Lab Session: III(b)

(Step-by-Step Implementation of Naïve-Bayes and K-NN Classifier)

Dr. JASMEET SINGH

ASSISTANT PROFESSOR, CSED

TIET, PATIALA

Naïve Bayes-Introduction

- Naïve Bayes classifier is a probabilistic classifier that uses Bayes theorem and Naïve assumption to classify test examples using the training examples.
- According to Bayes Theorem,

$$P(A|B) = \frac{P(A)\,P(B|A)}{P(B)}$$

where P(A|B) is called *posterior probability* of A given B; P(A) is the prior probability of A; P(B|A)is the *likelihood* of B given A; and P(B) is the *evidence* of B.

- For machine learning tasks; A is the target variable (y_i) and B is the input test case $(X = x_1 x_2 x_3 x_4 \ldots \ldots \ldots \ldots \ldots \ldots x_k)$
- Therefore we find, $P(y_i|X) = \frac{P(y_i)\,P(X|y_i)}{P(X)} \quad for\ all\ \ y_i \in Y$

Naïve Bayes-Introduction

- Since P(X) is constant w.r.t different values of y_i. Hence it can be ignored.
- Therefore, $P(y_i|X) \propto P(y_i)\, P(X|y_i)$
- According to Naïve assumption, the probability of each feature in the input is conditionally independent of each other.

$$P(y_i \mid X) = P(y_i)\prod_{j=1}^{n} P(x_j \mid y_i)$$

The final predicted label (y*) for a given input X is thus computed as:

$$y^* = \arg\max_{y} P(y_i)\prod_{j=1}^{n} P(x_j \mid y_i)$$

Computing Likelihood & Class Prior

- The likelihood of each feature (for each unique value of the feature) given each class label; and the class prior probability is computed from training data.

- **Prior Probabilities [P(y_i) for all y_i ε Y] can be learnt by counting the number of exmples in the training data for each class.**
  - If there are N documents/text in training data and n_i out of those are labeled with y_i, then $P(y_i) = n_i/N$
  - **Likelihood [P($x_j|y_i$) for all features x_j ε X and labels y_i in Y] can be computed by counting the number of times feature (x_j) occurs in y_i labeled data.**
  - If $f(x_j \cap y_i)=m$ $f(y_j)=p$

    then $P(x_j|y_i) = \frac{m+\alpha}{p+\alpha\times k}$

    Where α is the smoothing factor and k are the number of input features.

Steps for Naïve Bayes Implementation

Following steps are followed for implementation of Naïve Bayes Classifier:

1. Load the data set.
2. Perform Data Pre-processing (handle null values, noise, correlated features, imbalanced dataset).
3. Split the dataset into training and testing data.
4. Compute class prior probability of each class from the training data.
5. Compute likelihood (probability of each unique value for each feature given distinct values of class labels).
6. Predict the labels of the test data.
7. Perform evaluation between the predicted labels and actual labels.

Steps 1-3

- For implementation of the Naïve Bayes Classifier, we will be working on the **weather dataset** (discussed in class) in which we have to decide that whether the player should play golf or not on the basis of weather conditions (shown in figure).
- Code:

```
import pandas as pd
df = pd.read_csv('C:/Users/jasme/Desktop/weather.csv')
X_train=df.iloc[:,0:4]
Y_train=df.iloc[:,4]
df1=pd.read_csv('C:/Users/jasme/Desktop/weather_test.csv')
X_test=df1.iloc[:,0:4]
Y_test=df1.iloc[:,4]
```

Pre-processing is not required.

Training Data

Outlook	Temp	Humidity	Windy	Play
rainy	hot	high	0	0
rainy	hot	high	1	0
overcast	hot	high	0	1
sunny	mild	high	0	1
sunny	cool	normal	0	1
sunny	cool	normal	1	0
overcast	cool	normal	1	1
rainy	mild	high	0	0
rainy	cool	normal	0	1
sunny	mild	normal	0	1
rainy	mild	normal	1	1
overcast	mild	high	1	1
overcast	hot	normal	0	1
sunny	mild	high	1	0

Test Data

Outlook	Temp	Humidity	Windy	Play
rainy	cool	high	1	0
overcast	mild	normal	0	1

Step 4: Learning Class Prior Probabilities

- Prior Probabilities [P(y_i) for all y_i ε Y] can be learnt by counting the number of exmples in the training data for each class.
- **Code:**

```
#Computing class_priors
import numpy as np
train_size=X_train.shape[0]
class_priors={}
for outcome in np.unique(Y_train):
    outcome_count = sum(Y_train == outcome)
    class_priors[outcome] = outcome_count / train_size
print(class_priors)
```

Step 5: Learning Likelihoods

The likelihood of each feature (for each unique value of the feature) given each class label is computed from training data.

Code:

```
#Computing likelihoods
features=list(X_train.columns)
likelihoods={}
for outcome in np.unique(Y_train):
    outcome_count = sum(Y_train == outcome)
    for feature in features:
        for feat_value in np.unique(X_train[feature]):
            count=0
            for i in range(len(X_train)):
                if(X_train[feature][i]==feat_value  and Y_train[i]==outcome):
                    count=count+1
            likelihoods[(feature,feat_value,outcome)]=(count+1)/(outcome_count+(len(features)))
```

Step 6: Predicting Labels

- In this step, we will first compute probability of each label given each test case example and then we will assign the label for which the probability is maximum.
- **Code:**

```
#Computing probability for each feature value given each class label in the test examples
a=len(np.unique(Y_train))
prob=np.ones((a,len(X_test)),dtype=np.float)
for outcome in(np.unique(Y_train)):
    outcome_count = sum(Y_train == outcome)
    for feature in features:
        for i in range(len(X_test)):
            if  (feature,X_test[feature][i],outcome) in likelihoods.keys():
                prob[outcome][i]=prob[outcome][i]*likelihoods[(feature,X_test[feature][i],outcome)]
            else:
                prob[outcome][i]=prob[outcome][i]*(1/(outcome_count+len(features)))
```

Step 6: Predicting Labels (Contd...)

```
#Multiplying probabilities with class prior probabilities
for i in range(prob.shape[0]):
    prob[i][:]=prob[i][:]*class_priors[i]

#Predicting Labels
Y_label=np.zeros(len(Y_test))
for i in range(len(X_test)):
    if (prob[1,i]>=prob[0,i]):
        Y_label[i]=1
```

Step 7: Performance evaluation

- We can check the performance using classification report and confusion metrics.

Code:

```
from sklearn import metrics
print(metrics.classification_report(Y_test,Y_label))
print(metrics.confusion_matrix(Y_test,Y_label))
```

K-NN Classifier

- K-Nearest Neighbors (k-NN) is a simple algorithm that stores all available cases and predict the numerical target based on a similarity measure.
- There are many distance functions but Euclidean is the most commonly used measure. It is mainly used when data is continuous. Manhattan distance is also very common for continuous variables. The following three distance measures are only valid for continuous variables.

Euclidean $$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

Manhattan $$\sum_{i=1}^{k}|x_i - y_i|$$

Minkowski $$\left(\sum_{i=1}^{k}(|x_i - y_i|)^q\right)^{1/q}$$

K-NN Classifier (Contd.....)

- In the case of categorical variables, Hamming distance is used which is a measure of the number of instances in which corresponding symbols are different in two strings of equal length.

$$D_H = \sum_{i=1}^{k} |x_i - y_i|$$

$$x = y \Rightarrow D = 0$$

$$x \neq y \Rightarrow D = 1$$

X	Y	Distance
Male	Male	0
Male	Female	1

Steps for K-NN Implementation

Following steps are followed for implementation of Naïve Bayes Classifier:

1. Load the data set.
2. Perform Data Pre-processing (handle null values, noise, correlated features, imbalanced dataset).
3. Split the dataset into training and testing data.
4. Calculate the distance between the query-instance (from test set) and all the training samples. Sort the distance and determine nearest neighbors based on the K-th minimum distance.
5. Prediction: Gather the category Y of the nearest neighbors. Use simple majority of the category of nearest neighbors as the prediction value of the query instance.
6. Perform performance evaluation.

Steps 1-3

- In order to implement we will be working on the Iris dataset.

Code:

```
# **Loading dataset**

import pandas as pd

df = pd.read_csv('https://raw.githubusercontent.com/uiuc-cse/data-fa14/gh-pages/data/iris.csv')

# **Pre-processing : Sepearte input and target features; converting output strings to numeric (optional)**

X=df.iloc[:,0:4]

Y=df.iloc[:,4]

from sklearn.preprocessing import LabelEncoder

Y=LabelEncoder().fit_transform(Y)
```

Steps 1-3 (Contd....)

```
#Sepearte train and test set
from sklearn.model_selection  import train_test_split
X_train,X_test,Y_train,Y_test=train_test_split(X,Y,test_size=0.2,random_state=42)
```

Steps 4-5

# Computing distance of each test sample from all the training examples and predicting label according to nearest k- neighbors.

Code:

```
import numpy as np
import heapq
import scipy
Y_label=[]
for i in range(len(X_test)):
    X=np.array(X_test.iloc[i,:]).reshape(1,-1)
    ary = scipy.spatial.distance.cdist(X_train, X, metric='euclidean')
    indx=heapq.nsmallest(5, range(len(ary)), ary.take)
    Y_neighbors=[]
    for k in range(len(indx)):
        Y_neighbors.append(Y_train[indx[k]])
    Y_label.append(max(set(Y_neighbors), key = Y_neighbors.count))
```

Step 6: Performance Evaluation

- We can check the performance using classification report and confusion metrics.

Code:

```
from sklearn import metrics
print(metrics.classification_report(Y_test,Y_label))
print(metrics.confusion_matrix(Y_test,Y_label))
```

How to find best value of K in K-NN?

- In order to choose the best value of K, there are number of techniques.
- We can compute root mean square error between the labeled and actual values for different values of K and can then choose first value for which error if minimum.

Code:

```
test_mse=[]
n_neighbors=[i  for i  in range(1,50)]
for j in range(len(n_neighbors)):
    Y_label=[]
    for i in range(len(X_test)):
        X=np.array(X_test.iloc[i,:]).reshape(1,-1)
        ary = scipy.spatial.distance.cdist(X_train,  X, metric='euclidean')
        indx=heapq.nsmallest(n_neighbors[j],  range(len(ary)),  ary.take)
        Y_neighbors=[]
        for k in range(len(indx)):
            Y_neighbors.append(Y_train[indx[k]])
        Y_label.append(max(set(Y_neighbors),  key = Y_neighbors.count))

test_mse.append(np.sqrt((metrics.mean_squared_error(Y_test,Y_label))))
```

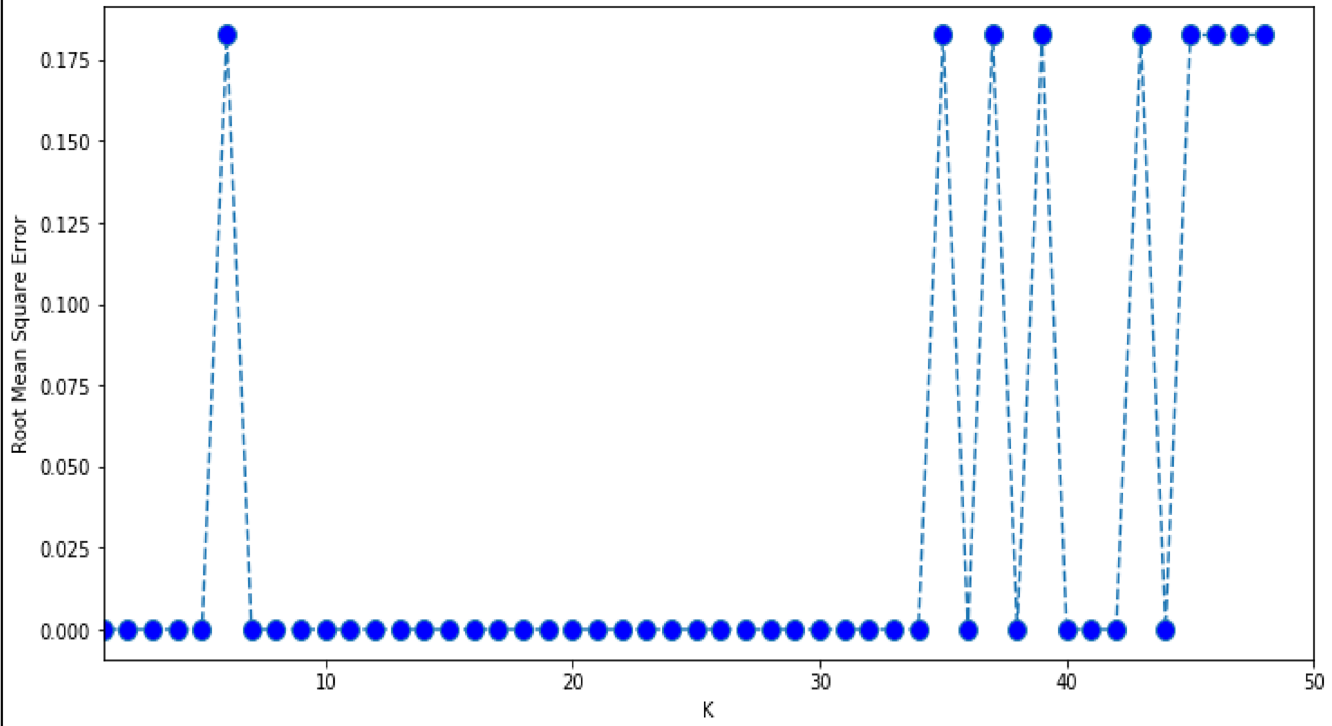
How to find best value of K in K-NN? (Contd….)

- We can then plot error against values of K and choose the best value.
- **Code:**

```
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
plt.xlabel('K')
plt.ylabel('Root Mean Square Error')
plt.xlim([1,50])
plt.plot(test_mse,linestyle='dashed',
marker='o',markerfacecolor='blue',
markersize=10)
```

So, we can set K=1 (because from K=1 to 6; error is zero).

* We can also combine this technique with cross validation for better results.



In-built K-NN Function

- We can also use in-built function **KNeighborsClassifier** from **sklearn.neighbors** as follows:

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, Y_train)
y_pred = classifier.predict(X_test)
y_pred = classifier.predict(X_test)
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(Y_test, y_pred))
print(classification_report(Y_test, y_pred))
```