Informatics Institute of Technology

Department of Computing

BSc (Hons) Artificial Intelligence and Data Science

C M2604 - Machine Learning

Coursework (Individual) Report Document

Nividula Solingage Dona

2236745 (RGU)

20220138 (IIT)

Github Repository Link

https://github.com/summyyyyy/NaiveBayes-vs-RandomForest-Income-Prediction.git

Table of Contents

Introduction

In this report, we explore the task of predicting whether an individual's income exceeds $50,000 per year based on various demographic and socioeconomic factors. The dataset used for this analysis is the 'Census Income' dataset, sourced from the UCI Machine Learning repository. This dataset contains information collected from the US Census Bureau and includes attributes such as age, education level, occupation, marital status, and more.

The objective of this analysis is to build and evaluate two machine learning models for classification: Naïve Bayes and Random Forest Classification. These models aim to accurately classify individuals into two income categories: those earning more than $50,000 annually and those earning $50,000 or less.

The report will first provide an overview of the dataset, including its structure and key attributes. We will then proceed to preprocess the data, handle missing values, encode categorical variables, and split the dataset into training and testing sets. Following this, we will build, train, and evaluate the Naïve Bayes and Random Forest models. Model performance metrics such as accuracy, precision, recall, and F1-score will be calculated to assess the effectiveness of each model in predicting income levels.

By the end of this report, we aim to provide insights into which model performs better for this classification task and gain a deeper understanding of the factors that influence income levels in the given population.

Data Exploration

In this section, we explore the 'Census Income' dataset to gain insights into its structure and characteristics.

```
df = pd.concat([X, pd.DataFrame(y, columns=['income'])],axis=1)
```

The above code merges all the information about individuals (like age, education, etc.) with their corresponding income data (whether they earn more than $50K or not) into one table. This makes it easier to analyse and build predictive models because all the necessary information is in one place.

Dataset Information

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48842 entries, 0 to 48841
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   age             48842 non-null  int64 
 1   workclass       47879 non-null  object
 2   fnlwgt          48842 non-null  int64 
 3   education       48842 non-null  object
 4   education-num   48842 non-null  int64 
 5   marital-status  48842 non-null  object
 6   occupation      47876 non-null  object
 7   relationship    48842 non-null  object
 8   race            48842 non-null  object
 9   sex             48842 non-null  object
 10  capital-gain    48842 non-null  int64 
 11  capital-loss    48842 non-null  int64 
 12  hours-per-week  48842 non-null  int64 
 13  native-country  48568 non-null  object
 14  income          48842 non-null  object
dtypes: int64(6), object(9)
memory usage: 5.6+ MB
```

The dataset includes attributes such as age, workclass, education level, marital status, occupation, race, gender, capital gains, capital losses, hours worked per week, native country, and income level.

Upon inspection, it was observed that some columns have missing values. Specifically, the 'workclass', 'occupation', and 'native-country' columns exhibit non-null counts lower than the total number of entries.

Most of the columns are of the object data type, indicating categorical variables, while some columns are of the int64 data type, representing numerical variables.

Preview of the Dataset

This allows us to understand the format of the data.

```
df.head()
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

This provides a preview of the first 5 rows, showcasing the structure of the data and the initial values of each column.

Additionally, I got a sample of five randomly selected entries from the dataset to capture the diversity of the data and its variability.

```
df.sample(5)
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
238	38	Local-gov	115076	Masters	14	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	70	United-States	>50K
5391	31	Private	101562	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Wife	White	Female	0	0	55	United-States	<=50K
14650	37	Self-emp-not-inc	112497	Masters	14	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	50	United-States	>50K
682	34	Self-emp-not-inc	190290	HS-grad	9	Married-civ-spouse	Craft-repair	Husband	White	Male	0	0	40	United-States	>50K
10570	21	?	188535	Some-college	10	Never-married	?	Own-child	White	Male	0	0	40	United-States	<=50K

This below one provides a preview of the last 5 rows.

```
df.tail()
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
48837	39	Private	215419	Bachelors	13	Divorced	Prof-specialty	Not-in-family	White	Female	0	0	36	United-States	<=50K.
48838	64	NaN	321403	HS-grad	9	Widowed	NaN	Other-relative	Black	Male	0	0	40	United-States	<=50K.
48839	38	Private	374983	Bachelors	13	Married-civ-spouse	Prof-specialty	Husband	White	Male	0	0	50	United-States	<=50K.
48840	44	Private	83891	Bachelors	13	Divorced	Adm-clerical	Own-child	Asian-Pac-Islander	Male	5455	0	40	United-States	<=50K.
48841	35	Self-emp-inc	182148	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	60	United-States	>50K.

Dataset Shape

The shape of the dataset indicates the number of rows and columns, which is crucial for understanding its size and dimensionality.

```
df.shape
```

```
(48842, 15)
```

Summary Statistics

This shows descriptive statistics, such as mean, median, minimum, maximum, and quartiles, provide insights into the central tendency and dispersion of numerical attributes. These statistics offer a high-level overview of the numerical features in the dataset.

```
df.describe()
```

	age	fnlwgt	education-num	capital-gain	capital-loss	hours-per-week
count	48842.000000	4.884200e+04	48842.000000	48842.000000	48842.000000	48842.000000
mean	38.643585	1.896641e+05	10.078089	1079.067626	87.502314	40.422382
std	13.710510	1.056040e+05	2.570973	7452.019058	403.004552	12.391444
min	17.000000	1.228500e+04	1.000000	0.000000	0.000000	1.000000
25%	28.000000	1.175505e+05	9.000000	0.000000	0.000000	40.000000
50%	37.000000	1.781445e+05	10.000000	0.000000	0.000000	40.000000
75%	48.000000	2.376420e+05	12.000000	0.000000	0.000000	45.000000
max	90.000000	1.490400e+06	16.000000	99999.000000	4356.000000	99.000000

The distribution of income levels

```
df['income'].value_counts()
```

```
<=50K      24720
<=50K.     12435
>50K        7841
>50K.       3846
Name: income, dtype: int64
```

To get some idea about Income column.

Data Pre-processing

1. Replacing erroneous income categories to standardize them for accurate classification

```
df['income'].replace({'<=50K.':'<=50K',    '>50K.':    '>50K'},
inplace = True)
```

2. Handling Missing Values

```
df.replace({'?': np.nan, ' ?': np.nan, '? ': np.nan, ' ? ':
np.nan}, inplace = True)
```

```
print("Missing Values ")
df.isna().sum()
```

```
Missing Values
age                  0
workclass         2799
fnlwgt               0
education            0
education-num        0
marital-status       0
occupation        2809
relationship         0
race                 0
sex                  0
capital-gain         0
capital-loss         0
hours-per-week       0
native-country     857
income               0
dtype: int64
```

```
# replacing NaN values with the mode of respective columns
for column in df.columns:
  mode_value = df[column].mode()[0]
  df[column].fillna(mode_value, inplace = True)

print("Missing Values after replacing NaN values with the mode")
print (df.isnull().sum())
```

```
Missing Values after replacing NaN values with the mode
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
```

Replaced '?' values with ‘NaN’ and imputed missing values with mode to maintain data integrity and ensure accurate model training.

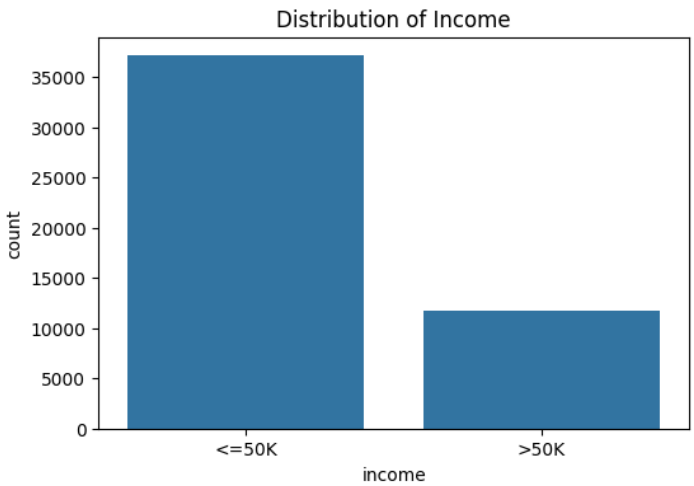
3. Duplicate Removal

Eliminated duplicate entries to avoid redundancy and maintain data quality.

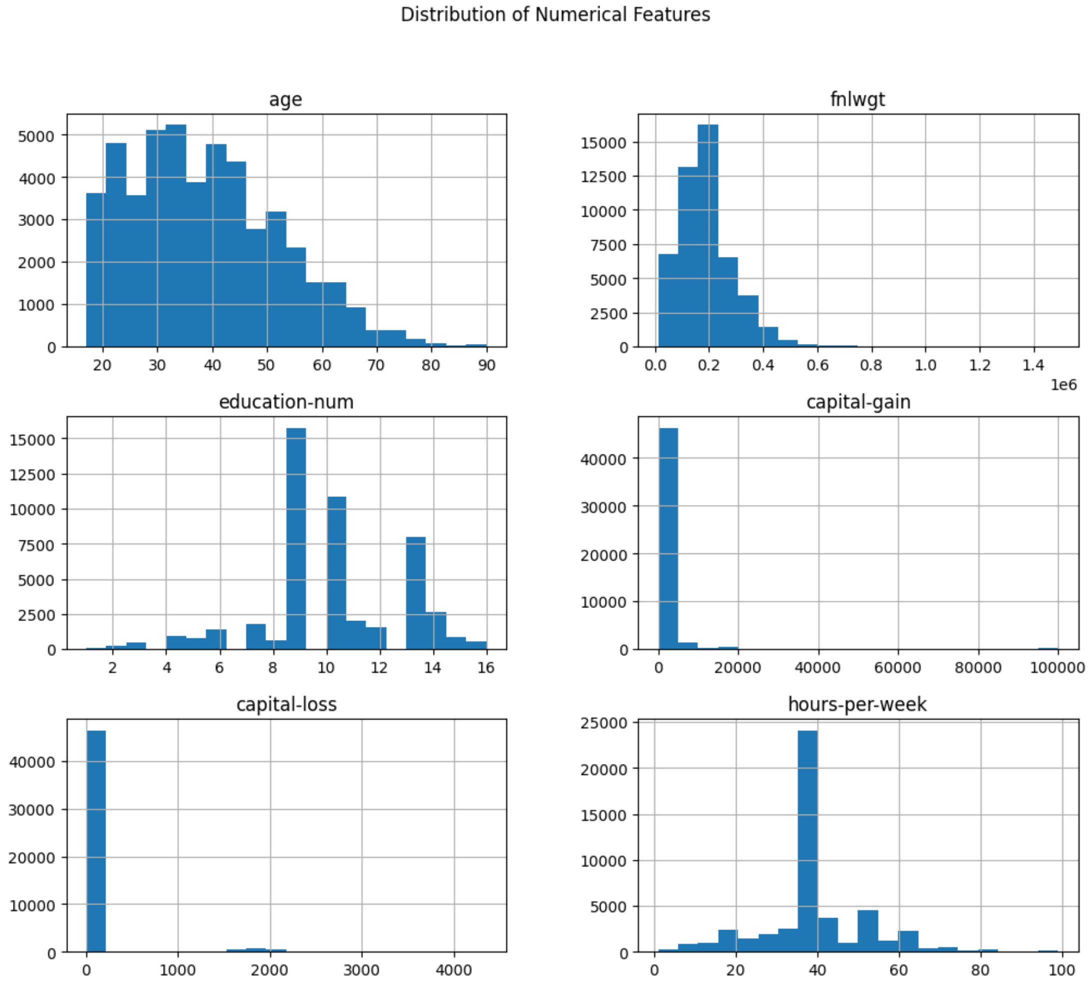
```
[17] df.shape

     (48842, 15)

[18] # drop duplicates
     df = df.drop_duplicates()

     # find duplicate values in the dataset
     print (df[df.duplicated()])

     Empty DataFrame
     Columns: [age, workclass, fnlwgt, education, education-num, marital-status, occupation, relation
     Index: []

[19] df.shape

     (48789, 15)
```

After done above tasks, I was Visualized distribution of target variable (Income), Visualized distribution of numerical features and Visualized relationship between numerical features and target (Income) for understand the distribution of the data. The below are the outputs,

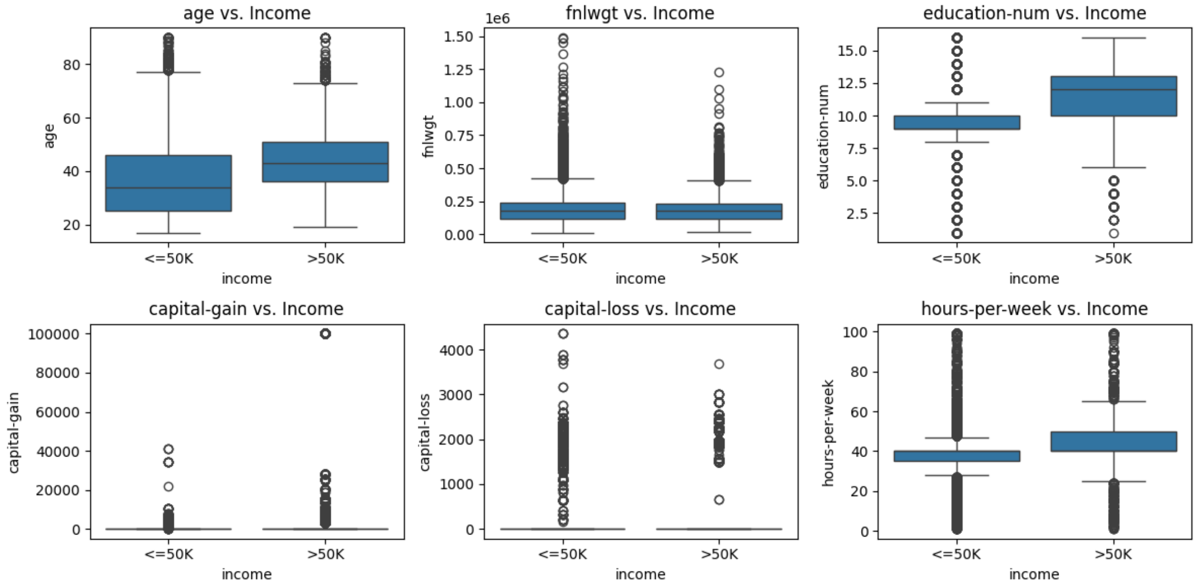
Visualize distribution of target variable (Income)



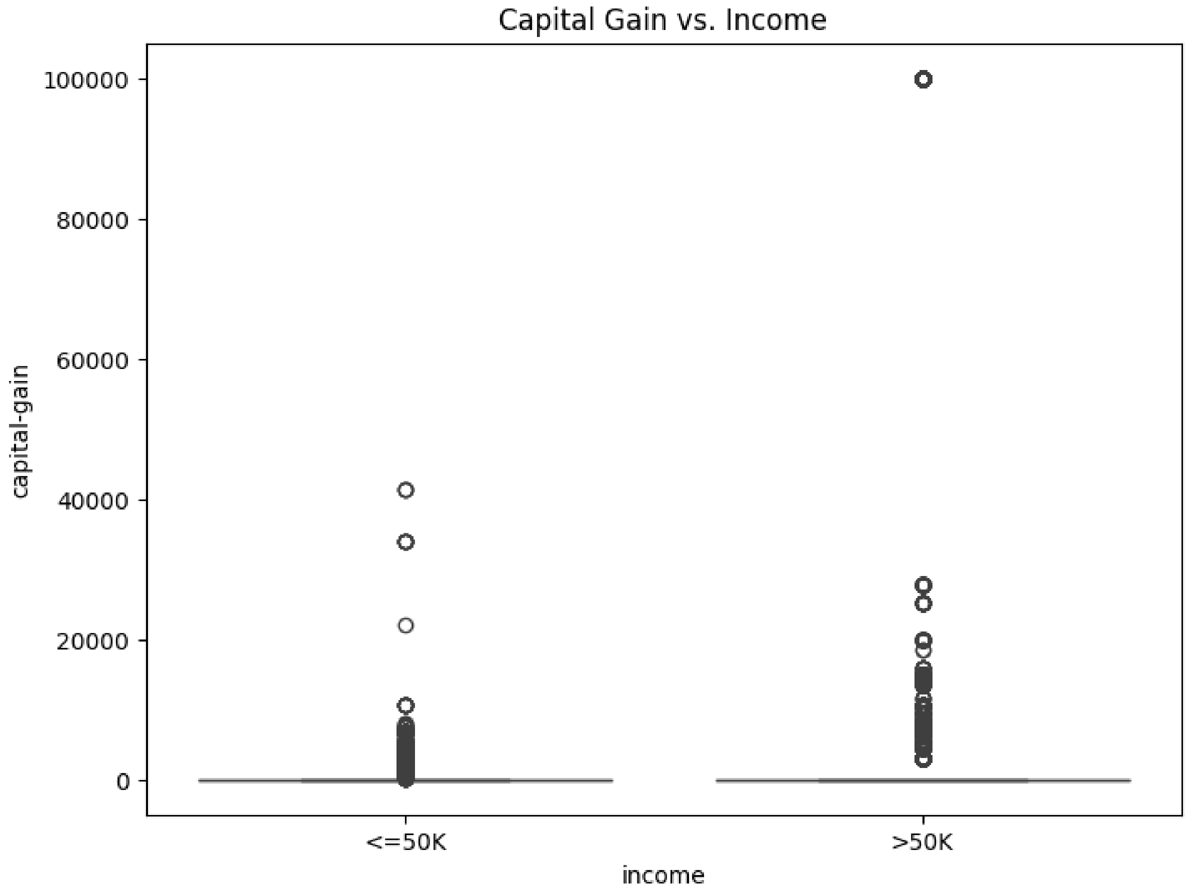
Visualize distribution of numerical features



Visualize relationship between numerical features and target (Income)



There is an outlier in the capital-gain vs. income plot.



Now we have to handle this outlier.

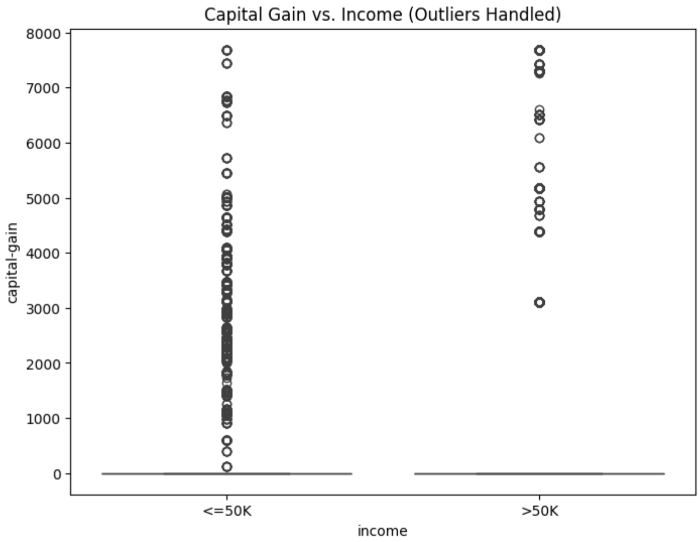
4. Outlier Handling

Outliers can significantly impact the performance of machine learning models, skewing results and reducing accuracy.

```
# Handle outlier of the capital-gain
print("Before Clipping:")
print(df['capital-gain'].describe())

# Define lower and upper bounds for clipping
lower_bound = df['capital-gain'].quantile(0.5)
upper_bound = df['capital-gain'].quantile(0.97)

# Clip the values of 'capital-gain' column
df['capital-gain'] = df['capital-gain'].clip(lower=lower_bound,
upper=upper_bound)

# Verify the changes
```

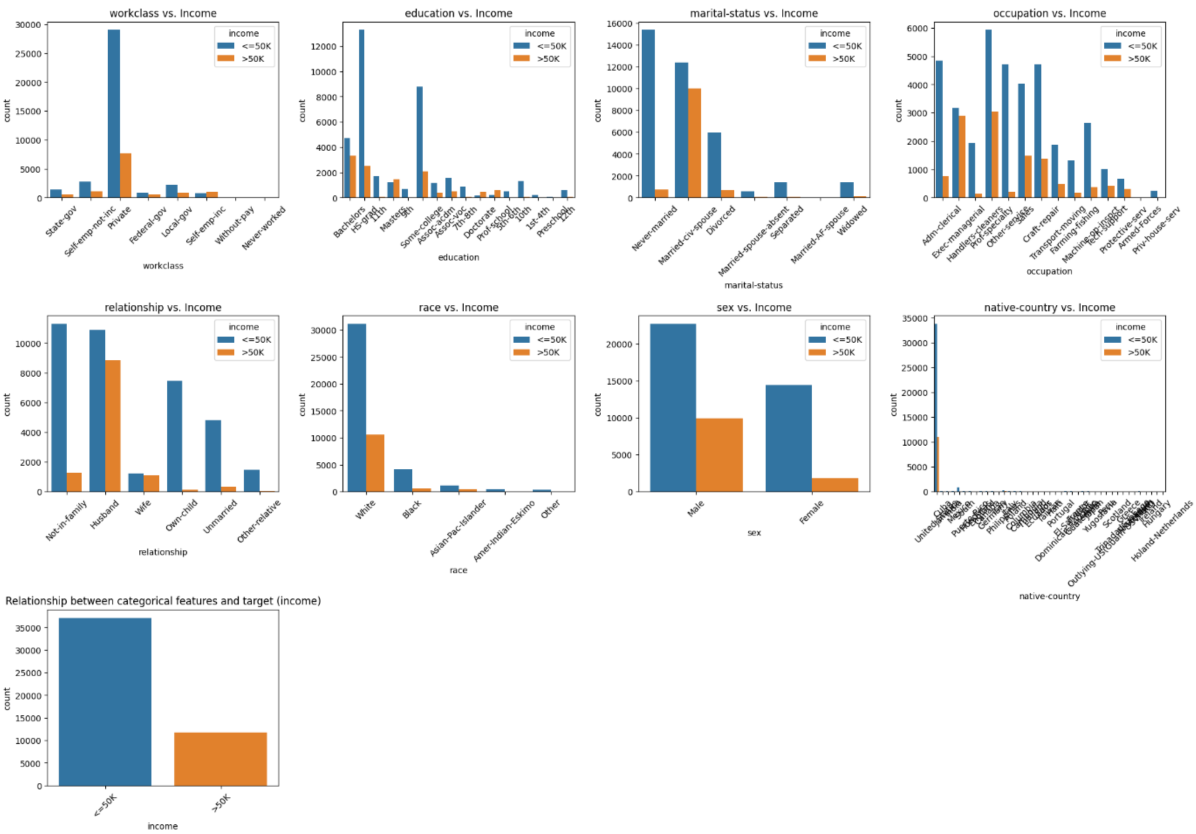
```
print("after Clipping:")
print(df['capital-gain'].describe())
```

```
Before Clipping:
count    48789.000000
mean      1080.239829
std       7455.980728
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max      99999.000000
Name: capital-gain, dtype: float64

After Clipping:
count    48789.000000
mean       464.402734
std       1679.685698
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max       7688.000000
Name: capital-gain, dtype: float64
```



I used Clipping outliers in 'capital-gain' method over other methods for handle outliers because it's a straightforward method that directly limits extreme values without altering the overall distribution significantly. Other methods like imputation or transformation might distort the data, affecting model performance.

Visualize relationship between categorical features and target



5. Irrelevant Feature Removal (Dropped 'fnlwgt' and 'education' columns)

```
df.drop(columns=['fnlwgt', 'education'], inplace=True)
```

The 'fnlwgt' (final weight) column in the dataset represents survey weights assigned by the Census Bureau. These weights are used to adjust the sample to match the actual population distribution. However, for our classification task of predicting income levels, this column doesn't provide any direct information or insight into an individual's income status. Therefore, including it in our analysis would not contribute to improving the accuracy of our income prediction model.

Similarly, the 'education' column in the dataset indicates an individual's level of education, such as 'Bachelor's degree', 'High school diploma', etc. While education level can be a significant factor influencing income, the specific details of education attainment are not necessary for our classification task. Instead, we are more interested in broader categories such as 'High school or less', 'Some college', 'Bachelor's degree or higher', which can be derived from the 'education-num' column. Hence, keeping both 'education' and 'education-num' would be redundant, and dropping 'education' simplifies our dataset without losing any crucial information for our classification task.

6. Removing Duplicates of ‘df’ again.

```
# drop duplicates
df = df.drop_duplicates()

# find duplicate values in the dataset
print (df[df.duplicated()])
```

Currant number of rows and columns - (42204, 13)

7. Categorization

When categorizing **'native-country', ‘marital-status’** and **'occupation'** into broader categories, I created another dataframe **to keep the original dataset intact for comparison and to preserve the original data.** This approach allows for experimentation without permanently altering the original data, providing flexibility in analysis and ensuring that the original information remains accessible if needed. Additionally, it facilitates the exploration of different feature engineering strategies without affecting the primary dataset, enabling better control over the data manipulation process.

Process of categorizing ‘native-country’ column

```
# categorizing
df_with_categories = df.copy()
```

```
df_with_categories.drop(columns=['native-country',
'occupation'], inplace=True)
```

```
# Define mapping for broader country categories
country_mapping = {
    'United-States': 'North America',
    'Mexico': 'North America',
    'Canada': 'North America',
    'Puerto-Rico': 'North America',
    'El-Salvador': 'North America',
    'Dominican-Republic': 'North America',
    'Jamaica': 'North America',
    'Cuba': 'North America',
    'Guatemala': 'North America',
    'Honduras': 'North America',
    'Philippines': 'Asia',
    'India': 'Asia',
    'China': 'Asia',
    'Japan': 'Asia',
    'Vietnam': 'Asia',
    'Taiwan': 'Asia',
    'Iran': 'Asia',
    'Thailand': 'Asia',
    'Hong': 'Asia',
    'Cambodia': 'Asia',
    'Laos': 'Asia',
    'Germany': 'Europe',
    'England': 'Europe',
    'Italy': 'Europe',
    'Poland': 'Europe',
    'Greece': 'Europe',
    'Portugal': 'Europe',
    'France': 'Europe',
    'Ireland': 'Europe',
    'Scotland': 'Europe',
    'Hungary': 'Europe',
    'Yugoslavia': 'Europe',
    'South': 'Latin America & Caribbean',
    'Columbia': 'Latin America & Caribbean',
    'Haiti': 'Latin America & Caribbean',
    'Peru': 'Latin America & Caribbean',
    'Ecuador': 'Latin America & Caribbean',
    'Trinadad&Tobago': 'Latin America & Caribbean',
    'Nicaragua': 'Latin America & Caribbean',
    'Outlying-US(Guam-USVI-etc)': 'Others',
    'Holand-Netherlands': 'Others',
}
```

```
# Replace specific country names with broader category names
df_with_categories['country-category'] = df['native-country'].replace(country_mapping)

# Verify the changes
print(df_with_categories['country-category'].value_counts())
```

```
North America                39981
Asia                           979
Europe                         778
Latin America & Caribbean      442
Others                          24
Name: country-category, dtype: int64
```

Process of categorizing 'occupation' column

```
# Define mapping for broader occupation categories
occupation_mapping = {
    'Prof-specialty': 'White-Collar Jobs',
    'Exec-managerial': 'White-Collar Jobs',
    'Adm-clerical': 'White-Collar Jobs',
    'Tech-support': 'White-Collar Jobs',
    'Craft-repair': 'Blue-Collar Jobs',
    'Machine-op-inspct': 'Blue-Collar Jobs',
    'Transport-moving': 'Blue-Collar Jobs',
    'Handlers-cleaners': 'Blue-Collar Jobs',
    'Sales': 'Sales & Service Jobs',
    'Other-service': 'Sales & Service Jobs',
    'Protective-serv': 'Protective & Security Jobs',
    'Priv-house-serv': 'Protective & Security Jobs',
    'Armed-Forces': 'Protective & Security Jobs',
    'Farming-fishing': 'Farming & Fishing Jobs'
}

# Replace specific occupation names with broader category names
df_with_categories['occupation-category'] = df['occupation'].replace(occupation_mapping)

# Verify the changes
print(df_with_categories['occupation-category'].value_counts())
```

```
White-Collar Jobs             19475
Blue-Collar Jobs              10991
Sales & Service Jobs           9105
Farming & Fishing Jobs         1434
Protective & Security Jobs     1199
Name: occupation-category, dtype: int64
```

Process of categorizing 'marital-status' column

```
# Mapping dictionary for categorizing marital status
marital_status_mapping = {
    'Married-civ-spouse': 'Married',
    'Married-AF-spouse': 'Married',
    'Married-spouse-absent': 'Married',
    'Never-married': 'Never-married',
    'Widowed': 'Widowed',
    'Divorced': 'Divorced',
    'Separated': 'Divorced'  # Considering 'Separated' as 'Divorced'
}

# Replace marital status with categorized values
df_with_categories['marital-status'] = df['marital-status'].replace(marital_status_mapping)

# Check the updated counts
print(df_with_categories['marital-status'].value_counts())
```

```
Married          19746
Never-married    13240
Divorced          7720
Widowed           1498
Name: marital-status, dtype: int64
```

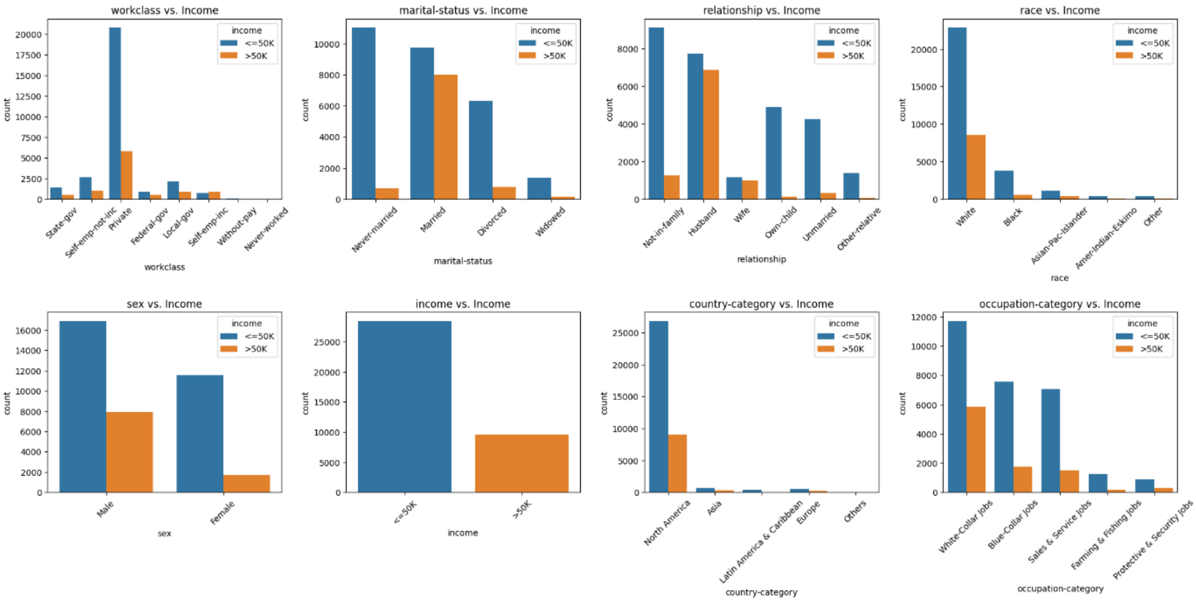
Dataset Information ('df_with_categories')

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 42204 entries, 0 to 48841
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   age                42204 non-null  int64
 1   workclass          42204 non-null  object
 2   education-num      42204 non-null  int64
 3   marital-status     42204 non-null  object
 4   relationship       42204 non-null  object
 5   race               42204 non-null  object
 6   sex                42204 non-null  object
 7   capital-gain       42204 non-null  int64
 8   capital-loss       42204 non-null  int64
 9   hours-per-week     42204 non-null  int64
 10  income             42204 non-null  object
 11  country-category   42204 non-null  object
 12  occupation-category  42204 non-null  object
dtypes: int64(5), object(8)
memory usage: 4.5+ MB
```

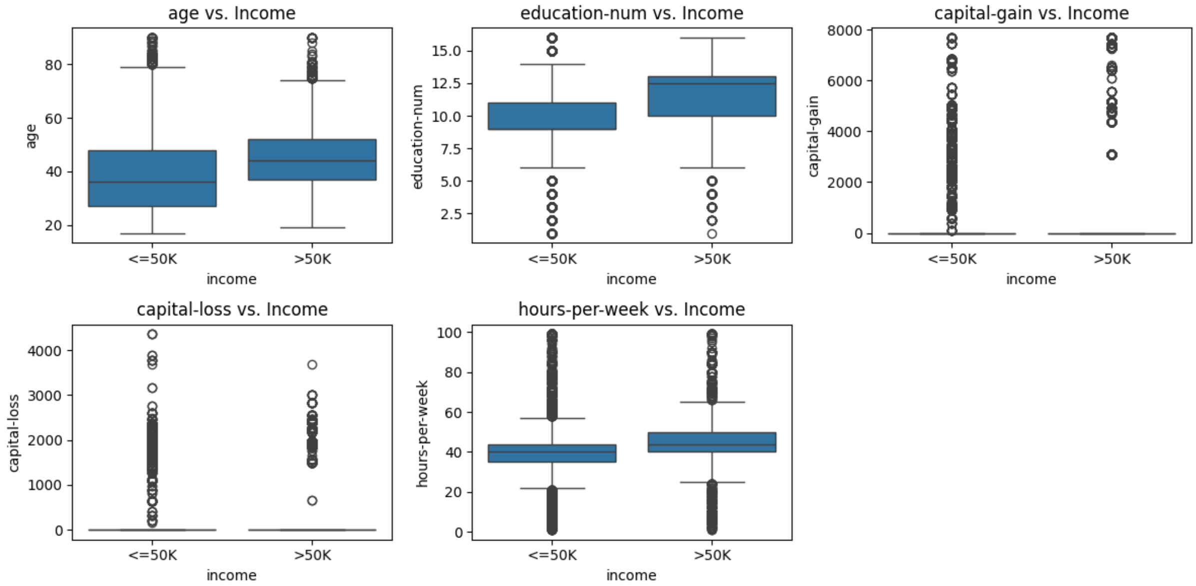
8. Remove duplicates of the ‘df_with_categories’ dataframe

```
[47] df_with_categories.shape

     (42204, 13)

[48] # drop duplicates
     df_with_categories = df_with_categories.drop_duplicates()

     # find duplicate values in the dataset
     print (df_with_categories[df_with_categories.duplicated()])

     Empty DataFrame
     Columns: [age, workclass, education-num, marital-status, relationship, race, sex, c
     Index: []

[49] df_with_categories.shape

     (38081, 13)
```

Visualizing relationship between Categorical features and target (income) for ‘df_with_categories’ Dataframe



Visualizing relationship between Numerical features and target (income) for 'df_with_categories' Dataframe



Feature Engineering

1. Normalize (Scale) the Numerical Data for both dataframes

To ensure numerical features have a similar scale, preventing bias towards features with larger ranges.

```
# Normalize (Scale) the Numerical Data for both dataframes
scaler = StandardScaler()

# Apply scaling to numerical features in both DataFrames
numerical_features = ['age', 'education-num', 'capital-gain',
'capital-loss', 'hours-per-week']

df[numerical_features] =
scaler.fit_transform(df[numerical_features])
df_with_categories[numerical_features] =
scaler.fit_transform(df_with_categories[numerical_features])
```

```
df.head()
```

	age	workclass	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	-0.037032	State-gov	1.093692	Never-married	Adm-clerical	Not-in-family	White	Male	0.930571	-0.233502	-0.051285	United-States	<=50K
1	0.761285	Self-emp-not-inc	1.093692	Married-civ-spouse	Exec-managerial	Husband	White	Male	-0.295782	-0.233502	-2.146279	United-States	<=50K
2	-0.109606	Private	-0.409648	Divorced	Handlers-cleaners	Not-in-family	White	Male	-0.295782	-0.233502	-0.051285	United-States	<=50K
3	0.979008	Private	-1.161318	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	-0.295782	-0.233502	-0.051285	United-States	<=50K
4	-0.835349	Private	1.093692	Married-civ-spouse	Prof-specialty	Wife	Black	Female	-0.295782	-0.233502	-0.051285	Cuba	<=50K

```
df_with_categories.head()
```

	age	workclass	education-num	marital-status	relationship	race	sex	capital-gain	capital-loss	hours-per-week	income	country-category	occupation-category
0	-0.071604	State-gov	1.062029	Never-married	Not-in-family	White	Male	0.873090	-0.245533	-0.056683	<=50K	North America	White-Collar Jobs
1	0.720133	Self-emp-not-inc	1.062029	Married	Husband	White	Male	-0.309687	-0.245533	-2.080265	<=50K	North America	White-Collar Jobs
2	-0.143580	Private	-0.411389	Divorced	Not-in-family	White	Male	-0.309687	-0.245533	-0.056683	<=50K	North America	Blue-Collar Jobs
3	0.936062	Private	-1.148099	Married	Husband	Black	Male	-0.309687	-0.245533	-0.056683	<=50K	North America	Blue-Collar Jobs
4	-0.863341	Private	1.062029	Married	Wife	Black	Female	-0.309687	-0.245533	-0.056683	<=50K	North America	White-Collar Jobs

2. Label Encoding for Categorical Features

To convert categorical features into numerical representations for model compatibility.

```
# Label Encoding for Categorical Features in both dataframes
le = LabelEncoder()

# Apply encoding to categorical features in both DataFrames
categorical_features1 = ['workclass', 'marital-status',
'occupation', 'relationship', 'race', 'sex', 'native-country']
categorical_features2 = ['workclass', 'marital-status',
'occupation-category', 'relationship', 'race', 'sex',
'country-category']

for feature in categorical_features1:
    df[feature] = le.fit_transform(df[feature])
for feature in categorical_features2:
    df_with_categories[feature] =
le.fit_transform(df_with_categories[feature])
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 42204 entries, 0 to 48841
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             42204 non-null  float64
 1   workclass       42204 non-null  int64
 2   education-num   42204 non-null  float64
 3   marital-status  42204 non-null  int64
 4   occupation      42204 non-null  int64
 5   relationship    42204 non-null  int64
 6   race            42204 non-null  int64
 7   sex             42204 non-null  int64
 8   capital-gain    42204 non-null  float64
 9   capital-loss    42204 non-null  float64
 10  hours-per-week  42204 non-null  float64
 11  native-country  42204 non-null  int64
 12  income          42204 non-null  object
dtypes: float64(5), int64(7), object(1)
memory usage: 4.5+ MB
```

The ‘income’ column shouldn't be an object type for machine learning modelling. In income prediction, we typically want the income to be a numerical value (e.g., integer or float).

Because most machine learning models for classification tasks expect numerical features and target variables. Representing income as an object type (text) makes it incompatible with these models.

```
# Create a new binary target variable (0 for <=50K, 1 for >50K)
df['income-binary'] = df['income'].map({'<=50K': 0, '>50K': 1})
df_with_categories['income-binary'] =
df_with_categories['income'].map({'<=50K': 0, '>50K': 1})

df.drop(columns=['income'], inplace=True)
df_with_categories.drop(columns=['income'], inplace=True)
```

3. Handle Imbalance ‘income’ column

```
# Check class distribution for df
income_counts = df['income-binary'].value_counts()
print(income_counts)
```

```
0    31838
1    10366
Name: income-binary, dtype: int64
```

```
# Check class distribution for df_with_categories
income_counts_categories = df_with_categories['income-binary'].value_counts()
print(income_counts_categories)
```

```
0    28493
1     9588
Name: income-binary, dtype: int64
```

Both ‘df’ and ‘df_with_categories’ seem to be imbalanced datasets.

Oversampling might introduce bias with duplicated data, while Undersampling discards information. SMOTE offers a balance. When Considering the trade-offs between complexity and effectiveness, SMOTE might be slightly more complex to implement compared to oversampling but can be more effective.

```
smote = SMOTE()

# Apply SMOTE to both DataFrames
df, df['income-binary'] = smote.fit_resample(df, df['income-
binary'])

df_with_categories, df_with_categories['income-binary'] =
smote.fit_resample(df_with_categories,
df_with_categories['income-binary'])
```

This technique creates synthetic data points for the minority class based on existing data points, increasing its representation without simply copying existing samples

```
df['income-binary'].value_counts()
```

```
0    31838
1    31838
Name: income-binary, dtype: int64
```

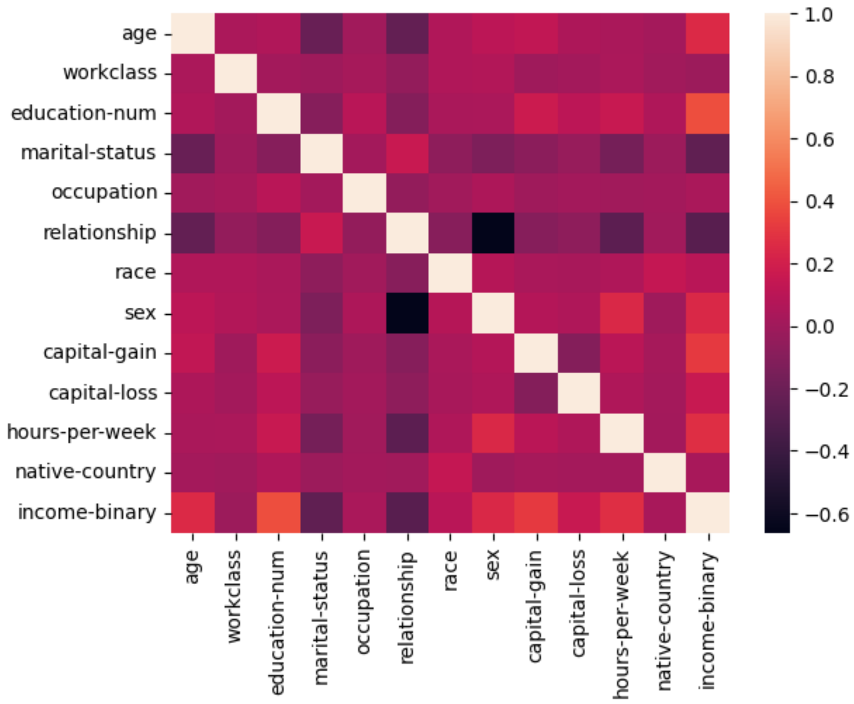
```
df_with_categories['income-binary'].value_counts()
```

```
0    28493
1    28493
Name: income-binary, dtype: int64
```

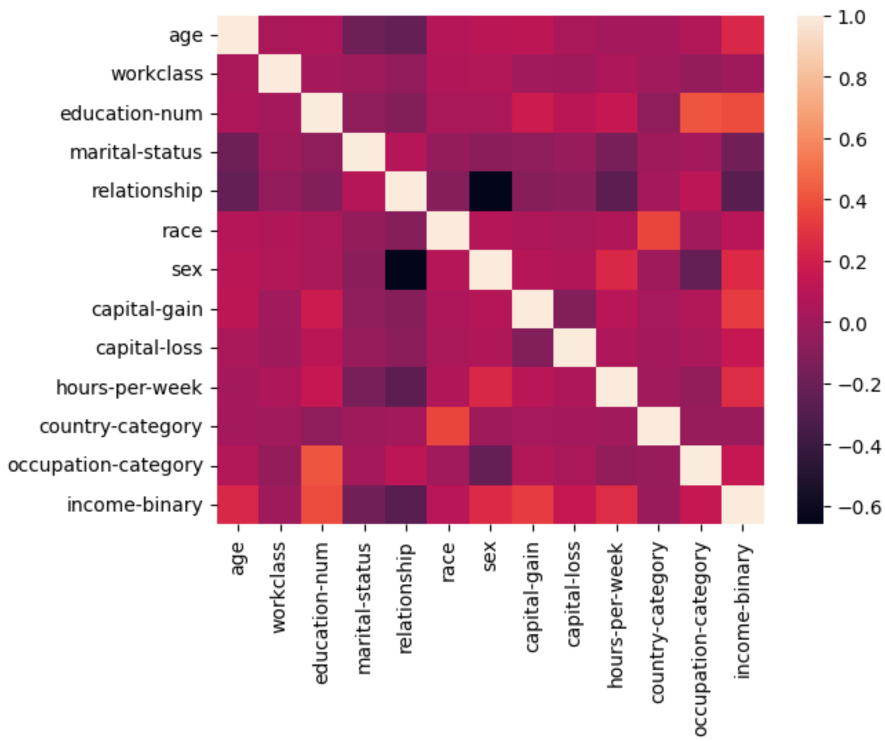
4. Correlation Analysis

Correlation analysis through the heatmap allows us to pinpoint which features are most relevant for predicting income levels. Darker shades on the heatmap indicate stronger correlations.

Heatmap for 'df'



Heatmap for 'df_with_categories'



Evaluation criteria

This is a classification problem. So appropriate evaluation metrics are like accuracy, precision, recall, F1-score, ROC-AUC, etc.

Metric	Value
TP	The number of positive instances that were correctly classified as positive by the model.
TN	The number of negative instances that were correctly classified as negative by the model.
FP	The number of negative instances that were incorrectly classified as positive by the model.
FN	The number of positive instances that were incorrectly classified as negative by the model.

- Accuracy measures the overall correctness of predictions

  Accuracy = TP + TN / TP + TN + FP + FN
- Precision measures the proportion of true positive predictions among all positive predictions

  Precision = TP / TP+FP
- Recall measures the proportion of true positives correctly identified

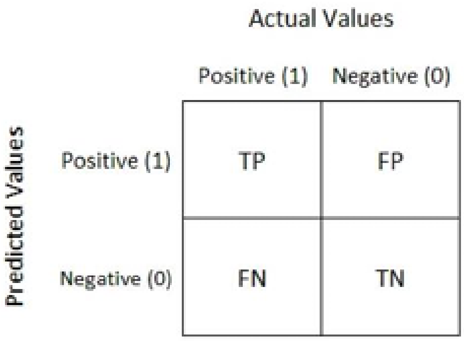
  Recall = TP / TP + FN
- F1-score is the harmonic mean of precision and recall.

  F1-score = 2 × Precision × Recall / Precision + Recall

These metrics help us understand how well our models are performing and which areas need improvement.

Confusion Matrix

A confusion matrix is a table used in classification to evaluate the performance of a classification model. It summarizes the predictions of a model on a classification problem and compares them to the actual ground truth labels. The confusion matrix consists of four sections: True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). These metrics help in assessing the performance of the model, especially in terms of identifying misclassifications.



Splitting the Dataset

In this step, the dataset is divided into two parts: training data and testing data. The training data is used to train the machine learning models, while the testing data is kept separate and used to evaluate the models' performance.

For DataFrame df:

```
X_train, X_test, y_train, y_test = X_train, X_test, y_train, y_test =
train_test_split(df.drop(columns=['income-binary']), df['income-
binary'], test_size=0.2, random_state=42)
```

For DataFrame df_with_categories:

```
X_train_cat, X_test_cat, y_train_cat, y_test_cat =
train_test_split(df_with_categories.drop(columns=['income-binary']),
df_with_categories['income-binary'], test_size=0.2, random_state=42)
```

Model Training

In this step, machine learning models (Naïve Bayes and Random Forest Classification) are trained using the training data. The models learn patterns and relationships in the data that enable them to make predictions.

For DataFrame df:

```
# Naïve Bayes
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)

# Random Forest Classification
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)
```

For DataFrame df_with_categories:

```
# Naïve Bayes
nb_model_cat = GaussianNB()
nb_model_cat.fit(X_train_cat, y_train_cat)

# Random Forest Classification
rf_model_cat = RandomForestClassifier(random_state=42)
rf_model_cat.fit(X_train_cat, y_train_cat)
```

Model Evaluation

This chapter provides a comprehensive analysis of the performance of two machine learning models, Naïve Bayes and Random Forest Classification, based on the outcomes obtained from the experiments conducted using two different datasets.

For DataFrame df

```
Naïve Bayes Model Accuracy: 0.7559116976981696
Naïve Bayes Classification Report:
              precision    recall  f1-score   support

           0       0.71      0.85      0.78      6324
           1       0.82      0.67      0.73      6405

    accuracy                           0.76     12729
   macro avg       0.76      0.76      0.75     12729
weighted avg       0.77      0.76      0.75     12729

Naïve Bayes Confusion Matrix:
 [[5358  966]
 [2141 4264]]
```

Achieved an accuracy of 75.59%. It demonstrated decent precision and recall for both classes, with a slightly higher precision for class 1 (income >50K). The confusion matrix revealed a good balance between true positives and true negatives.

```
Random Forest Classification Model Accuracy: 0.8776808861654489
Random Forest Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.87      0.88      6324
           1       0.88      0.88      0.88      6405

    accuracy                           0.88     12729
   macro avg       0.88      0.88      0.88     12729
weighted avg       0.88      0.88      0.88     12729

Random Forest Confusion Matrix:
 [[5520  804]
 [ 753 5652]]
```

Random Forest did better with an accuracy of 87.77%. It gave good results overall, with high precision, recall, and F1-score for both income categories. The confusion matrix showed that it could correctly predict incomes for different groups, which means it worked well for the task.

For DataFrame df_with_categories

```
Naïve Bayes Model Accuracy (with categories): 0.7506365791553253
Naïve Bayes Classification Report (with categories):
              precision    recall  f1-score   support

           0       0.71      0.86      0.77      5681
           1       0.82      0.64      0.72      5708

    accuracy                           0.75     11389
   macro avg       0.76      0.75      0.75     11389
weighted avg       0.76      0.75      0.75     11389

Naïve Bayes Confusion Matrix (with categories):
 [[4869  812]
 [2028 3680]]
```

The model showed an accuracy of 75.06%. While precision and recall were decent for both classes, there was a slight decrease in performance compared to the non-categorized dataset. The confusion matrix depicted a fair distribution of true positives and true negatives.

```
Random Forest Classification Model Accuracy (with categories): 0.8619720783211872
Random Forest Classification Report (with categories):
              precision    recall  f1-score   support

           0       0.86      0.86      0.86      5681
           1       0.86      0.87      0.86      5708

    accuracy                           0.86     11389
   macro avg       0.86      0.86      0.86     11389
weighted avg       0.86      0.86      0.86     11389

Random Forest Confusion Matrix (with categories):
 [[4869  812]
 [ 760 4948]]
```

Maintained strong performance with an accuracy of 86.20%. Similar to Naïve Bayes, there was a slight decline in precision and recall compared to the non-categorized dataset. However, the confusion matrix showed a well-balanced distribution of true positives and true negatives.

Overall, both models demonstrated good performance in predicting income levels based on census data. However, Random Forest Classification consistently outperformed Naïve Bayes in terms of accuracy and classification metrics for both datasets.

Experimental Results

After conducting experiments with two different datasets (df and df_with_categories), we have gained valuable insights into the performance of Naïve Bayes and Random Forest Classification models for predicting income levels based on census data.

Results without categorize 'native-country', 'marital-status' and **'occupation' Columns**

```
Naïve Bayes Model Accuracy: 0.7559116976981696
Naïve Bayes Classification Report:
               precision    recall  f1-score   support

           0       0.71      0.85      0.78      6324
           1       0.82      0.67      0.73      6405

    accuracy                           0.76     12729
   macro avg       0.76      0.76      0.75     12729
weighted avg       0.77      0.76      0.75     12729

Naïve Bayes Confusion Matrix:
 [[5358  966]
 [2141 4264]]

Random Forest Classification Model Accuracy: 0.8776808861654489
Random Forest Classification Report:
               precision    recall  f1-score   support

           0       0.88      0.87      0.88      6324
           1       0.88      0.88      0.88      6405

    accuracy                           0.88     12729
   macro avg       0.88      0.88      0.88     12729
weighted avg       0.88      0.88      0.88     12729

Random Forest Confusion Matrix:
 [[5520  804]
 [ 753 5652]]
```

Both Naïve Bayes and Random Forest Classification models demonstrated strong performance. Random Forest Classification outperformed Naïve Bayes with a higher accuracy of 87.77% compared to 75.59%. The precision, recall, and F1-score metrics also favored Random Forest Classification, indicating its superior ability to classify individuals into income categories. The confusion matrices further illustrated the effectiveness of Random Forest Classification in correctly predicting both classes.

Result when categorize 'native-country', 'marital-status' and 'occupation' Columns

```
Naïve Bayes Model Accuracy (with categories): 0.7506365791553253
Naïve Bayes Classification Report (with categories):
              precision    recall  f1-score   support

           0       0.71      0.86      0.77      5681
           1       0.82      0.64      0.72      5708

    accuracy                           0.75     11389
   macro avg       0.76      0.75      0.75     11389
weighted avg       0.76      0.75      0.75     11389

Naïve Bayes Confusion Matrix (with categories):
 [[4869  812]
 [2028 3680]]

Random Forest Classification Model Accuracy (with categories): 0.8619720783211872
Random Forest Classification Report (with categories):
              precision    recall  f1-score   support

           0       0.86      0.86      0.86      5681
           1       0.86      0.87      0.86      5708

    accuracy                           0.86     11389
   macro avg       0.86      0.86      0.86     11389
weighted avg       0.86      0.86      0.86     11389

Random Forest Confusion Matrix (with categories):
 [[4869  812]
 [ 760 4948]]
```

This section presents the performance metrics after categorizing the 'native-country', 'marital-status', and 'occupation' columns. The models' performance slightly decreased compared to df. However, both models still achieved reasonable accuracies. Random Forest Classification maintained its superiority with an accuracy of 86.20%, while Naïve Bayes yielded an accuracy of 75.06%. The precision-recall trade-off was evident, particularly in Naïve Bayes, where the recall for the '>50K' class decreased due to the categorization. Nevertheless, both models exhibited robust performance in classifying income levels, as indicated by their respective confusion matrices.

This experiments demonstrate the effectiveness of Random Forest Classification in predicting income levels based on census data. While both Naïve Bayes and Random Forest models performed well, Random Forest exhibited superior performance in both datasets. The results underscore the importance of feature engineering, as categorizing certain columns can provide valuable insights and potentially improve model performance.

Limitations

While our experiments yielded promising results, there are several limitations to consider,

- Limited Feature Engineering

Our feature engineering process focused on categorizing specific columns, such as 'native-country', ‘marital-status’, and 'occupation'. However, this approach may not capture all relevant information present in the dataset. Further exploration of feature engineering techniques, such as creating interaction terms or deriving new features, could enhance model performance.

- Data Quality

Balancing the income column does not address potential issues with data quality, such as missing values, outliers, or noise. It's essential to thoroughly preprocess the data to ensure its quality and reliability for model training.

Possible Enhancements

To address the limitations mentioned above, several enhancements could be considered,

- Advanced Feature Engineering

Exploring more advanced feature engineering techniques, such as principal component analysis (PCA) or feature selection algorithms, could help extract more informative features from the dataset. This could lead to better model performance and more accurate predictions.

- Balancing All Columns

Apply techniques such as SMOTE or other oversampling and undersampling methods to balance not only the income column but also all other columns in the dataset. This can help ensure that the models are trained on a more representative and unbiased dataset, leading to better generalization and predictive performance.

Appendix

```
# load dataset
pip install ucimlrepo

from ucimlrepo import fetch_ucirepo

# fetch dataset
adult = fetch_ucirepo(id=2)

# data (as pandas dataframes)
X = adult.data.features
y = adult.data.targets

# metadata
print("Meta Data\n",adult.metadata)

# variable information
print("Variables\n",adult.variables)

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE # to balance the income
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import LabelEncoder
from imblearn.combine import SMOTEENN

df = pd.concat([X, pd.DataFrame(y, columns=['income'])],axis=1)

df.info()
df.head()
df.sample(5)
df.tail()
df.shape
df.describe()
df['income'].value_counts()

# Data preprocessing
df['income'].replace({'<=50K.':'<=50K', '>50K.': '>50K'}, inplace =
True)
df['income'].value_counts()
```

```
df.replace({'?': np.nan, ' ?': np.nan, '? ': np.nan, ' ? ': np.nan},
inplace = True)

print("Missing Values ")
df.isna().sum()
# replacing NaN values with the mode of respective columns
for column in df.columns:
  mode_value = df[column].mode()[0]
  df[column].fillna(mode_value, inplace = True)

print("Missing Values after replacing NaN values with the mode")
print (df.isnull().sum())
# find duplicate values in the dataset
print (df[df.duplicated()])

# drop duplicates
df = df.drop_duplicates()
# find duplicate values in the dataset
print (df[df.duplicated()])

# Visualize distribution of target variable (Income)
plt.figure(figsize=(6, 4))
sns.countplot(x='income', data=df)
plt.title('Distribution of Income')
plt.show()

numeric_features =
df.select_dtypes(include=[np.number]).columns.tolist()
df[numeric_features].hist(figsize=(12, 10), bins=20)
plt.suptitle('Distribution of Numerical Features')
plt.show()

# Visualize relationship between numerical features and target (Income)
plt.figure(figsize=(12, 6))
numeric_features =
df.select_dtypes(include=[np.number]).columns.tolist()
for i, feature in enumerate(numeric_features):
    plt.subplot(2, 3, i+1)
    sns.boxplot(x='income', y=feature, data=df)
    plt.title(f'{feature} vs. Income')
plt.tight_layout()
plt.show()

# Create box plot for capital-gain grouped by income
plt.figure(figsize=(8, 6))
sns.boxplot(x='income', y='capital-gain', data=df)
plt.title('Capital Gain vs. Income')
```

```
plt.show()

# Handle outlier of the capital-gain
print("Before Clipping:")
print(df['capital-gain'].describe())

# Define lower and upper bounds for clipping
lower_bound = df['capital-gain'].quantile(0.5)
upper_bound = df['capital-gain'].quantile(0.97)

# Clip the values of 'capital-gain' column
df['capital-gain'] = df['capital-gain'].clip(lower=lower_bound,
upper=upper_bound)

# Verify the changes
print("\nAfter Clipping:")
print(df['capital-gain'].describe())

plt.figure(figsize=(20, 18))
for i, feature in
enumerate(df.select_dtypes(include=['object']).columns.tolist()):
    plt.subplot(4, 4, i+1)
    sns.countplot(x=feature, hue='income', data=df)
    plt.title(f'{feature} vs. Income')
    plt.xticks(rotation=45)
plt.tight_layout()
plt.title('Relationship between categorical features and target
(income)')
plt.show()

df.drop(columns=['fnlwgt', 'education'], inplace=True)

# drop duplicates
df = df.drop_duplicates()
# find duplicate values in the dataset
print (df[df.duplicated()])

# categorizing
df_with_categories = df.copy()
df_with_categories.drop(columns=['native-country', 'occupation'],
inplace=True)

# Define mapping for broader country categories
country_mapping = {
    'United-States': 'North America',
    'Mexico': 'North America',
    'Canada': 'North America',
    'Puerto-Rico': 'North America',
```

```
    'El-Salvador': 'North America',
    'Dominican-Republic': 'North America',
    'Jamaica': 'North America',
    'Cuba': 'North America',
    'Guatemala': 'North America',
    'Honduras': 'North America',
    'Philippines': 'Asia',
    'India': 'Asia',
    'China': 'Asia',
    'Japan': 'Asia',
    'Vietnam': 'Asia',
    'Taiwan': 'Asia',
    'Iran': 'Asia',
    'Thailand': 'Asia',
    'Hong': 'Asia',
    'Cambodia': 'Asia',
    'Laos': 'Asia',
    'Germany': 'Europe',
    'England': 'Europe',
    'Italy': 'Europe',
    'Poland': 'Europe',
    'Greece': 'Europe',
    'Portugal': 'Europe',
    'France': 'Europe',
    'Ireland': 'Europe',
    'Scotland': 'Europe',
    'Hungary': 'Europe',
    'Yugoslavia': 'Europe',
    'South': 'Latin America & Caribbean',
    'Columbia': 'Latin America & Caribbean',
    'Haiti': 'Latin America & Caribbean',
    'Peru': 'Latin America & Caribbean',
    'Ecuador': 'Latin America & Caribbean',
    'Trinadad&Tobago': 'Latin America & Caribbean',
    'Nicaragua': 'Latin America & Caribbean',
    'Outlying-US(Guam-USVI-etc)': 'Others',
    'Holand-Netherlands': 'Others',
}

# Replace specific country names with broader category names
df_with_categories['country-category'] = df['native-
country'].replace(country_mapping)
# Verify the changes
print(df_with_categories['country-category'].value_counts())

# Define mapping for broader occupation categories
occupation_mapping = {
    'Prof-specialty': 'White-Collar Jobs',
```

```
    'Exec-managerial': 'White-Collar Jobs',
    'Adm-clerical': 'White-Collar Jobs',
    'Tech-support': 'White-Collar Jobs',
    'Craft-repair': 'Blue-Collar Jobs',
    'Machine-op-inspct': 'Blue-Collar Jobs',
    'Transport-moving': 'Blue-Collar Jobs',
    'Handlers-cleaners': 'Blue-Collar Jobs',
    'Sales': 'Sales & Service Jobs',
    'Other-service': 'Sales & Service Jobs',
    'Protective-serv': 'Protective & Security Jobs',
    'Priv-house-serv': 'Protective & Security Jobs',
    'Armed-Forces': 'Protective & Security Jobs',
    'Farming-fishing': 'Farming & Fishing Jobs'
}

# Replace specific occupation names with broader category names
df_with_categories['occupation-category'] =
df['occupation'].replace(occupation_mapping)
# Verify the changes
print(df_with_categories['occupation-category'].value_counts())

# Mapping dictionary for categorizing marital status
marital_status_mapping = {
    'Married-civ-spouse': 'Married',
    'Married-AF-spouse': 'Married',
    'Married-spouse-absent': 'Married',
    'Never-married': 'Never-married',
    'Widowed': 'Widowed',
    'Divorced': 'Divorced',
    'Separated': 'Divorced'  # Considering 'Separated' as 'Divorced'
}

# Replace marital status with categorized values
df_with_categories['marital-status'] = df['marital-
status'].replace(marital_status_mapping)
# Check the updated counts
print(df_with_categories['marital-status'].value_counts())

# drop duplicates
df_with_categories = df_with_categories.drop_duplicates()
# find duplicate values in the dataset
print (df_with_categories[df_with_categories.duplicated()])

# Feature Engineering
# Normalize (Scale) the Numerical Data for both dataframes
scaler = StandardScaler()
# Apply scaling to numerical features in both DataFrames
```

```
numerical_features = ['age', 'education-num', 'capital-gain', 'capital-
loss', 'hours-per-week']
df[numerical_features] = scaler.fit_transform(df[numerical_features])
df_with_categories[numerical_features] =
scaler.fit_transform(df_with_categories[numerical_features])

# Label Encoding for Categorical Features in both dataframes
le = LabelEncoder()
# Apply encoding to categorical features in both DataFrames
categorical_features1 = ['workclass', 'marital-status', 'occupation',
'relationship', 'race', 'sex', 'native-country']
categorical_features2 = ['workclass', 'marital-status', 'occupation-
category', 'relationship', 'race', 'sex', 'country-category']
for feature in categorical_features1:
    df[feature] = le.fit_transform(df[feature])
for feature in categorical_features2:
    df_with_categories[feature] =
le.fit_transform(df_with_categories[feature])

# Create a new binary target variable (0 for <=50K, 1 for >50K)
df['income-binary'] = df['income'].map({'<=50K': 0, '>50K': 1})
df_with_categories['income-binary'] =
df_with_categories['income'].map({'<=50K': 0, '>50K': 1})

df.drop(columns=['income'], inplace=True)
df_with_categories.drop(columns=['income'], inplace=True)

# Check class distribution for df
income_counts = df['income-binary'].value_counts()
print(income_counts)
# Check class distribution for df_with_categories
income_counts_categories = df_with_categories['income-
binary'].value_counts()
print(income_counts_categories)

# Handle Imbalance ‘income’ column
smote = SMOTE()
# Apply SMOTE to both DataFrames
df, df['income-binary'] = smote.fit_resample(df, df['income-binary'])
df_with_categories, df_with_categories['income-binary'] =
smote.fit_resample(df_with_categories, df_with_categories['income-
binary'])
df['income-binary'].value_counts()
df_with_categories['income-binary'].value_counts()

# drop duplicates
df_with_categories = df_with_categories.drop_duplicates()
df = df.drop_duplicates()
```

```
sns.heatmap(df.corr())
sns.heatmap(df_with_categories.corr())
```

For DataFrame df:

```
# Step 1: Splitting the Dataset
X_train, X_test, y_train, y_test =
train_test_split(df.drop(columns=['income-binary']), df['income-
binary'], test_size=0.2, random_state=42)

# Step 2: Model Training
# Naïve Bayes
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)

# Random Forest Classification
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)

# Step 3: Model Evaluation
# Naïve Bayes
nb_pred = nb_model.predict(X_test)
nb_accuracy = accuracy_score(y_test, nb_pred)
nb_classification_report = classification_report(y_test, nb_pred)
nb_conf_matrix = confusion_matrix(y_test, nb_pred)

# Random Forest Classification
rf_pred = rf_model.predict(X_test)
rf_accuracy = accuracy_score(y_test, rf_pred)
rf_classification_report = classification_report(y_test, rf_pred)
rf_conf_matrix = confusion_matrix(y_test, rf_pred)

# Step 4: Model Comparison
print("Naïve Bayes Model Accuracy:", nb_accuracy)
print("Naïve Bayes Classification Report:\n", nb_classification_report)
print("Naïve Bayes Confusion Matrix:\n", nb_conf_matrix)

print("\nRandom Forest Classification Model Accuracy:", rf_accuracy)
print("Random Forest Classification Report:\n",
rf_classification_report)
print("Random Forest Confusion Matrix:\n", rf_conf_matrix)
```

For DataFrame df_with_categories:

```
# Step 1: Splitting the Dataset
```

```
X_train_cat, X_test_cat, y_train_cat, y_test_cat =
train_test_split(df_with_categories.drop(columns=['income-binary']),
df_with_categories['income-binary'], test_size=0.2, random_state=42)

# Step 2: Model Training
# Naïve Bayes
nb_model_cat = GaussianNB()
nb_model_cat.fit(X_train_cat, y_train_cat)

# Random Forest Classification
rf_model_cat = RandomForestClassifier(random_state=42)
rf_model_cat.fit(X_train_cat, y_train_cat)

# Step 3: Model Evaluation
# Naïve Bayes
nb_pred_cat = nb_model_cat.predict(X_test_cat)
nb_accuracy_cat = accuracy_score(y_test_cat, nb_pred_cat)
nb_classification_report_cat = classification_report(y_test_cat,
nb_pred_cat)
nb_conf_matrix_cat = confusion_matrix(y_test_cat, nb_pred_cat)

# Random Forest Classification
rf_pred_cat = rf_model_cat.predict(X_test_cat)
rf_accuracy_cat = accuracy_score(y_test_cat, rf_pred_cat)
rf_classification_report_cat = classification_report(y_test_cat,
rf_pred_cat)
rf_conf_matrix_cat = confusion_matrix(y_test_cat, rf_pred_cat)

# Step 4: Model Comparison
print("Naïve Bayes Model Accuracy (with categories):", nb_accuracy_cat)
print("Naïve Bayes Classification Report (with categories):\n",
nb_classification_report_cat)
print("Naïve Bayes Confusion Matrix (with categories):\n",
nb_conf_matrix_cat)

print("\nRandom Forest Classification Model Accuracy (with
categories):", rf_accuracy_cat)
print("Random Forest Classification Report (with categories):\n",
rf_classification_report_cat)
print("Random Forest Confusion Matrix (with categories):\n",
rf_conf_matrix_cat)
```