Understanding Naïve Bayes Algorithm in Machine Learning

Naïve Bayes classification is a fundamental and widely used algorithm in machine learning, particularly in the domain of classification tasks.

The algorithm is based on applying Bayes' theorem with a key assumption of feature independence.

Bayes' Theorem:

Before delving into Naïve Bayes, let's understand Bayes' theorem, a fundamental principle in **probability theory** --> Multipiaction rule --> Independent event & Dependent event.

Bayes' theorem relates the conditional and marginal probabilities of events A and B, allowing us to calculate one given the other:

P(A/B) = P(B/A) * P(A) / P(B)

Where:

- **P(A/B)** is the probability of event A occurring given that event B has occurred.
- **P(B/A)** is the probability of event B occurring given that event A has occurred.
- **P(A)** is the prior probability of event A.
- **P(B)** is the prior probability of event B.

Application of Bayes' Theorem in Naïve Bayes Algorithm:

In the context of Naïve Bayes classification, we use Bayes' theorem to calculate the probability of a particular class given the features associated with a given instance. The algorithm assumes that the features are conditionally independent.

The formula for Naïve Bayes classification can be simplified using the assumption of feature independence:

P(y/X) = P(X/y) * P(y) / P(X)

Where:

- **P(y/X)** is the probability of class y given the features X (what we want to calculate).
- **P(X/y)** is the probability of observing features X given class y (likelihood).
- **P(y)** is the prior probability of class y.
- **P(X)** is the prior probability of features X (evidence).

Naïve Bayes Assumption of Feature Independence:

The "naïve" assumption in Naïve Bayes is that the features (attributes) used to describe the instances are independent of each other given the class label. This simplifies the calculation of the likelihood P(X/y) and makes the algorithm computationally efficient.

By assuming feature independence, we can represent the likelihood P(X/y) as the product of individual feature probabilities:

P(X/y) = P(y) * P(X1/y) * P(X2/y) * ......* P(Xn/y)

This simplification allows us to compute the nearner probability P(y/X) easily.

Why Use Naïve Bayes:

- Naïve Bayes is computationally efficient and easy to implement, making it suitable for large datasets.
- It's particularly effective for text-based data analysis, such as sentiment analysis, spam detection, document classification, etc.
- Naïve Bayes often performs well even with a small amount of training data.
- Despite its simplifying assumptions, Naïve Bayes can yield competitive classification performance compared to more complex algorithms.

Types of Naive Bayes algorithm

There are 3 types of Naïve Bayes algorithm.

- Gaussian Naïve Bayes
- Multinomial Naïve Bayes
- Bernoulli Naïve Bayes

Gaussian Naïve Bayes Algorithm:

Gaussian Naïve Bayes designed for datasets with continuous attribute values. It makes the assumption that the values associated with each class are distributed according to a Gaussian (Normal) distribution.

we segment the training data by each class. Then compute the mean (μi) and variance (σ) of the continuous attribute values. Then, given a new observation (xi), calculate the probability distribution of xi for each class using the Gaussian distribution equation: https://scikit-learn.org/stable/modules/naive_bayes.html

This equation gives the probability of xi belonging to a certain class based on its Gaussian distribution.

Multinomial Naïve Bayes Algorithm:

Multinomial Naïve Bayes is suitable for datasets where samples (feature vectors) represent the frequencies of events generated by a multinomial distribution. It is commonly used in text categorization, where features are typically word counts or term frequencies.

Bernoulli Naïve Bayes Algorithm:

Bernoulli Naïve Bayes is another variant suitable for datasets where features are binary or boolean variables, describing inputs as either present or absent. It's often used in document classification tasks.

These Naïve Bayes variants are widely used in various domains and provide efficient and effective solutions for classification tasks based on the underlying distribution of the data.

Applying Naïve Bayes Algorithm in Machine Learning

Import Python Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
import seaborn as sns
```

Load Dataset

Dataset: https://www.kaggle.com/datasets/qizarafzaal/adult-dataset/data

Context of the Dataset : the dataset contains of adult population information and the thier income is more or less then 50k per year.

Goal: Using Gaussian Naïve Bayes to Predict whether income exceeds $50K/yr based on census data.

```
df = pd.read_csv('/content/adult.csv',sep=',\s',header=None)
df.head()
```

```
<ipython-input-306-4c0a0d3d0859>:1: ParserWarning: Falling back to the 'python' engine because the 'c' engine does not support regex separators (separators > 1 char and different from '\s+' are interpreted as re
  df = pd.read_csv('/content/adult.csv',sep=',\s',header=None)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

Set the Column name

```
column_names = ['age', 'workclass', 'fnlwgt', 'education', 'education_num', 'marital_status', 'occupation', 'relationship',
                'race', 'sex', 'capital_gain', 'capital_loss', 'hours_per_week', 'native_country', 'income']

df.columns = column_names
```

```
df.head()
```

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

Explotarory Data Analysis

```
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
```

Types of Variable

```
catagorical_feature = df.select_dtypes(include='object').columns
numerical_feature = df.select_dtypes(exclude='object').columns
```

```
catagorical_feature
```

```
Index(['workclass', 'education', 'marital_status', 'occupation',
       'relationship', 'race', 'sex', 'native_country', 'income'],
      dtype='object')
```

```
numerical_feature
```

```
Index(['age', 'fnlwgt', 'education_num', 'capital_gain', 'capital_loss',
       'hours_per_week'],
      dtype='object')
```

```
df[catagorical_feature].isnull().sum()
```

```
workclass         0
education         0
marital_status    0
occupation        0
relationship      0
race              0
sex               0
native_country    0
income            0
dtype: int64
```

```
df[numerical_feature].isnull().sum()
```

```
age               0
fnlwgt            0
education_num     0
capital_gain      0
capital_loss      0
hours_per_week    0
dtype: int64
```

Exploring Catagorical Data

```
df['workclass'].unique()
```

```
array(['State-gov', 'Self-emp-not-inc', 'Private', 'Federal-gov',
       'Local-gov', '?', 'Self-emp-inc', 'Without-pay', 'Never-worked'],
      dtype=object)
```

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

```
Private             22696
Self-emp-not-inc     2541
Local-gov            2093
?                    1836
State-gov            1298
Self-emp-inc         1116
Federal-gov           960
Without-pay            14
Never-worked            7
Name: workclass, dtype: int64
```

```
df['education'].unique()
```

```
array(['Bachelors', 'HS-grad', '11th', 'Masters', '9th', 'Some-college',
       'Assoc-acdm', 'Assoc-voc', '7th-8th', 'Doctorate', 'Prof-school',
       '5th-6th', '10th', '1st-4th', 'Preschool', '12th'], dtype=object)
```

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

```
HS-grad         10501
Some-college     7291
Bachelors        5355
Masters          1723
Assoc-voc        1382
11th             1175
Assoc-acdm       1067
10th              933
7th-8th           646
Prof-school       576
9th               514
12th              433
Doctorate         413
5th-6th           333
1st-4th           168
Preschool          51
Name: education, dtype: int64
```

```
df['marital_status'].unique()
```

```
array(['Never-married', 'Married-civ-spouse', 'Divorced',
       'Married-spouse-absent', 'Separated', 'Married-AF-spouse',
       'Widowed'], dtype=object)
```

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

```
Married-civ-spouse       14976
Never-married            10683
Divorced                  4443
Separated                 1025
Widowed                    993
Married-spouse-absent      418
Married-AF-spouse           23
Name: marital_status, dtype: int64
```

```
df['occupation'].unique()
```

```
array(['Adm-clerical', 'Exec-managerial', 'Handlers-cleaners',
       'Prof-specialty', 'Other-service', 'Sales', 'Craft-repair',
       'Transport-moving', 'Farming-fishing', 'Machine-op-inspct',
       'Tech-support', '?', 'Protective-serv', 'Armed-Forces',
       'Priv-house-serv'], dtype=object)
```

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

```
Prof-specialty       4140
Craft-repair         4099
Exec-managerial      4066
Adm-clerical         3770
Sales                3650
Other-service        3295
Machine-op-inspct    2002
?                    1843
Transport-moving     1597
Handlers-cleaners    1370
Farming-fishing       994
Tech-support          928
Protective-serv       649
Priv-house-serv       149
Armed-Forces            9
Name: occupation, dtype: int64
```

```
df['relationship'].unique()
```

```
array(['Not-in-family', 'Husband', 'Wife', 'Own-child', 'Unmarried',
       'Other-relative'], dtype=object)
```

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

```
Husband           13193
Not-in-family      8305
Own-child          5068
Unmarried          3446
Wife               1568
Other-relative      981
Name: relationship, dtype: int64
```

```
df['race'].unique()
```

```
array(['White', 'Black', 'Asian-Pac-Islander', 'Amer-Indian-Eskimo',
       'Other'], dtype=object)
```

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

```
White                 27816
Black                  3124
```

```
Asian-Pac-Islander     1039
Amer-Indian-Eskimo      311
Other                   271
Name: race, dtype: int64
```

```
df['sex'].unique()
```

```
array(['Male', 'Female'], dtype=object)
```

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

```
Male      21790
Female    10771
Name: sex, dtype: int64
```

```
df['native_country'].unique()
```

```
array(['United-States', 'Cuba', 'Jamaica', 'India', '?', 'Mexico',
       'South', 'Puerto-Rico', 'Honduras', 'England', 'Canada', 'Germany',
       'Iran', 'Philippines', 'Italy', 'Poland', 'Columbia', 'Cambodia',
       'Thailand', 'Ecuador', 'Laos', 'Taiwan', 'Haiti', 'Portugal',
       'Dominican-Republic', 'El-Salvador', 'France', 'Guatemala',
       'China', 'Japan', 'Yugoslavia', 'Peru',
       'Outlying-US(Guam-USVI-etc)', 'Scotland', 'Trinadad&Tobago',
       'Greece', 'Nicaragua', 'Vietnam', 'Hong', 'Ireland', 'Hungary',
       'Holand-Netherlands'], dtype=object)
```

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

```
United-States                 29170
Mexico                          643
?                               583
Philippines                     198
Germany                         137
Canada                          121
Puerto-Rico                     114
El-Salvador                     106
India                           100
Cuba                             95
England                          90
Jamaica                          81
South                            80
China                            75
Italy                            73
Dominican-Republic               70
Vietnam                          67
Guatemala                        64
Japan                            62
Poland                           60
Columbia                         59
Taiwan                           51
Haiti                            44
Iran                             43
Portugal                         37
Nicaragua                        34
Peru                             31
France                           29
Greece                           29
Ecuador                          28
Ireland                          24
Hong                             20
Cambodia                         19
Trinadad&Tobago                  19
Laos                             18
```

```
Thailand                          18
Yugoslavia                        16
Outlying-US(Guam-USVI-etc)        14
Honduras                          13
Hungary                           13
Scotland                          12
Holand-Netherlands                 1
Name: native_country, dtype: int64
```

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

```
array(['<=50K', '>50K'], dtype=object)
```

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

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

From the above exploration into Catagorical data, we can see that variables **workclass, occupation and native_country** contain **(?)** which is missing values.

To detect the missing values generally we see it contains NaN after python code df.isnull().sum(), but we did not see because ? does not cosider as missing in python.

So, we replace ? into NaN.

Repalcing '?' into NaN

```
df['workclass'].replace('?', np.NaN, inplace=True)
```

```
df['occupation'].replace('?', np.NaN, inplace=True)
```

```
df['native_country'].replace('?', np.NaN, inplace=True)
```

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

```
Prof-specialty       4140
Craft-repair         4099
Exec-managerial      4066
Adm-clerical         3770
Sales                3650
Other-service        3295
Machine-op-inspct    2002
Transport-moving     1597
Handlers-cleaners    1370
Farming-fishing       994
Tech-support          928
Protective-serv       649
Priv-house-serv       149
Armed-Forces            9
Name: occupation, dtype: int64
```

```
df[catagorical_feature].isnull().sum()
```

```
workclass         1836
education            0
marital_status       0
```

```
occupation        1843
relationship         0
race                 0
sex                  0
native_country     583
income               0
dtype: int64
```

Cardinality

Low cardinality: If there's only one category in a column, it won't provide any unique information to our model. Low cardinality means the observation in the columns has constant value which means same value in all the rows of the columns. Ex. type of building has only apartment. So don't include in the model. Drop

High cardinality: as low cardinality don't give you same information to the model, High cardinality doesn't give any information to the model. Can drop also.

As low cardinality gives low or no information and high cardinality is overload with information which both doesn't help the model since model looks for trend.

```
for col in catagorical_feature:
    print(col, len(df[col].unique()))
```

```
workclass 9
education 16
marital_status 7
occupation 15
relationship 6
race 5
sex 2
native_country 42
income 2
```

Numerical feature

```
df[numerical_feature].describe().T
```

	count	mean	std	min	25%	50%	75%	max
age	32561.0	38.581647	13.640433	17.0	28.0	37.0	48.0	90.0
fnlwgt	32561.0	189778.366512	105549.977697	12285.0	117827.0	178356.0	237051.0	1484705.0
education_num	32561.0	10.080679	2.572720	1.0	9.0	10.0	12.0	16.0
capital_gain	32561.0	1077.648844	7385.292085	0.0	0.0	0.0	0.0	99999.0
capital_loss	32561.0	87.303830	402.960219	0.0	0.0	0.0	0.0	4356.0
hours_per_week	32561.0	40.437456	12.347429	1.0	40.0	40.0	45.0	99.0

Split

A key part in any model-building project is separating your target (y) (the thing you want to predict) from your features (X) (the information your model will use to make its predictions).

```
X = df.drop(['income'], axis=1)
y = df['income']
```

Train Test split

```
# split X and y into training and testing sets

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
```

```
print('X_train:',X_train.shape)
print('y_train:',y_train.shape)
print('X_test:',X_test.shape)
print('y_test:',y_test.shape)
```

```
X_train: (22792, 14)
y_train: (22792,)
X_test: (9769, 14)
y_test: (9769,)
```

Feature Engineering

Feature Engineering is the process of transforming raw data into useful features that help us to understand our model better and increase its predictive power. I will carry out feature engineering on different types of variables.

- Imputaion
- Encoding
- Scaling

```
cat_col = X_train.select_dtypes(include='object').columns
num_col = X_train.select_dtypes(exclude='object').columns
```

Handling the Missing values

```
X_train[num_col].isnull().sum()
```

```
age               0
fnlwgt            0
education_num     0
capital_gain      0
capital_loss      0
hours_per_week    0
dtype: int64
```

```
X_train[cat_col].isnull().sum()
```

```
workclass         1276
education            0
marital_status       0
occupation        1278
relationship         0
race                 0
sex                  0
native_country     414
dtype: int64
```

Imputatiuon

There are two methods can be used to impute missing values.

- mean or median or mode imputation
- random sample imputation

When there are outliers in the dataset, we should use median imputation.

impute missing categorical variables with **most frequent value**

SimpleImputer is a scikit-learn class which is helpful in handling the missing data

implemented by the use of the

- SimpleImputer():
    - missing_values : The missing_values placeholder which has to be imputed. By default is NaN
    - strategy: The data which will replace the NaN values from the dataset. The strategy argument can take the values – 'mean'(default), 'median', 'most_frequent' and 'constant'.

```
from sklearn.impute import SimpleImputer

# Define columns with missing values
columns_with_missing_values = ['workclass', 'occupation', 'native_country']

# Create the SimpleImputer object
imputer = SimpleImputer(strategy='most_frequent')

# Fit and transform X_train
X_train[columns_with_missing_values] = imputer.fit_transform(X_train[columns_with_missing_values])

# Transform X_test
X_test[columns_with_missing_values] = imputer.transform(X_test[columns_with_missing_values])
```

```
X_test[cat_col].isnull().sum()
```

```
    workclass         0
    education         0
    marital_status    0
    occupation        0
    relationship      0
    race              0
    sex               0
    native_country    0
    dtype: int64
```

```
X_train[cat_col].isnull().sum()
```

```
    workclass         0
    education         0
    marital_status    0
    occupation        0
    relationship      0
    race              0
    sex               0
    native_country    0
    dtype: int64
```

```
X_train.head()
```

	age	workclass	fnlwgt	education	education_num	marital_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_week	native_country
32098	45	Private	170871	HS-grad	9	Married-civ-spouse	Craft-repair	Husband	White	Male	7298	0	60	United-States
25206	47	State-gov	108890	HS-grad	9	Divorced	Adm-clerical	Unmarried	White	Female	1831	0	38	United-States
23491	48	Private	187505	Some-college	10	Married-civ-spouse	Sales	Husband	White	Male	0	0	50	United-States
12367	29	Private	145592	HS-grad	9	Never-married	Craft-repair	Not-in-family	White	Male	0	0	40	Guatemala
7054	23	Private	203003	7th-8th	4	Never-married	Craft-repair	Not-in-family	White	Male	0	0	25	Germany

Encoding

OHE is the standard approach to encode categorical data.

One hot encoding (OHE) creates a binary variable for each one of the different categories present in a variable. These binary variables take 1 if the observation shows a certain category or 0 otherwise. OHE is suitable for linear models.

One hot encoding (OHE) creates by replacing the categorical variable by different boolean variables, which take value 0 or 1, to indicate whether or not a certain category / label of the variable was present for that observation. Each one of the boolean variables are also known as dummy variables or binary variables.

For example, from the categorical variable "Gender", with labels 'female' and 'male', we can generate the boolean variable "female", which takes 1 if the person is female or 0 otherwise. We can also generate the variable male, which takes 1 if the person is "male" and 0 otherwise.

import category_encoders as ce

encoder = ce.OneHotEncoder(cols=['workclass', 'education', 'marital_status', 'occupation', 'relationship', 'race', 'sex', 'native_country'])

X_train = encoder.fit_transform(X_train)

X_test = encoder.transform(X_test)

```
from sklearn.preprocessing import OneHotEncoder
# Assuming 'cat_col' contains the names of categorical columns
encoder = OneHotEncoder()

# Fit and transform X_train
X_train_encoded = encoder.fit_transform(X_train[cat_col])
X_train_encoded_df = pd.DataFrame(X_train_encoded.toarray(), columns=encoder.get_feature_names_out(cat_col))

# Transform X_test
X_test_encoded = encoder.transform(X_test[cat_col])
X_test_encoded_df = pd.DataFrame(X_test_encoded.toarray(), columns=encoder.get_feature_names_out(cat_col))



# Drop the original categorical columns
X_train = X_train.drop(cat_col, axis=1)
X_test = X_test.drop(cat_col, axis=1)



# Concatenate the encoded columns
X_train = pd.concat([X_train.reset_index(drop=True), X_train_encoded_df], axis=1)
```

```
X_test = pd.concat([X_test.reset_index(drop=True), X_test_encoded_df], axis=1)
```

```
X_train.shape
```

    (22792, 105)

```
X_test.shape
```

    (9769, 105)

Feature Scaling

- StandardScaler
- MinMaxScaler
- RobustScaler

RobustScaler is a method for scaling features in a dataset using statistics that are robust to outliers.

When you scale features using RobustScaler, it removes the median and scales the data based on the interquartile range (IQR). This scaling is more robust to outliers compared to standard scaling methods like mean and variance scaling.

```
cols = X_train.columns
```

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
X_train = pd.DataFrame(X_train, columns=[cols])
X_test = pd.DataFrame(X_test, columns=[cols])
```

```
X_train.head()
```

	age	fnlwgt	education_num	capital_gain	capital_loss	hours_per_week	workclass_Federal-gov	workclass_Local-gov	workclass_Never-worked	workclass_Private	...	native_country_Portugal	native_country_Puerto-Rico	nativ
0	0.40	-0.058906	-0.333333	7298.0	0.0	4.0	0.0	0.0	0.0	0.0	...	0.0	0.0	
1	0.50	-0.578076	-0.333333	1831.0	0.0	-0.4	0.0	0.0	0.0	-1.0	...	0.0	0.0	
2	0.55	0.080425	0.000000	0.0	0.0	2.0	0.0	0.0	0.0	0.0	...	0.0	0.0	
3	-0.40	-0.270650	-0.333333	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	
4	-0.70	0.210240	-2.000000	0.0	0.0	-3.0	0.0	0.0	0.0	0.0	...	0.0	0.0	

5 rows × 105 columns

Building Model

Baseline: The first step in building a model is baselining. To do this, ask yourself how you will know if the model you build is performing well?

Model Training

The steps to building and using a model are:

Define: What type of model will it be? A decision tree? Some other type of model? Some other parameters of the model type are specified too.

Fit: Capture patterns from provided data. This is the heart of modeling.

Predict: Just what it sounds like

Evaluate: Determine how accurate the model's predictions are.

```
# train a Gaussian Naive Bayes classifier on the training set
from sklearn.naive_bayes import GaussianNB

# instantiate the model
gnb = GaussianNB()

# fit the model
gnb.fit(X_train, y_train)
```

```
▾ GaussianNB
GaussianNB()
```

Predict the Model

```
y_pred = gnb.predict(X_test)

y_pred
```

```
array(['<=50K', '<=50K', '>50K', ..., '>50K', '<=50K', '<=50K'],
      dtype='<U5')
```

Predicting test set result

At this point, the model is now trained and ready to predict the output of new observations. Remember, we split our dataset into train and test sets. We will provide test sets to the model and check its performance.

```
#y_test and y_pred are your actual and predicted labels
prediction_df = pd.DataFrame({
    'Actual Value': y_test,
    'Predicted Value': y_pred,
    'Prediction Correct': y_test == y_pred  # True if prediction is correct, False otherwise
})

# Display the prediction_df DataFrame
print(prediction_df)
```

```
           Actual Value Predicted Value  Prediction Correct
22278        <=50K           <=50K                True
8950         <=50K           <=50K                True
7838         <=50K            >50K               False
16505        <=50K            >50K               False
19140         >50K            >50K                True
...            ...             ...                 ...
21949         >50K            >50K                True
26405         >50K            >50K                True
23236         >50K            >50K                True
26823        <=50K           <=50K                True
20721        <=50K           <=50K                True
```

```
[9769 rows x 3 columns]
```

Evaluate the Model

accuracy score

```
from sklearn.metrics import accuracy_score
print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
```

```
Model accuracy score: 0.8083
```

Null or Baseline accuracy

Comapring model accuracy with a null or baseline accuracy is a good practice to evaluate the model's performance. The null accuracy is the accuracy achieved by a model that always predicts the most frequent class in the dataset. It provides a baseline for comparison, helping to gauge whether the model's performance is meaningful and better than a simple baseline prediction strategy.

```
y_test.value_counts()
```

```
<=50K    7407
>50K     2362
Name: income, dtype: int64
```

```
baseline_accuracy = (7407/(7407+2362))

print('Baseline accuracy score: {0:0.4f}'. format(baseline_accuracy))
```

```
Baseline accuracy score: 0.7582
```

Check for overfitting and underfitting:

- Overfitting usually manifests as a significant gap between training and test accuracies.
- Underfitting is marked by low accuracies on both sets due to insufficient model complexity.
- Generalized Model is demonstrates consistent performance on both training and test sets, suggesting it is well-generalized and not overfit.

```
print('Training set score: {:.4f}'.format(gnb.score(X_train, y_train)))

print('Test set score: {:.4f}'.format(gnb.score(X_test, y_test)))
```

```
Training set score: 0.8067
Test set score: 0.8083
```

training set= 0.8067 and test set=0.8083 , it means that the model is performing consistently well on both the training and test data. The scores are close to each other, indicating that the model is likely well-generalized and not overfitting.

Based on the model classification accuracy and baseline accuracy, and genealized model, we can say model is performing very good. Our model is able to predict the class labels.

But, it does not give the underlying distribution of values and it does not tell anything about the type of errors our classifer is making. For that we use Confusion Matrix

Confusion Matrix A confusion matrix is a tool for summarizing the performance of a classification algorithm. A confusion matrix will give us a clear picture of classification model performance and the types of errors produced by the model. It gives us a summary of correct and incorrect predictions broken down by each category. The summary is represented in a tabular form.

Four types of outcomes are possible while evaluating a classification model performance. These four outcomes are described below:-

1. True Positives (TP) – True Positives occur when we predict an observation belongs to a certain class and the observation actually belongs to that class.
2. True Negatives (TN) – True Negatives occur when we predict an observation does not belong to a certain class and the observation actually does not belong to that class.
3. False Positives (FP) – False Positives occur when we predict an observation belongs to a certain class but the observation actually does not belong to that class. This type of error is called Type I error.
4. False Negatives (FN) – False Negatives occur when we predict an observation does not belong to a certain class but the observation actually belongs to that class. This is a very serious error and it is called Type II error.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

print('Confusion matrix\n\n', cm)

print('\nTrue Positives(TP) = ', cm[0,0])
print('\nTrue Negatives(TN) = ', cm[1,1])
print('\nFalse Positives(FP) = ', cm[0,1])
print('\nFalse Negatives(FN) = ', cm[1,0])
```

```
Confusion matrix

 [[5999 1408]
 [ 465 1897]]

True Positives(TP) =  5999

True Negatives(TN) =  1897

False Positives(FP) =  1408

False Negatives(FN) =  465
```

The confusion matrix shows 5999 + 1897 = 7896 correct predictions

and 1408 + 465 = 1873 incorrect predictions.

In this case, we have

True Positives (Actual Positive:1 and Predict Positive:1) - 5999

True Negatives (Actual Negative:0 and Predict Negative:0) - 1897

False Positives (Actual Negative:0 but Predict Positive:1) - 1408 (Type I error)

False Negatives (Actual Positive:1 but Predict Negative:0) - 465 (Type II error)

Classification metrices

Classification Report

Classification report is another way to evaluate the classification model performance. It displays the precision, recall, f1 and support scores for the model.

```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

       <=50K       0.93      0.81      0.86      7407
        >50K       0.57      0.80      0.67      2362

    accuracy                           0.81      9769
   macro avg       0.75      0.81      0.77      9769
weighted avg       0.84      0.81      0.82      9769
```

Precision

Precision can be defined as the percentage of correctly predicted positive outcomes out of all the predicted positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true and false positives (TP + FP).

precision = TP / float(TP + FP)

Recall

Recall can be defined as the percentage of correctly predicted positive outcomes out of all the actual positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true positives and false negatives (TP + FN). Recall is also called Sensitivity.

recall = TP / float(TP + FN)

f1-score

f1-score is the weighted harmonic mean of precision and recall. The best possible f1-score would be 1.0 and the worst would be 0.0.

f1-score is the harmonic mean of precision and recall. So, f1-score is always lower than accuracy measures as they embed precision and recall into their computation. The weighted average of f1-score should be used to compare classifier models, not global accuracy.

Support

Support is the actual number of occurrences of the class in our dataset.

k-Fold Cross Validation

```
# Applying 10-Fold Cross Validation

from sklearn.model_selection import cross_val_score

scores = cross_val_score(gnb, X_train, y_train, cv = 10, scoring='accuracy')

print('Cross-validation scores:{}'.format(scores))
```

```
Cross-validation scores:[0.81359649 0.80438596 0.81175954 0.8056165  0.79596314 0.79684072
 0.81044318 0.81175954 0.80210619 0.81044318]
```

We can summarize the cross-validation accuracy by calculating its mean.

```
print('Average cross-validation score: {:.4f}'.format(scores.mean()))
```

```
Average cross-validation score: 0.8063
```

original model accuracy is 0.8083, but the mean cross-validation accuracy is 0.8063.