Assignment-5: ZeroR, OneR, SVM, KNN, NB

Rezwan-Ul-Alam (ID: 2011659042)
Md. Nur Alam Jowel (ID: 2012355042)
Raian Ruku (ID: 2013409642)

Email addresses:
rezwan.alam1@northsouth.edu
alam.jowel@northsouth.edu
raian.ruku@northsouth.edu

I. Introduction

In this assignment, we explore the application of various supervised learning methods to different datasets. The datasets used for classification tasks are "car.csv" and "braincancer.csv", while for regression tasks, we utilize "wage.csv" and "credit.csv". The supervised learning methods employed includes ZeroR classifier, OneR classifier, K-nearest-neighbors (KNN) classifier, Naïve Bayesian Classifier, Support Vector Machine (SVM), and Support Vector Regression (SVR).

II. Method

A. Data Preprocessing

Data preprocessing involves handling missing values, encoding categorical variables and scaling numerical features to ensure compatibility with these learning Algorithm.

*1) **Handling missing values**:* First of all we check which column has how many missing values.

```
[for column in df.columns:
    total = df [column].isnull().sum()
    print (f"Column ’{column}’ has {total} null values")
    print ("\n") ]
```

Here, this code iterates over each column in the DataFrame df, calculates the number of null values in each column, and prints this information along with the column name.

*2) **Encoding Categorical variables**:* We use both label encoder and one hot encoder for encoding categorical variables into numerical representations.

i. Label encoder: A label encoder is a preprocessing techniques commonly used in machine learning to convert categorical data into numerical format. When dealing with categorical data many machine learning algorithm requires numerical input. Label encoders converts these categorical values into numerical labels. In our case we use label encoding these three "car.csv, credit.csv and wage.csv"datasets.

Below, the code for label encoding of the 3 datasets is shown separately.

For "Car.csv" data set:

```
[data.classes.replace((’unacc’, ’acc’, ’good’, ’vgood’), (0, 1, 2, 3),
inplace = True)
data[’classes’].value_counts()]
```

Here, These lines of code use a Label Encoder to replace categorical values ('unacc', 'acc', 'good', 'vgood') in the 'classes' column with numerical labels (0, 1, 2, 3).

Similarly, the rest of the columns having categorical values are converted to numerical values. For example in "estimated safety" column we replace categorical values ('low', 'med', 'high') with numerical values (0,1,2).

Here is the code,

```
[data['estimated␣safety'].replace(('low', 'med', 'high'), (0, 1, 2),
inplace = True)
data['estimated␣safety'].value_counts()]
```

For "credit.csv" dataset: In this data set these four ('own', 'Student', 'Married', 'Region') columns have categorical values. So, we converted these categorical features into numerical features. Here is label encoding code.

```
df.Own.replace(('Yes', 'No'), (1, 0), inplace = True)
df.Student.replace(('Yes', 'No'), (1, 0), inplace = True)
df.Married.replace(('Yes', 'No'), (1, 0), inplace = True)
df.Region.replace(('South', 'West', 'East'), (0, 1, 2), inplace = True)
```

This code replaces categorical values in columns 'Own', 'Student', 'Married', and 'Region' with corresponding numerical labels (1 for 'Yes' and 0 for 'No' in the first three columns, and 0 for 'South', 1 for 'West', and 2 for 'East' in the 'Region' column).

For "wage.csv" dataset: In this data set we use imported labelencoder class from scikit learn libraries and initialize it. Here is the code,

```
from sklearn.preprocessing import LabelEncoder
label_enc=LabelEncoder()
df['maritl'] = label_enc.fit_transform(df['maritl'])
df['race'] = label_enc.fit_transform(df['race'])
df['education'] = label_enc.fit_transform(df['education'])
df['region'] = label_enc.fit_transform(df['region'])
df['jobclass'] = label_enc.fit_transform(df['jobclass'])
df['health'] = label_enc.fit_transform(df['health'])
df['health_ins'] = label_enc.fit_transform(df['health_ins'])
```

It applies the LabelEncoder to encode categorical variables ('maritl', 'race', 'education', 'region', 'jobclass', 'health', and 'health_ins') into numerical labels. Each categorical column is transformed independently, assigning a unique numerical label to each category.

B. Splitting Dataset

As per the instruction, the first 70% of the data was used for training, and 15% of the data was used for validation and last 15% of the data is used for testing. The code is given below:

```
X = data.iloc[:,:6]
y = data.iloc[:, 6]

print("Shape␣of␣x:␣", X.shape)
print("Shape␣of␣y:␣", y.shape)
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)
```

C. Scaling features

In our case, we use both **MinMaxScalar()** and **StandardScalar()** in our all four datasets for scaling features.

*1) **MinMaxScalar()** :* MinMaxScalar is used for scaling features to specified range, typically between 0 and 1. The code below introduces a min max scaling function to perform min-max scaling on numeric columns in our data frame. It calculates the scaled values, transforming X features to a common scale between 0 and 1.

Here below the code is,

```
from sklearn.preprocessing import MinMaxScaler
 scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

*2) **StandardScalar()** :* The StandardScaler in scikit-learn is used for standardizing features by removing the mean and scaling them to unit variance.

Here below the code is,

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

III. Model Training

A. Classification

For the classification tasks involving datasets **"Car_evaluation.csv"** and **"Braincancer.csv,"** the following supervised learning methods are used to trained:

NOTE regarding to the report: It's important to understand that hyperparameter tuning isn't applicable to the **ZeroR** classifier. Here's why:

- **ZeroR Classifier:** This classifier simply predicts the majority class for all instances. It doesn't involve any learning from the data and hence doesn't have any hyperparameters to tune.

  Since the ZeroR classifier always predicts the majority class, its performance is solely dependent on the class distribution in the training data. There are no parameters to adjust to improve its performance on unseen data. Similar to the ZeroR classifier, OneR (One Rule) doesn't involve hyperparameter tuning in the traditional sense. Here's why:

- **OneR Classifier:** This classifier assigns data points to the class of the attribute that has the minimum within-class variance. It focuses on finding a single rule to classify instances based on the attribute with the best separation between classes.

*1) **ZeroR Classifier**:* This classifier predicts the most frequent class label for all instances. It serves as a baseline for comparing the performance of other classifiers. Here is the code for ZeroR classifier learning algorithm,

```
from sklearn.dummy import DummyClassifier

zeror_classifier = DummyClassifier(strategy="most_frequent")
zeror_classifier.fit(X_train_scaled, y_train)

#on train
y_train_pred_zeror= zeror_classifier.predict(X_train_scaled)
train_accuracy_zeror = calculate_accuracy(y_train, y_train_pred_zeror)
print("ZeroR Classifier Accuracy on training Set:", train_accuracy_zeror)

#on validation
y_val_pred_zeror= zeror_classifier.predict(X_val_scaled)
val_accuracy_zeror = calculate_accuracy(y_val, y_val_pred_zeror)
print("ZeroR Classifier Accuracy on Validation Set:", val_accuracy_zeror)

#on test
y_test_pred_zeror = zeror_classifier.predict(X_test_scaled)
test_accuracy_zeror = calculate_accuracy(y_test, y_test_pred_zeror)
print("ZeroR Classifier Accuracy on Test Set:", test_accuracy_zeror)
```

*2) **OneR Classifier**:* OneR builds a simple rule-based model using a single feature. It selects the single feature with the best accuracy for predicting the target variable. Here is the code for OneR classifier learning algorithm,

```
from mlxtend.classifier import OneRClassifier
oner = OneRClassifier()
oner.fit(X_train_scaled, y_train);

y_train_pred_oner = oner.predict(X_train_scaled)
train_accuracy_oner = calculate_accuracy(y_train, y_train_pred_oner)
print("OneR Classifier Accuracy on Training Set:", train_accuracy_oner)

#on val
y_val_pred_oner = oner.predict(X_val_scaled)
val_accuracy_oner = calculate_accuracy(y_val, y_val_pred_oner)
print("OneR Classifier Accuracy on Validation Set:", val_accuracy_oner)

#on test
y_test_pred_oner = oner.predict(X_test_scaled)
test_accuracy_oner = calculate_accuracy(y_test, y_test_pred_oner)
print("OneR Classifier Accuracy on Test Set:", test_accuracy_oner)
```

*3) **K-Nearest-Neighbor (KNN) Classifiers**:* KNN predicts the class of an instance based on the majority class among its k-nearest neighbors. It is a non-parametric method that does not make any assumptions about the underlying data distribution. Here is the code for KNN classifier learning algorithm.

First of we want try to find out optimal hyperparameter such as (number of neighbours and weight scheme) that maximize the accuracy of the KNN classifier on validation data set.

Here is the code for finding optimal hyperparameter,

```
from sklearn.neighbors import KNeighborsClassifier
best_accuracy_knn = 0
best_neighbors = 0
best_weights = ''

neighbor_values = [1, 3, 5, 7, 9]
weight_values = ['uniform', 'distance']

for neighbors in neighbor_values:
    for weights in weight_values:
        knn_classifier = KNeighborsClassifier(n_neighbors=neighbors, weights=weights)
        knn_classifier.fit(X_train_scaled, y_train)
        y_val_pred_knn = knn_classifier.predict(X_val_scaled)
        accuracy_knn = calculate_accuracy(y_val, y_val_pred_knn)
        print(f"Neighbors={neighbors}, Weights={weights}, accuracy={accuracy_knn}")

        if accuracy_knn > best_accuracy_knn:
            best_accuracy_knn = accuracy_knn
            best_neighbors = neighbors
            best_weights = weights

print("Best number of neighbors for KNN:", best_neighbors)
print("Best weights for KNN:", best_weights)
print("Best validation accuracy for KNN:", best_accuracy_knn)
```

After finding optimal hyperparameter,we use these parameter in final training set. Here below the code,

```
test_accuracy_knn = calculate_accuracy(y_test, y_test_pred_knn)
print("KNN Test accuracy :", test_accuracy_knn)
```

*4) **Naive Bayesian Classifier**:* This classifier applies Bayes' theorem with the "naive" assumption of independence between features. It is particularly effective for text classification tasks and is known for its simplicity and efficiency. Here below the code is Naïve Bayesian Classifier learning algorithm.

First of all, we want to try to find optimal hyperparameters such as (var_smoothing) that maximize the accuracy of the Naive Bayesian classifier on validation data set.

Here is the code for finding optimal hyperparameter,

```
from sklearn.naive_bayes import GaussianNB
best_accuracy_nb = 0
best_var_smoothing = 0
var_smoothing_values = [1e-9, 1e-8, 1e-7, 1e-6]
for var_smoothing in var_smoothing_values:
    nb_classifier = GaussianNB(var_smoothing=var_smoothing)
    nb_classifier.fit(X_train_scaled, y_train)
    y_val_pred_nb = nb_classifier.predict(X_val_scaled)
    accuracy_nb = calculate_accuracy(y_val, y_val_pred_nb)
    print(f"Var_smoothing={var_smoothing}, accuracy={accuracy_nb}")
    if accuracy_nb > best_accuracy_nb:
        best_accuracy_nb = accuracy_nb
        best_var_smoothing = var_smoothing
print("Best variance_smoothing for Gaussian Naive Bayes:", best_var_smoothing)
print("Best validation accuracy for Gaussian Naive Bayes:", best_accuracy_nb)
```

After finding optimal hyperparameters,we use these parameters in the final training set. Here below the code,

```
final_nb_classifier = GaussianNB(var_smoothing=best_var_smoothing)
final_nb_classifier.fit(X_train_scaled, y_train)
y_test_pred_nb = final_nb_classifier.predict(X_test_scaled)
test_accuracy_nb = calculate_accuracy(y_test, y_test_pred_nb)
print("naive␣Bayes␣Test␣accuracy␣:␣", test_accuracy_nb)
```

5) ***Support Vector Machine (SVM)*** *:* SVM finds the hyperplane that best separates different classes in the feature space. It works well in high-dimensional spaces and is effective for both linear and non-linear classification tasks. Here below the code is SVM classifier learning algorithm.

First of all, we want to try to find optimal hyperparameters such as (C, Kernel, degree) that maximize the accuracy of the Support Vector Machine (SVM) classifier on validation data set. Here is the code for finding optimal hyperparameter,

```
from sklearn.svm import SVC
best_accuracy = 0
best_c = 0
best_kernel = ''
best_degree = 0

C_values = [0.1, 1, 10, 100]
kernel_values = ['linear', 'rbf', 'poly']
degree_values = [2, 3, 4]

for kernel in kernel_values:
    if kernel == 'poly':
        for degree in degree_values:
            for C in C_values:
                svm_classifier = SVC(C=C, kernel=kernel, degree=degree)
                svm_classifier.fit(X_train_scaled, y_train)
                y_val_pred_svm = svm_classifier.predict(X_val_scaled)
                accuracy = calculate_accuracy(y_val, y_val_pred_svm)
                print(f"Kernel={kernel},␣Degree={degree},␣C={C},␣accuracy={accuracy}")
                if accuracy > best_accuracy:
                    best_accuracy = accuracy
                    best_c = C
                    best_kernel = kernel
                    best_degree = degree
    else:
        for C in C_values:
            svm_classifier = SVC(C=C, kernel=kernel)
            svm_classifier.fit(X_train_scaled, y_train)
            y_val_pred_svm = svm_classifier.predict(X_val_scaled)
            accuracy = calculate_accuracy(y_val, y_val_pred_svm)
            print(f"Kernel={kernel},␣C={C},␣accuracy={accuracy}")
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_c = C
                best_kernel = kernel

print("Best␣Kernel␣for␣SVM:", best_kernel)
print("Best␣Degree␣for␣SVM:", best_degree) if best_kernel == 'poly' else None
print("Best␣C␣for␣SVM:", best_c)
print("Best␣validation␣accuracy␣for␣SVM:", best_accuracy)
rint("naive␣Bayes␣Test␣accuracy␣:␣", test_accuracy_nb)
```

After finding optimal hyperparameters,we use these parameters in the final training set. Here below the code,

```
final_svm_classifier = SVC(C=best_c, kernel=best_kernel, degree=best_degree)
final_svm_classifier.fit(X_train_scaled, y_train)
y_test_pred_svm = final_svm_classifier.predict(X_test_scaled)
test_accuracy_svm = calculate_accuracy(y_test, y_test_pred_svm)
print("SVM Test accuracy : ", test_accuracy_svm)
```

*B. **Regression***

For regression tasks the involving datasets ‘wage.csv’ and ‘credit.csv’, the following supervised learning algorithm ‘Support Vector Regression’ (SVR) is used to train.

*1) **Support Vector Machine (SVM)** :* SVR predicts continuous values by finding the hyperplane that has the maximum margin to the nearest data points. It is effective for handling non-linear relationships between features and the target variable. Here below the code is SVR regression learning algorithm.

First, we want to try to find optimal hyperparameters such as (C, Kernel, degree) that maximize the Support Vector Regression (SVR) accuracy on validation data set. Here is the code for finding the optimal hyperparameter,

```
from sklearn.svm import SVR
best_mse = float('inf')
best_c = 0
best_kernel = ''
best_degree = 0
C_values = [0.1, 1, 10, 100]
kernel_values = ['linear', 'rbf', 'poly']
degree_values = [2, 3, 4]
for kernel in kernel_values:
    if kernel == 'poly':
        for degree in degree_values:
            for C in C_values:
                svr_regressor = SVR(C=C, kernel=kernel, degree=degree)
                svr_regressor.fit(X_train_scaled, y_train)
                y_val_pred_svr = svr_regressor.predict(X_val_scaled)
                mse = calculate_mse(y_val, y_val_pred_svr)
                print(f"Kernel={kernel}, Degree={degree}, C={C}, MSE={mse}")
                if mse < best_mse:
                    best_mse = mse
                    best_c = C
                    best_kernel = kernel
                    best_degree = degree
    else:
        for C in C_values:
            svr_regressor = SVR(C=C, kernel=kernel)
            svr_regressor.fit(X_train_scaled, y_train)
            y_val_pred_svr = svr_regressor.predict(X_val_scaled)
            mse = calculate_mse(y_val, y_val_pred_svr)
            print(f"Kernel={kernel}, C={C}, MSE={mse}")
            if mse < best_mse:
                best_mse = mse
                best_c = C
                best_kernel = kernel
print("Best Kernel for SVR:", best_kernel)
print("Best Degree for SVR:", best_degree) if best_kernel == 'poly' else None
print("Best C for SVR:", best_c)
print("Best validation MSE for SVR:", best_mse)
```

After finding optimal hyperparameters,we use these parameters in the final training set. Here below the code,

```
best_svr_model = SVR(C=best_c, kernel=best_kernel)
if best_kernel == 'poly':
    best_svr_model.degree = best_degree
best_svr_model.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_test_pred_svr = best_svr_model.predict(X_test_scaled)

# Calculate MSE on the test set
test_mse = calculate_mse(y_test, y_test_pred_svr)
print("Test MSE for SVR:", test_mse)
```

IV. Evaluation

A. *For Classification (car_evaluation.csv, Braincancer.csv):*

For evaluation in classification tasks we calculate accuracy, precision, recall, F1-score and confusion matrix.

- **Accuracy:** accuracy is the ratio of the number of correctly classified instances (true positives and true negatives) to the total number of instances in the dataset.
- **Precision:**Precision measures the proportion of true positive predictions among all positive predictions made by the classifier. It answers the question: "Out of all the instances predicted as positive, how many are actually positive?"
- **Recall:** Recall measures the proportion of true positive predictions among all actual positive instances in the dataset. It answers the question: "Out of all the actual positive instances, how many were correctly predicted as positive?"
- **F1-score:** The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall. F1-score reaches its best value at 1 and worst at 0.
- **Confusion matrix** A confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. It presents the count of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions made by the classifier.

Calculation process:: *here for calculating accuracy, precision, recall, F1-score and confusion matrix we use our own created function. In below we provide all the created function with necessary equation.*

NOTE here,

TP (True positive): Number of correctly predicted positive instances.
TN (True negative): Number of correctly predicted negative instances.
FP (False positive): Number of incorrectly predicted positive instances.
FN (False negative): Number of incorrectly predicted negative instances.

1) ***Precision [$TP/(TP+FP)$] :*** Here, this function calculate_precision(y_true, y_pred, label):takes the true labels, predicted labels, and a specific label for which precision is calculated. It then computes and returns the precision value for that label. If there are no predicted positives, it returns 0 to avoid division by zero.

```
def calculate_precision(y_true, y_pred, label):
    true_positives = sum((y_true == label) & (y_pred == label))
    predicted_positives = sum(y_pred == label)
    if predicted_positives > 0:
      precision = true_positives / predicted_positives
    else :0
    return precision
```

*2) **Recall** [$TP/(TP + FN)$]:* Here, The function calculate_recall(y_true, y_pred, label): essentially computes the recall metric for a given label based on the true and predicted labels provided. It ensures consistence by handling the scenario where there are no actual positive instances for the specified label.

```
def calculate_recall(y_true, y_pred, label):
    true_positives = sum((y_true == label) & (y_pred == label))
    actual_positives = sum(y_true == label)
    if actual_positives > 0:
      recall = true_positives / actual_positives
    else :0
    return recall
```

*3) **F1-Score** [$2 * (precision * recall)/(precicion + recall)$] :* Here, the function calculate_f1_score(precision, recall): essentially computes the F1-score metric based on the provided precision and recall values. It ensures consistence by handling the scenario where both precision and recall are zero to avoid division by zero.

```
def calculate_f1_score(precision, recall):
    if (precision + recall) > 0 :
      f1_score = 2 * (precision * recall) / (precision + recall)
    else :0
    return f1_score
```

*4) **Accuracy** [$(TP + TN)/(TP + TN + FP + FN)$] :* Here the function calculate_accuracy(y_true, y_pred) essentially computes the accuracy metric based on the provided true labels and predicted labels. It counts the number of correct predictions and divides it by the total number of predictions to obtain the accuracy value.

```
  def calculate_accuracy(y_true, y_pred):
.      correct_predictions = sum(y_true == y_pred)
.      total_predictions = len(y_true)
.      accuracy = correct_predictions / total_predictions
.      return accuracy
```

*5) **Confusion matrix*** $\begin{bmatrix} TN & FN \\ FP & TP \end{bmatrix}$ *:* Here the function prints the confusion matrix to the console. It first prints a header indicating that the confusion matrix is for a decision tree model. Then, it iterates over each row in the confusion matrix (conf_matrix) and prints each row. The confusion matrix is typically a 2x2 matrix where rows represent the actual classes and columns represent the predicted classes.

```
conf_matrix = calculate_confusion_matrix(y_test, y_test_pred)
         print("Decision␣tree␣Confusion␣Matrix:")
         for row in conf_matrix:
         print(row)
```

B. For Regression (credit.csv, wage.csv):

For evaluation in regression task we calculate mean squared error (mse).

- **Mean Squared Error (MSE):** MSE is a measure used to evaluate the average squared difference between the actual and predicted values in a regression problem. It quantifies the average squared deviation of predictions from the actual values, providing a measure of the model's predictive accuracy. Here is the calculation process of calculating MSE, **MSE**

  equation,

MSE= $1/n \sum_{i=0}^{n}(y - y^i)^2$

Here is the code for implementing MSE:

```
import numpy as np
def calculate_mse(y_true, y_pred):
      mse = np.mean((y_true - y_pred) ** 2)
      return mse
```

V. Results

In this section we will show the results of models trained with different learning algorithms such as for classification oneR, zeroR, KNN, Naïve Bayesian and SVM and for regression SVR.

*A. **Results For Classification Tasks***

Data set	Model (classifier)	Accuracy (Test set)	Precision	Recall	F1-Score	Confusion matrix
Car.csv	ZeroR	0.69230	0.173	0.250	0.205	[180, 0, 0, 0]
	OneR	0.69230	0.173	0.250	0.205	[180, 0, 0, 0]
	KNN	0.96153	0.928	0.907	0.909	[176, 4, 0, 0]
	Naive Bayesian	0.78076	0.562	0.657	0.557	[168, 10, 1, 1]
	SVM	0.98076	0.945	0.961	0.950	[179, 1, 0, 0]
Braincancer.csv	ZeroR	0.64285	0.642	1.0	0.782	[,]
	OneR	0.65285	0.642	1.0	0.782	[,]
	KNN	0.78571	0.875	0.67	0.78	[7.2,]
	Naive Bayesian	0.92857	1.0	0.83	0.89	[8, 1]
	SVM	0.92857	1.0	0.83	0.88	[8, 1]

Here, Looking at this table, we can observe that, for

*1) **Car.csv Dataset:**:*

- KNN and SVM classifiers achieved the highest accuracy of 96.15% and 98.08% respectively, outperforming ZeroR, OneR, and Naive Bayesian classifiers.
- KNN and SVM also show high precision, recall, and F1-score values, that's indicates good performance across all metrics.

*2) **Braincancer.csv Dataset**:*

- Naive Bayesian, and SVM classifiers achieved the highest accuracy of 92.85% for the binary classification task, outperforming ZeroR and OneR classifiers.
- Naive Bayesian and SVM classifiers exhibit perfect precision for class 0, indicating they correctly classified all instances of class 0.

B. *Results for Regression Tasks*

Data set	Model	Validation MSE	Test MSE
'credit.csv'	Support Vector Regression (SVR)	10945.229594997983	12033.412463087958
'wage.csv'	Support Vector Regression (SVR)	1.26394695472098	0.8096585861480677

Here, Looking at this table, we can observe that, for

*1) **'credit.csv' Dataset**:*

- The Support Vector Regression (SVR) model achieved a validation Mean Squared Error (MSE) of approximately 10945.23 and a test MSE of approximately 12033.41.
- The test MSE is slightly higher than the validation MSE, this indicates that the model may have slightly overfit the training data.

*2) **'wage.csv' Dataset:**:*

- The SVR model achieved a validation MSE of approximately 1.26 and a test MSE of approximately 0.81.
- Here, the test MSE is slightly lower than the validation MSE, this indicates that the model's performance on unseen data is slightly better than on the validation set.

C. *Graph (precision Recall curve) for Binary classification Task (BrainCancer.csv):*

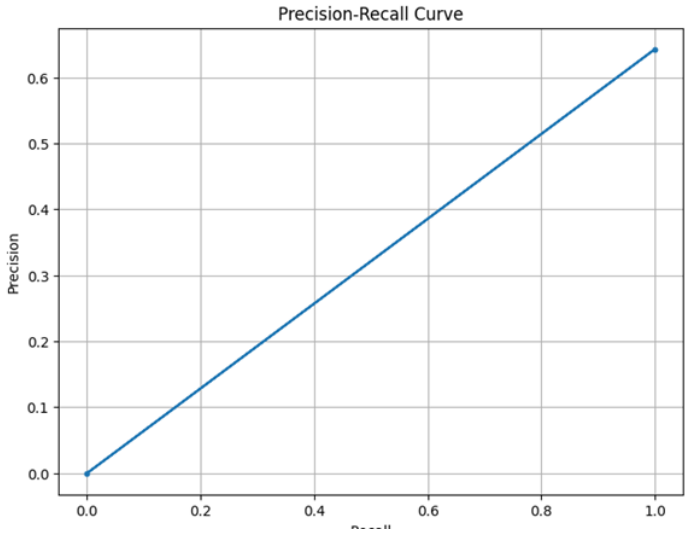


Fig. 1: ZeroR Classifier

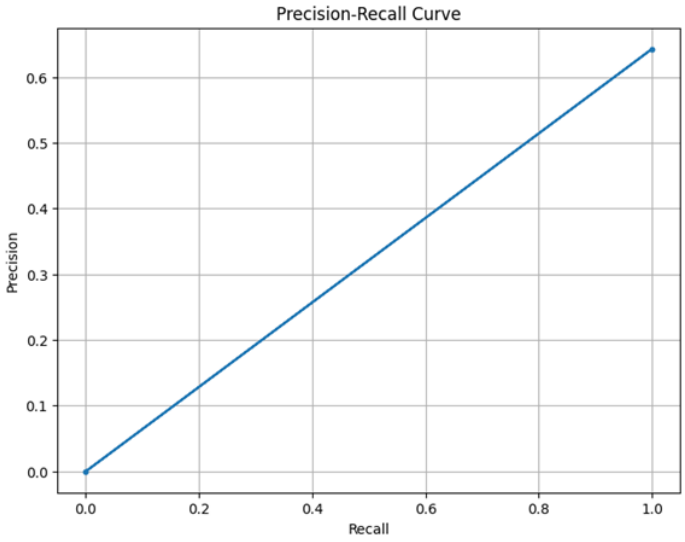


Fig. 2: OneR Classifier

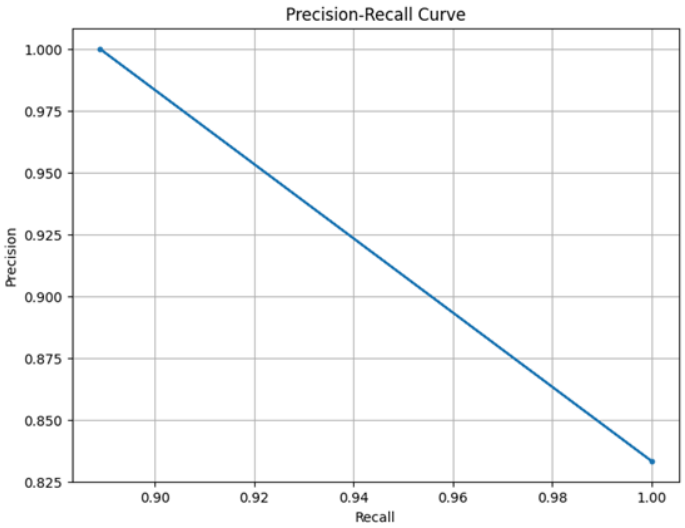
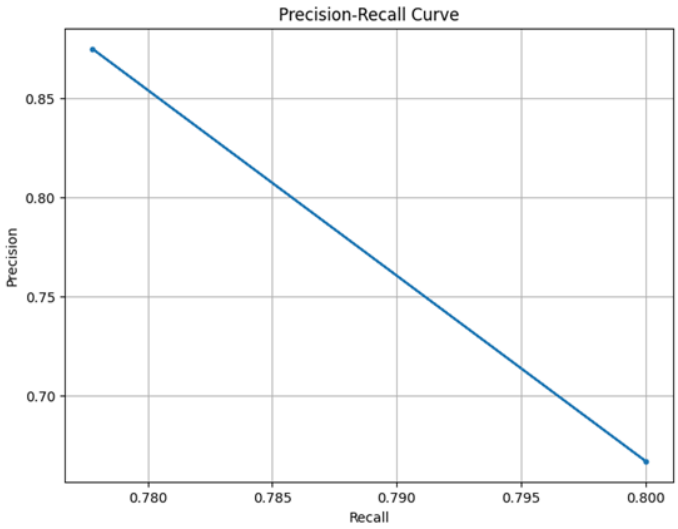


Fig. 3: SVM Classifier



[H]

Fig. 4: KNN Classifier

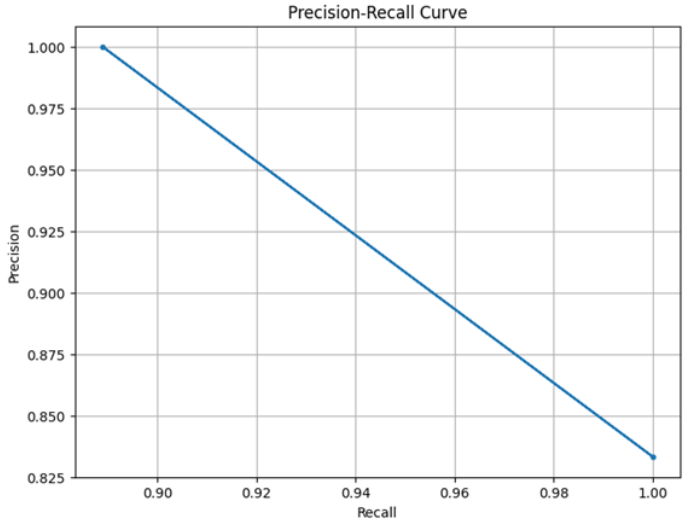


Fig. 5: Naïve Bayes Classifier

VI. Discussion

The evaluation results provide valuable insights into the performance of various supervised learning methods applied to both classification and regression tasks.

A. *Classification Tasks:*

- For the ”Car.csv” dataset, K-Nearest Neighbors (KNN) and Support Vector Machine (SVM) classifiers emerged as the top performers, achieving the highest accuracy of 96.15% and 98.08% respectively. These models also demonstrated high precision, recall, and F1-score values, indicating their robustness in correctly classifying instances across all metrics.

- Similarly, for the ”Braincancer.csv” dataset, Naive Bayesian and SVM classifiers outperformed ZeroR and OneR classifiers, achieving the highest accuracy of 92.85% for the binary classification task. Notably, Naive Bayesian and SVM classifiers exhibited perfect precision for class 0, showcasing their ability to accurately classify all instances of class 0.

B. *Regression Tasks:*

- In the ”credit.csv” dataset, the Support Vector Regression (SVR) model achieved a validation Mean Squared Error (MSE) of approximately 10945.23 and a test MSE of approximately 12033.41. Although the test MSE is slightly higher than the validation MSE, indicating potential overfitting, the model still provides valuable insights into predicting the target variable.

- Conversely, in the ”wage.csv” dataset, the SVR model achieved a validation MSE of approximately 1.26 and a test MSE of approximately 0.81. The lower test MSE compared to the validation MSE suggests that the model’s performance on unseen data is slightly better.

  Finally, these results highlight the effectiveness of KNN and SVM classifiers in handling complex classification tasks, while Naive Bayesian classifiers also demonstrate strong performance.