PREDICTING INCOME LEVELS with ML CLASSIFICATION MODELS

Logistic Regression - SVM - KNN

Duygu Jones | Data Scientist | Aug 2024
Follow me: duygujones.com | Linkedin | GitHub | Kaggle | Medium | Tableau

Introduction

The goal of this project is to predict whether an individual's annual income exceeds $50,000 using the "Adult" dataset from the 1994 Census Bureau.

- The performance of four machine learning models—Logistic Regression, K-Nearest Neighbors (KNN), and Support Vector Machine (SVM)—will be developed and compared.
- Data preprocessing, model training, evaluation, and comparison will be conducted to identify the best-performing model for this classification task.

Objectives

1. **Data Preprocessing**: Clean and encode data and handle missing values.
2. **Model Development**: Implement Logistic Regression, K-Nearest Neighbors (KNN),and Support Vector Machine (SVM).
3. **Model Training and Evaluation**: Train models on the training set and evaluate performance using metrics like accuracy, precision, recall, and F1-score.

4. **Model Comparison**: Compare models to identify the best performer.
5. **Conclusion**: Summarized findings and provided recommendations for improvement.

The dataset and results are used for educational purposes, demonstrating the application of advanced machine learning techniques on real-world data. We aim to build effective machine learning models to predict adults income and gain a deeper understanding of machine learning techniques.

About the Dataset

The dataset is a commonly used dataset known as the "Adult" dataset or "Census Income" dataset. It is primarily used for machine learning tasks, particularly classification. The goal is often to predict whether an individual earns more than $50,000 a year based on various demographic and employment-related attributes

The dataset is available on UCI Machine Learning Repository

Dataset: Census Adult Income

- **Content**: Data on various demographic and employment-related attributes of individuals.
- **Number of Rows**: 32,561
- **Number of Columns**: 15

INPUTS

No	Feature	Description
1	**age**	Integer value representing the age of the individual.
2	**workclass**	Categorical variable indicating the type of employer (e.g., Private, Self-emp-not-inc, etc.).
3	**fnlwgt**	Continuous variable representing the final weight, which is a proxy for the number of people represented by the individual.
4	**education**	Categorical variable indicating the highest level of education achieved (e.g., Bachelors, HS-grad, etc.).
5	**education.num**	Integer value representing the numerical encoding of education levels.
6	**marital.status**	Categorical variable indicating the marital status of the individual (e.g., Married-civ-spouse, Divorced, etc.).
7	**occupation**	Categorical variable representing the individual's occupation (e.g., Tech-support, Craft-repair, etc.).
8	**relationship**	Categorical variable representing the individual's relationship status within a family (e.g., Wife, Own-child, etc.).
9	**race**	Categorical variable indicating the race of the individual (e.g., White, Black, etc.).
10	**sex**	Categorical variable indicating the gender of the individual (Male or Female).
11	**capital.gain**	Continuous variable representing the capital gains received by the individual.
12	**capital.loss**	Continuous variable representing the capital losses incurred by the individual.
13	**hours.per.week**	Continuous variable indicating the number of hours the individual works per week.

No	Feature	Description
14	**native.country**	Categorical variable representing the country of origin for the individual (e.g., United-States, Mexico, etc.).
15	**income**	Categorical variable indicating the income category of the individual (<=50K or >50K).

The dataset is often used for predictive modeling to understand how different demographic and employment factors relate to income levels. It contains both categorical and continuous variables, making it a versatile dataset for various types of machine learning algorithms.

Details About the Dataset

This dataset was extracted from the 1994 Census Income Bureau database by Ronny Kohavi and Barry Becker. It contains clean records meeting specific criteria, such as age greater than 16 and hours worked per week greater than zero. The main goal is to predict whether an individual earns more than $50K per year.

Description of `fnlwgt` (Final Weight): To understand the dataset's origin, extraction conditions, and the methodology behind the `fnlwgt` feature. The `fnlwgt` feature represents weights controlled to independent estimates of the civilian noninstitutional population of the US, prepared monthly by the Census Bureau's Population Division. The weighting program uses three sets of controls:

1. Population aged 16+ for each state.
2. Hispanic origin by age and sex.
3. Race by age and sex.

These controls are applied multiple times to ensure accuracy. The weights ensure that people with similar demographic characteristics have similar weights, but this is only applicable within each state due to the sampling method.

Relevant Papers

- Ron Kohavi, Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid, Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, 1996.

Table of Contents

EXPLORATORY DATA ANALYSIS (EDA)

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns
import cufflinks as cf
%matplotlib inline

from scipy import stats
from sklearn.model_selection import train_test_split, GridSearchCV, cross_validate, Stratifi
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.preprocessing import PowerTransformer, OneHotEncoder, LabelEncoder
from sklearn.pipeline import Pipeline


from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

from sklearn.metrics import make_scorer, precision_score, recall_score, f1_score, accuracy_s
from sklearn.metrics import PrecisionRecallDisplay, roc_curve, average_precision_score, prec
from sklearn.metrics import RocCurveDisplay, roc_auc_score, auc
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay


from yellowbrick.regressor import ResidualsPlot, PredictionError

import warnings
warnings.filterwarnings("ignore")
```

In [3]:
```
df0 = pd.read_csv("adult.csv")
df = df0.copy()
```

In [4]:
```
df.shape
```

Out[4]: (32561, 15)

In [5]:
```
df.head().T
```

Out[5]:

	0	1	2	3	4
age	90	82	66	54	41
workclass	?	Private	?	Private	Private
fnlwgt	77053	132870	186061	140359	264663
education	HS-grad	HS-grad	Some-college	7th-8th	Some-college
education.num	9	9	10	4	10
marital.status	Widowed	Widowed	Widowed	Divorced	Separated
occupation	?	Exec-managerial	?	Machine-op-inspct	Prof-specialty
relationship	Not-in-family	Not-in-family	Unmarried	Unmarried	Own-child
race	White	White	Black	White	White
sex	Female	Female	Female	Female	Female
capital.gain	0	0	0	0	0
capital.loss	4356	4356	4356	3900	3900
hours.per.week	40	18	40	40	40
native.country	United-States	United-States	United-States	United-States	United-States
income	<=50K	<=50K	<=50K	<=50K	<=50K

In [6]:
```
df.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
```

Basic Statistics

In [7]:
```
# Basic statistics summary of Numerical features

df.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75%	m
age	32561.0	38.581647	13.640433	17.0	28.0	37.0	48.0	9
fnlwgt	32561.0	189778.366512	105549.977697	12285.0	117827.0	178356.0	237051.0	148470
education.num	32561.0	10.080679	2.572720	1.0	9.0	10.0	12.0	1
capital.gain	32561.0	1077.648844	7385.292085	0.0	0.0	0.0	0.0	9999
capital.loss	32561.0	87.303830	402.960219	0.0	0.0	0.0	0.0	435
hours.per.week	32561.0	40.437456	12.347429	1.0	40.0	40.0	45.0	9

In [8]:
```
# Basic statistics summary of Object features

df.describe(include= 'object').T
```

Out[8]:

	count	unique	top	freq
workclass	32561	9	Private	22696
education	32561	16	HS-grad	10501
marital.status	32561	7	Married-civ-spouse	14976
occupation	32561	15	Prof-specialty	4140
relationship	32561	6	Husband	13193
race	32561	5	White	27816
sex	32561	2	Male	21790
native.country	32561	42	United-States	29170
income	32561	2	<=50K	24720

In [9]:
```
# Summary of the Dataset

def summary(df, pred=None):
    obs = df.shape[0]
    Types = df.dtypes
    Counts = df.apply(lambda x: x.count())
    Min = df.min()
    Max = df.max()
    Uniques = df.apply(lambda x: x.unique().shape[0])
    Nulls = df.apply(lambda x: x.isnull().sum())
    print('Data shape:', df.shape)

    if pred is None:
        cols = ['Types', 'Counts', 'Uniques', 'Nulls', 'Min', 'Max']
        str = pd.concat([Types, Counts, Uniques, Nulls, Min, Max], axis = 1, sort=True)
```

```
    str.columns = cols
    print('_____________________________\nData Types:')
    print(str.Types.value_counts())
    print('_____________________________')
    return str

summary(df)
```

```
Data shape: (32561, 15)
_____________________________
Data Types:
Types
object    9
int64     6
Name: count, dtype: int64
_____________________________
```

Out[9]:

	Types	Counts	Uniques	Nulls	Min	Max
age	int64	32561	73	0	17	90
capital.gain	int64	32561	119	0	0	99999
capital.loss	int64	32561	92	0	0	4356
education	object	32561	16	0	10th	Some-college
education.num	int64	32561	16	0	1	16
fnlwgt	int64	32561	21648	0	12285	1484705
hours.per.week	int64	32561	94	0	1	99
income	object	32561	2	0	<=50K	>50K
marital.status	object	32561	7	0	Divorced	Widowed
native.country	object	32561	42	0	?	Yugoslavia
occupation	object	32561	15	0	?	Transport-moving
race	object	32561	5	0	Amer-Indian-Eskimo	White
relationship	object	32561	6	0	Husband	Wife
sex	object	32561	2	0	Female	Male
workclass	object	32561	9	0	?	Without-pay

Duplicated Values

In [10]:

```
df.duplicated().sum()
```

Out[10]: 24

In [11]:

```
# Checks duplicates and drops them

def duplicate_values(df):
    print("Duplicate check...")
    num_duplicates = df.duplicated(subset=None, keep='first').sum()
    if num_duplicates > 0:
        print("There are", num_duplicates, "duplicated observations in the dataset.")
```

```
        df.drop_duplicates(keep='first', inplace=True)
        print(num_duplicates, "duplicates were dropped!")
        print("No more duplicate rows!")
    else:
        print("There are no duplicated observations in the dataset.")

duplicate_values(df)
```

```
Duplicate check...
There are 24 duplicated observations in the dataset.
24 duplicates were dropped!
No more duplicate rows!
```

In [12]:
```
# Let's observe first the unique values

def get_unique_values(df):

    output_data = []

    for col in df.columns:

        # If the number of unique values in the column is less than or equal to 5
        if df.loc[:, col].nunique() <= 10:
            # Get the unique values in the column
            unique_values = df.loc[:, col].unique()
            # Append the column name, number of unique values, unique values, and data type
            output_data.append([col, df.loc[:, col].nunique(), unique_values, df.loc[:, col]
        else:
            # Otherwise, append only the column name, number of unique values, and data type
            output_data.append([col, df.loc[:, col].nunique(),"-", df.loc[:, col].dtype])

    output_df = pd.DataFrame(output_data, columns=['Column Name', 'Number of Unique Values',

    return output_df
```

In [13]:
```
get_unique_values(df)
```

Out[13]:

	Column Name	Number of Unique Values	Unique Values	Data Type
0	age	73	-	int64
1	workclass	9	[?, Private, State-gov, Federal-gov, Self-emp-...	object
2	fnlwgt	21648	-	int64
3	education	16	-	object
4	education.num	16	-	int64
5	marital.status	7	[Widowed, Divorced, Separated, Never-married, ...	object
6	occupation	15	-	object
7	relationship	6	[Not-in-family, Unmarried, Own-child, Other-re...	object
8	race	5	[White, Black, Asian-Pac-Islander, Other, Amer...	object
9	sex	2	[Female, Male]	object
10	capital.gain	119	-	int64
11	capital.loss	92	-	int64
12	hours.per.week	94	-	int64
13	native.country	42	-	object
14	income	2	[<=50K, >50K]	object

Missing Values

In [14]:

```
def missing_values(df):

    missing_count = df.isnull().sum()
    value_count = df.isnull().count()
    missing_percentage = round(missing_count / value_count * 100, 2)
    missing_df = pd.DataFrame({"count": missing_count, "percentage": missing_percentage})
    return missing_df

missing_values(df)
```

Out[14]:

	count	percentage
age	0	0.0
workclass	0	0.0
fnlwgt	0	0.0
education	0	0.0
education.num	0	0.0
marital.status	0	0.0
occupation	0	0.0
relationship	0	0.0
race	0	0.0
sex	0	0.0
capital.gain	0	0.0
capital.loss	0	0.0
hours.per.week	0	0.0
native.country	0	0.0
income	0	0.0

In [15]:

```
# As observed in the count graphics below, the workclass and occupation features contain "?"
# Replace the values with nan

df[df == '?'] = np.nan
```

In [16]:

```
# After replacing '?' symboll to 'nan' value, we can see the missing values now
missing_values(df)
```

Out[16]:

	count	percentage
age	0	0.00
workclass	1836	5.64
fnlwgt	0	0.00
education	0	0.00
education.num	0	0.00
marital.status	0	0.00
occupation	1843	5.66
relationship	0	0.00
race	0	0.00
sex	0	0.00
capital.gain	0	0.00
capital.loss	0	0.00
hours.per.week	0	0.00
native.country	582	1.79
income	0	0.00

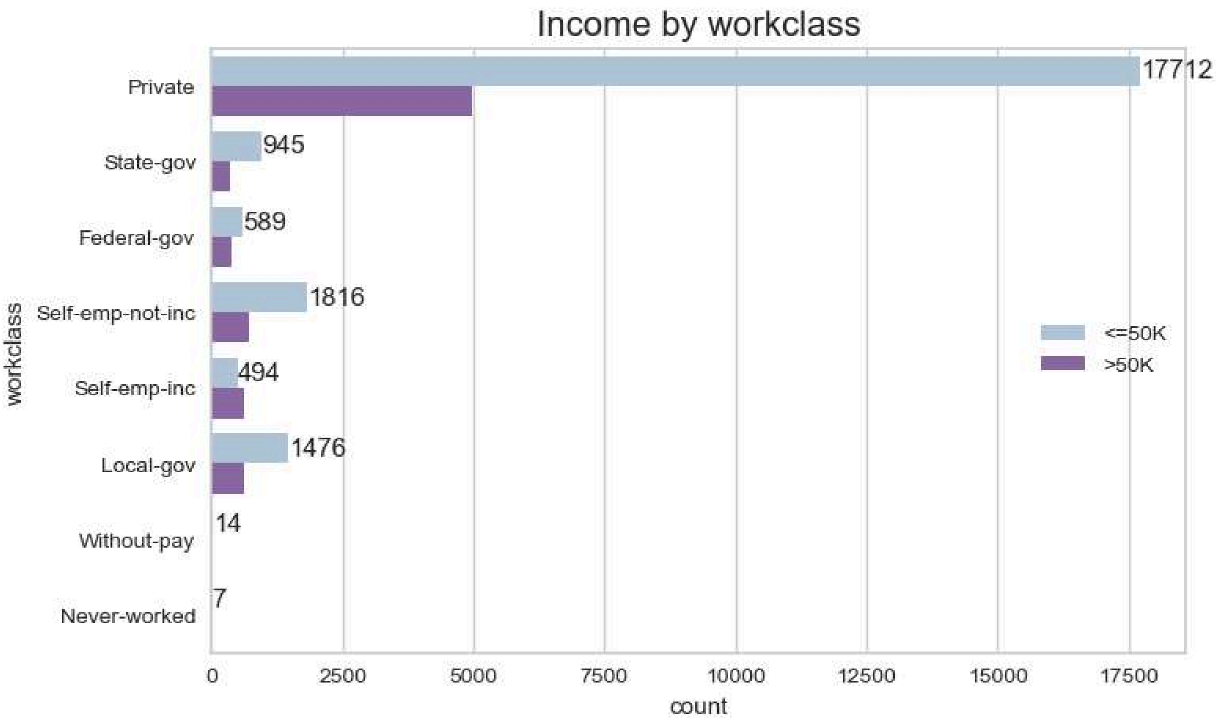
Handle Missing Values on the `workclass` Column

In [17]:
```
df['workclass'].value_counts(normalize=True)
```

Out[17]:
```
workclass
Private             0.738510
Self-emp-not-inc    0.082733
Local-gov           0.068174
State-gov           0.042279
Self-emp-inc        0.036351
Federal-gov         0.031269
Without-pay         0.000456
Never-worked        0.000228
Name: proportion, dtype: float64
```

In [18]:
```
plt.figure(figsize = (8,5))
ax = sns.countplot(y = df['workclass'], hue = df['income'] , palette='BuPu')
plt.title("Income by workclass", fontsize = 16)
ax.bar_label(ax.containers[0]);
ax.legend(loc='center right')
```

Out[18]:
```
<matplotlib.legend.Legend at 0x1b165087cb0>
```



- Distribution of income levels (<=50K and >50K) across different work classes, indicating that the majority of individuals in the 'Private' work class earn <=50K.
- To fill the missing values in the 'workclass' feature, it is generally better to use the mode (most frequent value) because it maintains the distribution and the majority representation in the data. In this case, the mode is 'Private'.

In [19]:
```
df['workclass'] = df['workclass'].fillna('Private')
```

In [20]:
```
##Check missing values
missing_values(df)
```

Out[20]:

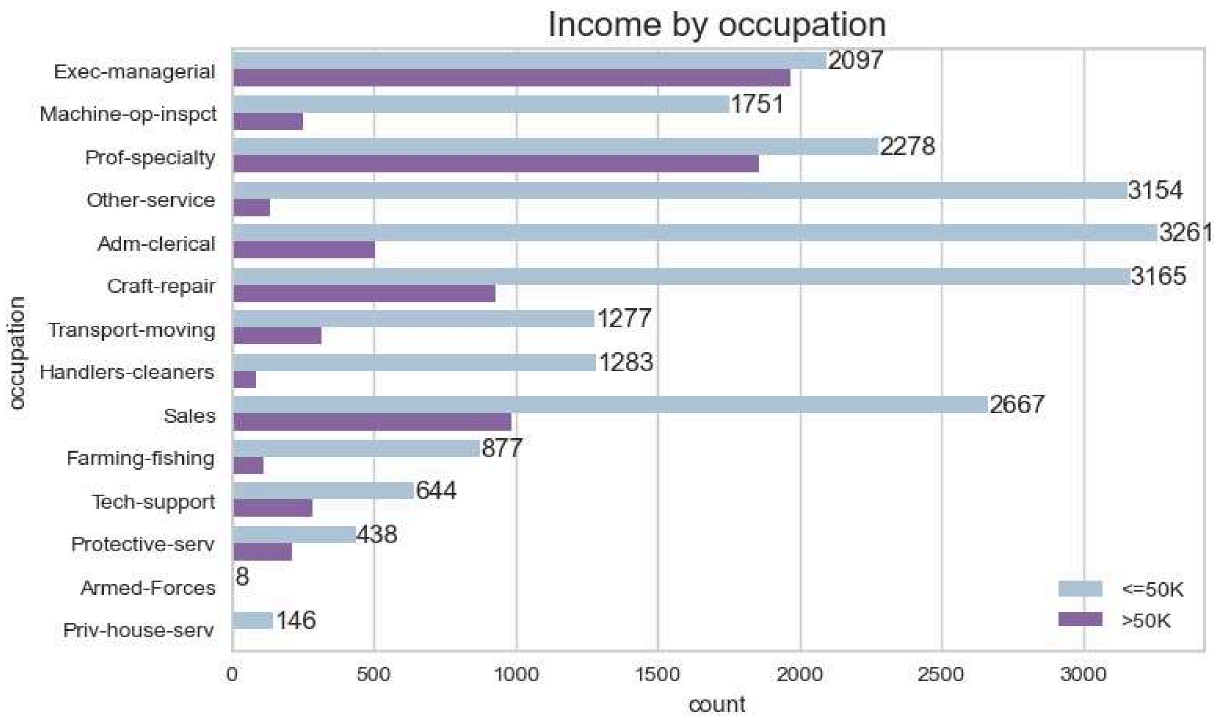
	count	percentage
age	0	0.00
workclass	0	0.00
fnlwgt	0	0.00
education	0	0.00
education.num	0	0.00
marital.status	0	0.00
occupation	1843	5.66
relationship	0	0.00
race	0	0.00
sex	0	0.00
capital.gain	0	0.00
capital.loss	0	0.00
hours.per.week	0	0.00
native.country	582	1.79
income	0	0.00

Handle Missing Values on the `occupation` Column

In [21]:
```
df['occupation'].value_counts(normalize=True)
```

Out[21]:
```
occupation
Prof-specialty       0.134749
Craft-repair         0.133381
Exec-managerial      0.132436
Adm-clerical         0.122760
Sales                0.118916
Other-service        0.107220
Machine-op-inspct    0.065159
Transport-moving     0.052030
Handlers-cleaners    0.044602
Farming-fishing      0.032319
Tech-support         0.030201
Protective-serv      0.021144
Priv-house-serv      0.004789
Armed-Forces         0.000293
Name: proportion, dtype: float64
```

In [22]:
```
plt.figure(figsize = (8,5))
ax = sns.countplot(y = df['occupation'], hue = df['income'], palette='BuPu')
plt.title("Income by occupation", fontsize = 16)
ax.bar_label(ax.containers[0])
ax.legend(loc='lower right')
plt.show()
```



In [23]:
```
df['occupation'] = df['occupation'].fillna(df['occupation'].mode()[0])
```

In [24]:
```
#Check missing values
missing_values(df)
```

Out[24]:

	count	percentage
age	0	0.00
workclass	0	0.00
fnlwgt	0	0.00
education	0	0.00
education.num	0	0.00
marital.status	0	0.00
occupation	0	0.00
relationship	0	0.00
race	0	0.00
sex	0	0.00
capital.gain	0	0.00
capital.loss	0	0.00
hours.per.week	0	0.00
native.country	582	1.79
income	0	0.00

Handle Missing Values on the `native country` Column

In [25]:
```
df['native.country'].value_counts(normalize=True)
```

Out[25]:
```
native.country
United-States                 0.912314
Mexico                        0.019997
Philippines                   0.006196
Germany                       0.004287
Canada                        0.003787
Puerto-Rico                   0.003568
El-Salvador                   0.003317
India                         0.003129
Cuba                          0.002973
England                       0.002816
Jamaica                       0.002535
South                         0.002504
China                         0.002347
Italy                         0.002284
Dominican-Republic            0.002191
Vietnam                       0.002097
Guatemala                     0.001940
Japan                         0.001940
Poland                        0.001878
Columbia                      0.001846
Taiwan                        0.001596
Haiti                         0.001377
Iran                          0.001346
Portugal                      0.001158
Nicaragua                     0.001064
Peru                          0.000970
Greece                        0.000908
France                        0.000908
Ecuador                       0.000876
Ireland                       0.000751
Hong                          0.000626
Trinadad&Tobago               0.000595
Cambodia                      0.000595
Thailand                      0.000563
Laos                          0.000563
Yugoslavia                    0.000501
Outlying-US(Guam-USVI-etc)    0.000438
Hungary                       0.000407
Honduras                      0.000407
Scotland                      0.000376
Holand-Netherlands            0.000031
Name: proportion, dtype: float64
```

In [26]:
```
df['native.country'].mode()[0]
```

Out[26]:
```
'United-States'
```

In [27]:
```
# Filling any missing values (NaN) in the native.country column with "United-States" which i
df['native.country'] = df['native.country'].fillna('United-States')
```

In [28]:
```
#Check missing values
missing_values(df)
```

Out[28]:

	count	percentage
age	0	0.0
workclass	0	0.0
fnlwgt	0	0.0
education	0	0.0
education.num	0	0.0
marital.status	0	0.0
occupation	0	0.0
relationship	0	0.0
race	0	0.0
sex	0	0.0
capital.gain	0	0.0
capital.loss	0	0.0
hours.per.week	0	0.0
native.country	0	0.0
income	0	0.0

Cleaning and Preparing Each Column

In [29]:

```
# Fonction for counting and normalizing values in the column

def value_cnt_fonc(df, column_name):
    vc = df[column_name].value_counts()
    vc_norm = df[column_name].value_counts(normalize=True)

    vc = vc.rename_axis(column_name).reset_index(name='counts')
    vc_norm = vc_norm.rename_axis(column_name).reset_index(name='norm_counts')

    df_result = pd.concat([vc[column_name], vc['counts'], vc_norm['norm_counts']], axis=1)

    return df_result
```

In [30]:

```
# Categorcal and Numerecal Features

cat_features = df.select_dtypes(include='object').columns
num_features = df.select_dtypes(include=['int64','float64']).columns

print('Categoricals:', list(cat_features))
print('---------------------------')
print('Numericals:',list(num_features))
```

```
Categoricals: ['workclass', 'education', 'marital.status', 'occupation', 'relationship', 'rac
e', 'sex', 'native.country', 'income']
---------------------------
Numericals: ['age', 'fnlwgt', 'education.num', 'capital.gain', 'capital.loss', 'hours.per.wee
k']
```

Target Feature `income`

In [31]:
```
value_cnt_fonc(df, 'income')
```

Out[31]:

	income	counts	norm_counts
0	<=50K	24698	0.759074
1	>50K	7839	0.240926

In [32]:
```
# Convert income values to binary: 0 for <=50K, 1 for >50K

df['income'] = df['income'].map({'<=50K': 0, '>50K': 1})
```

In [33]:
```
df.sample(3)
```

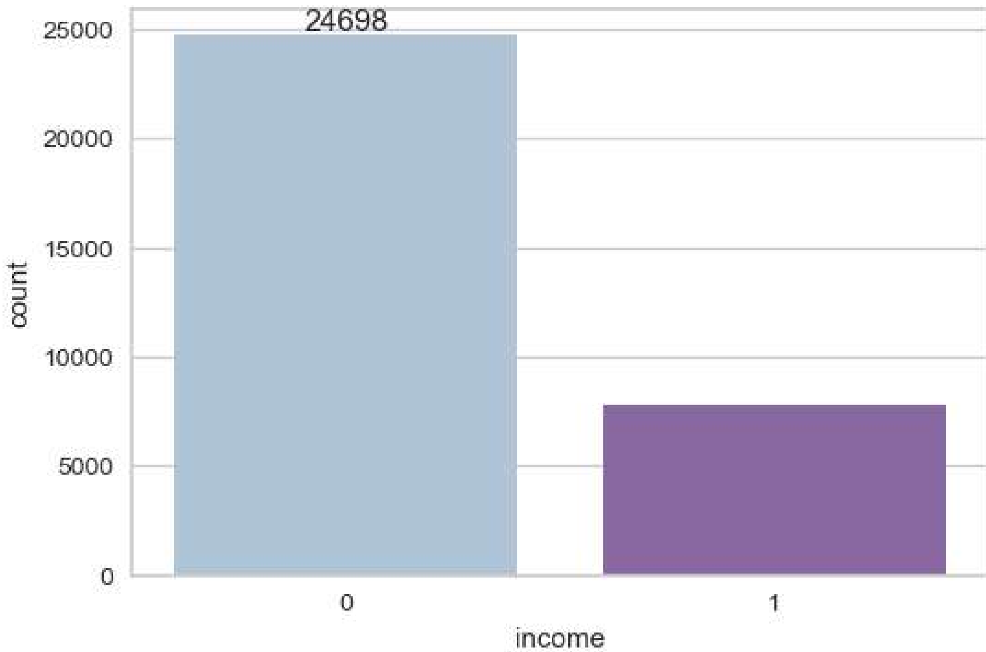
Out[33]:

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationship
11273	29	Private	132686	HS-grad	9	Divorced	Machine-op-inspct	Unmarried
113	72	Self-emp-not-inc	52138	Doctorate	16	Married-civ-spouse	Prof-specialty	Husband
28828	33	Private	182246	HS-grad	9	Never-married	Prof-specialty	Not-in-family

In [34]:
```
income_less_50K = df[df['income'] == 0].shape[0]
income_over_50K = df[df['income'] == 1].shape[0]

print(f"Income <= 50K (0) count: {income_less_50K}")
print(f"Income >  50K (1) count: {income_over_50K}")
```

```
Income <= 50K (0) count: 24698
Income >  50K (1) count: 7839
```

In [35]:
```
plt.figure(figsize=(6,4))
ax = sns.countplot( data=df, x="income", palette='BuPu')

ax.bar_label(ax.containers[0])
plt.show()
```



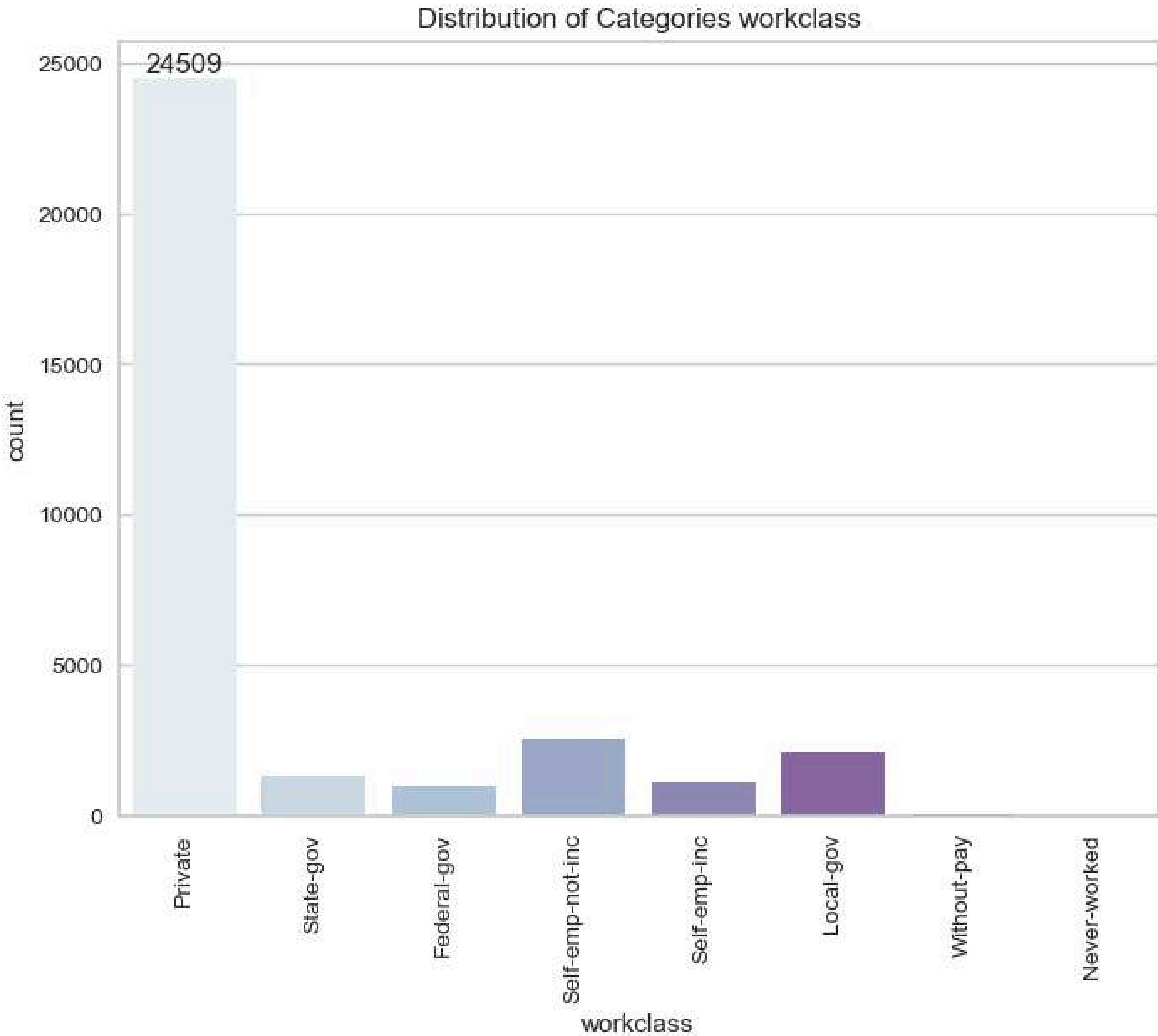
- Graphic clearly indicates that there are significantly more individuals in the <=50K income group, while the >50K group has considerably fewer individuals.
- The imbalance between the two income groups is evident, highlighting a noticeable disparity in the dataset.

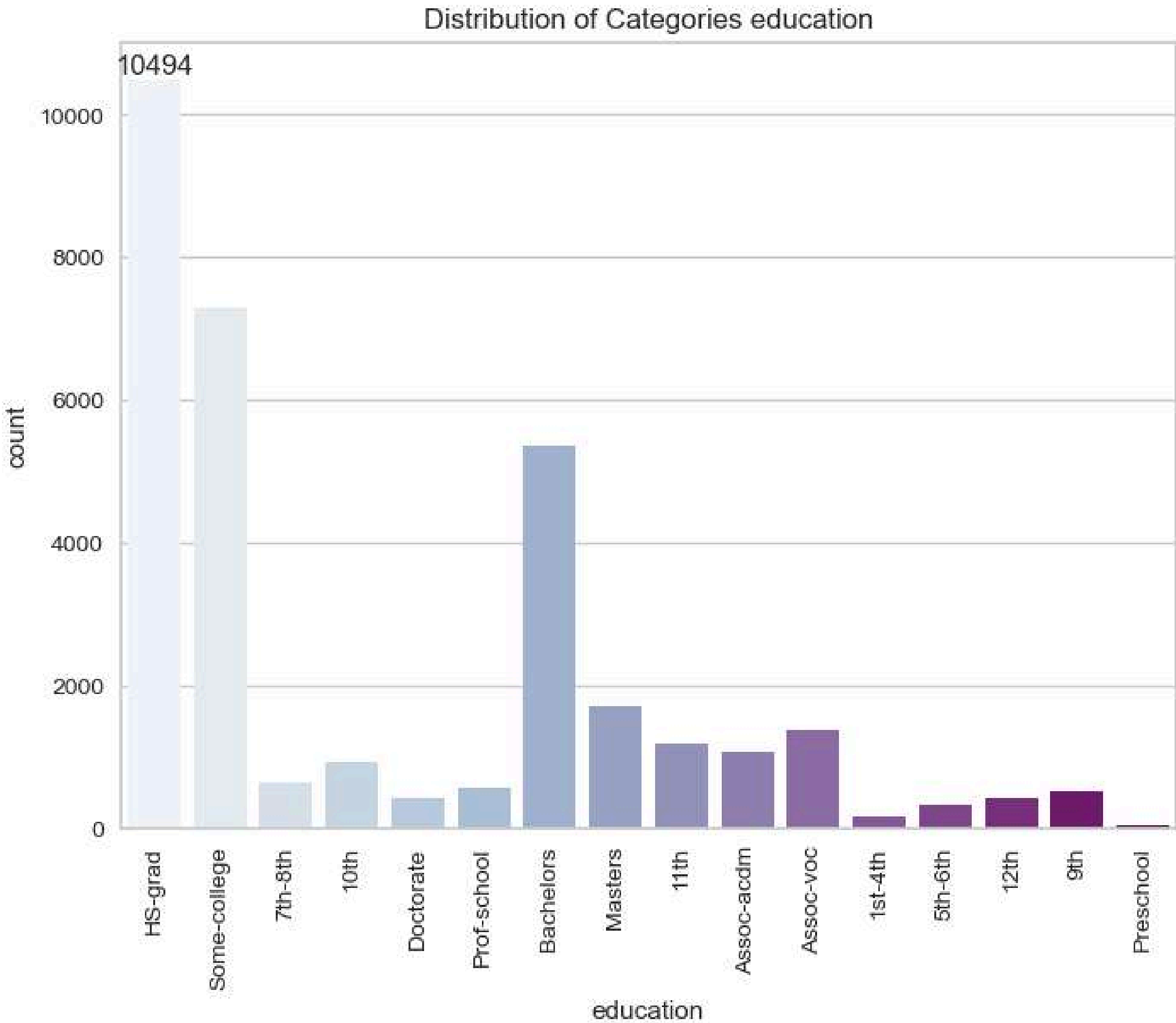
Categorical Features

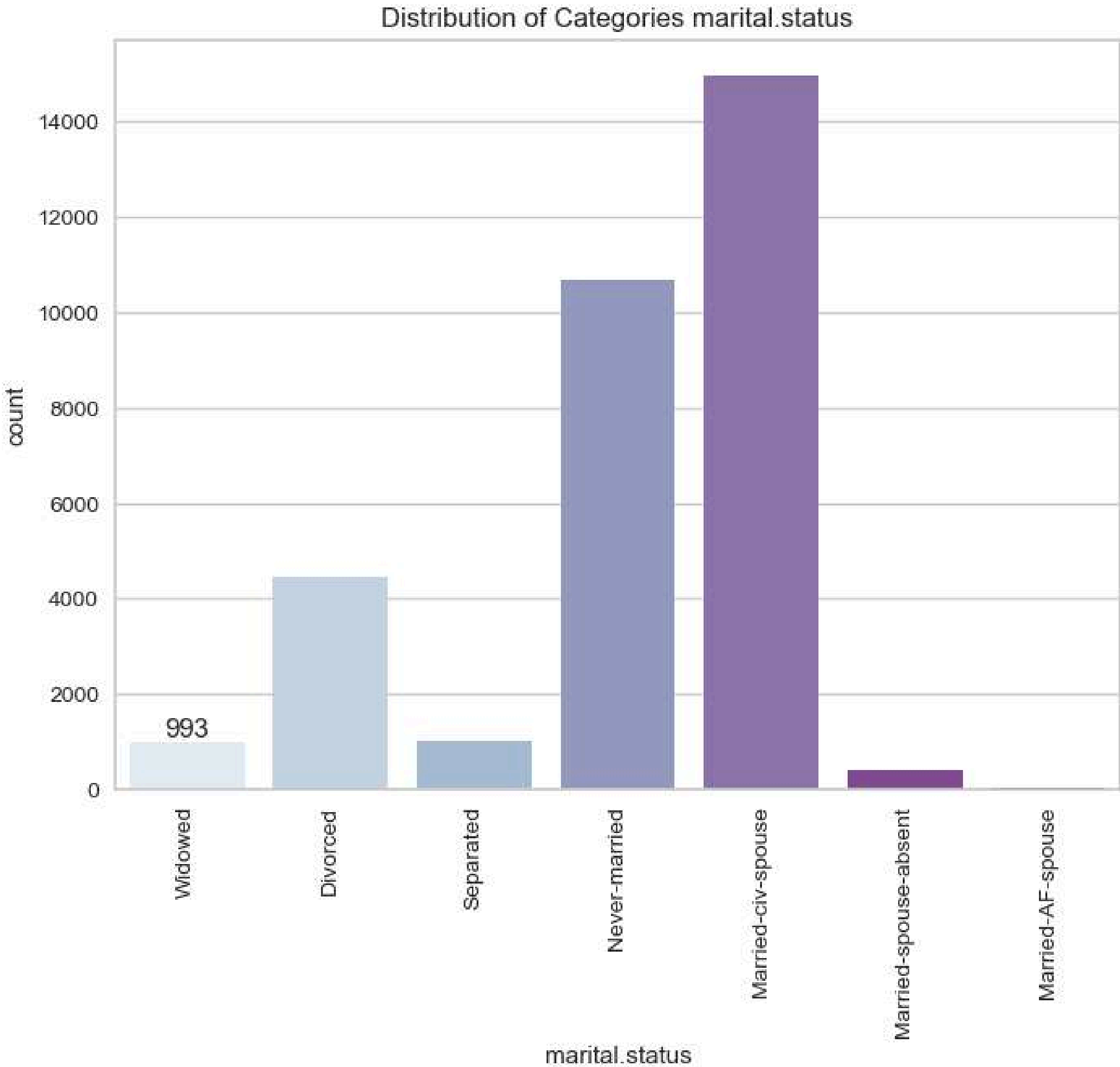
In [36]:
```
list(cat_features)
```

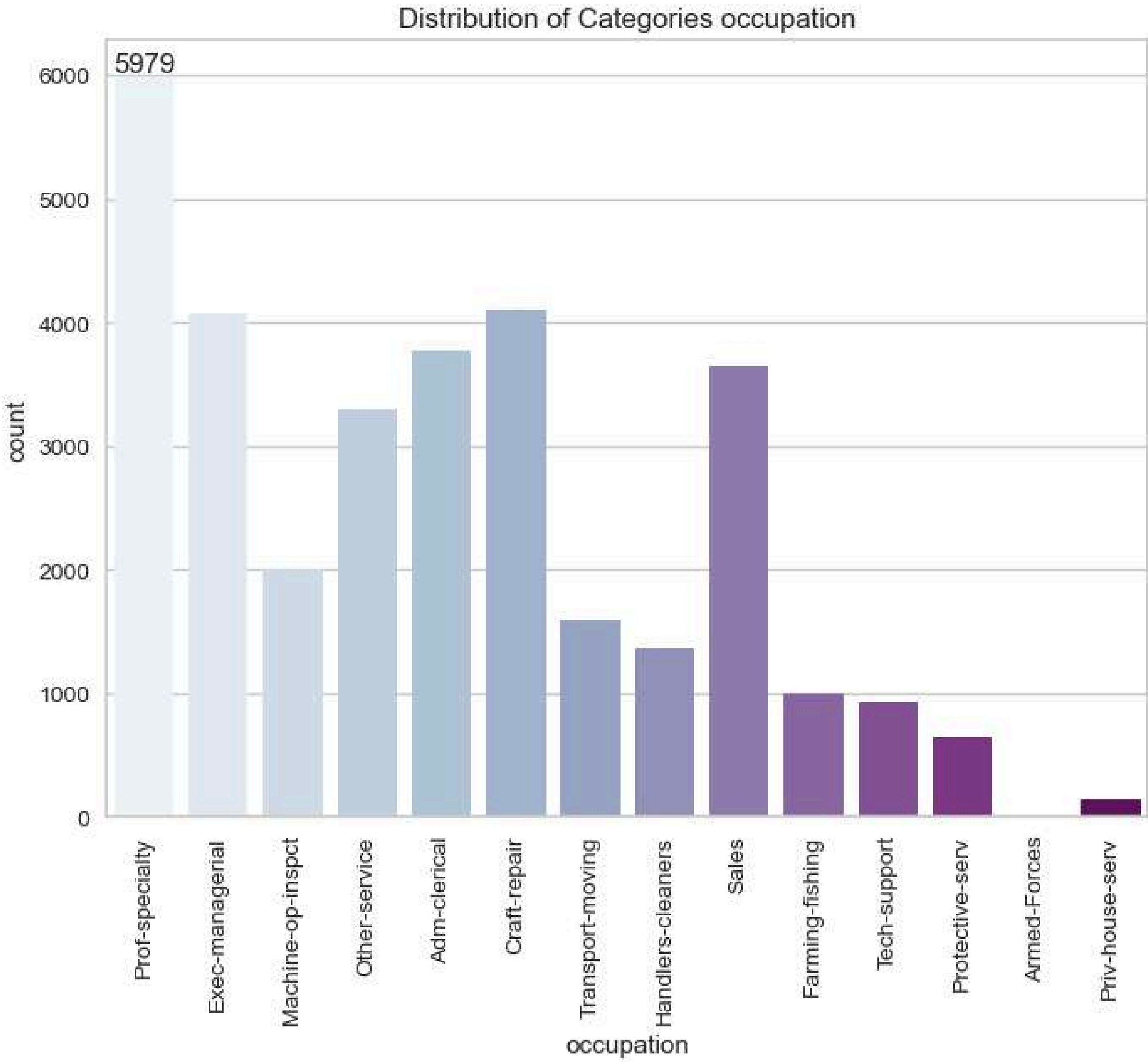
Out[36]:
```
['workclass',
 'education',
 'marital.status',
 'occupation',
 'relationship',
 'race',
 'sex',
 'native.country',
 'income']
```

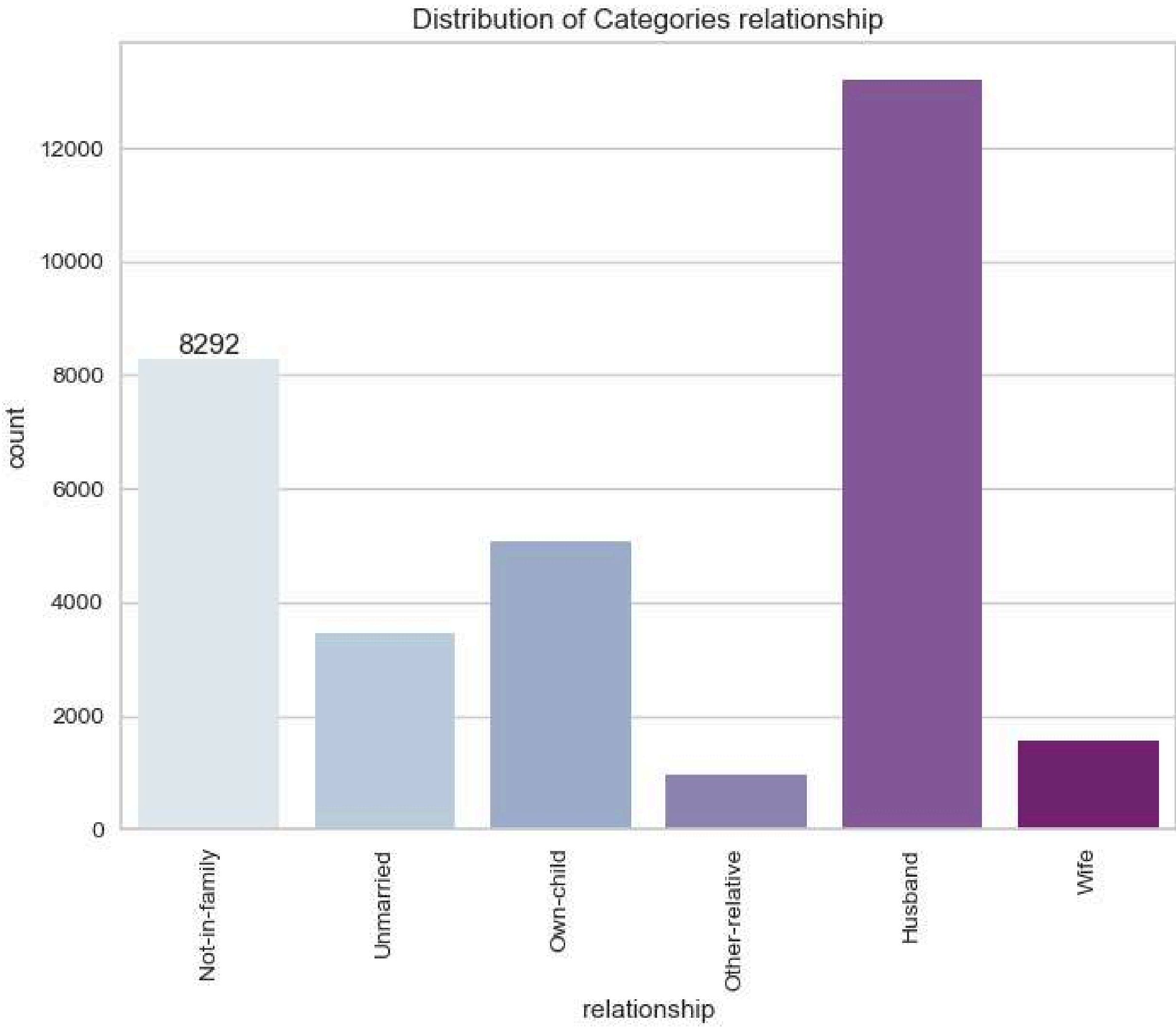
In [37]:
```
# DISTRIBUTIONS OF CATEGORICAL FEATURES;

for column in cat_features:
    plt.figure(figsize=(8, 6))
    ax = sns.countplot(x=column, data=df, palette='BuPu')
    plt.title(f'Distribution of Categories {column}')

    ax.bar_label(ax.containers[0])

    plt.xticks(rotation=90)
    plt.show()
```

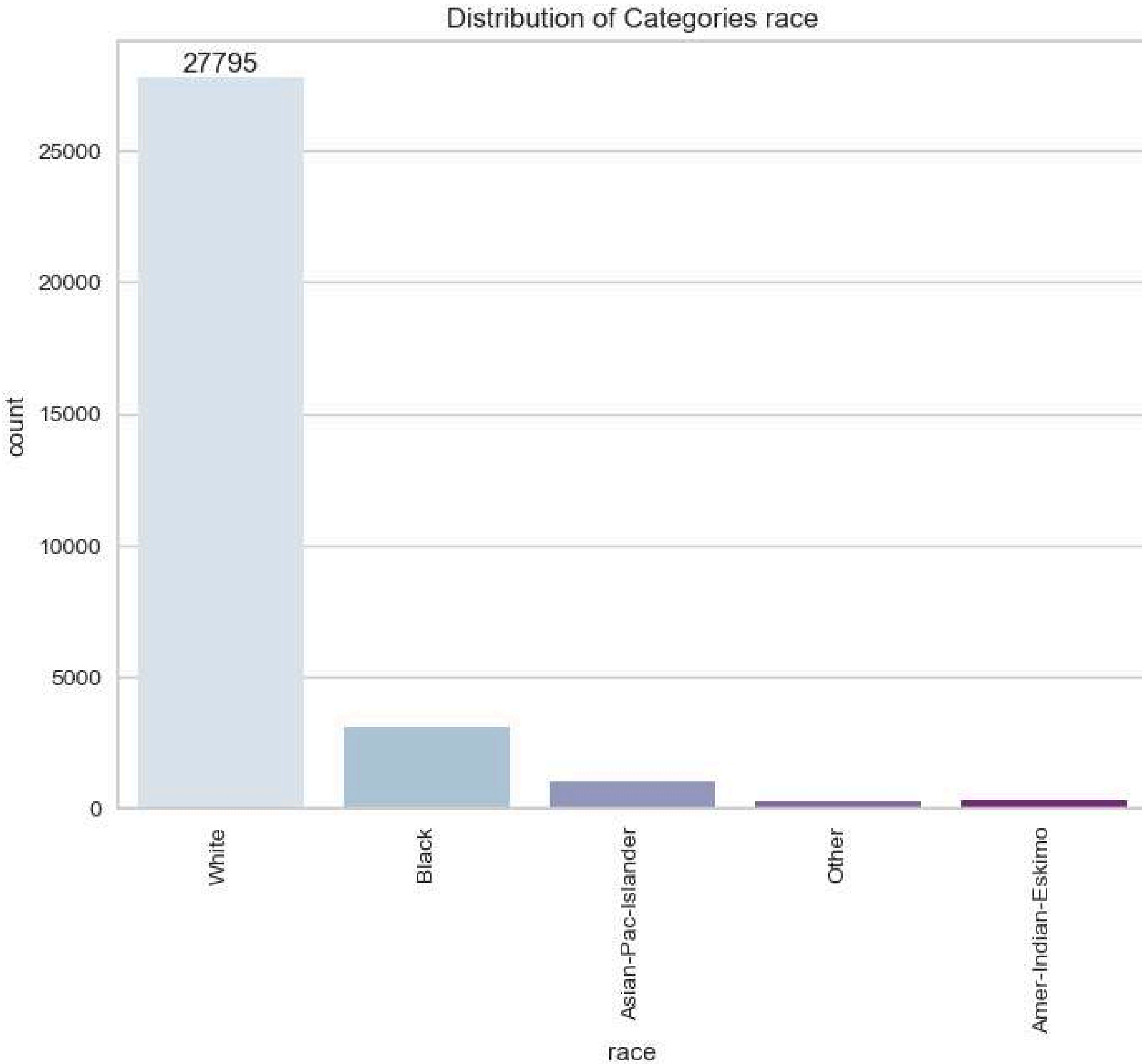

Distribution of Categories workclass
24509
25000
20000
15000
10000
5000
0
count
Private
State-gov
Federal-gov
Self-emp-not-inc
Self-emp-inc
Local-gov
Without-pay
Never-worked
workclass

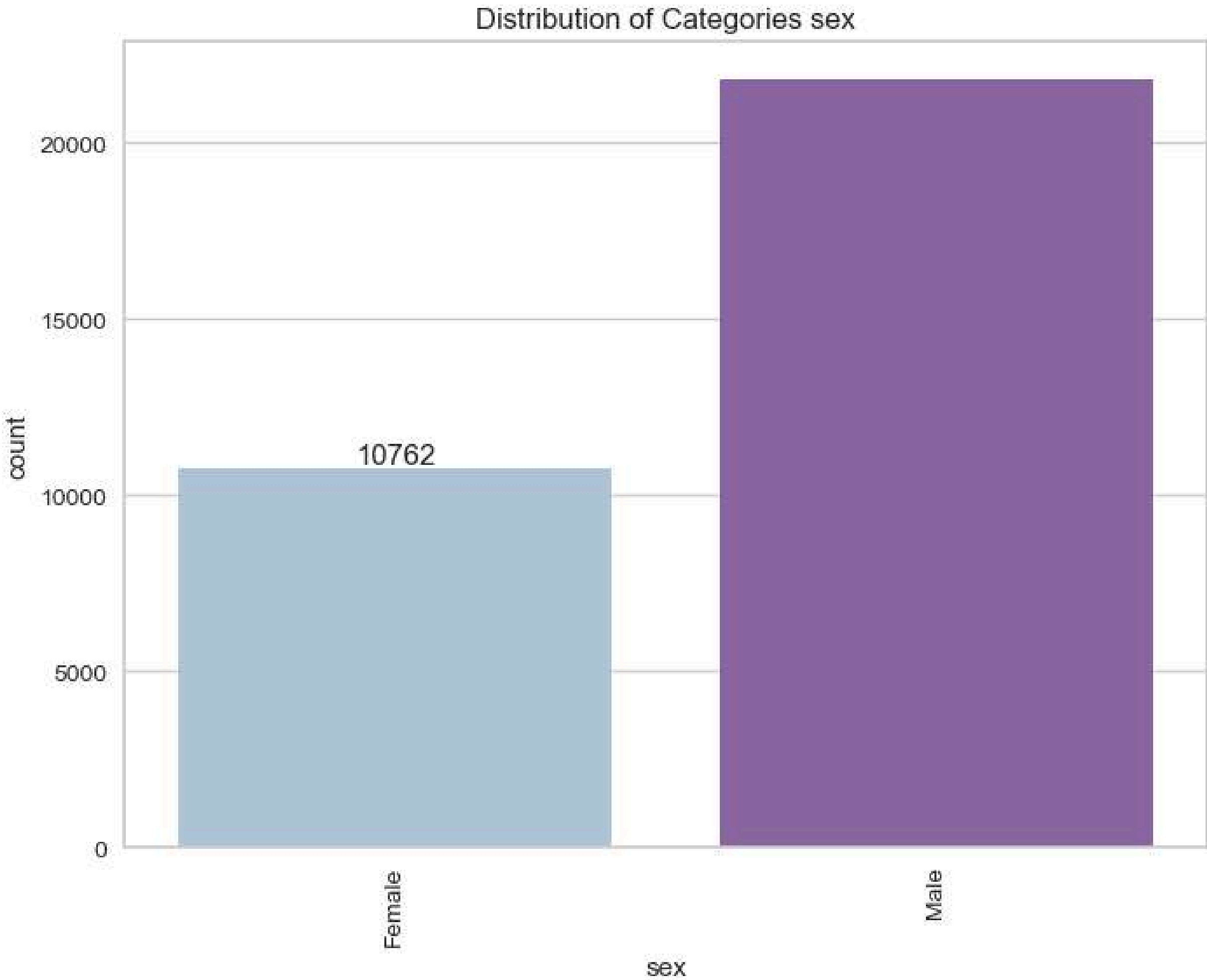

Distribution of Categories education
10494
count
10000
8000
6000
4000
2000
0
HS-grad
Some-college
7th-8th
10th
Doctorate
Prof-school
Bachelors
Masters
11th
Assoc-acdm
Assoc-voc
1st-4th
5th-6th
12th
9th
Preschool
education

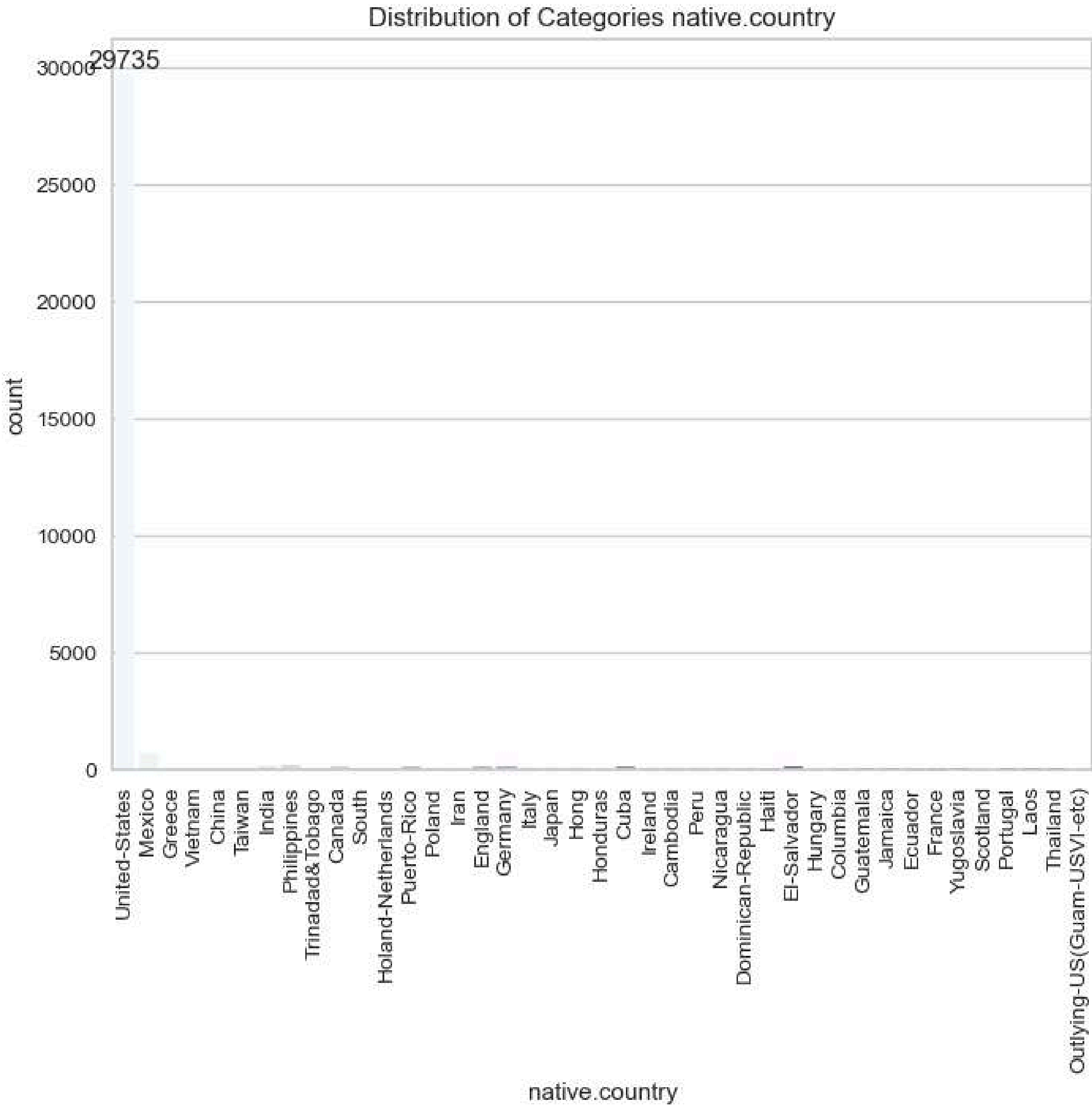

Distribution of Categories marital.status
count
0
2000
4000
6000
8000
10000
12000
14000
993
Widowed
Divorced
Separated
Never-married
Married-civ-spouse
Married-spouse-absent
Married-AF-spouse
marital.status

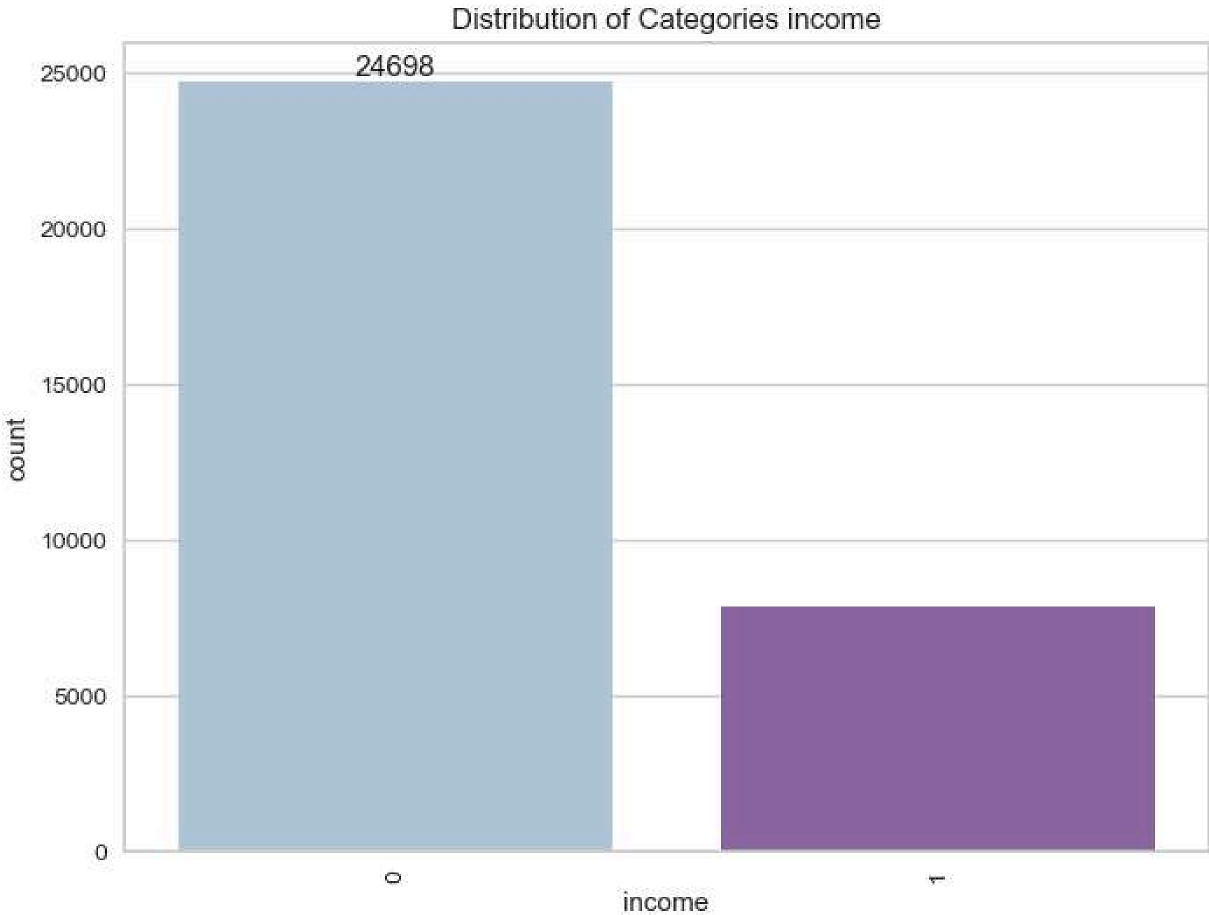

Distribution of Categories occupation
5979
6000
5000
4000
3000
2000
1000
0
count
Prof-specialty
Exec-managerial
Machine-op-inspct
Other-service
Adm-clerical
Craft-repair
Transport-moving
Handlers-cleaners
Sales
Farming-fishing
Tech-support
Protective-serv
Armed-Forces
Priv-house-serv
occupation


Distribution of Categories relationship
8292
count
12000
10000
8000
6000
4000
2000
0
Not-in-family
Unmarried
Own-child
Other-relative
Husband
Wife
relationship


Distribution of Categories race
27795
25000
20000
15000
10000
5000
0
count
White
Black
Asian-Pac-Islander
Other
Amer-Indian-Eskimo
race


Distribution of Categories sex
20000
15000
10000
5000
0
count
10762
Female
Male
sex


Distribution of Categories native.country
29735
30000
25000
20000
15000
10000
5000
0
count
United-States
Mexico
Greece
Vietnam
China
Taiwan
India
Philippines
Trinadad&Tobago
Canada
South
Holand-Netherlands
Puerto-Rico
Poland
Iran
England
Germany
Italy
Japan
Hong
Honduras
Cuba
Ireland
Cambodia
Peru
Nicaragua
Dominican-Republic
Haiti
El-Salvador
Hungary
Columbia
Guatemala
Jamaica
Ecuador
France
Yugoslavia
Scotland
Portugal
Laos
Thailand
Outlying-US(Guam-USVI-etc)
native.country



`Education` Column

- The education column was grouped to consolidate similar levels of education into broader categories.
- "Primary" includes 1st-4th', '5th-6th levels, "Middle-School" covers 6th to 8th, "High-School" represents high school graduates(HS_grad), "College" combines some college and associate degrees, while "Bachelors" and "Doctorate" remain as distinct categories for those specific degrees.
- This grouping simplifies analysis by reducing the number of unique categories.

In [38]:
```
value_cnt_fonc(df, 'education')
```

Out[38]:

	education	counts	norm_counts
0	HS-grad	10494	0.322525
1	Some-college	7282	0.223807
2	Bachelors	5353	0.164520
3	Masters	1722	0.052924
4	Assoc-voc	1382	0.042475
5	11th	1175	0.036113
6	Assoc-acdm	1067	0.032793
7	10th	933	0.028675
8	7th-8th	645	0.019824
9	Prof-school	576	0.017703
10	9th	514	0.015797
11	12th	433	0.013308
12	Doctorate	413	0.012693
13	5th-6th	332	0.010204
14	1st-4th	166	0.005102
15	Preschool	50	0.001537

In [39]:
```
df['education'].replace(['1st-4th', '5th-6th'], 'Primary', inplace=True)
df['education'].replace(['7th-8th', '9th', '10th', '11th', '12th'], 'Middle-School', inplace
df['education'].replace(['HS-grad'], 'High-School', inplace=True)
df['education'].replace(['Some-college', 'Assoc-voc', 'Assoc-acdm'], 'College', inplace=True
df['education'].replace(['Bachelors'], 'Bachelors', inplace=True)
df['education'].replace(['Prof-school', 'Doctorate'], 'Doctorate', inplace=True)
```

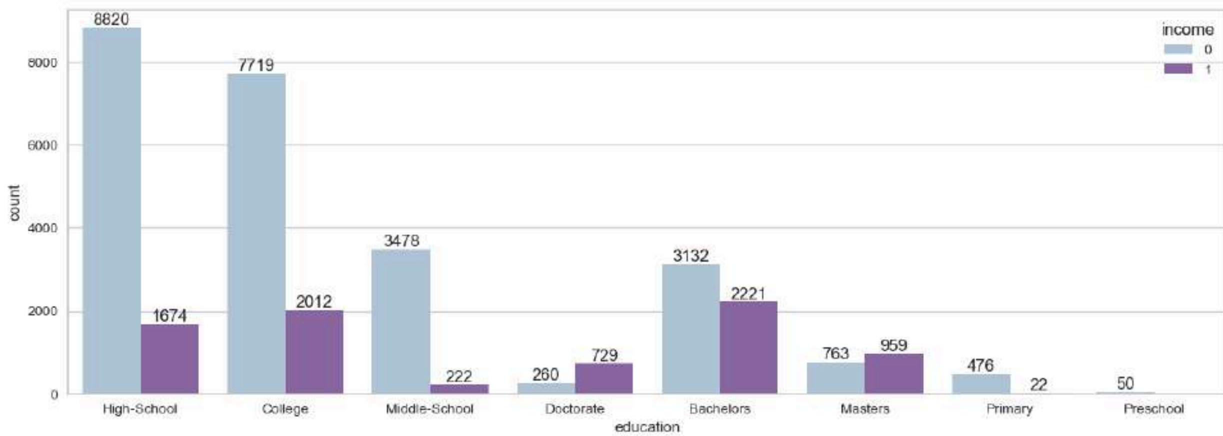
In [40]:
```
value_cnt_fonc(df, 'education')
```

Out[40]:

	education	counts	norm_counts
0	High-School	10494	0.322525
1	College	9731	0.299075
2	Bachelors	5353	0.164520
3	Middle-School	3700	0.113717
4	Masters	1722	0.052924
5	Doctorate	989	0.030396
6	Primary	498	0.015306
7	Preschool	50	0.001537

In [41]:
```
plt.figure(figsize=(15,5))
ax = sns.countplot( data=df, x="education",hue="income", palette='BuPu')
```

```
ax.bar_label(ax.containers[0])
ax.bar_label(ax.containers[1])
plt.show()
```



`Race` Column

- In the Race column, categories with a low number of observations can be combined under an "Other" category.

In [42]:
```
value_cnt_fonc(df, 'race')
```

Out[42]:

	race	counts	norm_counts
0	White	27795	0.854258
1	Black	3122	0.095952
2	Asian-Pac-Islander	1038	0.031902
3	Amer-Indian-Eskimo	311	0.009558
4	Other	271	0.008329

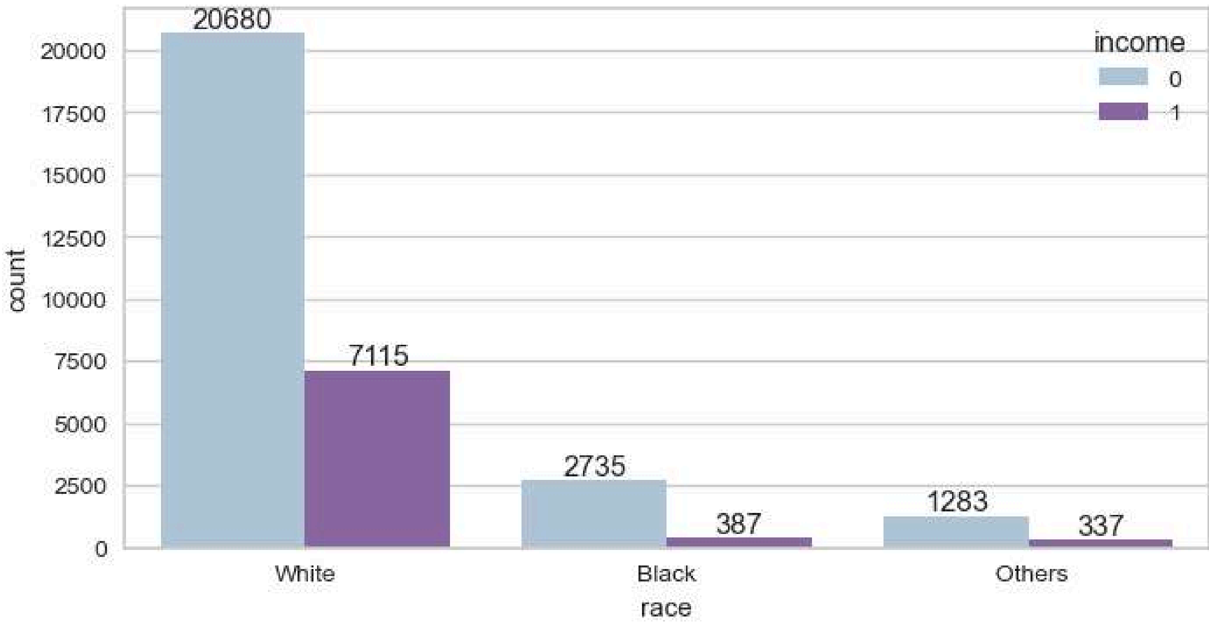
In [43]:
```
df['race'].replace(['Asian-Pac-Islander', 'Amer-Indian-Eskimo', 'Other'],' Others', inplace
```

In [44]:
```
value_cnt_fonc(df, 'race')
```

Out[44]:

	race	counts	norm_counts
0	White	27795	0.854258
1	Black	3122	0.095952
2	Others	1620	0.049789

In [45]:
```
plt.figure(figsize=(8,4))
ax = sns.countplot( data=df, x="race",hue='income', palette='BuPu')

ax.bar_label(ax.containers[0])
ax.bar_label(ax.containers[1])
plt.show()
```



`native.country` Column

- In the native.country column, countries other than the USA can be grouped as "Others."

In [46]:
```
value_cnt_fonc(df, 'native.country')
```

Out[46]:

	native.country	counts	norm_counts
0	United-States	29735	0.913883
1	Mexico	639	0.019639
2	Philippines	198	0.006085
3	Germany	137	0.004211
4	Canada	121	0.003719
5	Puerto-Rico	114	0.003504
6	El-Salvador	106	0.003258
7	India	100	0.003073
8	Cuba	95	0.002920
9	England	90	0.002766
10	Jamaica	81	0.002489
11	South	80	0.002459
12	China	75	0.002305
13	Italy	73	0.002244
14	Dominican-Republic	70	0.002151
15	Vietnam	67	0.002059
16	Guatemala	62	0.001906
17	Japan	62	0.001906
18	Poland	60	0.001844
19	Columbia	59	0.001813
20	Taiwan	51	0.001567
21	Haiti	44	0.001352
22	Iran	43	0.001322
23	Portugal	37	0.001137
24	Nicaragua	34	0.001045
25	Peru	31	0.000953
26	Greece	29	0.000891
27	France	29	0.000891
28	Ecuador	28	0.000861
29	Ireland	24	0.000738
30	Hong	20	0.000615
31	Trinadad&Tobago	19	0.000584
32	Cambodia	19	0.000584

	native.country	counts	norm_counts
33	Thailand	18	0.000553
34	Laos	18	0.000553
35	Yugoslavia	16	0.000492
36	Outlying-US(Guam-USVI-etc)	14	0.000430
37	Hungary	13	0.000400
38	Honduras	13	0.000400
39	Scotland	12	0.000369
40	Holand-Netherlands	1	0.000031

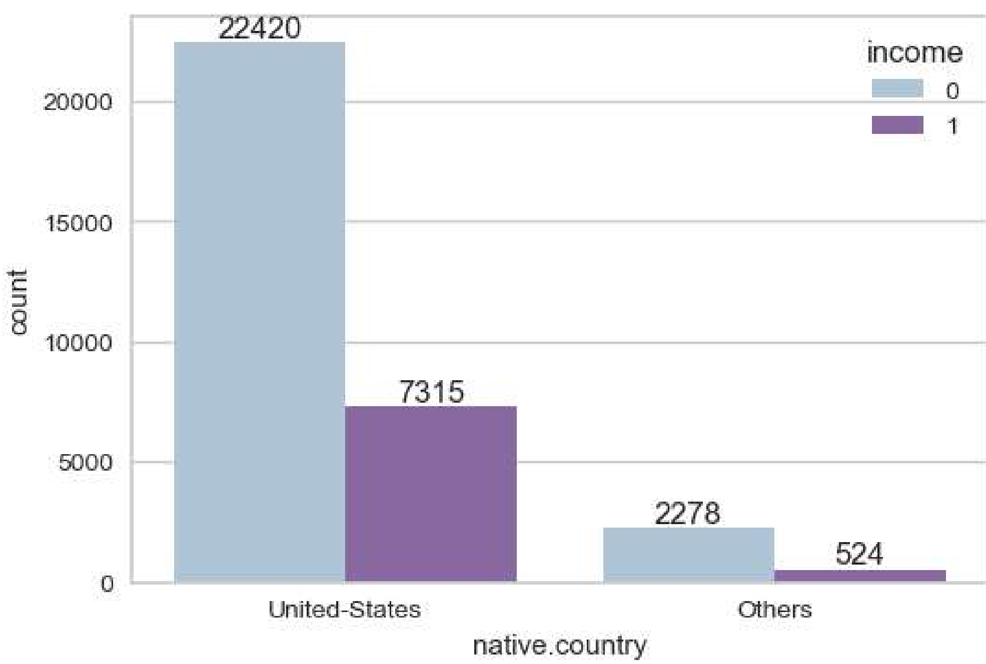
In [47]:
```
# Replaces all values in the native.country column that are not "United-States" with "Others

df['native.country'].loc[df['native.country'] != 'United-States'] = 'Others'
```

In [48]:
```
value_cnt_fonc(df, 'native.country')
```

Out[48]:

	native.country	counts	norm_counts
0	United-States	29735	0.913883
1	Others	2802	0.086117

In [49]:
```
plt.figure(figsize=(6,4))
ax = sns.countplot( data=df, x="native.country",hue='income', palette='BuPu')

ax.bar_label(ax.containers[0])
ax.bar_label(ax.containers[1])
plt.show()
```



NOTE

- This distinction is made to simplify the analysis and modeling process by reducing the number of categorical variables.
- Since the majority of the data is from the United States, grouping all other countries into a single "Others" category reduces the complexity of dealing with numerous country categories.
- Similarly, in the education column, grouping lower education levels (e.g., '11th', '9th', etc.) into "Pre-High School" and combining less frequent race categories into "Other" helps focus on the most significant groups while avoiding potential noise from less common categories.
- This approach ensures that the model remains robust and performs better by not being overwhelmed by too many distinct levels.

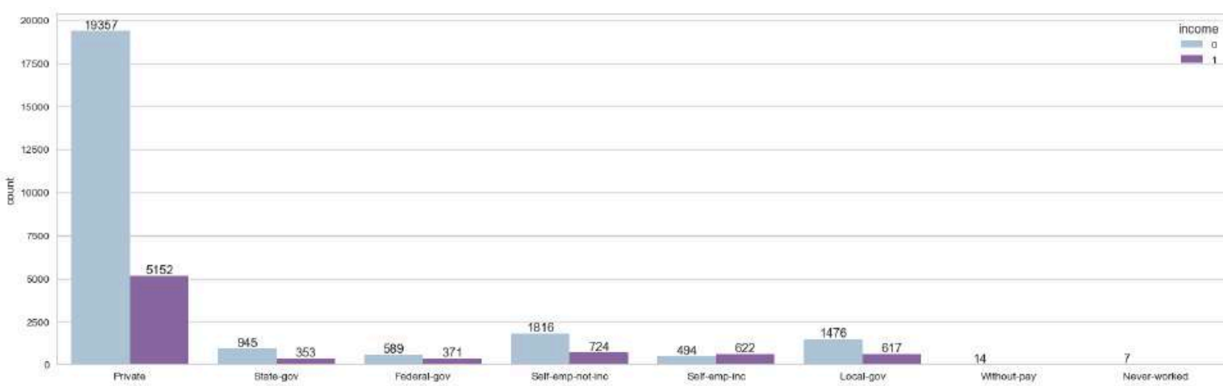
`WorkClass` Column

In [50]:
```
value_cnt_fonc(df, 'workclass')
```

Out[50]:

	workclass	counts	norm_counts
0	Private	24509	0.753266
1	Self-emp-not-inc	2540	0.078065
2	Local-gov	2093	0.064327
3	State-gov	1298	0.039893
4	Self-emp-inc	1116	0.034299
5	Federal-gov	960	0.029505
6	Without-pay	14	0.000430
7	Never-worked	7	0.000215

In [51]:
```
plt.figure(figsize=(20,6))
ax = sns.countplot( data=df, x="workclass",hue='income', palette='BuPu')

ax.bar_label(ax.containers[0])
ax.bar_label(ax.containers[1])
plt.show()
```



WorkClass

- **Private**: Individuals working in the private sector. This category has the largest number of individuals, with a significant portion earning `<=50K` and a smaller, yet noticeable, portion

earning `>50K`.
- **State-gov**: Individuals working in state government positions. Most of these individuals earn `<=50K`, with a small number earning `>50K`.
- **Federal-gov**: Individuals employed by the federal government. Similar to the state government category, most earn `<=50K`, with fewer earning `>50K`.
- **Self-emp-not-inc**: Self-employed individuals who do not have incorporated businesses. This category shows a mix of income levels, but more individuals earn `<=50K`.
- **Self-emp-inc**: Self-employed individuals with incorporated businesses. This group has a smaller population, but a higher proportion earning `>50K` compared to other categories.
- **Local-gov**: Individuals working in local government positions. Most earn `<=50K`, but there is a small group earning `>50K`.
- **Without-pay**: Individuals working without pay. This is a very small group, and the few individuals in this category earn `<=50K`.
- **Never-worked**: Individuals who have never worked. This is the smallest group, with all individuals earning `<=50K`.

- The majority of individuals in the dataset work in the private sector, with most of them earning `<=50K`.
- Self-employed individuals with incorporated businesses (`Self-emp-inc`) have a relatively higher proportion of individuals earning `>50K` compared to other categories.
- Government employees (state, federal, and local) generally earn `<=50K`, but there are exceptions, particularly in federal positions.

`Occupation` Column

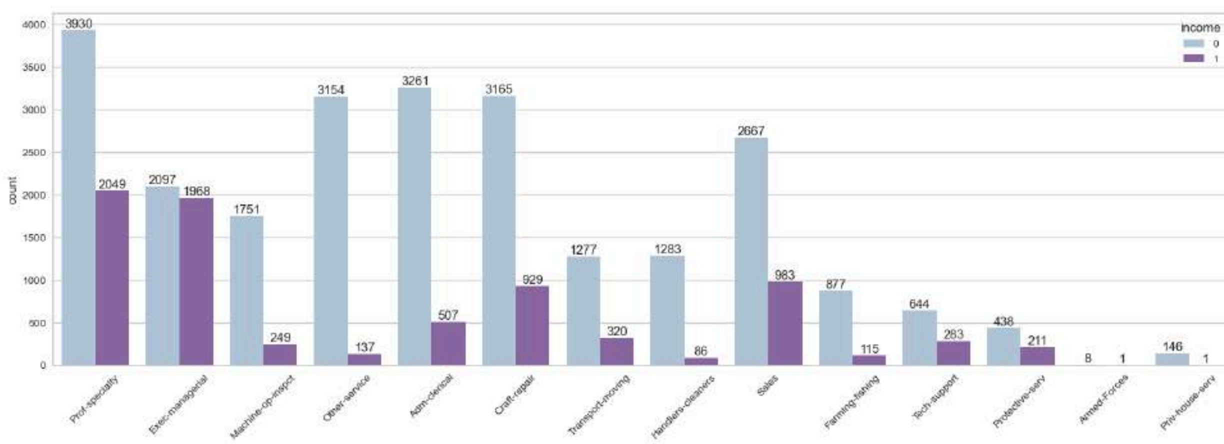
In [52]:
```
value_cnt_fonc(df, 'occupation')
```

Out[52]:

	occupation	counts	norm_counts
0	Prof-specialty	5979	0.183760
1	Craft-repair	4094	0.125826
2	Exec-managerial	4065	0.124935
3	Adm-clerical	3768	0.115807
4	Sales	3650	0.112180
5	Other-service	3291	0.101146
6	Machine-op-inspct	2000	0.061468
7	Transport-moving	1597	0.049083
8	Handlers-cleaners	1369	0.042075
9	Farming-fishing	992	0.030488
10	Tech-support	927	0.028491
11	Protective-serv	649	0.019947
12	Priv-house-serv	147	0.004518
13	Armed-Forces	9	0.000277

In [53]:

```
plt.figure(figsize=(20,6))
ax = sns.countplot( data=df, x="occupation",hue='income', palette='BuPu')

ax.bar_label(ax.containers[0])
ax.bar_label(ax.containers[1])
plt.xticks(rotation=45)
plt.show()
```



`Marital-Status` Column

- Widowed: Individuals who have lost their spouse and have not remarried.
- Divorced: Individuals who have legally ended their marriage.
- Separated: Individuals who are still legally married but are living separately from their spouse.
- Never-married: Individuals who have never been married.
- Married-civ-spouse: Individuals who are married and living with their spouse (civilian spouse).

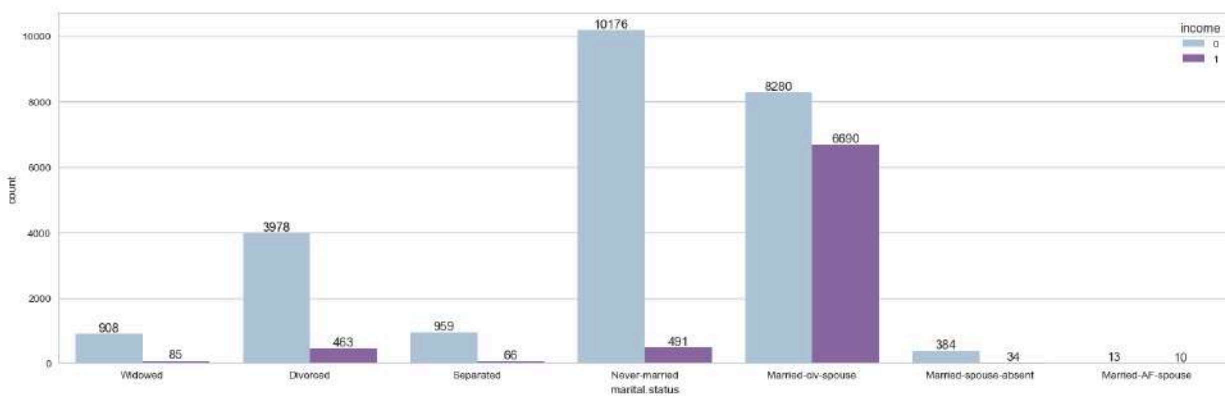
- Married-spouse-absent: Individuals who are married but not currently living with their spouse.
- Married-AF-spouse: Individuals married to someone in the Armed Forces, likely living separately due to military service.

In [54]:
```
value_cnt_fonc(df, 'marital.status')
```

Out[54]:

	marital.status	counts	norm_counts
0	Married-civ-spouse	14970	0.460092
1	Never-married	10667	0.327842
2	Divorced	4441	0.136491
3	Separated	1025	0.031503
4	Widowed	993	0.030519
5	Married-spouse-absent	418	0.012847
6	Married-AF-spouse	23	0.000707

In [55]:
```
plt.figure(figsize=(20,6))
ax = sns.countplot( data=df, x="marital.status",hue='income', palette='BuPu')

ax.bar_label(ax.containers[0])
ax.bar_label(ax.containers[1])
plt.show()
```



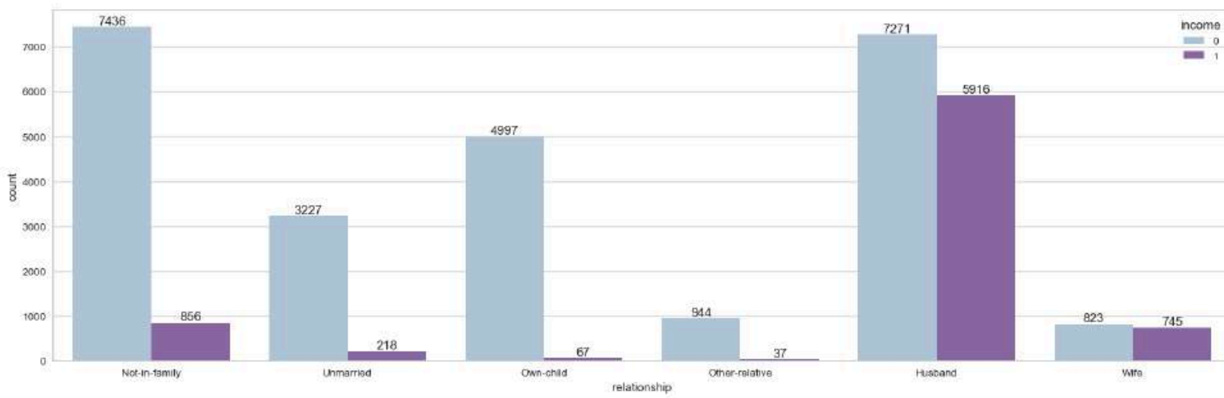
`Relationship` Column

In [56]:
```
value_cnt_fonc(df, 'relationship')
```

Out[56]:

	relationship	counts	norm_counts
0	Husband	13187	0.405292
1	Not-in-family	8292	0.254848
2	Own-child	5064	0.155638
3	Unmarried	3445	0.105879
4	Wife	1568	0.048191
5	Other-relative	981	0.030150

In [57]:
```
plt.figure(figsize=(20,6))
ax = sns.countplot( data=df, x="relationship",hue='income', palette='BuPu')

ax.bar_label(ax.containers[0])
ax.bar_label(ax.containers[1])
plt.show()
```



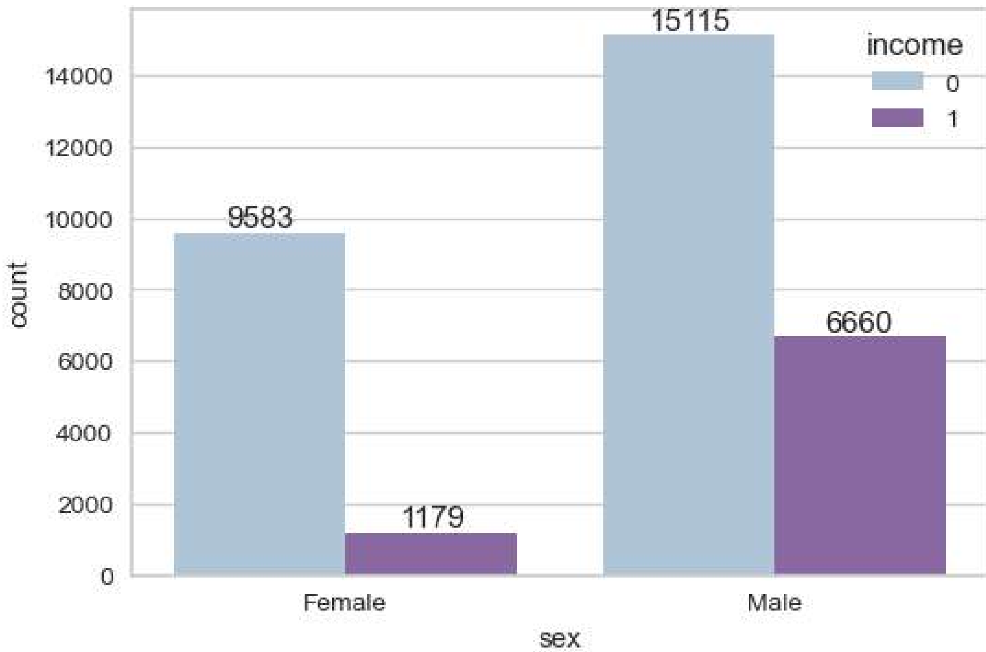
Sex Column

In [58]:
```
value_cnt_fonc(df, 'sex')
```

Out[58]:

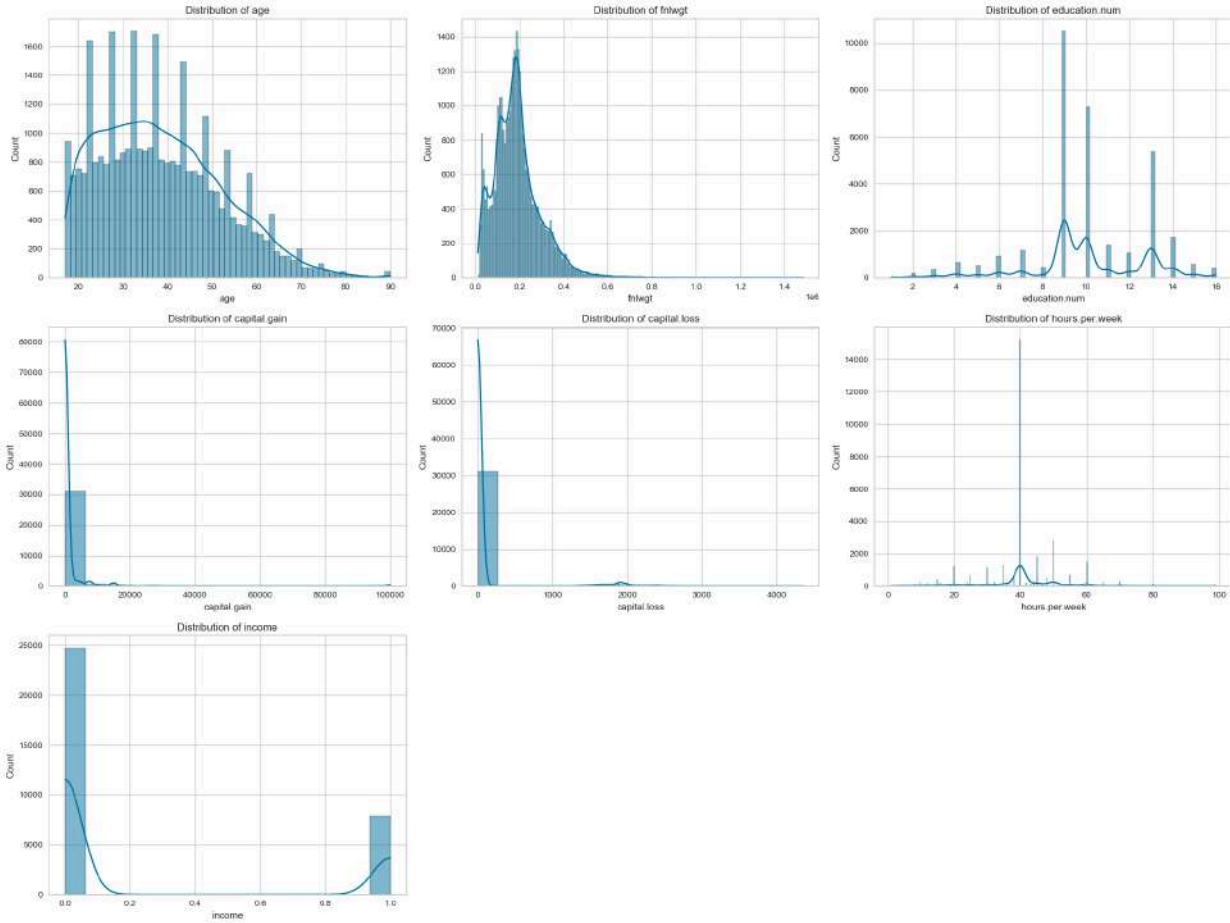
	sex	counts	norm_counts
0	Male	21775	0.669238
1	Female	10762	0.330762

In [59]:
```
plt.figure(figsize=(6,4))
ax = sns.countplot( data=df, x="sex",hue='income', palette='BuPu')

ax.bar_label(ax.containers[0])
ax.bar_label(ax.containers[1])
plt.show()
```



Numerical Features

In [60]:
```
# DISTRIBUTIONS OF NUMERICAL FEATURES;

numerical_df = df.select_dtypes(include=['number'])

plt.figure(figsize=(20,15))

num_vars = len(numerical_df.columns)

for i, var in enumerate(numerical_df.columns, 1):
    plt.subplot((num_vars // 3) + 1, 3, i)
    sns.histplot(data=df, x=var, kde=True)
    plt.title(f'Distribution of {var}')

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



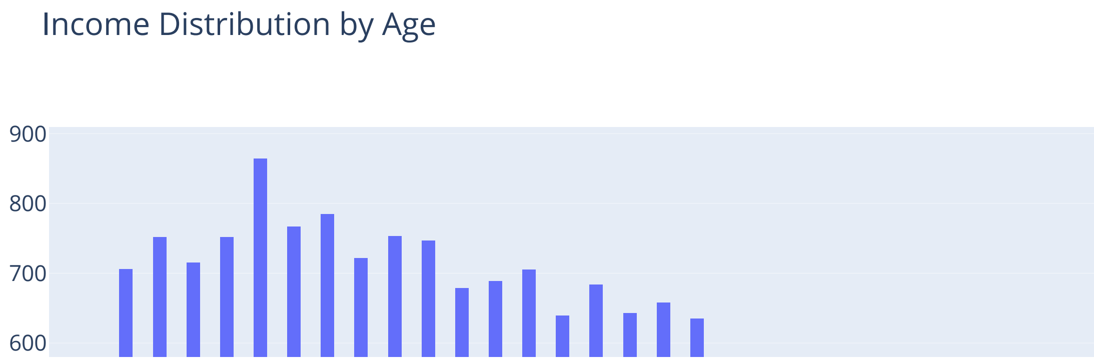
Analysis

1. **Age Distribution**: The age distribution is right-skewed, with most individuals clustered between 20 and 50 years old, gradually decreasing as age increases.
2. **Fnlwgt Distribution**: The `fnlwgt` (final weight) feature shows a right-skewed distribution, indicating that most individuals have a lower final weight.
3. **Education.num**: The distribution of education levels is multimodal, with significant peaks around levels 9 (high school graduate) and 10 (some college education).
4. **Capital Gain and Loss**: Both `capital.gain` (profit from the sale of assets) and `capital.loss` (loss from the sale of assets) are highly right-skewed, with most individuals reporting values close to zero and only a few reporting substantial gains or losses.
5. **Hours per Week**: The majority of individuals work around 40 hours per week, with a sharp peak at this value, indicating a standard workweek.
6. **Income**: The income distribution shows that most individuals earn less than or equal to 50K (indicated by 0), with fewer individuals earning more than 50K (indicated by 1).

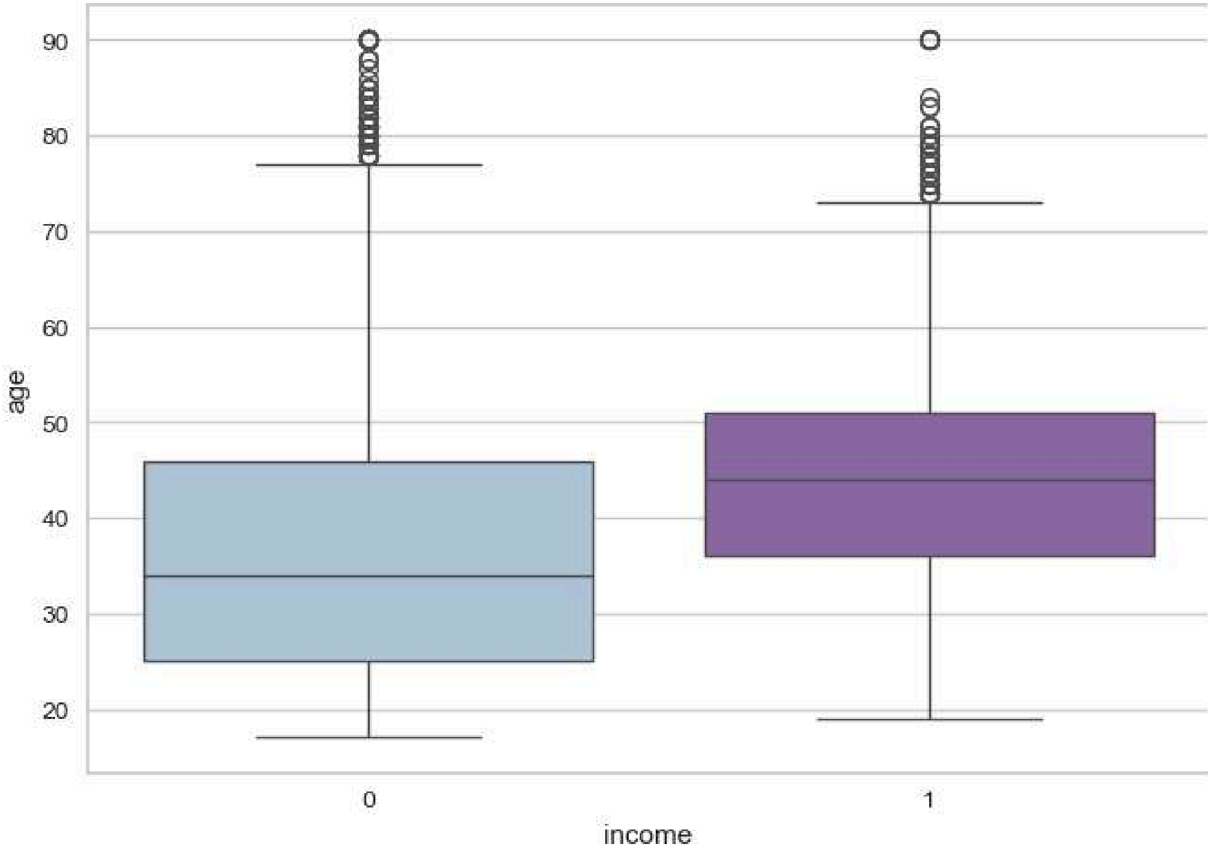
These insights highlight the skewed nature of certain features, particularly `capital.gain` and `capital.loss`, which may require special consideration during analysis or modeling.

`Age` Column

In [61]:
```
px.histogram(df, x='age', color="income", barmode='group', title='Income Distribution by Age
```



In [62]:
```
sns.boxplot(data=df,y="age",x='income',palette='BuPu');
```



`Education.num` Column

In [63]:
```
# Number of years of education completed by the individuals

value_cnt_fonc(df, 'education.num')
```

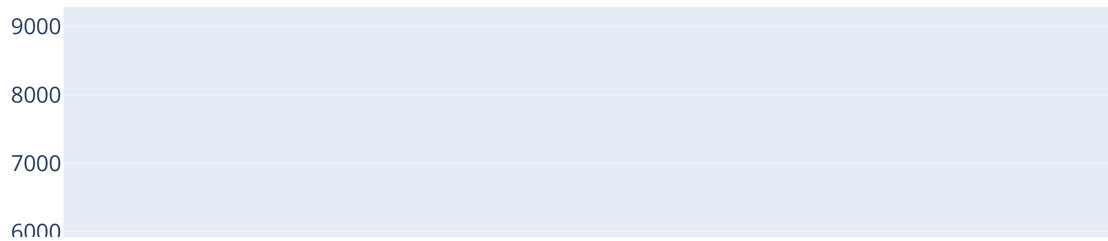
Out[63]:

	education.num	counts	norm_counts
0	9	10494	0.322525
1	10	7282	0.223807
2	13	5353	0.164520
3	14	1722	0.052924
4	11	1382	0.042475
5	7	1175	0.036113
6	12	1067	0.032793
7	6	933	0.028675
8	4	645	0.019824
9	15	576	0.017703
10	5	514	0.015797
11	8	433	0.013308
12	16	413	0.012693
13	3	332	0.010204
14	2	166	0.005102
15	1	50	0.001537

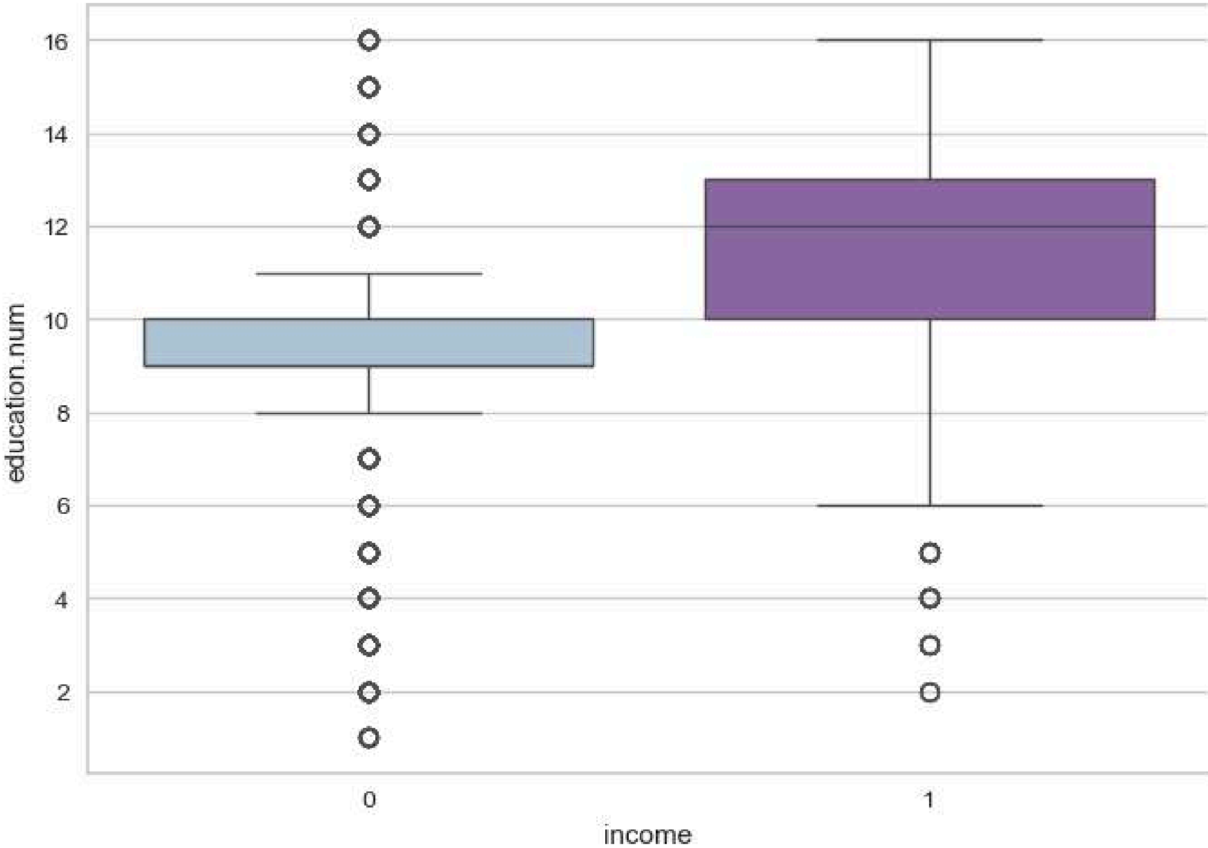
In [64]:

```
# Number of years of education completed by the individuals

px.histogram(df, x='education.num', color="income", barmode='group', title='Income Distribut
```

Income Distribution by Education Num



In [65]:
```
sns.boxplot(data=df,y="education.num",x='income', palette='BuPu');
```



`Capital Gain and Loss` Columns

- Createing a new feature `capital_diff` by calculating the difference between `capital.loss` (which represents profit from the sale of an asset) and `capital.gain` (which represents a loss from the sale of an asset).
- The difference is then categorized into 'Low' and 'High' based on specified bins, converted into a categorical object type, and the original capital.gain and capital.loss columns are dropped from the dataset.
- Values between -5000 and 5000 are assigned to the 'Low' category, while values between 5000 and 100000 are assigned to the 'High' category.
- The goal is to simplify the dataframe by combining the information from the capital.gain and capital.loss columns into a single categorical column and to categorize whether the total gain/loss obtained from these two columns is low or high.

Note:

- Capital Gain: The profit earned when an asset is sold for more than its purchase price.
- Capital Loss: The loss incurred when an asset is sold for less than its purchase price.
- *So, the difference between the two gives the net effect—whether ended up with an overall profit or loss from the transactions.*

In [66]:
```
value_cnt_fonc(df, 'capital.gain')
```

Out[66]:

	capital.gain	counts	norm_counts
0	0	29825	0.916649
1	15024	347	0.010665
2	7688	284	0.008729
3	7298	246	0.007561
4	99999	159	0.004887
...	...	...	...
114	1111	1	0.000031
115	4931	1	0.000031
116	7978	1	0.000031
117	5060	1	0.000031
118	2538	1	0.000031

119 rows × 3 columns

In [67]:
```
value_cnt_fonc(df, 'capital.loss')
```

Out[67]:

	capital.loss	counts	norm_counts
0	0	31018	0.953315
1	1902	202	0.006208
2	1977	168	0.005163
3	1887	159	0.004887
4	1485	51	0.001567
...	...	...	...
87	2201	1	0.000031
88	2163	1	0.000031
89	1944	1	0.000031
90	1539	1	0.000031
91	2472	1	0.000031

92 rows × 3 columns

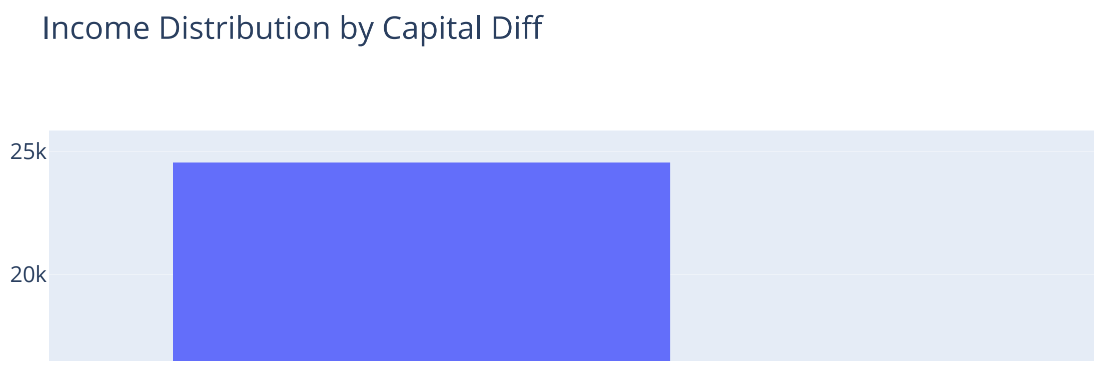
In [68]:
```
df['capital_diff'] = df['capital.gain'] - df['capital.loss']
df['capital_diff'] = pd.cut(df['capital_diff'], bins = [-5000, 5000, 100000], labels = ['Low
df['capital_diff'] = df['capital_diff'].astype('object')
df.drop(['capital.gain'], axis = 1, inplace = True)
df.drop(['capital.loss'], axis = 1, inplace = True)
```

In [69]:
```
value_cnt_fonc(df, 'capital_diff')
```

Out[69]:

	capital_diff	counts	norm_counts
0	Low	30889	0.94935
1	High	1648	0.05065

In [70]:
```
px.histogram(df, x='capital_diff', color="income", barmode='group', title='Income Distributi
```



`hours.per.week` Column

- Filtering the `hours.per.week` column to focus on individuals who work within a more typical range of hours per week.
- Working less than 20 hours or more than 72 hours is unusual and considered an outlier.
- Removing these outliers helps to ensure that the analysis is more accurate and reflects standard work patterns.

In [71]:
```
value_cnt_fonc(df, 'hours.per.week')
```

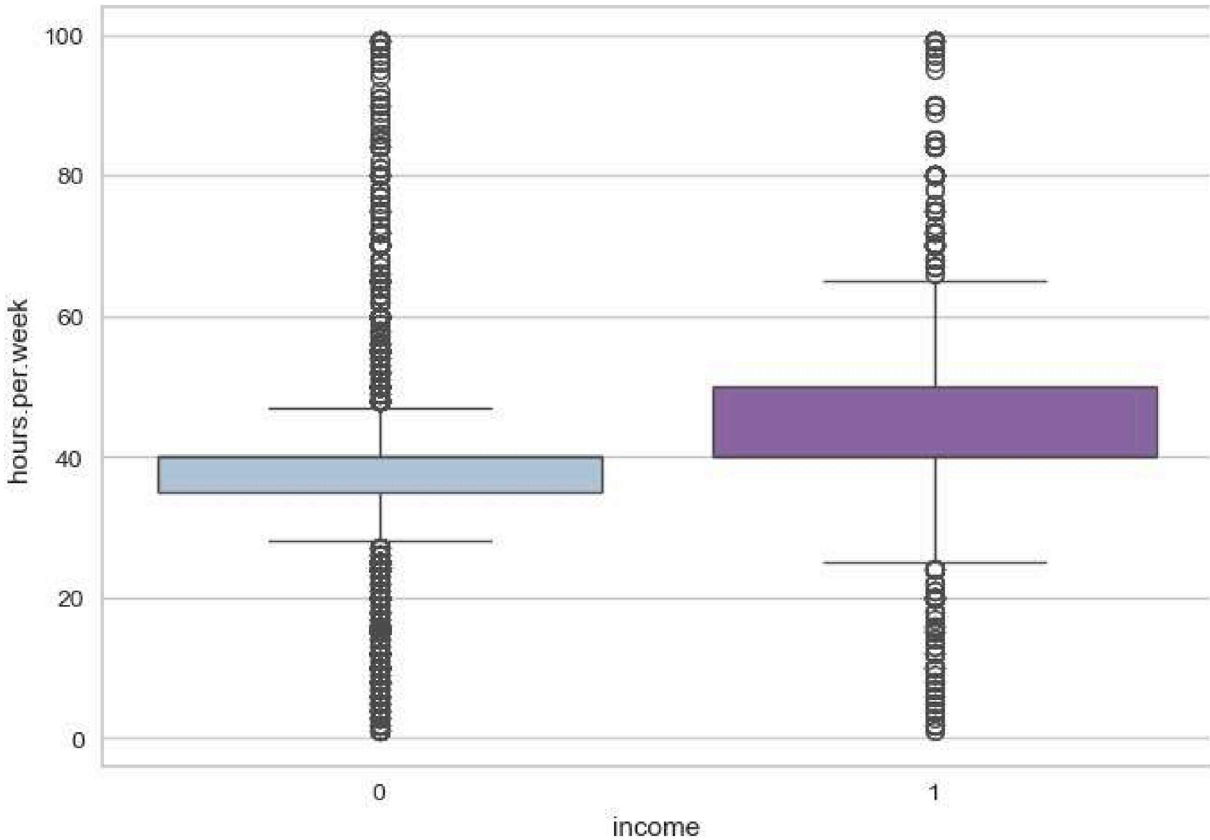
Out[71]:

	hours.per.week	counts	norm_counts
0	40	15204	0.467283
1	50	2817	0.086578
2	45	1823	0.056029
3	60	1475	0.045333
4	35	1296	0.039832
...	...	...	...
89	94	1	0.000031
90	82	1	0.000031
91	92	1	0.000031
92	87	1	0.000031
93	74	1	0.000031

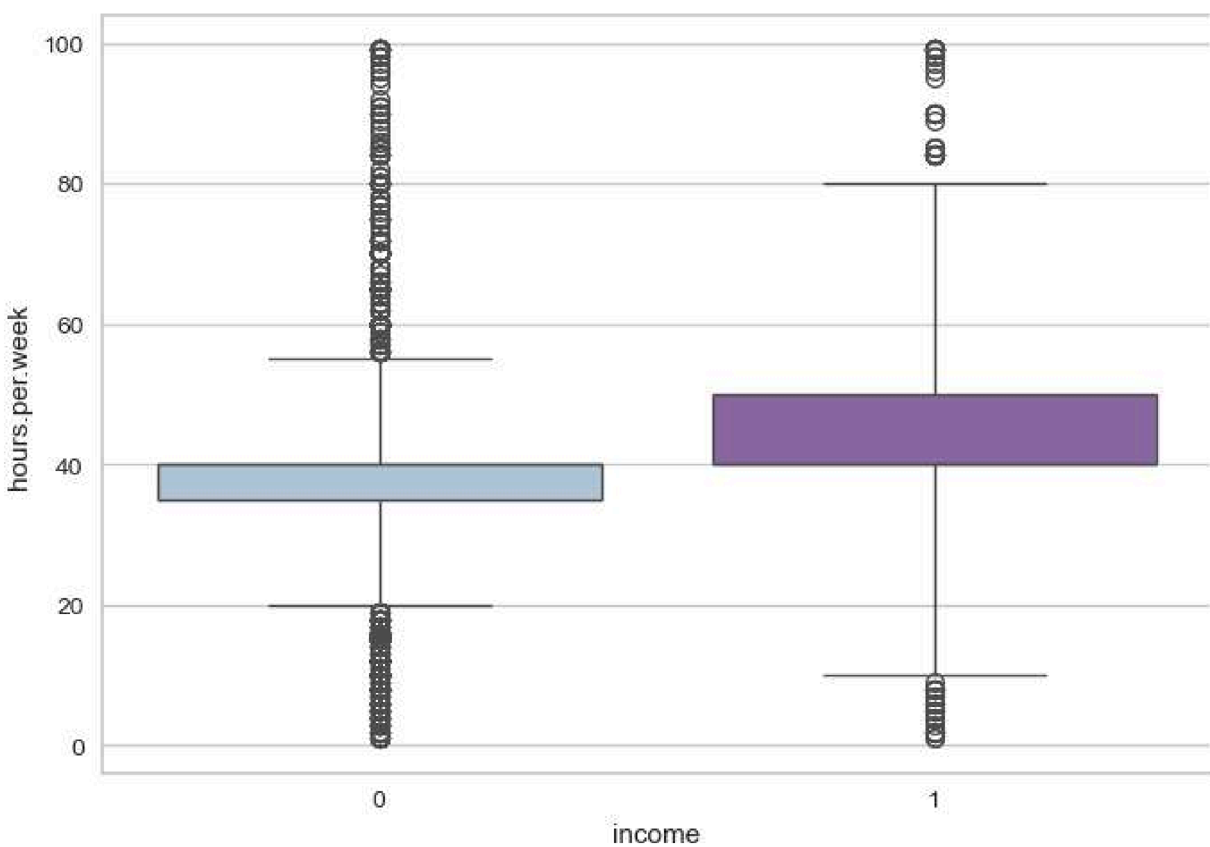
94 rows × 3 columns

In [72]:
```
px.histogram(df, x='hours.per.week', color="income", barmode='group', title='Income Distribu
```

In [73]:
```
sns.boxplot(data=df,y="hours.per.week",x='income', palette='BuPu');
```



In [74]:
```
# whis=3
sns.boxplot(data=df,y="hours.per.week",x='income', palette='BuPu', whis=3);
```



In [75]:
```
# Total number of individuals who work more than 72 hours per week
len(df[df["hours.per.week"]>72])
```

Out[75]: 427

In [76]:
```
# Total number of individuals who work less than 20 hours per week
len(df[df["hours.per.week"] < 20])
```

Out[76]: 1700

In [77]:
```
# Total number of individuals who work more than 72 hours or less than 20 hours per week
len(df[(df["hours.per.week"] > 72) | (df["hours.per.week"] < 20)])
```

Out[77]: 2127

In [78]:
```
# Remove the outlier on the column
df = df[~((df["hours.per.week"] > 72) | (df["hours.per.week"] < 20))]
```

In [79]:
```
df.shape
```

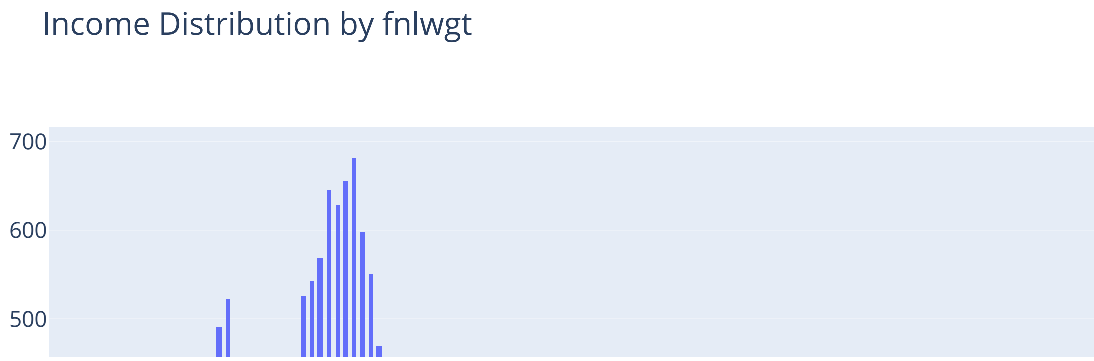
Out[79]: (30410, 14)

`fnlwgt` Column (final weigth)

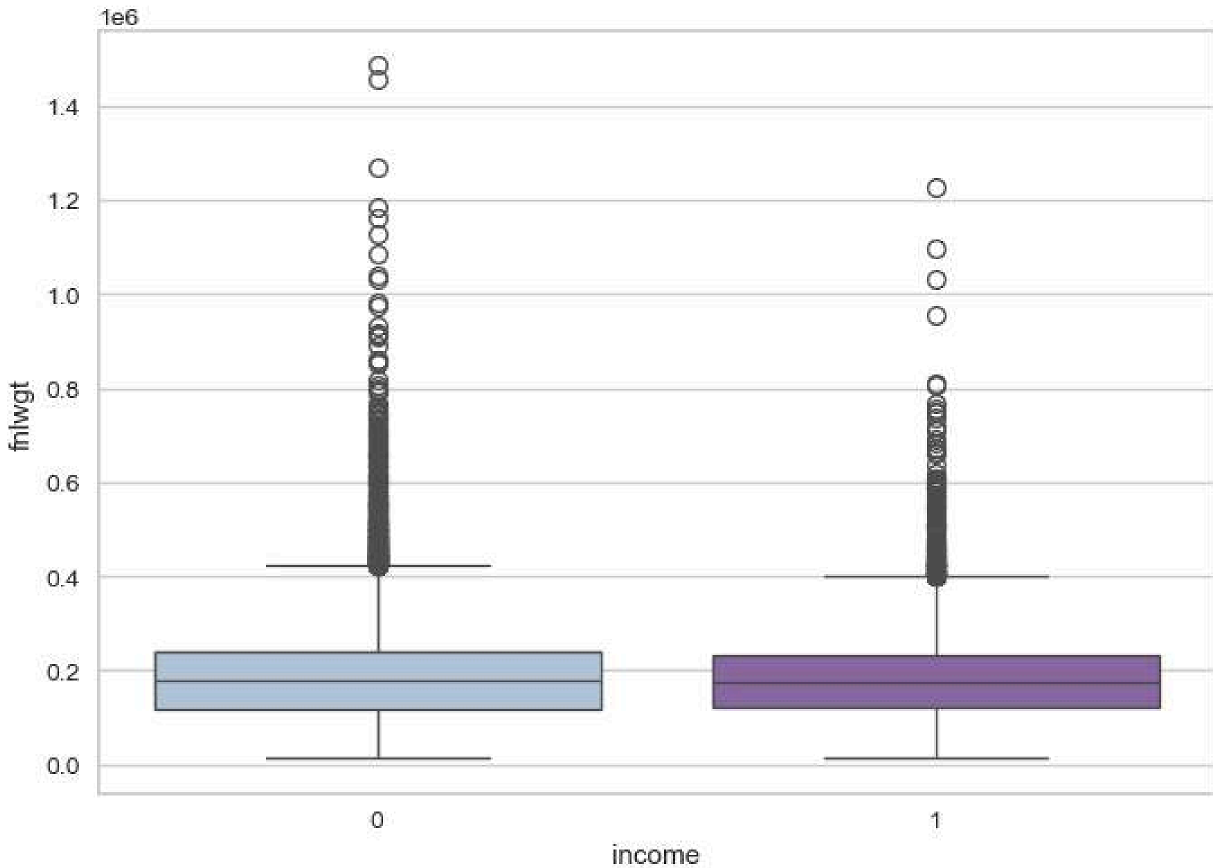
- The fnlwgt (final weight) column indicates the number of people the census estimates that each entry represents, helping to adjust the sample to more closely align with the overall population.
- Whether to drop this column depends on the context of the analysis.
- If the focus is on individual-level predictions, `fnlwgt` might add unnecessary noise, making it better to drop.
- If the model aims to reflect population-level outcomes or requires weighted statistics, keeping `fnlwgt` would be beneficial.
- In most individual prediction tasks, dropping `fnlwgt` can simplify the model without sacrificing accuracy.

For the purposes of this models, `fnlwgt` will be dropped to simplify the analysis and potentially improve model performance.

In [80]:
```
px.histogram(df, x='fnlwgt', color="income", barmode='group', title='Income Distribution by
```



In [81]:
```
sns.boxplot(data=df,y="fnlwgt",x='income', palette='BuPu');
```



In [82]:
```
# Drop the 'fnlwgt' column

df.drop(['fnlwgt'], axis = 1, inplace = True)
```

In [83]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 30410 entries, 0 to 32560
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   age             30410 non-null  int64 
 1   workclass       30410 non-null  object
 2   education       30410 non-null  object
 3   education.num   30410 non-null  int64 
 4   marital.status  30410 non-null  object
 5   occupation      30410 non-null  object
 6   relationship    30410 non-null  object
 7   race            30410 non-null  object
 8   sex             30410 non-null  object
 9   hours.per.week  30410 non-null  int64 
 10  native.country  30410 non-null  object
 11  income          30410 non-null  int64 
 12  capital_diff    30410 non-null  object
dtypes: int64(4), object(9)
memory usage: 3.2+ MB
```

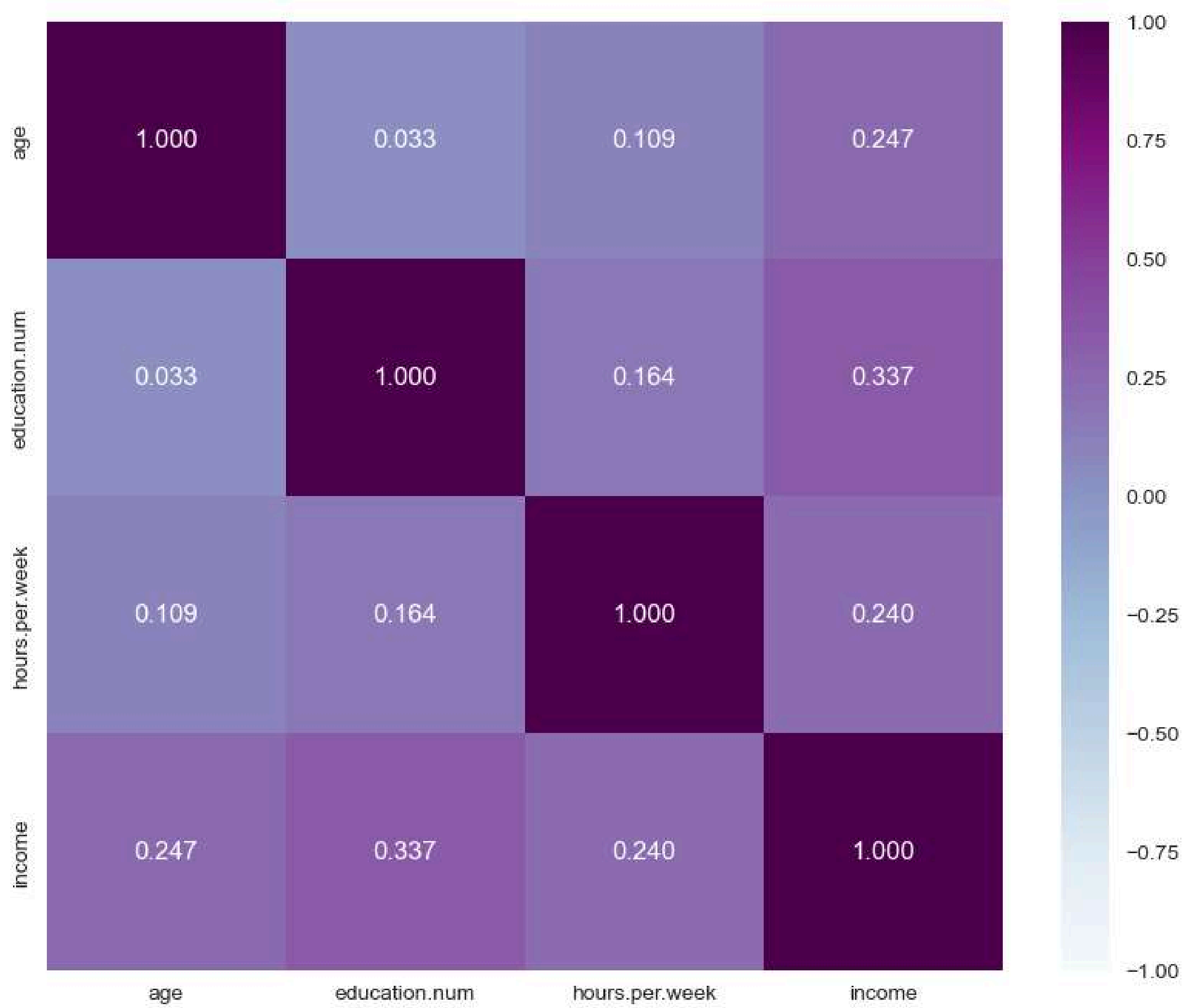
In [84]:
```
df.sample(3)
```

Out[84]:

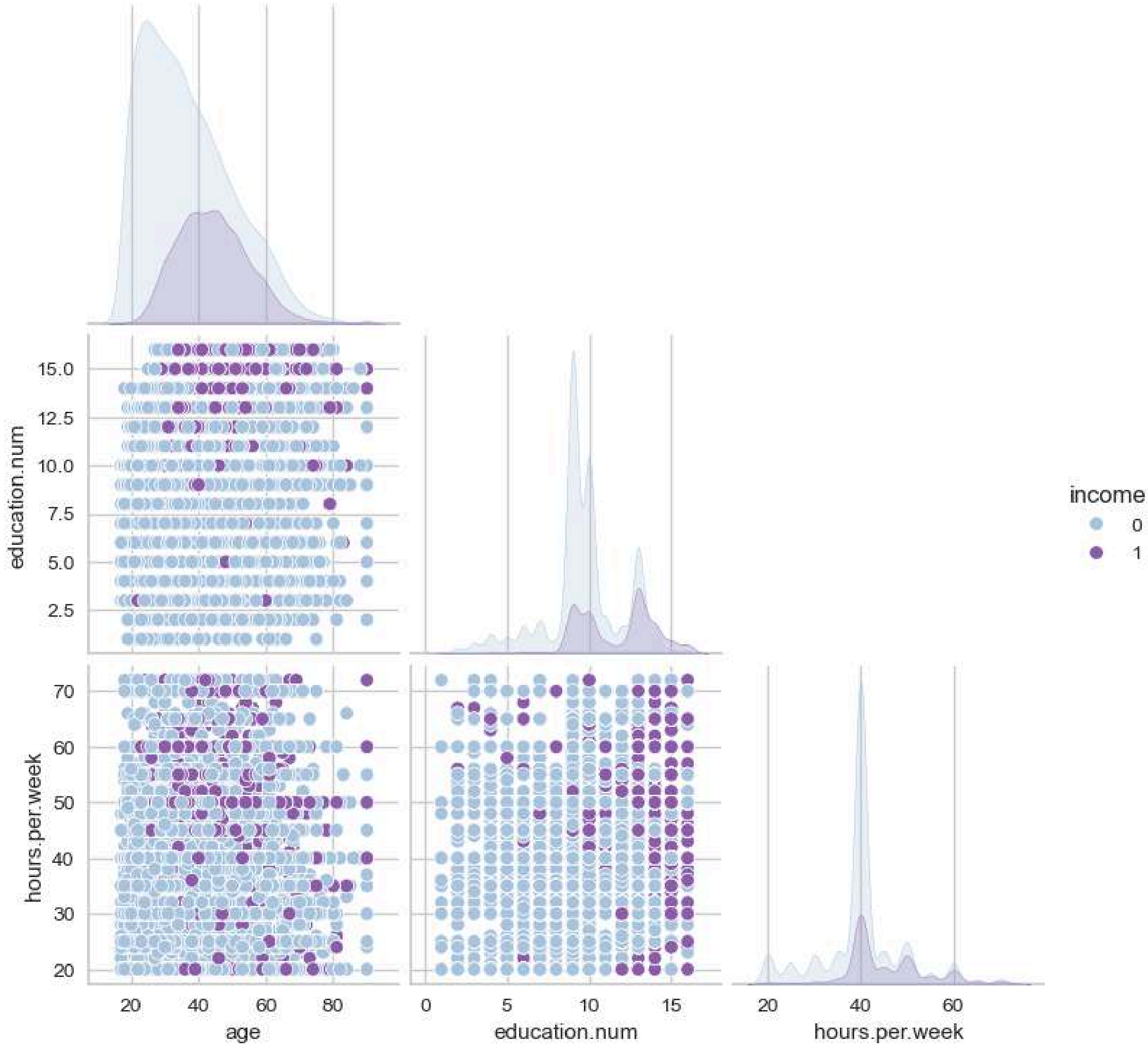
	age	workclass	education	education.num	marital.status	occupation	relationship	race	
26195	32	Local-gov	College	12	Divorced	Exec-managerial	Not-in-family	Black	Fe
322	34	Private	Bachelors	13	Never-married	Other-service	Not-in-family	White	
6507	28	Private	Primary	3	Never-married	Machine-op-inspct	Not-in-family	White	Fe

Correlations

In [85]:
```
plt.figure(figsize=(10,8))
sns.heatmap(df.select_dtypes("number").corr(), vmin = -1, vmax = 1, annot = True, fmt = '.3f
```



In [86]:
```
sns.pairplot(data=df, corner=True, hue='income',palette='BuPu');
```



In [87]:
```python
# Check Multicolinarty between features

def color_custom(val):
    if val > 0.90 and val < 0.99:
        color = 'red'
    elif val >= 1:
        color = 'blue'
    else:
        color = 'black'
    return f'color: {color}'

df.select_dtypes("number").corr().style.map(color_custom)
```

Out[87]:

	age	education.num	hours.per.week	income
age	1.000000	0.032813	0.109031	0.246707
education.num	0.032813	1.000000	0.164250	0.337060
hours.per.week	0.109031	0.164250	1.000000	0.239573
income	0.246707	0.337060	0.239573	1.000000

Correlation:

- The `income` feature has the highest positive correlation with `education.num` (0.335), indicating that higher education levels are moderately associated with higher income.
- Other features like `age`, and `hours.per.week` also show a positive but weaker correlation with income.
- Most features exhibit low correlation with each other, which suggests that multicollinearity is not a significant concern in this dataset.

Overall, the heatmap suggests that while some features like `education.num` is relevant to predicting income, multicollinearity is not a major issue in this dataset, making it easier to build a robust predictive model.

Outlier Analysis

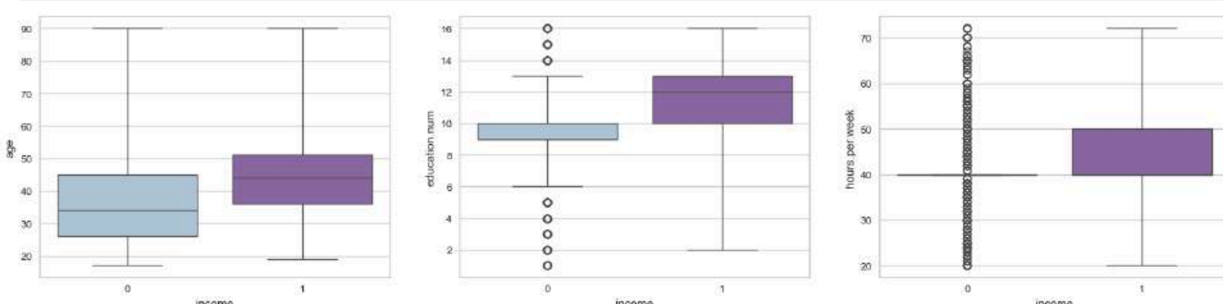
- In this study, Logistic Regression, SVM, and KNN models will be used.
- Outliers can significantly impact model performance, particularly for models like Logistic Regression, SVM, and KNN, which are sensitive to the scale and distribution of the data.
- Outliers may skew results and reduce model accuracy.
- Decision Trees, on the other hand, are generally more robust to outliers but still may lead to overfitting if not managed.
- Therefore, careful handling of outliers, such as using scaling or transformation techniques, is important to ensure reliable model performance.

Additionally, outliers handled during the analysis to ensure a more accurate representation of the data and to enhance model performance.

In [88]:
```
print(f"Income <= 50K (0) count: {income_less_50K}")
print(f"Income >  50K (1) count: {income_over_50K}")
```

```
Income <= 50K (0) count: 24698
Income >  50K (1) count: 7839
```

In [89]:
```
# Checking Outliers on Numerical Features by the Target // whis=3

index = 0
plt.figure(figsize=(20,15))
for feature in df.select_dtypes(include=['number']).columns:
    if feature != "income":
        index += 1
        plt.subplot(3,3,index)
        sns.boxplot(x='income',y=feature,data=df, whis=3, palette='BuPu')
plt.show()
```



Outlier Summary

1. **Age**: Individuals with higher income (`1`) tend to be slightly older on average compared to those with lower income (`0`), although the age ranges overlap significantly.
2. **Education**: There is a clear distinction in education levels (`education.num`) between the two income groups. Higher income earners tend to have significantly more years of education.
3. **Hours per Week**: Higher income earners (`1`) work more hours per week on average, with a wider range of working hours. Lower income earners (`0`) are concentrated around 40 hours per week, with fewer variations.

These insights suggest that age, education, and hours worked per week are all factors that differentiate income levels.

MACHINE LEARNING MODELS

Data Pre-Processing

In [90]:
```
# Updated Categorcal and Numerecal Features

cat_features = df.select_dtypes(include='object').columns
num_features = df.select_dtypes(include=['int64','float64']).columns

print('Categoricals:', list(cat_features))
print('---------------------------')
print('Numericals:',list(num_features))
```

```
Categoricals: ['workclass', 'education', 'marital.status', 'occupation', 'relationship', 'rac
e', 'sex', 'native.country', 'capital_diff']
---------------------------
Numericals: ['age', 'education.num', 'hours.per.week', 'income']
```

In [91]:
```
df.sample(3)
```

Out[91]:

	age	workclass	education	education.num	marital.status	occupation	relationship	race
1319	41	Private	Primary	3	Married-civ-spouse	Other-service	Husband	Others
21266	22	Private	College	10	Never-married	Adm-clerical	Own-child	Black
31892	52	Self-emp-not-inc	Doctorate	15	Married-civ-spouse	Prof-specialty	Husband	White

Splitting Data

In [92]:
```
X= df.drop(columns="income")
y= df.income
```

In [93]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_
```

Label Encoding and Scaling

In [94]:
```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder
```

In [96]:
```
onehot_categorics = ["workclass", "marital.status", "occupation", "relationship", "race", "s
ordinal_categorics = ["education", "capital_diff"]

column_transformed = make_column_transformer((OneHotEncoder(handle_unknown="ignore", sparse_
                                        (OrdinalEncoder(handle_unknown="use_encoded_value", u
                                        remainder=MinMaxScaler())
```

- **OneHotEncoder**: `handle_unknown="ignore"` is appropriate here because it avoids errors and simply doesn't create columns for unknown categories.
- **OrdinalEncoder**: It's better to use `handle_unknown="use_encoded_value"` with a specific `unknown_value` (e.g., `-1`) instead of ignoring the unknown categories. This way, the model can handle unseen categories in a controlled manner, rather than ignoring them entirely, which could lead to issues.
- **remainder=MinMaxScaler()**: To ensure that the remaining numerical columns are scaled to a range of 0-1.
- **make_column_transformer**: is being used to transform the columns.

In [97]:
```
# Fit train and Transform train-test data

X_train_trans = column_transformed.fit_transform(X_train)
X_test_trans = column_transformed.transform(X_test)
```

In [98]:
```
X_train_trans.shape, X_test_trans.shape
```

Out[98]:
```
((24328, 47), (6082, 47))
```

In [99]:
```
features = column_transformed.get_feature_names_out()
features
```

Out[99]:
```
array(['onehotencoder__workclass_Federal-gov',
       'onehotencoder__workclass_Local-gov',
       'onehotencoder__workclass_Never-worked',
       'onehotencoder__workclass_Private',
       'onehotencoder__workclass_Self-emp-inc',
       'onehotencoder__workclass_Self-emp-not-inc',
       'onehotencoder__workclass_State-gov',
       'onehotencoder__workclass_Without-pay',
       'onehotencoder__marital.status_Divorced',
       'onehotencoder__marital.status_Married-AF-spouse',
       'onehotencoder__marital.status_Married-civ-spouse',
       'onehotencoder__marital.status_Married-spouse-absent',
       'onehotencoder__marital.status_Never-married',
       'onehotencoder__marital.status_Separated',
       'onehotencoder__marital.status_Widowed',
       'onehotencoder__occupation_Adm-clerical',
       'onehotencoder__occupation_Armed-Forces',
       'onehotencoder__occupation_Craft-repair',
       'onehotencoder__occupation_Exec-managerial',
       'onehotencoder__occupation_Farming-fishing',
       'onehotencoder__occupation_Handlers-cleaners',
       'onehotencoder__occupation_Machine-op-inspct',
       'onehotencoder__occupation_Other-service',
       'onehotencoder__occupation_Priv-house-serv',
       'onehotencoder__occupation_Prof-specialty',
       'onehotencoder__occupation_Protective-serv',
       'onehotencoder__occupation_Sales',
       'onehotencoder__occupation_Tech-support',
       'onehotencoder__occupation_Transport-moving',
       'onehotencoder__relationship_Husband',
       'onehotencoder__relationship_Not-in-family',
       'onehotencoder__relationship_Other-relative',
       'onehotencoder__relationship_Own-child',
       'onehotencoder__relationship_Unmarried',
       'onehotencoder__relationship_Wife', 'onehotencoder__race_ Others',
       'onehotencoder__race_Black', 'onehotencoder__race_White',
       'onehotencoder__sex_Female', 'onehotencoder__sex_Male',
       'onehotencoder__native.country_Others',
       'onehotencoder__native.country_United-States',
       'ordinalencoder__education', 'ordinalencoder__capital_diff',
       'remainder__age', 'remainder__education.num',
       'remainder__hours.per.week'], dtype=object)
```

In [100…
```python
X_train= pd.DataFrame(X_train_trans, columns=features, index=X_train.index)
X_train.head()
```

Out[100…

	onehotencoder__workclass_Federal-gov	onehotencoder__workclass_Local-gov	onehotencoder__workclas
7378	0.0	0.0	
1937	0.0	0.0	
10749	0.0	0.0	
23929	0.0	0.0	
22481	0.0	0.0	

5 rows × 47 columns

In [101…

```
X_test= pd.DataFrame(X_test_trans, columns=features, index=X_test.index)
X_test.head()
```

Out[101…

	onehotencoder__workclass_Federal-gov	onehotencoder__workclass_Local-gov	onehotencoder__workclas
15234	0.0	0.0	
30963	0.0	1.0	
18499	0.0	0.0	
7790	0.0	0.0	
26879	0.0	0.0	

5 rows × 47 columns

In [102…

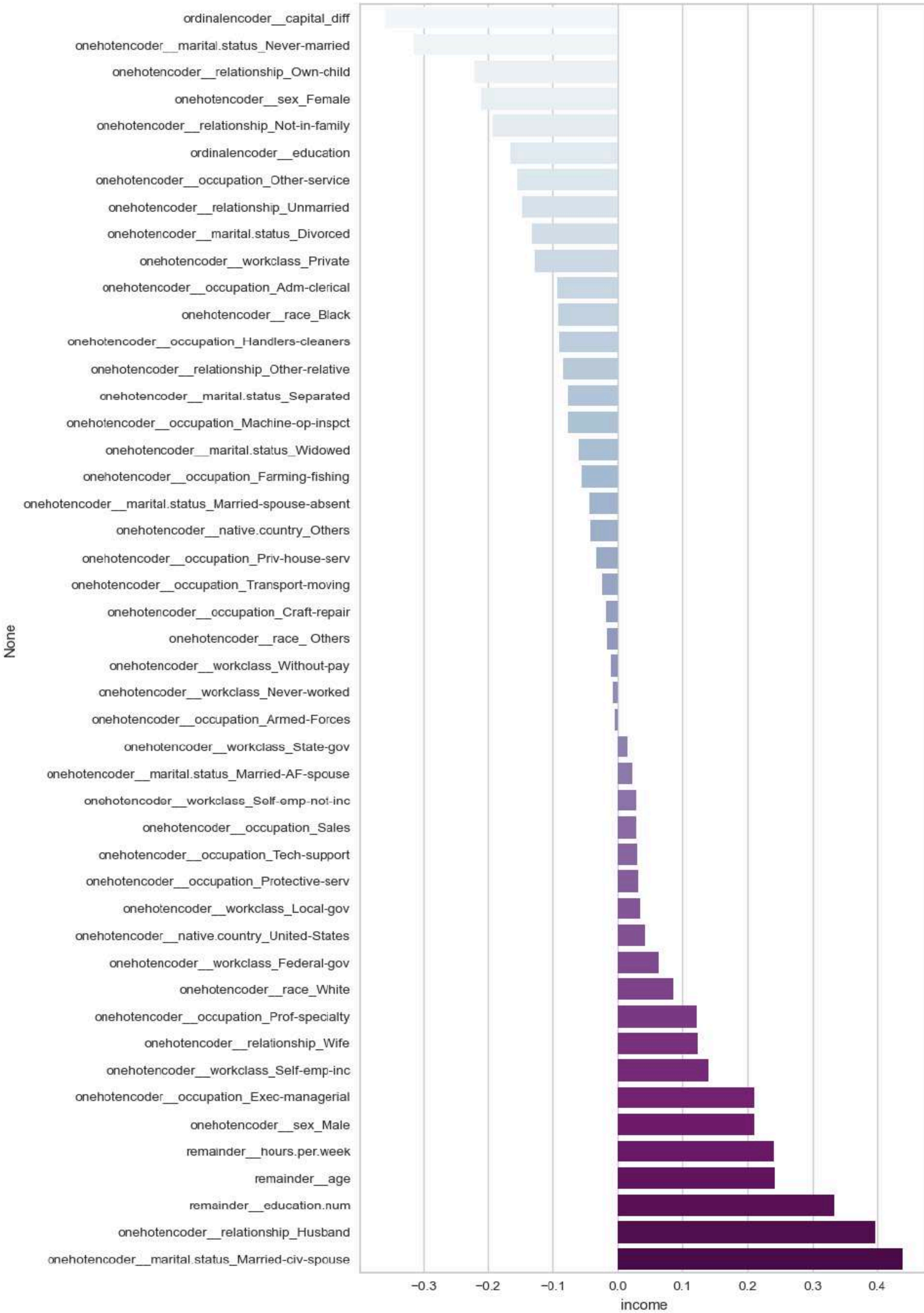
```
corr_by_income = X_train.join(y_train).corr()["income"].sort_values()[:-1]
corr_by_income
```

Out[102…

```
ordinalencoder__capital_diff                           -0.357570
onehotencoder__marital.status_Never-married            -0.314213
onehotencoder__relationship_Own-child                  -0.221005
onehotencoder__sex_Female                              -0.210939
onehotencoder__relationship_Not-in-family              -0.193319
ordinalencoder__education                              -0.164939
onehotencoder__occupation_Other-service                -0.154886
onehotencoder__relationship_Unmarried                  -0.147372
onehotencoder__marital.status_Divorced                 -0.132096
onehotencoder__workclass_Private                       -0.127657
onehotencoder__occupation_Adm-clerical                 -0.092074
onehotencoder__race_Black                              -0.090854
onehotencoder__occupation_Handlers-cleaners            -0.089431
onehotencoder__relationship_Other-relative             -0.083928
onehotencoder__marital.status_Separated                -0.076641
onehotencoder__occupation_Machine-op-inspct            -0.076483
onehotencoder__marital.status_Widowed                  -0.060090
onehotencoder__occupation_Farming-fishing              -0.055590
onehotencoder__marital.status_Married-spouse-absent    -0.043274
onehotencoder__native.country_Others                   -0.041719
onehotencoder__occupation_Priv-house-serv              -0.033910
onehotencoder__occupation_Transport-moving             -0.024402
onehotencoder__occupation_Craft-repair                 -0.017401
onehotencoder__race_ Others                            -0.016491
onehotencoder__workclass_Without-pay                   -0.009773
onehotencoder__workclass_Never-worked                  -0.007388
onehotencoder__occupation_Armed-Forces                 -0.004170
onehotencoder__workclass_State-gov                      0.015388
onehotencoder__marital.status_Married-AF-spouse         0.022290
onehotencoder__workclass_Self-emp-not-inc               0.028635
onehotencoder__occupation_Sales                         0.028951
onehotencoder__occupation_Tech-support                  0.030713
onehotencoder__occupation_Protective-serv               0.031129
onehotencoder__workclass_Local-gov                      0.034995
onehotencoder__native.country_United-States             0.041719
onehotencoder__workclass_Federal-gov                    0.061921
onehotencoder__race_White                               0.086126
onehotencoder__occupation_Prof-specialty                0.120956
onehotencoder__relationship_Wife                        0.123757
onehotencoder__workclass_Self-emp-inc                   0.139121
onehotencoder__occupation_Exec-managerial               0.210296
onehotencoder__sex_Male                                 0.210939
remainder__hours.per.week                               0.240863
remainder__age                                          0.243469
remainder__education.num                                0.334915
onehotencoder__relationship_Husband                     0.395761
onehotencoder__marital.status_Married-civ-spouse        0.439405
Name: income, dtype: float64
```

In [103…

```
plt.figure(figsize = (10,14))
sns.barplot(y = corr_by_income.index, x = corr_by_income,palette='BuPu')
plt.tight_layout();
```



Assessing the Importance of Features in Predicting Income:

- The most significant positive predictors of higher income include being "Married-civ-spouse" (married and living with a civilian spouse), the relationship status of "Husband", and higher values

in "education.num" (number of years of education) and "age".
- On the other hand, features such as being "Never-married" or having a low "capital_diff" (difference between capital gain and capital loss) are negatively associated with higher income. These insights highlight the key factors that the model considers important in distinguishing between income levels, with marital status and education being particularly influential.

Note:

- Capital Gain: The profit earned when an asset is sold for more than its purchase price.
- Capital Loss: The loss incurred when an asset is sold for less than its purchase price.
- *So, the difference between the two gives the net effect—whether ended up with an overall profit or loss from the transactions.*

Logistic Regression

In [104…

```
# Model Building, Scaling and Training Using a Pipeline

logistic_model = Pipeline([("scaler", MinMaxScaler()), ("logistic", LogisticRegression())])

logistic_model.fit(X_train, y_train)
```

Out[104…

```
Pipeline
    MinMaxScaler
    LogisticRegression
```

In [106…

```
# Prediction

y_pred=logistic_model.predict(X_test)
y_pred_proba = logistic_model.predict_proba(X_test)

log_f1 = f1_score(y_test, y_pred)
log_recall = recall_score(y_test, y_pred)
log_auc = roc_auc_score(y_test, y_pred)
```

Feature Importance

In [107…

```
# Get the coefficients
coefficients = logistic_model["logistic"].coef_[0]

feature_importances = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': coefficients
})

# Sort by importance
logistic_fi = feature_importances.sort_values(by='Importance', ascending=False)
logistic_fi.head(10)
```

Out[107…

	Feature	Importance
45	remainder__education.num	4.622300
44	remainder__age	1.906798
46	remainder__hours.per.week	1.820581
9	onehotencoder__marital.status_Married-AF-spouse	1.665009
34	onehotencoder__relationship_Wife	1.215140
10	onehotencoder__marital.status_Married-civ-spouse	0.905212
18	onehotencoder__occupation_Exec-managerial	0.784635
27	onehotencoder__occupation_Tech-support	0.686007
25	onehotencoder__occupation_Protective-serv	0.592064
0	onehotencoder__workclass_Federal-gov	0.453097

Evaluating the Logistic Model

In [108…

```
print(f"Income <= 50K (0) count: {income_less_50K}")
print(f"Income >  50K (1) count: {income_over_50K}")
```

```
Income <= 50K (0) count: 24698
Income >  50K (1) count: 7839
```

In [109…

```
# Evaluate the Model Performans

# Function to Evaluate the Model Performans using Classification Confusion_matrix()
# Also does the prediction in the function

def eval_metric(model, X_train, y_train, X_test, y_test, i):

    """ to get the metrics for the model """

    y_train_pred = model.predict(X_train)
    y_pred = model.predict(X_test)

    print(f"{i} Test_Set")
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
    print()
    print(f"{i} Train_Set")
    print(confusion_matrix(y_train, y_train_pred))
    print(classification_report(y_train, y_train_pred))
```

In [110…

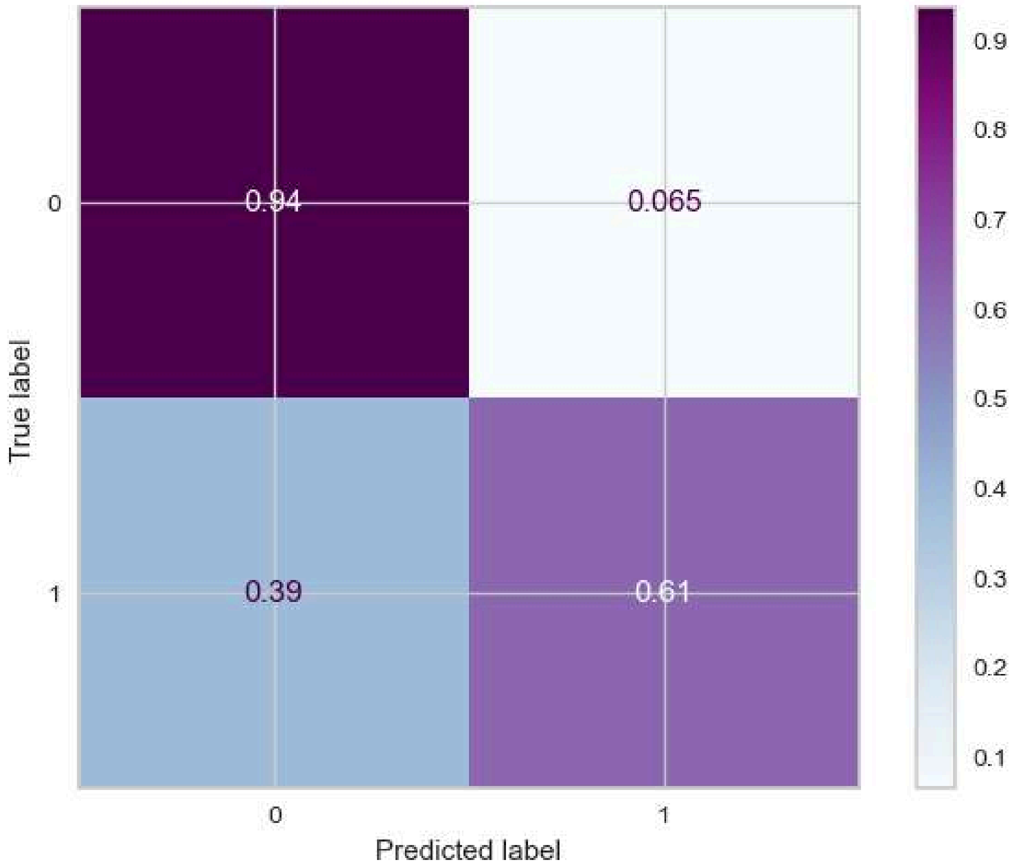
```
# Evaluating the Model Performance using Classification Metrics

eval_metric(logistic_model, X_train, y_train, X_test, y_test, 'logistic_model')
```

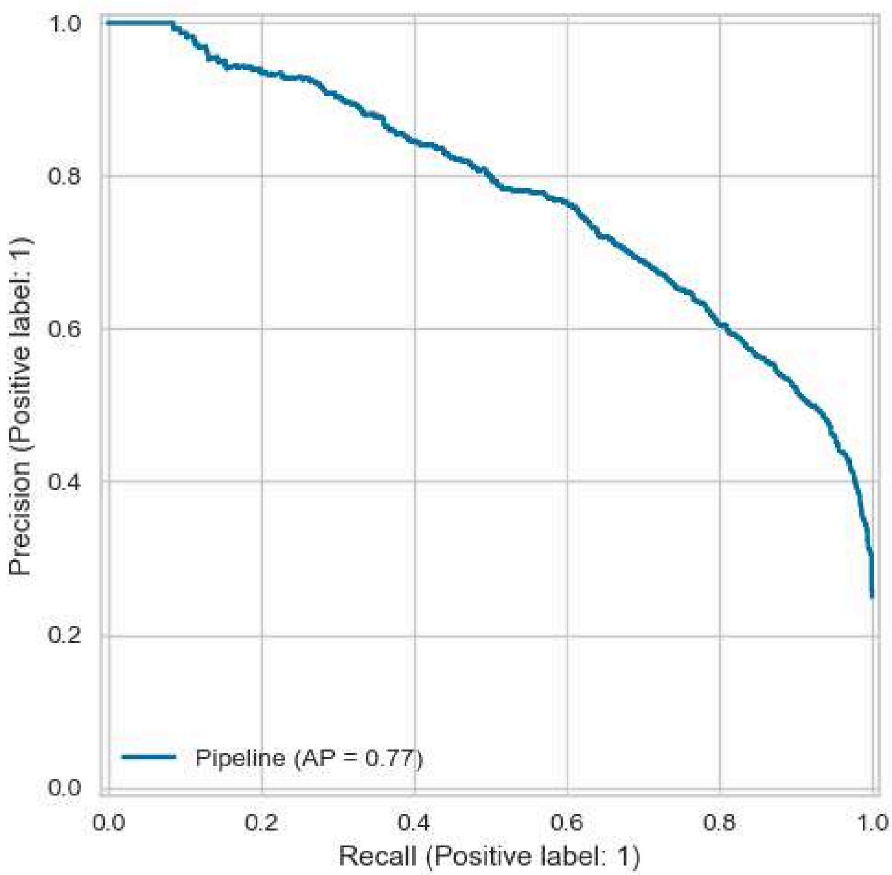
```
logistic_model Test_Set
[[4271  296]
 [ 587  928]]
              precision    recall  f1-score   support

           0       0.88      0.94      0.91      4567
           1       0.76      0.61      0.68      1515

    accuracy                           0.85      6082
   macro avg       0.82      0.77      0.79      6082
weighted avg       0.85      0.85      0.85      6082


logistic_model Train_Set
[[16990  1276]
 [ 2477  3585]]
              precision    recall  f1-score   support

           0       0.87      0.93      0.90     18266
           1       0.74      0.59      0.66      6062

    accuracy                           0.85     24328
   macro avg       0.81      0.76      0.78     24328
weighted avg       0.84      0.85      0.84     24328
```

In [111…

```
# Roc_AUC_score
print('logistic_model ROC_AUC Score:', roc_auc_score(y_test, y_pred_proba[:,1]))
print('---------------------------------------------------')

# Confusion Matrix
log_matrix = ConfusionMatrixDisplay.from_estimator(logistic_model, X_test, y_test, normalize

# Precision-Recall Curve
log_prCurve = PrecisionRecallDisplay.from_estimator(logistic_model, X_test, y_test)
```

```
logistic_model ROC_AUC Score: 0.904502236954591
---------------------------------------------------
```


0.94
0.065
0.39
0.61
True label
Predicted label
0
1
0
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1


Precision (Positive label: 1)
Recall (Positive label: 1)
Pipeline (AP = 0.77)
0.0
0.2
0.4
0.6
0.8
1.0

Model Validation

In [112…

```
# Cross Validation Scores of the Model Performance

model = Pipeline([("scaler", MinMaxScaler()), ("logistic", LogisticRegression())])

cv = StratifiedKFold(n_splits=10) # for unbalanced data validation

scores = cross_validate(model,
                        X_train,
                        y_train,
                        scoring=['accuracy', 'precision', 'recall', 'f1'],
                        cv=cv,
                        return_train_score=True)

df_scores = pd.DataFrame(scores, index=range(1, 11))
df_scores.mean()[2:]
```

Out[112…

```
test_accuracy      0.844664
train_accuracy     0.845377
test_precision     0.734822
train_precision    0.736641
test_recall        0.589573
train_recall       0.590619
test_f1            0.654016
train_f1           0.655595
dtype: float64
```

Hyperparameter Optimization

In [113…

```
logistic_model.get_params() #Parameters those are available for tuning for the model
```

Out[113…

```
{'memory': None,
 'steps': [('scaler', MinMaxScaler()), ('logistic', LogisticRegression())],
 'verbose': False,
 'scaler': MinMaxScaler(),
 'logistic': LogisticRegression(),
 'scaler__clip': False,
 'scaler__copy': True,
 'scaler__feature_range': (0, 1),
 'logistic__C': 1.0,
 'logistic__class_weight': None,
 'logistic__dual': False,
 'logistic__fit_intercept': True,
 'logistic__intercept_scaling': 1,
 'logistic__l1_ratio': None,
 'logistic__max_iter': 100,
 'logistic__multi_class': 'auto',
 'logistic__n_jobs': None,
 'logistic__penalty': 'l2',
 'logistic__random_state': None,
 'logistic__solver': 'lbfgs',
 'logistic__tol': 0.0001,
 'logistic__verbose': 0,
 'logistic__warm_start': False}
```

In [114…

```
# Hyperparameters Tuning with GridSearchSV

model = Pipeline([("scaler", MinMaxScaler()), ("logistic", LogisticRegression(max_iter = 100

# Define hyperparameters for tuning
penalty = ["l1", "l2"]       # Regularization terms: L1 (Lasso) and L2 (Ridge)
C  = [0.01, 0.1, 1]    # Regularization strength; inverse of regularization parameter
class_weight= ["balanced", None] # for unbalanced data


param_grid = [
    {
        "logistic__penalty" : ['l2', 'none'],
        "logistic__C" : C,
        "logistic__class_weight": class_weight,
        "logistic__solver": ['sag', 'lbfgs']
    },
    {
        "logistic__penalty" : ['l1', 'l2'],
        "logistic__C" : C,
        "logistic__class_weight": class_weight,
        "logistic__solver": ['liblinear', 'saga']
    }
]

cv = StratifiedKFold(n_splits = 5) # for unbalanced data

grid_model = GridSearchCV(model,
                          param_grid=param_grid,
                          cv=cv,
                          scoring = "recall",
                          n_jobs = -1, # Uses all available cores
                          verbose=1,
                          return_train_score=True).fit(X_train, y_train) # Returns training
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
```

In [115…

```
print('Best Params:', grid_model.best_params_)
print('Best Recall Score(test):', grid_model.best_score_)
print('Best Score Index:', grid_model.best_index_)
```

```
Best Params: {'logistic__C': 0.01, 'logistic__class_weight': 'balanced', 'logistic__penalty':
'l1', 'logistic__solver': 'saga'}
Best Recall Score(test): 0.8670377837453985
Best Score Index: 25
```

In [116…

```
# Checking overfiting with the GridSearch Cross-Val

pd.DataFrame(grid_model.cv_results_).loc[25, ["mean_test_score", "mean_train_score"]]

# The train and test scores are consistent, so we can say that there is no overfitting.
```

Out[116…

```
mean_test_score     0.867038
mean_train_score    0.868113
Name: 25, dtype: object
```

In [117…

```
# Prediction

y_pred=grid_model.predict(X_test)
y_pred_proba = grid_model.predict_proba(X_test)
```

```
log_grid_f1 = f1_score(y_test, y_pred)
log_grid_recall = recall_score(y_test, y_pred)
log_grid_auc = roc_auc_score(y_test, y_pred)
```

In [118…

```
# Checking the Incorrect Predictions

# Test Data df
test_data = pd.concat([X_test, y_test], axis=1)

# Create new column for 'predicted' classes to compore with actual target classes
test_data["pred"] = y_pred

# Filtering the wrong predicted obs
wrong_pred = test_data[((test_data["income"] == 0) & (test_data["pred"] == 1)) |
             ((test_data["income"] == 1) & (test_data["pred"] == 0))]

print('log_grid_model Total Incorrect Predictions:', wrong_pred.shape)


print('-----------------------------------------------')
# Actual-Predicted-Probalility of Pozitive Class(1)

my_dict = {"Actual": y_test, "Pred":y_pred, "Proba_1":y_pred_proba[:,1]}
pd.DataFrame.from_dict(my_dict).sample(10)
```

```
log_grid_model Total Incorrect Predictions: (1315, 49)
------------------------------------------------
```

Out[118…

	Actual	Pred	Proba_1
842	1	1	0.718746
31758	0	0	0.062813
7575	0	0	0.487752
3747	1	1	0.642698
11584	0	0	0.061448
10889	1	1	0.722025
12219	0	1	0.522838
14992	0	0	0.484244
2999	0	1	0.894391
20266	1	1	0.917002

Evaluating the Grid-Logistic Model

In [123…

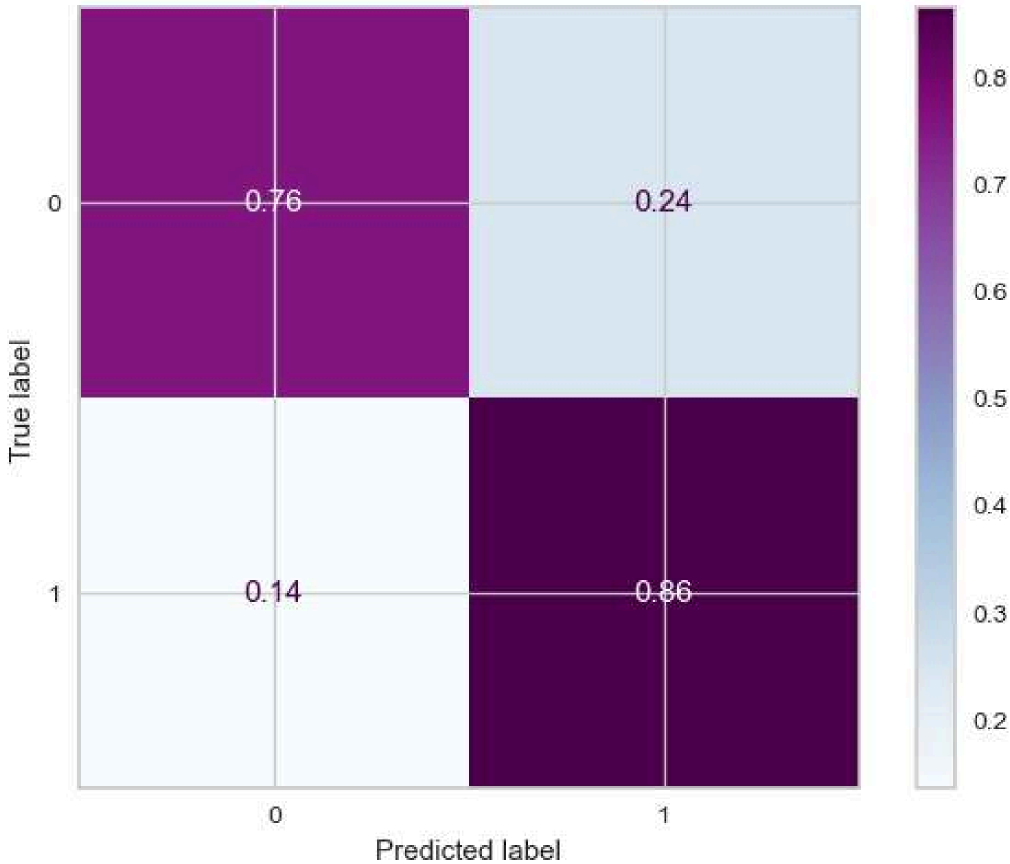
```
eval_metric(grid_model, X_train, y_train, X_test, y_test, 'log_grid_model')
```

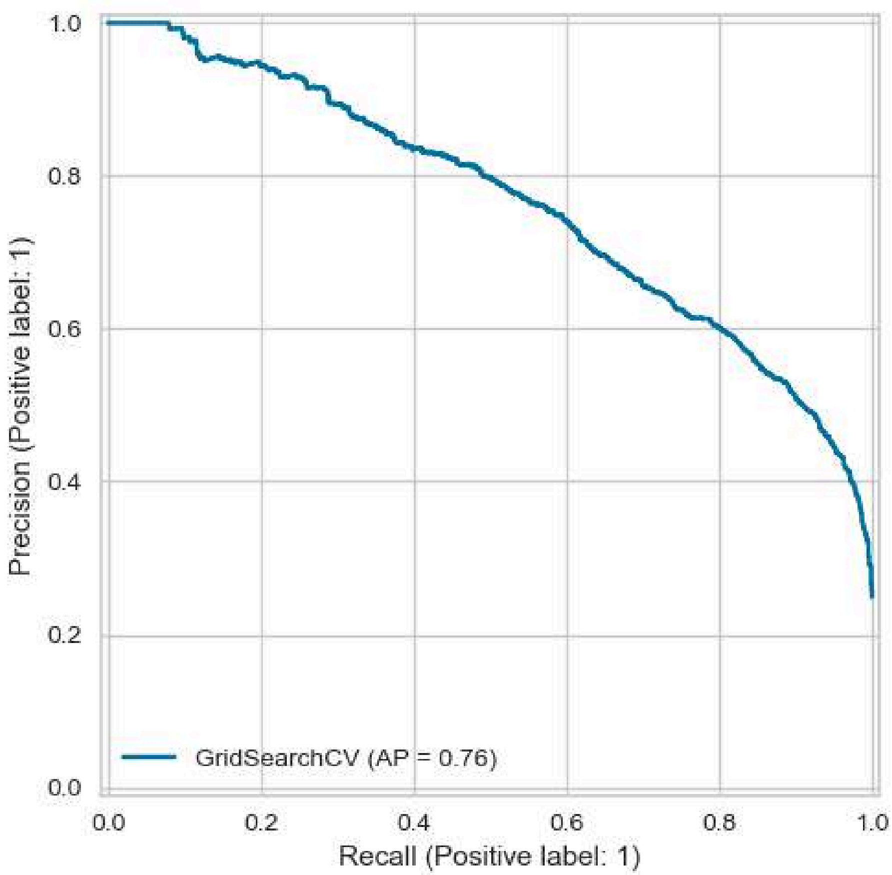
```
log_grid_model Test_Set
[[3458 1109]
 [ 206 1309]]
              precision    recall  f1-score   support

           0       0.94      0.76      0.84      4567
           1       0.54      0.86      0.67      1515

    accuracy                           0.78      6082
   macro avg       0.74      0.81      0.75      6082
weighted avg       0.84      0.78      0.80      6082


log_grid_model Train_Set
[[13750  4516]
 [  823  5239]]
              precision    recall  f1-score   support

           0       0.94      0.75      0.84     18266
           1       0.54      0.86      0.66      6062

    accuracy                           0.78     24328
   macro avg       0.74      0.81      0.75     24328
weighted avg       0.84      0.78      0.79     24328
```

In [120…

```
# Roc_AUC_score
print('log_grid_model ROC_AUC Score:', roc_auc_score(y_test, y_pred_proba[:,1]))
print('--------------------------------------------------')


# Confusion Matrix
grid_log_matrix = ConfusionMatrixDisplay.from_estimator(grid_model, X_test, y_test, normaliz

# Precision-Recall Curve
grid_log_prCurve = PrecisionRecallDisplay.from_estimator(grid_model, X_test, y_test)
```

```
log_grid_model ROC_AUC Score: 0.8983594461920463
--------------------------------------------------
```

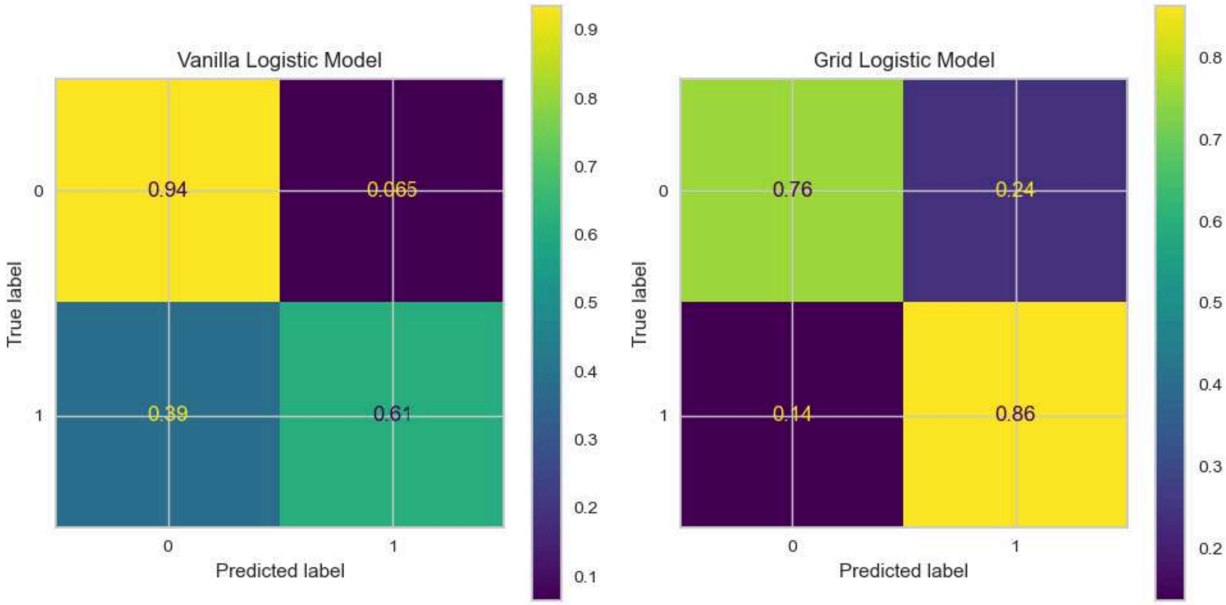




Comparing Vanilla and Grid Logistic Model

In [121…

```
# Confusion Matrix
fig, ax = plt.subplots(1, 2, figsize=(10,5))

log_matrix.plot(ax=ax[0])
ax[0].set_title("Vanilla Logistic Model")
grid_log_matrix.plot(ax=ax[1])
ax[1].set_title("Grid Logistic Model")

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

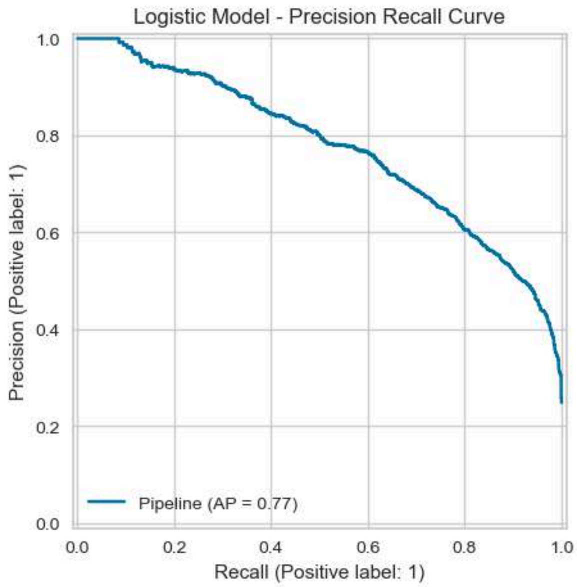


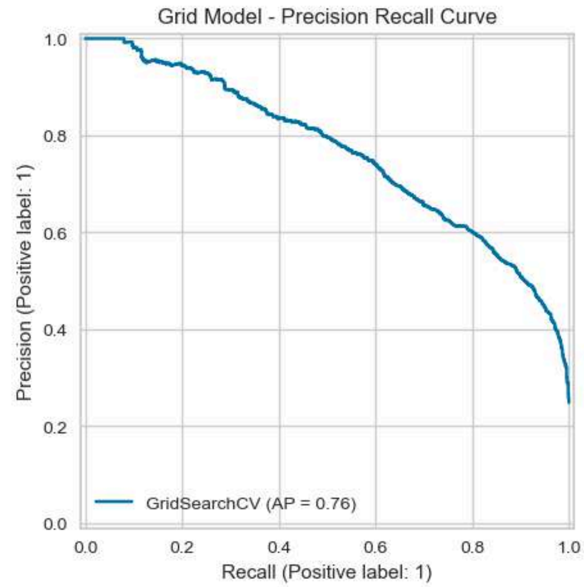
In [122…

```
#Precision-Recall Curves

fig, ax = plt.subplots(1, 2, figsize=(12, 5))

log_prCurve.plot(ax=ax[0])
ax[0].set_title("Logistic Model - Precision Recall Curve")
grid_log_prCurve.plot(ax=ax[1])
ax[1].set_title("Grid Model - Precision Recall Curve")
```

Out[122…

```
Text(0.5, 1.0, 'Grid Model - Precision Recall Curve')
```





Vanilla Logistic Model and the Grid Logistic Model:

1. **Confusion Matrix**:
   - **Vanilla Logistic Model**: Higher accuracy for the negative class (0.94) but lower recall for the positive class (0.61).
   - **Grid Logistic Model**: Better recall for the positive class (0.86) but sacrifices accuracy for the negative class (0.76).
2. **Precision-Recall Curve**: (Unbalanced Data)
   - **Vanilla Logistic Model**: Slightly higher average precision (AP = 0.77), indicating better overall balance between precision and recall.
   - **Grid Logistic Model**: Lower average precision (AP = 0.76), suggesting that hyperparameter tuning did not significantly improve model performance.

In summary, the Vanilla Logistic Model offers a more balanced performance, while the Grid Logistic Model focuses on improving recall for the positive class at the expense of negative class accuracy.

Support Vector Machine

In [124…

```
# Set and scale
svm_model = Pipeline([("scaler", MinMaxScaler()), ("SVC", SVC(probability=True))])

#Fit the model
svm_model.fit(X_train, y_train)

# Prediction
y_pred=svm_model.predict(X_test)
```

Evaluating The Model Performance

In [126…

```
# Evaluating the Model Performance using Classification Metrics
```

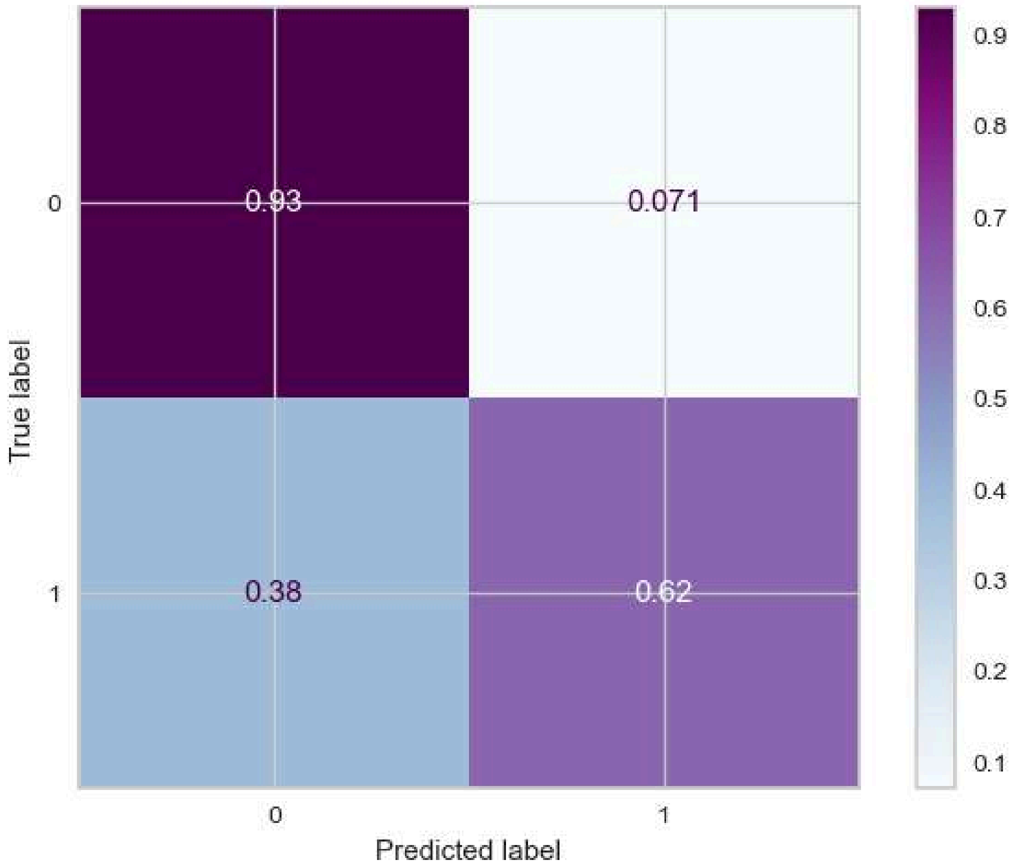
```
eval_metric(svm_model, X_train, y_train, X_test, y_test, 'svm_model')
```

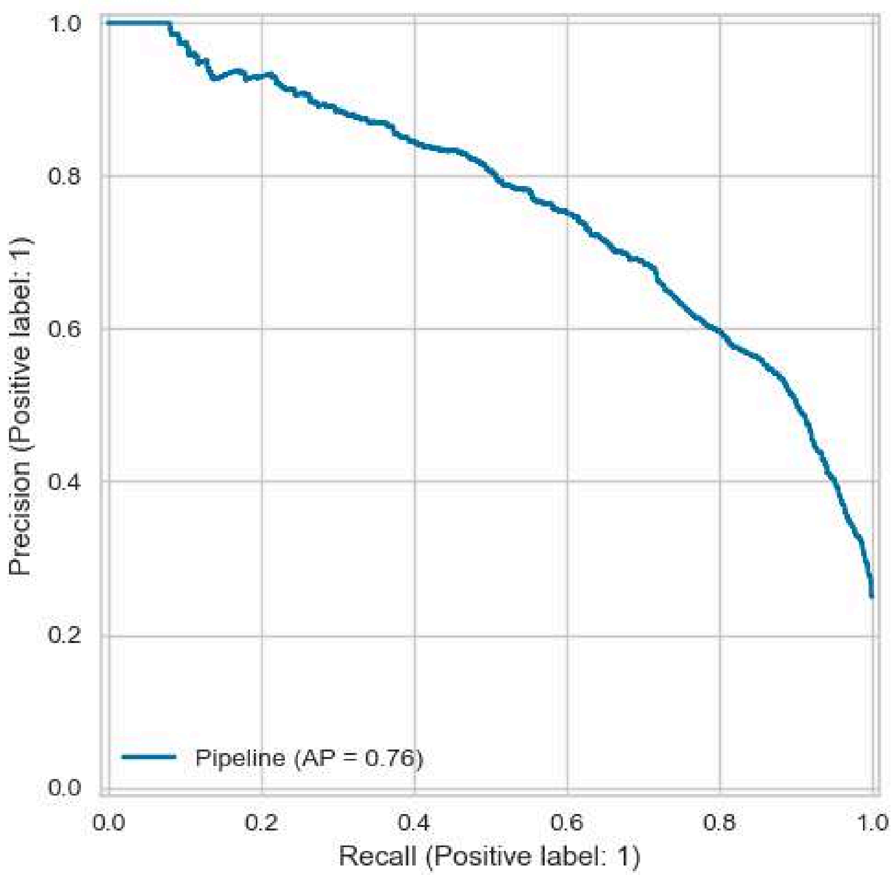
```
svm_model Test_Set
[[4243  324]
 [ 582  933]]
              precision    recall  f1-score   support

           0       0.88      0.93      0.90      4567
           1       0.74      0.62      0.67      1515

    accuracy                           0.85      6082
   macro avg       0.81      0.77      0.79      6082
weighted avg       0.85      0.85      0.85      6082


svm_model Train_Set
[[16977  1289]
 [ 2472  3590]]
              precision    recall  f1-score   support

           0       0.87      0.93      0.90     18266
           1       0.74      0.59      0.66      6062

    accuracy                           0.85     24328
   macro avg       0.80      0.76      0.78     24328
weighted avg       0.84      0.85      0.84     24328
```

In [125…

```
# Roc_AUC_score
print('svm_model ROC_AUC Score:', roc_auc_score(y_test, y_pred_proba[:,1]))
print('--------------------------------------------------')

# Confusion Matrix
svm_matrix = ConfusionMatrixDisplay.from_estimator(svm_model, X_test,y_test, normalize='true

# Precision-Recall Curve
svm_prCurve = PrecisionRecallDisplay.from_estimator(svm_model, X_test, y_test)
```

```
svm_model ROC_AUC Score: 0.8983594461920463
--------------------------------------------------
```


0.93
0.071
0.38
0.62
True label
Predicted label
0
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1


Pipeline (AP = 0.76)
Precision (Positive label: 1)
Recall (Positive label: 1)
0.0
0.2
0.4
0.6
0.8
1.0

Model Validation

In [127…

```
# Cross Validation Scores of the Model Performance

model = Pipeline([("scaler", MinMaxScaler()), ("SVC", SVC())])

cv = StratifiedKFold(n_splits=5) # for unbalanced data validation

scores = cross_validate(model,
                        X_train,
                        y_train,
                        scoring=['accuracy', 'precision', 'recall', 'f1'],
                        cv=cv,
                        return_train_score=True)

df_scores = pd.DataFrame(scores, index=range(1, 6))
df_scores.mean()[2:]
```

Out[127…

```
test_accuracy      0.842815
train_accuracy     0.845641
test_precision     0.729220
train_precision    0.736658
test_recall        0.587264
train_recall       0.592338
test_f1            0.650547
train_f1           0.656617
dtype: float64
```

Hyperparameter Optimization for SVM Model

In [128…

```
svm_model.get_params() #Parameters those are available for tuning for the model
```

Out[128…

```
{'memory': None,
 'steps': [('scaler', MinMaxScaler()), ('SVC', SVC(probability=True))],
 'verbose': False,
 'scaler': MinMaxScaler(),
 'SVC': SVC(probability=True),
 'scaler__clip': False,
 'scaler__copy': True,
 'scaler__feature_range': (0, 1),
 'SVC__C': 1.0,
 'SVC__break_ties': False,
 'SVC__cache_size': 200,
 'SVC__class_weight': None,
 'SVC__coef0': 0.0,
 'SVC__decision_function_shape': 'ovr',
 'SVC__degree': 3,
 'SVC__gamma': 'scale',
 'SVC__kernel': 'rbf',
 'SVC__max_iter': -1,
 'SVC__probability': True,
 'SVC__random_state': None,
 'SVC__shrinking': True,
 'SVC__tol': 0.001,
 'SVC__verbose': False}
```

In [130…

```
# Hyperparameters Tuning with GridSearchSV

model = Pipeline([("scaler", MinMaxScaler()), ("SVC", SVC(class_weight="balanced"))])


param_grid = {"SVC__C":[0.5,1],
             "SVC__gamma":["scale","auto", 0.1,0.3],
             "SVC__kernel":["rbf", "linear"]}

cv = StratifiedKFold(n_splits = 5) # for unbalanced data

svm_grid_model = GridSearchCV(model,
                              param_grid=param_grid,
                              cv=cv,
                              scoring = "recall_macro",
                              n_jobs = -1, # Uses all available cores
                              verbose=1,
                              return_train_score=True).fit(X_train, y_train) # fit the model
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
```

In [131…

```
print('Best Params:', svm_grid_model.best_params_)
print('Best Recall Score(test):', svm_grid_model.best_score_)
print('Best Score Index:', svm_grid_model.best_index_)
```

```
Best Params: {'SVC__C': 1, 'SVC__gamma': 'scale', 'SVC__kernel': 'rbf'}
Best Recall Score(test): 0.8119453688234672
Best Score Index: 8
```

In [133…

```
# Checking overfiting with the CV scores

pd.DataFrame(svm_grid_model.cv_results_).loc[8, ["mean_test_score", "mean_train_score"]]
```

Out[133…

```
mean_test_score     0.811945
mean_train_score    0.820518
Name: 8, dtype: object
```

In [135…

```
# Prediction

y_pred=svm_grid_model.predict(X_test)
decision_fonc = svm_grid_model.decision_function(X_test)
# In an SVM model with probability=True, predict_proba uses the decision function's output,

svm_grid_f1 = f1_score(y_test, y_pred)
svm_grid_recall = recall_score(y_test, y_pred)
svm_grid_auc = roc_auc_score(y_test, y_pred)
```

In [136…

```
# Checking the Incorrect Predictions

# Test Data df
test_data = pd.concat([X_test, y_test], axis=1)

# Create new column for 'predicted' classes to compore with actual target classes
test_data["pred"] = y_pred

# Filtering the wrong predicted obs
wrong_pred = test_data[((test_data["income"] == 0) & (test_data["pred"] == 1)) |
             ((test_data["income"] == 1) & (test_data["pred"] == 0))]

print('svm_grid_model Total Incorrect Predictions:', wrong_pred.shape)
```

```
print('--------------------------------------------------')
# Actual-Predicted-Probalility of Pozitive Class(1)

my_dict = {"Actual": y_test, "Pred":y_pred, "Proba_1":y_pred_proba[:,1]}
pd.DataFrame.from_dict(my_dict).sample(10)
```

```
svm_grid_model Total Incorrect Predictions: (1241, 49)
--------------------------------------------------
```

Out[136…

	Actual	Pred	Proba_1
30761	0	0	0.026332
30135	0	0	0.342146
28629	0	0	0.063687
23626	0	0	0.156667
20081	0	0	0.120072
20292	0	0	0.679350
6352	0	0	0.124349
20022	1	1	0.889762
8576	1	1	0.826103
25913	0	0	0.111561

Evaluating the SVM_Grid Model

In [137…

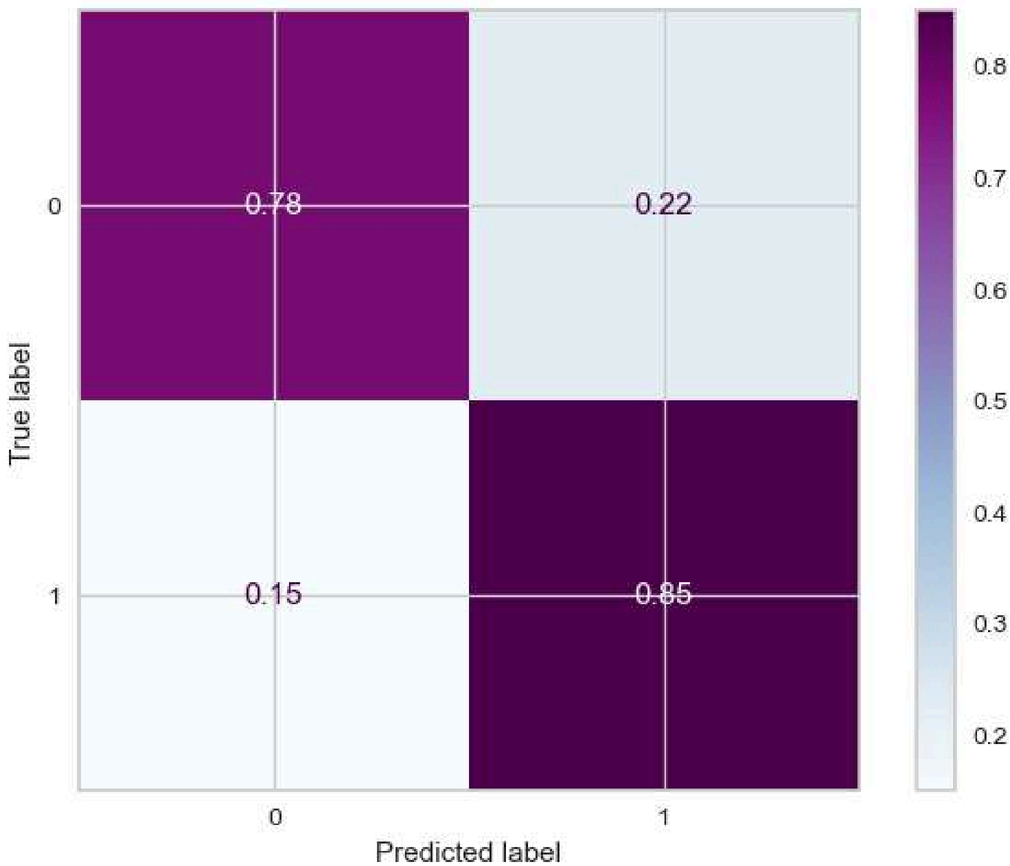
```
eval_metric(svm_grid_model, X_train, y_train, X_test, y_test, 'svm_grid_model')
```

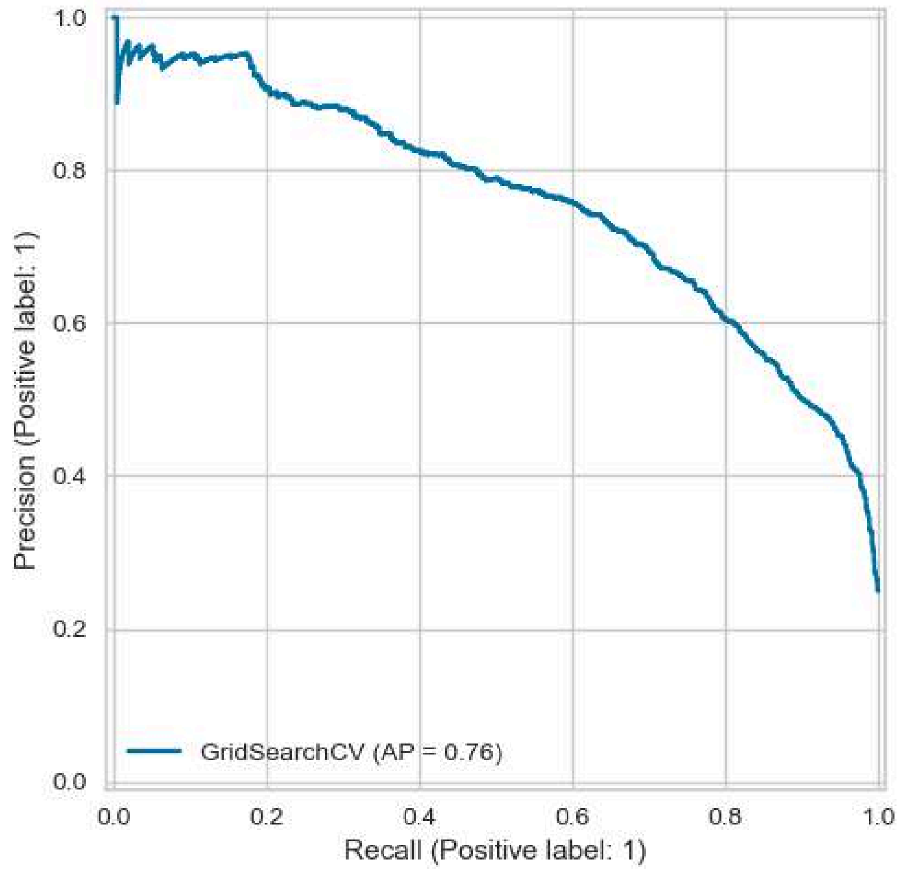
```
svm_grid_model Test_Set
[[3554 1013]
 [ 228 1287]]
              precision    recall  f1-score   support

           0       0.94      0.78      0.85      4567
           1       0.56      0.85      0.67      1515

    accuracy                           0.80      6082
   macro avg       0.75      0.81      0.76      6082
weighted avg       0.85      0.80      0.81      6082


svm_grid_model Train_Set
[[14179  4087]
 [  815  5247]]
              precision    recall  f1-score   support

           0       0.95      0.78      0.85     18266
           1       0.56      0.87      0.68      6062

    accuracy                           0.80     24328
   macro avg       0.75      0.82      0.77     24328
weighted avg       0.85      0.80      0.81     24328
```

In [138…

```
# Roc_AUC_score
print('svm_grid_model ROC_AUC Score:', roc_auc_score(y_test, decision_fonc))
print('--------------------------------------------------')

# Confusion Matrix
svm_grid_matrix = ConfusionMatrixDisplay.from_estimator(svm_grid_model, X_test, y_test, norm

# Precision-Recall Curve
svm_grid_prCurve = PrecisionRecallDisplay.from_estimator(svm_grid_model, X_test, y_test)
```

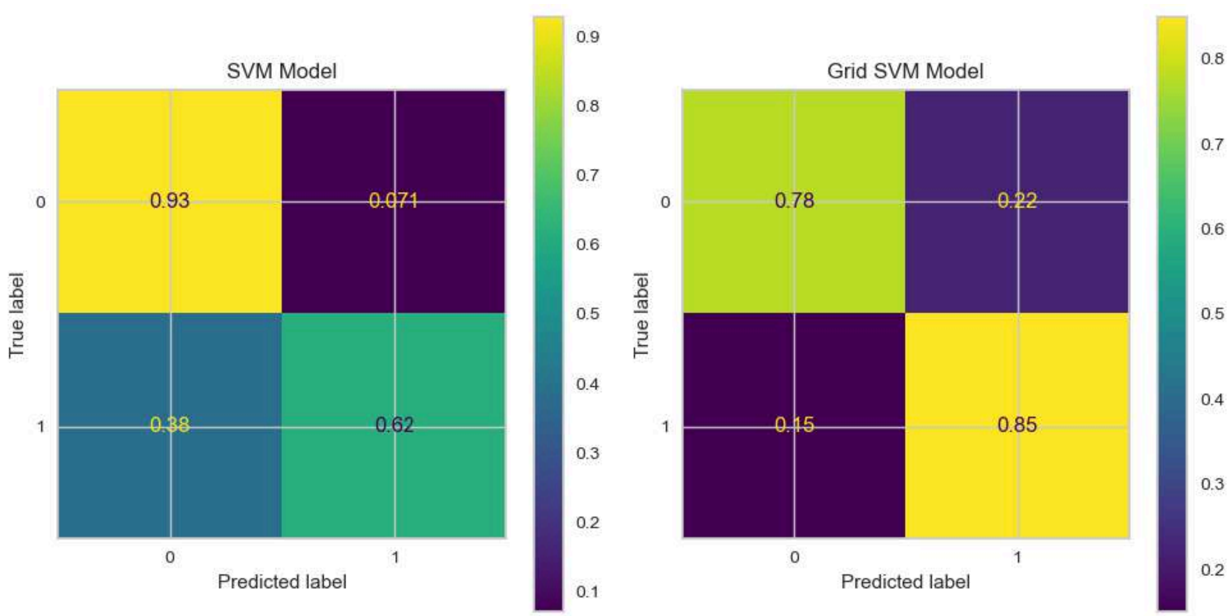
```
svm_grid_model ROC_AUC Score: 0.8997057380360326
--------------------------------------------------
```





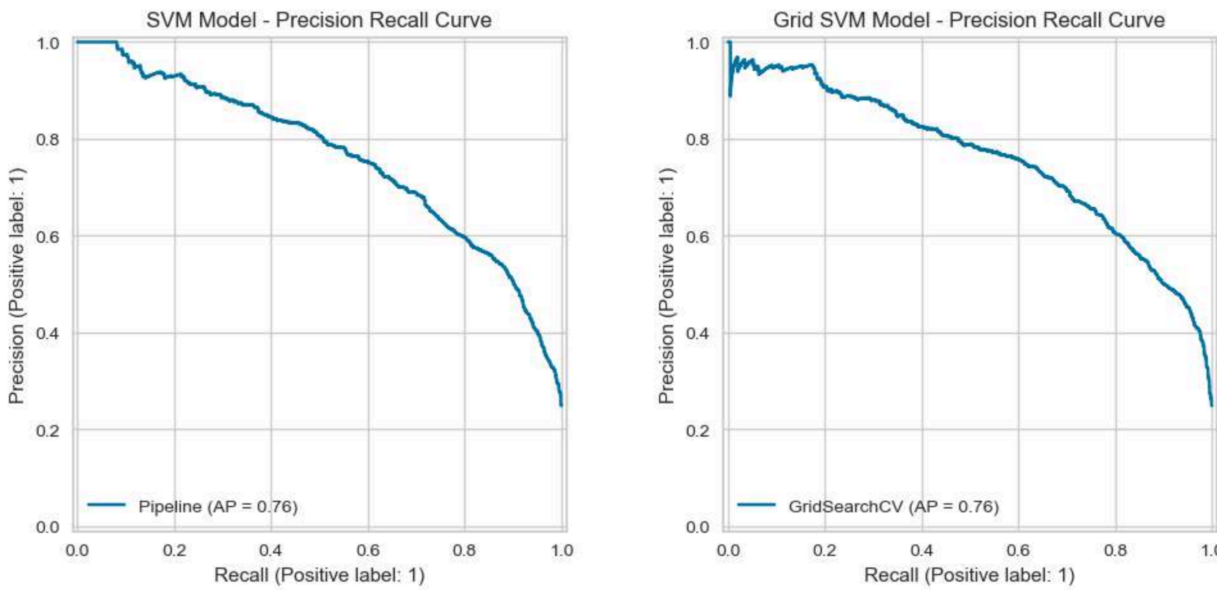
In [139…

```
fig, ax = plt.subplots(1, 2, figsize=(10,5))

svm_matrix.plot(ax=ax[0])
ax[0].set_title("SVM Model")
svm_grid_matrix.plot(ax=ax[1])
ax[1].set_title("Grid SVM Model")

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



In [140…
```
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

svm_prCurve.plot(ax=ax[0])
ax[0].set_title("SVM Model - Precision Recall Curve")
svm_grid_prCurve.plot(ax=ax[1])
ax[1].set_title("Grid SVM Model - Precision Recall Curve")
```

Out[140…
```
Text(0.5, 1.0, 'Grid SVM Model - Precision Recall Curve')
```



SVM Models:

- The SVM Model and the Grid SVM Model show very similar performance.
- Both models have high recall for the negative class (0.93) but lower recall for the positive class (around 0.61-0.62).
- The Precision-Recall curves also indicate similar average precision (AP).
- Overall, hyperparameter tuning with GridSearch improved the recall.

K-Nearest Neighbours (KNN)

In [141…
```
knn_model = Pipeline([("scaler", MinMaxScaler()),("knn", KNeighborsClassifier())])

knn_model.fit(X_train, y_train)
```

Out[141…
```
Pipeline
  MinMaxScaler
  KNeighborsClassifier
```

In [142…
```
# Prediction
y_pred = knn_model.predict(X_test)
y_pred_proba = knn_model.predict_proba(X_test)

# Scores to compare the models at the end.
```

```
knn_f1 = f1_score(y_test, y_pred)
knn_recall = recall_score(y_test, y_pred)
knn_auc = roc_auc_score(y_test, y_pred)
```

Evaluating The Model Performance

In [144…

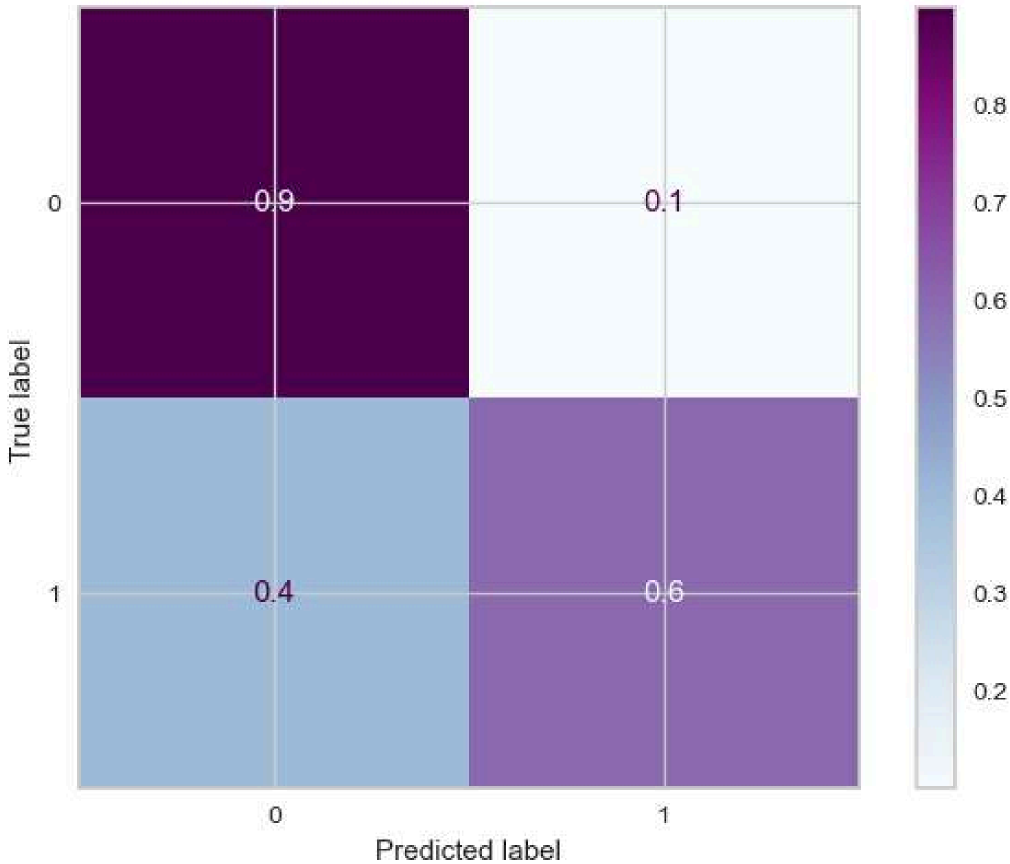
```
# Evaluating the Model Performance using Classification Metrics

eval_metric(knn_model, X_train, y_train, X_test, y_test, 'knn_model')
```

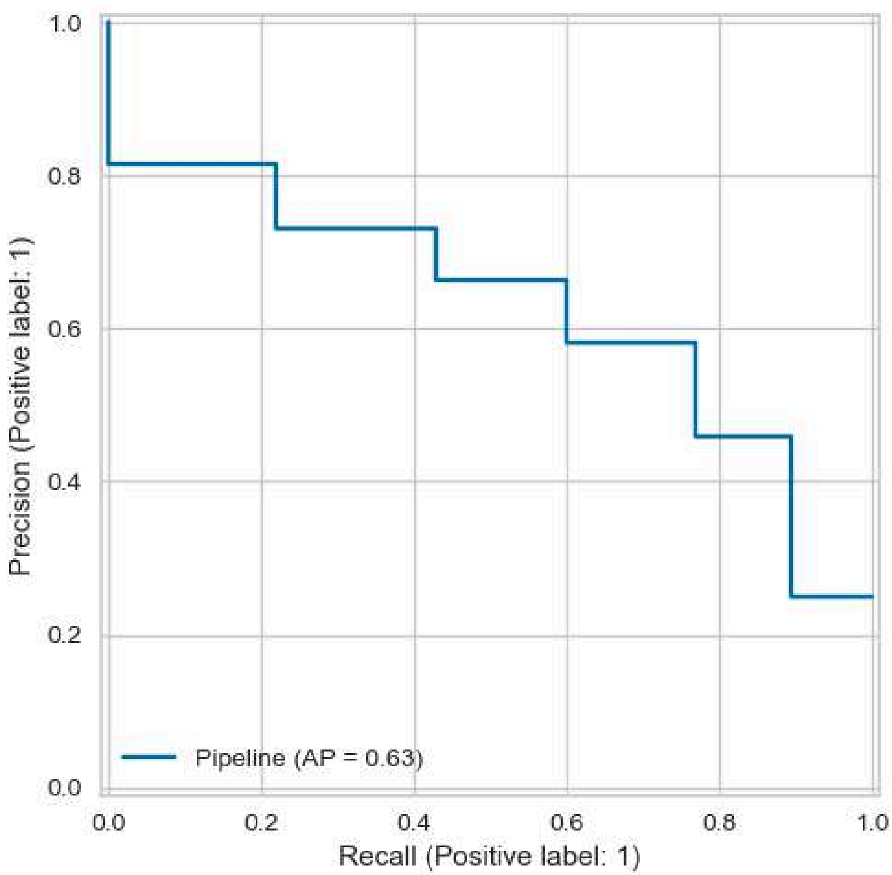
```
knn_model Test_Set
[[4106  461]
 [ 606  909]]
              precision    recall  f1-score   support

           0       0.87      0.90      0.89      4567
           1       0.66      0.60      0.63      1515

    accuracy                           0.82      6082
   macro avg       0.77      0.75      0.76      6082
weighted avg       0.82      0.82      0.82      6082


knn_model Train_Set
[[17079  1187]
 [ 1839  4223]]
              precision    recall  f1-score   support

           0       0.90      0.94      0.92     18266
           1       0.78      0.70      0.74      6062

    accuracy                           0.88     24328
   macro avg       0.84      0.82      0.83     24328
weighted avg       0.87      0.88      0.87     24328
```

In [145…

```
# Roc_AUC_score
print('knn_model ROC_AUC Score:', roc_auc_score(y_test, y_pred_proba[:,1]))
print('--------------------------------------------------')

# Confusion Matrix
knn_matrix = ConfusionMatrixDisplay.from_estimator(knn_model, X_test,y_test, normalize='true

# Precision-Recall Curve
knn_prCurve = PrecisionRecallDisplay.from_estimator(knn_model, X_test, y_test)
```

```
knn_model ROC_AUC Score: 0.8491127698274537
--------------------------------------------------
```





Elbow Method for Choosing Reasonable K-Values

In [146…

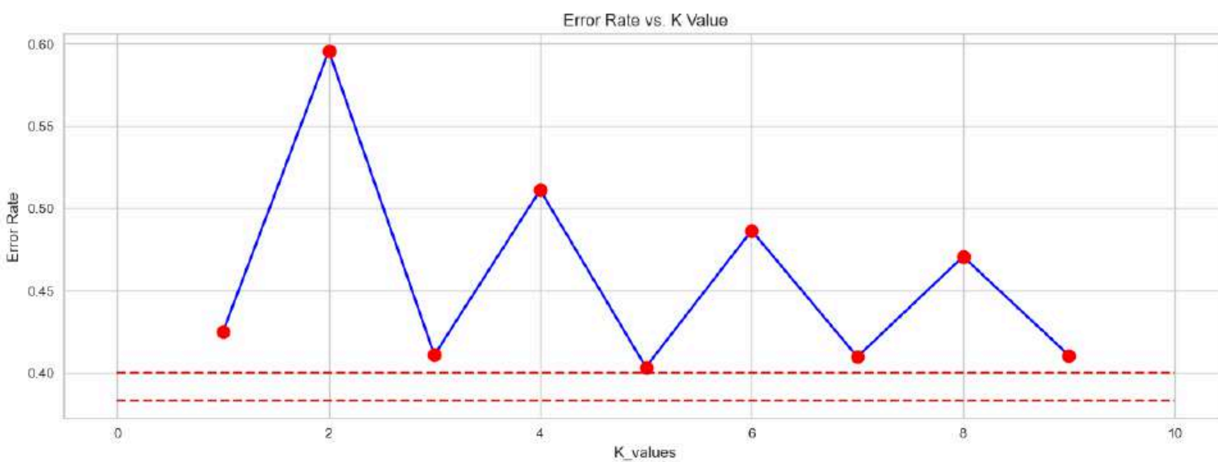
```
test_error_rates = []

for k in range(1,10):

    knn_model = Pipeline([("scaler", MinMaxScaler()), ("knn", KNeighborsClassifier(n_neighbo

    scores = cross_validate(knn_model, X_train, y_train, scoring = ['recall'], cv = 10)

    recall_mean = scores["test_recall"].mean()

    test_error = 1 - recall_mean

    test_error_rates.append(test_error)
```

In [173…

```
# Error Rate vs. K Value

plt.figure(figsize=(15,5))
plt.plot(range(1,10), test_error_rates, color='blue', marker='o',
         markerfacecolor='red', markersize=10)

plt.title('Error Rate vs. K Value')
plt.xlabel('K_values')
plt.ylabel('Error Rate')
plt.hlines(y=0.4, xmin = 0, xmax = 10, colors= 'r', linestyles="--")
plt.hlines(y=0.383, xmin = 0, xmax = 10, colors= 'r', linestyles="--")
```

Out[173…

```
<matplotlib.collections.LineCollection at 0x1b108792660>
```



Overfiting and Underfiting Control for K-Values

In [174…

```
test_error_rates = []
train_error_rates = []

for k in range(1,10):

    knn_model = Pipeline([("scaler", MinMaxScaler()), ("knn", KNeighborsClassifier(n_neighbo

    knn_model.fit(X_train,y_train)

    scores = cross_validate(knn_model, X_train, y_train, scoring = ['recall'], cv = 10, retu

    recall_test_mean = scores["test_recall"].mean()
    recall_train_mean = scores["train_recall"].mean()
```

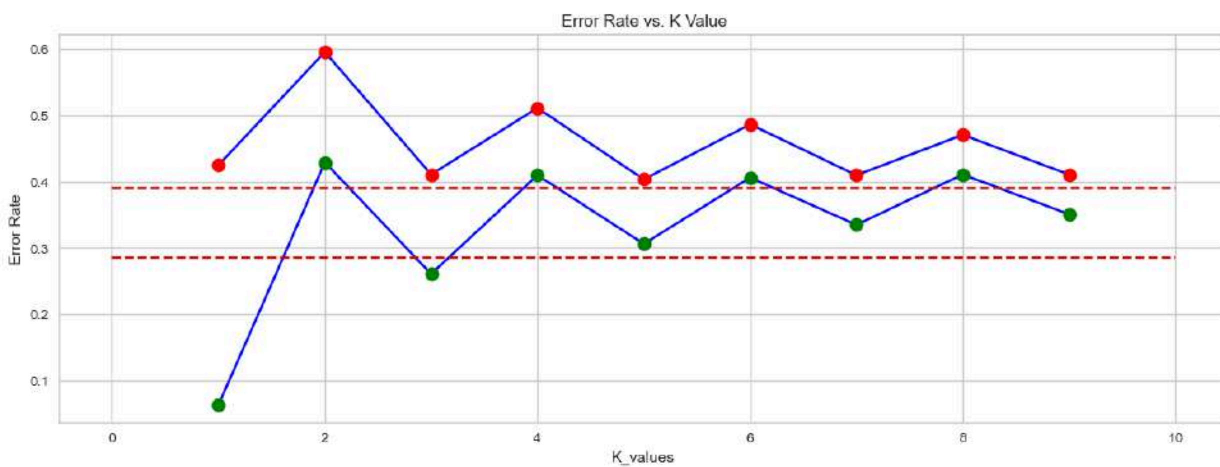
```
    test_error = 1 - recall_test_mean
    train_error = 1 - recall_train_mean
    test_error_rates.append(test_error)
    train_error_rates.append(train_error)
```

In [175…

```
plt.figure(figsize=(15,5))
plt.plot(range(1,10), test_error_rates, color='blue', marker='o',
         markerfacecolor='red', markersize=10)

plt.plot(range(1,10), train_error_rates, color='blue', marker='o',
         markerfacecolor='green', markersize=10)

plt.title('Error Rate vs. K Value')
plt.xlabel('K_values')
plt.ylabel('Error Rate')
plt.hlines(y=0.39, xmin = 0, xmax = 10, colors= 'r', linestyles="--")
plt.hlines(y=0.286, xmin = 0, xmax = 10, colors= 'r', linestyles="--")

# Red color: Test data erorr rates
# Green color: Train data error rates
```

Out[175…

```
<matplotlib.collections.LineCollection at 0x1b1075205f0>
```



Scores by Various K-Values

In [147…

```
k_list = [3,5,7, 8, 16]

for i in k_list:

    knn = Pipeline([("scaler", MinMaxScaler()), ("knn", KNeighborsClassifier(n_neighbors=k))
    knn.fit(X_train, y_train)

    print(f'WITH K={i}\n')

    eval_metric(knn, X_train, y_train, X_test, y_test, 'knn_model')
```

```
WITH K=3

knn_model Test_Set
[[4152  415]
 [ 606  909]]
              precision    recall  f1-score   support

           0       0.87      0.91      0.89      4567
           1       0.69      0.60      0.64      1515

    accuracy                           0.83      6082
   macro avg       0.78      0.75      0.77      6082
weighted avg       0.83      0.83      0.83      6082


knn_model Train_Set
[[16986  1280]
 [ 2122  3940]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91     18266
           1       0.75      0.65      0.70      6062

    accuracy                           0.86     24328
   macro avg       0.82      0.79      0.80     24328
weighted avg       0.86      0.86      0.86     24328

WITH K=5

knn_model Test_Set
[[4152  415]
 [ 606  909]]
              precision    recall  f1-score   support

           0       0.87      0.91      0.89      4567
           1       0.69      0.60      0.64      1515

    accuracy                           0.83      6082
   macro avg       0.78      0.75      0.77      6082
weighted avg       0.83      0.83      0.83      6082


knn_model Train_Set
[[16986  1280]
 [ 2122  3940]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91     18266
           1       0.75      0.65      0.70      6062

    accuracy                           0.86     24328
   macro avg       0.82      0.79      0.80     24328
weighted avg       0.86      0.86      0.86     24328

WITH K=7

knn_model Test_Set
[[4152  415]
 [ 606  909]]
              precision    recall  f1-score   support
```

```
           0       0.87      0.91      0.89      4567
           1       0.69      0.60      0.64      1515

    accuracy                           0.83      6082
   macro avg       0.78      0.75      0.77      6082
weighted avg       0.83      0.83      0.83      6082


knn_model Train_Set
[[16986  1280]
 [ 2122  3940]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91     18266
           1       0.75      0.65      0.70      6062

    accuracy                           0.86     24328
   macro avg       0.82      0.79      0.80     24328
weighted avg       0.86      0.86      0.86     24328

WITH K=8

knn_model Test_Set
[[4152  415]
 [ 606  909]]
              precision    recall  f1-score   support

           0       0.87      0.91      0.89      4567
           1       0.69      0.60      0.64      1515

    accuracy                           0.83      6082
   macro avg       0.78      0.75      0.77      6082
weighted avg       0.83      0.83      0.83      6082


knn_model Train_Set
[[16986  1280]
 [ 2122  3940]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91     18266
           1       0.75      0.65      0.70      6062

    accuracy                           0.86     24328
   macro avg       0.82      0.79      0.80     24328
weighted avg       0.86      0.86      0.86     24328

WITH K=16

knn_model Test_Set
[[4152  415]
 [ 606  909]]
              precision    recall  f1-score   support

           0       0.87      0.91      0.89      4567
           1       0.69      0.60      0.64      1515

    accuracy                           0.83      6082
   macro avg       0.78      0.75      0.77      6082
weighted avg       0.83      0.83      0.83      6082
```

```
knn_model Train_Set
[[16986  1280]
 [ 2122  3940]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91     18266
           1       0.75      0.65      0.70      6062

    accuracy                           0.86     24328
   macro avg       0.82      0.79      0.80     24328
weighted avg       0.86      0.86      0.86     24328
```

Cross Validation For Optimal K Value

In [148…

```
#Cross Validation, k=7;
model = Pipeline([("scaler", StandardScaler()), ("knn", KNeighborsClassifier(n_neighbors=7))

scores = cross_validate(model, X_train, y_train, scoring = ['accuracy', 'precision','recall'
                                                            'f1'], cv = 10, return_tr
df_scores = pd.DataFrame(scores, index = range(1, 11))
df_scores.mean()[2:]
```

Out[148…

```
test_accuracy      0.832580
train_accuracy     0.867249
test_precision     0.685028
train_precision    0.763561
test_recall        0.607388
train_recall       0.676839
test_f1            0.643719
train_f1           0.717584
dtype: float64
```

Gridsearch Method for Choosing Reasonable K Values

In [149…

```
knn_model.get_params()
```

Out[149…

```
{'memory': None,
 'steps': [('scaler', MinMaxScaler()),
  ('knn', KNeighborsClassifier(n_neighbors=9))],
 'verbose': False,
 'scaler': MinMaxScaler(),
 'knn': KNeighborsClassifier(n_neighbors=9),
 'scaler__clip': False,
 'scaler__copy': True,
 'scaler__feature_range': (0, 1),
 'knn__algorithm': 'auto',
 'knn__leaf_size': 30,
 'knn__metric': 'minkowski',
 'knn__metric_params': None,
 'knn__n_jobs': None,
 'knn__n_neighbors': 9,
 'knn__p': 2,
 'knn__weights': 'uniform'}
```

In [151…

```
#Cross Validation
model = Pipeline([("scaler", StandardScaler()), ("knn", KNeighborsClassifier())])
```

```
k_values = range(1,10)

param_grid = {
        "knn__n_neighbors": k_values,
        "knn__metric": ['minkowski'],
        "knn__p": [1, 2],
        "knn__weights": ['uniform', 'distance']
        }


knn_grid_model = GridSearchCV(model,
                              param_grid,
                              scoring='recall',
                              cv=5,
                              n_jobs=-1,
                              return_train_score=True).fit(X_train, y_train)
```

In [152…

```
print('Best Params:', knn_grid_model.best_params_)
print('Best Recall Score(test):', knn_grid_model.best_score_)
print('Best Score index:', knn_grid_model.best_index_)
```

```
Best Params: {'knn__metric': 'minkowski', 'knn__n_neighbors': 9, 'knn__p': 2, 'knn__weights':
'uniform'}
Best Recall Score(test): 0.6044203472284574
Best Score index: 34
```

In [153…

```
# Checking overfiting with the CV scores

pd.DataFrame(knn_grid_model.cv_results_).loc[34, ["mean_test_score", "mean_train_score"]]
```

Out[153…

```
mean_test_score      0.60442
mean_train_score    0.659559
Name: 34, dtype: object
```

In [154…

```
# Prediction
y_pred = knn_grid_model.predict(X_test)
y_pred_proba = knn_grid_model.predict_proba(X_test)

# Scores to compare the models at the end.
knn_grid_f1 = f1_score(y_test, y_pred)
knn_grid_recall = recall_score(y_test, y_pred)
knn_grid_auc = roc_auc_score(y_test, y_pred)
```

In [155…

```
# Checking the Incorrect Predictions

# Test Data df
test_data = pd.concat([X_test, y_test], axis=1)

# Create new column for 'predicted' classes to compore with actual target classes
test_data["pred"] = y_pred

# Filtering the wrong predicted obs
wrong_pred = test_data[((test_data["income"] == 0) & (test_data["pred"] == 1)) |
             ((test_data["income"] == 1) & (test_data["pred"] == 0))]

print('knn_grid_model Total Incorrect Predictions:', wrong_pred.shape)


print('------------------------------------------------')
# Actual-Predicted-Probalility of Pozitive Class(1)
```

```
my_dict = {"Actual": y_test, "Pred":y_pred, "Proba_1":y_pred_proba[:,1]}
pd.DataFrame.from_dict(my_dict).sample(10)
```

```
knn_grid_model Total Incorrect Predictions: (1000, 49)
------------------------------------------------
```

Out[155…

	Actual	Pred	Proba_1
3065	1	1	1.000000
7993	1	0	0.222222
25111	0	0	0.333333
25536	0	0	0.000000
24265	1	0	0.111111
30717	0	0	0.000000
18784	0	1	0.888889
14676	1	0	0.111111
12813	1	1	1.000000
27900	0	0	0.333333

Evaluating The Model Performance

In [156…

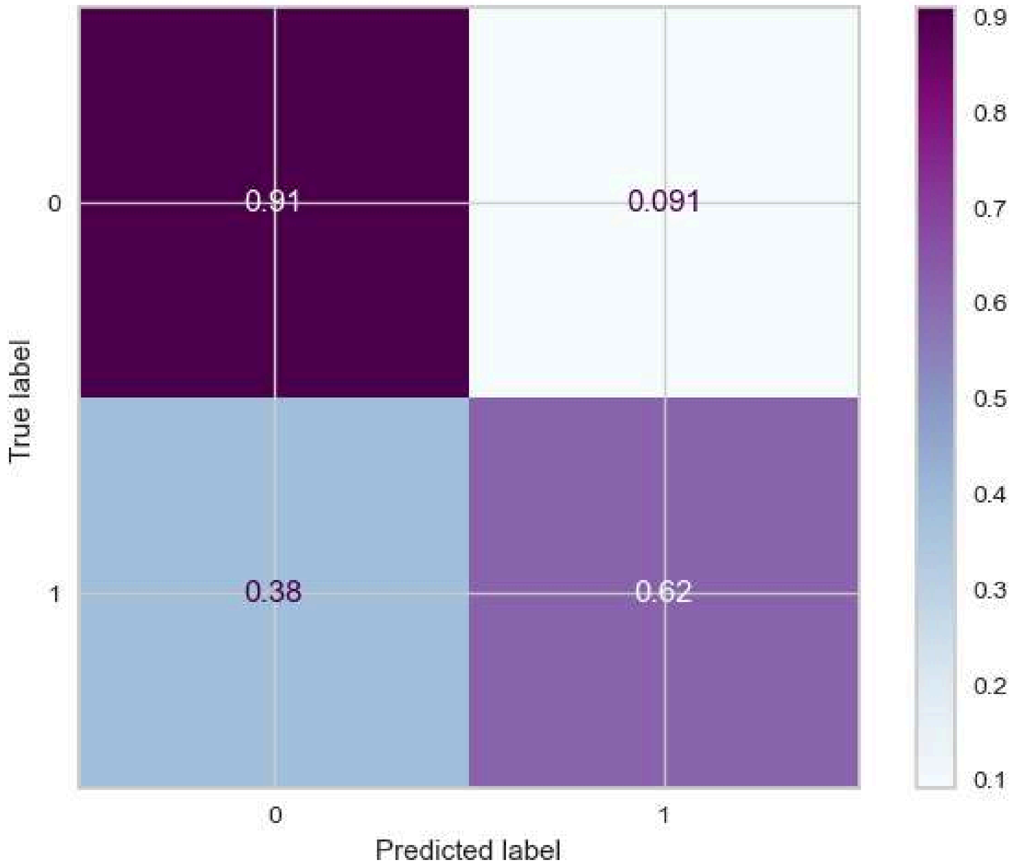
```
# Evaluating the Model Performance using Classification Metrics

print('WITH K=7\n')
eval_metric(knn_grid_model, X_train, y_train, X_test, y_test,'knn_model')
```

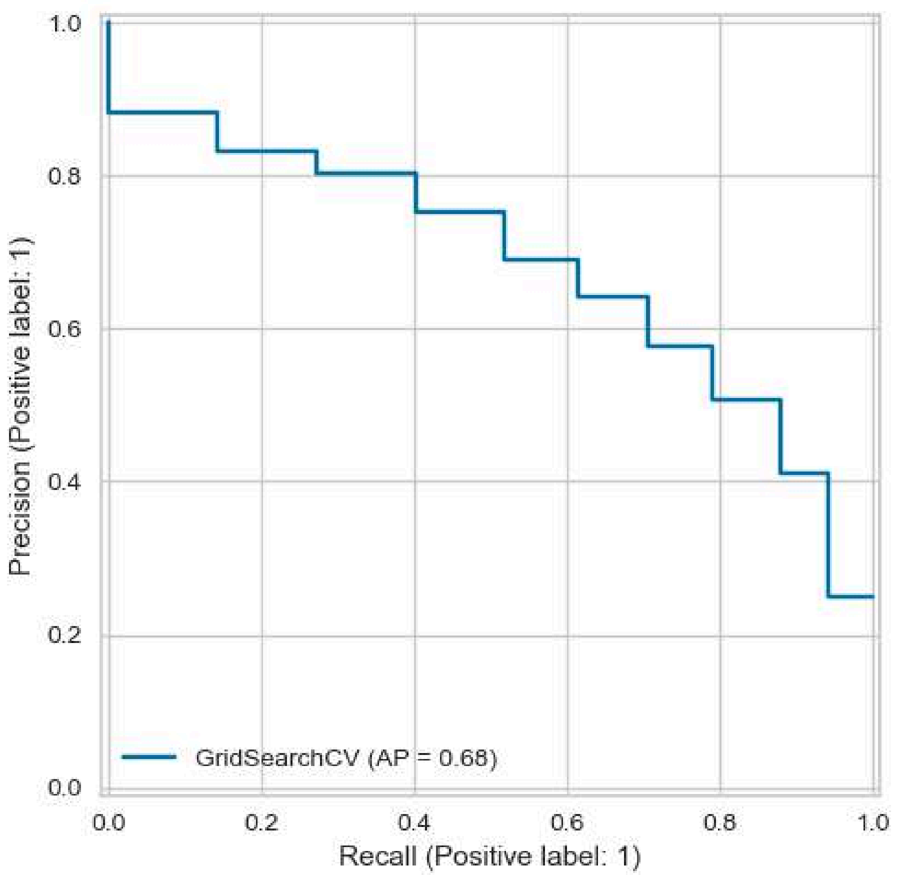
```
WITH K=7

knn_model Test_Set
[[4150  417]
 [ 583  932]]
              precision    recall  f1-score   support

           0       0.88      0.91      0.89      4567
           1       0.69      0.62      0.65      1515

    accuracy                           0.84      6082
   macro avg       0.78      0.76      0.77      6082
weighted avg       0.83      0.84      0.83      6082


knn_model Train_Set
[[16964  1302]
 [ 2040  4022]]
              precision    recall  f1-score   support

           0       0.89      0.93      0.91     18266
           1       0.76      0.66      0.71      6062

    accuracy                           0.86     24328
   macro avg       0.82      0.80      0.81     24328
weighted avg       0.86      0.86      0.86     24328
```

In [157…

```
# Roc_AUC_score
print('knn_grid_model ROC_AUC Score:', roc_auc_score(y_test, y_pred_proba[:,1]))
print('--------------------------------------------------')

# Confusion Matrix
knn_grid_matrix = ConfusionMatrixDisplay.from_estimator(knn_grid_model, X_test,y_test, norma

# Precision-Recall Curve
knn_grid_prCurve = PrecisionRecallDisplay.from_estimator(knn_grid_model, X_test, y_test)
```

```
knn_grid_model ROC_AUC Score: 0.8726853788947977
--------------------------------------------------
```

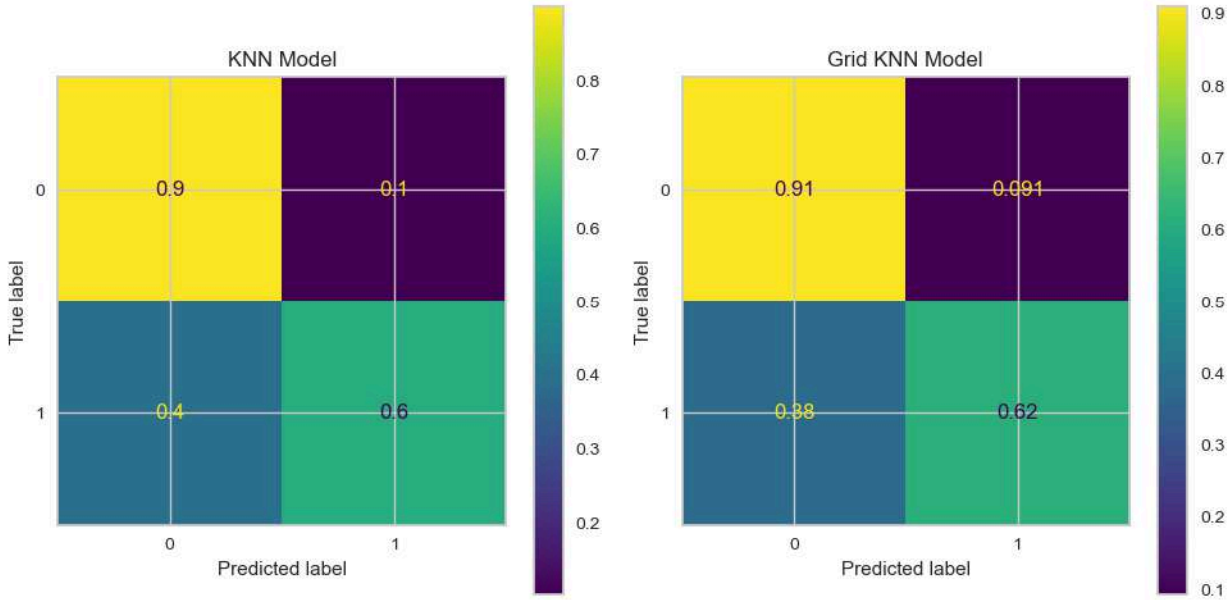




In [158…

```
fig, ax = plt.subplots(1, 2, figsize=(10,5))

knn_matrix.plot(ax=ax[0])
```

```
ax[0].set_title("KNN Model")
knn_grid_matrix.plot(ax=ax[1])
ax[1].set_title("Grid KNN Model")

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

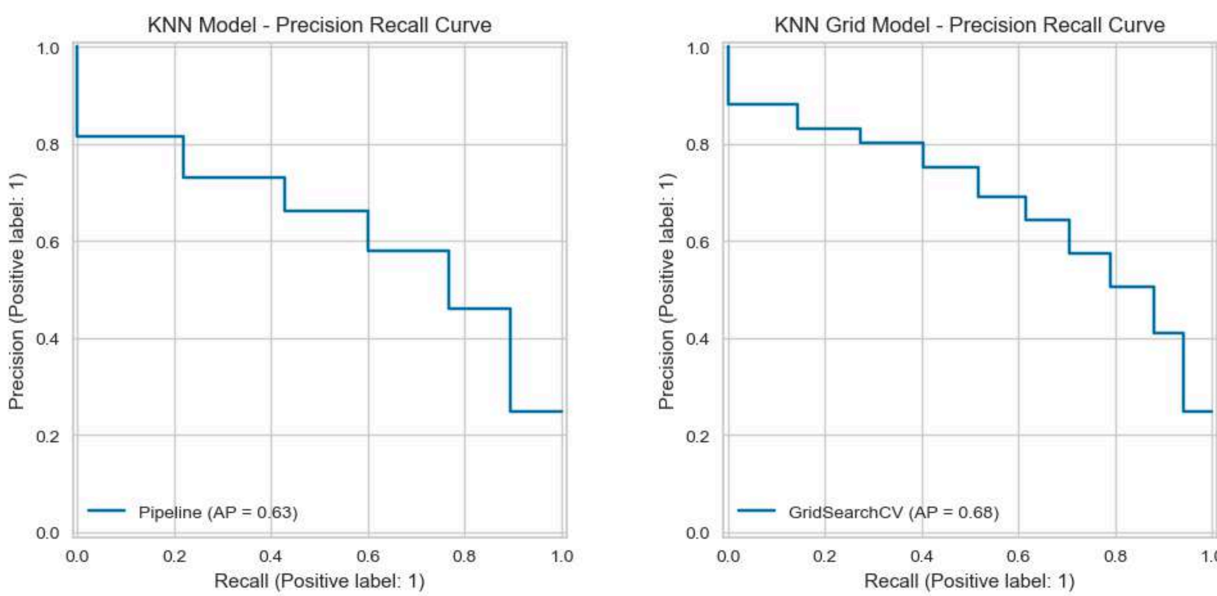


In [159...

```
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

knn_prCurve.plot(ax=ax[0])
ax[0].set_title("KNN Model - Precision Recall Curve")
knn_grid_prCurve.plot(ax=ax[1])
ax[1].set_title("KNN Grid Model - Precision Recall Curve")
```

Out[159...

```
Text(0.5, 1.0, 'KNN Grid Model - Precision Recall Curve')
```

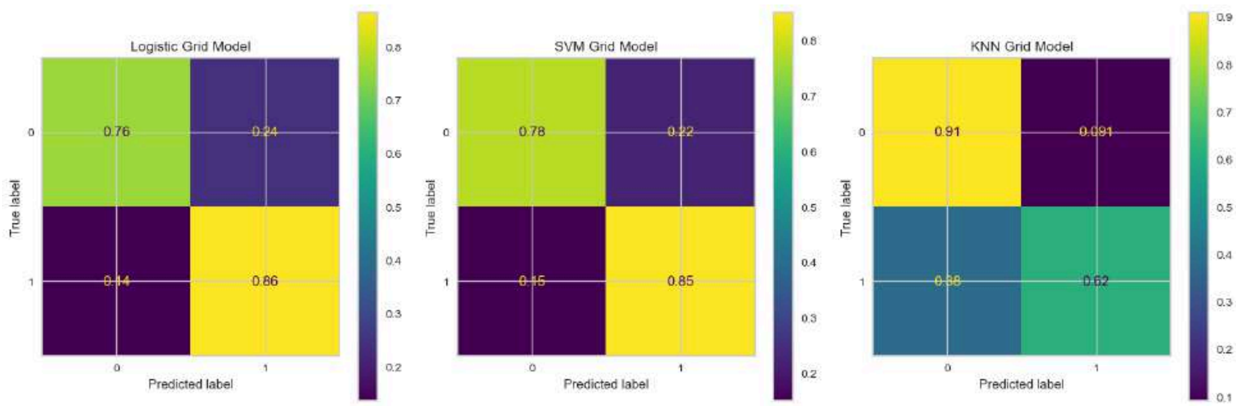


Comparing the All Models (Logistic-SVM-KNN)

- We focus on Precision-Recall because the data is unbalanced, meaning there are significantly more instances of one class than the other.
- In such cases, accuracy can be misleading, as it may be high simply by predicting the majority class.
- Precision-Recall provides a clearer picture of the model's ability to correctly identify and handle the minority class, by evaluating how well the model avoids false positives (Precision) and captures true positives (Recall).
- This is crucial when the minority class is of particular interest.

In [169…

```
# Confusion Matrix
fig, ax = plt.subplots(1, 3, figsize=(15,5))

grid_log_matrix.plot(ax=ax[0])
ax[0].set_title("Logistic Grid Model")
svm_grid_matrix.plot(ax=ax[1])
ax[1].set_title("SVM Grid Model")
knn_grid_matrix.plot(ax=ax[2])
ax[2].set_title("KNN Grid Model")

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

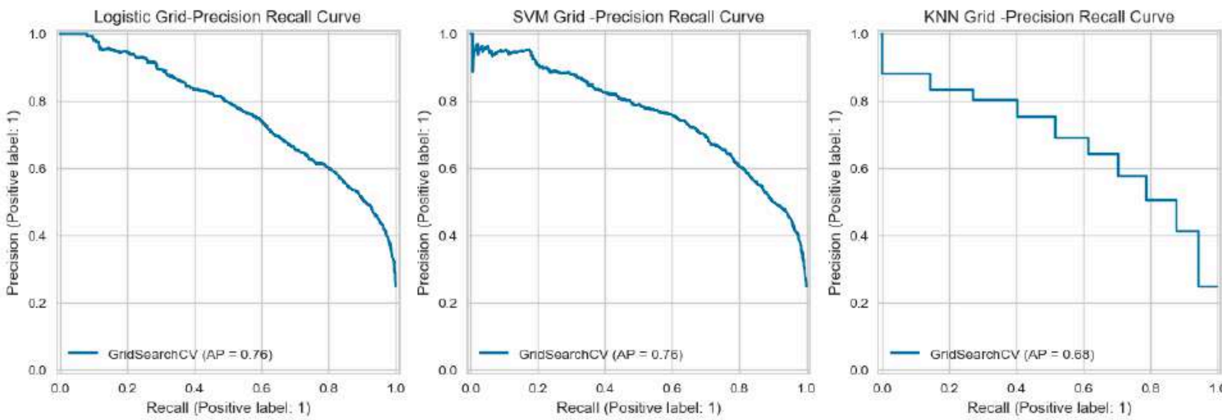


In [168…

```
# Precision-Recall Curves

fig, ax = plt.subplots(1, 3, figsize=(15, 5))

grid_log_prCurve.plot(ax=ax[0])
ax[0].set_title("Logistic Grid-Precision Recall Curve")
svm_grid_prCurve.plot(ax=ax[1])
ax[1].set_title("SVM Grid -Precision Recall Curve")
knn_grid_prCurve.plot(ax=ax[2])
ax[2].set_title("KNN Grid -Precision Recall Curve")
```

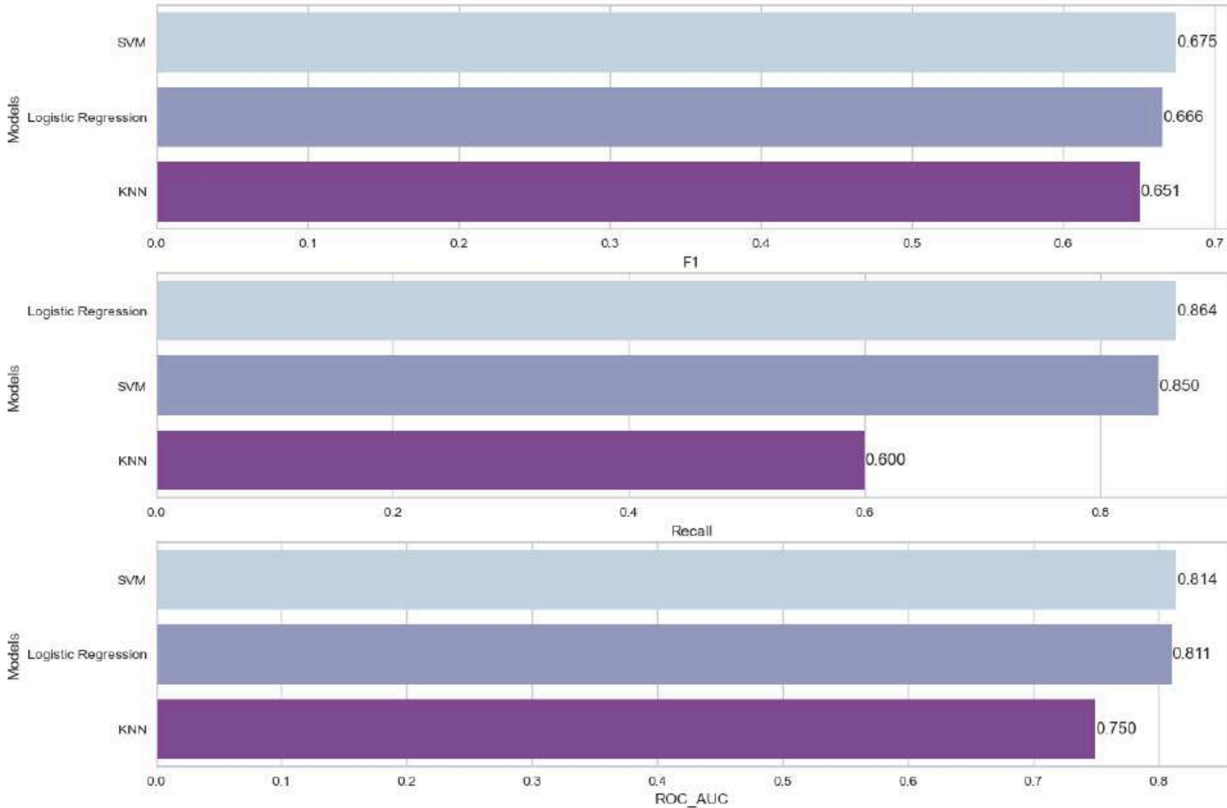
Out[168…

```
Text(0.5, 1.0, 'KNN Grid -Precision Recall Curve')
```



In [165…

```
# F1 - Recall - ROC_AUC Scores

compare = pd.DataFrame({"Models": ["Logistic Regression","SVM", "KNN"],
                        "F1": [log_grid_f1,svm_grid_f1, knn_grid_f1],
                        "Recall": [log_grid_recall, svm_grid_recall, knn_recall],
                        "ROC_AUC": [log_grid_auc,svm_grid_auc, knn_auc]})

def labels(ax):
    for p in ax.patches:
        width = p.get_width()                        # get bar length
        ax.text(width,                               # set the text at 1 unit right of the b
                p.get_y() + p.get_height() / 2,      # get Y coordinate + X coordinate / 2
                '{:1.3f}'.format(width),             # set variable to display, 2 decimals
                ha = 'left',                         # horizontal alignment
                va = 'center')                       # vertical alignment


plt.figure(figsize=(14,10))

plt.subplot(311)
compare = compare.sort_values(by="F1", ascending=False)
ax=sns.barplot(x="F1", y="Models", data=compare, palette="BuPu")
labels(ax)

plt.subplot(312)
compare = compare.sort_values(by="Recall", ascending=False)
ax=sns.barplot(x="Recall", y="Models", data=compare, palette="BuPu")
labels(ax)

plt.subplot(313)
compare = compare.sort_values(by="ROC_AUC", ascending=False)
ax=sns.barplot(x="ROC_AUC", y="Models", data=compare, palette="BuPu")
labels(ax)

plt.show()
```



Final Model and Deployment

In [170…

```
# SVM Model with the Best Parameters

model = Pipeline([("scaler", MinMaxScaler()), ("SVC", SVC(class_weight="balanced"))])


param_grid = {"SVC__C":[1],
              "SVC__gamma":["scale"],
              "SVC__kernel":["rbf"]}

cv = StratifiedKFold(n_splits = 5) # for unbalanced data

final_svm_model = GridSearchCV(model,
                          param_grid=param_grid,
                          cv=cv,
                          scoring = "recall_macro",
                          n_jobs = -1, # Uses all available cores
                          verbose=1,
                          return_train_score=True).fit(X_train, y_train) # fit the model
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
```

In [171…

```
# Export the final model to your local -> serilarization

import pickle
pickle.dump(final_svm_model, open("final_classification_model", "wb"))
```

In [172…

```
# Import the final model to use -> deserilization
```

```
new_model = pickle.load(open("final_classification_model", "rb"))
new_model
```

Out[172…

```
GridSearchCV
estimator: Pipeline
MinMaxScaler
SVC
```

Conclusion

Final Model: SVM Model

Parameters:

- recall: 85,
- f1: 0.87
- prc: 0.76

In this project, we used logistic regression, SVM, and KNN models to predict income levels on an unbalanced dataset. We focused on F1 and recall scores to evaluate performance, as they are critical in **unbalanced datasets** where the minority class (higher income) is key.

Why SVM was Chosen:

- **Balanced Performance**: The SVM model achieved a strong balance between precision and recall, with an F1 score of 0.87 and a recall of 0.85 on the test set. This makes it effective at identifying high-income individuals while keeping false positives low.
- **Consistency**: SVM showed stable performance across training and test sets, indicating good generalization without overfitting.

Importance of F1 and Recall:

- **F1 Score**: This metric combines precision and recall, ensuring the model performs well with both false positives and false negatives in mind.
- **Recall**: Prioritizing recall ensures we capture most high-income individuals, which is vital in unbalanced datasets.

In short, the SVM model's balanced precision and recall, along with its consistent performance, make it the best choice as the final model for predicting income levels.

If you find this work helpful, don't forget to give it an 👍 UPVOTE! and join the discussion! 💬

Thank you...

Duygu Jones | Data Scientist | 2024

Follow me: duygujones.com | Linkedin | GitHub | Kaggle | Medium | Tableau

In [ ]:

```
```