IE7374 FINAL PROJECT REPORT

US Census Income Prediction (1994 – 1995)

Shubham Patidar

Patidar.sh@northeastern.edu

Submission Date: 8/8/22

PROBLEM SETTING

Given a dataset consisting of information about people. The goal is to predict whether the income of a person is 50000, more than 50000, or less based on the columns given.

PROBLEM DEFINITION

The dataset consists of 199523 rows and 42 Columns. The set is for US Census income from the year 1994 - 1995. It includes details about the person such as age, sex, place, education, type of employment, minimum age, and so on. Most of the Data is either categorical or numerical. One of the biggest challenges with the given dataset is we have an imbalanced target variable which has to be balanced using the sampling technique for optimum prediction.

The problem will classify based on the people's information, whether they have an income of 5000 or more or less than the 50000. Our approach will be to implement the classification model such as SVM, Logistic regression, and Naive Byes and compare those models based on accuracy and running time.

DATA SOURCE

The Data is taken from the UCI Machine learning repository. Following is the link for the storage: https://archive.ics.uci.edu/ml/datasets/Census-Income+%28KDD%29.

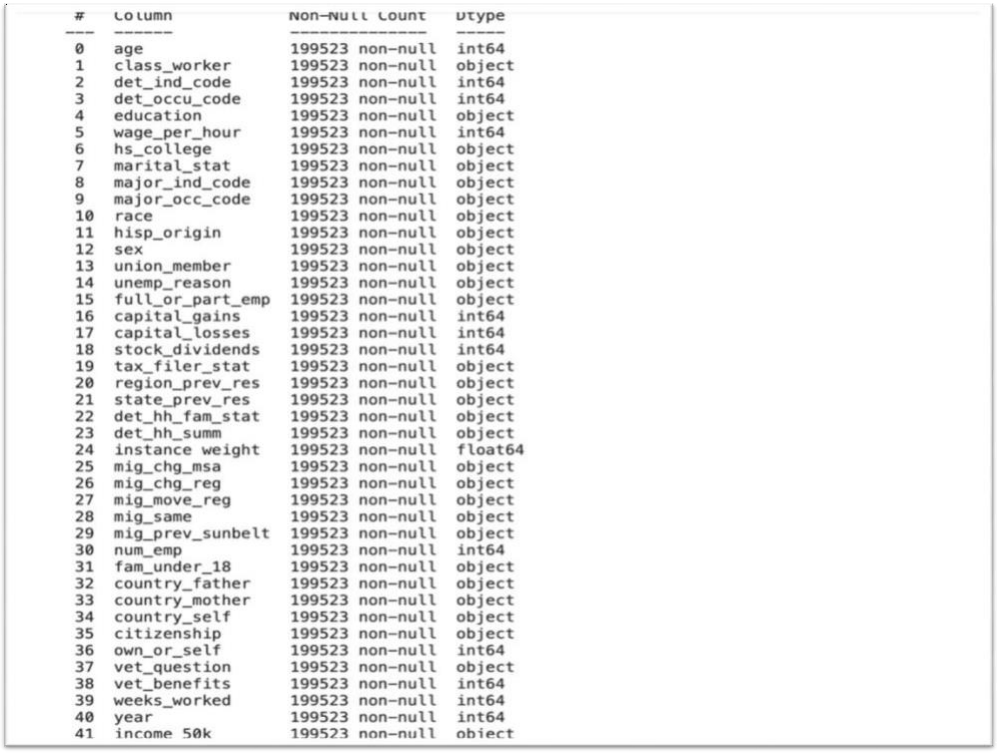
The donor for the repository is Ronny Kohavi and Barry Becker.

DATA DESCRIPTION

We have 2,99,285 instances and 42 attributes, with 29 categorical features and 13 numeric features.

1. Age – Age of the worker
2. class_worker - Class of worker
3. det_ind_code - Industry code
4. det_occ_code - Occupation code
5. education - Level of education
6. wage_per_hour- Wage per hour
7. hs_college - Enrolled in educational institution last week Enrolled in educational institution last week
8. marital_stat - Marital status
9. major_ind_code - Major industry code
10. major_occ_code - Major occupation code
11. race - Race
12. hisp_origin - Hispanic origin
13. sex - Sex
14. union_member - Member of a labor union
15. unemp_reason - Reason for unemployment
16. full_or_part_emp - Full- or part-time employment status

17. capital_gains - Capital gains
18. capital_losses - Capital losses
19. stock_dividends - Dividends from stocks
20. tax_filer_stat - Tax filer status
21. region_prev_res - Region of previous residence
22. state_prev_res - State of the previous residence
23. det_hh_fam_stat - Detailed household and family status
24. det_hh_summ - Detailed household summary in household
25. mig_chg_msa - Migration code - change in MSA
26. mig_chg_reg - Migration code - change in region
27. mig_move_reg - Migration code - move within region
28. mig_same - Live in this house one year ago
29. mig_prev_sunbelt - Migration - previous residence in sunbelt
30. num_emp - Number of persons that worked for the employer
31. fam_under_18 - Family members under 18
32. country_father - Country of the birth father
33. country_mother - Country of the birth mother
34. country_self - Country of birth
35. citizenship - Citizenship
36. own_or_self - Own business or self-employed?
37. vet_question - Fill included questionnaire for Veterans Administration
38. vet_benefits - Veterans benefits
39. weeks_worked - Weeks worked in the year
40. year - Year of the survey
41. income_50k - Income less than or greater than $50,000
42. edu_year - Number of years of education



#	Column	Non-Null Count	Dtype
0	age	199523 non-null	int64
1	class_worker	199523 non-null	object
2	det_ind_code	199523 non-null	int64
3	det_occu_code	199523 non-null	int64
4	education	199523 non-null	object
5	wage_per_hour	199523 non-null	int64
6	hs_college	199523 non-null	object
7	marital_stat	199523 non-null	object
8	major_ind_code	199523 non-null	object
9	major_occ_code	199523 non-null	object
10	race	199523 non-null	object
11	hisp_origin	199523 non-null	object
12	sex	199523 non-null	object
13	union_member	199523 non-null	object
14	unemp_reason	199523 non-null	object
15	full_or_part_emp	199523 non-null	object
16	capital_gains	199523 non-null	int64
17	capital_losses	199523 non-null	int64
18	stock_dividends	199523 non-null	int64
19	tax_filer_stat	199523 non-null	object
20	region_prev_res	199523 non-null	object
21	state_prev_res	199523 non-null	object
22	det_hh_fam_stat	199523 non-null	object
23	det_hh_summ	199523 non-null	object
24	instance weight	199523 non-null	float64
25	mig_chg_msa	199523 non-null	object
26	mig_chg_reg	199523 non-null	object
27	mig_move_reg	199523 non-null	object
28	mig_same	199523 non-null	object
29	mig_prev_sunbelt	199523 non-null	object
30	num_emp	199523 non-null	int64
31	fam_under_18	199523 non-null	object
32	country_father	199523 non-null	object
33	country_mother	199523 non-null	object
34	country_self	199523 non-null	object
35	citizenship	199523 non-null	object
36	own_or_self	199523 non-null	int64
37	vet_question	199523 non-null	object
38	vet_benefits	199523 non-null	int64
39	weeks_worked	199523 non-null	int64
40	year	199523 non-null	int64
41	income_50k	199523 non-null	object

METHODOLOGY

To solve our problem, we carried out the following steps:

1. Data Exploration
2. Data Cleaning and Feature Engineering
3. Data Manipulation
4. Exploratory Data Analysis
5. Model Implementation
   - 5.1 Data Normalization
   - 5.2 Data Sampling
     - 5.2.1 Undersampling
     - 5.2.2 Oversampling
     - 5.2.3 SMOTE
   - 5.3 Principal Component Analysis
   - 5.4 Logistic Regression
   - 5.5 SVM
   - 5.6 Naïve Bayes

1. DATA EXPLORATION

In data exploration, we perform our initial analysis and learn about the characteristics of our dataset.

```
[ ] data.describe()
```

	age	det_ind_code	det_occu_code	wage_per_hour	capital_gains	capital_losses	stock_dividends	instance weight	num_emp	own_or_self	vet_benefits	weeks_worked	year
count	199523.000000	199523.000000	199523.000000	199523.000000	199523.00000	199523.000000	199523.000000	199523.000000	199523.000000	199523.000000	199523.000000	199523.000000	199523.000000
mean	34.494199	15.352320	11.306556	55.426908	434.71899	37.313788	197.529533	1740.380269	1.956180	0.175438	1.514833	23.174897	94.499672
std	22.310895	18.067129	14.454204	274.896454	4697.53128	271.896428	1984.163658	993.768156	2.365126	0.553694	0.851473	24.411488	0.500001
min	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	37.870000	0.000000	0.000000	0.000000	0.000000	94.000000
25%	15.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	1061.615000	0.000000	0.000000	2.000000	0.000000	94.000000
50%	33.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	1618.310000	1.000000	0.000000	2.000000	8.000000	94.000000
75%	50.000000	33.000000	26.000000	0.000000	0.00000	0.000000	0.000000	2188.610000	4.000000	0.000000	2.000000	52.000000	95.000000
max	90.000000	51.000000	46.000000	9999.000000	99999.00000	4608.000000	99999.000000	18656.300000	6.000000	2.000000	2.000000	52.000000	95.000000

While using the describe function, we noticed that attributes "Wage_per_hour," "Capital_gains," "Capital_Loss," and "Stock_dividends" have a minimum vvalueaof0.
This means that the Data is missing for these features.

Also, attributes like “det_ind_code” and “det_occu_code” are index numbers. So, we can remove these columns as well.

Next, we want to check for the "Frequency of values of Categorical Attribute."s." For that we have created a function that would return the result.

```
# Defining a function for counting the values for each attribute

def value_count(col):
  return data[col].value_counts()
```

After utilizing the above function for all the categorical values, we notice some attributes with Missing values ("Not in Universe").

Some of them are listed below –

major_occ_code

```
[ ] value_count('major_occ_code')

Not in universe                    100684
Adm support including clerical      14837
Professional specialty              13940
Executive admin and managerial      12495
Other service                       12099
Sales                               11783
Precision production craft & repair 10518
Machine operators assmblrs & inspctrs  6379
Handlers equip cleaners etc         4127
Transportation and material moving   4020
Farming forestry and fishing        3146
Technicians and related support     3018
Protective services                 1661
Private household services            780
Armed Forces                           36
Name: major_occ_code, dtype: int64
```

3. Hs_College

```
[ ] value_count('hs_college')

Not in universe         186943
High school               6892
College or university     5688
Name: hs_college, dtype: int64
```

Since most of the values are stated as "Not in Universe," it is, "er to drop the attributes to avoid bias and decreasing thedecreaseof the model.

Some attributes have missing values in form of "?" –

mig_move_reg

```
value_count('mig_move_reg')

?                                99696
Nonmover                         82538
Same county                       9812
Different county same state       2797
Not in universe                   1516
Different state in South           973
Different state in West            679
Different state in Midwest         551
Abroad                             530
Different state in Northeast       431
Name: mig_move_reg, dtype: int64
```

So, we will remove these attributes as well. Also, there are some attributes with 10 + values, so it will be difficult to perform dummy encoding or label encoding.

2. DATA CLEANING AND FEATURE ENGINEERING

After understanding and finding out which variables are important, we will clean and pre-process the data by -

A. Dropping Irrelevant Columns for Categorical Attributes

Since most of the columns either had "?" and "Not in Universe" values, we will drop them.

```
[ ] data_copy.drop(columns = ['class_worker', 'hs_college', 'major_ind_code', 'major_occ_code', 'union_member', 'unemp_reason',
                              'region_prev_res', 'state_prev_res' ,'det_hh_fam_stat', 'mig_chg_msa' ,'mig_chg_reg' ,'mig_move_reg',
                              'mig_same' ,'mig_prev_sunbelt', 'fam_under_18' ,'country_father', 'country_mother', 'country_self', 'vet_question'],
                    axis = 1, inplace = True)
```

B. Handling numeric columns

Since, det_ind_code and det_occu_code columns are index values, we will drop them.

```
# handling numeric columns

data_copy.drop(columns = ['det_ind_code', 'det_occu_code'], axis = 1, inplace = True)
```

C. Feature Engineering

In this step, we will be replacing "Categorical columns" with appropriate values, so that it becomes easy for us to perform Label encoding and One-Hot Encoding.

Some of them are shown below –

1. Martial Status

```
[ ] value_count_Cat("marital_stat")
```

```
Never married                        86485
Married-civilian spouse present      84222
Divorced                             12710
Widowed                              10463
Separated                             3460
Married-spouse absent                 1518
Married-A F spouse present             665
Name: marital_stat, dtype: int64
```

```
[ ] # Right strip the marital status column
    data_copy["marital_stat"] = data_copy["marital_stat"].str.strip()
```

```
[ ] # Replacing with appropriate values
    data_copy["marital_stat"].replace({"Separated" : "Divorced", "Never married": "Not married",
                                       "Married-A F spouse present": "Married-spouse present"}, inplace = True)
```

```
value_count_Cat("marital_stat")
```

```
Not married                          86485
Married-civilian spouse present      84222
Divorced                             16170
Widowed                              10463
Married-spouse absent                 1518
Married-spouse present                 665
Name: marital_stat, dtype: int64
```

For example, for "Marital_Status" –

- We rstrip() the column to make the values consistent.
- Replacing the values with same meaning for better readability.
- Count values to check

3. Hisp Origin

```
data_copy['hisp_origin'] = data_copy['hisp_origin'].str.strip()
```

+ Code + Text

```
[ ] value_count_Cat("hisp_origin")
```

```
All other                   171907
Mexican-American              8079
Mexican (Mexicano)            7234
Central or South American     3895
Puerto Rican                  3313
Other Spanish                 2485
Cuban                         1126
NA                             874
Do not know                    306
Chicano                        304
Name: hisp_origin, dtype: int64
```

```
# Replacing with appropriate values
data_copy["hisp_origin"].replace({"All other" : "Others", "Other Spanish" : "Spanish", "NA" : "Others", "Mexican (Mexicano)" : "Mexican",
                                  "Do not know" : "Others", "Mexican-American": "Mexican"}, inplace = True)
```

```
[ ] value_count_Cat("hisp_origin")
```

```
Others                      173087
Mexican                      15313
Central or South American     3895
Puerto Rican                  3313
Spanish                       2485
Cuban                         1126
Chicano                        304
Name: hisp_origin, dtype: int64
```

3. DATA MANIPULATION

In this module, we will check the `*Data Distribution*` of data for Numeric Attributes and replace the irrelevant or missing values with "Mean".

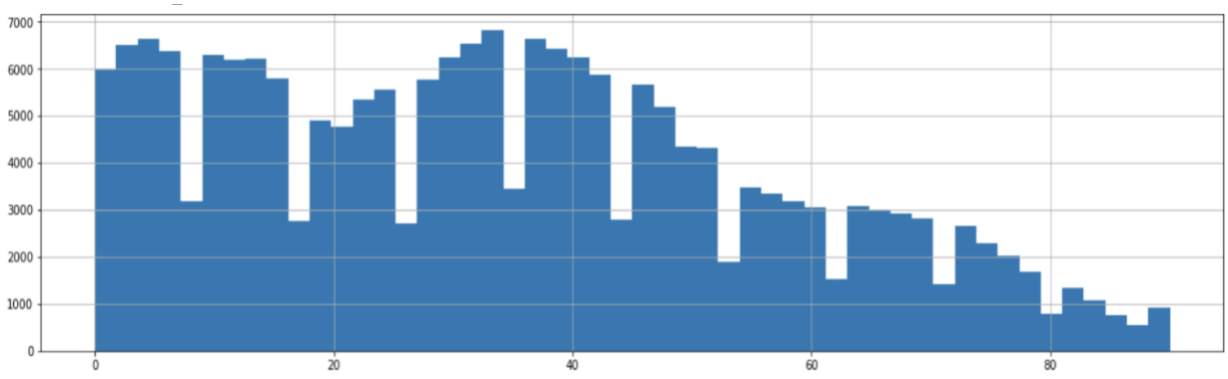
I also tried to understand the spread of our data. Following are the things we focused on:

- Number of unique entries in each column
- Highest occurring entry in the columns
- And frequency of the entry

Understanding spreads of Data

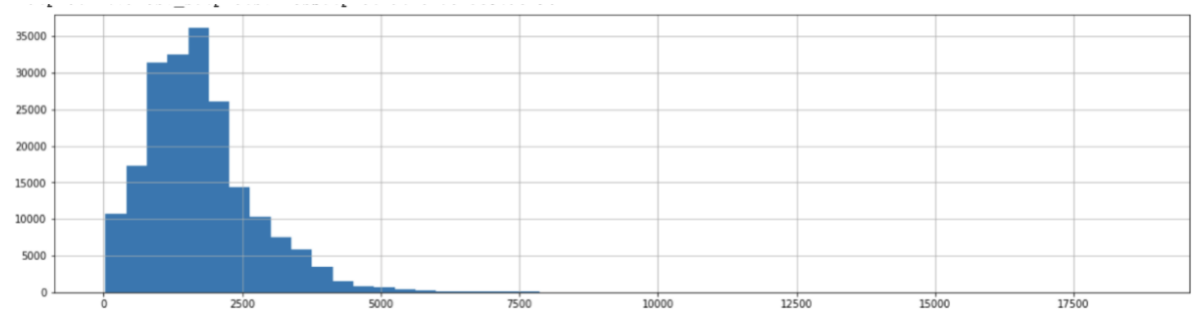
Age -

Most the data we have is of the young population and the distribution of data as per age can be fully understood using this simple histogram below -



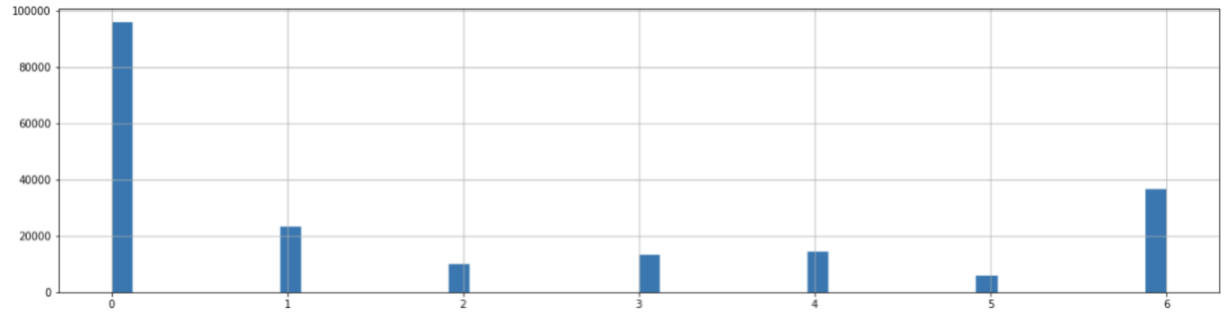
Instance Weight -

The distribution of data of instance weight can roughly be compared to Gaussian distribution. And the distribution can be accessed using following graph:



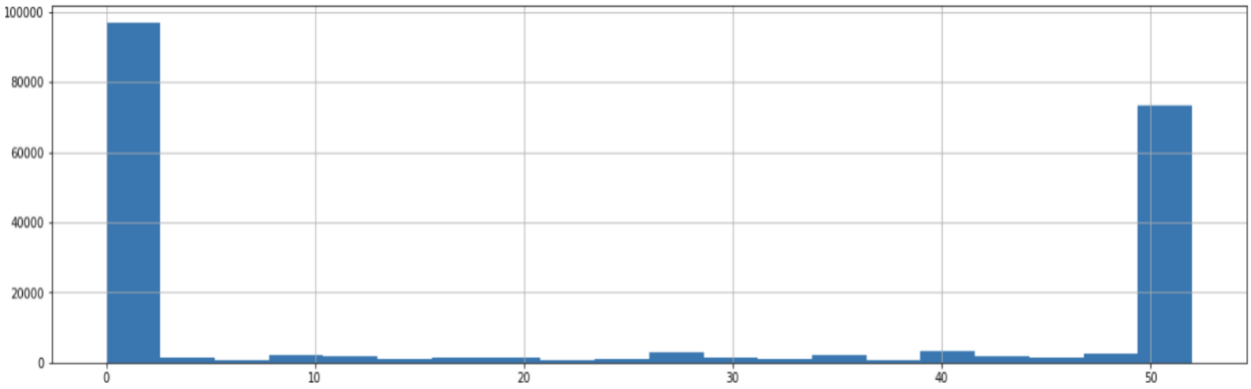
Emp_Num -

Graph showing the number of employments ranging from 0 to 6



Weeks_worked -

Number of hours an employee spends working is described in the graph below. More the number of hours an employee works, the more will be his chances to getting a higher income compared to other people who works less.



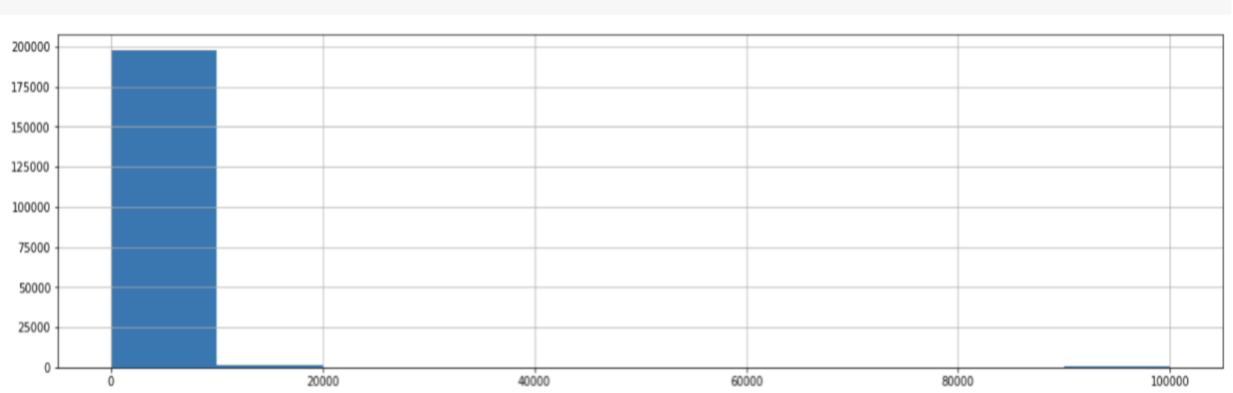
Note: For most the histograms shown above, zero has to be ignored since zero for most cases denotes the number of null entries and it interferes with the data. Of course, most of these missing values has been taken care of in the data cleaning part.

Now the graphs which cannot show the Actual distribution of data because of high number of missing values or null values are visualization below using the graphs:

All the features in the below mentioned category has been dealt with using all the different techniques.

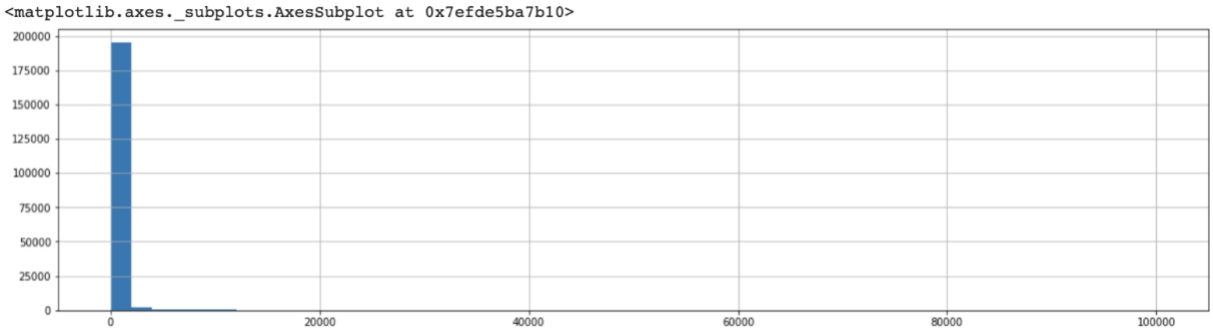
Capital gain –

Graph showing too many null values for capital gain. Also, since there is a huge difference between the entries, we need the normalize the number to get an excellent margin.



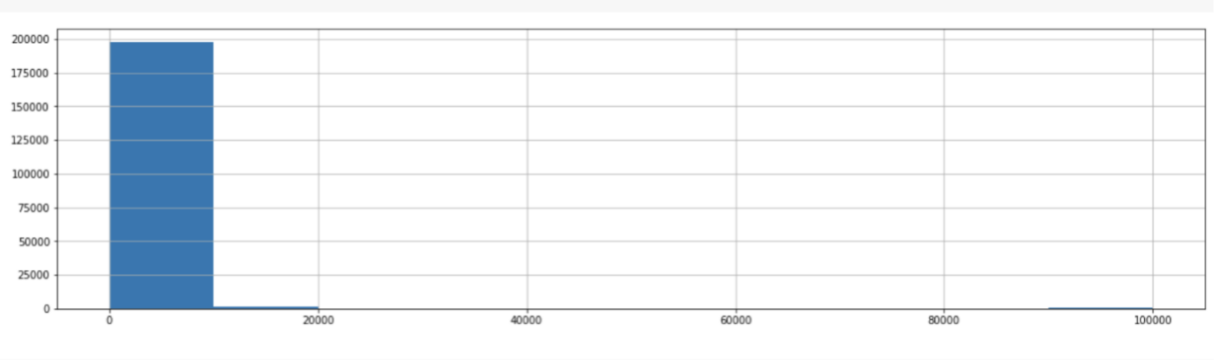
Wage per hour -

The graph shows the same kind of errors as the one mentioned above.

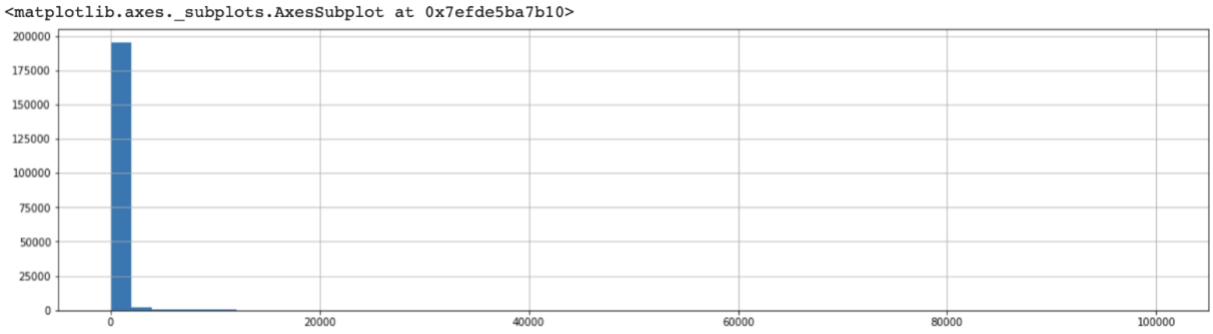


Capital gains –

Since most of the data in the given graph are null, which is either zero or not available, we could use the option to eliminate the whole column from the Machine learning model, or we could use other techniques to handle the data. Data handling can be seen in the forwarding materials.



Stock dividends -



4. DATA ENCODING

In this module, I will convert categorical values to numeric using *Label encoding* and *Dummy encoding*.

A. LABEL ENCODING

I implemented *label encoding* for the following attributes –

1. Sex
2. Citizenship
3. Tax filer status
4. income_50k (Target Variable)
5. year - 1994, 1995

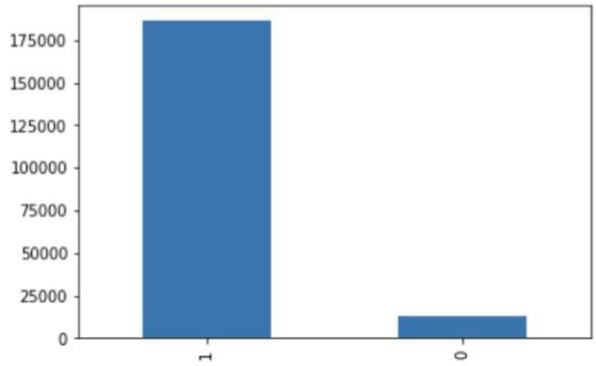
created a function for *Label encoding* and plotted the results for better understanding.

```
[ ]  # Label Encoding

     def label_encoding(col):
       # Creating an instance of label_encoder
       label_encoder = LabelEncoder()
       data_encoded[col] = label_encoder.fit_transform(data_encoded[col])
       return data_encoded[col].value_counts().plot(kind = 'bar')
```

For example, for the "Citizenship" column –

```
label_encoding('citizenship')
```

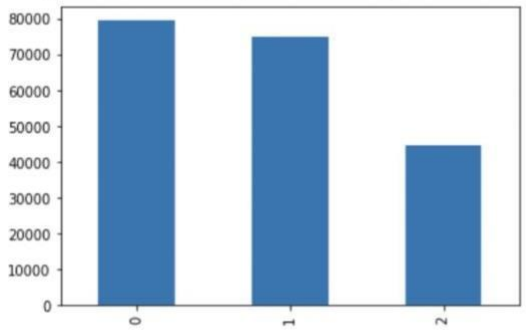
```
<matplotlib.axes._subplots.AxesSubplot at 0x7faf15466f90>
```



For Tax Filler Status –

```
label_encoding('tax_filer_stat')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7faf1548a090>
```



B. DUMMY ENCODING

Implementng *Dummy encoding* for the following attributes –

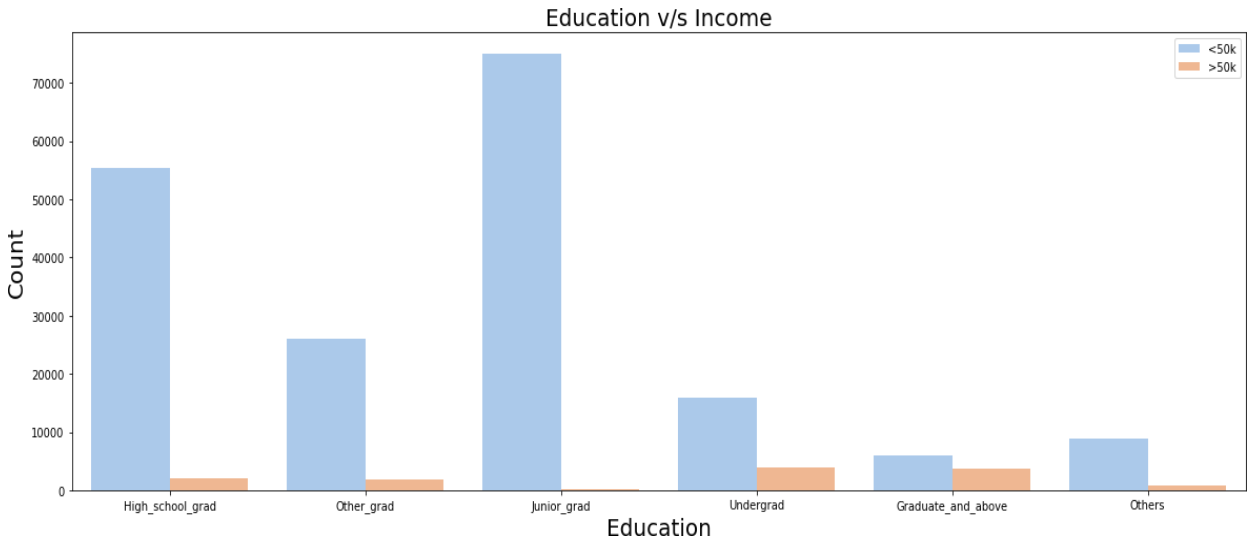
1. Education
2. Marital Status
3. Race
4. Hisp Origin
5. Full_or_part_emp
6. Detailed household summary in the household (det_hh_summ)

```
# Encoding Education
edu = pd.get_dummies(data_encoded['education'], drop_first=True, prefix='Edu')
data_encoded = data_encoded.join(edu)
data_encoded = data_encoded.drop(columns = 'education')
```

5. EXPLORATORY DATA ANALYSIS

To clearly understand the importance of each feature in the dataset and how much each feature contributes to the predictive variable, we can draw relation bar graphs. In other words, we can figure out the relation between the features using the following graphs

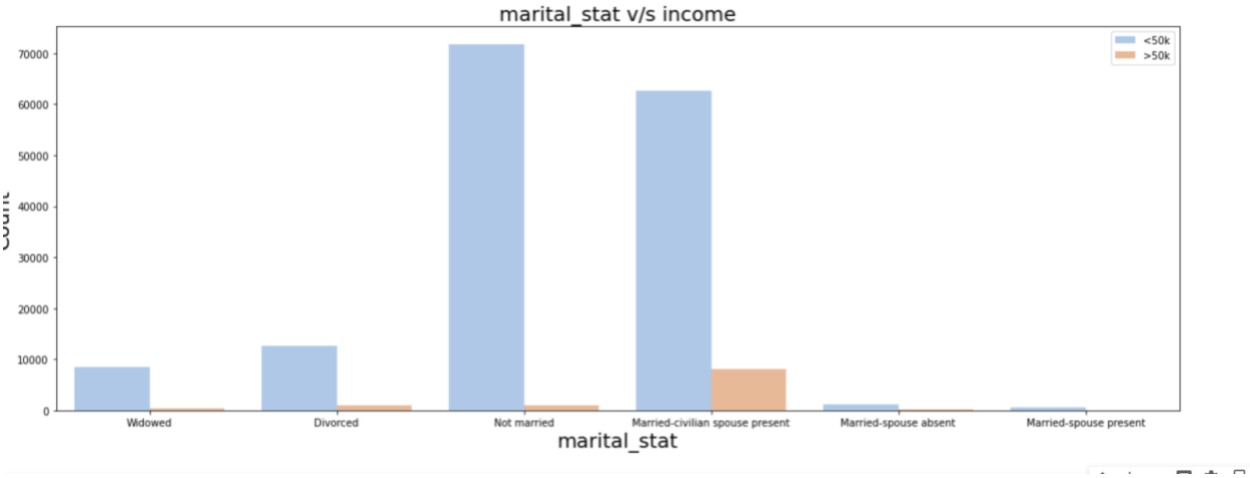
A. What is the `Education status` of adults with income less than 50k?



Insights:

- Graduate people earn the most out of all the people
- The junior grade category has the least amount of income
- Education is a crucial factor in predicting the income of the people since it affects each category of people

B. What is the Marital Status of adults with an income of more than 50k?

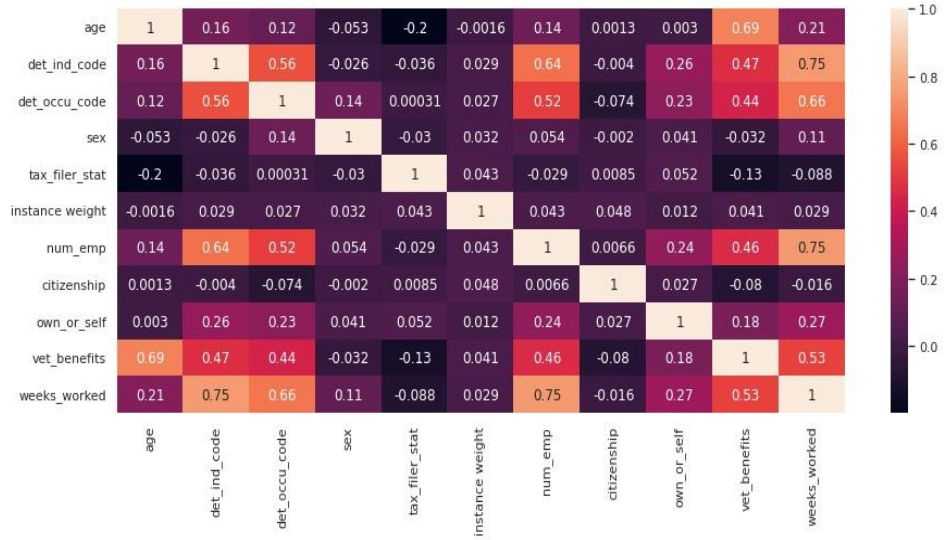


Insights:

- As it shows in the plot, There is a significant difference between married_civilant_present groups and others.
- We can observe from the given dataset that people with a spouse earn the most compared to others.
- The divorced and widowed groups still earn more than married_spous_absent and married_spouse present, but it is still not comparable with married_civilian_spouse present

C. Co-relation Analysis

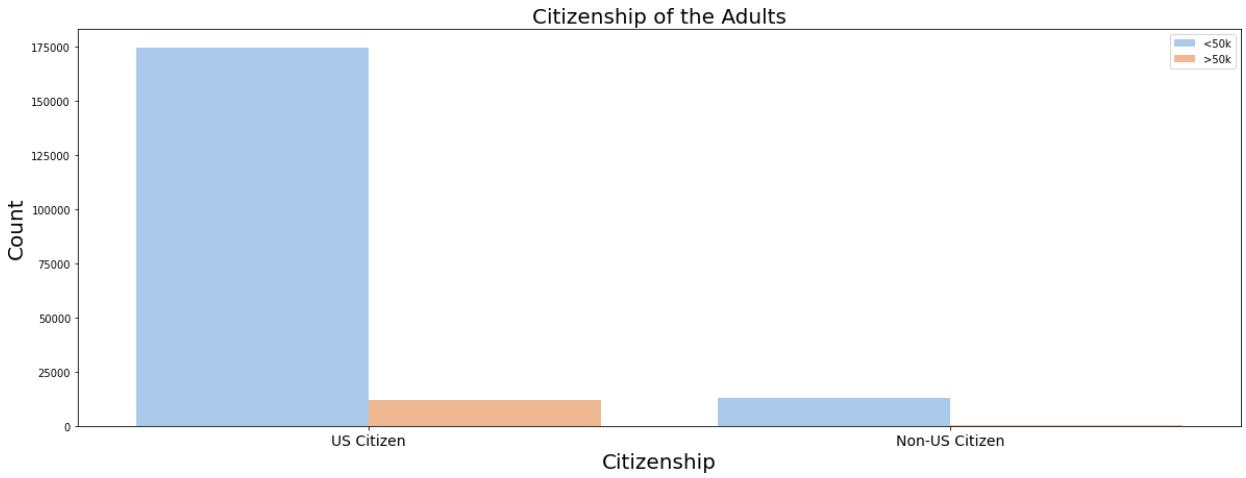
We can find the co-relation between all the columns using the heat map. Lighter color implies more relationship between those columns and hence plays a more significant part in the predicting variable.



Insights:

- It is noticeable from the above graph that the relation between Age and Net benefit are highly co-related.
- And features like "Tax_filter_status" and "Age" do not have much relation.

D. What is the Citizenship of the adults?



Insights:

- It is apparent in the data set that we are dealing with the status of us citizens rather than non_us citizens.
- Almost 90 percent of people who has us citizen status earn less than 50000.
- As the plot shows us, 100 percent of the people with non-us_citizen status earn less than 50000

6. MACHINE LEARNING

In this module, we will implement machine learning models –

1. Normalize the dataset using *Min-max scaling*
2. Since our Target variable (Income_50k) is "*imbalanced*," we will be using Sampling methods to upscale the data –
   - Undersampling Method,
   - Oversampling Method,
   - SMOTE Method
3. Implement Principal Component Analysis (PCA) to check which attributes have most of the information
4. Implement Machine Learning algorithms on this dataset -
   - Logistic Regression
   - SVM
   - Naive Bayes

SPLITTING THE DATASET

For implementing our machine learning algorithms, we will split the data into training and testing data in a 0.7 : 0.3. i.e., training is 70%, and testing is 30%

DATA NORMALIZATION

Min-max normalization is one of the most common ways to normalize data. For every feature, the minimum value of that feature gets transformed into a 0, the maximum value transforms into a 1, and every other matter transforms into a decimal between 0 and 1.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

To normalize the dataset, we have created a function –

```
# data normalization
data_encoded_1 = data_encoded.copy()

for column in data_encoded_1.columns:
    data_encoded_1[column] = (data_encoded_1[column] - data_encoded_1[column].min()) / (data_encoded_1[column].max() - data_encoded_1[column].min())

# viewing normalized data
display(data_encoded_1.head(5))
```

The results are shown after normalizing the dataset –

	age	sex	tax_filer_stat	instance weight	num_emp	citizenship	own_or_self	year	income_50k	Edu_High_school_grad	...	Hisp_Spanish	Emp_Full-Time	E
0	0.811111	0.0	0.5	0.089278	0.000000	1.0	0.0	1.0	1.0	1.0	...	0.0	0.0	
1	0.644444	1.0	1.0	0.054552	0.166667	1.0	0.0	0.0	1.0	0.0	...	0.0	0.0	
2	0.200000	0.0	0.5	0.051244	0.000000	0.0	0.0	1.0	1.0	0.0	...	0.0	0.0	
3	0.100000	0.0	0.5	0.092396	0.000000	1.0	0.0	0.0	1.0	0.0	...	0.0	0.0	
4	0.111111	0.0	0.5	0.055391	0.000000	1.0	0.0	0.0	1.0	0.0	...	0.0	0.0	

5 rows × 38 columns

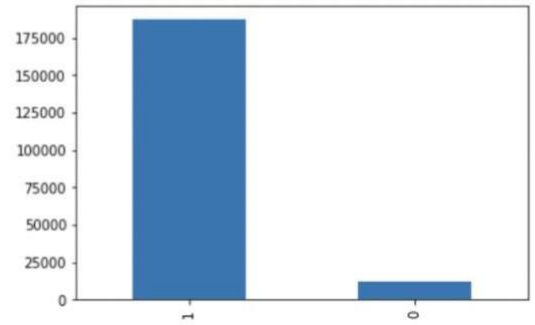
The above data has been normalized using a min-max scale. All the Data is clubbed into the range between zero to one. The categorical will gets zero and one, and the numeric will be set between zero and one.

DATA SAMPLING

Since our target variable is *imbalanced*, i.e., there are a more significant number of 1's and a smaller number of zeros, and to avoid *Overfitting*, we will use data sampling techniques –

```
data_encoded.income_50k.value_counts().plot(kind = "bar")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7faf15358cd0>
```



Since our target variable is imbalanced, we are going to implement 3 sampling techniques –

1. UNDERSAMPLING

In Undersampling, we balance the dataset by keeping all the data from the minority class (0) and decreasing the sample size of the majority class (1)

```
[ ]  # Undersampling
     def undersampled(X1, y1):
       rus = RandomUnderSampler(random_state=21)
       X_Usampled, y_Usampled = rus.fit_resample(X1, y1)
       return X_Usampled, y_Usampled
```

```
[ ]  pd.Series(y_Usampled).value_counts()
```

```
0.0    12382
1.0    12382
Name: income_50k, dtype: int64
```

Here, the new sample size is 24,764.

2. OVERSAMPLING

In Oversampling, we balance the dataset by keeping all the data from the Majority class (1) and increasing the sample size of the Minority class (0)

```
[ ]  # Oversampling
     def oversampled(X1, y1):
       ros = RandomOverSampler(random_state=21)
       X_Osampled, y_Osampled = ros.fit_resample(X1, y1)
       return X_Osampled, y_Osampled
```

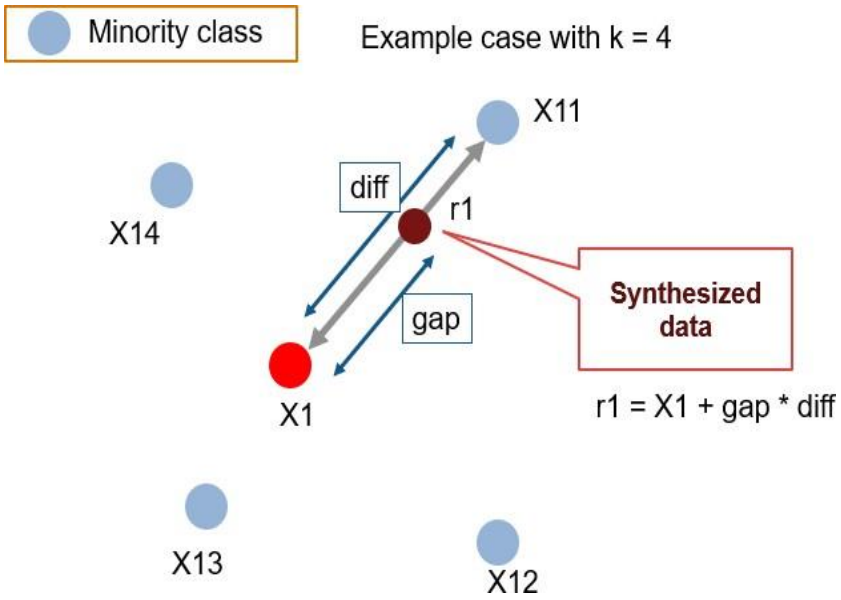
```
[ ]  pd.Series(y_Osampled).value_counts()
```

```
1.0    187141
0.0    187141
Name: income_50k, dtype: int64
```

Here, the new sample size is 3,74,282.

3. SMOTE

It is an oversampling technique in which the class distribution is balanced by randomly creating synthetic instances of the minority class.



It uses K-means to replicate the data and ensures that the gap between the original and synthetic data is significantly less.

```
[ ]  # SMOTE
     def smote(X1, y1):
       sm = SMOTE(random_state=21)
       X_SMOTE, y_SMOTE = sm.fit_resample(X1, y1)
       return X_SMOTE, y_SMOTE
```

```
[ ]  pd.Series(y_SMOTE).value_counts()

     1.0    187141
     0.0    187141
     Name: income_50k, dtype: int64
```

Here, the new sample size is 3,74,282.

DIMENSION REDUCTION TECHNIQUE

Applying Principal Component Analysis (PCA)

Principal component analysis, or PCA, is a statistical procedure allowing you to summarize the information in large data tables using a smaller set of "summary indices" that can be more easily visualized and analyzed. In short, PCA projects the original dataset in the direction that captures most of the variance.

STEPS TO PERFORM PCA:

1. We first Standardize the dataset. It is necessary to standardize the dataset so that all the features have the same Standard Deviation.

We have created a function "Standardize" and passed the Standard scalar() library that returns normalized data points.

```
[ ]  def Standardize(data):
        scaler = StandardScaler()
        scaler.fit(data)
        X_pca = scaler.transform(data)
        return X_pca
```

After standardizing, the normalized Data looks like this –

```
[ ]  Standardize(data_encoded)

     array([[ 1.72587866, -0.84973982, -0.78223502, ...,  4.42301048,
             -0.02572967, -0.51409351],
            [ 1.05355971, -0.62834271,  1.57002771, ..., -0.22609035,
             -0.02572967, -0.51409351],
            [-0.73929082, -0.84973982, -0.78223502, ..., -0.22609035,
             -0.02572967, -0.51409351],
            ...,
            [ 0.56052581, -0.84973982, -0.78223502, ..., -0.22609035,
             -0.02572967, -0.51409351],
            [-0.82893335, -0.84973982, -0.78223502, ..., -0.22609035,
             -0.02572967, -0.51409351],
            [-0.11179313,  1.47492979,  1.29329092, ..., -0.22609035,
             -0.02572967, -0.51409351]])
```

2. Make the values of the data mean centered
3. Calculate the covariance of the matrix

```
def fit(self,X):
    self.mean = np.mean(X,axis=0)
    X = X - self.mean
    cov = np.cov(X.T)
    # Calculate Eigan Values and Eigan vectors
    eigenvalues, eigenvectors = np.linalg.eig(cov)
    eigenvectors = eigenvectors.T
    indexs = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[indexs]
    eigenvectors = eigenvectors[indexs]
    top_comp = eigenvalues[0:self.n_components]
    print("The Variance Captured by", self.n_components, "components:", round(top_comp.sum()/eigenvalues.sum(),2)*100, "%",'\n')
    self.components = eigenvectors[0:self.n_components]
```

4. Find the Eigenvalues and Eigenvectors of the covariance matrix
5. Sort the Eigenvalues using argsort
6. Select top "n" components
7. Calculate the variance captured by the "n" components

```
print("Variance Captured by", self.n_components, "components:", round(top.sum()/eigenvalues.sum(),2)*100, "%",'\n')
self.components = eigenvectors[0:self.n_components]
```

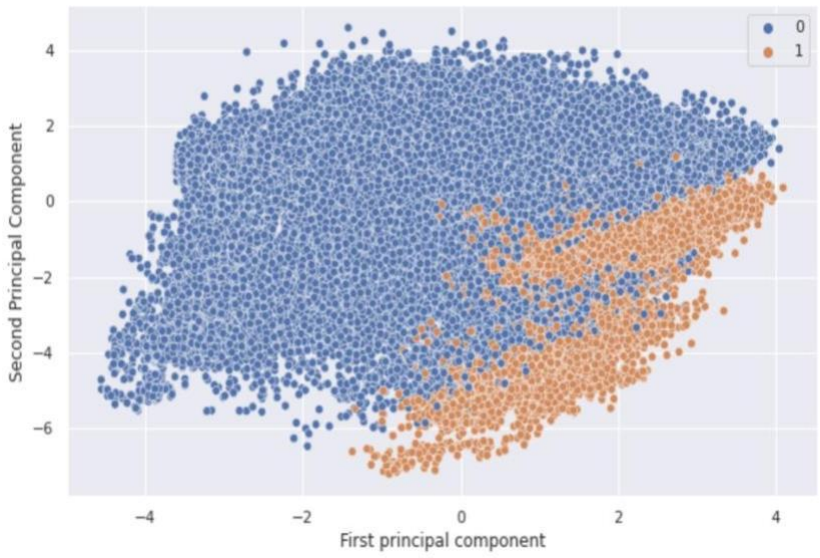
RESULTS:

After selecting 30 components, we noticed that 30 components capture 96% of the variance.

```
Variance Captured by 30 components: 96.0 %

[[ 0.58115918  2.77642369  1.96012866 ...  0.45149113  0.54653692
   0.04456479]
 [-0.54818397 -0.8632638   1.0835388  ... -0.14574804  0.38196753
   0.65122866]
 [ 3.01101173  0.87409758 -0.57643831 ... -0.95777306  0.10989833
  -1.08970611]
 ...
 [-1.70992454  0.50654652  0.56605073 ...  0.21913738 -0.48928448
  -0.16905995]
 [ 2.62852515  1.13081004  1.21511778 ... -0.26869084 -0.05634399
  -1.42163696]
 [-0.18551435 -2.91800536  0.30905771 ... -3.52490051 -0.13037682
  -0.44822072]]
```

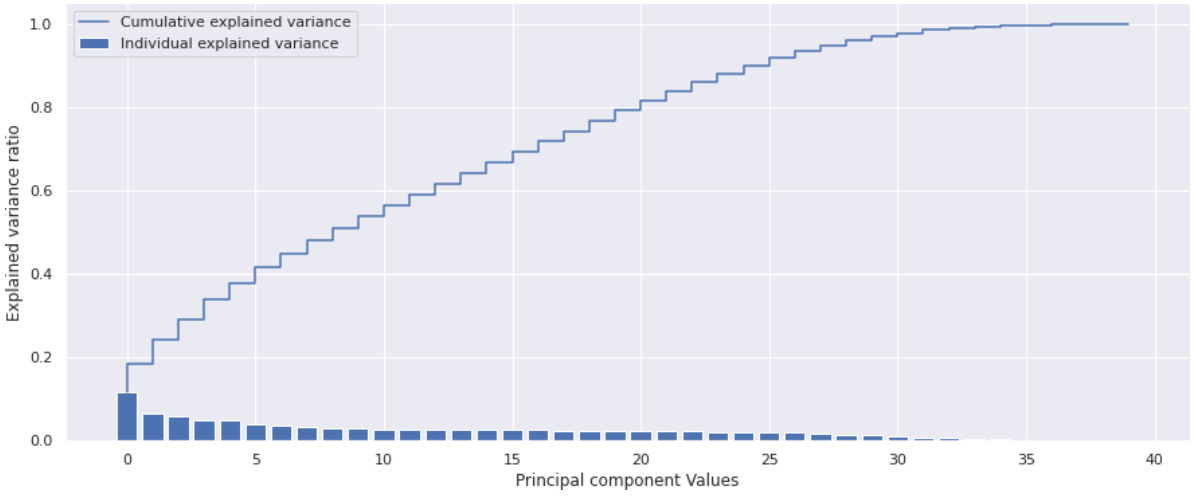
We are going to use the scatter plot to summarize the contents of 'Principal component 1' and 'Principal component 2' using scatter plot:

```
<matplotlib.legend.Legend at 0x7fd672a98090>
```



Since the approach of the PCA is to choose the highest variance, the first principal component contains the most information as it appears in the above plot.

Graph for Individual Explained Variance and Cumulative Explained Variance



MODEL IMPLEMENTATION

We have solved the classification problem using the following algorithms.

1. Logistic Regression
2. SVM
3. Naïve Bayes

1. LOGISTIC REGRESSION:

Logistic regression is one of the standard classification algorithms in machine learning, and it has the sigmoid function in which the output will be zero and one.

We have implemented Logistic Regression for the following dataset –

A) Original Data

To perform logistic regression, we have split the dataset into training and testing data (0.7 : 0.3)

```
def datasetReader(self):
  X1 = data_encoded_1.copy()
  X1.drop('income_50k', axis = 1, inplace = True)
  y1 = data_encoded['income_50k']
  X_train, X_test, y_train, y_test = train_test_split(X1, y1, test_size=0.3, stratify=y1, random_state=21)
  return X_train, y_train, X_test, y_test
```

The primary step is implementing the sigmoid function, which is classifying the data:

```
def sigmoid(self, z_value):
    sig_value = 1/(1 + np.exp(-z_value))
    return sig_value
```

We have used Negative likelihood as our Cost Function:

```
# Calculating the cost function
def costFunction(self, X, y):
    # Calculating Negative log likelihood
    pred_value_ = np.log(np.ones(X.shape[0]) + np.exp(X.dot(self.w))) - X.dot(self.w)*y
    cost_value = pred_value_.sum()
    return cost_value
```

For solving the model, we used the gradient distant, which the function would be as below:

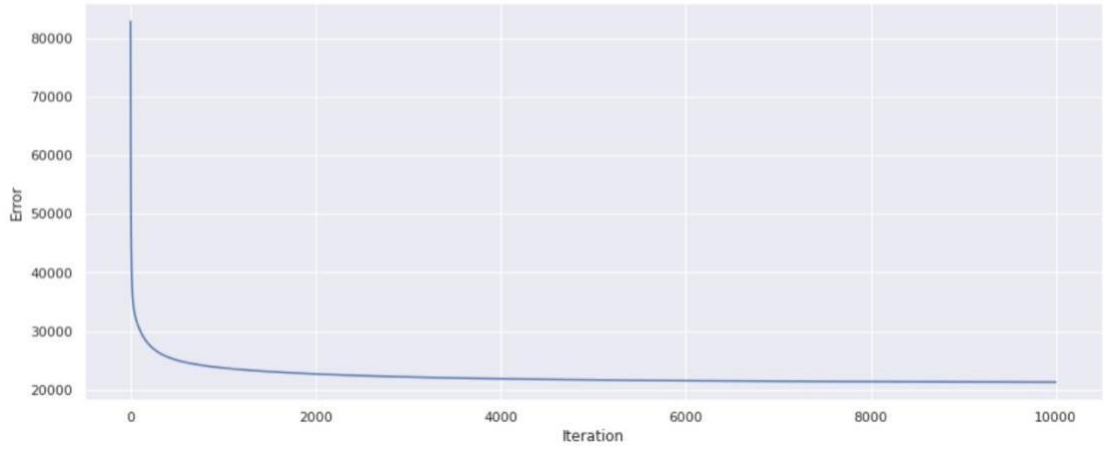
```
# Calculating gradient Descent
def gradientDescent(self, X, y):
    cost_sequences = []
    last_cost = float('inf')
    for i in tqdm(range(self.maxIteration)):
        self.w = self.w - self.learningRate * self.gradient(X, y)
        cur_cost = self.costFunction(X, y)

        diff = last_cost - cur_cost
        last_cost = cur_cost
        cost_sequences.append(cur_cost)
        if diff < self.tolerance:
            print('The model Converged')
            break

    self.plotCost(cost_sequences)
    return
```

RESULT:

With "Learning rate" as 0.00001, "Tolerance" as 0, and "maxIterations" as 10000, We got an accuracy of 94%, and the running time shows 484 seconds.

Since the model was biased (1's = 1,87,141 and 0's = 12,382), it was obvious that the accuracy would be very high.

```
Solving Logistic regression using Gradient Descent
100%|██████████| 10000/10000 [08:04<00:00, 20.65it/s]

Accuracy : 94.19999999999999
Precision : 0.95
Recall : 0.991
Time:  484.608305967
```



We have used Undersampling, oversampling, and SMOTE techniques to avoid Overfitting.

B) UnderSampled Data

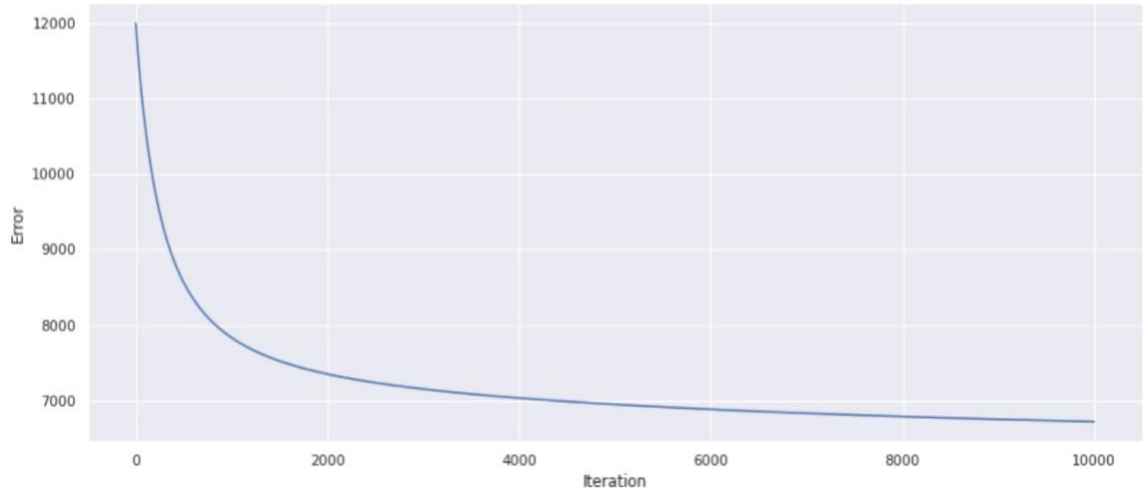
To perform logistic regression, we have split the dataset into training and testing data (0.7 : 0.3)

```
def datasetReader(self):
  X2 = data_encoded_1.copy()
  X2.drop('income_50k', axis = 1, inplace = True)
  y2 = data_encoded_1['income_50k']
  rus = RandomUnderSampler(random_state=21)
  X_Usampled, y_Usampled = rus.fit_resample(X2, y2)

  X_train, X_test, y_train, y_test = train_test_split(X_Usampled, y_Usampled, test_size=0.3, stratify=y_Usampled, random_state=42) #test_size=0.2, 0.3
  return X_train, y_train, X_test, y_test
```

RESULT:

With "Learning rate" as 0.00001, "Tolerance" as 0, and "maxIterations" as 10000, We got an accuracy of 79.9%
Also, the error value decreased significantly as we increased the number of iterations.
The running time shows 102 seconds; compared to the original Data, the running time increases.

```
Solving Logistic regression using Gradient Descent
100%|██████████| 10000/10000 [01:42<00:00, 97.50it/s]

Accuracy : 82.5
Precision : 0.856
Recall : 0.782
Time:  102.95 seconds
```



C) Oversampled Data

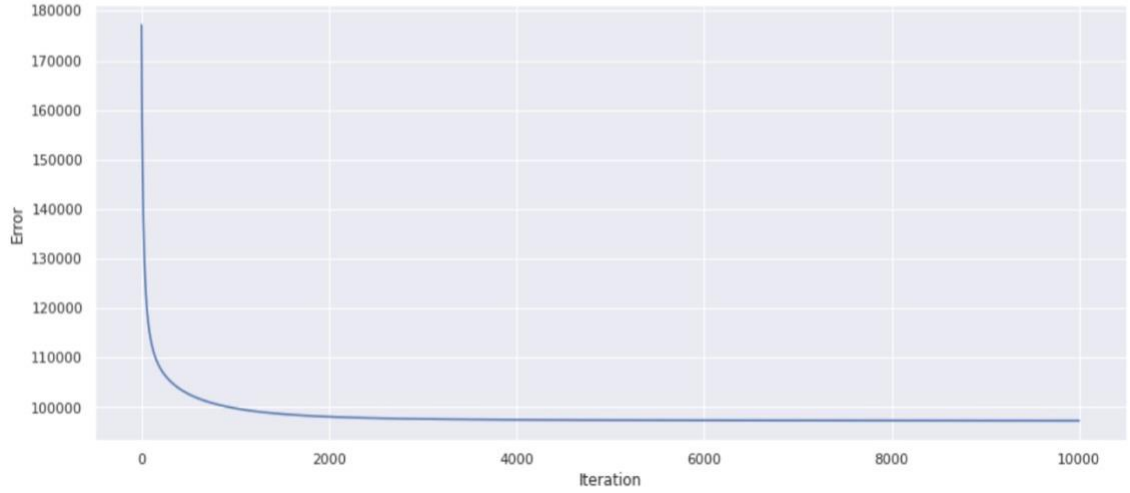
To perform logistic regression, we have split the dataset into training and testing data (0.7 : 0.3)

```
def datasetReader(self, indexes):
  X3 = data_encoded_1.copy()
  X3.drop('income_50k', axis = 1, inplace = True)
  y3 = data_encoded_1['income_50k']
  ros = RandomOverSampler(random_state=21)
  X_Osampled, y_Osampled = ros.fit_resample(X3, y3)

  X_train, X_test, y_train, y_test = train_test_split(X_Osampled, y_Osampled, test_size=0.3, stratify=y_Osampled, random_state=21)
  return X_train, y_train, X_test, y_test
```

RESULT:

With "Learning rate" as 0.00001, "Tolerance" as 0, and "maxIterations" as 10000, We got an accuracy of 83.2%.
Running time is around 824 seconds. Oversampling Running time is increased compared to the original and undersampling methods.

```
Solving Logistic regression using Gradient Descent
100%|██████████| 10000/10000 [13:42<00:00, 12.15it/s]

Accuracy : 83.2
Precision : 0.855
Recall : 0.799
Time:  824.18 seconds
```



D) SMOTE data

To perform logistic regression, we have split the dataset into training and testing data (0.7 : 0.3)

```
def datasetReader(self, indexes):
  X3 = data_encoded_1.copy()
  X3.drop('income_50k', axis = 1, inplace = True)
  y3 = data_encoded_1['income_50k']
  ros = RandomOverSampler(random_state=21)
  X_Osampled, y_Osampled = ros.fit_resample(X3, y3)

  X_train, X_test, y_train, y_test = train_test_split(X_Osampled, y_Osampled, test_size=0.3, stratify=y_Osampled, random_state=21)
  return X_train, y_train, X_test, y_test
```

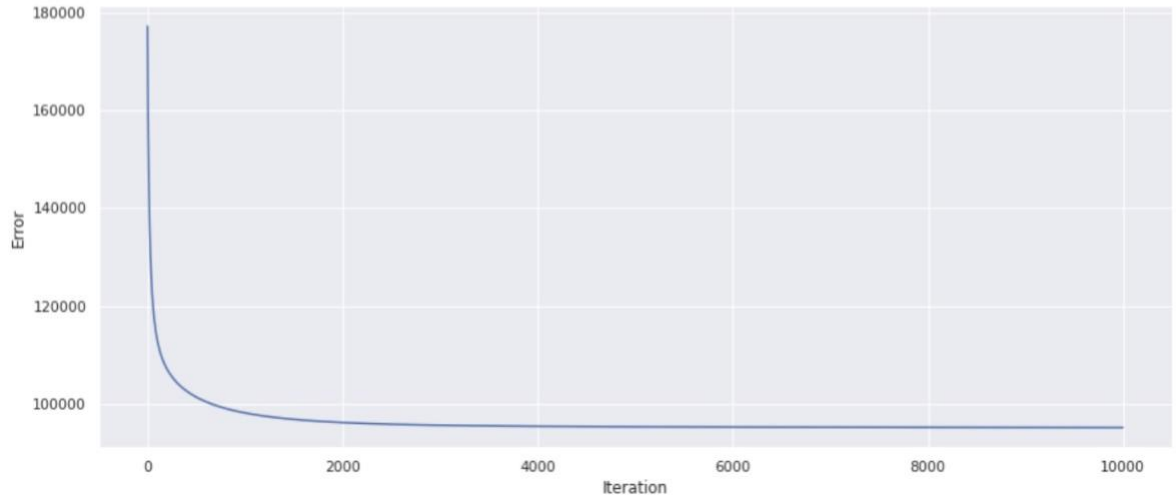
RESULT:

With "Learning rate" as 0.00001, "Tolerance" as 0, and "maxIterations" as 10000, We got an accuracy of 83.5%.

Also, we noticed that the error value decreased as we increased the number of iterations.

Running time shows around 848 seconds, which has the highest running time compared to other methods and is very close to oversampling.

```
Solving Logistic regression using Gradient Descent
  0%|          | 2/10000 [00:00<10:40, 15.62it/s]177141.66278008692
 10%|▉         | 1003/10000 [01:15<11:06, 13.50it/s]98110.61941289448
 20%|█▉        | 2002/10000 [02:31<09:53, 13.48it/s]96144.42464317448
 30%|██▉       | 3003/10000 [03:51<08:34, 13.59it/s]95586.09973286201
 40%|███▉      | 4003/10000 [05:16<09:33, 10.46it/s]95374.3151578054
 50%|████▉     | 5002/10000 [06:55<07:40, 10.84it/s]95274.11814798924
 60%|█████▉    | 6003/10000 [08:30<05:59, 11.12it/s]95216.72134334779
 70%|██████▉   | 7003/10000 [10:03<04:09, 12.01it/s]95178.54309924495
 80%|███████▉  | 8003/10000 [11:28<02:39, 12.54it/s]95150.45828082341
 90%|████████▉ | 9003/10000 [12:47<01:14, 13.38it/s]95128.4910765284
100%|██████████| 10000/10000 [14:03<00:00, 11.85it/s]

Accuracy : 83.7
Precision : 0.861
Recall : 0.803
Time:  848.04 seconds
```



INFERENCES

Out of all the Sampling methods, SMOTE data has the highest accuracy of 83.5%, as it creates synthetic instances of the minority class.

2. SUPPORT VECTOR MACHINE (SVM)

Another common classification is SVM, the most famous ML algorithm of time. However, SVM works well for complex data, the size of data matters. Since we went through some data exploration, we found we are dealing with an extensive data set, which might not be suitable for SVM but to overcome this issue, the PCA method has been applied to reduce the dimension. This method gives us the initial proper data for giving to SVM. On the other hand, we have nonlinear data, which the SVM would work well for nonlinear and complex data. Hard margin SVM uses the kernel trick inside the function to overcome nonlinearity issues.

Steps were taken during modeling:

1) We used the dataset and applied the Support Vector Machine model to the dataset.
2) The model does not perform well (high actuary) on the initial dataset because of Overfitting, which we will resolve using the sampling method.
3) After using sampling methods, improvements can be seen in the model, and the model performs better in terms of accuracy, time, and error.

Overfitting explained:
One of the constant troubles we have been facing with our dataset is the non-uniform quantity of our target variable. The dataset is imbalanced and consists of a more significant number of 1s than 0s. This makes the model biased and interferes with the model's ability to train for the zeros. The most scenario-based solution for this problem appears to be sampling. If we take samples from our target variable to train data and that eliminates the discriminatory behavior caused by more significant number of 1s.

Laplace: There is constraint on our data points, I.e., (x*w+b>= 1), so we used Laplace to accommodate constraint in our SVM equation.

$$w = w + \alpha \cdot (y_i \cdot x_i - 2\lambda w)$$

Note: Our dataset shows satisfactory results with the current SVM model. Our accuracy is ---- and implies linearity among the features; hence we did not need to apply Kernel to our SVM model.

A) Original Data:

To perform SVM, we have used *Laplace Equation*

```
if condition:
    self.w = self.w + self.lRate *(2 * self.lamda * self.w)
else:
    self.w = self.w - self.lRate*(2 * self.lamda * self.w - np.dot(x_i_value, y_[indx]))
    self.b = self.b - self.lRate * (-y_[indx])
```

Result:
Keeping the Learning rate as 0.00001, lambda = 0.001, and the number of iterations as 100, we achieved an accuracy of 94%. Since our model is imbalanced, we will compare our results with our sampled data.

```
The Accuracy value is: 94.0 %
The Precision value is: 0.94
The Recall value is: 1.0
Time:  147.58 seconds
```

B) UnderSampled data:

In Undersampling, we have reduced the majority class to scale it with the class of minority.

```
# splitting the dataset
X1 = data_encoded_1.copy()
X1.drop('income_50k', axis = 1, inplace = True)
y1 = data_encoded_1['income_50k']

rus = RandomUnderSampler(random_state=21)
X_Usampled, y_Usampled = rus.fit_resample(X1, y1)
```

Result:
With a Learning rate of 0.00001, lambda = 0.001, and a number of iterations of 100, we achieved an accuracy of 81%

```
import timeit
start = timeit.default_timer()
clf = HardMarginSVM()
clf.fit(np.array(X_train),np.array(y_train))
clf.predict(np.array(X_train),np.array(y_train))
stop = timeit.default_timer()
print('Time: ', round((stop - start),2), 'seconds')
```

```
The Accuracy value is: 81.0 %
The Precision value is: 0.86
The Recall value is: 0.75
Time:  22.89 seconds
```

C) SMOTE:

In SMOTE, we Introduce synthetic data points to increase the number of minority class and scale it up to the majority class.

```
# splitting the dataset
X1 = data_encoded_1.copy()
X1.drop('income_50k', axis = 1, inplace = True)
y1 = data_encoded_1['income_50k']

sm = SMOTE(random_state=21)
X_SMOTE, y_SMOTE = sm.fit_resample(X1, y1)
```

Result:

Learning rate as 0.00001, lambda = 0.001, and number of iterations as 100, we achieved an accuracy of 84%

```
The Accuracy value is: 84.0 %
The Precision value is: 0.87
The Recall value is: 0.79
Time:  300.44 seconds
```

D) Oversampled data

In oversampling, we are increasing the data point of the minority class to match the sample size of the majority class.

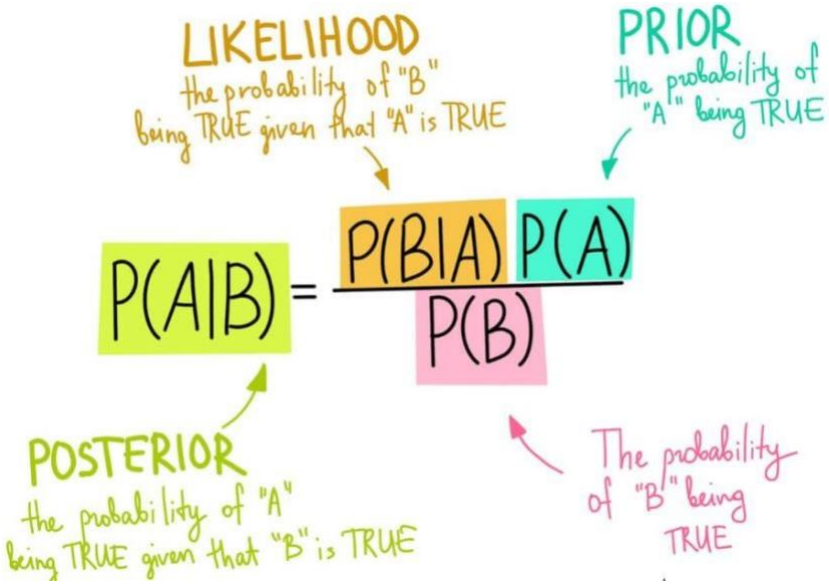
```
# splitting the dataset
X1 = data_encoded_1.copy()
X1.drop('income_50k', axis = 1, inplace = True)
y1 = data_encoded_1['income_50k']

ros = RandomOverSampler(random_state=21)
X_Osampled, y_Osampled = ros.fit_resample(X1, y1)
```

Learning rate as 0.00001, lambda = 0.001, and number of iterations as 100, we achieved an accuracy of 83%

```
The Accuracy value is: 83.0 %
The Precision value is: 0.86
The Recall value is: 0.79
Time:  303.87 seconds
```

3. NAÏVE BAYES

The Naïve Bayes method is widely used in machine learning and is considered a classification problem. Naïve Bayes and logistic regression work well for linear data. Still, one advantage of the naïve algorithm is speed; it works fast, which makes it comparable with the other ML methods.



For Naïve Bayes, we need to follow some steps such as calculating the likelihood and the probability of each class, and the primary assumption in this method is we assume all the variables are independent.

To normalize *the data*, we have created a function to calculate the mean and standard deviation and return the normalized data points.

```
# Normalizing the dataset
def normalising(X1):
  mean = np.mean(X1.values)
  std = np.std(X1.values)
  X_norm = (X1.values - mean) / std
  return X_norm
```

We define the *likelihood function* as the following: the essential parameter for calculating joint probability is standard deviation and mean for given to gaussian distribution function.

```
def calculate_likelihood(self,x,mean,sigma):
  lkhd = 1
  for n in range(len(mean)):
    exponent = exp(-((x[n]-mean[n])**2 / (2 * sigma[n]**2 )))
    c = (1 / (sqrt(2 * pi) * sigma[n])) * exponent
    lkhd = lkhd * c
  return lkhd
```

The next step is calculating the independent *conditional probability* using the maximum likelihood.

```
def probability(self, X, prior, mean, sigma):
  return prior * self.calculate_likelihood(X,mean,sigma)
```

The above steps are repeated for the *Oversampling*, Undersampling, and *SMOTE* data. We will go through each step to evaluate the result in the following.

A) Original Data:

While using the original sample dataset, the model's accuracy was around 68%, which proves that the model was unable to learn from Data properly. So to deal with this, we will use the sampled data.

```
Accuracy: 68.0 %
F1 score: 0.7964995550281816
Precision: 0.9852952740426201
Recall: 0.6684211462204599
Time:  13.12 seconds
```

B) Undersampled data:

For the Undersampling method, the result comes up with 76%.

```
Accuracy: 76.0 %
F1 score: 0.7258413090460017
Precision: 0.843862167982771
Recall: 0.6367822318526544
Time:  3.69 seconds
```

C) Oversampled data:

We got 76% accuracy for the oversampling, which is the same result as Undersampling.

```
Accuracy: 76.0 %
F1 score: 0.7258303106445926
Precision: 0.8396390060947023
Recall: 0.6391912630714872
Time:  23.41 seconds
```

D) Smote data:

The result for the smote shows us 85%, which is a higher accuracy than the oversampling and the Undersampling.

```
Accuracy: 85.0 %
F1 score: 0.8318305848855143
Precision: 0.9416675993284835
Recall: 0.7449400598516105
Time:  23.67 seconds
```

RESULTS

This project's goal is to compare the different ML classification models for the given data set. We defined our priority based on the accuracy of the models and the time. In the following, we will provide the summary table, which gives us information about each model's performance.

	Logistic Regression			SVM			Naïve Bayes		
	Over_ sampling	Under_ sampling	SMOTE	Over_ sampling	Under_ sampling	SMOTE	Over_ Sampling	Under_ Sampling	SMOTE
Accuracy	83.2	82.5	83.7	83	81.0	84.0	76.0	76.0	85
Time	824.2s	102.9s	848s	303.9s	22.89s	300.4s	23.41s	3.69s	23.67s

CONCLUSION

In this project, I went through the adult_census_income data set, and by exploring the Data, and figured out we were dealing with unbalanced data. To overcome this issue, decided to chose three approaches: oversampling, under sampling, and the smote method.
Since I defined the problem statement as classifications, I implemented machine learning methods such as Logistic Regression, SVM, and the naïve Bayes. Each sampling method was repeated for each ML model, and in the last step, we compared the time and accuracy of all the models under all types of the sampling method for better presentation of results.
Please refer to the table above for the best comparison of the models.

FUTURE STUDY

There is always a gap between academia and industry problems. In the future, I'm trying to implement those approaches to the real problem, and as we know, every problem is different and specific and has its priority; I intend to define our problem to another constraint such as cost, which plays a crucial factor these days in a real word problem.