Introduction To Machine Learning and Applications

Assignment 2: Programming Questions on Generative Classifiers.
Submitted by,
Md Rifat Ul Karim Shovon
Access ID: hr1396

Task List:

1. Download the data. Create an account with Kaggle (if you have not previously done so) and download the heart disease dataset. You should split the 920 instances into training and test sets (8:2) for Naive Bayes classifier.

   While for the k-NN classifier, you need to split the instances into training, validation, and test sets as (6:2:2). The validation set is used to fine-tune the hyper- parameter k. Data can be downloaded from https://www.kaggle.com/datasets/redwankarimsony/heart-disease-data.

2.Train your Naive Bayes and k-NN Classifiers on the training set. (70 points)

3.After training/validation, test them on the test set, construct confusion matrices for the testing set results, and show these confusion matrices calculate accuracy, precision, recall, and F-score. (20 points)

4. Compare the results between the two classifiers. Which classifier performs better? Why? (10 points)

```
import numpy as np
import pandas as pd
from collections import defaultdict
from math import sqrt, exp, pi

# Read the CSV file into a pandas DataFrame
file_path = 'heart_disease_uci.csv'
heart_disease_df = pd.read_csv(file_path)

# Display the first few rows to verify
print(heart_disease_df.head())

# Function to normalize continuous features using z-score
normalization
def normalize_features(X, mean, std):
   return (X - mean) / std

# Step 1: Prepare Data
heart_disease_df_cleaned =
heart_disease_df.drop(columns=['id'])
heart_disease_df_encoded =
pd.get_dummies(heart_disease_df_cleaned, drop_first=True)

X = heart_disease_df_encoded.drop(columns=['num'])  # Features
y = heart_disease_df_encoded['num']             # Labels

# Step 2: Split into training and test sets for Naive Bayes (80-20
split)
train_size = int(0.8 * len(X))
X_train_nb, X_test_nb = X.iloc[:train_size], X.iloc[train_size:]
y_train_nb, y_test_nb = y.iloc[:train_size], y.iloc[train_size:]

# Step 3: Naive Bayes Classifier Implementation
```

```
class NaiveBayes:
   def __init__(self, epsilon=1e-9):  # Add epsilon to avoid division
by zero
      self.class_stats = defaultdict(dict)
      self.epsilon = epsilon  # Small constant for numerical stability
   # Calculate mean and variance for each feature per class
   def calculate_statistics(self, X, y):
      classes = np.unique(y)
      for c in classes:
         X_c = X[y == c]
         self.class_stats[c]['mean'] = X_c.mean(axis=0)
         self.class_stats[c]['var'] = X_c.var(axis=0)
         self.class_stats[c]['prior'] = len(X_c) / len(X)

  # Gaussian probability density function
   def gaussian_probability(self, x, mean, var):
      var = max(var, self.epsilon)  # Ensure variance is never zero
      exponent = exp(-((x - mean) ** 2) / (2 * var))
      return (1 / sqrt(2 * pi * var)) * exponent

   # Calculate class probabilities
   def calculate_class_probabilities(self, X):
      probabilities = {}
      for c, stats in self.class_stats.items():
         probabilities[c] = stats['prior']
         for i in range(len(X)):
            probabilities[c] *= self.gaussian_probability(X[i],
stats['mean'][i], stats['var'][i])
      return probabilities

   # Predict function
   def predict(self, X):
      predictions = []
```

```
    for instance in X:
      probabilities = self.calculate_class_probabilities(instance)
      best_class = max(probabilities, key=probabilities.get)
      predictions.append(best_class)
    return np.array(predictions)

# Step 4: Train Naive Bayes
nb_classifier = NaiveBayes()
nb_classifier.calculate_statistics(X_train_nb.values,
y_train_nb.values)

# Step 5: Test Naive Bayes
y_pred_nb = nb_classifier.predict(X_test_nb.values)

# Step 6: Calculate Naive Bayes Accuracy
nb_accuracy = np.mean(y_pred_nb == y_test_nb.values)
print(f'Naive Bayes Test Accuracy: {nb_accuracy}')

# Step 7: k-NN Classifier Implementation
class KNN:
  def __init__(self, k=5):
    self.k = k

  # Euclidean distance function
  def euclidean_distance(self, instance1, instance2):
    return sqrt(np.sum((instance1 - instance2) ** 2))

  # Get the k-nearest neighbors
  def get_neighbors(self, X_train, y_train, test_instance):
    distances = []
    for i in range(len(X_train)):
      dist = self.euclidean_distance(test_instance, X_train[i])
      distances.append((y_train[i], dist))
```

```
        distances.sort(key=lambda x: x[1])
        neighbors = distances[:self.k]
        return neighbors

    # Predict the class based on majority voting
    def predict(self, X_train, y_train, X_test):
        predictions = []
        for test_instance in X_test:
            neighbors = self.get_neighbors(X_train, y_train,
test_instance)
            classes = [neighbor[0] for neighbor in neighbors]
            predicted_class = max(set(classes), key=classes.count)
            predictions.append(predicted_class)
        return np.array(predictions)

# Step 8: Split for k-NN (60% training, 20% validation, 20% test)
train_size_knn = int(0.6 * len(X))
val_size_knn = int(0.2 * len(X))
X_train_knn, X_val_knn, X_test_knn = np.split(X.values,
[train_size_knn, train_size_knn + val_size_knn])
y_train_knn, y_val_knn, y_test_knn = np.split(y.values,
[train_size_knn, train_size_knn + val_size_knn])

# Step 9: Tune k for k-NN using validation set
best_k = 1
best_accuracy = 0
for k in range(1, 21):
    knn_classifier = KNN(k=k)
    y_val_pred = knn_classifier.predict(X_train_knn, y_train_knn,
X_val_knn)
    val_accuracy = np.mean(y_val_pred == y_val_knn)
    if val_accuracy > best_accuracy:
        best_accuracy = val_accuracy
```

```
        best_k = k

# Step 10: Train k-NN with the best k and test on the test set
knn_classifier_best = KNN(k=best_k)
y_pred_knn = knn_classifier_best.predict(X_train_knn, y_train_knn,
X_test_knn)

# Step 11: Calculate k-NN accuracy
knn_accuracy = np.mean(y_pred_knn == y_test_knn)
print(f'k-NN Test Accuracy: {knn_accuracy} (Best k={best_k})')
##################################################
# Function to construct the confusion matrix
def confusion_matrix(y_true, y_pred):
    TP = np.sum((y_true == 1) & (y_pred == 1))
    TN = np.sum((y_true == 0) & (y_pred == 0))
    FP = np.sum((y_true == 0) & (y_pred == 1))
    FN = np.sum((y_true == 1) & (y_pred == 0))

    return TP, TN, FP, FN

# Function to calculate accuracy, precision, recall, F1-score
def calculate_metrics(TP, TN, FP, FN):
    accuracy = (TP + TN) / (TP + TN + FP + FN)
    precision = TP / (TP + FP) if (TP + FP) != 0 else 0
    recall = TP / (TP + FN) if (TP + FN) != 0 else 0
    f1_score = 2 * precision * recall / (precision + recall) if (precision
+ recall) != 0 else 0

    return accuracy, precision, recall, f1_score



# Naive Bayes results (for example)
```

```
TP_nb, TN_nb, FP_nb, FN_nb = confusion_matrix(y_test_nb,
y_pred_nb)

# k-NN results
TP_knn, TN_knn, FP_knn, FN_knn = confusion_matrix(y_test_knn,
y_pred_knn)

# Calculate metrics for Naive Bayes
nb_accuracy, nb_precision, nb_recall, nb_f1_score =
calculate_metrics(TP_nb, TN_nb, FP_nb, FN_nb)
print(f"Naive Bayes Confusion Matrix: TP={TP_nb}, TN={TN_nb},
FP={FP_nb}, FN={FN_nb}")
print(f"Naive Bayes Metrics: Accuracy={nb_accuracy:.4f},
Precision={nb_precision:.4f}, Recall={nb_recall:.4f}, F1
Score={nb_f1_score:.4f}")

# Calculate metrics for k-NN
knn_accuracy, knn_precision, knn_recall, knn_f1_score =
calculate_metrics(TP_knn, TN_knn, FP_knn, FN_knn)
print(f"k-NN Confusion Matrix: TP={TP_knn}, TN={TN_knn},
FP={FP_knn}, FN={FN_knn}")
print(f"k-NN Metrics: Accuracy={knn_accuracy:.4f},
Precision={knn_precision:.4f}, Recall={knn_recall:.4f}, F1
Score={knn_f1_score:.4f}")
```

Output from the shell :

```
hr1396@DS035010:/mnt/c/Users/hr1396/OneDrive - Wayne State University/PhD semesters/Fall 2024/Introduction to machine learning/ML_assignments/Generative_clclassc
classifier$ python3 final.py
   id  age     sex               cp  trestbps   chol    fbs         restecg  thalch  exang  oldpeak        slope   ca               thal  num
0   1   63    Male   typical angina     145.0  233.0   True  lv hypertrophy   150.0  False      2.3  downsloping  0.0       fixed defect    0
1   2   67    Male     asymptomatic     160.0  286.0  False  lv hypertrophy   108.0   True      1.5         flat  3.0             normal    2
2   3   67    Male     asymptomatic     120.0  229.0  False  lv hypertrophy   129.0   True      2.6         flat  2.0  reversable defect    1
3   4   37    Male      non-anginal     130.0  250.0  False          normal   187.0  False      3.5  downsloping  0.0             normal    0
4   5   41  Female  atypical angina     130.0  204.0  False  lv hypertrophy   172.0  False      1.4    upsloping  0.0             normal    0
Naive Bayes Test Accuracy: 0.24456521739130435
k-NN Test Accuracy: 0.24456521739130435 (Best k=1)
Naive Bayes Confusion Matrix: TP=0, TN=45, FP=0, FN=50
Naive Bayes Metrics: Accuracy=0.4737, Precision=0.0000, Recall=0.0000, F1 Score=0.0000
k-NN Confusion Matrix: TP=1, TN=44, FP=0, FN=49
k-NN Metrics: Accuracy=0.4787, Precision=1.0000, Recall=0.0200, F1 Score=0.0392
```

k-NN performs better overall due to the higher Recall and F1-Score, which makes it better in identifying heart patients. However, Naive Bayes's higher precision suggests that k-NN makes fewer false positive errors. Which classifier to choose probably depends on what exactly is required from the task-at hand-if it is crucial to minimize false positives, one may want to use Naive Bayes. If we want better recall (minimizing false negatives), k-NN is the superior choice.