**Machine Learning for Data Science**

**Assignment 1**

**Group Member**

**Mamoona Akbar**

**Anosha Khan**

# Experiment Description:

This study aims to construct and evaluate a Naïve Bayes Classifier for predicting income levels (>50K or <=50K) using a subset of the census income dataset provided by the Course Instructor. This dataset comprises four attributes: Sector, Education, Relationship, and Sex, each associated with corresponding income labels.

The primary steps of the experiment are:

## Data Preprocessing:

* Removal of samples with missing values.
* Transformation of the Education attribute to reduce the number of distinct values to five categories: Master, Bachelor, Doctorate, Primary, and High School.
* Eliminating samples with relationship values other than Wife, Husband, or Unmarried.

## Development of Naïve Bayes Classifier:

* Utilization of the assumption of conditional independence between features given the class label to construct the classifier.

## Training Model and Testing the Results:

* Application of the developed Naïve Bayes Classifier on the dataset.
* Evaluation of the model's accuracy by comparing the predicted and actual values for the last 100 test samples.

In the data preprocessing phase, we meticulously handled missing data and standardized the Education attribute to simplify the classification process. Subsequently, we crafted a Naïve Bayes Classifier, leveraging the feature independence assumption, to predict income levels based on the provided attributes. Finally, we evaluated the accuracy of our model by comparing its predictions against the ground truth for a subset of test samples.

# Methodology of Naïve Bayes Classifier:

## Understanding Naïve Bayes:

* + Naïve Bayes is a probabilistic classification algorithm based on Bayes' Theorem, which predicts the probability of a class label given the values of input features.
  + It assumes that the features are conditionally independent given the class label, simplifying the computation of probabilities.

## Data Preprocessing:

* + Cleanse the dataset by handling missing values and encoding categorical features.
  + Split the dataset into training and testing sets for model evaluation.

## Model Development:

* + Initialize the Naïve Bayes Classifier.
  + Calculate the prior probabilities of each class label by counting the frequency of each class in the training set.
  + Estimate the likelihood probabilities of each feature value given each class label by counting the frequency of feature-value-class triples in the training set.
  + Apply Laplace smoothing to handle zero probabilities.
  + Store the calculated probabilities in the classifier for later use.

## Model Training:

* + Train the Naïve Bayes Classifier using the training dataset.
  + Use the prior and likelihood probabilities to make predictions on the testing dataset.

## Model Evaluation:

* + Evaluate the performance of the classifier using various metrics such as accuracy.
  + Generate a confusion matrix to visualize the classification results and identify any misclassifications.
  + Analyse the model's strengths and weaknesses based on the evaluation metrics and adjust the parameters or features if necessary.

## Optimization and Fine-Tuning:

* + Experiment with different preprocessing techniques, such as feature scaling or feature engineering, to improve model performance.
  + Fine-tune the model hyperparameters, such as the smoothing parameter for Laplace smoothing, using techniques like cross-validation.
  + Consider ensemble methods or other advanced techniques to enhance the classifier's accuracy further.

## Deployment and Monitoring:

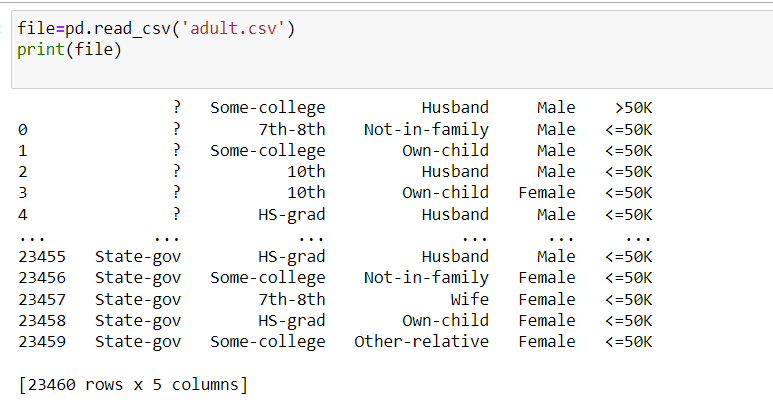
* + Deploy the trained Naïve Bayes Classifier in production environments to make real-time predictions.
  + Monitor the model's performance over time and retrain it periodically with new data to maintain its accuracy and relevance.

By following this methodology, one can effectively develop, train, and evaluate a Naïve Bayes Classifier for various classification tasks.

# Experimental setup

We used Python programming language to ease our process of data preprocessing, Model Development, Training, and Testing. We have performed the following task in data preprocessing.

## Loading Data Set

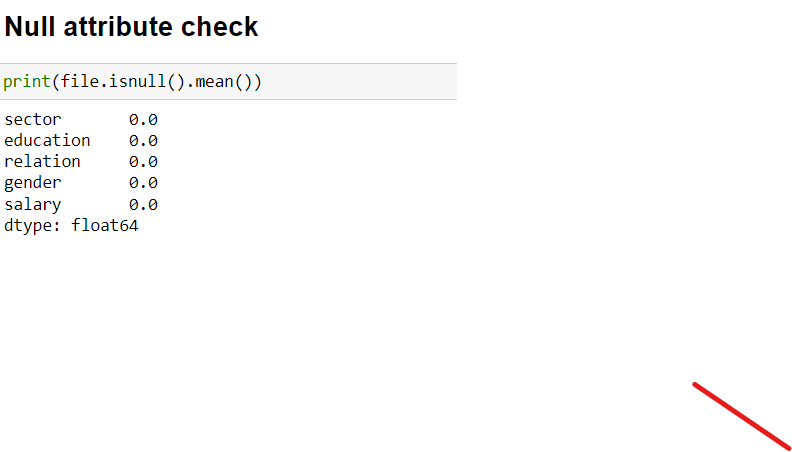
****

## Add Header in Data Set

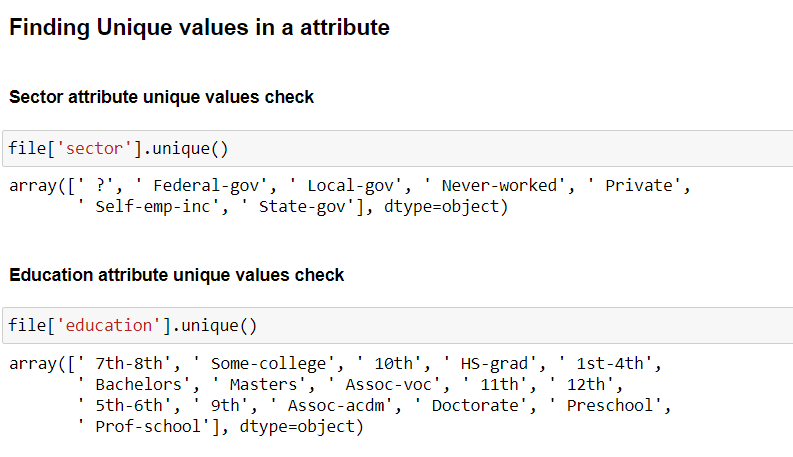
A screen shot of a computer

Description automatically generated

## Null values check



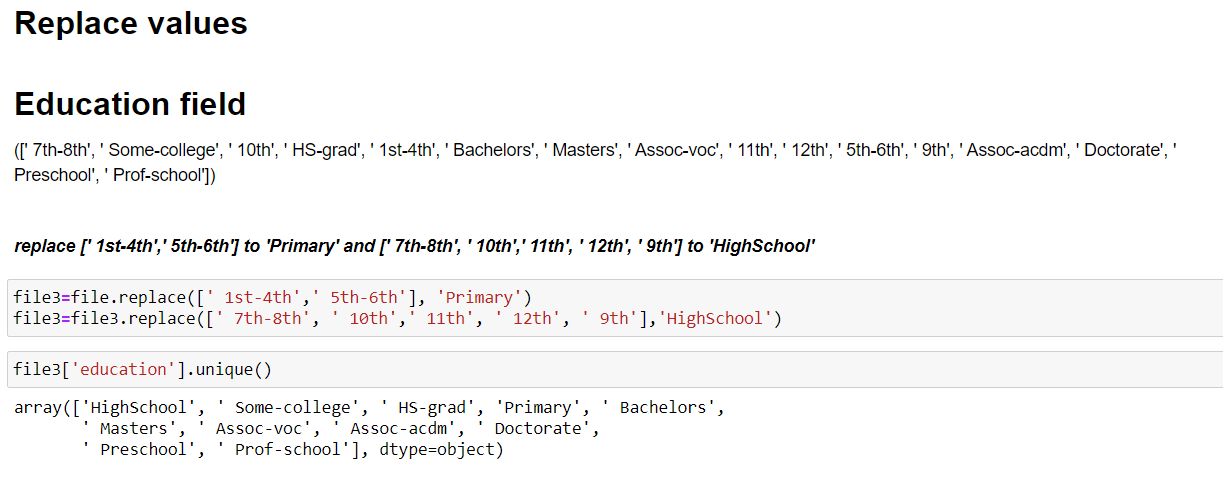
## Finding Unique Values in an Attribute



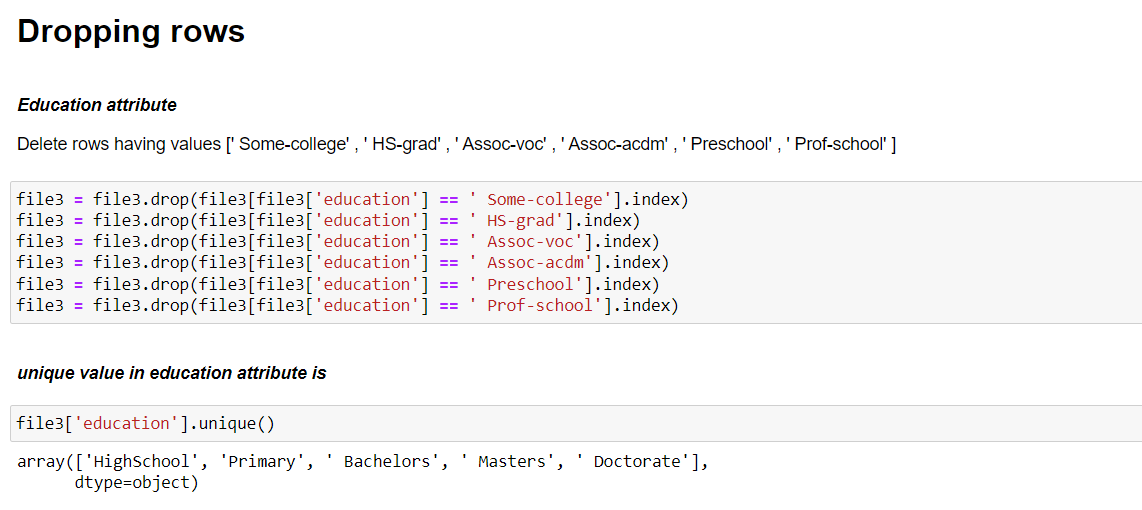
A screenshot of a computer

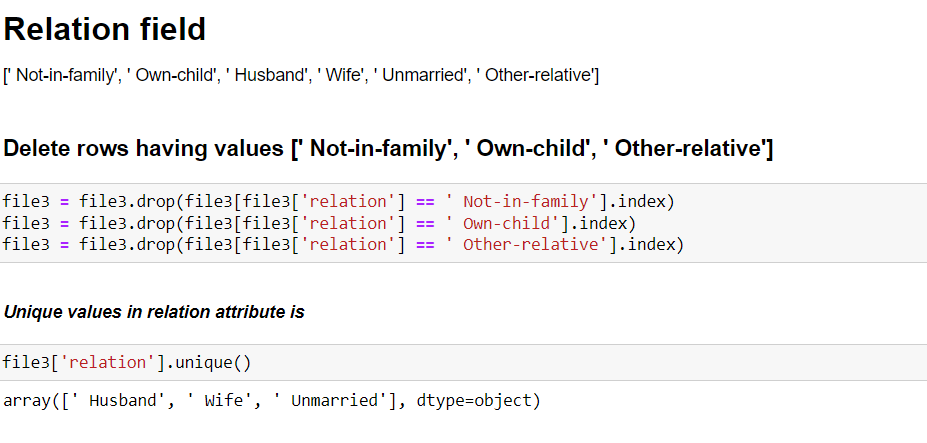
Description automatically generated

## Replace Values



## Dropping Rows





A screenshot of a computer

Description automatically generated

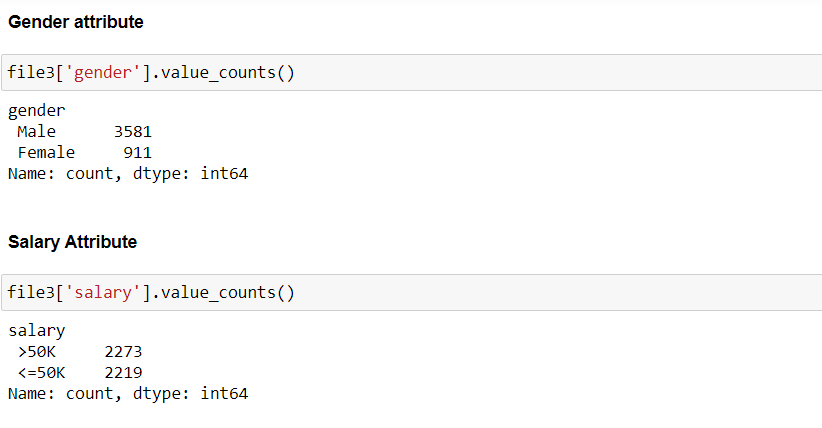
## After Preprocessing Unique Values in an Attribute

A screenshot of a computer

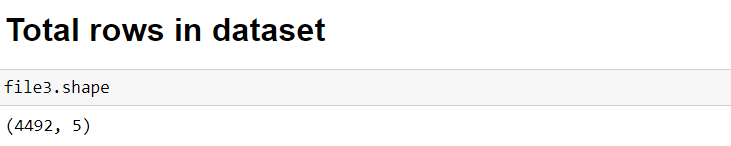
Description automatically generated

**A screenshot of a computer program

Description automatically generated**

****

## After Preprocessing Total Samples



# Naïve Model Training

## Splitting Data set

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

## Model Training and Testing

Calculate the prior probabilities of each class label by counting the frequency of each class in the training set.

Estimate the likelihood probabilities of each feature value given each class label by counting the frequency of feature-value-class triples in the training set.

Apply Laplace smoothing to handle zero probabilities.

Store the calculated probabilities in the classifier for later use

## Complete Code

import matplotlib.pyplot as plt

import numpy

from sklearn import metrics

class naivebayes:

def \_\_init\_\_(self,x\_train,y\_train,x\_test,y\_test):

self.x\_train=x\_train

self.y\_train=y\_train

self.x\_test=x\_test

self.y\_test=y\_test

##### Unique values in y\_train or y\_test

self.y=y\_train.unique()

##### prior\_probality

self.pri\_probality=[]

##### likelihood probality

self.probabilities={}

##### predicted probalities

self.predicted\_probalities=[]

##### predicted\_value

self.predicted\_value=[]

def prior\_probality(self):

length\_y=len(self.y)

length\_x\_train=len(self.x\_train)

for i in range(length\_y):

count=0

for j in range(length\_x\_train):

if self.y[i]==self.y\_train.iloc[j]:

count=count+1

self.pri\_probality.append([self.y[i],(count/length\_x\_train)])

# print(self.pri\_probality)

def probality\_store(self):

# Initialize an empty dictionary to store the probabilities

# self.probabilities = {}

# Iterate over each column in x\_train

for column in x\_train.columns:

# Initialize an empty dictionary for the current column

column\_dict = {}

# Get the unique values in the current column

unique\_values = x\_train[column].unique()

# Iterate over each unique value in the current column

for value in unique\_values:

# Initialize an empty dictionary for probabilities of each target value

target\_probabilities = {}

# Iterate over each unique target value in y\_train

for target\_value in y\_train.unique():

# Count occurrences of (value, target\_value) pairs in the dataset

count = ((x\_train[column] == value) & (y\_train == target\_value)).sum()

# Calculate the probability of (value, target\_value) pair

probability = count / len(y\_train[y\_train == target\_value])

# Store the probability in the dictionary

target\_probabilities[target\_value] = probability

# Store the dictionary of probabilities for the current value in the column

column\_dict[value] = target\_probabilities

# Store the dictionary of probabilities for the current column in the main dictionary

self.probabilities[column] = column\_dict

# print(self.probabilities)

# # Now you can access the probabilities for each category in each column

# print(probabilities['sector'][' Federal-gov'])

def predict\_data(self):

for i in range(len(self.x\_test)): ############### test data values

pre\_result=[]

for j in range(len(self.y)): ######## target specific value

prob=float(self.pri\_probality[j][1])

# print("prob",prob)

unique\_values = x\_train.columns

for k in range(len(unique\_values)): ##### column

if(float(self.probabilities[unique\_values[k]][self.x\_test.iloc[i,k]][self.y[j]]) == 0.0): ## if any probality zero laplace smoothing apply

print("zero probality:",float(self.probabilities[unique\_values[k]][self.x\_test.iloc[i,k]][self.y[j]]))

prob=prob\*((float(self.probabilities[unique\_values[k]][self.x\_test.iloc[i,k]][self.y[j]])+1)/1+len(self.probabilities[unique\_values[k]]))

else:

prob=prob\*float(self.probabilities[unique\_values[k]][self.x\_test.iloc[i,k]][self.y[j]])

pre\_result.append(prob)

self.predicted\_probalities.append(pre\_result)

def accuracy(self):

#### self.predicted\_probalities [-----,------] two probality

#### self.y\_test

##### self.y list unique y\_test label

count=0

for i in range(len(self.y\_test)):

maxi=0.0

k=0

for j in range(len(self.y)): #### highest probality find

# print("print",self.predicted\_probalities[i][j])

if( float(self.predicted\_probalities[i][j])> float(maxi)):

maxi=float(self.predicted\_probalities[i][j])

k=j #specific index highest value

self.predicted\_value.append(self.y[k])

for j in range(len(self.y)): #### highest probality find

if ( (self.y[j]== self.y\_test.iloc[i]) & (j==k)): ### actual == predicted

count=count+1

print("accuracy",count/len(self.y\_test))

def confusion\_matrixx(self):

actual = self.y\_test

predicted = self.predicted\_value

confusion\_matrix= metrics.confusion\_matrix(actual, predicted)

cm\_display = metrics.ConfusionMatrixDisplay(confusion\_matrix = confusion\_matrix, display\_labels = self.y)

cm\_display.plot()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

abc=naivebayes(x\_train,y\_train,x\_test,y\_test)

abc.prior\_probality()

abc.probality\_store()

abc.predict\_data()

abc.accuracy()

abc.confusion\_matrixx()

## Accuracy

## A chart of blue yellow and purple squares Description automatically generated

# Result and Discussion

In this study, we developed a Naive Bayes classifier to predict income class (>50K or <=50K) based on four attributes: Sector, education, relationship, and sex. Upon training the classifier on the provided dataset, we attained an accuracy of 79% on the test set. However, upon examining the confusion matrix of the results, it became evident that there were shortcomings in the predictions. Specifically, the classifier misclassified 3 samples belonging to the label >50K as <=50K, and 18 samples belonging to the label <=50K as >50K.

While achieving a 79% accuracy may seem satisfactory, it's clear that there is room for improvement. One potential limitation contributing to the achieved accuracy is the simplicity of both the dataset and the Naive Bayes classifier's assumption of feature independence. While Naive Bayes is well-suited for categorical features and is relatively simple to implement, it may struggle with datasets containing continuous or correlated features.

Furthermore, the large number of dropped samples during preprocessing may have also impacted the accuracy negatively. By including more samples, we could potentially enhance the classifier's performance.

To address these limitations and improve accuracy, future analyses could involve:

* Exploring additional features that may better capture the nuances of income prediction.
* Experimenting with different classification algorithms that may better handle the complexities of the dataset.
* Employing feature engineering techniques to extract more meaningful information from the existing attributes.

In conclusion, while Naive Bayes served as a suitable starting point for predicting salary class based on the selected attributes, further refinement, and exploration are necessary to achieve higher accuracy levels.

# Conclusion

The Naive Bayes classifier demonstrated a commendable accuracy of 79%, suggesting its effectiveness in predicting salary classes based on the selected attributes. However, this study reveals avenues for further refinement and exploration to enhance predictive performance.

Addressing the limitations inherent in the Naive Bayes approach and considering alternative algorithms or feature engineering strategies could potentially improve the model's predictive capabilities. Exploring these avenues could lead to a more robust and accurate classifier, better suited to handle the complexities of real-world datasets.

In summary, while the Naive Bayes classifier provides a promising foundation for salary prediction, continued research and development efforts are warranted to unlock its full potential and achieve even higher levels of accuracy.