies the number of trees in the forest. More trees generally improve performance but also increase computational cost. The values specified are 50, 100, and 200.
- **max_samples**: This parameter determines the fraction of the training data to be used for fitting each individual tree. Values specified are 1 (100% of the training data), 0.8 (80% of the training data), and 0.6 (60% of the training data).

```
# HYPERPARAMETERS OPTIMIZATION
param_grid = {
'max_depth' : [8,10,12,16,18,20],
'n_estimators': [50,100,200],
'max_samples': [1,0.8,0.6]
}

cv = StratifiedKFold(n_splits=3)

optimal_params = GridSearchCV(
    estimator = RandomForestClassifier(),
    param_grid = param_grid,
    scoring = 'accuracy',
    verbose = 2,
    cv = cv
)
optimal_params.fit(X,y)
```

```
optimal_params.best_params_
```

```
Fitting 3 folds for each of 54 candidates, totalling 162 fits
[CV] END ........max_depth=8, max_samples=1, n_estimators=50; total time=   0.0s
[CV] END ........max_depth=8, max_samples=1, n_estimators=50; total time=   0.0s
[CV] END ........max_depth=8, max_samples=1, n_estimators=50; total time=   0.0s
[CV] END .......max_depth=8, max_samples=1, n_estimators=100; total time=   0.1s
[CV] END .......max_depth=8, max_samples=1, n_estimators=100; total time=   0.0s
[CV] END .......max_depth=8, max_samples=1, n_estimators=100; total time=   0.1s
[CV] END .......max_depth=8, max_samples=1, n_estimators=200; total time=   0.2s
[CV] END .......max_depth=8, max_samples=1, n_estimators=200; total time=   0.2s
[CV] END .......max_depth=8, max_samples=1, n_estimators=200; total time=   0.3s
[CV] END ......max_depth=8, max_samples=0.8, n_estimators=50; total time=   0.7s
[CV] END ......max_depth=8, max_samples=0.8, n_estimators=50; total time=   0.2s
[CV] END ......max_depth=8, max_samples=0.8, n_estimators=50; total time=   0.1s
[CV] END .....max_depth=8, max_samples=0.8, n_estimators=100; total time=   0.4s
[CV] END .....max_depth=8, max_samples=0.8, n_estimators=100; total time=   0.4s
[CV] END .....max_depth=8, max_samples=0.8, n_estimators=100; total time=   0.4s
[CV] END .....max_depth=8, max_samples=0.8, n_estimators=200; total time=   1.1s
[CV] END .....max_depth=8, max_samples=0.8, n_estimators=200; total time=   1.0s
[CV] END .....max_depth=8, max_samples=0.8, n_estimators=200; total time=   1.6s
[CV] END ......max_depth=8, max_samples=0.6, n_estimators=50; total time=   0.1s
[CV] END ......max_depth=8, max_samples=0.6, n_estimators=50; total time=   0.1s
[CV] END ......max_depth=8, max_samples=0.6, n_estimators=50; total time=   0.1s
[CV] END .....max_depth=8, max_samples=0.6, n_estimators=100; total time=   0.4s
[CV] END .....max_depth=8, max_samples=0.6, n_estimators=100; total time=   0.4s
[CV] END .....max_depth=8, max_samples=0.6, n_estimators=100; total time=   0.5s
...
[CV] END ....max_depth=20, max_samples=0.6, n_estimators=100; total time=   0.7s
[CV] END ....max_depth=20, max_samples=0.6, n_estimators=200; total time=   1.5s
[CV] END ....max_depth=20, max_samples=0.6, n_estimators=200; total time=   1.7s
[CV] END ....max_depth=20, max_samples=0.6, n_estimators=200; total time=   1.5s
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

{'max_depth': 18, 'max_samples': 0.6, 'n_estimators': 100}
```

After choosing the best hyperparameters, we can model the training dataset by calling `RandomForestClassifier`:

```
# MODEL
Rforest = RandomForestClassifier(max_depth=18,max_samples=0.6,
n_estimators=100)
Rforest = Rforest.fit(X, y)

# CROSS VALIDATION
cv = StratifiedKFold(n_splits=3)
val_Rforest = cross_val_score(Rforest, X, y, cv=cv).mean()
val_Rforest # validation score

------------------------------
0.8605916951979634
```

Unsurprisingly, **Random Forest** outweighs **Decision Tree** since it's actually a set of trees and choose the best result from the "forest". We can now do the predictions:

```
# PREDICTIONS
Rforest_predictions = Rforest.predict(T)
with open('RandomForest.txt', 'w') as f:
    for prediction in Rforest_predictions:
        f.write(f"{prediction}\n")

print("Predictions saved to RandomForest.txt")
```

## XGBoost(Extreme Gradient Boosting)

Similarly, we need to do hyperparameters optimization before modeling the training dataset. Since the number of hyperparameters is quite large, we will test two rounds of tuning:

```
# HYPERPARAMETER OPTIMIZATION

# ROUND 1

param_grid = {
    'max_depth': [3, 5, 7],
    'learning_rate': [0.3, 0.1, 0.05],
    'gamma': [0, 1, 10],
    'reg_lambda': [0, 1, 10]
}


cv = StratifiedKFold(n_splits=3)

optimal_params = GridSearchCV(
    estimator=xgb.XGBClassifier(objective='binary:logistic', #for binary classification
                                eval_metric="logloss",
                                use_label_encoder=False), #avoid warning (since we have done encoding)
    param_grid=param_grid,
    scoring='accuracy',
    verbose=2,
    cv = cv
)
optimal_params.fit(X,y)
optimal_params.best_params_
```

```
26
27  #{'gamma': 0, 'learning_rate': 0.3, 'max_depth': 5, 'reg_lambda':
    1}
28
29  # ROUND 2
30
31
32  param_grid = {
33      'max_depth': [4, 5, 6],
34      'learning_rate': [0.3, 0.5],
35      'subsample': [1, 0.8, 0.6, 0.4],
36      'gamma' : [10, 50, 100]
37  }
38
39
40
41  cv = StratifiedKFold(n_splits=3)
42
43  optimal_params = GridSearchCV(
44      estimator=xgb.XGBClassifier(objective='binary:logistic', #for
    binary classification
45                                  eval_metric="logloss",
46                                  learning_rate= 0.1,
47                                  reg_lambda=1,
48                                  use_label_encoder=False), #avoid
    warning (since we have done encoding)
49      param_grid=param_grid,
50      scoring='accuracy',
51      verbose=2,
52      cv = cv
53  )
54  optimal_params.fit(X,y)
55  optimal_params.best_params_
56  #{'gamma': 10, 'learning_rate': 0.3, 'max_depth': 6, 'subsample':
    0.8}
```

Then choosing the best hyperparameter, we continue modeling by calling `XGBClassifier`

```
1  # MODEL
2  xgbm = xgb.XGBClassifier(eval_metric="logloss",
3                           learning_rate= 0.3,
4                           reg_lambda=10,
5                           use_label_encoder=False, # as we have
    done encoding
6                           max_depth=8,
```

```
                          subsample=1)

xgbm = xgbm.fit(X, y)

# CROSS VALIDATION
cv = StratifiedKFold(n_splits=3)
val_xgbm = cross_val_score(xgbm, X, y, cv=cv).mean()
val_xgbm


------------------------
0.8704240189623387
```

## Artificial Neural Network

I will leave the prediction and data scaling part, only explaining the modeling part. The library I use to train neural networks is `tensorflow.Keras`, which is an advanced API for training neural networks conveniently.

First of all, we need to define a neural network framework based on `Keras`. Since we are dealing with binary classfication problem, the activation function for output layer should be `sigmoid`:

```
def ANN_1(neurons=10, hidden_layers=0, dropout_rate=0, learn_rate= 0.1):
    # model
    model = keras.Sequential()
    model.add(keras.layers.Dense(neurons, input_shape = (X_train.shape[1], ), activation='relu'))
    for i in range(hidden_layers):
        # Add one hidden layer
        model.add(keras.layers.Dense(neurons, activation='relu'))
        model.add(keras.layers.Dropout(dropout_rate))
    model.add(keras.layers.Dense(1, activation='sigmoid')) #Output layers
    # Compile model
    optimizer = keras.optimizers.SGD(learning_rate=learn_rate, momentum = 0.01)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
```

Then we use `KerasClassifier` to wrap the neural network into cross validation using `GridSearchCV`:

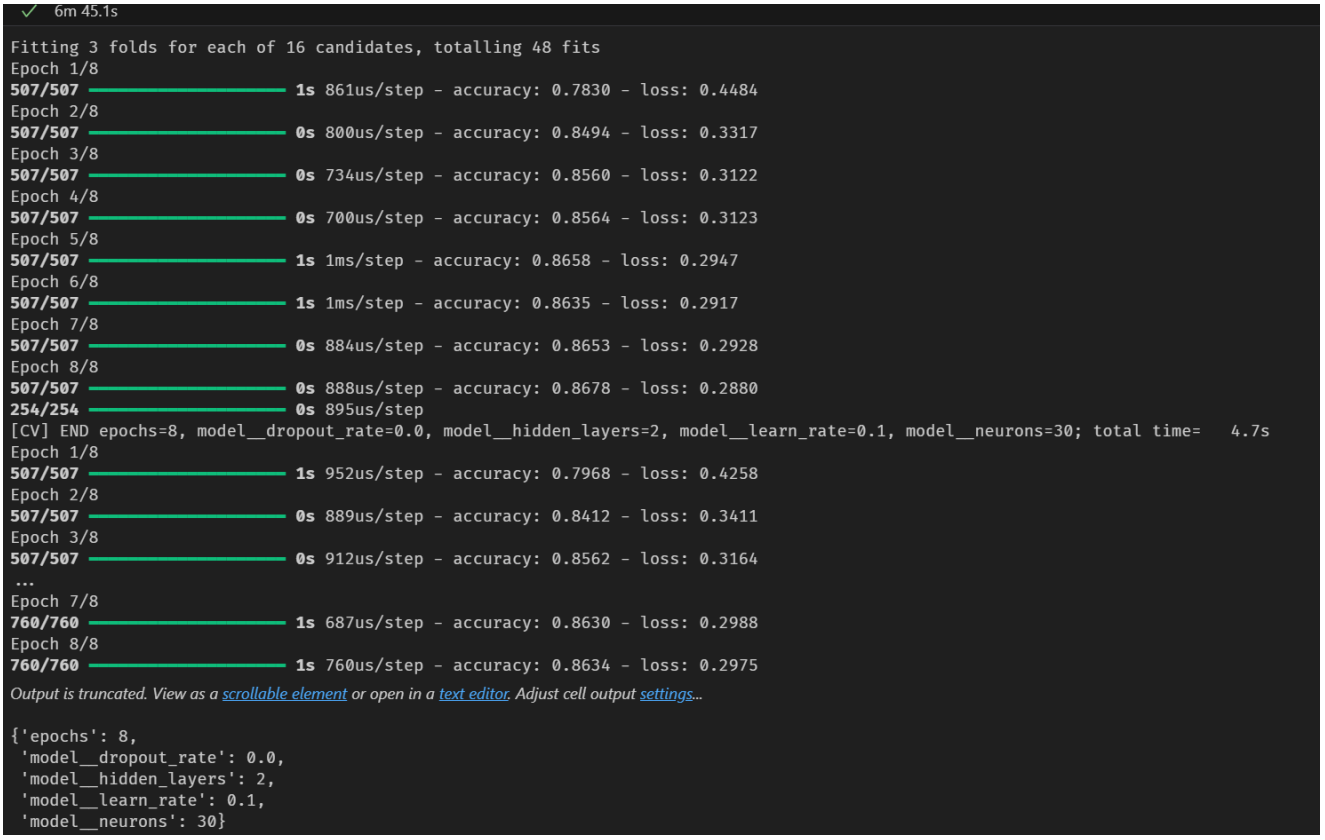
```
# we will do the grid search with KerasClassifier
ann = KerasClassifier(build_fn=ANN_1, batch_size=30)


param_grid = {
    'model__neurons': [30, 60],
    'model__hidden_layers': [2],
    'model__dropout_rate': [0.0, 0.1],
    'model__learn_rate': [0.1, 0.03],
    'epochs': [8, 15]
}

cv = StratifiedKFold(n_splits=3)

optimal_params = GridSearchCV(estimator=ann,
param_grid=param_grid, verbose=2, cv=cv)
optimal_params.fit(X_train,y)
optimal_params.best_params_
```

It will take a long time to train the Neural Network and do cross validation:



Then we will do another round of optimization, focusing on the `init_mode` and `activation_function` of the neural network, while choosing the best hyperparameter above in `ANN_1`:

```
# ROUND 2

def ANN_2(init_mode='uniform', activation='relu'):
    # model
    model = keras.Sequential()
    model.add(keras.layers.Dense(30,kernel_initializer=init_mode,
                                 input_shape = (X_train.shape[1],
), activation=activation))
    model.add(keras.layers.Dense(30,
kernel_initializer=init_mode,activation=activation))
    model.add(keras.layers.Dropout(0.0))
    model.add(keras.layers.Dense(1,kernel_initializer=init_mode,
activation='sigmoid'))
    # Compile model
    optimizer = keras.optimizers.SGD(learning_rate=0.1, momentum
= 0.01)
    model.compile(loss='binary_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
    return model


ann = KerasClassifier(build_fn=ANN_2, epochs= 8, batch_size=30)


param_grid = {
    'model__init_mode': ['uniform', 'lecun_uniform', 'normal',
'zero', 'glorot_normal',
                  'glorot_uniform', 'he_normal', 'he_uniform'],
    'model__activation': ['softmax','relu', 'tanh', 'sigmoid']
}
cv = StratifiedKFold(n_splits=3)


optimal_params = GridSearchCV(estimator=ann,
param_grid=param_grid, verbose=2, cv=cv)
optimal_params.fit(X_train,y)
optimal_params.best_params_
```

Then the result is :

```
✓ 8m 56.6s

Fitting 3 folds for each of 32 candidates, totalling 96 fits
Epoch 1/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.7543 - loss: 0.5739
Epoch 2/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 0s 856us/step - accuracy: 0.7555 - loss: 0.5563
Epoch 3/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 0s 885us/step - accuracy: 0.7579 - loss: 0.5536
Epoch 4/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 0s 882us/step - accuracy: 0.7644 - loss: 0.5463
Epoch 5/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 1s 963us/step - accuracy: 0.7592 - loss: 0.5524
Epoch 6/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 1s 967us/step - accuracy: 0.7579 - loss: 0.5535
Epoch 7/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 0s 900us/step - accuracy: 0.7566 - loss: 0.5552
Epoch 8/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 0s 927us/step - accuracy: 0.7599 - loss: 0.5513
254/254 ━━━━━━━━━━━━━━━━━━━━ 0s 835us/step
[CV] END model__activation=softmax, model__init_mode=uniform; total time=   4.7s
Epoch 1/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 1s 894us/step - accuracy: 0.7615 - loss: 0.5727
Epoch 2/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 0s 907us/step - accuracy: 0.7630 - loss: 0.5479
Epoch 3/8
507/507 ━━━━━━━━━━━━━━━━━━━━ 0s 927us/step - accuracy: 0.7565 - loss: 0.5553
 ...
Epoch 7/8
760/760 ━━━━━━━━━━━━━━━━━━━━ 1s 824us/step - accuracy: 0.8603 - loss: 0.2962
Epoch 8/8
760/760 ━━━━━━━━━━━━━━━━━━━━ 1s 739us/step - accuracy: 0.8617 - loss: 0.3036
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

{'model__activation': 'relu', 'model__init_mode': 'normal'}
```

Then we will continue tuning the `ANN_2` by stopping the learning early when it has 3 consecutive epoch without improvement:

```python
def ANN_():
    model = keras.Sequential()
    model.add(keras.layers.Dense(30,kernel_initializer='normal',
                                 input_shape = (X_train.shape[1],
), activation='relu'))
    model.add(keras.layers.Dense(30,
kernel_initializer='normal',activation='relu'))
    model.add(keras.layers.Dropout(0.0))
    model.add(keras.layers.Dense(1,kernel_initializer='uniform',
activation='sigmoid'))
    # Compile model
    optimizer = keras.optimizers.SGD(learning_rate=0.1, momentum
= 0.01)
    model.compile(loss='binary_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
    return model
# we define a learning rate schedule in order to decrease the
learning rate
# as we epoch increases.
def scheduler(epoch, lr):
    if epoch < 5:
        return lr
```

```
    elif epoch < 8:
        return 0.05
    else:
        return 0.01

# Early stopping: stop the learning when it has 3 consecutive epoch without improvement
callback2 = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
# Learning rate schedule
callback = tf.keras.callbacks.LearningRateScheduler(scheduler)
ann = KerasClassifier(build_fn=ANN_, epochs= 8,  batch_size=30, verbose=0)
# CROSS VALIDATION
cv = StratifiedKFold(n_splits=3)
val_ann= cross_val_score(ann, X_train, y,
                         cv=cv, fit_params={'callbacks': [callback,callback2]}).mean()
val_ann # validation score

--------------------
0.8511983144587832
```

Indeed, after three rounds of training. The final `ANN_` is way better than the original `ANN_1`, we are getting some improvements. So before we make any predictions, cross validation will be used to check the effect of ensembling.Then we will do ensembling on `ANN_` and we will make 10 Neural Networks and then join its predictions by averaging.

`VotingClassifier` is used to ensemble the 10 `ann` models.

```
# Cross Validation
n_members = 10
ann = KerasClassifier(build_fn=ANN_, epochs=8,  batch_size=30, verbose=0,callbacks=[callback,callback2])
models = [('ann' + str(i), ann) for i in range(n_members)]
ensemble = VotingClassifier(estimators=models, voting='soft')
ensemble = ensemble.fit(X_train,y)
scores = cross_val_score(ensemble, X_train, y, cv=3, scoring='accuracy')
print("Cross Validation Scores:", scores)
print("Mean Accuracy:", scores.mean())
```

Then just use `ensemble` to predict on the testing dataset:
`ann_ensemble_predictions = ensemble.predict(X_test)`.

The predictions result will be stored into `ANN.txt`.

### Ensembling

We ensemble the cross validation score of the several models:

```
# Select the models by looking at the validation score
np.array([val_logit, val_cnb, val_knn, val_suppvm, val_tree,
val_Rforest, val_xgbm, scores.mean()])

-------------------------
array([0.85075937, 0.75010974, 0.83623036, 0.84922307, 0.85462207,
       0.8605917 , 0.87042402, 0.85198841])
```

Then we would use `xgboost` model at last.

## Conclusion

Comparing the cross validation scores of the models, we get:

| | Accuracy |
|---|---|
| Logistic Regression | 0.850759 |
| Categorical Naive Bayes | 0.750110 |
| Gaussian Naive Bayes | 0.794970 |
| K-Nearest Neighbors | 0.836230 |
| Support Vector Machines | 0.849223 |
| Decision Trees | 0.854622 |
| Random Forest | 0.860592 |
| XGBoost | 0.870424 |
| Artificial Neural Networks | 0.851988 |

**Categorical Naive Bayes** and **Gaussian Naive Bayes** have the lowest score unsurprisingly since only few columns are used. **K-Nearest Neighbors** is rather simple, leading to a low accuracy too. The accuracy of **Decision Trees**, **Random Forest** and **XGBoost** is improving with the complexity of the model and they are actually applying bagging method from decision trees to XGBoost. What is

unexpected is the accuracy of **ANN** is lower than Logistic Regression, maybe it's because the classification skill on this dataset of Logistic Regression model outweighs the latter, even with three rounds of exhausting hyperparameters tuning.

The model I choose at last is the **XGBoost** model. The limitation of it is prone to overfitting, especially if the number of trees is too large or the model is too complex. Proper tuning of hyperparameters like `max_depth`, `learning_rate`, and `n_estimators` is crucial to mitigate this. I am really curious about how to train and tune **ANN** better, and I will look into it when I have more time. More time should be spent on tuning the hyperparameters.

# Reference


1. https://scikit-learn.org/stable/auto_examples/tree/index.html
2. https://www.kaggle.com/datasets/uciml/adult-census-income/data