BITS F464 - Machine Learning
Assignment – 2

Group members:

Vani Jain (2021A7PS2062H)

Yeshika Bharatiya (2021A7PS2779H)

Anushka Agnihotri (2021A7PS3114H)

Part A - Naive Bayes Classifier to predict income

Task 1: Data Preprocessing

Importing Libraries

```
import pandas as pd
import numpy as np
from scipy.stats import norm
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
import seaborn as sns
```

Loading the dataset into a pandas DataFrame

```
data=pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.data",names=["age","workclass","fnlwgt","education",
"education-num","marital-
status","occupation","relationship","race","sex","capital-gain","capital-
loss","hours-per-week","native-country","salary"])
```

Checking for missing values and replacing them with the mode of that column

```
def replaceNull(data):
  data1=data.copy()
  x=data1.filter(["workclass","occupation","native-country"])
  for feature in x:
    val=data1[feature].mode()[0]
    #print((val))
    data1[feature]=data1[feature].replace(' ?',val)
  return data1
```

Splitting the dataset into training and testing sets (80% for training, 20% for testing

```
def splitData(data):
  data1 = data.sample(frac=1,axis=0).reset_index(drop=True)
  x=int(0.80*len(data1))
  data_train=data1.iloc[:x,:]
  data_test=data1.iloc[x:,:]
  return data_train,data_test
```

Task 2: Naive Bayes Classifier Implementation

Naive Bayes Classifier is based on Bayes Theorem with an assumption of independence among the predictors. The Naive Bayes classifier assumes that the presence of a feature in a class is not related to any other feature.

Bayes Theorem

Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the following equation:

$$P(A/B) = \frac{P(B/A)P(A)}{P(B)}$$

where A and B are events and $P(B) \neq 0$.

- Basically, we are trying to find probability of event A, given the event B is true. Event B is also termed as evidence.
- **Prior probability** = *P(A)* is the probability before getting the evidence
- **Posterior probability** = *P(A|B)* is the probability after getting evidence
- Now, with regards to our dataset, we can apply Bayes' theorem in following way:
- $P(y/X) = \frac{P(X/y)P(y)}{P(X)}$
- where, y is class variable and X is a dependent feature vector (of size *n*) where:
- $X = (x1, x2, x3, \ldots\ldots, xn)$

Naive assumption

- Now, its time to put a naive assumption to the Bayes' theorem, which is, **independence** among the features. So now, we split **evidence** into the independent parts.
- Now, if any two events A and B are independent, then,
- $P(A,B) = P(A)P(B)$

- Hence, we reach to the result:
- $P(y/x1,...,xn) \; = \frac{P(x1/y)P(x2/y)....P(xn/y)P(y)}{P(x1)P(x2)...P(xn)}$
- which can be expressed as:
- $P(y/x1,...,xn) \; = \frac{P(y)\prod_{i=1}^{n} P(x_i/y)}{P(x1)P(x2)...P(xn)}$
- Now, as the denominator remains constant for a given input, we can remove that term:
- $P(y/x1,...,xn) \propto P(y)\prod_{i=1}^{n} P(x_i/y)$

Calculating the prior probability of each class

```
def calculate_prior_prob(data_train):

  p_lessthan50,p_morethan50=data_train["salary"].value_counts('<=50K')

  return p_lessthan50,p_morethan50
```

Calculating the conditional probability of each feature given to each class

# Dividing data into continuous and categorical

```
def divide_data(data):

  data_train=data.copy()

  data_cont=data_train.filter(["age","fnlwgt","education-num","capital-gain","capital-loss","hours-per-week","salary"],axis=1)

  data_categ=data_train.filter(["workclass","education","marital-status","occupation","relationship","race","sex","native-country","salary"],axis=1)

  return data_cont,data_categ
```

Likelihood probabilities for categorical data:

```
def calculate_likelihood_categ(data_categ):
  likelihood_probs={"workclass":{},"education":{},"marital-
status":{},"occupation":{},"relationship":{},"race":{},"sex":{},"native-
country":{},"salary":{}}
  x=data_categ.drop(["salary"],axis=1)
  y=data_categ["salary"]
  for feature in x:
    for outcome in np.unique(y):
      total_outcome=sum(y==outcome)

feature_likelihood=x[feature][y[y==outcome].index.values.tolist()].value_c
ounts().to_dict()
      for val,count in feature_likelihood.items():
        likelihood_probs[feature][val+"_"+outcome]=count/total_outcome


  return likelihood_probs
```

Likelihood probabilities for categorical data:

```
def calculate_likelihood_cont(data_cont):
  likelihood_probs={"age":{},"fnlwgt":{},"education-num":{},"capital-
gain":{},"capital-loss":{},"hours-per-week":{},"salary":{}}
  x=data_cont.drop(["salary"],axis=1)
  y=data_cont["salary"]
  for feature in x:
    for outcome in np.unique(y):
      feature_mean=x[feature][y[y==outcome].index.values.tolist()].mean()
      feature_std=x[feature][y[y==outcome].index.values.tolist()].std()
      likelihood_probs[feature]['mean_'+outcome]=feature_mean
      likelihood_probs[feature]['std_'+outcome]=feature_std
  return likelihood_probs
```

Predicting the class of a given instance using the Naive Bayes algorithm:

```
def
predict_class(arr,prior_lessthan50,prior_morethan50,likelihood_categ,likel
ihood_cont):
  numerator_lessthan=prior_lessthan50
  numerator_morethan=prior_morethan50
  for x in [1,3,5,6,7,8,9,13]:
    print(data.columns[x])
    print(arr[x])
    if((arr[x]+"_ <=50K") in likelihood_categ[data.columns[x]].keys()):
      numerator_lessthan*=likelihood_categ[data.columns[x]][arr[x]+"_
<=50K"]
    else:
      numerator_lessthan=0
      break
    if((arr[x]+"_ >50K") in likelihood_categ[data.columns[x]].keys()):
      numerator_morethan*=likelihood_categ[data.columns[x]][arr[x]+"_
>50K"]
    else:
      numerator_morethan=0
      break

  for x in [0,2,4,10,11,12]:
    mean_lessthan=likelihood_cont[data.columns[x]]['mean_ <=50K']
    std_lessthan=likelihood_cont[data.columns[x]]['std_ <=50K']
    mean_morethan=likelihood_cont[data.columns[x]]['mean_ >50K']

    std_morethan=likelihood_cont[data.columns[x]]['std_ >50K']

numerator_lessthan*=norm.pdf(arr[x],loc=mean_lessthan,scale=std_lessthan)

numerator_morethan*=norm.pdf(arr[x],loc=mean_morethan,scale=std_morethan)

  if(numerator_lessthan>=numerator_morethan):

    sal=" <=50K"

  else:

    sal=" >50K"

  return sal
```

Calculating all quantities before prediction:

```
prior_lessthan50,prior_morethan50=calculate_prior_prob(data_train)
likelihood_categ=calculate_likelihood_categ(data_categ)
likelihood_cont=calculate_likelihood_cont(data_cont)
```

Task 3: Evaluation and Improvement

Calculating Performance Metrics:

```
def NBTest(data_test):
  TN,TP,FN,FP=0,0,0,0
  for i,r in data_test.iterrows():
    print(i)

sal=predict_class(r.values,prior_lessthan50,prior_morethan50,likelihood_ca
teg,likelihood_cont)
    if(sal==r.values[-1]):
      if(sal==' >50K'):
        TP+=1
      else:
        TN+=1
    else:
      if(sal==' >50K'):
        FP+=1
      else:
        FN+=1
  print( TP,TN,FP,FN)
  accuracy=(TP+TN)/(TP+TN+FP+FN)
  precision=TP/(TP+FP)
  recall=TP/(TP+FN)
  f1_score=(2*precision*recall)/(precision+recall)
  return accuracy,precision,recall,f1_score
```

```
accuracy,precision,recall,f1_score=(0.8343313373253493, 0.718132854578097,
0.5111821086261981, 0.597237775289287)
```

Smoothing Techniques

Laplace Smoothing:

```
#no. of features
k=data_train.shape[1]-1

N_less=p_lessthan50*data_train.shape[0]
N_more=p_morethan50*data_train.shape[0]
```

```
def calculate_likelihood_categ_laplace(data_categ,alpha):
  likelihood_probs={"workclass":{},"education":{},"marital-
status":{},"occupation":{},"relationship":{},"race":{},"sex":{},"native-
country":{},"salary":{}}
  x=data_categ.drop(["salary"],axis=1)
  y=data_categ["salary"]
  for feature in x:
    for outcome in np.unique(y):
      total_outcome=sum(y==outcome)
```

```
feature_likelihood=x[feature][y[y==outcome].index.values.tolist()].value_c
ounts().to_dict()
      for val,count in feature_likelihood.items():

likelihood_probs[feature][val+"_"+outcome]=(count+alpha)/(total_outcome+k*
alpha)

  return likelihood_probs
```

```
def calculate_likelihood_cont(data_cont):
  likelihood_probs={"age":{},"fnlwgt":{},"education-num":{},"capital-
gain":{},"capital-loss":{},"hours-per-week":{},"salary":{}}
  x=data_cont.drop(["salary"],axis=1)
  y=data_cont["salary"]
  for feature in x:
    for outcome in np.unique(y):
      feature_mean=x[feature][y[y==outcome].index.values.tolist()].mean()
      feature_std=x[feature][y[y==outcome].index.values.tolist()].std()
      likelihood_probs[feature]['mean_'+outcome]=feature_mean
      likelihood_probs[feature]['std_'+outcome]=feature_std
  return likelihood_probs
```

```
#k is number of features,N_less is no of less than tuples
def
predict_class_laplace(arr,prior_lessthan50,prior_morethan50,likelihood_cat
eg_laplace,likelihood_cont,alpha):
  numerator_lessthan=prior_lessthan50
  numerator_morethan=prior_morethan50
  for x in [1,3,5,6,7,8,9,13]:
    print(data.columns[x])
    print(arr[x])
    if((arr[x]+"_ <=50K") in
likelihood_categ_laplace[data.columns[x]].keys()):
```

```
numerator_lessthan*=likelihood_categ_laplace[data.columns[x]][arr[x]+"_
<=50K"]
    else:
      numerator_lessthan*=(alpha/(N_less+alpha*k))

    if((arr[x]+"_ >50K") in
likelihood_categ_laplace[data.columns[x]].keys()):

numerator_morethan*=likelihood_categ_laplace[data.columns[x]][arr[x]+"_
>50K"]
    else:
      numerator_morethan*=(alpha/(N_more+alpha*k))


  for x in [0,2,4,10,11,12]:
    mean_lessthan=likelihood_cont[data.columns[x]]['mean_ <=50K']
    std_lessthan=likelihood_cont[data.columns[x]]['std_ <=50K']
    mean_morethan=likelihood_cont[data.columns[x]]['mean_ >50K']
    std_morethan=likelihood_cont[data.columns[x]]['std_ >50K']

numerator_lessthan*=norm.pdf(arr[x],loc=mean_lessthan,scale=std_lessthan)

numerator_morethan*=norm.pdf(arr[x],loc=mean_morethan,scale=std_morethan)

  if(numerator_lessthan>=numerator_morethan):
    sal=" <=50K"
  else:
    sal=" >50K"

  return sal
```

```
def NBTest_laplace(data_test):
  TN,TP,FN,FP=0,0,0,0
  for i,r in data_test.iterrows():
    print(i)
```

```
sal=predict_class_laplace(r.values,prior_lessthan50,prior_morethan50,likel
ihood_categ,likelihood_cont,1)
    if(sal==r.values[-1]):
      if(sal==' >50K'):
        TP+=1
      else:
        TN+=1
    else:
      if(sal==' >50K'):
        FP+=1
      else:
        FN+=1
  print( TP,TN,FP,FN)
  accuracy=(TP+TN)/(TP+TN+FP+FN)
  precision=TP/(TP+FP)
  recall=TP/(TP+FN)
  f1_score=(2*precision*recall)/(precision+recall)
  return accuracy,precision,recall,f1_score
```

```
accuracy,precision,recall,f1_score=(0.8163672654690619,
 0.5927601809954751,
 0.7533546325878594,
 0.6634777715250422)
```

Comparison with other models

Logistic regression

```
X_LR=data2.drop(["salary"],axis=1)
y_LR=data2["salary"]
y_LR=y_LR.map({' >50K':1, ' <=50K': 0})
X_LR.sex = X_LR.sex.map({' Male': 0, ' Female': 1})
for x in [1,3,5,6,7,8,9,13]:
  col=data_train.columns[x]
  ports = pd.get_dummies(X_LR[col], prefix=col)
  X_LR= X_LR.join(ports)
  X_LR.drop([col], axis=1, inplace=True)
```

```
X_LR_train, X_LR_test, y_LR_train, y_LR_test = train_test_split(X_LR,
y_LR, test_size=0.33, random_state=42)


clf = LogisticRegression(random_state=0).fit(X_LR_train, y_LR_train)


y_LR_pred = pd.Series(clf.predict(X_LR_test))


print("Accuracy:", metrics.accuracy_score(y_LR_test, y_LR_pred))
print("Precision:", metrics.precision_score(y_LR_test, y_LR_pred))
print("Recall:", metrics.recall_score(y_LR_test, y_LR_pred))
```

```
Accuracy: 0.802252000744463
Precision: 0.7332601536772777
Recall: 0.2619607843137255
F1 Score:0.38601560242704421519346169698246
```

KNN

```
X_KNN=X_LR.copy()
y_KNN=y_LR.copy()


X_KNN_train, X_KNN_test, y_KNN_train, y_KNN_test = train_test_split(X_KNN,
y_KNN, test_size = 0.33, random_state = 0)


K = []
training = []
test = []
scores = {}

#it was found that the model performed good with
n_neighbours=2


knn = KNeighborsClassifier(n_neighbors=2)
```

```
knn.fit(X_KNN_train, y_KNN_train)

y_KNN_pred = knn.predict(X_KNN_test)

print("Accuracy:", metrics.accuracy_score(y_KNN_test, y_KNN_pred))
print("Precision:", metrics.precision_score(y_KNN_test, y_KNN_pred))
print("Recall:", metrics.recall_score(y_KNN_test, y_KNN_pred))
```

```
Accuracy: 0.788107202680067
Precision: 0.6417112299465241
Recall: 0.2774566473988439
```

Average performance metrics with 10 training and testing splits

Naive Bayes

```
def getAvgPerformance(data2):
  acc_f,pre_f,rec_f,f1_f=0,0,0,0
  for i in range(10):
    data_train_f,data_test_f=splitData(data2)
    acc,pre,rec,f1=NBTest_laplace(data_test_f)
    acc_f+=acc
    pre_f+=pre
    rec_f+=rec
    f1_f+=f1
  return acc_f/10,pre_f/10,rec_f/10,f1_f/10
```

Result

```
Average accuracy: 0.8194073391678183,
Average precision: 0.5945694477150589,
Average recall: 0.7540671030098555,
Average F1score: 0.6648775716763952
```

Logistic Regression

```
def getAvgPerformance_LR(X_LR,y_LR):
  acc,pre,rec,f1=0,0,0,0
  for i in range(10):
    X_LR_train, X_LR_test, y_LR_train, y_LR_test = train_test_split(X_LR,
y_LR, test_size = 0.33, random_state = i)
    y_LR_pred = pd.Series(clf.predict(X_LR_test))
    acc+=metrics.accuracy_score(y_LR_test, y_LR_pred)
    pre+=metrics.precision_score(y_LR_test, y_LR_pred)
    rec+=metrics.accuracy_score(y_LR_test, y_LR_pred)
  return acc/10,pre/10,rec/10,2*pre/10*rec/(pre+rec)
```

Result

```
Average accuracy: 0.800428066257212,
Average precision: 0.8727500959196831,
Average recall: 0.800428066257212,
Average F1score: 0.8350260449179017
```

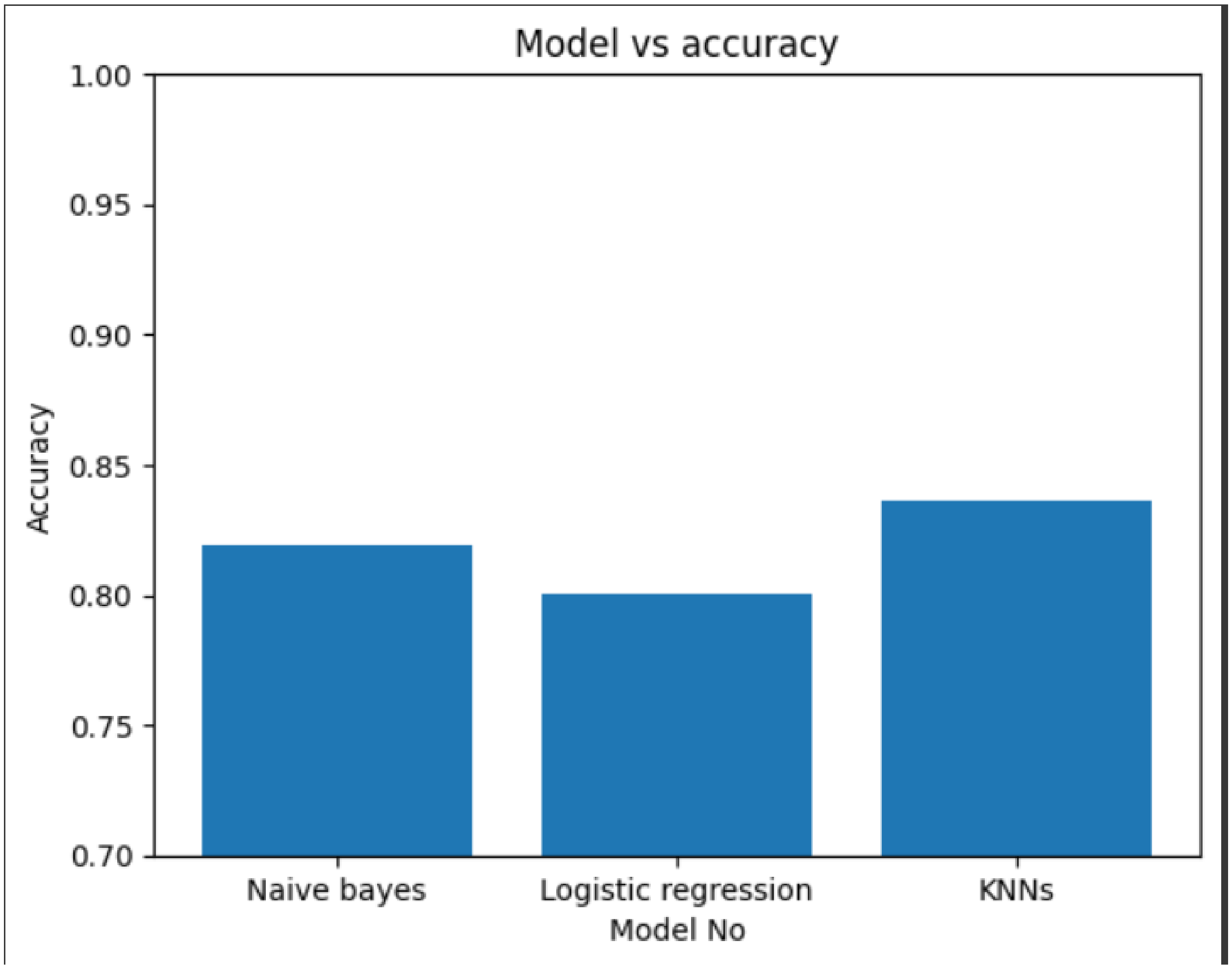
KNN

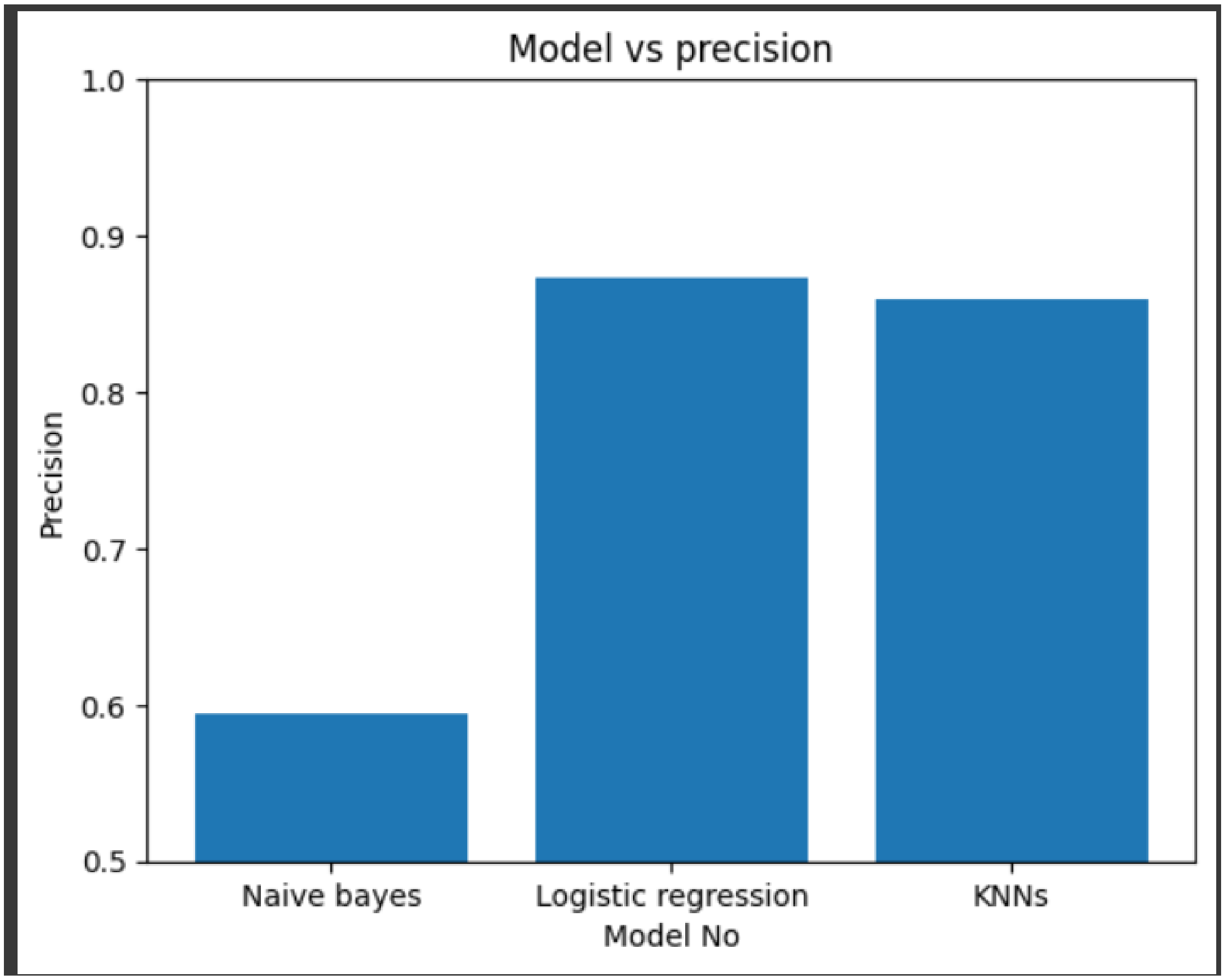
```
def getAvgPerformance_KNN(X_KNN,y_KNN):
  acc,pre,rec,f1=0,0,0,0
  for i in range(10):
    X_KNN_train, X_KNN_test, y_KNN_train, y_KNN_test =
train_test_split(X_KNN, y_KNN, test_size = 0.33, random_state = i)
    y_KNN_pred = knn.predict(X_KNN_test)
    acc+=metrics.accuracy_score(y_KNN_test, y_KNN_pred)
    pre+=metrics.precision_score(y_KNN_test, y_KNN_pred)
    rec+=metrics.accuracy_score(y_KNN_test, y_KNN_pred)
  return acc/10,pre/10,rec/10,2*pre/10*rec/(pre+rec)
```

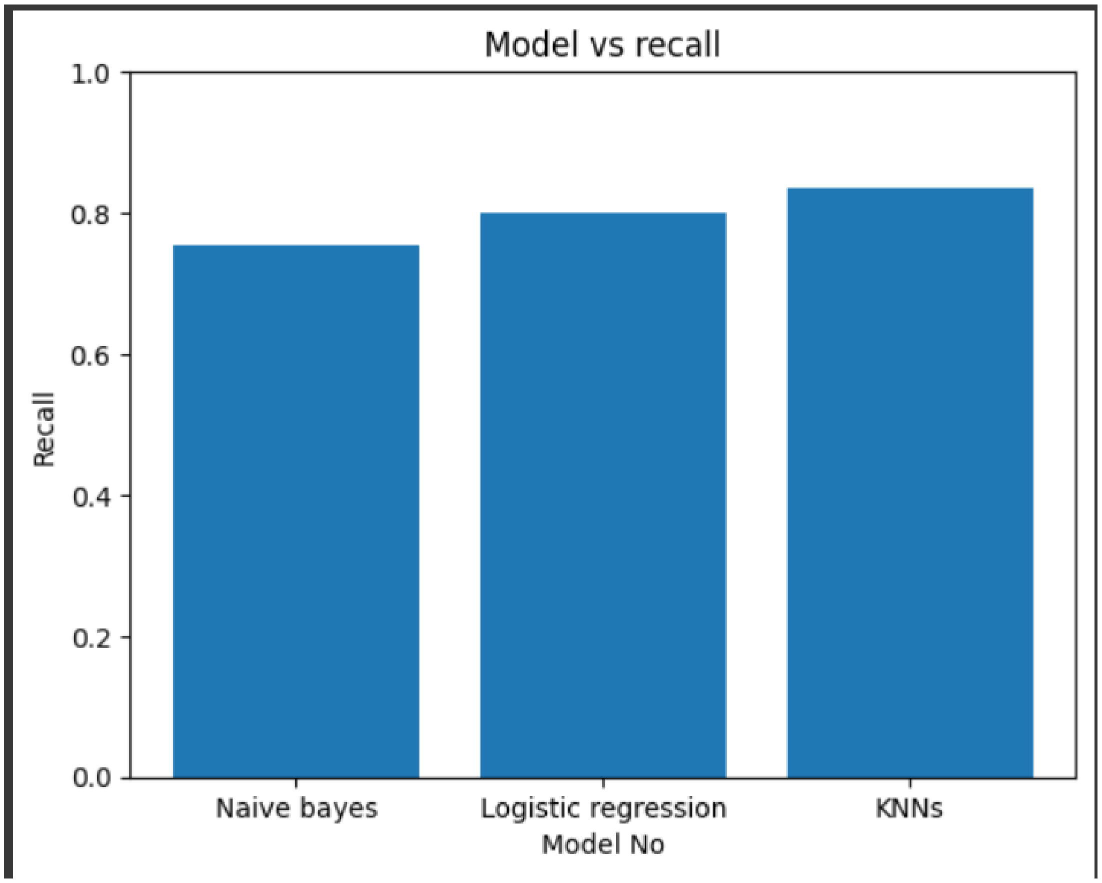
Result

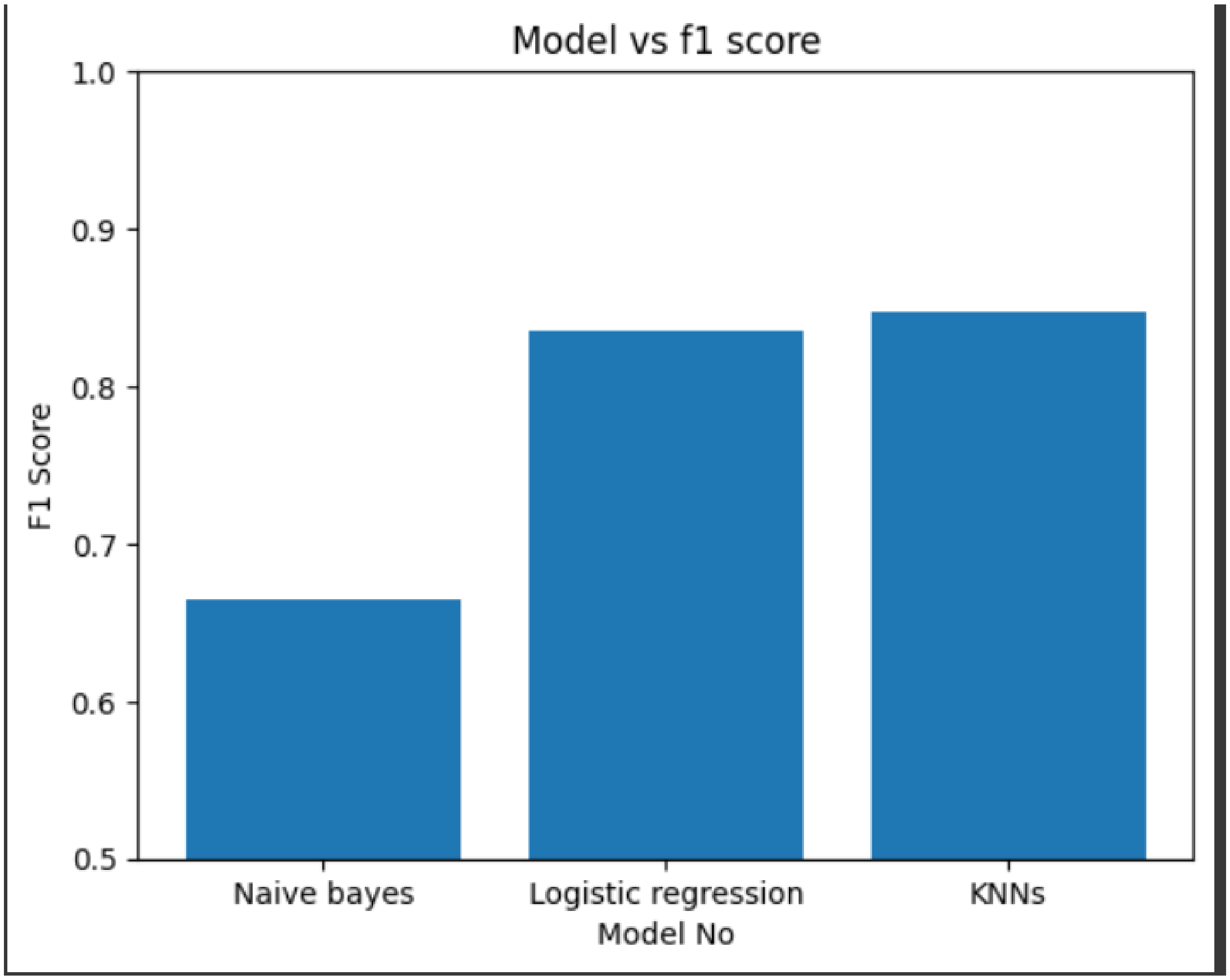
```
Average accuracy:0.8360413176996092,
Average precision: 0.8601440970995154,
Average recall: 0.8360413176996092,
Average F1score: 0.8479214572609478
```

Naive Bayes


Model vs accuracy
1.00
0.95
0.90
0.85
0.80
0.75
0.70
Accuracy
Naive bayes
Logistic regression
KNNs
Model No


Model vs precision
Precision
1.0
0.9
0.8
0.7
0.6
0.5
Naive bayes
Logistic regression
KNNs
Model No


Model vs recall
Recall
1.0
0.8
0.6
0.4
0.2
0.0
Naive bayes
Logistic regression
KNNs
Model No


Model vs f1 score
F1 Score
1.0
0.9
0.8
0.7
0.6
0.5
Naive bayes
Logistic regression
KNNs
Model No