**Census Income**

In this project, we are going to predict whether a person’s income is above 50k or below 50k using various features like age, education, and occupation etc. The dataset we are going to use is the census income dataset which is provided by FlipRoboTechnologies and it contains about 32560 rows and 15 features.

**Understanding the problem:**

The dataset contains the labels which we have to predict which is the dependent feature ‘Income level’. This feature is discrete consisting of two categories income less than 50k and more than 50k. So the problem we have is a Supervised Binary Classification type.

**Step 1 : Import libraries and dataset**

All the standard libraries like numpy, pandas, matplotlib, and seaborn are imported in this step. We use numpy for linear algebra operations, pandas for using data frames, matplotlib, and seaborn for plotting graphs. The dataset is imported using the pandas command read\_csv().

# Import libraries

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Importing dataset

dataset = pd.read\_csv(“<https://raw.githubusercontent.com/FlipRoboTechnologies/ML_-Datasets/main/Census%20Income/Census%20Income.csv>”)

**Step 2 : Descriptive analysis**

# Preview dataset:

df.head()

# Shape of dataset:

df.shape()

# Information of dataset

df.info()

# Statistical summary of dataset

df.describe()

# Check for Null Values

df.isnull().sum()

# Check for ‘?’ inside dataset

df.isin([‘?’]).sum()

# Check the counts of label categories

income = df['Income'].value\_counts(normalize=True)

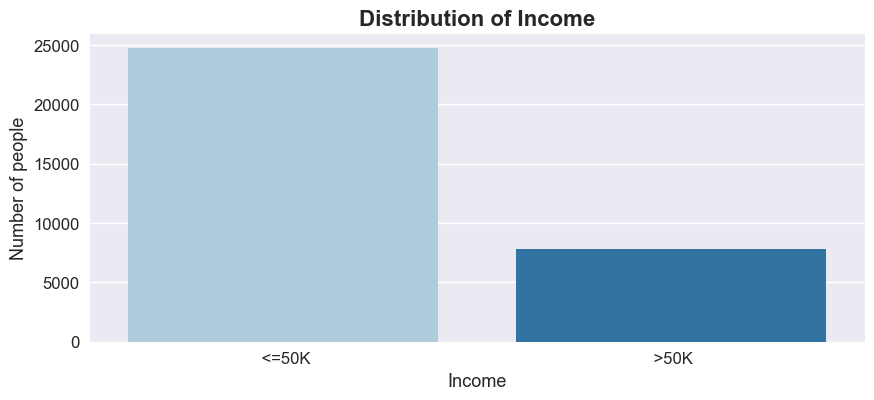
print(income)

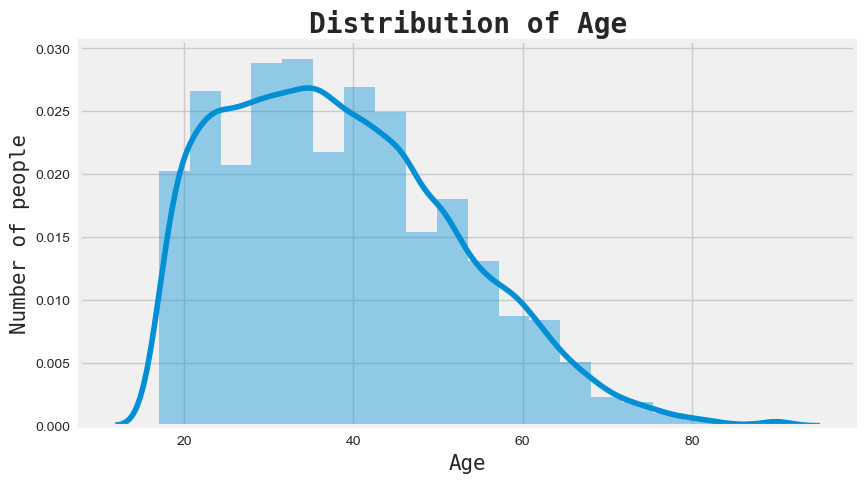
After the ‘Descriptive Analysis’ we observe that :-

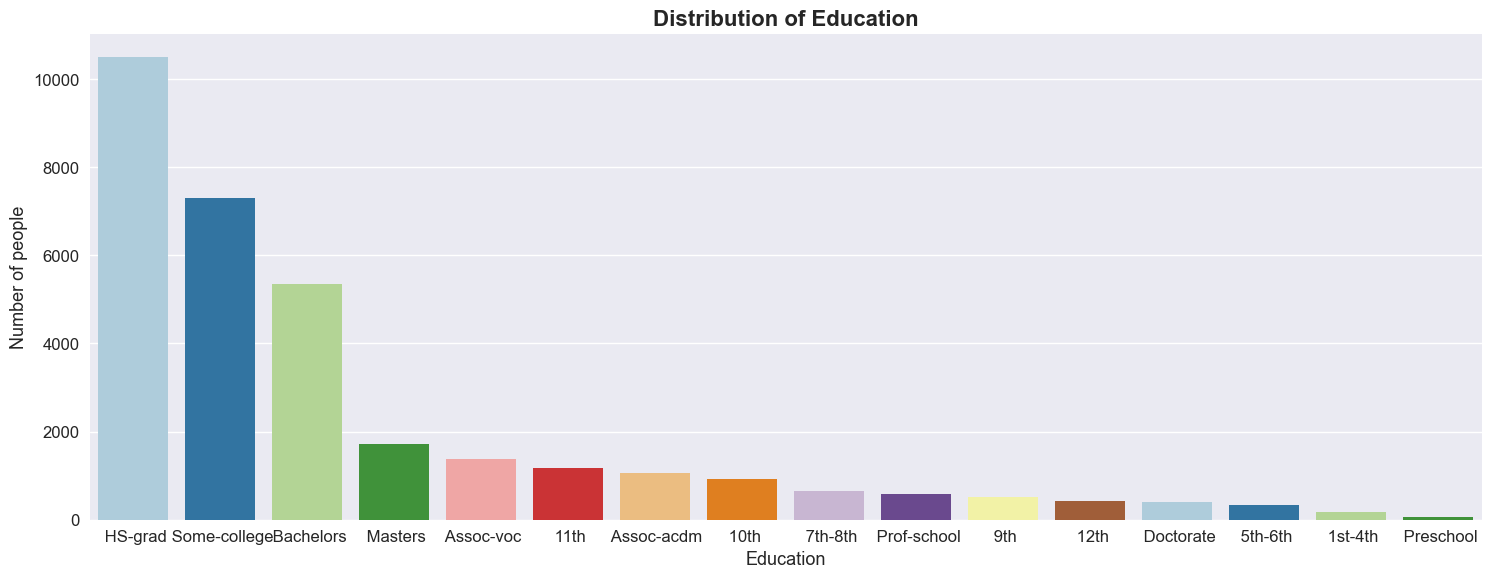
* The dataset doesn’t have any null values, but it contains missing values in the form of ‘?’ which needs to be preprocessed.
* The dataset is unbalanced, as the dependent feature ‘income’ contains 75.92% values have income less than 50k, and 24.08% values have income more than 50k.

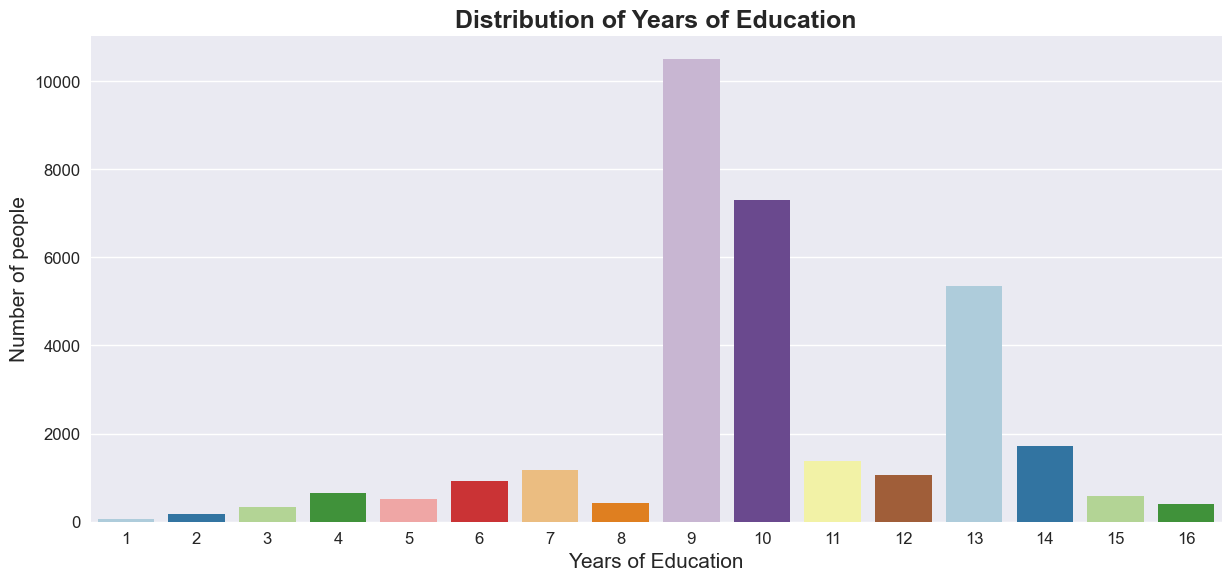
**Step 3 : Exploratory Data Analysis**

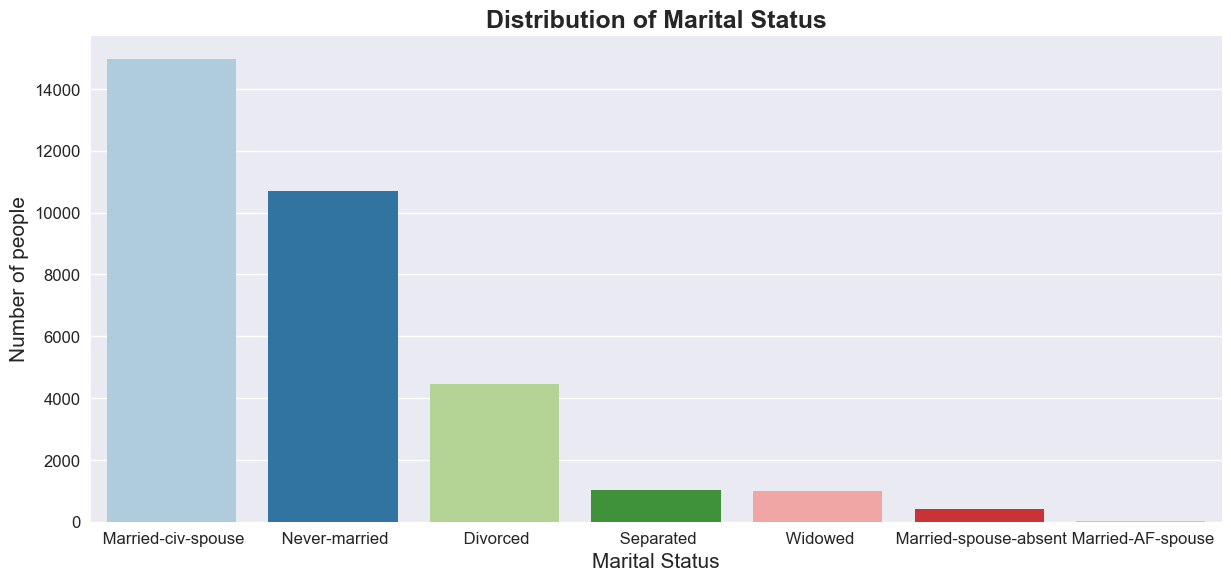
3.1-Univariate Analysis

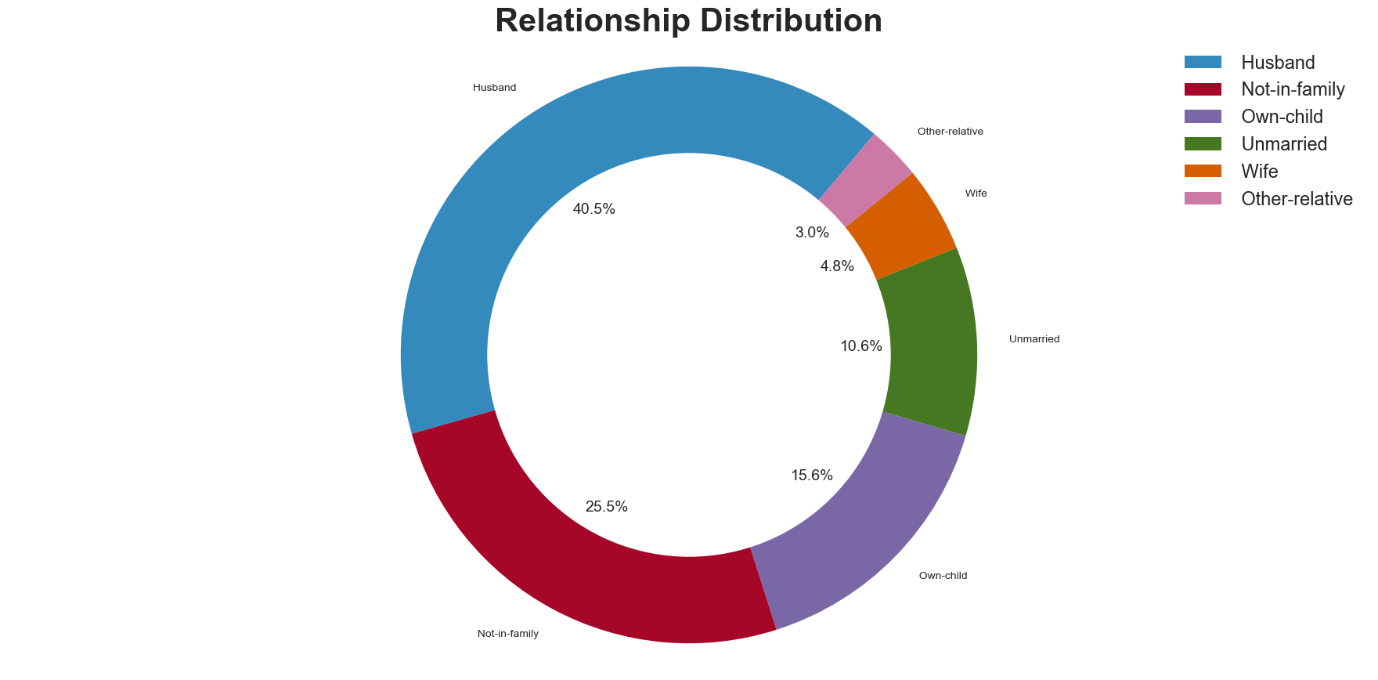


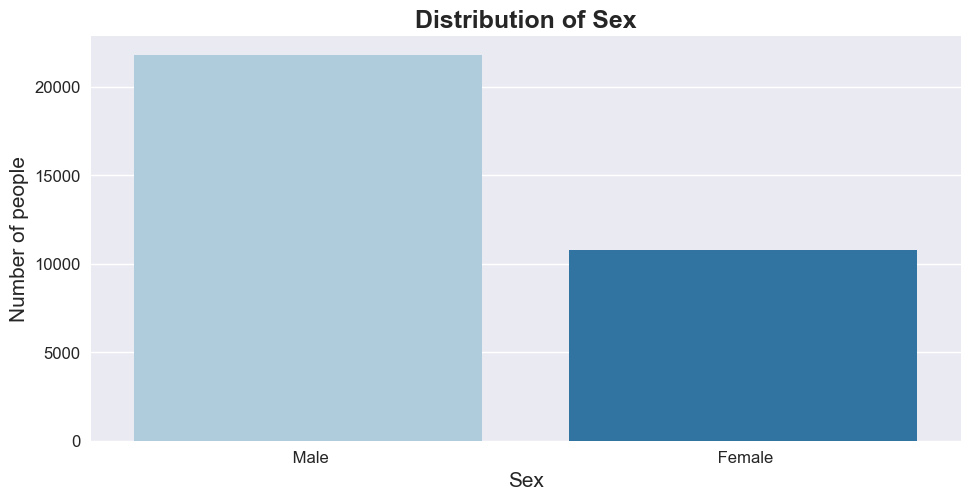


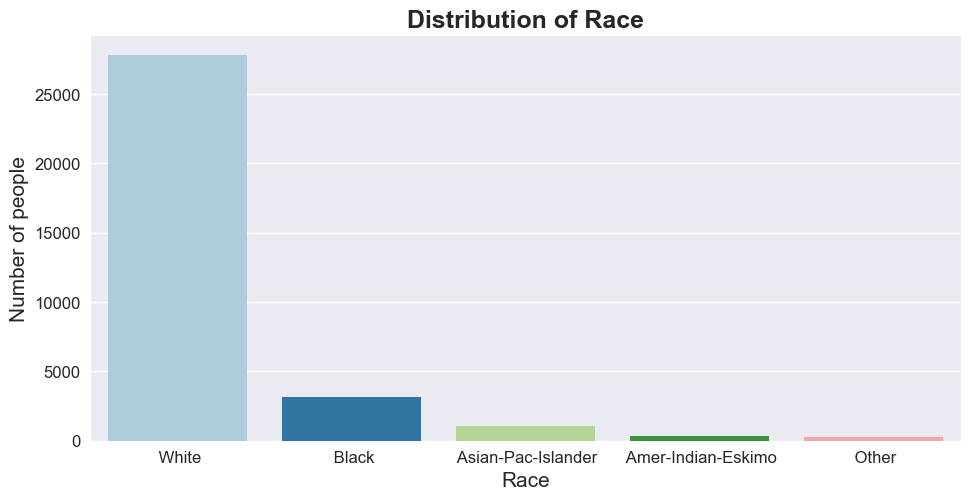


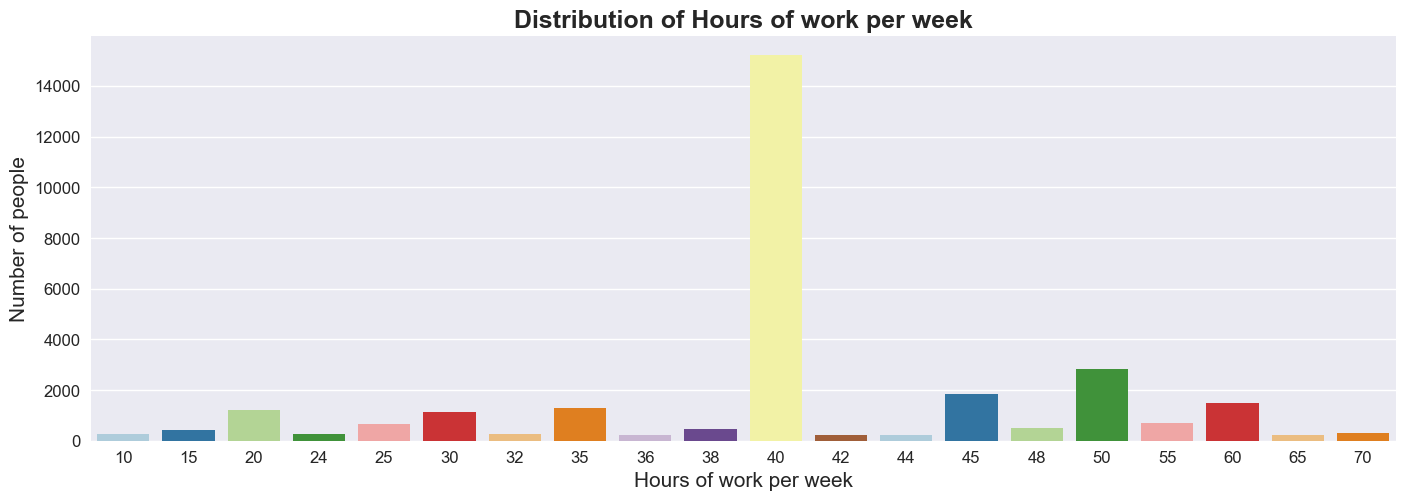




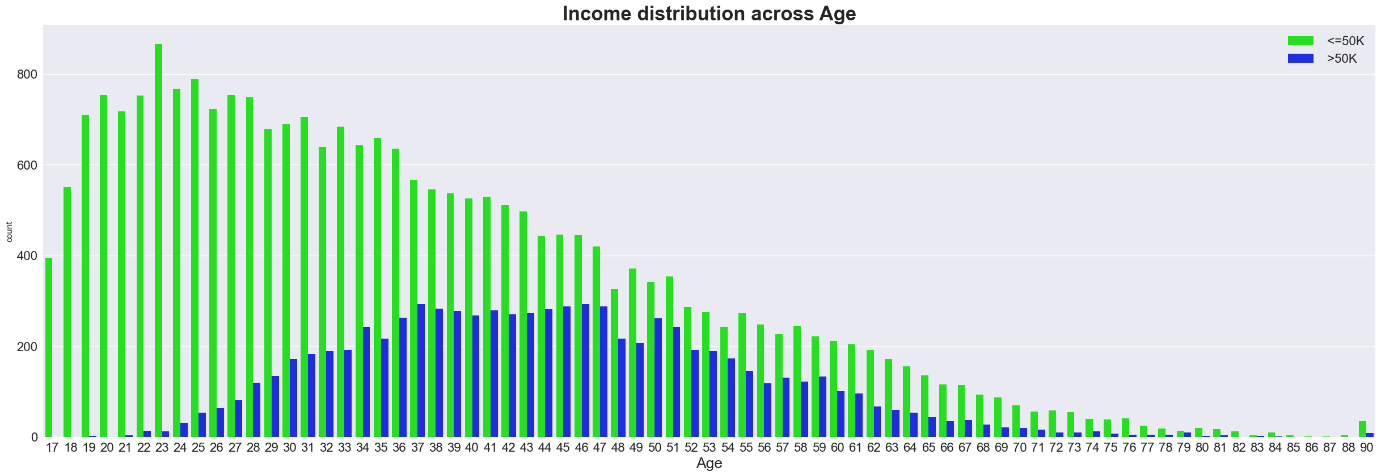


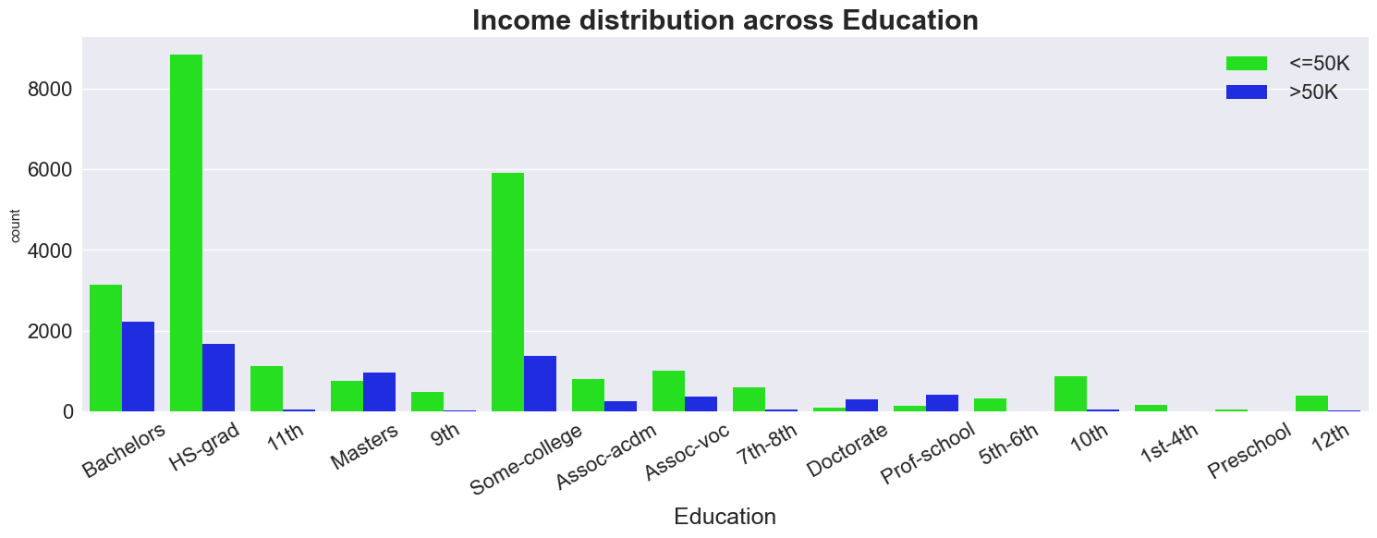


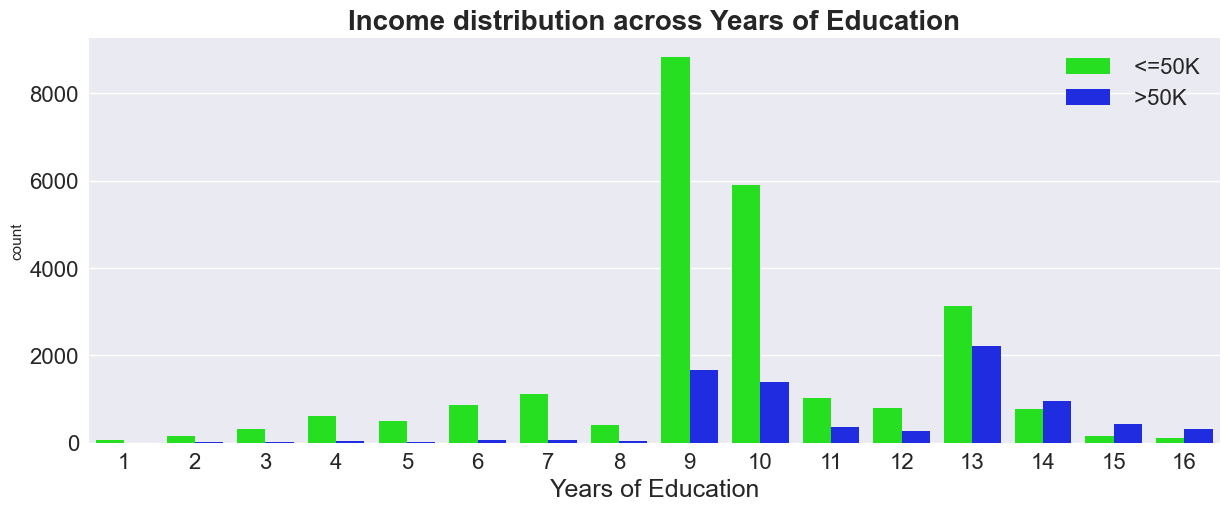


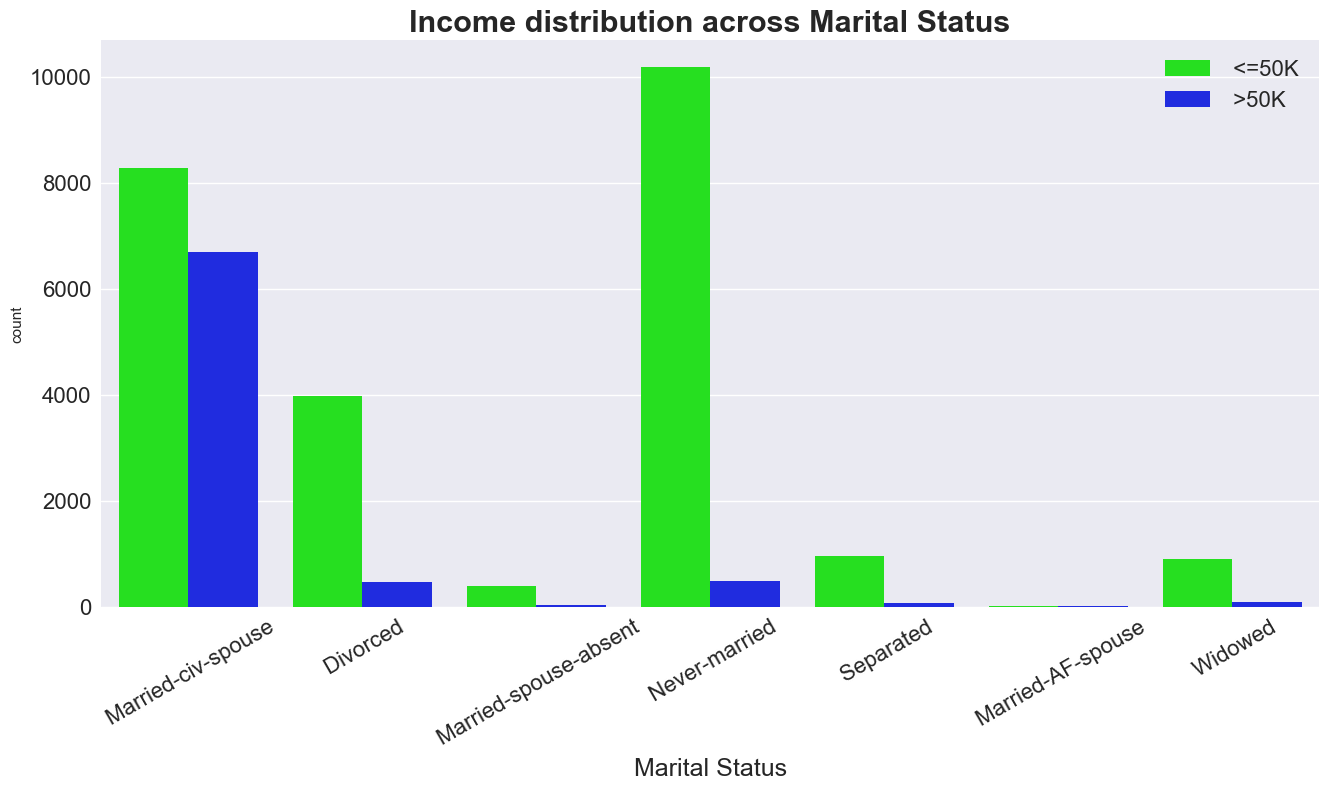


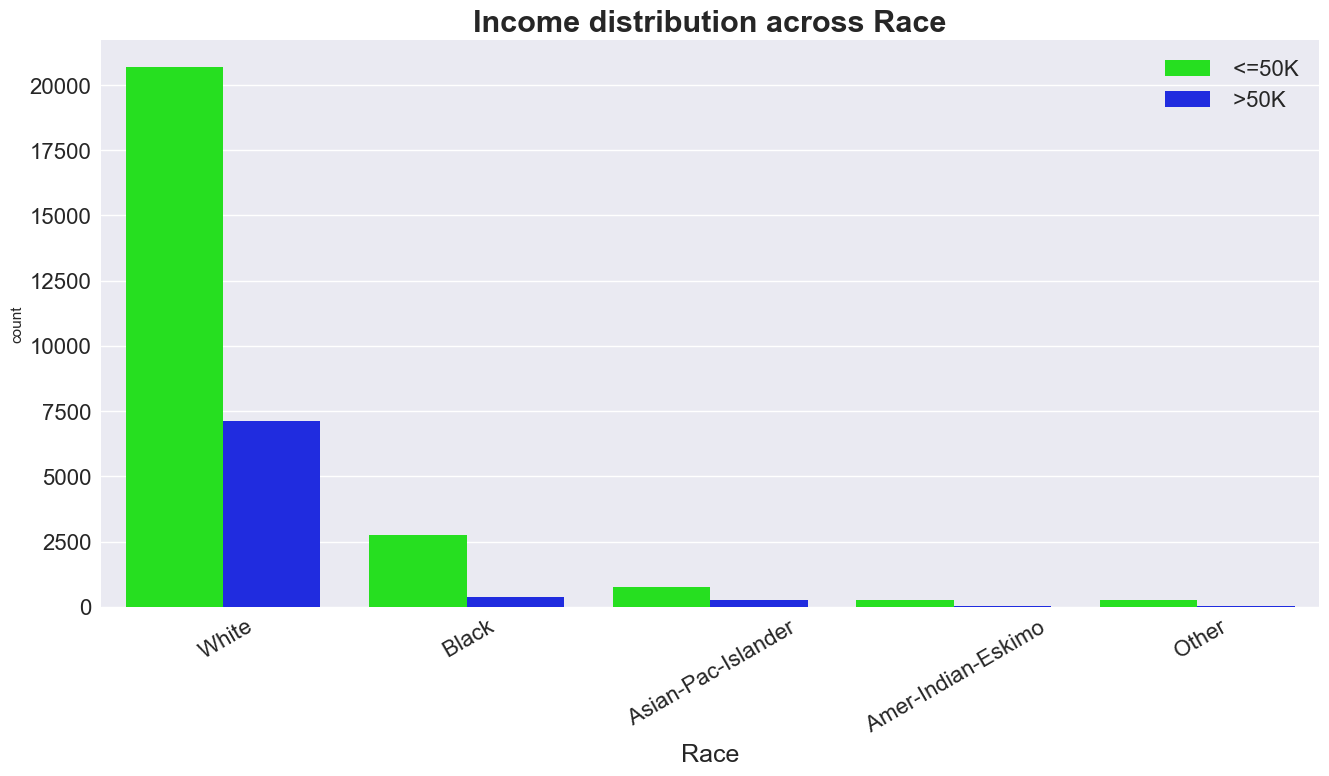
3.2- Bivariate Analysis

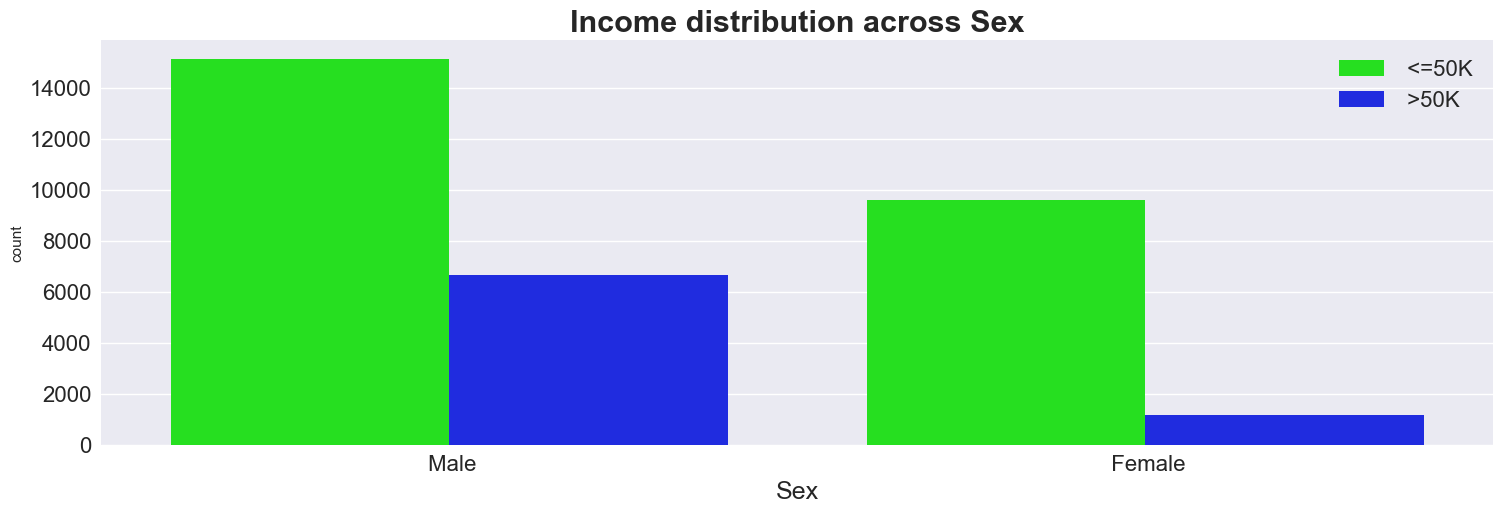




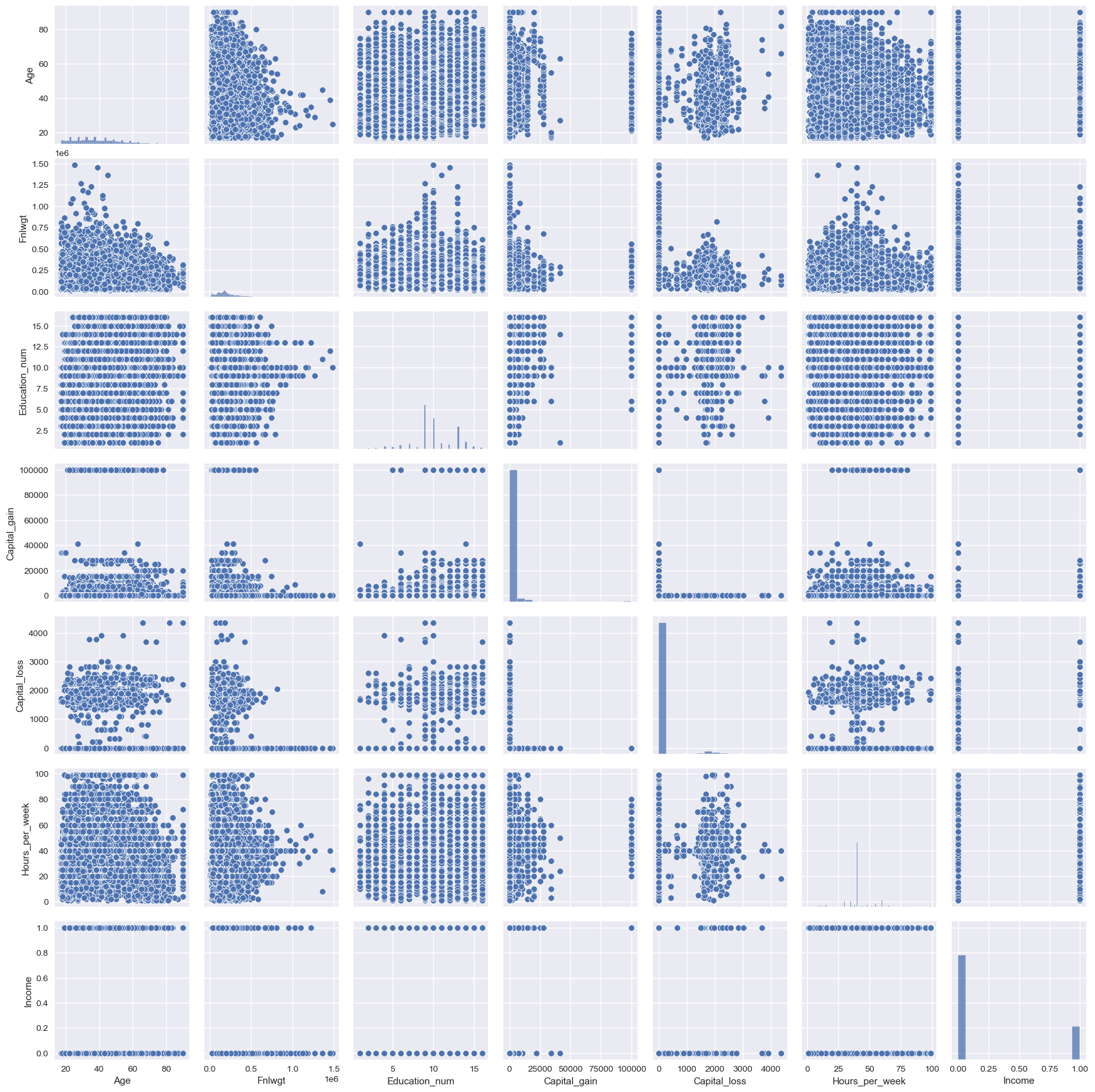








3.3- Multivariate Analysis



**Step 4: Data Preprocessing**

The null values are in the form of ‘?’ which can be easily replaced with the most frequent value(mode) using the fillna() command.

df**.**isin([' ?'])**.**sum()

round((df**.**isin(['?'])**.**sum() **/** df**.**shape[0]) **\*** 100, 2)**.**astype(str) **+** ' %'

df **=** df**.**replace('?', np**.**nan)

columns\_with\_nan **=** ['Workclass', 'Occupation', 'Native\_country']

**for** col **in** columns\_with\_nan:

df[col]**.**fillna(df[col]**.**mode()[0], inplace**=True**)

The object columns in the dataset need to be encoded so that they can be further used. This can be done using Label Encoder in the sklearn’s preprocessing library.

**# Label Encoding:**

**for** col **in** df**.**columns:

**if** df[col]**.**dtypes **==** 'object':

encoder **=** LabelEncoder()

df[col] **=** encoder**.**fit\_transform(df[col])

The dataset is then split into X which contains all the independent features and Y which contains the dependent feature ‘Income’.

X **=** df**.**drop('Income', axis**=**1)

Y **=** df['Income']

The curse of multicollinearity and the problem of overfitting can be solved by performing Feature Selection. The feature importances can be easily found by using the ExtraTreesClassifier.

**# Feature Selection**

X **=** df**.**drop('Income', axis**=**1)

Y **=** df['Income']

selector **=** ExtraTreesClassifier(random\_state**=**42)

selector**.**fit(X, Y)

feature\_imp **=** selector**.**feature\_importances\_

**for** index, val **in** enumerate(feature\_imp):

print(index, round((val **\*** 100), 2))

Using Feature Scaling we can standardize the dataset to help the model learn the patterns. This can be done with StandardScaler() from sklearn’s preprocessing library.

**for** col **in** X**.**columns:

scaler **=** StandardScaler()

X[col] **=** scaler**.**fit\_transform(X[col]**.**values**.**reshape(**-**1, 1))

The dependent feature ‘Income’ is highly imbalanced as 75.92% values have income less than 50k and 24.08% values have income more than 50k. This needs to be fixed as it results in a low F1 score. As we have a small dataset we can perform Oversampling using a technique like RandomOverSampler.

**# Fixing Imbalanced Dataset**

round(Y**.**value\_counts(normalize**=True**) **\*** 100, 2)**.**astype('str') **+** ' %'

ros **=** RandomOverSampler(random\_state**=**42)

ros**.**fit(X, Y)

X\_resampled, Y\_resampled **=** ros**.**fit\_resample(X, Y)

round(Y\_resampled**.**value\_counts(normalize**=True**) **\*** 100, 2)**.**astype('str') **+** ' %'

The dataset is split into training data and testing data in the ratio 80:20 using the train\_test\_split() command.

X\_train, X\_test, Y\_train, Y\_test **=** train\_test\_split(X\_resampled, Y\_resampled, test\_size**=**0.2, random\_state**=**42)

print("X\_train shape:", X\_train**.**shape)

print("X\_test shape:", X\_test**.**shape)

print("Y\_train shape:", Y\_train**.**shape)

print("Y\_test shape:", Y\_test**.**shape)

**Step 5- Data Modelling**

**Random Forest Classifier**

ran\_for **=** RandomForestClassifier(random\_state**=**42)

ran\_for**.**fit(X\_train, Y\_train)

Y\_pred\_ran\_for **=** ran\_for**.**predict(X\_test)

Random forest is a Supervised learning algorithm that is used for both classification and regression. It is a type of bagging ensemble algorithm, which creates multiple decision trees simultaneously trying to learn from the dataset independent of one another. The final prediction is selected using majority voting.

Random forests are very flexible and give high accuracy as it overcomes the problem of overfitting by combining the results of multiple decision trees. Even for large datasets, random forests give a good performance. They also give good accuracy if our dataset has a large number of missing values. But random forests are more complex and computationally intensive than decision trees resulting in a time-consuming model building process. They are also harder to interpret and less intuitive than a decision tree.

This algorithm has some important parameters like max\_depth, max\_features, n\_estimators, and min\_sample\_leaf. The number of trees which can be used to build the model is defined by n\_estimators. Max\_features determines the maximum number of features the random forest can use in an individual tree. The maximum depth of the decision trees is given by the parameter max\_depth. The minimum number of samples required at a leaf node is given by min\_sample\_leaf.

**Step 6- Model Evaluation**

In this step, we will evaluate our model using two metrics which are accuracy\_score and f1\_score. Accuracy is the ratio of correct predicted values over the total predicted values. It tells us how accurate our prediction is. F1 score is the weighted average of precision and recall and higher its value better the model. We will use the accuracy score with f1 score as we have an imbalanced dataset.

print('Random Forest Classifier:')

print('Accuracy score:', round(accuracy\_score(Y\_test, Y\_pred\_ran\_for) **\*** 100, 2))

print('F1 score:', round(f1\_score(Y\_test, Y\_pred\_ran\_for) **\*** 100, 2))

**Step 7- Hyperparameter Tuning**

We will tune the hyperparameters of our random forest classifier using RandomizedSearchCV which finds the best hyperparameters by searching randomly avoiding unnecessary computation. We will try to find the best values for ‘n\_estimators’ and ‘max\_depth’.

n\_estimators **=** [int(x) **for** x **in** np**.**linspace(start**=**40, stop**=**150, num**=**15)]

max\_depth **=** [int(x) **for** x **in** np**.**linspace(40, 150, num**=**15)]

param\_dist **=** {'n\_estimators': n\_estimators, 'max\_depth': max\_depth,}

rf\_tuned **=** RandomForestClassifier(random\_state**=**42)

rf\_cv **=** RandomizedSearchCV(estimator**=**rf\_tuned,

param\_distributions**=**param\_dist, cv**=**5, random\_state**=**42)

rf\_cv**.**fit(X\_train, Y\_train)

rf\_cv**.**best\_score\_

rf\_cv**.**best\_params\_

rf\_best **=** RandomForestClassifier(max\_depth**=**102, n\_estimators**=**40, random\_state**=**42)

rf\_best**.**fit(X\_train, Y\_train)

Y\_pred\_rf\_best **=** rf\_best**.**predict(X\_test)

print('Random Forest Classifier:')

print('Accuracy score:', round(accuracy\_score(Y\_test, Y\_pred\_rf\_best) **\*** 100, 2))

print('F1 score:', round(f1\_score(Y\_test, Y\_pred\_rf\_best) **\*** 100, 2))

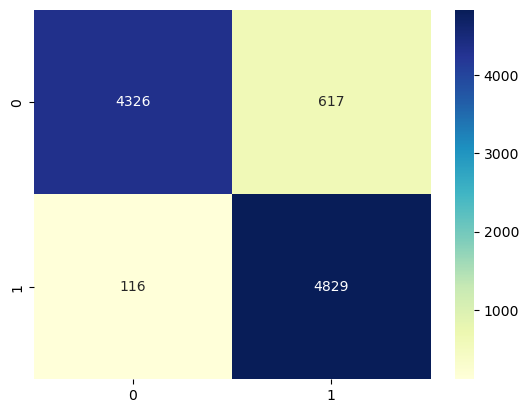
cm **=** confusion\_matrix(Y\_test, Y\_pred\_rf\_best)

plt**.**style**.**use('default')

sns**.**heatmap(cm, annot**=True**, fmt**=**'d', cmap**=**'YlGnBu')

plt**.**savefig('heatmap.png')

plt**.**show()



print(classification\_report(Y\_test, Y\_pred\_rf\_best))

precision recall f1-score support

0 0.97 0.88 0.92 4943

1 0.89 0.98 0.93 4945

accuracy 0.93 9888

macro avg 0.93 0.93 0.93 9888

weighted avg 0.93 0.93 0.93 9888

The model gives us the best values for an accuracy score of 92.59 and f1 score of 92.95 after tuning its hyperparameters.