Income Classification Using the Adult Dataset - Comparing the K-Nearest Neighbors and Random Forest Algorithms (Python)

Jose Leonardo Wong

Contents

1. Introduction

Classifying clients into groups based on income, demographics or social characteristics to make decisions is a common task in companies. Supervised learning, a subset of Machine Learning that constructs models to predict a target variable using features in the data, offers algorithms to solve classification problems that can help automate and expedite these tasks (Larose & Larose, 2015).

The following exercise uses K-Nearest Neighbours (KNN) and Random Forest (RF) supervised learning algorithms to classify individuals using data from a census population into income groups based on attributes like age, education and job, to assess how well these algorithms perform under real-world classification problems. The features in the dataset are the independent variables, and the target variable is income (more or less than $50,000).

2. Dataset

For this exercise, we use the Adult dataset, which contains 32,561 samples of information about individuals in 14 demographic and social features, including age, education, marital status, occupation, how many hours per week they work and income, the target variable indicating whether they earn over $50,000 or not. The dataset contains information curated from the 1994 US Census Bureau database and is available in the UCI Machine Learning repository under a Creative Commons Attribution 4.0 International (CC BY 4.0) license (Becker & Kohavi, 1996).

Some features in the dataset are sensitive, such as age, race, sex and native country; they should be used carefully only if there are justified reasons to avoid bias and discrimination. Additionally, despite the information being anonymised, it should be used with caution to avoid re-identification through the features and compliance with regulations such as GDPR.

3. Classification with Python

3.1 Exploratory Data Analysis and Data Processing

Upon uploading the file, we started the exploratory data analysis by confirming its shape, 32,561 samples and 15 features, and displaying its first five and last samples. This information showed that there is variety of variable types in the dataset and possibly unknown information indicated by the “?” character (Figure 1).

Figure 1

Overview of the Dataset's First and Last Rows

```
# Load and read the data

data = pd.read_csv('/content/adult.csv')
```

```
# Show the shape of the dataset

data.shape
```

```
(32561, 15)
```

```
# Show five first samples

data.head()
```

workclass	fnlwgt	education	education.num	marital.status	occupation	relationship	race	sex	capital.gain	capital.loss	hours.per.week	native.country	income
?	77053	HS-grad	9	Widowed	?	Not-in-family	White	Female	0	4356	40	United-States	<=50K
Private	132870	HS-grad	9	Widowed	Exec-managerial	Not-in-family	White	Female	0	4356	18	United-States	<=50K
?	186061	Some-college	10	Widowed	?	Unmarried	Black	Female	0	4356	40	United-States	<=50K
Private	140359	7th-8th	4	Divorced	Machine-op-inspct	Unmarried	White	Female	0	3900	40	United-States	<=50K
Private	264663	Some-college	10	Separated	Prof-specialty	Own-child	White	Female	0	3900	40	United-States	<=50K

```
# Show five last samples

data.tail()
```

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationship	race	sex	capital.gain	capital.loss	hours.per.week	native.cou
32556	22	Private	310152	Some-college	10	Never-married	Protective-serv	Not-in-family	White	Male	0	0	40	United-St
32557	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-St
32558	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-St
32559	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-St
32560	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-St

We continued by displaying the features and their datatypes (Figure 2), observing that there were age, education.num, capital.gain, capital.loss and hours.per.week continuous features which are inherent to the sampled individual, and a fnlwgt continuous feature which is only relevant for the census methodology. Additionally, we found workclass, education, marital.status, occupation, relationship, race, sex and native.country categorical features in addition to the binary categorical target feature income.

Figure 2

Features and Their Data Types in the Dataset

```
# Examine Columns data types

data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32561 entries, 0 to 32560
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             32561 non-null  int64
 1   workclass       32561 non-null  object
 2   fnlwgt          32561 non-null  int64
 3   education       32561 non-null  object
 4   education.num   32561 non-null  int64
 5   marital.status  32561 non-null  object
 6   occupation      32561 non-null  object
 7   relationship    32561 non-null  object
 8   race            32561 non-null  object
 9   sex             32561 non-null  object
 10  capital.gain    32561 non-null  int64
 11  capital.loss    32561 non-null  int64
 12  hours.per.week  32561 non-null  int64
 13  native.country  32561 non-null  object
 14  income          32561 non-null  object
dtypes: int64(6), object(9)
memory usage: 3.7+ MB
```

We continued by displaying the dataset's numerical features basic statistics (Figure 3), finding that the average age is 38.58 years, individuals work an average of 40 hours, education.num average is approximately 10, aligning with high school-level education, and at least 75% of the samples did not show either capital.gain or capital.loss. Moreover, the minimum and maximum values in each category showed a range within expected values.

Figure 3

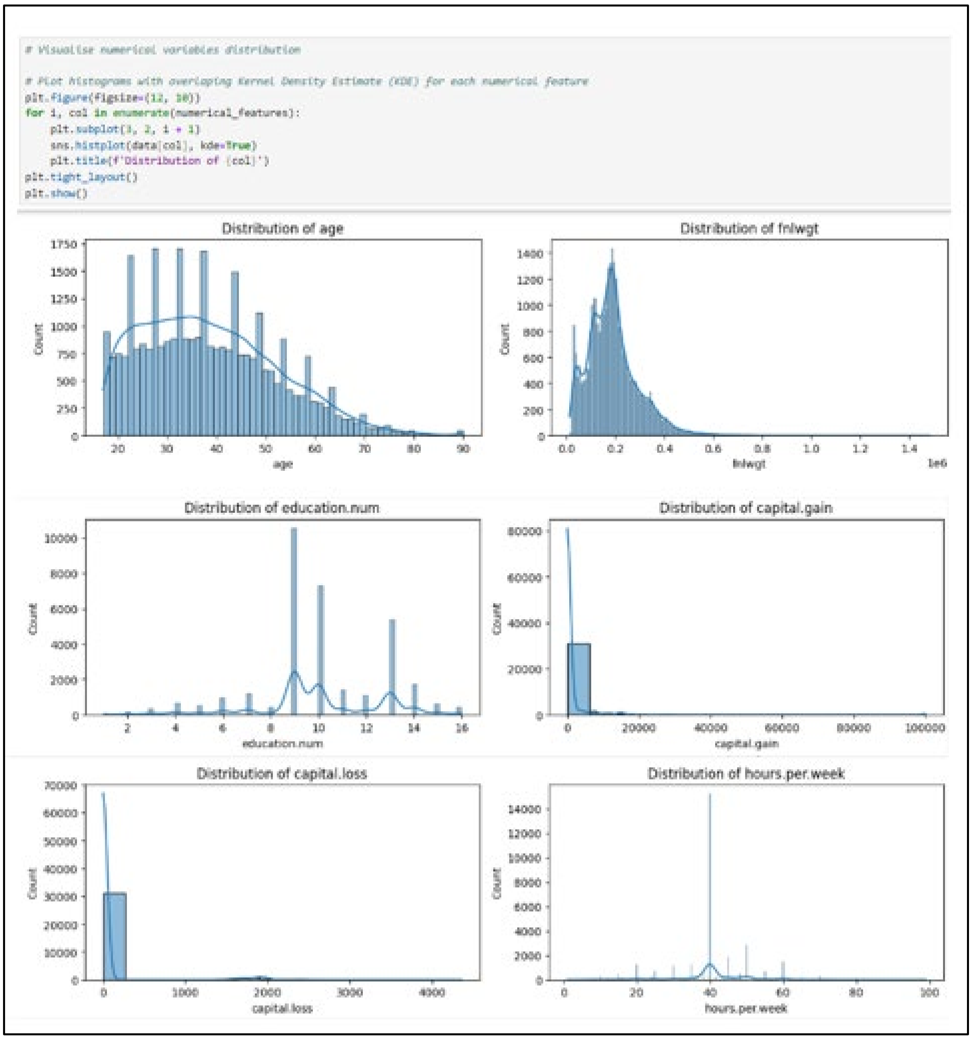
Summary Statistics of Numerical Features

```
# Show basic statistics for numerical features

data.describe()
```

	age	fnlwgt	education.num	capital.gain	capital.loss	hours.per.week
count	32561.000000	3.256100e+04	32561.000000	32561.000000	32561.000000	32561.000000
mean	38.581647	1.897784e+05	10.080679	1077.648844	87.303830	40.437456
std	13.640433	1.055500e+05	2.572720	7385.292085	402.960219	12.347429
min	17.000000	1.228500e+04	1.000000	0.000000	0.000000	1.000000
25%	28.000000	1.178270e+05	9.000000	0.000000	0.000000	40.000000
50%	37.000000	1.783560e+05	10.000000	0.000000	0.000000	40.000000
75%	48.000000	2.370510e+05	12.000000	0.000000	0.000000	45.000000
max	90.000000	1.484705e+06	16.000000	99999.000000	4356.000000	99.000000

Then we moved onto plotting the distribution of numerical variables, finding that age is right-skewed with most individuals between 20 and 50 years, capital.gain and capital.loss are predominantly zero, hours.per.week as expected shows a peak at 40 hours and education.num aligns with distinct education levels (Figure 4).

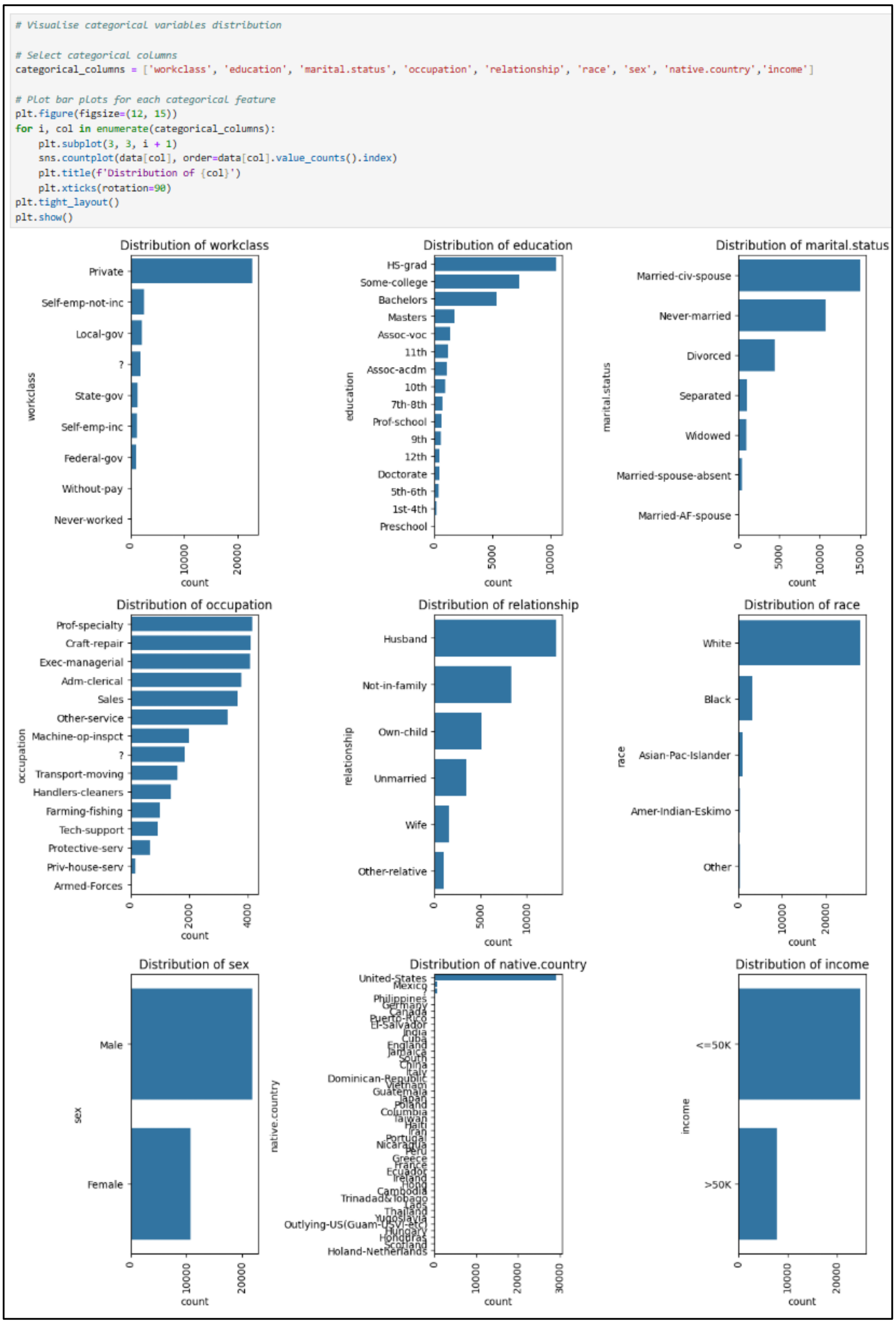
Figure 4
Distribution of Numerical Variables



We then plotted the categorical feature distributions (Figure 5), finding that most individuals work in the private sector and are mostly high school graduates, followed by significant quantities of bachelors and individuals with some college education. Regarding marital status, most individuals were married in a civil union, followed by many never-married individuals while the dominant relationship status where Husband and not-in-family. The occupation variable showed a large variety of professions and some unknown (?). The race of the individuals was largely white, and their native country was the US. There were twice as many males as females. Lastly, the target variable showed a large imbalance, with most individuals earning under $50,000.

Figure 5

Distribution of Categorical Variables



Our next step was identifying rare categories in the categorical features to consider potential consolidation or exclusion during preprocessing. Several rare categories with a representation of less than 1% were found (Figure 6).

Figure 6

Rare Categories in Categorical Features

```
# Identify rare categories (appearing in less than 1% of the total entries in the dataset)

categorical_columns = data.select_dtypes(include=['object']).columns
for column in categorical_columns:
    value_counts = data[column].value_counts()
    rare_categories = value_counts[value_counts < len(data) * 0.01].index
    if len(rare_categories) > 0:
        print(f"\nRare categories in {column}:")
        print(rare_categories)
```

```
Rare categories in workclass:
Index(['Without-pay', 'Never-worked'], dtype='object', name='workclass')

Rare categories in education:
Index(['1st-4th', 'Preschool'], dtype='object', name='education')

Rare categories in marital.status:
Index(['Married-AF-spouse'], dtype='object', name='marital.status')

Rare categories in occupation:
Index(['Priv-house-serv', 'Armed-Forces'], dtype='object', name='occupation')

Rare categories in race:
Index(['Amer-Indian-Eskimo', 'Other'], dtype='object', name='race')

Rare categories in native.country:
Index(['Philippines', 'Germany', 'Canada', 'Puerto-Rico', 'El-Salvador',
       'India', 'Cuba', 'England', 'Jamaica', 'South', 'China', 'Italy',
       'Dominican-Republic', 'Vietnam', 'Guatemala', 'Japan', 'Poland',
       'Columbia', 'Taiwan', 'Haiti', 'Iran', 'Portugal', 'Nicaragua', 'Peru',
       'Greece', 'France', 'Ecuador', 'Ireland', 'Hong', 'Cambodia',
       'Trinadad&Tobago', 'Laos', 'Thailand', 'Yugoslavia',
       'Outlying-US(Guam-USVI-etc)', 'Hungary', 'Honduras', 'Scotland',
       'Holand-Netherlands'],
      dtype='object', name='native.country')
```

Then we checked for data quality issues, finding no missing values, 24 duplicate samples and no unexpected negative values (Figure 7).

Figure 7

Identifying Data Quality Issues in the Dataset

```
# Check for missing values

missing_values = data.isnull().sum()
print("Missing values in each column:\n", missing_values)
```

```
Missing values in each column:
 age              0
workclass         0
fnlwgt            0
education         0
education.num     0
marital.status    0
occupation        0
relationship      0
race              0
sex               0
capital.gain      0
capital.loss      0
hours.per.week    0
native.country    0
income            0
dtype: int64
```

```
# Check for duplicate samples

# Check for duplicate rows
duplicates = data.duplicated()

# Count the number of duplicates
num_duplicates = duplicates.sum()
print(f"Number of duplicate rows: {num_duplicates}")

# Display the duplicate rows
if num_duplicates > 0:
    duplicate_rows = data[duplicates]
    print("Duplicate rows:")
    display(duplicate_rows)
else:
    print("No duplicate rows found.")
```

```
Number of duplicate rows: 24
```

```
# Check for negative values

def check_non_negative_values(df, columns):
    issues = {}

    for column in columns:
        # Find rows where values are negative
        negative_values = df[df[column] < 0]
        if not negative_values.empty:
            issues[column] = negative_values[column]

    return issues

# Define columns to check for non-negative values
non_negative_columns = ['age', 'education.num', 'capital.gain', 'capital.loss', 'hours.per.week']


negative_value_issues = check_non_negative_values(data, non_negative_columns)

# Display results
if negative_value_issues:
    for col, invalid_values in negative_value_issues.items():
        print(f"\nColumn '{col}' contains negative values:")
        print(invalid_values)
else:
    print("All specified columns contain only non-negative values.")
```

```
All specified columns contain only non-negative values.
```

Similarly, we used a regular expression to look for unusual characters (Figure 8), finding several instances of question marks, which we had observed before and referred to unknown occupation and work class as well as ampersand and parentheses characters which we explored further finding that they belong to country and territory names such as “Outlying-US(Guam-USVI-etc)” and “Trinadad&Tobago”.

Figure 8

Detection of Unusual Characters in Feature Values

```python
# Check for unusual characters

def detect_unusual_chars_and_whitespace(df, exclusions=None, general_acceptable_pattern=None):
    # Define a regex pattern to detect any character that is not alphanumeric, whitespace, or in exclusions
    unusual_pattern = re.compile(r'[^\w\s<>=-]')
    # Dictionary to store columns with unusual characters and whitespace issues
    issues = {'unusual_chars': {}, 'whitespace_issues': {}}

    # Loop through each object column to check for both unusual characters and whitespace issues
    for column in df.select_dtypes(include=['object']).columns:
        # Apply specified exclusions for income column
        if exclusions and column in exclusions:
            column_pattern = exclusions[column] + (general_acceptable_pattern or "")
            unusual_chars = df[column].apply(lambda x: unusual_pattern.findall(re.sub(column_pattern, "", str(x))))
        else:
            # Apply the general pattern for other columns
            unusual_chars = df[column].apply(lambda x: unusual_pattern.findall(re.sub(general_acceptable_pattern or "", "", str(x))))

        # Count occurrences of each unusual character in the column
        unusual_counts = Counter([char for sublist in unusual_chars if sublist for char in sublist])
        if unusual_counts:
            issues['unusual_chars'][column] = unusual_counts

        # Count rows with leading or trailing whitespace
        whitespace_issues_count = df[column].apply(lambda x: x != x.strip()).sum()
        if whitespace_issues_count > 0:
            issues['whitespace_issues'][column] = whitespace_issues_count

    return issues

# Define exclusions of known characters for income column (Ignore "<=", ">", "=" for the 'income' column)
exclusions = {
    'income': r'[<=|>]',
}

# General pattern to allow hyphens in any column
general_acceptable_pattern = r'-'


issues_detected = detect_unusual_chars_and_whitespace(data, exclusions=exclusions, general_acceptable_pattern=general_acceptable_pattern)

# Display summary results
for col, counts in issues_detected['unusual_chars'].items():
    print(f"\nColumn '{col}':")
    for char, count in counts.items():
        print(f"  {count} instances of the character '{char}' found")

for col, count in issues_detected['whitespace_issues'].items():
    print(f"\nColumn '{col}' has {count} entries with leading or trailing whitespace.")
```

```
Column 'workclass':
  1836 instances of the character '?' found

Column 'occupation':
  1843 instances of the character '?' found

Column 'native.country':
  583 instances of the character '?' found
  19 instances of the character '&' found
  14 instances of the character '(' found
  14 instances of the character ')' found
```

```python
# Filter rows with '&', '(', or ')' characters in 'native.country' to investigate

special_char_rows = data[data['native.country'].str.contains('[&()]', na=False)]
print("Rows with '&', '(', or ')' in 'native.country':")
display(special_char_rows)
```

```
Rows with '&', '(', or ')' in 'native.country':
```

kclass	fnlwgt	education	education.num	marital.status	occupation	relationship	race	sex	capital.gain	capital.loss	hours.per.week	native.country	income
Private	191765	HS-grad	9	Never-married	Adm-clerical	Other-relative	Black	Female	0	2339	40	Trinadad&Tobago	<=50K
Private	169329	HS-grad	9	Married-civ-spouse	Tech-support	Husband	Black	Male	0	1887	40	Trinadad&Tobago	>50K
Private	111797	Some-college	10	Never-married	Other-service	Not-in-family	Black	Female	0	0	35	Outlying-US(Guam-USVI-etc)	<=50K

We then carried out a check to ensure the feature variable would contain only the two expected values, finding that it was the case (Figure 9).

Figure 9

Validation of Target Feature Values

```
# check that the target column contains only valid specified values

def check_target_values(df, column, valid_values):
    # Identify any values in the target column that are not in the valid values list
    invalid_values = df[~df[column].isin(valid_values)][column].unique()

    if len(invalid_values) > 0:
        print(f"Column '{column}' contains unexpected values:")
        print(invalid_values)
    else:
        print(f"Column '{column}' contains only valid target values: {valid_values}")

# Define the valid values for the target column
valid_income_values = ['<=50K', '>50K']

check_target_values(data, 'income', valid_income_values)

Column 'income' contains only valid target values: ['<=50K', '>50K']
```

In the next steps, we looked for the symmetry and variance of the numerical variables, as the presence of skewness and outliers can affect the performance of the algorithms, especially for distance-based algorithms such as the KNN. We examined the skewness of numerical features using the Fisher-Pearson standardised moment coefficient through the pandas .skew() function (Figure 10), which measures the asymmetry of a distribution relative to its mean (Doane & Seward, 2011). A value of zero indicates a symmetric distribution, positive values indicate right-skewed distributions, and negative values indicate left-skewed distributions. We found that capital.gain and capital.loss showed high positive skewness and most values concentrated near zero, while other features had close to symmetric distributions in line with the previous findings throw plotting the distributions.

Figure 10

Skewness Analysis of Numerical Variables

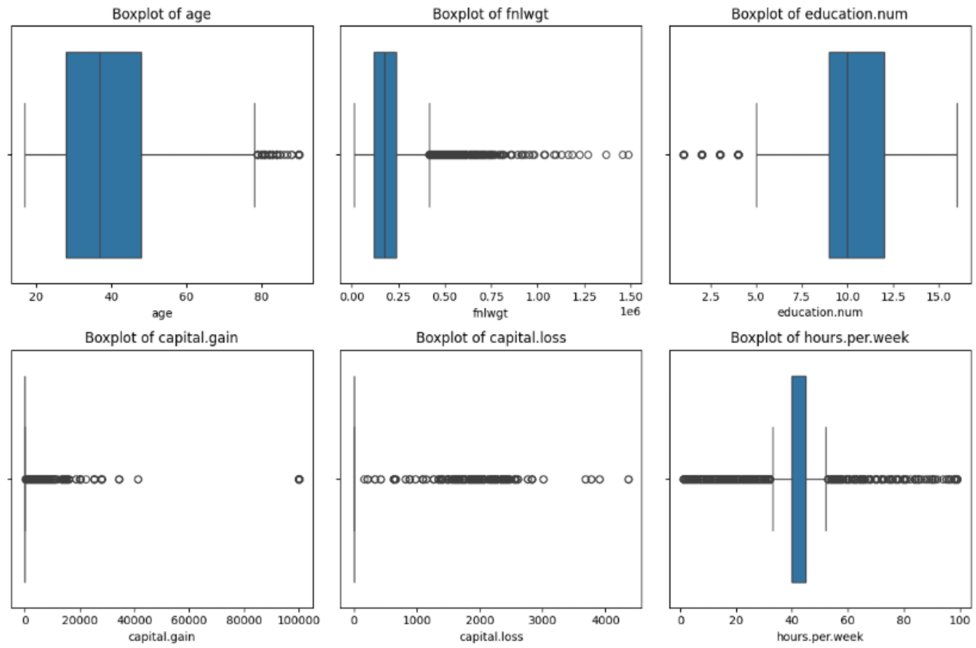
```
# Calculate the skewness of numerical features
skewness = numerical_features.skew()

# Print the skewness values
print(skewness)

age                0.558743
fnlwgt             1.446980
education.num     -0.311676
capital.gain      11.953848
capital.loss       4.594629
hours.per.week     0.227643
dtype: float64
```

Similarly, we used box plots to identify outliers using the Interquartile Range (IQR) method (Figure 11), which calculates the range between the first (Q1) and third (Q3) quartiles and defines outliers as values outside 1.5 times the IQR from Q1 and Q3. These revealed significant outliers in, capital.gain, capital.loss, and hours.per.week, which were identified for further handling in the preprocessing steps. Fewer outliers were present in hours.per.week and education.num, while the outliers on fnlwgt were less concerning, as this variable is a strong candidate for being dropped, as it is not an inherent feature of the individuals.

Figure 11

Boxplot of Outliers in Numerical Features

```
# Check for outliers  (outliers can significantly impact distance-based algorithms like KNN)


# Boxplot for continuous variables
plt.figure(figsize=(12, 8))
for i, column in enumerate(numerical_features, 1):
    plt.subplot(2, 3, i)
    sns.boxplot(x=data[column])
    plt.title(f'Boxplot of {column}')

plt.tight_layout()
plt.show()

# Calculate IQR for each continuous variable (Interquartile Range represents the range between the first quartile (Q1) and the third quartile (Q3) of th
IQRs = {}
for column in numerical_features:
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    IQRs[column] = IQR

# Display IQR table as a DataFrame
IQR_table = pd.DataFrame.from_dict(IQRs, orient='index', columns=['IQR'])
display(IQR_table)

# Detect and print outliers based on IQR
for column in numerical_features:
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]

    if len(outliers) > 0:
        print(f"\nOutliers in {column}:")
        print(outliers[column].describe())
```



	IQR
age	20.0
fnlwgt	119224.0
education.num	3.0
capital.gain	0.0
capital.loss	0.0
hours.per.week	5.0

Then we attempted to look for initial obvious correlations among numerical features, as correlated variances could indicate redundant variables which would not provide additional information to the model, creating opportunities to simplify the model by dropping redundant variables. For this purpose, we used pairplots (Figure 12), finding that there were no obvious strong correlations.

Figure 12

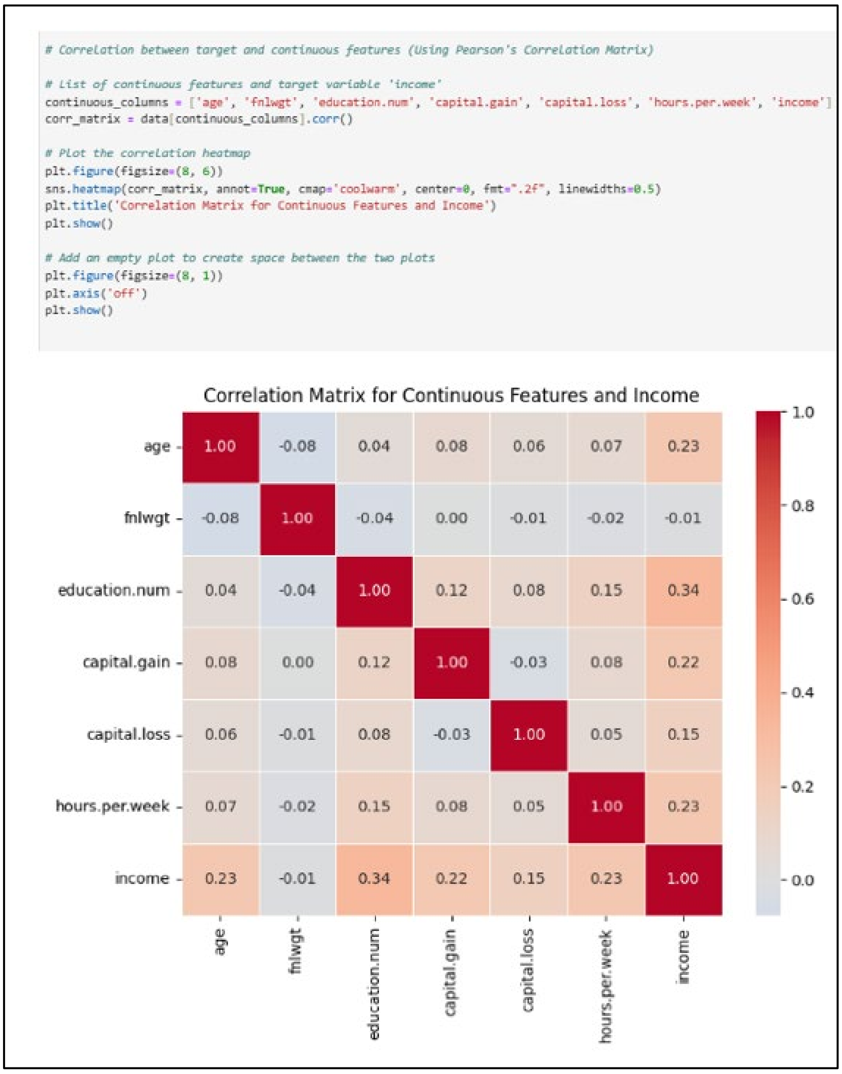
Pairplot Showing Correlations Among Numerical Features

```python
# Create a pairplot to visualize relationships between variables

sns.pairplot(data, hue='income', diag_kind='kde')
plt.show()
```

We also looked for correlations between variables and the target feature, as this can indicate the variable's predictive power with respect to the income target variable. For the continuous variables, we used Pearson's correlation matrix (Figure 13), which calculates the correlation by dividing the covariance of two variables by the product of their standard deviations, providing a value between -1, indicating perfect negative correlation and 1, indicating perfect positive correlation, while 0 indicates no correlation. We found that education.num has the strongest positive correlation with income (0.34), followed by age (0.23), which could also be seen in the previous step with the pairplots visualisation (Figure 12), where the binary values of the target variable were represented with distinct colours. Additionally, capital.gain and hours.per.week showed moderate positive correlations with income at 0.22 and 0.23. Other features had a weak or null correlation with income. The matrix confirmed the weak multicollinearity between numerical features themselves observed in the pairplots.

Figure 13

Pearson Correlation Matrix for Numerical Features

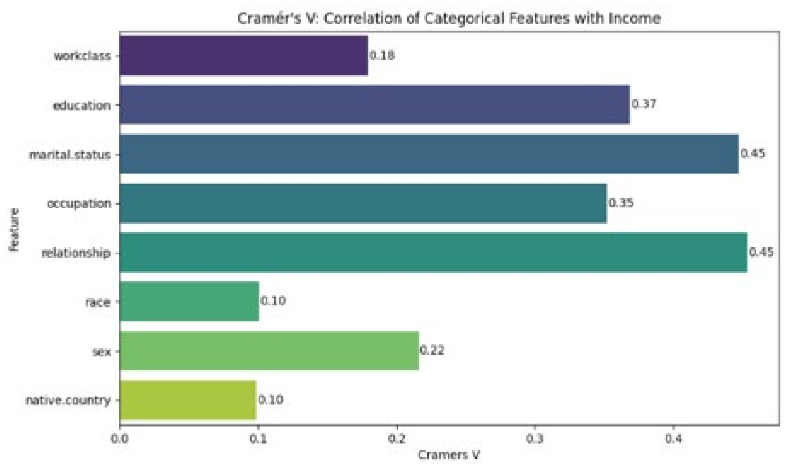


For the categorical features correlation with the target variable, we used the Cramer's V method (Figure 14), which measures the strength of correlation between two categorical variables from 0, no association, to 1, perfect association. These values are the result of normalising the Chi-squared statistic for each feature and target variable pair, which indicates the deviation from expected behaviour of uncorrelated variables (Everitt, 2002). The expected behaviour of uncorrelated variables is that the class proportions in the dependent variable, in our case income, with respect to the whole population, would resemble the proportion within each class in the independent variable that is related to each of the dependent variable classes. For instance, if 20% of the population earns above $50,000 no correlation with marital.status variable would mean that within each category in the marital.status variable also 20% of the samples earn above $50,000. We found that marital.status and relationship showed the highest correlation with income (both 0.45), followed by education (0.37) and occupation (0.35), while other features like native.country, race, and workclass showed weak correlations.

Figure 14

Cramer's V Correlation Analysis Between Categorical Features and Target Variable

```python
#Correlation between target and categorical features (Cramer's V)

# Define a function to calculate Cramér's V (measures the strength of association between two categorical variables)
def cramers_v(confusion_matrix):
    chi2 = chi2_contingency(confusion_matrix)[0] #Chi-square test tests whether two categorical variables are independent or not.
    n = confusion_matrix.sum()
    r, k = confusion_matrix.shape
    return np.sqrt(chi2 / (n * (min(r, k) - 1)))

# List of categorical features
categorical_columns = ['workclass', 'education', 'marital.status', 'occupation',
                       'relationship', 'race', 'sex', 'native.country']

# Calculate Cramér's V for each categorical feature
categorical_corr = {}
for column in categorical_columns:
    confusion_matrix = pd.crosstab(data[column], data['income'])
    cramers_v_value = cramers_v(confusion_matrix.to_numpy())
    categorical_corr[column] = cramers_v_value

# Convert to DataFrame for plotting, renaming column to avoid encoding issues
categorical_corr_df = pd.DataFrame(list(categorical_corr.items()), columns=['Feature', 'Cramers V'])

# Plot Barplot for Cramér's V
plt.figure(figsize=(10, 6))
cramer_plot = sns.barplot(x='Cramers V', y='Feature', data=categorical_corr_df, palette='viridis', hue='Feature', dodge=False)
plt.legend([],[], frameon=False)
plt.title("Cramér's V: Correlation of Categorical Features with Income")

# Annotate each bar with its Cramér's V value
for index, value in enumerate(categorical_corr_df["Cramers V"]):
    cramer_plot.text(value + 0.01, index, f"{value:.2f}", color='black', ha="center", va="center")

plt.show()

# Convert 'income' back to object type after correlation analysis to avoid issues in numerical analysis steps
data['income'] = data['income'].apply(lambda x: '>50K' if x == 1 else '<=50K')
```



After the Exploratory Data Analysis, we moved into the preprocessing stage. We started by removing duplicate rows to ensure data integrity and avoid redundancy (Figure 15).

Figure 15

Handling Duplicate Rows

```
# Remove duplicate rows
data = data.drop_duplicates()

# Verify that duplicates are removed
print("Number of duplicate rows after removal:", data.duplicated().sum())

Number of duplicate rows after removal: 0
```

To address the finding is the workclass variable, we replaced the “?” entries, which accounted for 5.6% of the data, with the label 'Unknown' to ensure completeness and binned the categories 'Without-pay' and 'Never-worked' into a single category named 'Other' due to their low count (Figure 16)

Figure 16

Handling Findings in the “workclass” Variable

```
# Handling findings in "workclass" variable

# 5.6% of the values are '?'. Replace '?' with 'unknown'.
data['workclass'] = data['workclass'].replace('?', 'Unknown')

# Grouping "Whitout-pay" and "Never-Worked" into a new category "Other" because these are low-count categories.
data['workclass'] = data['workclass'].replace(['Without-pay', 'Never-worked'], 'Other')

# Show the updated distribution
print(data['workclass'].value_counts())

workclass
Private             22673
Self-emp-not-inc     2540
Local-gov            2093
Unknown              1836
State-gov            1298
Self-emp-inc         1116
Federal-gov           960
Other                  21
Name: count, dtype: int64
```

Regarding the findings in the marital.status variable, we binned low count categories Married-spouse-absent and Married-AF-spouse with the large count category Married-civ-spouse to form a new category Married for simplicity without affecting overall interpretability (Figure 17).

Figure 17

Handling Findings in "marital.status" Variable

```
# Handling findings in marital status variable.

# 'Married-AF-spouse' (23) and 'Married-spouse-absent' (418) have very low sample counts compared to 'Married-civ-spouse' (14976),
# so it makes sense to combine them into one 'Married' category  to reduce the dataset complexity while preserving essential information.
married_categories = ['Married-civ-spouse', 'Married-AF-spouse', 'Married-spouse-absent']
data['marital.status'] = data['marital.status'].replace(married_categories, 'Married')

# Show adjusted distribution
print("\nFinal marital status categories:")
print(data['marital.status'].value_counts())
```

```
Final marital status categories:
marital.status
Married          15411
Never-married    10667
Divorced          4441
Separated         1025
Widowed            993
Name: count, dtype: int64
```

We also reduced the number of categories found in the occupation variable, binning similar roles into broader categories that are more interpretable and common (Figure 18)

Figure 18

Handling Findings in the "occupation" Variable

```
# Handling findings in the occupation variable

# Define mapping for broader occupation categories
occupation_mapping = {
    'Prof-specialty': 'Affluent Professionals',
    'Exec-managerial': 'Affluent Professionals',
    'Adm-clerical': 'White-Collar Workers',
    'Sales': 'White-Collar Workers',
    'Tech-support': 'White-Collar Workers',
    'Craft-repair': 'Manual Workers',
    'Machine-op-inspct': 'Manual Workers',
    'Transport-moving': 'Manual Workers',
    'Handlers-cleaners': 'Manual Workers',
    'Farming-fishing': 'Manual Workers',
    'Other-service': 'Others',
    'Protective-serv': 'Others',
    'Priv-house-serv': 'Others',
    'Armed-Forces': 'Others',
    '?': 'Others'
}

# Apply mapping  to the 'occupation' column to replace it
data['occupation'] = data['occupation'].map(occupation_mapping)

# Display the updated distribution for verification
print("\nUpdated Occupation Categories:")
print(data['occupation'].value_counts())
```

```
Updated Occupation Categories:
occupation
Manual Workers            10052
White-Collar Workers       8345
Affluent Professionals     8201
Others                     5939
Name: count, dtype: int64
```

Regarding the relationship feature, it contained a category called Unmarried, which created ambiguity because Unmarried is not a relationship role but rather a marital status, a distinction already captured in the marital.status variable. This overlap made the interpretation of Unmarried unclear and redundant within the context of the dataset, so we merged the Unmarried category with the Not-in-family category (Figure 19).

Figure 19

Handling Findings in the "relationship" Variable

```
# Handling findings in the relationship variable

# This attribute seems to capture the role of a person within a family. However, the "Unmarried" category refers to marital status
# (for which we have another attribute in the dataset). There is substantial ambiguity in the categories. For instance, a person could
# be unmarried and at the same time live with relatives.
# We could consider dropping the category or merging it with another category.

# Investigate the percenatge of samples with "unmarried" category
unmarried_percentage = (data['relationship'] == 'Unmarried').mean() * 100
print(f"Percentage of 'Unmarried': {unmarried_percentage:.2f}%")
```

```
Percentage of 'Unmarried': 10.59%
```

```
# The percentage is significant and dropping the samples could affect model's performance.
# Investigate the relationship of the unmarried varaible with other variables.
# Create a crosstab
relationship_marital_crosstab = pd.crosstab(data['relationship'], data['marital.status'])

# Display the crosstab
print(relationship_marital_crosstab)
```

```
marital.status  Divorced  Married  Never-married  Separated  Widowed
relationship
Husband                0    13187              0          0        0
Not-in-family       2403      228           4694        420      547
Other-relative       110      157            611         55       48
Own-child            328      141           4481         99       15
Unmarried           1600      130            881        451      383
Wife                   0     1568              0          0        0
```

```
# Based on the crosstab information, clearly "unmarried" does not mean someone who has never married as there are overlaps with "Divorced",
# "Separated" and "Widowed" in the marital status variable.
# 130 samples state "Unmarried" in the relationship varaible and "Married" in the marital status variable, which is
# odd and contributes to the ambiguity
# Given that the category is very ambiguous, we decide to merge it with the "not-in-family" as the "unmarried" category seems to
# also refer to individuals who do not live in a traditional family arrangements.
# We do this to reduce the ambiguity and simplify the data while keeping the samples represented.
# Merge 'Unmarried' with 'Not-in-family'
data['relationship'] = data['relationship'].replace({'Unmarried': 'Not-in-family'})

# Show the updated distribution
print("\nRelationship categories:")
print(data['relationship'].value_counts())
```

```
Relationship categories:
relationship
Husband           13187
Not-in-family     11737
Own-child          5064
Wife               1568
Other-relative      981
Name: count, dtype: int64
```

For the capital.gain and capital.loss variables which were highly skewed, we decided to transform them into a binary indicator, creating two new binary features has_capital_gain and has_capital_loss, where 1 indicates the presence of a gain or loss and 0 its absence. This achieved a simplification of the model while keeping relevant information for modelling (Figure 20).

Figure 20

Handling Findings in the "capital.loss" and "capital.gain" Variables

```
# Handling findings in capital.gain and capital.loss features

# Both 'capital.gain' and 'capital.loss' are highly skewed with most values at 0, which could reduce the predictive power of models if left as-is.
# Despite this skewness, these features show a moderate correlation with the target variable (income), suggesting they hold relevant information.
# To retain their information without introducing skew, we transform these columns into binary indicators:
# - 'has_capital_gain': 1 if a person has capital gain, 0 otherwise.
# - 'has_capital_loss': 1 if a person has capital loss, 0 otherwise.
# This approach preserves the key information for affordability profiling by indicating if a person has experienced financial gains or losses,
# without the complexity of continuous skewed values that may affect model performance.


# Creating binary indicators for 'capital.gain' and 'capital.loss' and converting them to categorical data type
data['has_capital_gain'] = data['capital.gain'].apply(lambda x: 1 if x > 0 else 0).astype('category')
data['has_capital_loss'] = data['capital.loss'].apply(lambda x: 1 if x > 0 else 0).astype('category')

# Drop the original 'capital.gain' and 'capital.loss' columns to simplify the dataset
data = data.drop(['capital.gain', 'capital.loss'], axis=1)

# Display sample of the updated DataFrame to verify the changes
print(data[['has_capital_gain', 'has_capital_loss']].head())
```

```
  has_capital_gain has_capital_loss
0                0                1
1                0                1
2                0                1
3                0                1
4                0                1
```

For the education variable, we binned categories below high school into a single category called Below-HS to reduce complexity while keeping educational distinctions. Moreover, we applied ordinal encoding to preserve the natural order in education levels (Figure 21).

Figure 21

Handling Findings in the "education" Variable

```
# Handling findings in 'education' variable

# To reduce categories we group categories below HS-grad becasue they have low individual counts and grouping them does not
# affect the educational distinctions relevant for the analysis.

# Define categories below HS-grad
below_hs = ['Preschool', '1st-4th', '5th-6th', '7th-8th', '9th', '10th', '11th', '12th']

# Group these categories
data['education'] = data['education'].replace(below_hs, 'Below-HS')


# This categorical variable follows a natural order which we can preserve by applying ordinal encoding

# Define education order representing the natural progression of education levels
education_order = ['Below-HS', 'HS-grad', 'Some-college', 'Assoc-voc', 'Assoc-acdm', 'Bachelors', 'Masters', 'Prof-school', 'Doctorate']


# Convert education to Categorical.  This ensures that the categories maintain their natural order in the dataset
data['education'] = pd.Categorical(data['education'], categories=education_order, ordered=True)

# Show the updated distribution
print("\nEducation Categories:")
print(data['education'].value_counts().reindex(education_order))
```

```
Education Categories:
education
Below-HS        4248
HS-grad        10494
Some-college    7282
Assoc-voc       1382
Assoc-acdm      1067
Bachelors       5353
Masters         1722
Prof-school      576
Doctorate        413
Name: count, dtype: int64
```

In the age feature, we replaced it with a new categorical feature age_group, representing age brackets to help reduce the skewness while still keeping important patterns. (Figure 22)

Figure 22

Handling Findings in the "age" Variable

```
# Handling findings in 'age' feature

# In EDA the distribution of the "age" feature is right-skewed, with a higher concentration of individuals in the younger age range (20-50).
# Binning the feature into age brackets helps capture meaningful patterns in income and spending habits while also helps with the skewness.

# Define age bins and labels
age_bins = [0, 25, 35, 45, 55, 70, 90]
age_labels = ['Young', 'Early Adulthood', 'Mid Adulthood', 'Mature Adulthood', 'Senior Adulthood', 'Elderly']

# Create a new categorical age_group feature
data['age_group'] = pd.cut(data['age'], bins=age_bins, labels=age_labels, right=True)

# Drop the age feature
data = data.drop('age', axis=1)

# Display the distribution for verification
print(data['age_group'].value_counts())

age_group
Early Adulthood     8510
Mid Adulthood       8005
Young               6400
Mature Adulthood    5534
Senior Adulthood    3549
Elderly              539
Name: count, dtype: int64
```

Additionally, we dropped the race, sex and country.origen features on the grounds of their potential for introducing discrimination and bias when predicting, as well as the fnlwgt feature, as this feature was only relevant in the context of the senses the dataset comes from, lacking a meaningful relationship with the target variable (Figure 23).

Figure 23

Handling Findings in "age", "sex" and "country.origen" Variables

```
# Handling finding in "race", 'sex' and "country.origen" variables

# To avoid potential legal/ethical issues, we drop the sex, race and native country varaibles. Using these variable for instance to
# assess affordability or profile customers for marketing campaings target audience could have significant risk of incurring in
# discrimination practices.
# This features would need a justified reason to be included in the analyisis.
# For instance modeling to identify risk of developing a medical condition.
data = data.drop(['race','native.country', 'sex'], axis=1, errors='ignore')

# Drop the 'fnlwgt' column due to not contributing to the analysis becasue it is a sampling weight with no meaningful relationship to income.

data = data.drop('fnlwgt', axis=1)
data.info()

<class 'pandas.core.frame.DataFrame'>
Index: 32537 entries, 0 to 32560
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   workclass          32537 non-null  object
 1   education          32537 non-null  category
 2   education.num      32537 non-null  int64
 3   marital.status     32537 non-null  object
 4   occupation         32537 non-null  object
 5   relationship       32537 non-null  object
 6   hours.per.week     32537 non-null  int64
 7   income             32537 non-null  object
 8   has_capital_gain   32537 non-null  category
 9   has_capital_loss   32537 non-null  category
 10  age_group          32537 non-null  category
dtypes: category(4), int64(2), object(5)
memory usage: 2.1+ MB
```

To address the outliers found in the continuous features education.num and hours.per.week, we capped the values exceeding 1.5 times the Inter Quantile Range beyond the lower un upper bound (Figure 24).

Figure 24

Capping Outliers in Continuous Features

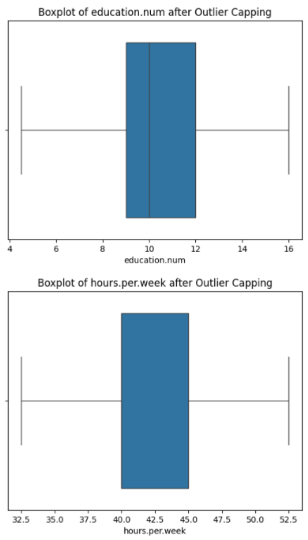
```python
# Addressing outliers found in the EDA for education.num and hours.per.week features

# Applies IQR-based capping which sets upper and lower bounds using the Interquartile Range (IQR). Any values beyond these bounds (outliers) are capped

# Function to apply IQR-based capping for outliers
def cap_outliers(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Cap/floor the outliers
    data[column] = np.where(data[column] < lower_bound, lower_bound,
                            np.where(data[column] > upper_bound, upper_bound, data[column]))

# Apply capping to the selected features
for column in ['education.num', 'hours.per.week']:
    cap_outliers(data, column)
```

And then proceeded to verify the effect of outlier capping on the features through boxplots (Figure 25), finding a distribution without visible outliers.

Figure 25

Boxplot Visualizing the Effect of Outlier Capping

```python
# Verify the distribution post-capping
for column in ['education.num', 'hours.per.week']:
    plt.figure()
    sns.boxplot(x=data[column])
    plt.title(f'Boxplot of {column} after Outlier Capping')
    plt.show()
```



Then we examined the correlation between education and education.num features as both represent levels of education, potentially providing redundant information. We cross-tabulated the features (Figure 26), finding a one-to-one correspondence, confirming redundancy and consequently dropped the education feature.

Figure 26

Cross-Tabulation of "education and education.num" Features

```
# Cross-tabulate to explore the relationship between `education` and `education.num`

pd.crosstab(index=data['education'], columns=data['education.num'])
```

education.num education	4.5	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0
Below-HS	1193	514	933	1175	433	0	0	0	0	0	0	0	0
HS-grad	0	0	0	0	0	10494	0	0	0	0	0	0	0
Some-college	0	0	0	0	0	0	7282	0	0	0	0	0	0
Assoc-voc	0	0	0	0	0	0	0	1382	0	0	0	0	0
Assoc-acdm	0	0	0	0	0	0	0	0	1067	0	0	0	0
Bachelors	0	0	0	0	0	0	0	0	0	5353	0	0	0
Masters	0	0	0	0	0	0	0	0	0	0	1722	0	0
Prof-school	0	0	0	0	0	0	0	0	0	0	0	576	0
Doctorate	0	0	0	0	0	0	0	0	0	0	0	0	413

```
# Since education.num and education essentially give the same information (each category in education corresponds to a single, specific value
# in education.num), keeping only one of them would simplify the model without losing predictive power.
# The model would not learn more by getting information from the two variables compared with just getting information from one of the variables.

# Dropping the 'education' feature
data = data.drop(columns=['education'])

# Display the updated DataFrame structure to verify the change
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 32537 entries, 0 to 32560
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   workclass         32537 non-null  object
 1   education.num     32537 non-null  float64
 2   marital.status    32537 non-null  object
 3   occupation        32537 non-null  object
 4   relationship      32537 non-null  object
 5   hours.per.week    32537 non-null  float64
 6   income            32537 non-null  object
 7   has_capital_gain  32537 non-null  category
 8   has_capital_loss  32537 non-null  category
 9   age_group         32537 non-null  category
dtypes: category(3), float64(2), object(5)
memory usage: 2.1+ MB
None
```

In preparation for the classification models, we encoded the features (Figure 27). The target feature income was binary encoded using 1 to represent income greater than or equal to $50,000 and 0 otherwise. For the age_group we used ordinal encoding to preserve its natural order and for nominal categorical features, we used one-hot encoding, creating a new feature for each of their categories and 1 and 0 to represent the absence of lack of each category in a given sample.

Figure 27

Encoding of Features for Use in Classification Models

```
# Encode non-numerical variables

# Encode binary variables
data['income'] = data['income'].apply(lambda x: 1 if x == '>50K' else 0)

# Encode the ordinal variable 'age_group'
ordinal_encoder = OrdinalEncoder()
data['age_group'] = ordinal_encoder.fit_transform(data[['age_group']])

# One-hot encode nominal categorical variables
data = pd.get_dummies(data, columns=['workclass', 'marital.status', 'occupation', 'relationship'], drop_first=True)
data = data.astype({col: 'int' for col in data.select_dtypes(include='bool').columns})

# Display the first few rows to confirm the encoding
data.head()
```

	education.num	hours.per.week	income	has_capital_gain	has_capital_loss	age_group	workclass_Local-gov	workclass_Other	workclass_Private	workclass_Self-emp-inc	...
0	9.0	40.0	0	0	1	1.0	0	0	0	0	...
1	9.0	32.5	0	0	1	1.0	0	0	1	0	...
2	10.0	40.0	0	0	1	4.0	0	0	0	0	...
3	4.5	40.0	0	0	1	2.0	0	0	1	0	...
4	10.0	40.0	0	0	1	3.0	0	0	1	0	...

5 rows × 24 columns

Next, we separated the target variable income from the features in the dataset to prepare for the classification task (Figure 28).

Figure 28

Splitting Target Variable from Features

```python
# Separate the features (X) and the target variable (y)

# Find the column index for the 'income' column
income_index = data.columns.get_loc('income')

# Separate the features and target using iloc
X = data.iloc[:, data.columns != 'income']
y = data.iloc[:, income_index]

# Display to confirm separation
print("Features (X):")
print(X.head())
print("\nTarget (y):")
print(y.head())
```

```
Features (X):
   education.num  hours.per.week  has_capital_gain  has_capital_loss  age_group  \
0            9.0            40.0                 0                 1        1.0
1            9.0            32.5                 0                 1        1.0
2           10.0            40.0                 0                 1        4.0
3            4.5            40.0                 0                 1        2.0
4           10.0            40.0                 0                 1        3.0

   workclass_Local-gov  workclass_Other  workclass_Private  \
0                    0                0                  0
1                    0                0                  1
2                    0                0                  0
3                    0                0                  1
4                    0                0                  1

   workclass_Self-emp-inc  workclass_Self-emp-not-inc  ...  \
0                       0                           0  ...
1                       0                           0  ...
2                       0                           0  ...
3                       0                           0  ...
4                       0                           0  ...

   marital.status_Never-married  marital.status_Separated  \
0                             0                         0
1                             0                         0
2                             0                         0
3                             0                         0
4                             0                         1

   marital.status_Widowed  occupation_Manual Workers  occupation_Others  \
0                       1                          0                  1
1                       1                          0                  0
2                       1                          0                  1
3                       0                          1                  0
4                       0                          0                  0

   occupation_White-Collar Workers  relationship_Not-in-family  \
0                                0                           1
1                                0                           1
2                                0                           1
3                                0                           1
4                                0                           0

   relationship_Other-relative  relationship_Own-child  relationship_Wife
0                            0                       0                  0
1                            0                       0                  0
2                            0                       0                  0
3                            0                       0                  0
4                            0                       1                  0

[5 rows x 23 columns]

Target (y):
0    0
1    0
2    0
3    0
4    0
Name: income, dtype: int64
```

Then, to ensure that differences in feature scales do not lead to biases in the models, particularly in the distance-based KNN algorithm, we normalised the values in the training set to fit within the range of 0 and 1 and then applied the same scaling parameters to the test set to maintain consistency (Figure 29).

Figure 29

Normalisation of Numerical Features

```
# Scaling data

scaler = MinMaxScaler()
X_train_val_scaled = scaler.fit_transform(X_train_val)
X_test_scaled = scaler.transform(X_test)
```

Then we carried out feature selection (Figure 30) to reduce the dataset dimensionality and focus on the most relevant features for the model using a Decision Tree (Han, Pei, & Tong, 2023), which is an algorithm that uses a criterion to select a single feature to split a root node that contains all the samples in the dataset into two child nodes. In our case, the criterion uses the Gini Index improvement after the split. The Gini Index measures the purity of nodes with respect to the target variable by measuring the probability that a randomly chosen pair of samples from the same node belongs to different classes. Consequently, the criterion for selecting the best feature for splitting is selecting the one that most effectively divides the dataset into two homogeneous child nodes, each containing samples that predominantly belong to a single class of the target feature. The process then continues iteratively branching out and creating splits and new internal nodes until a stopping condition is met, such as nodes becoming 100% pure, nodes having too few samples or maximum tree depth is reached. The resulting nodes are called leaf nodes. At each split, the features are scored and ranked based on their ability to reduce homogeneity. The feature importance score of a feature represents its accumulated score across the entire tree. Using this algorithm through the sklearn.tree module in the Scikit-learn library, using its default Gini split and stop criteria as well as setting a feature importance threshold of 0.01, we could reduce the dataset to the top 12 features, including marital.status_Married, education.num, and has_capital_gain, while less important features like relationship_Own-child and workclass_Other were dropped.

Figure 30

Feature Selection Process Using a Decision Tree

```python
# Feature Selection using a Decision Tree


model = DecisionTreeClassifier(random_state=42)
model.fit(X_train_val_scaled, y_train_val)

# Get feature importances from the trained model
importances = model.feature_importances_
features = X_train_val.columns

# Create a DataFrame to organize and display feature importances clearly
feature_importances = pd.DataFrame({'Feature': features, 'Importance': importances})
feature_importances = feature_importances.sort_values(by='Importance', ascending=False)

# Set a threshold to select top features (keeping features with importance above 0.01)
threshold = 0.01
selected_features = feature_importances[feature_importances['Importance'] > threshold]['Feature']

# Display feature importances
print("Feature Importances:")
print(feature_importances)

# Use only the selected features for training, validation, and test sets
X_train_val_selected = X_train_val_scaled[:, selected_features.index]
X_test_selected = X_test_scaled[:, selected_features.index]

# Display the selected features and the ones that were not selected
print("\nSelected features based on threshold:")
print(selected_features)
unselected_features = feature_importances[feature_importances['Importance'] <= threshold]['Feature']
print("\nFeatures not selected (dropped):")
print(unselected_features)
```

```
Feature Importances:
                          Feature  Importance
12           marital.status_Married    0.283368
0                     education.num    0.224692
1                    hours.per.week    0.117005
4                         age_group    0.094338
2                  has_capital_gain    0.052906
16         occupation_Manual Workers    0.024560
7                 workclass_Private    0.023562
3                  has_capital_loss    0.022021
18   occupation_White-Collar Workers    0.021392
22                relationship_Wife    0.018131
9          workclass_Self-emp-not-inc    0.016488
5                workclass_Local-gov    0.014842
19        relationship_Not-in-family    0.014524
17                occupation_Others    0.014144
8            workclass_Self-emp-inc    0.013781
10               workclass_State-gov    0.011472
13      marital.status_Never-married    0.008848
14         marital.status_Separated    0.005376
15           marital.status_Widowed    0.005076
11                workclass_Unknown    0.004778
21           relationship_Own-child    0.004467
20      relationship_Other-relative    0.003996
6                   workclass_Other    0.000234

Selected features based on threshold:
12           marital.status_Married
0                     education.num
1                    hours.per.week
4                         age_group
2                  has_capital_gain
16         occupation_Manual Workers
7                 workclass_Private
3                  has_capital_loss
18   occupation_White-Collar Workers
22                relationship_Wife
9          workclass_Self-emp-not-inc
5                workclass_Local-gov
19        relationship_Not-in-family
17                occupation_Others
8            workclass_Self-emp-inc
10               workclass_State-gov
Name: Feature, dtype: object

Features not selected (dropped):
13    marital.status_Never-married
14        marital.status_Separated
15          marital.status_Widowed
11               workclass_Unknown
21          relationship_Own-child
20     relationship_Other-relative
6                  workclass_Other
Name: Feature, dtype: object
```

3.2 Implementation

In our first approach to implementing supervised learning algorithms for the task at hand we used the K-Nearest Neighbors (KNN) algorithm (Tan et al., 2019) a classification algorithm that predicts the target class of a sample based on the majority target class of its nearest neighbours in the feature space, the multidimensional space of size "n" where samples are represented, being "n" the number of features. The main parameter is the number of neighbours considered for the analysis, represented by the letter K. The distance from a sample to each of their neighbouring samples is determined alternatively by the Euclidean distance, the length of the straight line between two samples in the multidimensional space where the features are represented, the Manhattan distance, the sum of the absolute differences between their corresponding feature values, or the Chebyshev distance, the largest absolute difference between their corresponding feature values in any one feature.

A first model using the KNN algorithm was instantiated with the purpose of finding the best value for the K parameter, the value that would produce the best model accuracy in predicting the target value, trying a grid of values for K ranging from 1 to 26 (Figure 31). The process included the use of the Synthetic Minority Oversampling Technique (SMOTE) in the training set, a technique to deal with unbalanced datasets, as was our case, as seen during the EDA stage, indicating that most of the samples would fall under the label corresponding to income lower than $50,000. SMOTE oversamples the minority class until its number of samples matches that of the majority class by placing synthetic samples between minority class samples and their neighbours in the feature space (Chawla et al., 2002). Similarly, this model used cross-validation, an approach which implies splitting the training data into a number of folds and training the model several times, each time leaving a different fold for validation and using the rest for training. The best prediction accuracy was found using a K value of 6.

Figure 31
Grid Search Results for Optimising K Parameter

```
# Training and evaluation to find best k (Hyperparameter tunning)

# Define a pipeline with SMOTE (to handle class imbalance) and KNN classifier
pipeline = ImbPipeline([
    ('smote', SMOTE(random_state=42)),
    ('knn', KNeighborsClassifier())
])

# Defines the range of k values to test with GridSearchCV for k tunning
param_grid = {
    'knn__n_neighbors': range(1, 26)
}

# Set up GridSearchCV with the pipeline
# Initialises an instance of GridSearchCV object which performs K parameter hypertuning using a grid method while using cross-validation (CV)
# by splitting the data into 5 folds, maintaining the proportion of each class in each fold.
# GridSearchCV uses Stratified K-Folds by default, so each fold has a similar class distribution as the whole dataset.
grid_search = GridSearchCV(estimator=pipeline, param_grid=param_grid, cv=5, scoring='accuracy')

# Executes the k hyperparameter tuning with cross validation using the training and validation data
grid_search.fit(X_train_val_selected, y_train_val)

# Best k parameter
best_k = grid_search.best_params_['knn__n_neighbors']
print(f'Best k value: {best_k}')

Best k value: 6
```

Then we instantiated a second model using KNN, but this time with the focus on obtaining the performance metrics of the trained model on validation data and validating the selection of the value for K (Figure 32). On this occasion, we also used cross-validation and SMOTE in the training set, and the parameter K was given the optimal value found in the previous step. The accuracy metric, the overall proportion of correctly classified samples was found to be 81.79%, the precision metric, the proportion of people correctly predicted to earn above $50,000 out of all the people the model predicted as earning above $50,000 was found to be 75.10%, the recall metric, the proportion of people correctly predicted to earn above $50,000 out of all the people who actually earn above $50,000 in the dataset, was found to be 75.40% and the f1-Score metric that combines precision and recall metrics was found to be 75.25%

Figure 32

Performance Metrics for KNN Model on Validation Data

```python
# Training with optimised k hyperparameter, SMOTE and cross validation to obtaining validation metrics


# instantiates an object of the scikit Learn StratifiedKFold Class to split data for cross validation
knn_cv = StratifiedKFold(n_splits=5)

# Collect cross-validation metrics (accuracy, precision, recall, F1) from all folds
all_cv_accuracies = []
all_cv_precisions = []
all_cv_recalls = []
all_cv_f1s = []

# Iterates over the 5 folds defined by StratifiedKFold generating training and validation subsets in each fold.
for train_idx, val_idx in knn_cv.split(X_train_val_selected, y_train_val):
    X_train, X_val = X_train_val_selected[train_idx], X_train_val_selected[val_idx]
    y_train, y_val = y_train_val.iloc[train_idx], y_train_val.iloc[val_idx]

    # Apply SMOTE to the training part of the current fold
    smote = SMOTE(random_state=42)
    X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

    # Instantiates a knn model with the best k
    knn = KNeighborsClassifier(n_neighbors=best_k)

    #Train the model using the smote'd training data
    knn.fit(X_train_smote, y_train_smote)

    # Predict and calculate metrics on the fold's validation data
    y_val_pred = knn.predict(X_val)

    # Store this fold's validation metrics
    all_cv_accuracies.append(accuracy_score(y_val, y_val_pred))
    all_cv_precisions.append(precision_score(y_val, y_val_pred, average='macro', zero_division=0))
    all_cv_recalls.append(recall_score(y_val, y_val_pred, average='macro'))
    all_cv_f1s.append(f1_score(y_val, y_val_pred, average='macro'))

# Calculates and print aggregated cross-validation metrics
print('Aggregated metrics from cross-validation:')
print(f'Average CV Accuracy: {sum(all_cv_accuracies) / len(all_cv_accuracies)}')
print(f'Average CV Precision: {sum(all_cv_precisions) / len(all_cv_precisions)}')
print(f'Average CV Recall: {sum(all_cv_recalls) / len(all_cv_recalls)}')
print(f'Average CV F1 Score: {sum(all_cv_f1s) / len(all_cv_f1s)}')
```

```
Aggregated metrics from cross-validation:
Average CV Accuracy: 0.8178956373031487
Average CV Precision: 0.7510905611849782
Average CV Recall: 0.7540534476801667
Average CV F1 Score: 0.7525294181735156
```

The next step was training a final instance of the model using KNN and SMOTE (Figure 33) but this time avoiding cross validation as this technique holds a fold of data on each iteration and for the final model the intention is to give it all the available training data without performing validation to increase its opportunity to learn the patterns and because its final test and metrics will be later performed using the testing data.

Figure 33

Training Final KNN Model

```
# Training of the optimised model

# Apply SMOTE to the entire training/validation set before final model training
smote = SMOTE(random_state=42)
X_train_val_smote, y_train_val_smote = smote.fit_resample(X_train_val_selected, y_train_val)

# Instantiates a new instance of knn model with the best k
final_knn = KNeighborsClassifier(n_neighbors=best_k)

# Train the final KNN model with the best k on the full SMOTE-balanced training set
final_knn.fit(X_train_val_smote, y_train_val_smote)
```

Next, the final model was tested using the testing set of data, and its performance metrics are obtained (Figure 34), showing an accuracy of 81.82%, a precision of 75.15%, a recall of 74.98%, and an F1-score of 75.06%.

Figure 34

KNN Model Test on Test Data

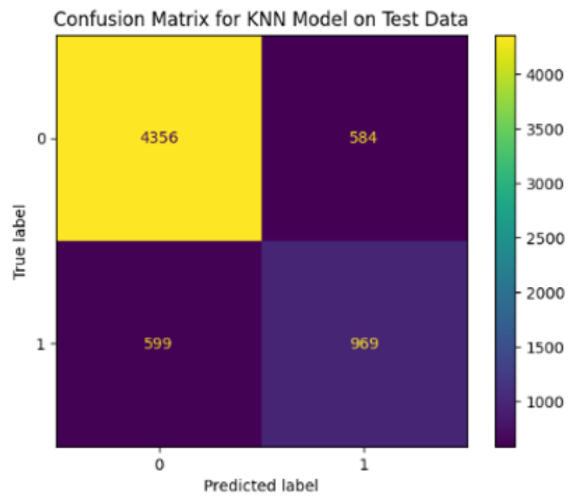
```
# Evaluation
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Evaluate the model on the test set
knn_y_test_pred = final_knn.predict(X_test_selected)

# Performance metrics
final_test_accuracy = accuracy_score(y_test, knn_y_test_pred)
final_precision = precision_score(y_test, knn_y_test_pred, average='macro', zero_division=0)
final_recall = recall_score(y_test, knn_y_test_pred, average='macro')
final_f1 = f1_score(y_test, knn_y_test_pred, average='macro')

# Print final test evaluation metrics
print('Metrics after testing on the test dataset:')
print(f'Final Test Accuracy: {final_test_accuracy}')
print(f'Final Test Precision: {final_precision}')
print(f'Final Test Recall: {final_recall}')
print(f'Final Test F1 Score: {final_f1}')
```

```
Metrics after testing on the test dataset:
Final Test Accuracy: 0.8182237246465888
Final Test Precision: 0.7515328230962111
Final Test Recall: 0.7498830351978849
Final Test F1 Score: 0.7506997455985702
```

We also generated a confusion matrix, which is a table used to evaluate the performance of classification models by comparing the true labels to the predicted labels. The confusion matrix (Figure 35) showed that the model correctly predicted 4,356 instances for individuals earning less than $50,000 (true negatives) an 88.2% of the individuals who actually earn less than $50,000, and 969 instances for individuals earning $50,000 or more (true positives) a 61.8% of the individuals who actually earn $ 50,000 or more. Therefore, the model struggles more with correctly identifying individuals earning $50,000 or more (positive class) relative to their total population in the dataset. Moreover, the model misclassified 584 individuals who earn less than $50,000 as earning $50,000 or more (false positives) and 599 individuals who earn $50,000 or more as earning less than $50,000 (false negatives).

Figure 35
Confusion Matrix of KNN Model Predictions

```
# Generate a confusion matrix
knn_conf_matrix = confusion_matrix(y_test, knn_y_test_pred)

# Display the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=knn_conf_matrix)
disp.plot()
plt.title("Confusion Matrix for KNN Model on Test Data")
plt.show()
```



As our next approach, we opted for implementing a model using a Random Forest, which is an ensemble learning method that combines multiple decision trees, like the one previously described, where each tree is trained on a random subset of the data and the final prediction is made by aggregating the predictions from all the individual trees (Han, Pei, & Tong, 2023).

We started by using a grid of possible parameters to train a Random Forest model to search for the best hyperparameters for the task (Figure 36). The grid search considered the number of trees, maximum depth of each tree, minimum samples required to split an internal node and minimum samples required at a leaf node. The optimal values found were 150, no maximum depth, 10 and 1, respectively. On this occasion, we used cross-validation and, to handle the imbalance of the dataset, the class_weight parameter in the random forest object was instantiated.

Figure 36

Grid Search Results for Optimising Random Forest Parameters

```
# Training with grid method for hyperparameter tunning

# Defines the range of hyperparameters to tune with GridSearchCV
param_grid = {
    # number of trees in the Random Forest
    'n_estimators': [50, 100, 150],
     # maximum depth of each tree in the forest
    'max_depth': [None, 10, 20],
     # minimum number of samples required to split an internal node
    'min_samples_split': [2, 5, 10],
     # minimum number of samples required to be at a leaf node
    'min_samples_leaf': [1, 2, 4],
    # handle class imbalance by assigning weights to the classes
    'class_weight': ['balanced', 'balanced_subsample']
}

# Initialise a Random Forest classifier for grid search
rf_base = RandomForestClassifier(random_state=42)

# Set up Grid Search with 5-fold cross-validation using the preprocessed training/validation data
# n_jobs=-1 parameter inidcates to use all the available cpu's in parallel to expedite the search
grid_search = GridSearchCV(estimator=rf_base, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)

# Executes the grid search for hyperparameters tuning with cross validation using the training and validation data
grid_search.fit(X_train_val_selected, y_train_val)

# Best hyperparameters
best_params = grid_search.best_params_
print(best_params)
{'class_weight': 'balanced', 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 10, 'n_estimators': 150}
```

Then we generated a new instance of random forest model, also using cross-validation and the class_weight parameter to handle the dataset imbalance, this time with the purpose of obtaining the metrics on the validation data after the training (Figure 37) finding an average accuracy of 80.88%, a precision of 83.09%, a recall of 89.07%, and an average F1 score of 86.10%

Figure 37

Performance Metrics for Random Forest Model on Validation Data

```python
# Cross-validation with the best model to obtain validation metrics

# instantiates an object of the scikit learn StratifiedKFold Class to split data for cross validation
rf_cv = StratifiedKFold(n_splits=5)

# Lists to hold cross-validation metrics
rf_all_cv_accuracies = []
rf_all_cv_precisions = []
rf_all_cv_recalls = []
rf_all_cv_f1s = []

# Splits the data into the number of folds specified by rf_cv, each folds contains training and validation data subsets
for rf_train_idx, rf_val_idx in rf_cv.split(X_train_val_selected, y_train_val):
    rf_X_train, rf_X_val = X_train_val_selected[rf_train_idx], X_train_val_selected[rf_val_idx]
    rf_y_train, rf_y_val = y_train_val.iloc[rf_train_idx], y_train_val.iloc[rf_val_idx]


    # Instantiates a new random forest model using the optimized parameters
    rf = RandomForestClassifier(
    n_estimators=best_params['n_estimators'],
    max_depth=best_params['max_depth'],
    min_samples_split=best_params['min_samples_split'],
    min_samples_leaf=best_params['min_samples_leaf'],
    class_weight=best_params['class_weight'],
    random_state=42
)

    # Train the best model with this fold's train data
    rf.fit(rf_X_train, rf_y_train)

    # Evaluate the model on the corresponding fold's validation set
    rf_y_val_pred = rf.predict(rf_X_val)

    # Calculate performance metrics
    rf_acc = accuracy_score(rf_y_val, rf_y_val_pred)
    rf_prec = precision_score(rf_y_val, rf_y_val_pred, average='macro', zero_division=0)
    rf_rec = recall_score(rf_y_val, rf_y_val_pred, average='macro')
    rf_f1 = f1_score(rf_y_val, rf_y_val_pred, average='macro')

    # Store performance metrics for each fold
    rf_all_cv_accuracies.append(rf_acc)
    rf_all_cv_precisions.append(rf_prec)
    rf_all_cv_recalls.append(rf_rec)
    rf_all_cv_f1s.append(rf_f1)

# Summarise and print results from cross-validation
print('Aggregated metrics from Random Forest cross-validation:')
print(f'Average RF CV Accuracy: {sum(rf_all_cv_accuracies) / len(rf_all_cv_accuracies)}')
print(f'Average RF CV Precision: {sum(rf_all_cv_precisions) / len(rf_all_cv_precisions)}')
print(f'Average RF CV Recall: {sum(rf_all_cv_recalls) / len(rf_all_cv_recalls)}')
print(f'Average RF CV F1 Score: {sum(rf_all_cv_f1s) / len(rf_all_cv_f1s)}')
```

```
Aggregated metrics from Random Forest cross-validation:
Average RF CV Accuracy: 0.8088288876759726
Average RF CV Precision: 0.7501049834733697
Average RF CV Recall: 0.8029979122510629
Average RF CV F1 Score: 0.7661024795180243
```

Then we trained a new instance of a random forest model using all the available training data and no validation to increase the prediction performance of this model looking at validating the model's performance later using the testing data (Figure 38)

Figure 38

Training Final Random Forest Model

```
# Train the model on the entire training/validation set with the best cross-validated model

# Instantiates a new instance of random forest model with the best hyperparameters
rf_final = RandomForestClassifier(
    n_estimators=best_params['n_estimators'],
    max_depth=best_params['max_depth'],
    min_samples_split=best_params['min_samples_split'],
    min_samples_leaf=best_params['min_samples_leaf'],
    class_weight=best_params['class_weight'],
    random_state=42
)

# Train the final random forest model with the optimised hyperparameters on the full training set
rf_final.fit(X_train_val_selected, y_train_val)
```

Next, we tested the model using the test dataset and produced its performance metrics, finding an overall accuracy of 80.87%, precision of 74.89%, recall of 79.97% and f1-score of 76.48% (Figure 39)

Figure 39

Performance Metrics for KNN Model on Validation Data

```
# Evaluation

# Evaluate the model on the test set
rf_y_test_pred = rf_final.predict(X_test_selected)

# Performance metrics
rf_final_test_accuracy = accuracy_score(y_test, rf_y_test_pred)
rf_final_precision = precision_score(y_test, rf_y_test_pred, average='macro', zero_division=0)
rf_final_recall = recall_score(y_test, rf_y_test_pred, average='macro')
rf_final_f1 = f1_score(y_test, rf_y_test_pred, average='macro')

# Print the final test set evaluation metrics
print('Random Forest metrics after testing on the test dataset:')
print(f'Final RF Test Accuracy: {rf_final_test_accuracy}')
print(f'Final RF Test Precision: {rf_final_precision}')
print(f'Final RF Test Recall: {rf_final_recall}')
print(f'Final RF Test F1 Score: {rf_final_f1}')
```

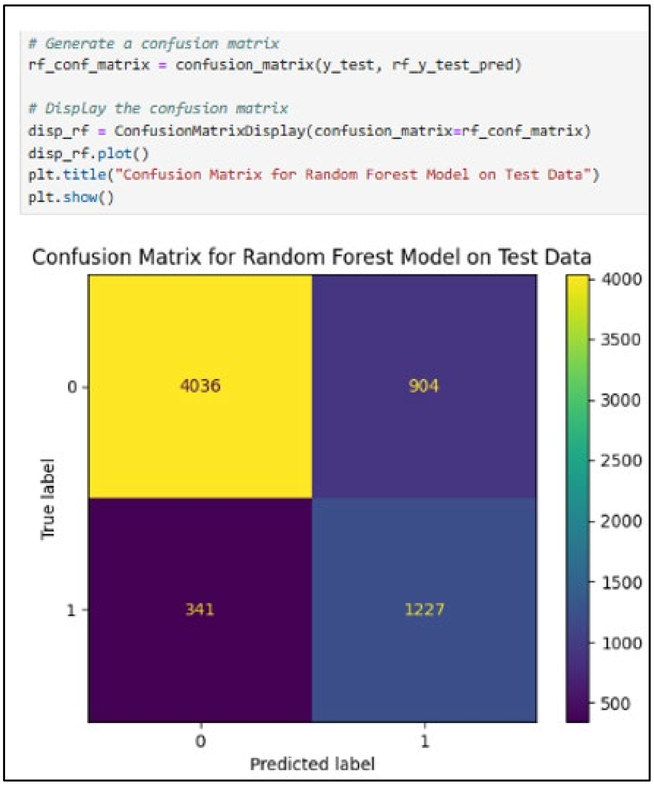
```
Random Forest metrics after testing on the test dataset:
Final RF Test Accuracy: 0.8086969883220652
Final RF Test Precision: 0.748939386775739
Final RF Test Recall: 0.7997647793935387
Final RF Test F1 Score: 0.764897921379794
```

A confusion matrix (Figure 40) showed that the model correctly predicted 4,036 instances for individuals earning less than $50,000 (true negatives), covering 81.7% of the individuals who actually earn less than $50,000, and 1,227 instances for individuals earning

$50,000 or more (true positives), representing 78.25% of the individuals who actually earn $50,000 or more. Therefore, the model performs relatively better at predicting individuals earning $50,000 or more (positive class) than the other class. Additionally, the model misclassified 904 individuals who earn less than $50,000 as earning $50,000 or more (false positives) and 341 individuals who earn $50,000 or more as earning less than $50,000 (false negatives).

Figure 40

Confusion Matrix of Random Forest Model Predictions



3.3 Results Analysis and Discussion

The overall performance of both models implemented with Python was found to be satisfactory, allowing to make predictions that are above random guessing. Most metrics were found to have marginal differences. It is worth noting that both models struggled to predict the $50,000 or over income class more than the below $50,000 income class, possibly due to the imbalance in the training set, with the $50,000 or over income being the minority class and despite the efforts to balance it using SMOTE in the KNN model and class_weight parameter in the random forest model. The Random Forest model achieved a higher recall, correctly identifying 78.25% of individuals earning $50,000 or more, compared to 61.8% by the KNN model, which makes it a better choice when identifying as many high earners as possible is important, such as in marketing campaigns targeting high-income individuals for luxury products. However, the KNN model produces fewer false positives, misclassifying only 584 individuals earning less than $50,000 as earning more, compared to 904 in the Random Forest model, which makes KNN more suitable when it is important to avoid allocating resources incorrectly, such as offering benefits to individuals wrongly classified as high earners.

Similarly, the overall performance of both Azure models was satisfactory, demonstrating they can make predictions significantly better than random guessing. Most metrics showed similar performance, with the Boosted Decision Tree achieving slightly better recall (71.36% vs. 70.98%) and F1-score (63.77% vs. 63.67%), suggesting it may be marginally better at identifying individuals earning $50,000 or more (positive class), which could make it the best choice in scenarios prioritising the identification of high earners.

While the Azure models achieved better recall for high-income individuals, their lower precision (57.72% and 57.65%) compared to the Python models indicates they misclassified more individuals as high earners (false positives) However, the Azure models (Two-Class Decision Forest and Boosted Decision Tree) have higher recall metrics for this class, which indicates they perform better in identifying high-income individuals than the Python-based models (especially KNN).

4. Conclusions

The analysis demonstrated that Machine Learning classification algorithms can effectively classify individuals into income groups in real-world classification tasks, allowing businesses to implement them, for instance, to identify high-income segments for premium offerings or luxury products, ensuring campaigns focus on the clients most likely to acquire them, expediting the decision-making process. Companies should consider the nuances of the results for each algorithm based on their particular requirements, as, despite overall performance, some algorithms are better suited for predicting higher earners, avoiding false positives than others.

References

Becker, B., & Kohavi, R. (1996). *Adult* [Dataset]. https://doi.org/10.24432/C5XW20

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

Doane, D. P., & Seward, L. E. (2011). Measuring skewness: A forgotten statistic? *Journal of Statistics Education, 19*(2). https://doi.org/10.1080/10691898.2011.11889611

Everitt, B. S. (2002). *The Cambridge dictionary of statistics* (2nd ed.). Cambridge University Press.

Han, J., Pei, J., & Tong, H. (2023). *Data mining: Concepts and techniques* (4th ed.). Elsevier.

Larose, D. T., & Larose, C. D. (2015). *Data mining and predictive analytics.* John Wiley & Sons.