Assignment-5: Other Classical Supervised Machine Learning

Sajan Kumer Sarker
Id: 2111131642
Email: sajan.sarker@northsouth.edu

Rafsan Jani Chowdhury
Id: 2011424642
Email: rafsan.chowdhury@northsouth.edu

Rosely Mohammad
Id: 2014219642
Email: rosely.mohammad@northsouth.edu

November 22, 2024

1. Introduction

In this assignment report, we will discuss about the training, evaluation, and testing process of six different model: ZeroR Classifier, OneR Classifier, K-Nearest-Neighbor Classifier, Naive Bayesian Classifier, Support Vector Machine Classifier, and Support Vector Machine Regression on four different datasets: 'Car Evaluation (for Multi-class Classification)', 'DT-BrainCancer (for Binary Classification)', 'DT-Wage (for Regression)', 'DT-Credits (for Regression)'. For these models we will report Accuracy, Precision, Average Precision, Recall, Average Recall, Confusion Matrix, F-Score, Precision-Recall Curve and Mean-Squared Error. Here, we split this assignment into 4 parts: 1. Binary Classification on 'DT-BrainCancer' dataset, 2. Multi-class Classification on 'Car Evaluation' dataset, 3. Regression on 'DT-Wage' dataset, 4. Regression on 'DT-Credits' dataset. Each part is handled in a separate notebook file, with each file focused on a single dataset. After implementing Binary Classification problem, we follow same step for Multi-class classification problem as well, and for Regression problem we follow same step in both two files. To implement these models, we have follow multiples resources. [1] [2] [3] [4] [5] [6]

2. Methods

Necessary Imports:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import math
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler
from sklearn.svm import SVC, SVR
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from collections import defaultdict
```

We have imported all necessary libraries and modules to be used in each notebook.

User Define Evaluation Metrics Functions:

```python
# functions to calculate: accuracy, average precision, average recall, average f-score.

# Accuracy
def accuracy_score(y_actual, y_predict):
  correct = 0
  total_samples = len(y_actual)

  for predict, actual in zip(y_predict, y_actual):
    if predict == actual:
      correct += 1

  return (correct/ total_samples)

# Precision
def precision_score(y_actual, y_predict):
  precision_scores = []

  for class_label in set(y_actual):
    true_positive = 0
    false_positive = 0

    for actual, predict in zip(y_actual, y_predict):
      if predict == class_label:
        if actual == class_label:
          true_positive += 1
        else:
          false_positive += 1

    if true_positive+false_positive ==0:
      class_precision = 0.0
    else:
      class_precision = true_positive / (true_positive + false_positive)
    precision_scores.append(class_precision)
  return precision_scores

# average precision
def average_precision_score_macro(precision_score, y_actual):
  return sum(precision_score)/len(set(y_actual))

# recall
def recall_score(y_actual, y_predict):
  recall_scores = []

  for class_label in set(y_actual):
    true_positive = 0
    false_negative = 0

    for actual, predict in zip(y_actual, y_predict):
      if actual == class_label:
        if predict == class_label:
          true_positive += 1
        else:
          false_negative += 1

```

```
    if true_positive+false_negative ==0:
      class_recall = 0.0
    else:
      class_recall = true_positive / (true_positive + false_negative)

    recall_scores.append(class_recall)
  return recall_scores

# average recall
def average_recall_score_macro(recall_score, y_actual):
  return sum(recall_score)/len(set(y_actual))

# f-score
def fscore(y_actual,y_predict):
  f_score_total = []

  for class_label in set(y_actual):
    true_positive = 0
    false_positive = 0
    false_negative = 0

    for actual, predict in zip(y_actual, y_predict):
      if predict == class_label:
        if actual == class_label:
          true_positive += 1
        else:
          false_positive += 1
      elif actual == class_label:
        false_negative += 1

    if true_positive + false_positive == 0:
      precision = 0.0
    else:
      precision = true_positive/(true_positive+false_positive)
    if true_positive + false_negative == 0:
      precision = 0.0
    else:
      recall = true_positive/(true_positive+false_negative)
    if precision + recall ==0:
      f_score = 0.0
    else:
      f_score = 2*true_positive/((2*true_positive)+false_positive+false_negative)
    f_score_total.append(f_score)
  return f_score_total

# fscore average
def fscore_average_macro(fscore, y_actual):
  return sum(fscore)/len(set(y_actual))

# display confusion matrix
def display_confusion_matrix(y_test, y_predict):
  num_class = len(set(y_test))
  class_labels = list(set(y_test))

  plt.figure(figsize=(8, 6))

  conf_matrix = np.zeros((num_class, num_class), dtype=int)
  for actual, predict in zip(y_test, y_predict):
```

```
        conf_matrix[actual, predict] += 1

    plt.imshow(conf_matrix, cmap='Blues', interpolation='nearest')
    plt.title("Confusion␣Matrix", fontsize=16, fontweight='bold')
    plt.xlabel("Predicted␣Labels", fontsize=12)
    plt.ylabel("True␣Labels", fontsize=12)

    for i in range(num_class):
        for j in range(num_class):
            plt.text(j, i, str(conf_matrix[i, j]), ha='center', va='center', color='red', fontsi

    plt.xticks(ticks=range(num_class), labels=class_labels, fontsize=10)
    plt.yticks(ticks=range(num_class), labels=class_labels, fontsize=10)
    plt.tight_layout()
    plt.show()
```

Here we define some functions manually to report the evaluation matrices (f-score, precision, recall, average precision, average recall, confusion matrix) without using scikit learn's evaluation matrices library function.

2.1 Binary Classification on 'DT-BrainCancer' dataset

Loading Dataset:

```
df = pd.read_csv('/content/drive/MyDrive/CSE445-Assignment/dataset/DT-BrainCancer.csv')
print(df.shape)
print(df.head())
```

Load the dataset using pandas library and check first few data from the dataset and its shape.

Checking Missing Values:

```
print('Number␣of␣missing␣values:␣')
print(df.isnull().sum())
```

Checking if any missing values are founds in any features.

Drop Column:

```
df = df.drop(columns=['Unnamed:␣0', 'sex'])
df = df.dropna()
print(df.shape)
print(df.head())
```

Drop the unwanted column and remove the missing values row.

Encoded Categorical Data:

```
le = LabelEncoder()
df['diagnosis'] = le.fit_transform(df['diagnosis'])
df['loc'] = le.fit_transform(df['loc'])
print(df.head(), df.tail())
```

Since there are few complex relationships between the categories, we encoded them using LabelEnocer. Then we print some samples from the first and some from the last of this encoded dataset.

Split Data into Training, Validation and Test Sets:

```
X = df.drop(columns=['status'])
y = df['status']

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

# this will used after cross-validation
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

print(f'Train Data Shape (X, y): {X_train.shape, y_train.shape}')
print(f'Validation Data Shape (X, y): {X_val.shape, y_val.shape}')
print(f'Test Data Shape (X, y): {X_test.shape, y_test.shape}')
```

Splitting 70 percent data for training, 15 percent for validation and rest 15 percent for testing. And concatenate the train & validation set to retrain the model.

Implementing ZeroR Classifier Model:

```
class_counts = y.value_counts()
majority_class = class_counts.idxmax()
majority_count = class_counts.max()

baseline_accuracy = majority_count/len(y) * 100   # calculate baseline accuracy

print(f"Majority Class: {majority_class}")
print(f"ZeroR Baseline Accuracy: {baseline_accuracy:.2f}%")

# plot data
plt.figure(figsize=(6, 4))
plt.title(f'ZeroR Baseline Class Distribution')
plt.bar(class_counts.index, class_counts.values, color=['lightblue', 'lightcoral'])
plt.xticks([0,1],['0','1'])
plt.xlabel('class')
plt.ylabel('count')
plt.show()
```

Implemented the ZeroR classifier model that calculates the ZeroR baseline accuracy, which measures the accuracy of always predicting the majority class in a classification problem. Identifies the majority class in the target variable and computes the percentage of data points belonging to this class to determine the accuracy of the baseline. This serves as a simple benchmark to compare the performance of more complex and advance models.

Implementing OneR Classifier Model:

```
# OneR Classifier implementation
class OneRClassifier:
  def __init__(self):
    self.rule = None

  def fit(self, X, y):
    best_rule = None
    best_error = float('inf')

    for column in X.columns:
      freq_table = defaultdict(lambda: defaultdict(int))
      for value, label in zip(X[column], y):
        freq_table[value][label] += 1
```

```
      error_rate = sum(max(freq_table[value].values()) for value in freq_table)/len(y)

      if error_rate < best_error:
        best_rule = (column, freq_table)
        best_error = error_rate
    self.rule = best_rule

  def predict(self, X):
    if self.rule is None:
      raise Exception("Classifier␣has␣not␣been␣trained␣yet!")
    column, freq_table = self.rule

    predict = []
    for value in X[column]:
      predict.append(max(freq_table[value], key=freq_table[value].get))
    return predict
```

Implemented the OneR Classifier model that is a simple rule-based algorithm that selects the single best predictor from the dataset based on its ability to minimize classification error. During training, it evaluates each feature by creating a frequency table of feature values and target labels, then selects the feature with the lowest error as the best rule. This rule is used to assign the majority class for new data based on the stored frequency table.

Implementing ZeroR Classifier Model:

```
#oneR classifier initialize
oner = OneRClassifier()
oner.fit(X_train, y_train)

y_val_predict = oner.predict(X_val)
print("OneR␣validation␣set␣Accuracy:", accuracy_score(y_val, y_val_predict))

y_test_predict = oner.predict(X_test)
print("OneR␣test␣set␣Accuracy:", accuracy_score(y_test, y_test_predict))
```

Initialize the OneR Classifier and train it using train set, then evaluate it on validation and test set, and find the overall accuracy.

Implementing KNN, Naive Bayesian and SVM Base Model:

```
# Base model train and accuracy on validation dataset
knn_base = KNeighborsClassifier()
svc_base = SVC()
nb_base = GaussianNB()

knn_base.fit(X_train, y_train)
svc_base.fit(X_train, y_train)
nb_base.fit(X_train, y_train)

knn_pred_val = knn_base.predict(X_val)
svc_pred_val = svc_base.predict(X_val)
nb_pred_val = nb_base.predict(X_val)

print("KNN␣Accuracy:", accuracy_score(y_val, knn_pred_val))
print("SVM␣Accuracy:", accuracy_score(y_val, svc_pred_val))
print("Naive␣Bayes␣Accuracy:", accuracy_score(y_val, nb_pred_val))
```

Implemented the KNN, Naive Bayesian and SVM Classification Base model and train these models. Then evaluate each of these models on validation set and find the overall accuracy of these models.

Optimizing KNN Model with GridSearchCV:

```
# initialize the classifier models
knn_classifier = KNeighborsClassifier(n_neighbors=5)

knn_param_grid = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}

knn_grid = GridSearchCV(estimator=knn_classifier, param_grid=knn_param_grid, cv=5, scoring='ac
knn_grid.fit(X_train, y_train)

knn_best_params = knn_grid.best_params_
print("Best KNN Parameters:", knn_best_params)
```

We initializes a KNN Classifier and perform hyper-parameter tuning using GridSearchCV to identify the best configuration model. This GridSearchCV evaluates all combinations of these hyper-parameters using 5-fold cross-validation and accuracy as the scoring metric. Then model was trained on the training set, and best set of hyper-parameters is selected and printed.

Initialize New KNN model with best Hyper-parameters:

```
knn_final_model = KNeighborsClassifier(**knn_best_params)
knn_final_model.fit(X_train, y_train)

knn_pred_val = knn_final_model.predict(X_val)
print("KNN Accuracy:", accuracy_score(y_val, knn_pred_val))
```

Initialize a new KNN model with the best hyper-parameters that we found from the cross-validation. And find the models accuracy on validation set.

Retrain the KNN model and Evaluate it on Testset:

```
# retrain knn final model using train_val set and evaluate on test set
knn_final_model.fit(X_train_val, y_train_val)

y_test_pred_knn = knn_final_model.predict(X_test)

print("KNN: ")
print(f'Accuracy: {accuracy_score(y_test, y_test_pred_knn)}')
print(f'Precision: {precision_score(y_test, y_test_pred_knn)}')
print(f'Recall: {recall_score(y_test, y_test_pred_knn)}')
print(f'F1-Score: {fscore(y_test, y_test_pred_knn)}')
print()

display_confusion_matrix(y_test, y_test_pred_knn)

from sklearn.metrics import precision_recall_curve, average_precision_score
y_scores_knn = knn_final_model.predict_proba(X_test)[:, 1]

precision, recall, thresholds = precision_recall_curve(y_test, y_scores_knn)
average_precision = average_precision_score(y_test, y_scores_knn)

plt.figure(figsize=(8, 6))
plt.plot(recall, precision, label=f'Precision-Recall curve (AP = {average_precision:.2f})')
plt.xlabel('Recall', fontsize=14)
plt.ylabel('Precision', fontsize=14)
plt.title('Precision-Recall Curve', fontsize=16)
plt.legend(loc='best')
```

```
plt.grid()
plt.show()
```

Retrain the new KNN model with the concatenated train, validation set. Then we Evaluate the model on test set and find and plot the models classification accuracy, precision score, recall score, f1-score, confusion matrix, and the precision-recall curve. Here we used the scikit-learn library function to plot the precision-recall curve, because implementing it manually was too much tough.

Optimizing Naive Bayesian Model with GridSearchCV:

```
# initialize the classifier models
nb_classifier = GaussianNB()

nb_param_grid = {
    ’var_smoothing’: [1e-9, 1e-8, 1e-7, 1e-6, 1e-5]
}

nb_grid = GridSearchCV(estimator=nb_classifier, param_grid=nb_param_grid, cv=5, scoring=’accura

nb_grid.fit(X_train, y_train)

nb_best_params = nb_grid.best_params_
print("Best␣KNN␣Parameters:", nb_best_params)
```

We initializes a Naive Bayesian Classifier and perform hyper-parameter tuning using GridSearchCV to identify the best configuration model. This GridSearchCV evaluates all combinations of these hyper-parameters using 5-fold cross-validation and accuracy as the scoring metric. Then model was trained on the training set, and best set of hyper-parameters is selected and printed.

Initialize New Naive Bayesian model with best Hyper-parameters:

```
nb_final_model = GaussianNB(**nb_best_params)
nb_final_model.fit(X_train, y_train)

nb_pred_val = nb_final_model.predict(X_val)
print("Naive␣Bayesian␣Accuracy:", accuracy_score(y_val, nb_pred_val))
```

Initialize a new Naive Bayesian model with the best hyper-parameters that we found from the cross-validation. And find the models accuracy on validation set.

Retrain the Naive Bayesian model and Evaluate it on Testset:

```
# retrain naive bayesian final model using train_val set and evaluate on test set
nb_final_model.fit(X_train_val, y_train_val)

y_test_pred_nb = nb_final_model.predict(X_test)

print("Naive␣Bayesian:␣")
print(f’Accuracy:␣{accuracy_score(y_test,␣y_test_pred_nb)}’)
print(f’Precision:␣{precision_score(y_test,␣y_test_pred_nb)}’)
print(f’Recall:␣{recall_score(y_test,␣y_test_pred_nb)}’)
print(f’F1-Score:␣{fscore(y_test,␣y_test_pred_nb)}’)
print()

display_confusion_matrix(y_test, y_test_pred_nb)

y_scores_nb = nb_final_model.predict_proba(X_test)[:, 1]

precision, recall, thresholds = precision_recall_curve(y_test, y_scores_nb)
```

```
average_precision = average_precision_score(y_test, y_scores_nb)

plt.figure(figsize=(8, 6))
plt.plot(recall, precision, label=f'Precision-Recall curve (AP = {average_precision:.2f})')
plt.xlabel('Recall', fontsize=14)
plt.ylabel('Precision', fontsize=14)
plt.title('Precision-Recall Curve', fontsize=16)
plt.legend(loc='best')
plt.grid()
plt.show()
```

Retrain the new Naive Bayesian model with the concatenated train, validation set. Then we Evaluate the model on test set and find and plot the models classification accuracy, precision score, recall score, f1-score, confusion matrix, and the precision-recall curve.

Optimizing SVM Model with GridSearchCV:

```
# initialize the classifier models
svm_classifier = SVC()

svm_param_grid = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'gamma': ['scale', 'auto']
}

svm_grid = GridSearchCV(estimator=svm_classifier, param_grid=svm_param_grid, cv=5, scoring='ac
svm_grid.fit(X_train, y_train)

svm_best_params = svm_grid.best_params_
print("Best SVM Parameters:", svm_best_params)
```

Here, we do the same thing again. We initializes a SVM Classifier and perform hyper-parameter tuning using GridSearchCV to identify the best configuration model. This GridSearchCV evaluates all combinations of these hyper-parameters using 5-fold cross-validation and accuracy as the scoring metric. Then model was trained on the training set, and best set of hyper-parameters is selected and printed.

Initialize New SVM model with best Hyper-parameters:

```
svm_final_model = SVC(probability=True, **svm_best_params)
svm_final_model.fit(X_train, y_train)

svm_pred_val = svm_final_model.predict(X_val)
print("SVM validation set Accuracy:", accuracy_score(y_val, svm_pred_val))
```

Here also we do the same thing again. Initialize a new SVM model with the best hyper-parameters that we found from the cross-validation. And find the models accuracy on validation set.

Retrain the SVM model and Evaluate it on Testset:

```
# retrain svm final model using train_val set and evaluate on test set
svm_final_model.fit(X_train_val, y_train_val)

y_test_pred_svm = svm_final_model.predict(X_test)

print("SVM Test: ")
print(f'Accuracy: {accuracy_score(y_test, y_test_pred_svm)}')
print(f'Precision: {precision_score(y_test, y_test_pred_svm)}')
print(f'Recall: {recall_score(y_test, y_test_pred_svm)}')
```

```
print(f'F1-Score: {fscore(y_test, y_test_pred_svm)}')
print()

display_confusion_matrix(y_test, y_test_pred_svm)

y_scores_svm = svm_final_model.predict_proba(X_test)[:, 1]

precision, recall, thresholds = precision_recall_curve(y_test, y_scores_svm)
average_precision = average_precision_score(y_test, y_scores_svm)

plt.figure(figsize=(8, 6))
plt.plot(recall, precision, label=f'Precision-Recall curve (AP = {average_precision:.2f})')
plt.xlabel('Recall', fontsize=14)
plt.ylabel('Precision', fontsize=14)
plt.title('Precision-Recall Curve', fontsize=16)
plt.legend(loc='best')
plt.grid()
plt.show()
```

Here again Retrain the new SVM model with the concatenated train, validation set. Then we Evaluate the model on test set and find and plot the models classification accuracy, precision score, recall score, f1-score, confusion matrix, and the precision-recall curve.

2.2 Multi-class Classification on 'Car Evaluation' dataset

Loading Dataset:

```
car_evaluation = pd.read_csv('/content/drive/MyDrive/CSE445-Assignment/dataset/car.data')

car_evaluation.head()
```

Load the Dataset using pandas library and check first few data from the dataset.

Assign Column name:

```
column_names = ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety', 'class']
car_evaluation.columns = column_names
car_evaluation.head()
```

Assign each columns with a name, since the original dataset didn't have any headings.

Checking Missing Values:

```
print('Number of missing values: ')
print(car_evaluation.isnull().sum())
```

Checking if any missing values are founds in any features.

Mapping Categorical Data:

```
car_evaluation['buying'] = car_evaluation['buying'].map({'vhigh':3, 'high':2, 'med':1, 'low':0
car_evaluation['maint'] = car_evaluation['maint'].map({'vhigh':3, 'high':2, 'med':1, 'low':0})
car_evaluation['doors'] = car_evaluation['doors'].map({'2':0, '3':1, '4':2, '5more':3})
car_evaluation['persons'] = car_evaluation['persons'].map({'2':0, '4':1, 'more':2})
car_evaluation['lug_boot'] = car_evaluation['lug_boot'].map({'small':0, 'med':1, 'big': 2})
```

```
car_evaluation['safety'] = car_evaluation['safety'].map({'low':0, 'med':1, 'high': 2})

car_evaluation['class'] = car_evaluation['class'].map({'unacc':0, 'acc':1, 'good': 2, 'vgood':
car_evaluation.head()
```

Since there is no complex relationship between the categories, so we just mapped them to values ranging from 0 to 4. Then each categorical columns are encoded in 0 to 4.

Split Data into Training, Validation and Test Sets:

```
X = car_evaluation.drop(columns='class')
y = car_evaluation['class']
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

# this will used after cross-validation
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

print(f'Train Data Shape (X, y): {X_train.shape, y_train.shape}')
print(f'Validation Data Shape (X, y): {X_val.shape, y_val.shape}')
print(f'Test Data Shape (X, y): {X_test.shape, y_test.shape}')
```

Splitting 70 percent data for training, 15 percent for validation and rest 15 percent for testing. And concatenate the train & validation set to retrain the model.

Implementing ZeroR Classifier Model:

```
class_counts = y.value_counts()
majority_class = class_counts.idxmax()
majority_count = class_counts.max()

baseline_accuracy = majority_count/len(y) * 100   # calculate baseline accuracy

print(f"Majority Class: {majority_class}")
print(f"ZeroR Baseline Accuracy: {baseline_accuracy:.2f}%")

# plot data
plt.figure(figsize=(6, 4))
plt.title(f'ZeroR Baseline Class Distribution')
plt.bar(class_counts.index, class_counts.values, color=['lightblue', 'lightcoral'])
plt.xticks([0,1,2,3],['0','1','2','3'])
plt.xlabel('class')
plt.ylabel('count')
plt.show()
```

Implemented the ZeroR classifier model that calculates the ZeroR baseline accuracy, which measures the accuracy of always predicting the majority class in a classification problem. Identifies the majority class in the target variable and computes the percentage of data points belonging to this class to determine the accuracy of the baseline. This serves as a simple benchmark to compare the performance of more complex and advance models.

Implementing OneR Classifier Model:

```
# OneR Classifier implementation
class OneRClassifier:
  def __init__(self):
    self.rule = None

```

```
  def fit(self, X, y):
    best_rule = None
    best_error = float('inf')

    for column in X.columns:
      freq_table = defaultdict(lambda: defaultdict(int))
      for value, label in zip(X[column], y):
        freq_table[value][label] += 1

      error_rate = sum(max(freq_table[value].values()) for value in freq_table)/len(y)

      if error_rate < best_error:
        best_rule = (column, freq_table)
        best_error = error_rate
    self.rule = best_rule

  def predict(self, X):
    if self.rule is None:
      raise Exception("Classifier has not been trained yet!")
    column, freq_table = self.rule

    predict = []
    for value in X[column]:
      predict.append(max(freq_table[value], key=freq_table[value].get))
    return predict
```

Implemented the OneR Classifier model that is a simple rule-based algorithm that selects the single best predictor from the dataset based on its ability to minimize classification error. During training, it evaluates each feature by creating a frequency table of feature values and target labels, then selects the feature with the lowest error as the best rule. This rule is used to assign the majority class for new data based on the stored frequency table.

Implementing ZeroR Classifier Model:

```
#oneR classifier initialize
oner = OneRClassifier()
oner.fit(X_train, y_train)

y_val_predict = oner.predict(X_val)
print("OneR validation set Accuracy:", accuracy_score(y_val, y_val_predict))

y_test_predict = oner.predict(X_test)
print("OneR test set Accuracy:", accuracy_score(y_test, y_test_predict))
```

Initialize the OneR Classifier and train it using train set, then evaluate it on validation and test set, and find the overall accuracy.

Implementing KNN, Naive Bayesian and SVM Base Model:

```
# Base model train and accuracy on validation dataset
knn_base = KNeighborsClassifier()
svc_base = SVC()
nb_base = GaussianNB()

knn_base.fit(X_train, y_train)
svc_base.fit(X_train, y_train)
nb_base.fit(X_train, y_train)

knn_pred_val = knn_base.predict(X_val)
svc_pred_val = svc_base.predict(X_val)
```

```
nb_pred_val = nb_base.predict(X_val)

print("KNN Accuracy:", accuracy_score(y_val, knn_pred_val))
print("SVM Accuracy:", accuracy_score(y_val, svc_pred_val))
print("Naive Bayes Accuracy:", accuracy_score(y_val, nb_pred_val))
```

Implemented the KNN, Naive Bayesian and SVM Classification Base model and train these models. Then evaluate each of these models on validation set and find the overall accuracy of these models.

Optimizing KNN Model with GridSearchCV:

```
# initialize the classifier models
knn_classifier = KNeighborsClassifier(n_neighbors=5)

knn_param_grid = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}

knn_grid = GridSearchCV(estimator=knn_classifier, param_grid=knn_param_grid, cv=5, scoring='ac
knn_grid.fit(X_train, y_train)

knn_best_params = knn_grid.best_params_
print("Best KNN Parameters:", knn_best_params)
```

We initializes a KNN Classifier and perform hyper-parameter tuning using GridSearchCV to identify the best configuration model. This GridSearchCV evaluates all combinations of these hyper-parameters using 5-fold cross-validation and accuracy as the scoring metric. Then model was trained on the training set, and best set of hyper-parameters is selected and printed.

Initialize New KNN model with best Hyper-parameters:

```
knn_final_model = KNeighborsClassifier(**knn_best_params)
knn_final_model.fit(X_train, y_train)

knn_pred_val = knn_final_model.predict(X_val)
print("KNN Accuracy:", accuracy_score(y_val, knn_pred_val))
```

Initialize a new KNN model with the best hyper-parameters that we found from the cross-validation. And find the models accuracy on validation set.

Retrain the KNN model and Evaluate it on Testset:

```
# retrain knn final model using train_val set and evaluate on test set
knn_final_model.fit(X_train_val, y_train_val)

y_test_pred_knn = knn_final_model.predict(X_test)

print("KNN: ")
print(f'Accuracy: {accuracy_score(y_test, y_test_pred_knn)}')
precision = precision_score(y_test, y_test_pred_knn)
print(f'Average Precision: {average_precision_score_macro(precision, y_test)}')
recall = recall_score(y_test, y_test_pred_knn)
print(f'Average Recall: {average_recall_score_macro(recall, y_test)}')
f_score = fscore(y_test, y_test_pred_knn)
print(f'Aveerage F1-Score: {fscore_average_macro(f_score, y_test)}')

display_confusion_matrix(y_test, y_test_pred_knn)
```

Retrain the new KNN model with the concatenated train, validation set. Then we Evaluate the model on test set and find and plot the models classification accuracy, precision score, recall score, f1-score, confusion matrix.

Optimizing Naive Bayesian Model with GridSearchCV:

```
# initialize the classifier models
nb_classifier = GaussianNB()

nb_param_grid = {
    'var_smoothing': [1e-9, 1e-8, 1e-7, 1e-6, 1e-5]
}

nb_grid = GridSearchCV(estimator=nb_classifier, param_grid=nb_param_grid, cv=5, scoring='accur

nb_grid.fit(X_train, y_train)

nb_best_params = nb_grid.best_params_
print("Best␣KNN␣Parameters:", nb_best_params)
```

We initializes a Naive Bayesian Classifier and perform hyper-parameter tuning using GridSearchCV to identify the best configuration model. This GridSearchCV evaluates all combinations of these hyper-parameters using 5-fold cross-validation and accuracy as the scoring metric. Then model was trained on the training set, and best set of hyper-parameters is selected and printed.

Initialize New Naive Bayesian model with best Hyper-parameters:

```
nb_final_model = GaussianNB(**nb_best_params)
nb_final_model.fit(X_train, y_train)

nb_pred_val = nb_final_model.predict(X_val)
print("Naive␣Bayesian␣Accuracy:", accuracy_score(y_val, nb_pred_val))
```

Initialize a new Naive Bayesian model with the best hyper-parameters that we found from the cross-validation. And find the models accuracy on validation set.

Retrain the Naive Bayesian model and Evaluate it on Testset:

```
# retrain naive bayesian final model using train_val set and evaluate on test set
nb_final_model.fit(X_train_val, y_train_val)

y_test_pred_nb = nb_final_model.predict(X_test)

print("Naive␣Bayesian:␣")
print(f'Accuracy:␣{accuracy_score(y_test,␣y_test_pred_nb)}')
precision = precision_score(y_test, y_test_pred_nb)
print(f'Average␣Precision:␣{average_precision_score_macro(precision,␣y_test)}')
recall = recall_score(y_test, y_test_pred_nb)
print(f'Average␣Recall:␣{average_recall_score_macro(recall,␣y_test)}')
f_score = fscore(y_test, y_test_pred_nb)
print(f'Aveerage␣F1-Score:␣{fscore_average_macro(f_score,␣y_test)}')

print()

display_confusion_matrix(y_test, y_test_pred_nb)
```

Retrain the new Naive Bayesian model with the concatenated train, validation set. Then we Evaluate the model on test set and find and plot the models classification accuracy, precision score, recall score, f1-score, confusion matrix.

Optimizing SVM Model with GridSearchCV:

```
# initialize the classifier models
svm_classifier = SVC()

```

```
svm_param_grid = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'gamma': ['scale', 'auto']
}

svm_grid = GridSearchCV(estimator=svm_classifier, param_grid=svm_param_grid, cv=5, scoring='ac
svm_grid.fit(X_train, y_train)

svm_best_params = svm_grid.best_params_
print("Best SVM Parameters:", svm_best_params)
```

Here, we do the same thing again. We initializes a SVM Classifier and perform hyper-parameter tuning using GridSearchCV to identify the best configuration model. This GridSearchCV evaluates all combinations of these hyper-parameters using 5-fold cross-validation and accuracy as the scoring metric. Then model was trained on the training set, and best set of hyper-parameters is selected and printed.

Initialize New SVM model with best Hyper-parameters:

```
svm_final_model = SVC(probability=True, **svm_best_params)
svm_final_model.fit(X_train, y_train)

svm_pred_val = svm_final_model.predict(X_val)
print("SVM validation set Accuracy:", accuracy_score(y_val, svm_pred_val))
```

Here also we do the same thing again. Initialize a new SVM model with the best hyper-parameters that we found from the cross-validation. And find the models accuracy on validation set.

Retrain the SVM model and Evaluate it on Testset:

```
# retrain svm final model using train_val set and evaluate on test set
svm_final_model.fit(X_train_val, y_train_val)

y_test_pred_svm = svm_final_model.predict(X_test)

print("SVM: ")
print(f'Accuracy: {accuracy_score(y_test, y_test_pred_svm)}')
precision = precision_score(y_test, y_test_pred_svm)
print(f'Average Precision: {average_precision_score_macro(precision, y_test)}')
recall = recall_score(y_test, y_test_pred_svm)
print(f'Average Recall: {average_recall_score_macro(recall, y_test)}')
f_score = fscore(y_test, y_test_pred_svm)
print(f'Aveerage F1-Score: {fscore_average_macro(f_score, y_test)}')

print()

display_confusion_matrix(y_test, y_test_pred_svm)
```

Here again Retrain the new SVM model with the concatenated train, validation set. Then we Evaluate the model on test set and find and plot the models classification accuracy, precision score, recall score, f1-score, confusion matrix.

2.3 Regression on 'DT-Wage' dataset

Loading Dataset:

```
df = pd.read_csv('/content/drive/MyDrive/CSE445-Assignment/dataset/DT-Wage.csv')
print(df.shape)
print(df.head())
```

Load the Dataset using pandas library and check first few data from the dataset and its Shape.

Checking Missing Values:

```
print('Number of missing values: ')
print(df.isnull().sum())
```

Checking if any missing values are founds in any features.

Encoded Categorical Data:

```
df = pd.get_dummies(df, columns=['maritl', 'race', 'education', 'region', 'jobclass', 'health'
df.columns = df.columns.str.replace('[', '_').str.replace(']', '_').str.replace('<', '_')
print(df.shape)
print(df.head())
```

Since there has few complex relationship between the categories, so we used `pd.get_dummies` to convert the categorical columns into one-hot encoded columns Then, we replace special characters ([,], ¡) in the column names with underscores. Then we print few samples from this encoded dataset.

Split Data into Training, Validation and Test Sets:

```
X = df.drop(columns=['wage'])
y = df['wage']

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

print(f'Train Data Shape (X, y): {X_train.shape, y_train.shape}')
print(f'Validation Data Shape (X, y): {X_val.shape, y_val.shape}')
print(f'Test Data Shape (X, y): {X_test.shape, y_test.shape}')
```

Splitting 70 percent data for training, 15 percent for validation and rest 15 percent for testing.

User Define Function to Calculate MSE:

```
# mse calculation
def mse(y_actual, y_pred):
    error = 0.0

    for actual, predicted in zip(y_actual, y_pred):
        error += (actual - predicted) ** 2
    mse = error / len(y_actual)
    return mse
```

This function will calculate the MSE of the model.

Initialize SVM Regressor Base Model:

```
# SVM base model
svm_base = SVR()
svm_base.fit(X_train, y_train)

y_val_pred = svm_base.predict(X_val)
print('Validation Set MSE:', mse(y_val, y_val_pred))
```

Initialize a SVM Regressor base model and train it using train dataset and then evaluate it on validation dataset and find the MSE.

Optimizing SVM Regressor Model with GridSearchCV:

```
param_grid = {
    ’C’: [0.1, 1, 10, 100],
    ’epsilon’: [0.01, 0.1, 0.2, 0.5],
    ’kernel’: [’linear’, ’rbf’, ’poly’]
}

svr = SVR()
svm_grid_search = GridSearchCV(estimator=svr, param_grid=param_grid, cv=5, scoring=’neg_mean_s
svm_grid_search.fit(X_train, y_train)

best_params = svm_grid_search.best_params_
print("Best␣Parameters:", best_params)
```

We initializes a SVM Regressor model and perform hyper-parameter tuning using GridSearchCV to identify the best configuration model. This GridSearchCV evaluates all combinations of these hyper-parameters using 5-fold cross-validation and neg_mean_squared_error as the scoring metric. Then model was trained on the training set, and best set of hyper-parameters is selected and printed.

Initialize the SVM Regressor Model with Best Hyper-parameters:

```
svm_final = SVR(**best_params)
svm_final.fit(X_train, y_train)

y_val_pred = svm_final.predict(X_val)
print(’Validation␣Set␣MSE:’, mse(y_val, y_val_pred))
```

Initialize a new SVM Regressor model with the best hyper-parameters that we found from the cross-validation. And calculate the MSE for validation set.

Retrain the SVM Regressor model and Evaluate it on Testset:

```
# now retrain the model on the combined training and validation set with the best hyperparamet
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

svm_final.fit(X_train_val, y_train_val)

y_test_pred = svm_final.predict(X_test)
print(’Test␣Set␣MSE:’, mse(y_test, y_test_pred))
```

Here, we retrain the new SVM Regressor model with the concatenated train, validation set. Then we again calculate the MSE for the testset.

2.4 Regression on ’DT-Credit’ dataset

Loading Dataset:

```
df = pd.read_csv(’/content/drive/MyDrive/CSE445-Assignment/dataset/DT-Credit.csv’)
print(df.shape)
print(df.head())
```

Load the Dataset using pandas library and check first few data from the dataset and its Shape.

Checking Missing Values:

```
print('Number of missing values: ')
print(df.isnull().sum())
```

Checking if any missing values are founds in any features.

Encoded Categorical Data:

```
df = pd.get_dummies(df, columns=['Own', 'Student', 'Married', 'Region'])
print(df.shape)
print(df.head())
```

Since there has few complex relationship between the categories, so we used `pd.get_dummies` to convert the categorical columns into one-hot encoded columns. Then we print few samples from this encoded dataset.

Split Data into Training, Validation and Test Sets:

```
X = df.drop(columns=['Balance'])
y = df['Balance']

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42

print(f'Train Data Shape (X, y): {X_train.shape, y_train.shape}')
print(f'Validation Data Shape (X, y): {X_val.shape, y_val.shape}')
print(f'Test Data Shape (X, y): {X_test.shape, y_test.shape}')
```

Splitting 70 percent data for training, 15 percent for validation and rest 15 percent for testing.

User Define Function to Calculate MSE:

```
# mse calculation
def mse(y_actual, y_pred):
    error = 0.0

    for actual, predicted in zip(y_actual, y_pred):
        error += (actual - predicted) ** 2
    mse = error / len(y_actual)
    return mse
```

This function will calculate the MSE of the model.

Initialize SVM Regressor Base Model:

```
# SVM base model
svm_base = SVR()
svm_base.fit(X_train, y_train)

y_val_pred = svm_base.predict(X_val)
print('Validation Set MSE:', mse(y_val, y_val_pred))
```

Initialize a SVM Regressor base model and train it using train dataset and then evaluate it on validation dataset and find the MSE.

Optimizing SVM Regressor Model with GridSearchCV:

```
param_grid = {
    'C': [0.1, 1, 10],
    'epsilon': [0.01, 0.1, 0.5],
    'kernel': ['linear', 'rbf', 'poly']
}

svr = SVR()
```

```
svm_grid_search = GridSearchCV(estimator=svr, param_grid=param_grid, cv=5, scoring='neg_mean_s
svm_grid_search.fit(X_train, y_train)

best_params = svm_grid_search.best_params_
print("Best␣Parameters:", best_params)
```

We initializes a SVM Regressor model and perform hyper-parameter tuning using GridSearchCV to identify the best configuration model. This GridSearchCV evaluates all combinations of these hyper-parameters using 5-fold cross-validation and neg_mean_squared_error as the scoring metric. Then model was trained on the training set, and best set of hyper-parameters is selected and printed.

Initialize the SVM Regressor Model with Best Hyper-parameters:

```
svm_final = SVR(**best_params)
svm_final.fit(X_train, y_train)

y_val_pred = svm_final.predict(X_val)
print('Validation␣Set␣MSE:', mse(y_val, y_val_pred))
```

Initialize a new SVM Regressor model with the best hyper-parameters that we found from the cross-validation. And calculate the MSE for validation set.

Retrain the SVM Regressor model and Evaluate it on Testset:

```
# now retrain the model on the combined training and validation set with the best hyperparamete
X_train_val = np.concatenate((X_train, X_val))
y_train_val = np.concatenate((y_train, y_val))

svm_final.fit(X_train_val, y_train_val)

y_test_pred = svm_final.predict(X_test)
print('Test␣Set␣MSE:', mse(y_test, y_test_pred))
```

Here, we retrain the new SVM Regressor model with the concatenated train, validation set. Then we again calculate the MSE for the testset.

3. Experiment results (Test Set)

3.1 Binary Classification on 'DT-BrainCancer' Dataset Results

Accuracy:

ZeroR	OneR	KNN	SVM	NB
59.77%	64.28%	78.57%	64.28%	92.85%

For Binary Classification dataset Naive Bayes Classifier model perform better then rest four model.

Precision:

ZeroR		OneR		KNN		SVM		NB	
Class 0	Class 1	CLass 0	Class 1	CLass 0	Class 1	CLass 0	Class 1	CLass 0	Class 1
-	-	-	-	0.80	0.75	0.64	0.00	1.00	0.83

Overall Naive Bayes has higher precision score compare to other models.

Recall:

ZeroR		OneR		KNN		SVM		NB	
Class 0	**Class 1**	**CLass 0**	**Class 1**	**CLass 0**	**Class 1**	**CLass 0**	**Class 1**	**CLass 0**	**Class 1**
-	-	-	-	0.88	0.60	1.00	0.00	0.88	1.00

Overall Naive Bayes has higher recall score compare to other models.

F-Score:

ZeroR		OneR		KNN		SVM		NB	
Class 0	**Class 1**	**CLass 0**	**Class 1**	**CLass 0**	**Class 1**	**CLass 0**	**Class 1**	**CLass 0**	**Class 1**
-	-	-	-	0.84	0.66	0.78	0.00	0.94	0.90

Overall Naive Bayes has higher f-score compare to other models.

Confusion Matrix and Precision-Recall Curve:

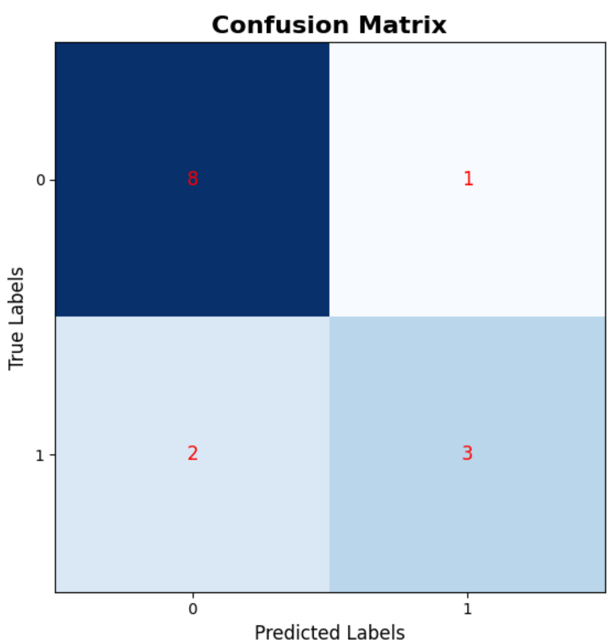

Fig. 1: KNN Models Confusion Matrix

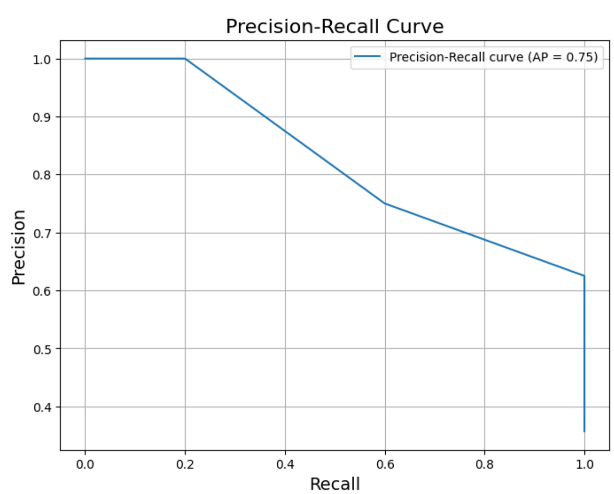

Fig. 2: KNN Models Precision-Recall Curve

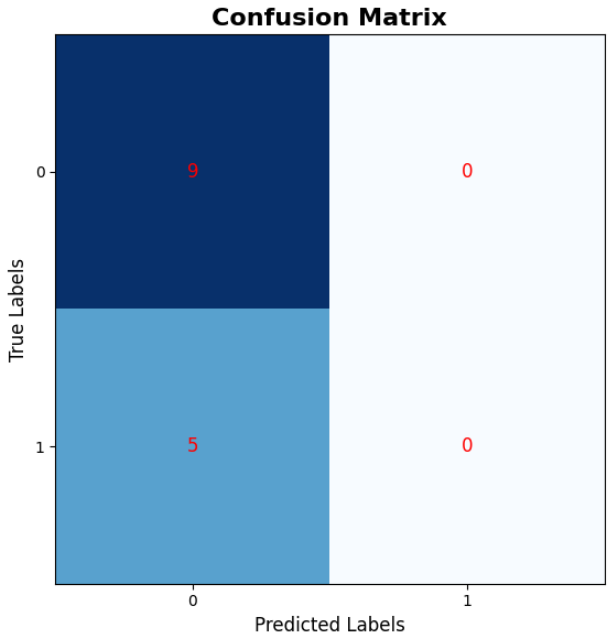

Fig. 3: SVM Models Confusion Matrix

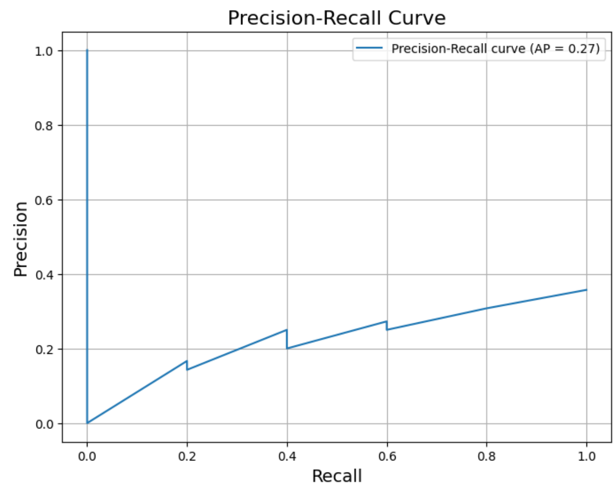

Fig. 4: SVM Models Precision-Recall Curve

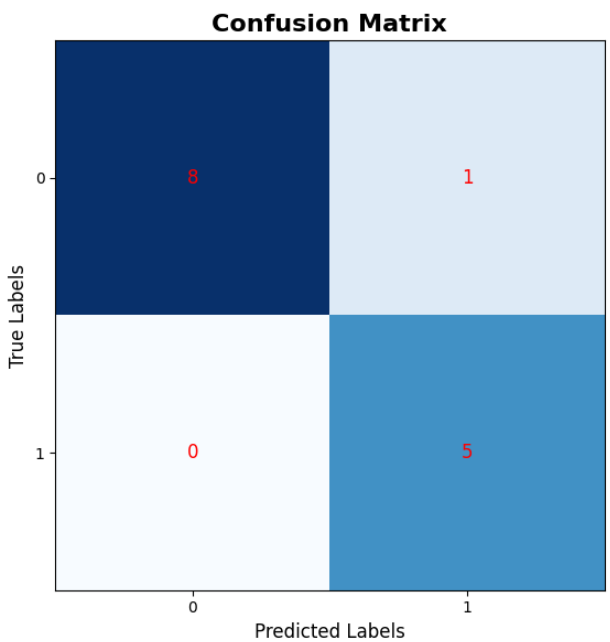

Fig. 5: Naive Bayes Models Confusion Matrix

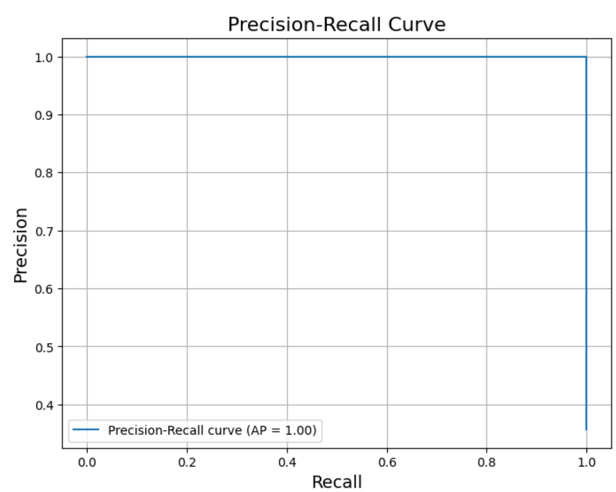

Fig. 6: Naive Bayes Precision-Recall Curve

3.2 Multi-class Classification on ’Car Evaluation’ dataset

Accuracy:

ZeroR	OneR	KNN	SVM	NB
70.01%	70.38%	93.07%	99.23%	80.38%

For Multi-class Classification dataset SVM Classifier model perform better then rest four model.

Average Precision:

ZeroR	OneR	KNN	SVM	NB
-	-	0.87	0.96	0.65

For Multi-class Classification dataset SVM model has higher Precision.

Average Recall:

ZeroR	OneR	KNN	SVM	NB
-	-	0.73	0.96	0.64

For Multi-class Classification dataset SVM model has higher Average Recall.

Average F-score:

ZeroR	OneR	KNN	SVM	NB
-	-	0.77	0.96	0.58

Here, SVM Model has higher F-score compare to other models.

Confusion Matrix:

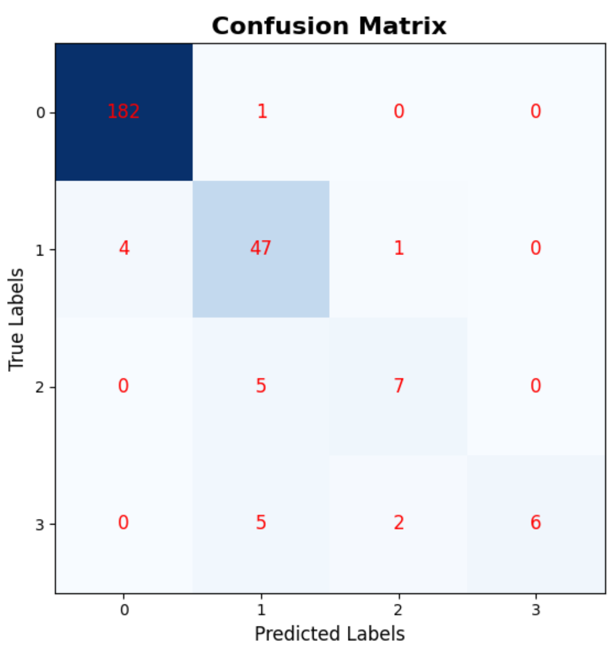

Fig. 7: KNN Confusion Matrix

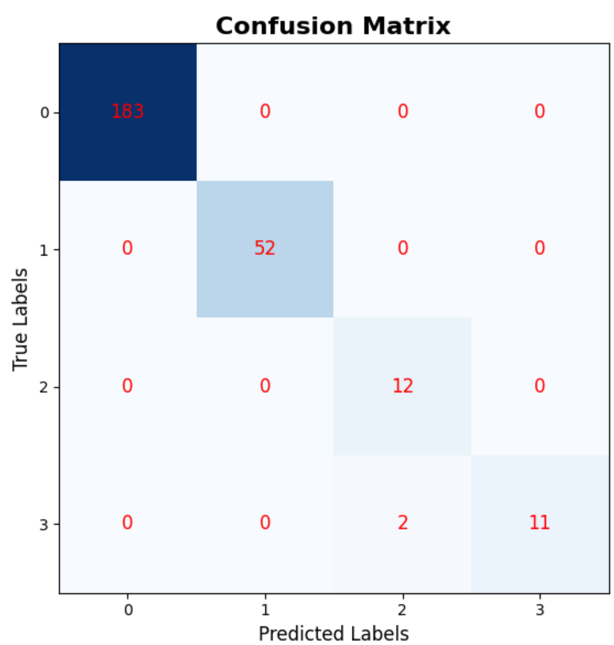

Fig. 8: SVM Confusion Matrix

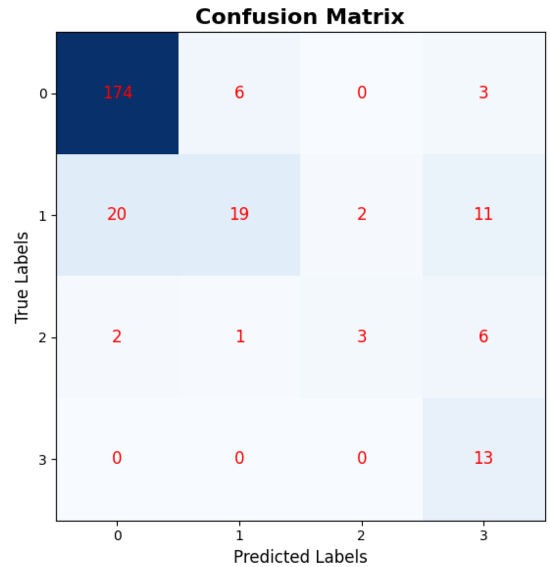

Fig. 9: Naive Bayes Confusion Matrix

3.3 Regression on 'DT-Wage' Dataset and 'DT-Credit' Dataset:

Mean Squared Error (Test Set):

DT-Wage	DT-Credit
358.61	13779.73

For the Regression Problem of two dataset, Dt-Wage datasets model has Lower MSE compare to Dt-Credit datasets model. For both model we apply same method.

4. Discussion

During this assignment, we learned more about some Classical Machine Learning algorithm which are OneR classifier, ZeroR classifier, KNN, Naive Bayes, SVM, etc. During implementation we first implement the base model using test set and check the models performance on validation set. Then we perform hyper-parameter tuning on test set and then check its performance using validation set. after hyper-parameter tuning the model was performing well better then the base model. Then we merge the training set and validation set and return our models with this dataset and lastly we check the models classification performance on test set. Now the model was performing more better. Here, for Regression task we check the MSE to check our models performance. During this whole assignment we follow multiple resources, whole links are given below on the References section.

References

[1] S. Joshi. Gridsearchcv implementation, 2024. https://www.kaggle.com/code/shrushtijoshi/gridsearchcv-implementation.
[2] mfreyeso. oner-scratch, 2024. https://github.com/mfreyeso/oner-scratch.
[3] prabhat kumar singh. Oner zeror, 2020. https://www.kaggle.com/code/prabhat12/oner-zeror.
[4] scikit-learn developers. Naive bayes — scikit-learn 1.3.0 documentation, 2024. https://scikit-learn.org/stable/modules/naive_bayes.html.
[5] scikit-learn developers. Nearest neighbors — scikit-learn 1.3.0 documentation, 2024. https://scikit-learn.org/stable/modules/neighbors.html.
[6] scikit-learn developers. Support vector machines — scikit-learn 1.3.0 documentation, 2024. https://scikit-learn.org/stable/modules/svm.html.