Appendix

1. Detail information about WBC data set

Wisconsin Breast Cancer (WBC) (Original) dataset from the University of California at Irvine (UCI) Machine Learning repository is used to test the proposed model ([34]). Dr. William H. Wolberg of the University of Wisconsin Hospitals in Madison provided this breast cancer database. He evaluated breast tumor biopsies for 699 patients up to July 15, 1992; each of nine features (attributes) was scored on a scale of 1 to 10. There are 699 rows and 11 columns in all. Among the 699 samples in the WBC data set, 458 were benign and 241 were malignant. The WBC data set has nine features and contains 699 samples. The patient's ID and class properties are included in all of these aspects. Nine characteristics (attributes) of WBCD are shown in Table 8.

Table 8: Wisconsin Breast Cancer (WBC) Dataset.

No.	Attributes	Range
1	Clump Thickness (CT)	1-10
2	Uniformity of cell size (UCS)	1-10
3	Uniformity of cell shape (UCSH)	1-10
4	Marginal Adhesion (MA)	1-10
5	Single Epithelial cell size (SEC)	1-10
6	Bare Nuclei (BN)	1-10
7	Bland Chromatin (BC)	1-10
8	Normal Nucleoli (NN)	1-10
9	Mitoses (Mit)	1-10
	Class	2 for Benign 4 for Malignant

Description of data is given as follows [127] [133]: In terms of CT, benign cells tend to be clustered in monolayers, but malignant cells are frequently grouped in multilayers. In the UCS and UCSH, the size and form of cancer cells might vary. Because of this, these factors are useful in determining whether or not the cells are malignant. In MA, Normal cells have a proclivity to adhere to one another. This capacity is often lost in cancer cells. As a result, adhesion loss is an indication of cancer. SEC has

something to do with the previously mentioned homogeneity. Significantly expanded epithelial cells may be cancerous. Nuclei that are not surrounded by cytoplasm are referred to as BNs (the rest of the cell). These are most commonly found in benign tumours. BC describes the homogeneous "texture" of the nucleus seen in benign cells. The chromatin in cancer cells is coarser. Nucleoli are tiny structures that can be seen within the nucleus. The nucleolus is normally relatively tiny, if present at all, in normal cells. The nucleoli grow more visible, and there are sometimes more of them, in cancer cells.

2. Detail information about WDBC data set

WDBC data set was introduced in November 1995. This data set contains 569 samples of patients out of which 357 benign and 212 malignant cases. This data set contains 32 features including class distribution and patient's ID. The class attribute '4' is used for malignant (M) and '2' is used for benign (B). Measurements of features of this data set are obtained with the help of digitised image of Fine Needle Aspirate (FNA) of a breast mass. These features describes the characteristic of the cell nuclie. The features of this data set are radius (mean of distances from the centre to points on the parameter), texture (standard deviation of grey-scale values), perimeter, area, smoothness (local variation in radius length), compactness (perimeter2/area-1), concavity (severity of concave portions of the contour), concave points (number of concave portions of the contour), symmetry and fractal dimension ('coastline approximation'-1) [34].

3. Python code for diagnosis of breast cancer using Support Vector machines

1. **Python code for WDBC data set**

```
#Cancer Classification Linear kernel + without PCA + random
    split train-test data + error + accuracy

import time

tic = time.time()

import pandas as pd
import numpy as np
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
```

```
from sklearn.metrics import classification_report

f = pd.ExcelFile('dataset.xlsx')
data = f.parse('Sheet2')
d = np.array(data)

X = np.array(d[0:684,1:10])
y = np.array(d[0:684,10])

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.30, random_state=42)

clf = svm.SVC(C=1.0, cache_size=200, class_weight=None, coef0
    =0.0,
     decision_function_shape='ovr', degree=3, gamma=10, kernel='
    linear',
     max_iter=-1, probability=False, random_state=None,
    shrinking=True,
     tol=0.001, verbose=False)

clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
print("Train Accuracy :: ", accuracy_score(y_train,
    y_pred_train))
print("Test Accuracy :: ", accuracy_score(y_test, y_pred_test))

from sklearn.metrics import f1_score
f1_score=f1_score(y_test, y_pred,average=None)
print('f1_score',f1_score)


CM = confusion_matrix(y_test, y_pred)
print('Confusion matrix for is \n',CM)

accuracy = accuracy_score(y_test, y_pred)
print('accuracy is',accuracy*100, '%')

TP = CM[0][0]
FP = CM[0][1]
FN = CM[1][0]
TN = CM[1][1]

from sklearn.metrics import mean_squared_error
```

```
MSE = mean_squared_error(y_test, y_pred)
print('MSE is:- ', MSE)
'''
recall = TP/(TP+FN)
print('recall is ',recall)
precision = TP/(TP+FP)
print('precision is',precision)
f1 = 2*(precision*recall)/(precision + recall)
print('f1 is',f1)
'''
error = (FP + FN)/(TP + FP + FN + TN)
print('error is:-\n',error)

result = ['benigne', 'malignant',]
print(classification_report(y_test, y_pred, target_names =
    result))


toc = time.time()

print('total time elapsed for test dataset',toc-tic,'second')

print('w = ',clf.coef_)
print('b = ',clf.intercept_)
print('Indices of support vectors = ', clf.support_)
print('Support vectors = ', clf.support_vectors_)
print('Number of support vectors for each class = ', clf.
    n_support_)
print('Coefficients of the support vector in the decision
    function = ', np.abs(clf.dual_coef_))
```

Listing 1: SVM without PCA & without k-Fold for WBC and WDBC data set

```
#SVM with PCA & with k-Fold- Linear
import time

import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

from sklearn import svm
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
```

```
from sklearn.model_selection import KFold

import seaborn as sns
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import cross_val_score

#extract dataset from Excel to python
f = pd.ExcelFile('dataset.xlsx')
data = f.parse('Sheet2')
d = np.array(data)

#define a matrix
x = np.array(d[0:684,1:10],dtype = 'float64')
y = np.array(d[0:684,10],dtype = 'float64')

print('matrix of input x is:',x)
print('matrix of output y is:',y)

#plot number of Malignant and Bengin cancer
ax = sns.countplot(y, label="Count", palette="muted")
B, M = y.value_counts()
plt.savefig('count.png')
print('Number of benign cancer: ', B)
print('Number of malignant cancer: ', M)

#Applying PCA
#Standardizing the features
X = StandardScaler().fit_transform(x)
print('Standardizing the features X:-\n',X)

pca = PCA(n_components = 2)
principalComponents = pca.fit_transform(X)
#principalDf = pd.DataFrame(data = principalComponents, columns
    = ['principal component 1', 'principal component 2'])
print('principalComponents are:-\n',principalComponents)
#print('principalDf is :-\n',principalDf)


#finalDf = pd.concat([principalDf, d[['y']]], axis = 1)
#print('finalDf is :-\n',finalDf)
'''
plt.clf
for i in range(y.shape[0]):
```

```
58      if y[i] == 2.0:
59          plt.scatter(principalComponents[i][0],
    principalComponents[i][1],marker='.',color='red')
60      if y[i] == 4.0:
61          plt.scatter(principalComponents[i][0],
    principalComponents[i][1],marker='o',color='green')
62 plt.xlabel('Principal component 1')
63 plt.ylabel('Principal component 2')
64 plt.title('Data reduction using PCA, Malignant: Red dots,
    Benign: Green Circle')
65 '''
66
67 tic = time.time()
68 #Applying SVM on new dataset
69 X_new = principalComponents
70 y = np.array(d[0:684,10],dtype = 'float64')
71
72
73 #applying k-fold CV and split dataset into traing-testing
74 kf = KFold(n_splits=10)
75 kf.get_n_splits(X_new)
76
77 print(kf)
78
79 for train_index, test_index in kf.split(X_new):
80    print("TRAIN:", train_index, "TEST:", test_index)
81    X_train, X_test = X_new[train_index], X_new[test_index]
82    y_train, y_test = y[train_index], y[test_index]
83
84 clf = svm.SVC(C=1, cache_size=200, class_weight=None, coef0
    =0.0,
85      decision_function_shape='ovr', degree=3, gamma='auto',
    kernel='linear',
86      max_iter=-1, probability=False, random_state=None,
    shrinking=True,
87      tol=0.001, verbose=False)
88
89 clf.fit(X_train,y_train)
90 y_pred = clf.predict(X_test)
91 y_pred_test = clf.predict(X_test)
92 y_pred_train = clf.predict(X_train)
93
94
95
96 CM = confusion_matrix(y_test, y_pred)
```

```
print('Confusion matrix for is \n',CM)

TP = CM[0][0]
FP = CM[0][1]
FN = CM[1][0]
TN = CM[1][1]

accuracy = accuracy_score(y_test, y_pred)
print('accuracy is',accuracy*100, '%')


# plot confusion matrix
y_pred = clf.predict(X_test)
CM = confusion_matrix(y_test, y_pred)
df_CM = pd.DataFrame(CM, range(2), range(2), columns=['4','2'])
plt.figure(figsize=(7,5))
sns.set(font_scale=1.4)#for label size
cm_plot = sns.heatmap(df_CM, annot=True, fmt='n', annot_kws={"
    size": 15})


error = (FP + FN)/(TP + FP + FN + TN)
print('error is:-\n',error)

result = ['benigne', 'malignant',]
print(classification_report(y_test, y_pred, target_names =
    result))


toc = time.time()

print('total time elapsed for test dataset',toc-tic,'second')
print("Train Accuracy :: ", accuracy_score(y_train,
    y_pred_train))
print("Test Accuracy :: ", accuracy_score(y_test, y_pred_test))
```

Listing 2: SVM with PCA & with k-Fold for WBC and WDBC data set

3. Python code for diagnosis of breast cancer using Deep Neural Network

1. Python code for WDBC and WBC data set

```
# Load data
import pandas as pd
```

```
import numpy as np
import time
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report
from sklearn import preprocessing
from sklearn.metrics import roc_auc_score, auc, roc_curve,
    precision_score, recall_score, f1_score

import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import label_binarize
import sklearn.metrics as metrics

tic = time.time()
data = pd.read_excel('WDBC.xlsx')

# Head method show first 5 rows of data
#print(data.head())

# Drop unused columns
columns = ['ID', 'diagnosis']

 # Convert strings -> integers
d = {'M': 0, 'B': 1}

# Define features and labels
y = data['diagnosis'].map(d)
X = data.drop(columns, axis=1).values

#print(y)
#print(X)


#split training-testing datset randomly
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.30, random_state=42)


net = MLPClassifier(hidden_layer_sizes=(25,10),
                    activation='logistic', solver='sgd', alpha
```

```
    =0.01 ,
46                      batch_size='auto', learning_rate='constant'
    , learning_rate_init=0.0001,
47                      power_t=0.5, max_iter=50000, shuffle=True,
    random_state=1, tol=0.0001,
48                      verbose=False, warm_start=False, momentum
    =0.9, nesterovs_momentum=True,
49                      early_stopping=False, validation_fraction
    =0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
50                      n_iter_no_change=10)
51
52 net.fit(X_train,y_train)
53 w = net.coefs_
54 z = net.intercepts_
55 predict = net.predict(X_test)
56 CM = confusion_matrix(y_test,predict)
57 #print('Confusion matrix for is \n',CM)
58 accuracy = accuracy_score(y_test, predict)
59 print('accuracy is :- \n',accuracy*100, '%')
60 print('netwrok trained after',net.n_iter_,'iterations')
61 print('total loss after training is',net.loss_)
62 toc = time.time()
63 print('total time elapsed for test dataset',toc-tic,'second')
```

Listing 3: DNN without ICA and without k-fold CV for WDBC and WBC data set

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Aug 29 01:48:28 2020
4
5 @author: Dell
6 """
7
8
9 # Load data
10 import pandas as pd
11 import numpy as np
12 import time
13 from sklearn.model_selection import train_test_split
14 from sklearn.metrics import confusion_matrix
15 from sklearn.metrics import accuracy_score
16 from sklearn.neural_network import MLPClassifier
17 from sklearn.metrics import classification_report
18 from sklearn.model_selection import KFold
19 from sklearn import preprocessing
20 from sklearn.decomposition import FastICA
```

```
from sklearn.metrics import roc_auc_score, auc, roc_curve,
    precision_score, recall_score, f1_score

import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import label_binarize
import sklearn.metrics as metrics

data = pd.read_excel('WDBC.xlsx')

# Head method show first 5 rows of data
#print(data.head())

# Drop unused columns
columns = ['ID', 'diagnosis']

 # Convert strings -> integers
d = {'M': 0, 'B': 1}

# Define features and labels
y = data['diagnosis'].map(d)
X = data.drop(columns, axis=1)

tic = time.time()

from sklearn.preprocessing import StandardScaler

#applying ICA
transformer = FastICA(n_components=3,
        random_state=0)
X1 = transformer.fit_transform(X)
X1.shape

#traing-testing by k-fold and SVM
kf = KFold(n_splits=10)
kf.get_n_splits(X1)

for train_index, test_index in kf.split(X1):
    #print("TRAIN:", train_index,"Validation:", "TEST:",
    test_index)
    X1_train, X1_test = X1[train_index], X1[test_index]
    y_train, y_test = y[train_index], y[test_index]

```

```
64
65 scaler = StandardScaler()
66 X1_train = scaler.fit_transform(X1_train)
67 X1_test = scaler.transform(X1_test)
68
69 net = MLPClassifier(hidden_layer_sizes=(25,10),
70                     activation='relu', solver='adam',
71                     alpha=0.5, batch_size='auto',
72                     learning_rate='constant',
   learning_rate_init=0.0001,
73                     power_t=0.5, max_iter=100000, shuffle=True,
    random_state=None,
74                     tol=0.0001, verbose=True, warm_start=True,
   momentum=0.9,
75                     nesterovs_momentum=True, early_stopping=
   False, validation_fraction=0.2,
76                     beta_1=0.9, beta_2=0.999, epsilon=1e-08,
   n_iter_no_change=10,max_fun=15000)
77
78
79 net.fit(X1_train,y_train)
80 predict = net.predict(X1_test)
81 toc = time.time()
82 CM = confusion_matrix(y_test,predict)
83 print('Confusion matrix for is \n',CM)
84 accuracy = accuracy_score(y_test, predict)
85 print('accuracy is :- \n',accuracy*100, '%')
86 result = ['benigne', 'malignant',]
87 print(classification_report(y_test, predict, target_names =
   result))
88 print('total time elapsed for test dataset',toc-tic,'second')
89 w = net.coefs_
90 print('weights are :',w)
91 z = net.intercepts_
92 print('bias values are :',z)
93 print('X1_test :', X1_test)
94 print('y_test (Actual output) :', y_test)
95 print('Network output :', predict)
96 print('Normalized X1 :', X1)
97 print('ICA components : ', X1)
98 print('accuracy is :- \n',accuracy*100, '%')
99
100
101
102 print("Precision score {}%".format(round(precision_score(y_test
```

```
    , predict),3)))
print("Recall score {}%".format(round(recall_score(y_test,
    predict),3)))
print("F1 Score {}%".format(round(f1_score(y_test, predict,
    average='weighted'),3)))

y_score = net.fit(X1_train, y_train).predict_proba(X1_test)
    [:,1]

fpr, tpr, thresholds = roc_curve(y_test, y_score)


fig, ax = plt.subplots(1, figsize=(12, 6))
plt.plot(fpr, tpr, color='blue',marker='o', label='ROC curve
    for DNN')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate (1 - specificity)')
plt.ylabel('True Positive Rate (sensitivity)')
plt.title('ROC Curve for WDBC data Classifier')
plt.legend(loc="lower right")

#Plot confusion matrix
def make_confusion_matrix(cf,
                          group_names=None,
                          categories='auto',
                          count=True,
                          percent=True,
                          cbar=True,
                          xyticks=True,
                          xyplotlabels=True,
                          sum_stats=True,
                          figsize=None,
                          cmap='Blues',
                          title='Confusion matrix for WDBC -
    test data'):


# CODE TO GENERATE TEXT INSIDE EACH SQUARE
    blanks = ['' for i in range(cf.size)]

    if group_names and len(group_names)==cf.size:
        group_labels = ["{}\n".format(value) for value in
    group_names]
    else:
        group_labels = blanks
```

```
141
142     if count:
143         group_counts = ["{0:0.0f}\n".format(value) for value in
        cf.flatten()]
144     else:
145         group_counts = blanks
146
147     if percent:
148         group_percentages = ["{0:.2%}".format(value) for value
    in cf.flatten()/np.sum(cf)]
149     else:
150         group_percentages = blanks
151
152     box_labels = [f"{v1}{v2}{v3}".strip() for v1, v2, v3 in zip
    (group_labels,group_counts,group_percentages)]
153     box_labels = np.asarray(box_labels).reshape(cf.shape[0],cf.
    shape[1])
154
155
156     # CODE TO GENERATE SUMMARY STATISTICS & TEXT FOR SUMMARY
    STATS
157     if sum_stats:
158         #Accuracy is sum of diagonal divided by total
    observations
159         accuracy  = np.trace(cf) / float(np.sum(cf))
160
161         #if it is a binary confusion matrix, show some more
    stats
162         if len(cf)==2:
163             #Metrics for Binary Confusion Matrices
164             precision = cf[1,1] / sum(cf[:,1])
165             recall    = cf[1,1] / sum(cf[1,:])
166             f1_score  = 2*precision*recall / (precision +
    recall)
167             stats_text = "\n\nAccuracy={:0.3f}\nPrecision={:0.3
    f}\nRecall={:0.3f}\nF1 Score={:0.3f}".format(
168                 accuracy,precision,recall,f1_score)
169         else:
170             stats_text = "\n\nAccuracy={:0.3f}".format(accuracy
    )
171     else:
172         stats_text = ""
173
174
175     # SET FIGURE PARAMETERS ACCORDING TO OTHER ARGUMENTS
```

```
    if figsize==None:
        #Get default figure size if not set
        figsize = plt.rcParams.get('figure.figsize')

    if xyticks==False:
        #Do not show categories if xyticks is False
        categories=False


    # MAKE THE HEATMAP VISUALIZATION
    plt.figure(figsize=figsize)
    sns.heatmap(cf,annot=box_labels,fmt="",cmap=cmap,cbar=cbar,
   xticklabels=categories,yticklabels=categories)

    if xyplotlabels:
        plt.ylabel('True label')
        plt.xlabel('Predicted label' + stats_text)
    else:
        plt.xlabel(stats_text)

    if title:
        plt.title('Confusion matrix for WDBC - test data')

cf_matrix = confusion_matrix(predict, y_test)
print(cf_matrix)
labels = ['True Negative','False Positive','False Negative','
   True Positive']
categories = ['Benign', 'Malignant']
make_confusion_matrix(cf_matrix, group_names=labels,categories=
   categories, percent=True, cbar=False, xyticks=True,)


#Precision-Recall

#Compute the average precision score
from sklearn.metrics import average_precision_score,
   recall_score
average_precision = average_precision_score(y_test, y_score)

print('Average precision-recall score: {0:0.2f}'.format(
      average_precision))


print("Precision score {}%".format(round(precision_score(y_test
   , predict,average='micro'),3)))
```

```
print("Recall score {}%".format(round(recall_score(y_test,
    predict,average='micro'),3)))
print("F1 Score {}%".format(round(f1_score(y_test, predict,
    average='micro'),3)))

#Plot the Precision-Recall curve
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import plot_precision_recall_curve
import matplotlib.pyplot as plt

disp = plot_precision_recall_curve(net, X1_test, y_test, )
disp.ax_.set_title('2-class Precision-Recall curve for WDBC
    data classifier: '
                   'AP={0:0.2f}'.format(average_precision))


print('total time elapsed for test dataset',toc-tic,'second')
```

Listing 4: DNN with ICA and k-fold CV for WDBC and WBC data set

4. Python code for diagnosis of breast cancer using Adaptive Neuro Fuzzy Inference System

1. Python code for ANFIS with modified Relief algorithm for WBC data set

```
# Modified Date: 06 May 2021

# import pandas as pd
import itertools
import numpy as np

from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt

import openpyxl

from functions import *

from MembershipFunctions import *
```

```
18
19 class Anfis:
20     mf_types = {'gauss': Gauss, 'triangular': Triangular, '
   trapezoidal': Trapezoidal, 'bellshaped': BellShaped}
21
22     def __init__(self, mf_type='gauss', fuzzy_types=['low', '
   medium', 'high'], max_features=4, epoch=100, threshold
   =0.001):
23         self.mf_type = mf_type
24         self.mf = Anfis.mf_types[mf_type]()
25         self.fuzzy_types = fuzzy_types
26         self.max_features = max_features
27         self.epoch = epoch
28         self.is_trained = False
29         self.is_tested = False
30         self.is_test_accuracy_calculated = False
31         self.error_threshold = threshold
32         self.is_dummy_trained = False
33         self.errors = []
34
35     def fit(self, x_training, y_training):
36         x_training = x_training.iloc[:,:self.max_features]
37         self.x_training = x_training
38         self.y_training = y_training
39
40         self.input_columns = list(self.x_training.keys())
41         self.combinations = list(itertools.product(*[self.
   fuzzy_types]*len(self.input_columns)))
42
43         x_maximum = {}
44         x_minimum = {}
45
46         for i in self.input_columns:
47             x_maximum[i] = self.x_training[i].max()
48             x_minimum[i] = self.x_training[i].min()
49
50         self.fuzzified = {}
51         for i in self.input_columns:
52             self.fuzzified[i] = self.mf.get_fuzzified(x_minimum
   [i], x_maximum[i])
53
54         plot_x = {}
55         plot_data = {}
56         for i in self.x_training.index:
57             l1 = layer1(self.x_training, i, self.input_columns)
```

```
58            mu_values = layer2(l1, self.fuzzified, self.mf.
    membership_func)
59            # print(mu_values)
60            for j in mu_values:
61                if j not in plot_x:
62                    plot_x[j] = []
63                plot_x[j].append(l1[j])
64                if j not in plot_data:
65                    plot_data[j] = {}
66                for fuzzy in mu_values[j]:
67                    if fuzzy not in plot_data[j]:
68                        plot_data[j][fuzzy] = []
69                    plot_data[j][fuzzy].append(mu_values[j][
    fuzzy])
70        #print(plot_data)
71        self.plot_data = plot_data
72        self.plot_x = plot_x
73
74        # print("Plotting Graph..")
75        # time.sleep(2)
76
77        # self.plot_mf()
78
79        self.dimension = {'p': self.y_training.shape[0], 'n':
    len(self.combinations), 'm': len(self.input_columns)}
80
81        iterations = 0
82        pred_values = []
83
84        while iterations<self.epoch:
85            row_weights = []
86            for i in self.x_training.index:
87                l1 = layer1(self.x_training, i, self.
    input_columns)
88                forward_output = self.mf.forward_pass(l1, self.
    fuzzified, self.combinations)
89                row_weights.append(forward_output['
    normalized_weight_values'])
90            A_matrix = A_matrix_final(row_weights, self.
    dimension, self.x_training, self.input_columns)
91            A_matrix_T = np.transpose(A_matrix)
92            A_dot_A_T = A_matrix_T.dot(A_matrix)
93            A_dot_A_T_Inv = np.linalg.pinv(A_dot_A_T)
94            A_T_dot_Y = A_matrix_T.dot(self.y_training)
95            self.K_matrix = A_dot_A_T_Inv.dot(A_T_dot_Y)
```

```
96
97              k_count = len(self.input_columns) + 1
98              self.K_matrix = self.K_matrix.reshape(len(self.
    K_matrix)//k_count, k_count)
99
100             temp_values = []
101             indexes = list(self.x_training.index)
102             for i in range(len(row_weights)):
103                 l1 = layer1(self.x_training, indexes[i], self.
    input_columns)
104                 layer5_v = layer5_value(row_weights[i], l1,
    self.K_matrix)
105                 layer6_v = layer6_value(layer5_v)
106                 temp_values.append(layer6_v)
107
108             print("--------------Y Values-------------")
109             print(temp_values)
110             pred_values = temp_values
111
112             error = abs(np.sum(np.subtract(pred_values, self.
    y_training))) / 2
113             print("-------------Error----------------")
114             print(error)
115             time.sleep(1)
116
117             self.errors.append(error)
118
119             if error<self.error_threshold:
120                 print("Done.....")
121                 break
122
123             delta_final = {}
124             indexes = list(self.x_training.index)
125             for i in range(len(indexes)):
126                 l1 = layer1(self.x_training, indexes[i], self.
    input_columns)
127                 print(pred_values[i] - self.y_training[indexes[
    i]])
128                 # time.sleep(1)
129                 delta = self.mf.backward_pass(l1, self.
    fuzzified, self.combinations, self.K_matrix, pred_values[i]
     - self.y_training[indexes[i]])
130                 if i==0:
131                     delta_final = delta
132                 else:
```

```
                    for attr in delta_final:
                        for attr_range in delta_final[attr]:
                            for j in range(len(delta_final[attr
][attr_range])):
                                delta_final[attr][attr_range][j
] += delta[attr][attr_range][j]

            fuzzified = {}
            for attr in self.fuzzified:
                fuzzified[attr] = {}
                for attr_range in self.fuzzified[attr]:
                    fuzzified[attr][attr_range] = []
                    for c in range(len(self.fuzzified[attr][
attr_range])):
                        fuzzified[attr][attr_range].append(self
.fuzzified[attr][attr_range][c] + delta_final[attr][
attr_range][c])

            self.fuzzified = fuzzified
            print("\n")

            iterations+=1

        # print(self.K_matrix)

        # print(self.fuzzified)

        plot_x = {}
        plot_data = {}
        for i in self.x_training.index:
            l1 = layer1(self.x_training, i, self.input_columns)
            mu_values = layer2(l1, self.fuzzified, self.mf.
membership_func)
            # print(mu_values)
            for j in mu_values:
                if j not in plot_x:
                    plot_x[j] = []
                plot_x[j].append(l1[j])
                if j not in plot_data:
                    plot_data[j] = {}
                for fuzzy in mu_values[j]:
                    if fuzzy not in plot_data[j]:
                        plot_data[j][fuzzy] = []
                    plot_data[j][fuzzy].append(mu_values[j][
fuzzy])
```

```
171         print(plot_data)
172         self.plot_data = plot_data
173         self.plot_x = plot_x
174 
175         actual_values = self.y_training.to_numpy()
176         predicted_values = np.around(pred_values, decimals=0).
    astype(int)
177 
178         for i in range(len(predicted_values)):
179             if actual_values[i]==2 and predicted_values[i]!=2:
180                 predicted_values[i] = 4
181             elif actual_values[i]==4 and predicted_values[i
    ]!=4:
182                 predicted_values[i] = 2
183 
184         output_labels = list(set(actual_values))
185         conf_mat = confusion_matrix(actual_values,
    predicted_values, labels=output_labels)
186 
187         print(conf_mat)
188 
189         tp, fn, fp, tn  = conf_mat.reshape(-1)
190         self.training_conf_mat = {
191             'True Positive': tp,
192             'False Negative': fn,
193             'False Positive': fp,
194             'True Negative': tn
195         }
196 
197         conf_report = classification_report(actual_values,
    predicted_values, labels=output_labels)
198         print(conf_report)
199 
200         self.training_conf_report = classification_report(
    actual_values, predicted_values, labels=output_labels,
    output_dict=True)
201 
202         actual = np.array(actual_values)
203         predicted = np.array(predicted_values)
204 
205         missed = np.subtract(actual, predicted)
206 
207         print(len(missed))
208         print(missed)
209 
```

```
        missed_samples = {}
        for i in range(len(missed)):
            if missed[i]!=0 and missed[i] not in output_labels:
                missed_samples[i] = missed[i]

        print("Missed Samples and its difference Values")
        print(missed_samples)

        acc = accuracy_score(actual_values, predicted_values)
        print("Accuracy: ", acc)

        self.is_trained = True
        self.training_accuracy = acc
        self.training_predicted_values = pred_values
        self.training_actual_values = actual_values
        self.training_error_values = np.subtract(pred_values,
actual_values)

    def dummy_training(self, x_test_train, y_test_train):
        x_test_train = x_test_train.iloc[:,:self.max_features]
        self.dummy_x_training = x_test_train
        self.dummy_y_training = y_test_train

        self.dummy_input_columns = list(self.dummy_x_training.
keys())
        self.dummy_combinations = list(itertools.product(*[self
.fuzzy_types]*len(self.dummy_input_columns)))

        x_maximum = {}
        x_minimum = {}

        for i in self.input_columns:
            x_maximum[i] = self.dummy_x_training[i].max()
            x_minimum[i] = self.dummy_x_training[i].min()

        self.dummy_fuzzified = {}
        for i in self.dummy_input_columns:
            self.dummy_fuzzified[i] = self.mf.get_fuzzified(
x_minimum[i], x_maximum[i])

        self.dummy_dimension = {'p': self.dummy_y_training.
shape[0], 'n': len(self.dummy_combinations), 'm': len(self.
dummy_input_columns)}

        iterations = 0
```

```
249         pred_values = []
250 
251         while iterations<self.epoch:
252             row_weights = []
253             for i in self.dummy_x_training.index:
254                 l1 = layer1(self.dummy_x_training, i, self.
    dummy_input_columns)
255                 forward_output = self.mf.forward_pass(l1, self.
    dummy_fuzzified, self.dummy_combinations)
256                 row_weights.append(forward_output['
    normalized_weight_values'])
257             A_matrix = A_matrix_final(row_weights, self.
    dummy_dimension, self.dummy_x_training, self.
    dummy_input_columns)
258             A_matrix_T = np.transpose(A_matrix)
259             A_dot_A_T = A_matrix_T.dot(A_matrix)
260             A_dot_A_T_Inv = np.linalg.pinv(A_dot_A_T)
261             A_T_dot_Y = A_matrix_T.dot(self.dummy_y_training)
262             self.dummy_K_matrix = A_dot_A_T_Inv.dot(A_T_dot_Y)
263 
264             k_count = len(self.dummy_input_columns) + 1
265             self.dummy_K_matrix = self.dummy_K_matrix.reshape(
    len(self.dummy_K_matrix)//k_count, k_count)
266 
267             temp_values = []
268             indexes = list(self.dummy_x_training.index)
269             for i in range(len(row_weights)):
270                 l1 = layer1(self.dummy_x_training, indexes[i],
    self.dummy_input_columns)
271                 layer5_v = layer5_value(row_weights[i], l1,
    self.dummy_K_matrix)
272                 layer6_v = layer6_value(layer5_v)
273                 temp_values.append(layer6_v)
274 
275             print("--------------Y Values-------------")
276             print(temp_values)
277             pred_values = temp_values
278 
279             error = np.sum(np.subtract(pred_values, self.
    dummy_y_training))
280             print("-------------Error----------------")
281             print(error)
282             time.sleep(1)
283 
284             if abs(error)<self.error_threshold:
```

```
285                 print("Done.....")
286                 break
287 
288             delta_final = {}
289             indexes = list(self.dummy_x_training.index)
290             for i in range(len(indexes)):
291                 l1 = layer1(self.dummy_x_training, indexes[i],
    self.dummy_input_columns)
292                 print(pred_values[i] - self.dummy_y_training[
    indexes[i]])
293                 # time.sleep(1)
294                 delta = self.mf.backward_pass(l1, self.
    dummy_fuzzified, self.dummy_combinations, self.
    dummy_K_matrix, pred_values[i] - self.dummy_y_training[
    indexes[i]])
295                 if i==0:
296                     delta_final = delta
297                 else:
298                     for attr in delta_final:
299                         for attr_range in delta_final[attr]:
300                             for j in range(len(delta_final[attr
    ][attr_range])):
301                                 delta_final[attr][attr_range][j
    ] += delta[attr][attr_range][j]
302 
303             fuzzified = {}
304             for attr in self.dummy_fuzzified:
305                 fuzzified[attr] = {}
306                 for attr_range in self.dummy_fuzzified[attr]:
307                     fuzzified[attr][attr_range] = []
308                     for c in range(len(self.dummy_fuzzified[
    attr][attr_range])):
309                         fuzzified[attr][attr_range].append(self
    .dummy_fuzzified[attr][attr_range][c] + delta_final[attr][
    attr_range][c])
310 
311             self.dummy_fuzzified = fuzzified
312             print("\n")
313 
314             iterations+=1
315 
316         print(self.dummy_K_matrix)
317 
318         actual_values = self.dummy_y_training.to_numpy()
319         predicted_values = np.around(pred_values, decimals=0).
```

```
    astype(int)
320
321         for i in range(len(predicted_values)):
322             if actual_values[i]==2 and predicted_values[i]!=2:
323                 predicted_values[i] = 4
324             elif actual_values[i]==4 and predicted_values[i
    ]!=4:
325                 predicted_values[i] = 2
326
327         output_labels = list(set(actual_values))
328         conf_mat = confusion_matrix(actual_values,
    predicted_values, labels=output_labels)
329
330         print(conf_mat)
331
332         tp, fn, fp, tn  = conf_mat.reshape(-1)
333         self.dummy_training_conf_mat = {
334             'True Positive': tp,
335             'False Negative': fn,
336             'False Positive': fp,
337             'True Negative': tn
338         }
339
340         conf_report = classification_report(actual_values,
    predicted_values, labels=output_labels)
341         print(conf_report)
342
343         self.dummy_training_conf_report = classification_report
    (actual_values, predicted_values, labels=output_labels,
    output_dict=True)
344
345         actual = np.array(actual_values)
346         predicted = np.array(predicted_values)
347
348         missed = np.subtract(actual, predicted)
349
350         print(len(missed))
351         print(missed)
352
353         missed_samples = {}
354         for i in range(len(missed)):
355             if missed[i]!=0 and missed[i] not in output_labels:
356                 missed_samples[i] = missed[i]
357
358         print("Missed Samples and its difference Values")
```

```
359         print(missed_samples)
360 
361         acc = accuracy_score(actual_values, predicted_values)
362         print("Accuracy: ", acc)
363 
364         self.is_dummy_trained = True
365         self.dummy_accuracy = acc
366         self.dummy_predicted_values = pred_values
367         self.dummy_actual_values = actual_values
368         self.dummy_error_values = np.subtract(pred_values,
    actual_values)
369 
370     def predict(self, x_test):
371         if self.is_trained:
372             x_test = x_test.iloc[:,:self.max_features]
373             self.x_test = x_test
374 
375             pred_values = []
376 
377             for i in self.x_test.index:
378                 l1 = layer1(self.x_test, i, self.input_columns)
379                 forward_output = self.mf.forward_pass(l1, self.
    fuzzified, self.combinations)
380                 layer5_v = layer5_value(forward_output['
    normalized_weight_values'], l1, self.K_matrix)
381                 layer6_v = layer6_value(layer5_v)
382                 pred_values.append(layer6_v)
383 
384             self.is_tested = True
385             self.test_predicted_values = pred_values
386 
387             return pred_values
388         else:
389             # print("Not Trained")
390             return []
391 
392     def test_accuracy(self, pred_values, actual_values):
393         if self.is_tested:
394             predicted_values = np.around(pred_values, decimals
    =0).astype(int)
395 
396             for i in range(len(predicted_values)):
397                 if actual_values[i]==2 and predicted_values[i
    ]!=2:
398                     predicted_values[i] = 4
```

```
399                 elif actual_values[i]==4 and predicted_values[i
    ]!=4:
400                     predicted_values[i] = 2
401 
402             output_labels = list(set(actual_values))
403             conf_mat = confusion_matrix(actual_values,
    predicted_values, labels=output_labels)
404 
405             print(conf_mat)
406 
407             tp, fn, fp, tn  = conf_mat.reshape(-1)
408             self.testing_conf_mat = {
409                 'True Positive': tp,
410                 'False Negative': fn,
411                 'False Positive': fp,
412                 'True Negative': tn
413             }
414 
415             conf_report = classification_report(actual_values,
    predicted_values, labels=output_labels)
416             print(conf_report)
417 
418             self.testing_conf_report = classification_report(
    actual_values, predicted_values, labels=output_labels,
    output_dict=True)
419 
420             actual = np.array(actual_values)
421             predicted = np.array(predicted_values)
422 
423             missed = np.subtract(actual, predicted)
424 
425             print(len(missed))
426             print(missed)
427 
428             missed_samples = {}
429             for i in range(len(missed)):
430                 if missed[i]!=0 and missed[i] not in
    output_labels:
431                     missed_samples[i] = missed[i]
432 
433             print("Missed Samples and its difference Values")
434             print(missed_samples)
435 
436             self.test_accuracy = accuracy_score(actual_values,
    predicted_values)
```

```
437             self.is_test_accuracy_calculated = True
438             self.test_actual_values = actual_values
439             self.test_error_values = np.subtract(self.
    test_predicted_values, actual_values)
440 
441             return self.test_accuracy
442         else:
443             print("Testing is Not Done")
444             return 0.0
445 
446     def generate_report(self):
447         if self.is_test_accuracy_calculated:
448             filename = input("Enter File Name: ")
449 
450             training_data = {
451                 'Predicted Values': [],
452                 'Actual Values': [],
453                 'Error': []
454             }
455 
456             for i, j, k in zip(self.training_predicted_values,
    self.training_actual_values, self.training_error_values):
457                 training_data['Predicted Values'].append(i)
458                 training_data['Actual Values'].append(j)
459                 training_data['Error'].append(k)
460 
461             training_df = pd.DataFrame(training_data)
462 
463             testing_data = {
464                 'Predicted Values': [],
465                 'Actual Values': [],
466                 'Error': []
467             }
468 
469             for i, j, k in zip(self.test_predicted_values, self
    .test_actual_values, self.test_error_values):
470                 testing_data['Predicted Values'].append(i)
471                 testing_data['Actual Values'].append(j)
472                 testing_data['Error'].append(k)
473 
474             testing_df = pd.DataFrame(testing_data)
475 
476             accuracy_df = pd.DataFrame(data = {
477                 'Title': [
478                     'Membership Function',
```

```
                    'Threshold',
                    'Training_Accuracy',
                    'Testing_Accuracy',
                ],
                'Values': [
                    self.mf_type,
                    self.error_threshold,
                    self.training_accuracy,
                    self.test_accuracy,
                ]
            })

            kmatrix_df = pd.DataFrame(data=self.K_matrix)
            fuzzified_df = pd.DataFrame(data=self.fuzzified,
columns=list(self.fuzzified.keys()))

            training_conf_mat_df = pd.DataFrame(data = {
                'Title': list(self.training_conf_mat.keys()),
                'Values': list(self.training_conf_mat.values())
            })
            training_conf_report_df = pd.DataFrame(self.
training_conf_report)
            testing_conf_mat_df = pd.DataFrame(data = {
                'Title': list(self.testing_conf_mat.keys()),
                'Values': list(self.testing_conf_mat.values())
            })
            testing_conf_report_df = pd.DataFrame(self.
testing_conf_report)

            if filename[-5::-1]!='.xlsx':
                filename += '.xlsx'

            with pd.ExcelWriter(filename) as writer:
                training_df.to_excel(writer, sheet_name='
Training Data')
                testing_df.to_excel(writer, sheet_name='Testing
 Data')
                accuracy_df.to_excel(writer, sheet_name='
Accuracy')
                kmatrix_df.to_excel(writer, sheet_name='K
Matrix')
                fuzzified_df.to_excel(writer, sheet_name='
Fuzzified Values')
                # errors_df.to_excel(writer, sheet_name='Error
Values')
```

```
515                 training_conf_mat_df.to_excel(writer,
    sheet_name='Training Confusion Matrix')
516                 training_conf_report_df.to_excel(writer,
    sheet_name='Training Confusion Report')
517                 testing_conf_mat_df.to_excel(writer, sheet_name
    ='Testing Confusion Matrix')
518                 testing_conf_report_df.to_excel(writer,
    sheet_name='Testing Confusion Report')
519                 print("Report Generated in File:", filename)
520
521     def generate_dummy_report(self):
522         if self.is_dummy_trained:
523             filename = input("Enter Dummy Training Filename: ")
524
525             dummy_training = {
526                 'Predicted Values': [],
527                 'Actual Values': [],
528                 'Error': []
529             }
530
531             for i, j, k in zip(self.dummy_predicted_values,
    self.dummy_actual_values, self.dummy_error_values):
532                 dummy_training['Predicted Values'].append(i)
533                 dummy_training['Actual Values'].append(j)
534                 dummy_training['Error'].append(k)
535
536             dummy_df = pd.DataFrame(dummy_training)
537             kmatrix_df = pd.DataFrame(data=self.dummy_K_matrix)
538             fuzzified_df = pd.DataFrame(data=self.
    dummy_fuzzified, columns=list(self.dummy_fuzzified.keys()))
539             accuracy_df = pd.DataFrame(data = {
540                 'Title': [
541                     'Membership Function',
542                     'Threshold',
543                     'Dummy_Accuracy'
544                 ],
545                 'Values': [
546                     self.mf_type,
547                     self.error_threshold,
548                     self.dummy_accuracy,
549                 ]
550             })
551             conf_mat_df = pd.DataFrame(data = {
552                 'Title': list(self.dummy_training_conf_mat.keys
    ()),
```

```
553                 'Values': list(self.dummy_training_conf_mat.
    values())
554             })
555             conf_report_df = pd.DataFrame(self.
    dummy_training_conf_report)
556 
557             if filename[-5::-1]!='.xlsx':
558                 filename += '.xlsx'
559 
560             with pd.ExcelWriter(filename) as writer:
561                 dummy_df.to_excel(writer, sheet_name='Dummy
    Training Data')
562                 accuracy_df.to_excel(writer, sheet_name='Dummy
    Accuracy')
563                 kmatrix_df.to_excel(writer, sheet_name='Dummy K
     Matrix')
564                 fuzzified_df.to_excel(writer, sheet_name='Dummy
     Fuzzified Values')
565                 conf_mat_df.to_excel(writer, sheet_name='Dummy
    Confusion Matrix')
566                 conf_report_df.to_excel(writer, sheet_name='
    Dummy Confusion Report')
567                 print("Report Generated in File:", filename)
568 
569     def plot_error_graph(self):
570         if self.is_trained:
571             print(self.errors)
572             # plt.plot(self.errors)
573             plt.plot([f"epoch {x}" for x in range(1, len(self.
    errors)+1)], self.errors, marker = 'o', c = 'red')
574             plt.xlabel("Epoch", fontdict = {'family':'serif','
    color':'darkred','size':15})
575             plt.ylabel("Error", fontdict = {'family':'serif','
    color':'darkred','size':15})
576             plt.xticks(ticks=range(len(self.errors)), labels=[f
    "epoch {x}" for x in range(1, len(self.errors)+1)],
    rotation='vertical')
577             plt.subplots_adjust(bottom = 0.2, top=0.9)
578             plt.show()
579 
580     def plot_mf(self):
581         for type, x_values in self.plot_x.items():
582             # print(len(x_values),'->',len(self.plot_data[type
    ]['low']))
583             # print(len(x_values),'->',len(self.plot_data[type
```

```
    ]['medium']))
584             # print(len(x_values),'->',len(self.plot_data[type
    ]['high']))
585             # plt.subplot(3,1,1)
586             plt.scatter(x_values, self.plot_data[type]['low'])
587             # plt.plot(x_values, self.plot_data[type]['low'])
588             # plt.subplot(3,1,2)
589             plt.scatter(x_values, self.plot_data[type]['medium'
    ])
590             #plt.plot(x_values, self.plot_data[type]['medium'])
591             # plt.subplot(3,1,3)
592             plt.scatter(x_values, self.plot_data[type]['high'])
593             #plt.plot(x_values, self.plot_data[type]['high'])
594
595             plt.title(type)
596             plt.xlabel("X Values")
597             plt.ylabel("Mf(X) Values")
598
599             plt.show()
```

Listing 5: ANFIS with modified Relief algorithm for WBC data set

```
1  # Modified Date: 07 May 2021
2
3  from matplotlib import pyplot as plt
4
5  from functions import *
6
7  import numpy as np
8  import time
9
10 import math
11
12 class Gauss:
13     def get_fuzzified(self, start, end):
14         mid = get_mid(start, end)
15
16         return {
17             'low': [1.911, start],
18             'medium': [1.911, mid],
19             'high': [1.911, end]
20         }
21
22     def plot_graph(self, l1, fuzzified):
23         print(l1)
24         print(mu_values)
```

```
25          # print(fuzzy_values.values())
26          # for k,v in fuzzy_values.items():
27          #     plt.plot(list(v.keys()), list(v.values()))
28          #     plt.xlabel("Features")
29          #     plt.xticks(ticks=range(len(v)), labels = list(v.
    keys()), rotation='vertical')
30          #     plt.ylabel("Weights")
31          #     plt.subplots_adjust(bottom = 0.4, top=0.9)
32          #     plt.show()
33
34      def membership_func(self, v, mean, std):
35          num = pow(v - mean,2)
36          den = 2 * pow(std, 2)
37          return math.exp(-num / den)
38
39      def derivative(self, gauss_mf, v, mean, std, type):
40          if type=='mean':
41              return gauss_mf * (v - mean) / pow(std, 2)
42          else:
43              return gauss_mf * pow(v - mean, 2) / std
44
45      def forward_pass(self, x_values, fuzzified, combinations):
46          mu_values = layer2(x_values, fuzzified, self.
    membership_func)
47          # print(mu_values)
48          weight_values = layer3(x_values, mu_values,
    combinations)
49          # print(weight_values)
50          normalized_weight_values = layer4(weight_values)
51          # print(normalized_weight_values)
52          equations = layer5(normalized_weight_values, x_values)
53          # for equation in equations:
54          #     print(equation)
55          equation_sum = layer6(equations)
56          # print(equation_sum)
57          return {
58              'mu_values': mu_values,
59              'weight_values': weight_values,
60              'normalized_weight_values':
    normalized_weight_values,
61          }
62
63      def backward_pass(self, x_values, fuzzified, combinations,
    k_matrix, error):
64          mu_values = layer2(x_values, fuzzified, self.
```

```
    membership_func)
65          weight_values = layer3(x_values, mu_values,
    combinations)
66
67          learning_rate = 0.01
68
69          delta = {}
70          for attr in mu_values:
71              delta[attr] = {}
72              for attr_range in mu_values[attr]:
73                  f_i_w_i = 0
74                  for it in range(len(weight_values)):
75                      w_i = weight_values[it]
76                      k_i = k_matrix[it]
77                      x_values_list = list(x_values.values())
78                      f_i = k_i[0]
79                      for temp_it in range(1, len(k_i)):
80                          f_i += k_i[temp_it] * x_values_list[
    temp_it-1]
81                      f_i_w_i += f_i * w_i * (1-w_i)
82                  if mu_values[attr][attr_range]==0:
83                      delta[attr][attr_range] = [0,0]
84                  else:
85                      delta[attr][attr_range] = [
86                          learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(mu_values[
    attr][attr_range], x_values[attr], *fuzzified[attr][
    attr_range], 'mean'),
87                          learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(mu_values[
    attr][attr_range], x_values[attr], *fuzzified[attr][
    attr_range], 'std')
88                      ]
89          return delta
90
91
92  class Triangular:
93      def get_fuzzified(self, start, end):
94          mid = get_mid(start, end)
95          diff = end - start
96
97          return {
98              'low': [mid - diff, get_mid(mid - diff, mid), mid],
99              'medium': [start, mid, end],
100             'high': [mid, get_mid(mid, mid + diff), mid + diff]
```

```
        }

    def membership_func(self, v, a, b, c):
        t1 = (v - a) / (b - a)
        t2 = (c - v) / (c - b)
        return max(min(t1, t2), 0)

    def derivative(self, v, a, b, c, type):
        if type=='a':
            return v - b / pow(b-a, 2)
        elif type=='b':
            return a - v / pow(b-a, 2)
        else:
            return v - b / pow(c-b, 2)

    def forward_pass(self, x_values, fuzzified, combinations):
        mu_values = layer2(x_values, fuzzified, self.
membership_func)
        # print(mu_values)
        weight_values = layer3(x_values, mu_values,
combinations)
        # print(weight_values)
        normalized_weight_values = layer4(weight_values)
        # print(normalized_weight_values)
        equations = layer5(normalized_weight_values, x_values)
        # for equation in equations:
        #     print(equation)
        equation_sum = layer6(equations)
        # print(equation_sum)
        return {
            'mu_values': mu_values,
            'weight_values': weight_values,
            'normalized_weight_values':
normalized_weight_values,
        }

    def backward_pass(self, x_values, fuzzified, combinations,
k_matrix, error):
        mu_values = layer2(x_values, fuzzified, self.
membership_func)
        weight_values = layer3(x_values, mu_values,
combinations)

        learning_rate = 0.01

```

```
        delta = {}
        for attr in mu_values:
            delta[attr] = {}
            for attr_range in mu_values[attr]:
                f_i_w_i = 0
                for it in range(len(weight_values)):
                    w_i = weight_values[it]
                    k_i = k_matrix[it]
                    x_values_list = list(x_values.values())
                    f_i = k_i[0]
                    for temp_it in range(1, len(k_i)):
                        f_i += k_i[temp_it] * x_values_list[
    temp_it-1]
                    f_i_w_i += f_i * w_i * (1-w_i)
                if mu_values[attr][attr_range]==0:
                    delta[attr][attr_range] = [0,0,0]
                else:
                    delta[attr][attr_range] = [
                        learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(x_values[
    attr], *fuzzified[attr][attr_range], 'a'),
                        learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(x_values[
    attr], *fuzzified[attr][attr_range], 'b'),
                        learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(x_values[
    attr], *fuzzified[attr][attr_range], 'c')
                    ]
        return delta


class Trapezoidal:
    def get_fuzzified(self, start, end):
        low = start
        high = end
        mid = get_mid(start, end)
        diff_by_2 = end-start / 2
        diff_70 = diff_by_2 * 0.7
        diff_30 = diff_by_2 * 0.3

        return {
            'low': [low - diff_70, low - diff_30, mid - diff_70
    , mid - diff_30],
            'medium': [mid - diff_70, mid - diff_30, mid +
    diff_30, mid + diff_70],
```

```
            'high': [mid + diff_30, mid + diff_70, high +
diff_30, high + diff_70]
        }

    def membership_func(self, v, a, b, c, d):
        t1 = (v - a) / (b - a)
        t2 = (d - v) / (d - c)
        return max(min(t1, t2), 0)

    def derivative(self, v, a, b, c, d, type):
        if type=='a':
            return v - b / pow(b-a, 2)
        elif type=='b':
            return a - v / pow(b-a, 2)
        elif type=='c':
            return d - v / pow(d-c, 2)
        else:
            return v - c / pow(d-c, 2)

    def forward_pass(self, x_values, fuzzified, combinations):
        mu_values = layer2(x_values, fuzzified, self.
membership_func)
        # print(mu_values)
        weight_values = layer3(x_values, mu_values,
combinations)
        # print(weight_values)
        normalized_weight_values = layer4(weight_values)
        # print(normalized_weight_values)
        equations = layer5(normalized_weight_values, x_values)
        # for equation in equations:
        #     print(equation)
        equation_sum = layer6(equations)
        # print(equation_sum)
        return {
            'mu_values': mu_values,
            'weight_values': weight_values,
            'normalized_weight_values':
normalized_weight_values,
        }

    def backward_pass(self, x_values, fuzzified, combinations,
k_matrix, error):
        mu_values = layer2(x_values, fuzzified, self.
membership_func)
        weight_values = layer3(x_values, mu_values,
```

```
    combinations)
215
216         learning_rate = 0.01
217
218         delta = {}
219         for attr in mu_values:
220             delta[attr] = {}
221             for attr_range in mu_values[attr]:
222                 f_i_w_i = 0
223                 for it in range(len(weight_values)):
224                     w_i = weight_values[it]
225                     k_i = k_matrix[it]
226                     x_values_list = list(x_values.values())
227                     f_i = k_i[0]
228                     for temp_it in range(1, len(k_i)):
229                         f_i += k_i[temp_it] * x_values_list[
    temp_it-1]
230                     f_i_w_i += f_i * w_i * (1-w_i)
231                 if mu_values[attr][attr_range]==0:
232                     delta[attr][attr_range] = [0,0,0,0]
233                 else:
234                     delta[attr][attr_range] = [
235                         learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(x_values[
    attr], *fuzzified[attr][attr_range], 'a'),
236                         learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(x_values[
    attr], *fuzzified[attr][attr_range], 'b'),
237                         learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(x_values[
    attr], *fuzzified[attr][attr_range], 'c'),
238                         learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(x_values[
    attr], *fuzzified[attr][attr_range], 'd')
239                     ]
240         return delta
241
242
243 class BellShaped:
244     def get_fuzzified(self, start, end):
245         mid = get_mid(start, end)
246         diff_by_4 = end - start / 4
247         start_2 = start * 2
248
249         return {
```

```
            'low': [diff_by_4, start_2, start],
            'medium': [diff_by_4, start_2, mid],
            'high': [diff_by_4, start_2, end]
        }

    def membership_func(self, v, a, b, c):
        # # print(v, a, b, c)
        # # print(v-a)
        # # print((v-a)/c)
        # # print(2 * b)
        a = round(a, 2)
        b = round(b, 2)
        c = round(c, 2)
        term = pow(round(abs((v - c) / a), 2), 2 * b)
        # # print(term)
        # print(1 / (1 + term))
        # time.sleep(0.01)
        return round(1 / (1 + term), 2)
        # return gbellmf(v, a, b, c)

    def derivative(self, mf, v, a, b, c, type):
        mf = round(mf, 4)
        a = round(a, 2)
        b = round(b, 2)
        c = round(c, 2)
        if type=='a':
            return pow(mf, 2) * (2.0 * b / c) * (pow(pow((v - a) / c, 2), b) / ((v-a) / c))
        elif type=='b':
            t1 = pow(mf, 2) * pow(pow((v - a) / c, 2), b)
            # print(v,a,b,c)
            # print(abs(v - a))
            # print(abs(v - a) / c)
            t2 = math.log(abs((v - a) / c))
            return -1 * t1 * 2 * t2
        else:
            return pow(mf, 2) * (2 * b / c) * pow(pow(v - a, 2),b)
        # if type=='a':
        #     t1 = 2.0 * b * pow(c-v, 2) * pow(abs((c-v)/a), ((2 * b) - 2))
        #     t2 = pow(a, 3) * pow(pow(abs((c-v)/a), 2*b) + 1, 2)
        #     return t1 / t2
        # elif type=='b':
```

```
291         #     t1 = -1 * (2 * pow(abs((c-v)/a), 2*b) * np.log(
    abs((c-v)/a)))
292         #     t2 = pow(pow(abs((c-v)/a), 2*b) + 1, 2)
293         #     return t1 / t2
294         # else:
295         #     t1 = 2.0 * b * (c-v) * pow(abs((c-v)/a), ((2 * b)
     - 2))
296         #     t2 = pow(a, 2) * pow(pow(abs((c-v)/a), 2*b) + 1,
    2)
297         #     return t1 / t2
298 
299     def forward_pass(self, x_values, fuzzified, combinations):
300         mu_values = layer2(x_values, fuzzified, self.
    membership_func)
301         # print(mu_values)
302         weight_values = layer3(x_values, mu_values,
    combinations)
303         # print(weight_values)
304         normalized_weight_values = layer4(weight_values)
305         # print(normalized_weight_values)
306         equations = layer5(normalized_weight_values, x_values)
307         # for equation in equations:
308         #     print(equation)
309         equation_sum = layer6(equations)
310         # print(equation_sum)
311         return {
312             'mu_values': mu_values,
313             'weight_values': weight_values,
314             'normalized_weight_values':
    normalized_weight_values,
315         }
316 
317     def backward_pass(self, x_values, fuzzified, combinations,
    k_matrix, error):
318         mu_values = layer2(x_values, fuzzified, self.
    membership_func)
319         weight_values = layer3(x_values, mu_values,
    combinations)
320 
321         learning_rate = 0.01
322 
323         delta = {}
324         for attr in mu_values:
325             delta[attr] = {}
326             for attr_range in mu_values[attr]:
```

```
                f_i_w_i = 0
                for it in range(len(weight_values)):
                    w_i = weight_values[it]
                    k_i = k_matrix[it]
                    x_values_list = list(x_values.values())
                    f_i = k_i[0]
                    for temp_it in range(1, len(k_i)):
                        f_i += k_i[temp_it] * x_values_list[
    temp_it-1]
                    f_i_w_i += f_i * w_i * (1-w_i)
                if mu_values[attr][attr_range]==0:
                    delta[attr][attr_range] = [0,0,0]
                else:
                    delta[attr][attr_range] = [
                        learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(mu_values[
    attr][attr_range], x_values[attr], *fuzzified[attr][
    attr_range], 'a'),
                        learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(mu_values[
    attr][attr_range], x_values[attr], *fuzzified[attr][
    attr_range], 'b'),
                        learning_rate * error * f_i_w_i * (1 /
    mu_values[attr][attr_range]) * self.derivative(mu_values[
    attr][attr_range], x_values[attr], *fuzzified[attr][
    attr_range], 'c')
                    ]
        return delta
```

Listing 6: ANFIS with modified Relief algorithm for WBC data set

```
# Modified Date: 06 May 2021
#11F-normalized-4F-MahaloDist.xlsx
#11F-WBCD-Data.xlsx


import sys
import subprocess
import pkg_resources

if __name__=='__main__':
    required = {'pandas', 'numpy', 'openpyxl', 'sklearn'}
    installed = {pkg.key for pkg in pkg_resources.working_set}
    missing = required - installed

    try:
```

```
        # if missing:
        #     print("Installing Required Dependencies...")
        #     python = sys.executable
        #     subprocess.check_call([python, '-m', 'pip', '
    install', '-r', 'requirements.txt'], stdout=subprocess.
    DEVNULL)
        pass
    except Exception as E:
        print("Please Connect to Internet")
    else:
        import pandas as pd
        import numpy as np
        from sklearn.metrics import confusion_matrix
        from sklearn.metrics import classification_report
        from sklearn.metrics import accuracy_score

        from algorithm import Anfis
        #from testing_algo import Anfis

        print("1. Gauss Function")
        print("2. Triangular Function")
        print("3. Trapezoidal Function")
        print("4. Bell Shape Function")
        choice = int(input("Enter Choice: "))

        if 1<=choice<=4:

            mf_types = ['gauss', 'triangular', 'trapezoidal', '
    bellshaped']

            filepath = input("Enter File Path: ")
            df = pd.read_excel(filepath)

            df_testing = df.sample(frac=0.3)
            df_training = df.drop(df_testing.index)

            X_train = df_training.iloc[:,:-1]
            y_train = df_training.iloc[:,-1]

            X_test = df_testing.iloc[:,:-1]
            y_test = df_testing.iloc[:,-1]

            print(X_train)
            print(y_train)
            print(X_test)
```

```
            print(y_test)

            threshold = float(input("Enter Thresold Accuracy: "
    ))

            model = Anfis(mf_type=mf_types[choice-1], threshold
    =threshold)

            # Train Model
            model.fit(X_train, y_train)

            model.plot_error_graph()

           # model.plot_mf()

            # Dummy Training
            model.dummy_training(X_test, y_test)

            # Generate Dummy Report
            model.generate_dummy_report()

            # Predict Values using Model
            pred_values = model.predict(X_test)

            # Test Accuracy of Model
            acc_score = model.test_accuracy(pred_values, y_test
    .to_numpy())

            print("Test Accuracy: ", acc_score)

            # Generate Report
            model.generate_report()

        else:
            print("Wrong Choice")
```

Listing 7: ANFIS with modified Relief algorithm for WBC data set

```
# Modified Date: 07 May 2021

import pandas as pd
import numpy as np

import itertools
import time

```

```
import math

#------------Fuzzified Value Generator-------------#
def get_mid(low, high):
    return (low+high)/2

#-----------------Layer 1-------------------#
def layer1(df, i, input_keys):
    output_row = {}
    for key in input_keys:
        output_row[key] = df.loc[i][key]
    return output_row

#-----------------Layer 2-------------------#
def layer2(row, fuzzified, membership_func):
    output_row = {}
    for key in row:
        output_row[key] = {}
        fuzzi = fuzzified[key]
        for i, fuzzi_values in fuzzi.items():
            output_row[key][i] = round(membership_func(row[key
   ], *fuzzi_values), 3)
    return output_row

#-------------------------------------------#

#---------------Layer 3-------------------#

def layer3(row, fuzzified_data, combinations):
    keys = list(fuzzified_data.keys())
    products = []
    for i in combinations:
        l = []
        for j in range(len(i)):
            l.append(fuzzified_data[keys[j]][i[j]])
        products.append(min(l))
    return products

#-----------------------------------------#

#---------------Layer 4-------------------#

def layer4(weights):
    normalized = []
    weight_sum = sum(weights)
```

```
    if weight_sum==0:
        return []
    for w in weights:
        normalized.append(round(w / weight_sum, 4))
    return normalized

#----------------------------------------#

#---------------Layer 5--------------------#

def layer5(weights, row):
    row_values = list(row.values())
    output = []
    for w in weights:
        y = f"{w}[K0"
        for i in range(len(row)):
            y += " + "
            y += f"K{i+1}({row_values[i]})"
        else:
            y += "]"
            output.append(y)
    return output

def display_list(lst):
    for ele in lst:
        print(ele)

#----------------------------------------#

#---------------Layer 6-------------------#

def layer6(rules):
    output = "y = "
    for i in range(len(rules)):
        if i==0:
            output+=rules[i]
        else:
            output+=("\n + " + rules[i])
    return output

#----------------------------------------#

def get_output_matrix(data):
    return np.transpose([data.to_numpy()])

```

```
98  def make_A_matrix(weights, dimension, sample, input_keys):
99      A_dimension = (dimension['p'], dimension['n']*(1 +
        dimension['m']))
100     A_matrix = np.full(A_dimension, 0.0)
101
102     k = 0
103     for w in weights:
104         A_matrix[0][k] = round(w,4)
105         k+=1
106         for key in input_keys:
107             A_matrix[0][k] = round(w * sample[key],4)
108             k+=1
109
110     return A_matrix
111
112 def layer5_value(weights, row, k_matrix):
113     row_values = list(row.values())
114     output = []
115     for i in range(len(weights)):
116         w = weights[i]
117         k = k_matrix[i]
118         f = k[0]
119         # print(k)
120         # print(row)
121         for j in range(len(row)):
122             f += k[j+1] * row_values[j]
123         else:
124             y = f * w
125             output.append(y)
126     return output
127
128 def layer5_value_only_constant(weights, k_matrix):
129     # row_values = list(row.values())
130     output = []
131     for i in range(len(weights)):
132         w = weights[i]
133         k = k_matrix[i]
134         f = k[0]
135         # print(k)
136         y = f * w
137         output.append(y)
138     return output
139
140 def layer6_value(rule_values):
141     return sum(rule_values)
```

```
142
143
144 def A_matrix_final(weights, dimension, df, input_keys):
145     A_dimension = (len(weights), dimension['n']*(1 + dimension[
    'm']))
146     A_matrix = np.full(A_dimension, 0.0)
147
148     indexes = list(df.index)
149
150     for i in range(len(weights)):
151         sample = df.loc[indexes[i]]
152         weight_row = weights[i]
153         k = 0
154         for w in weight_row:
155             A_matrix[i][k] = round(w,4)
156             k+=1
157             for key in input_keys:
158                 A_matrix[i][k] = round(w * sample[key],4)
159                 k+=1
160
161     return A_matrix
```

Listing 8: ANFIS with modified Relief algorithm for WBC data set

5. Python code for diagnosis of breast cancer using Radial Basis Function Network

1. Python code for Ensemble LR-RBFN for WBC data set

```
1 import pandas as pd
2 import random
3 import helper
4 import datetime
5
6
7 class KMean:
8     def __init__(self, K, epoch=20, is_random=False):
9         self.K = K
10        self.epoch = epoch
11        self.is_random = is_random
12        self.input_data = []
13        self.centroid_data = []
14        self.classified_data = []
15        self.classification = {}
```

```
16          self.no_of_features = 0
17
18      def fit(self, input_data):
19          self.input_data = input_data
20          self.no_of_features = len(self.input_data[0])
21          if self.is_random:
22              self.centroid_data = random.sample(self.input_data.
    copy(), self.K)
23          else:
24              self.centroid_data = self.input_data.copy()[:self.K
    ]
25
26          it = 0
27          while it < self.epoch:
28              classification = []
29              for i in range(len(self.input_data)):
30                  sample = self.input_data[i]
31                  dist_values = []
32                  for j in range(len(self.centroid_data)):
33                      centroid = self.centroid_data[j]
34                      centroid_dist = helper.distance(sample,
    centroid)
35                      dist_values.append(centroid_dist)
36                  min_dist = min(dist_values)
37                  for j in range(len(dist_values)):
38                      if dist_values[j] == min_dist:
39                          classification.append(j)
40                          break
41              self.classified_data.append(classification)
42              if self.is_classified():
43                  break
44              self.update_centroid_data(classification.copy())
45              it += 1
46          return self.centroid_data
47
48      def get_centroid(self):
49          return self.centroid_data
50
51      def update_centroid_data(self, classification):
52          centroid_sum = []
53          for i in range(len(self.centroid_data)):
54              centroid_sum.append((0,) * len(self.centroid_data[i
    ]))
55              count = 0
56              for j in range(len(classification)):
```

```
                if classification[j] == i:
                    centroid_sum[i] = helper.vector_addition(centroid_sum[i], self.input_data[j])
                    count += 1
            else:
                if count > 1:
                    centroid_sum[i] = helper.scalar_vector_divide(centroid_sum[i], count)
        self.centroid_data = centroid_sum

    def update_centroid(self, dist_values, sample):
        min_dist = min(dist_values)
        for i in range(len(dist_values)):
            if dist_values[i] == min_dist:
                self.classified_data[-1].append(i)
                self.centroid_data[i] = helper.midpoint(self.centroid_data[i], sample)
                break

    def is_classified(self):
        if len(self.classified_data) < 2:
            return False
        current_classification = self.classified_data[-1]
        previous_classification = self.classified_data[-2]
        for curr, prev in zip(current_classification, previous_classification):
            if curr != prev:
                return False
        return True

    def classify(self):
        self.classification = {}
        for i in range(len(self.classified_data[-1])):
            if self.classified_data[-1][i] in self.classification:
                self.classification[self.classified_data[-1][i]].append(self.input_data[i])
            else:
                self.classification[self.classified_data[-1][i]] = [self.input_data[i]]

    def get_output(self):
        output_data = {}
        for i in range(self.no_of_features):
            output_data[f"input-{i+1}"] = [pair[i] for pair in
```

```
    self.input_data.copy()]

        distances = {}
        min_dist = []
        for i in range(len(self.centroid_data)):
            distances[f"centroid-{i+1}"] = []
            for j in range(len(self.input_data)):
                dist = helper.distance(self.centroid_data[i],
self.input_data[j])
                distances[f"centroid-{i+1}"].append(dist)
            output_data[f"centroid-{i+1}"] = distances[f"
centroid-{i+1}"]

        for i in range(len(self.input_data)):
            min_value = min([output_data[f"centroid-{j+1}"][i]
for j in range(len(self.centroid_data))])
            for j in range(len(self.centroid_data)):
                if min_value == output_data[f"centroid-{j+1}"][
i]:
                    min_dist.append(j)
                    break

        output_data["Min"] = min_dist

        cluster_data = {"Clusters": range(len(self.
centroid_data)),
                        "Count": [min_dist.count(i) for i in
range(len(self.centroid_data))]}
        for i in range(len(self.centroid_data)):
            for j in range(len(self.centroid_data[i])):
                if f"Feature-{j+1}" in cluster_data:
                    cluster_data[f"Feature-{j+1}"].append(self.
centroid_data[i][j])
                else:
                    cluster_data[f"Feature-{j+1}"] = [self.
centroid_data[i][j]]

        dt = datetime.datetime.now()
        out_df = pd.DataFrame(data=output_data)
        out_df.to_excel(f"Outputs/output_{dt.timestamp()}.xlsx"
, index=None)
        cluster_df = pd.DataFrame(data=cluster_data)
        cluster_df.to_excel(f"Outputs/cluster_output_{dt.
```

```
    timestamp()}.xlsx", index=None)
```

Listing 9: Ensemble LR-RBFN for WBC data set

```
import pandas as pd
import math
import numpy as np
from kmean import KMean
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import helper
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import datetime
import random


class RBFN:
    def __init__(self, no_of_pattern, biased=0, threshold=0, epoch=10, weight_rate=0.75,
                 centroid_rate=0.75, norm_std_ch=1, rbf_ch=1):
        self.no_of_pattern = no_of_pattern
        self.biased = biased
        self.threshold = threshold
        self.epoch = epoch
        self.learning_rate = {"weight": weight_rate, "centre": centroid_rate}
        if norm_std_ch == 2:
            self.standardize = True
            self.normalize = False
        else:
            self.standardize = False
            self.normalize = True
        self.rbf_ch = rbf_ch if 1<=rbf_ch<=3 else 1

        self.weightsComputed = False
        self.is_trained = False
        if self.standardize:
            self.x_scalar = StandardScaler()
        else:
            self.x_scalar = MinMaxScaler()
        self.x_scalar_fitted = False
        self.input_data = []
        self.weights = []
        self.centres = []
        self.no_of_samples = 0
```

```
41          self.k_mean = KMean(self.no_of_pattern, epoch=self.
    epoch)
42          self.error_vector = []
43          self.error = 0
44          self.errors = []
45          #self.lambda_values = {1: 0.03, 2: 0.02}
46          self.lambda_values = {1: 0.08, 2: 0.02}
47
48      def __preprocess_input(self, x_data):
49          self.input_data = []
50          input_values = [list(x.values()) for x in list(x_data.
    to_dict().values())]
51          for i in range(len(input_values)):
52              if i == 0:
53                  for value in input_values[i]:
54                      self.input_data.append([value])
55              else:
56                  for j in range(len(input_values[i])):
57                      self.input_data[j] = self.input_data[j] + [
    input_values[i][j]]
58          self.input_data = list(map(tuple, self.input_data))
59          self.no_of_features = len(self.input_data[0])
60
61          if self.x_scalar_fitted:
62              self.input_data = list(map(tuple, self.x_scalar.
    transform(self.input_data)))
63          else:
64              self.input_data = list(map(tuple, self.x_scalar.
    fit_transform(self.input_data)))
65              self.x_scalar_fitted = True
66          self.no_of_samples = len(self.input_data)
67
68      def __preprocess_output(self, y_data):
69          self.output_data = list(y_data)
70
71      def preprocess(self, x_data, y_data):
72          self.__preprocess_input(x_data)
73          self.__preprocess_output(y_data)
74          self.centres = self.k_mean.fit(self.input_data)
75          self.k_mean.get_output()
76
77      def __compute_rbf_matrix(self):
78          self.calculated_output = []
79          self.input_rbf_matrix = {}
80          self.spr_values = []
```

```
81
82          for i in range(len(self.centres)):
83              t_i = self.centres[i]
84              neighbours = helper.get_neighbours(t_i, self.
    centres.copy())
85              spr = helper.spread(t_i, neighbours)
86              self.spr_values.append(spr)
87              for j in range(len(self.input_data)):
88                  x_val = self.input_data[j]
89                  rbf_val = helper.calc_rbf(x_val, t_i, spr, self
    .rbf_ch)
90                  if self.input_rbf_matrix.get(j) is not None:
91                      self.input_rbf_matrix[j].append(rbf_val)
92                  else:
93                      self.input_rbf_matrix[j] = [rbf_val]
94
95      def __compute_weights(self):
96          rbf_matrix = list(self.input_rbf_matrix.values())
97
98          rbf_matrix_transpose = np.transpose(rbf_matrix)
99
100         rbf_transpose_dot_rbf = np.dot(rbf_matrix_transpose,
    rbf_matrix)
101
102         det_value = np.linalg.det(rbf_transpose_dot_rbf)
103
104         if det_value != 0:
105             rbf_transpose_dot_rbf_inv = np.linalg.inv(
    rbf_transpose_dot_rbf)
106
107             rbf_transpose_dot_output = np.dot(
    rbf_matrix_transpose, self.output_data)
108
109             weight_matrix = np.dot(rbf_transpose_dot_rbf_inv,
    rbf_transpose_dot_output)
110             # print("-------------Computed Weights
    -------------")
111             # print(weight_matrix)
112             # print
    ("-----------------------------------------")
113
114             self.weights = weight_matrix.copy()
115             self.weightsComputed = True
116         else:
117             raise Exception("Matrix Inverse is not possible!!
```

```
    Weights can not be computed")

    def __compute_output(self):
        for i in range(len(self.input_data)):
            output_val = 0
            rbf_values = self.input_rbf_matrix[i]
            for wei, rbf_val in zip(self.weights, rbf_values):
                output_val += (wei * rbf_val)
            else:
                output_val += self.biased

            if output_val > 0:
                self.calculated_output.append(1.0)
            else:
                self.calculated_output.append(0.0)

    def fit(self, x_train, y_train):
        self.preprocess(x_train, y_train)

        it = 0
        while it < self.epoch:
            print(f"------------Iteration: {it +
    1}---------------")

            self.__compute_rbf_matrix()

            # self.print_rbf()

            if not self.weightsComputed:
                # self.__compute_weights()
                self.weights = []
                for i in range(self.no_of_pattern):
                    self.weights.append(random.random()*2 - 1)
                self.weightsComputed = True
            print(self.weights)

            # print("-------------Computed Weights
    -------------")
            # print(self.weights)
            # print
    ("-----------------------------------------")

            self.__compute_output()

            # print("-------------------Output Data
```

```
----------------------")
            # print("Original Output: ", self.output_data)
            # print("Network Output: ", self.calculated_output)
            # print
("--------------------------------------------------")

            self.calc_error_vector()

            if self.is_trained:
                break

            self.back_propagate()

            it += 1

        if self.is_trained:
            print("Total Iterations: ", it + 1)
        else:
            print("Maximum Iterations Reached!! Model not
trained")

    def calc_error_vector(self):
        def calculate_error():
            e_sum = 0
            for e_i in self.error_vector:
                e_sum += math.pow(e_i, 2)
            return e_sum / self.no_of_samples

        def calc_lambda():
            return abs(self.lambda_values[1] * sum(self.weights
)) + (self.lambda_values[2] * sum([math.pow(x, 2) for x in
self.weights]))

        self.error_vector = []
        for i, j in zip(self.calculated_output, self.
output_data):
            # self.error_vector.append(i - j)
            self.error_vector.append(j - i)

        # print("----------------Error Vector--------------")
        # print(self.error_vector)
        # print("Total Missed Samples:", len([1 for i in self.
error_vector if i != 0]))
        # print("------------------------------------------")

```

```
197         self.error = calculate_error() + calc_lambda()
198         self.errors.append(self.error)
199         if self.error < self.threshold:
200             self.is_trained = True
201         print("####   Error: ", self.error, "  ####")
202 
203     def back_propagate(self):
204         def derivative_weight(centre_index):
205             error_rbf_sum = 0
206             for pair_index, error in zip(self.input_rbf_matrix,
     self.error_vector):
207                 val = error * self.input_rbf_matrix[pair_index
    ][centre_index]
208                 error_rbf_sum += val
209             ret_val = (-2 * error_rbf_sum) / self.no_of_samples
210             return ret_val
211 
212         def derivative_centre(centre_index, centre_selected):
213             if self.rbf_ch==1:
214                 sum_vector = (0, 0)
215                 for pair_index, error in zip(self.
    input_rbf_matrix, self.error_vector):
216                     val = error * self.input_rbf_matrix[
    pair_index][centre_index]
217                     val = val / math.pow(self.spr_values[
    centre_index], 2)
218                     sum_vector = helper.vector_addition(
    sum_vector,
219                                                         helper.
    scalar_vector_multiply(
220 
    helper.vector_subtract(self.input_data[pair_index],
221 
                         centre_selected), val))
222                 ret_val = helper.scalar_vector_multiply(
    sum_vector, (2 * self.weights[centre_index]) / self.
    no_of_samples)
223                 return ret_val
224             else:
225                 sum_vector = (0, 0)
226                 for pair_index, error in zip(self.
    input_rbf_matrix, self.error_vector):
227                     pair_selected = self.input_data[pair_index]
228                     diff = helper.vector_subtract(pair_selected
    , centre_selected)
```

```
229                     diff_dist = helper.distance(pair_selected,
    centre_selected)
230                     scalar_value = (error * self.weights[
    centre_index]) / self.input_rbf_matrix[pair_index][
    centre_index]
231                     diff_div_diff_dist = helper.
    scalar_vector_divide(diff, diff_dist)
232                     sub_result = helper.vector_multiply(helper.
    scalar_vector_multiply(diff, scalar_value),
    diff_div_diff_dist)
233                     sum_vector = helper.vector_addition(
    sum_vector, sub_result)
234                 ret_val = helper.scalar_vector_multiply(
    sum_vector, 2 / self.no_of_samples)
235                 return ret_val
236 
237         def sign(wmi):
238             if wmi<0:
239                 return -1
240             elif wmi>0:
241                 return 1
242             else:
243                 return random.random()*2 - 1
244 
245         if self.is_trained:
246             return
247 
248         updated_weights = []
249         updated_centres = []
250         for i in range(len(self.centres)):
251             # print(f"--------------------Back Iteration {i +
    1}------------------")
252             t_i = self.centres[i]
253             d_w = derivative_weight(i)
254             d_w = d_w + (2 * self.lambda_values[2] * self.
    weights[i]) + (self.lambda_values[1] * sign(self.weights[i
    ]))
255             # print("Derivative Weight: ", d_w)
256             delta_w = -1 * self.learning_rate["weight"] * d_w
257             # print("Delta Weight: ", delta_w)
258             new_weight = round(self.weights[i] + delta_w, 4)
259             updated_weights.append(new_weight)
260             d_c = derivative_centre(i, t_i)
261             # print("Derivative Centre: ", d_c)
262             delta_c = helper.scalar_vector_multiply(d_c, -1 *
```

```
    self.learning_rate["centre"])
263             # print("Delta Centre: ", delta_c)
264             new_centre = helper.vector_addition(t_i, delta_c)
265             updated_centres.append(new_centre)
266             # print
    ("-------------------------------------------------")
267
268         self.weights = updated_weights.copy()
269         self.centres = updated_centres.copy()
270         # self.print_final_output()
271
272     def predict(self, x_test):
273         self.__preprocess_input(x_test)
274         self.__compute_rbf_matrix()
275         self.__compute_output()
276
277         return self.calculated_output
278
279     def generate_accuracy_report(self, actual_values,
    predicted_values):
280         acc_score = accuracy_score(actual_values,
    predicted_values)
281         # print("Accuracy:", acc_score)
282         output_labels = list(set(actual_values))
283         conf_matrix = confusion_matrix(actual_values,
    predicted_values,
284                                        labels=output_labels)
285         print(conf_matrix)
286
287         tn, fp, fn, tp = conf_matrix.ravel()
288         testing_conf_mat = {
289             'True Positive': tp,
290             'False Negative': fn,
291             'False Positive': fp,
292             'True Negative': tn
293         }
294         # print(testing_conf_mat)
295
296         conf_report = classification_report(actual_values,
    predicted_values, labels=output_labels)
297         # print(conf_report)
298         conf_report_data = [z for z in [[y.strip() for y in x.
    strip().split("  ") if len(y) > 0] for x in str(conf_report
    ).split("\n")] if len(z) > 0]
299         conf_report_data[0].insert(0, "")
```

```
300         conf_report_data[3].insert(1, "")
301         conf_report_data[3].insert(2, "")
302 
303         output_data = {}
304         for i in range(len(conf_report_data)):
305             if i == 0:
306                 for label in conf_report_data[i]:
307                     output_data[label] = []
308             else:
309                 for j in range(len(conf_report_data[i])):
310                     output_data[conf_report_data[0][j]].append(
    conf_report_data[i][j])
311 
312         output_data[conf_report_data[0][0]].extend(["", ""] +
    list(testing_conf_mat.keys()) + ["", "Accuracy"])
313         output_data[conf_report_data[0][1]].extend(["", ""] +
    list(testing_conf_mat.values()) + ["", acc_score])
314         for i in range(2, len(conf_report_data[0])):
315             output_data[conf_report_data[0][i]].extend([""]*8)
316 
317         out_df = pd.DataFrame(data=output_data)
318         dt = datetime.datetime.now()
319         out_df.to_excel(f"Reports/accuracy_report_{dt.timestamp
    ()}.xlsx", index=None)
320 
321         return {
322             "accuracy": acc_score,
323             "conf_matrix": conf_matrix,
324             "report": conf_report
325         }
326 
327     def print_final_output(self):
328         print("------------------Final Output
    ---------------------")
329         print("Weights:", self.weights)
330         print("
    ---------------------------------------------------")
331         print("Centres:", self.centres)
332         print("
    ---------------------------------------------------")
333 
334     def print_rbf(self):
335         print("--------------------RBF Matrix
    ---------------------")
336         for key, value in self.input_rbf_matrix.items():
```

```
            print(key, "->", value)
        print("
    ------------------------------------------------")

    def error_report(self):
        out_df = pd.DataFrame(data={
            "Error": self.errors
        })
        dt = datetime.datetime.now()
        out_df.to_excel(f"Reports/Error_{dt.timestamp()}.xlsx",
    index=None)
```

Listing 10: Ensemble LR-RBFN for WBC data set for WBC data set

```
import pandas as pd
from algorithm import RBFN
import time

filename = input("Enter Input Filename: ")
try:
    df = pd.read_excel(filename)
except FileNotFoundError:
    print("Input File Not Found:", filename)
else:
    print("1. 90-10 Ratio (default)\n2. 80-20 Ratio\n3. 70-30
    Ratio\n")
    ratio_ch = int(input("Enter Choice: "))
    # ratio_ch = 1

    df_testing = df.sample(frac=[0.1, 0.2, 0.3][ratio_ch - 1 if
    ratio_ch in range(1, 4) else 0])
    df_training = df.drop(df_testing.index)

    # print("-----------Training Data----------")
    # print(df_training)

    # print("-----------Testing  Data----------")
    # print(df_testing)

    num_of_models = int(input("Enter Number of Models: "))

    accuracy_storage = list()

    start_time = time.time()

    try:
```

```
        i=0
        while i<num_of_models:
            print(f"------------------Enter Data for Model {i
+1}----------------------")
            no_of_pattern = int(input("Enter No of Patterns: ")
)

            biased = int(input("Enter Biased Value: "))

            threshold = float(input("Enter Threshold Value: "))

            epoch = int(input("Enter No of Max Iterations(Epoch
): "))

            print("1. Normalized(default)\n2. Standardized\n")
            norm_std_ch = int(input("Enter Choice: "))
            # norm_std_ch = 1

            print("1. Gaussian Function(default)\n2.
Multiquadratics\n3. Inverse Multiquadratics\n")
            rbf_ch = int(input("Select any one function from
above choice: "))
            # rbf_ch = 1

            algo = RBFN(no_of_pattern, biased=biased, threshold
=threshold, epoch=epoch, norm_std_ch=norm_std_ch, rbf_ch=
rbf_ch)

            data_df = df_training.sample(frac=(1/num_of_models)
)
            # print(data_df)

            x_train = data_df.iloc[:, :-1]
            y_train = data_df.iloc[:, -1]

            # Fit Model to Training Data
            algo.fit(x_train, y_train)

            print("Model Trained: ", algo.is_trained)

            # algo.error_report()

            if algo.is_trained:
                predicted_values = algo.predict(df_testing.iloc
[:, :-1])
```

```
                # print(predicted_values)
                actual_values = list(df_testing.iloc[:, -1])
                accuracy_data = algo.generate_accuracy_report(actual_values, predicted_values)
                accuracy_storage.append(accuracy_data)
                print("Accuracy:", accuracy_data["accuracy"], "\n")

            i+=1

        end_time = time.time()
        time_taken = end_time - start_time
        minutes = time_taken // 60
        seconds = time_taken % 60
    except Exception as e:
        print("Exception:", e)
    else:
        time.sleep(3)
        for i in range(len(accuracy_storage)):
            print(f"------------------------Accuracy Report for Model {i+1}----------------------")
            print("Accuracy:", accuracy_storage[i]["accuracy"])
            print(accuracy_storage[i]["conf_matrix"])
            print(accuracy_storage[i]["report"])
            print("\n")
        print(f"Time Taken: {minutes} minutes {seconds} seconds\n")
```

Listing 11: Ensemble LR-RBFN for WBC data set for WBC data set

```
import math


def distance(p1, p2):
    dist = 0
    for i, j in zip(p1, p2):
        diff = i - j
        dist += math.pow(diff, 2)
    return math.sqrt(dist)


def norm(p1, p2):
    norm_val = 0
    for i, j in zip(p1, p2):
        norm_val += math.pow(i - j, 2)
    return norm_val
```

```
17
18
19 def calc_rbf(x_i, t_i, spr, rbf_ch):
20     if rbf_ch==1:
21         return math.exp((-1 * norm(x_i, t_i)) / (2 * math.pow(
   spr, 2)))
22     elif rbf_ch==2:
23         return math.sqrt(math.pow(spr, 2) + norm(x_i, t_i))
24     else:
25         return 1 / math.sqrt(math.pow(spr, 2) + norm(x_i, t_i))
26
27
28 def get_neighbours(t_i, t_list):
29     if t_i in t_list:
30         t_list.remove(t_i)
31     distances = dict()
32     for t_k in t_list:
33         distances[tuple(t_k)] = distance(t_i, t_k)
34     distances = {k: v for k, v in distances.items() if v <= min
   (distances.values())}
35     return distances
36
37
38 def spread(t_i, neighbours):
39     norm_sum = 0
40     for t_k in neighbours:
41         norm_sum += norm(t_i, t_k)
42     return math.sqrt(norm_sum / len(neighbours))
43
44
45 def print_centroid(centroids):
46     print("--------------------")
47     for centroid in centroids:
48         print(centroid)
49     print("--------------------")
50
51
52 def preprocess(input_dataset):
53     samples = []
54     for i in input_dataset.index:
55         sample_i = input_dataset.loc[i]
56         samples.append(tuple(sample_i.to_dict().values()))
57     return samples
58
59
```

```
def vector_addition(v1, v2):
    result_vector = []
    for v_i, v_j in zip(v1, v2):
        result_vector.append(v_i + v_j)
    return tuple(result_vector)


def vector_subtract(v1, v2):
    result_vector = []
    for v_i, v_j in zip(v1, v2):
        result_vector.append(v_i - v_j)
    return tuple(result_vector)

def vector_multiply(v1, v2):
    result_vector = []
    for v_i, v_j in zip(v1, v2):
        result_vector.append(v_i * v_j)
    return tuple(result_vector)

def scalar_vector_multiply(vec, k):
    result_vector = []
    for v_i in vec:
        result_vector.append(v_i * k)
    return tuple(result_vector)


def scalar_vector_divide(vec, k):
    result_vector = []
    for v_i in vec:
        result_vector.append(v_i / k)
    return tuple(result_vector)


def midpoint(v1, v2):
    result_vector = []
    for v_i, v_j in zip(v1, v2):
        mid = (v_i + v_j) / 2
        result_vector.append(mid)
    return tuple(result_vector)
```

Listing 12: Ensemble LR-RBFN for WBC data set for WBC data set

6. Python code for diagnosis of breast cancer using Novel RBF kernel with PSO

1. **Python code for diagnosis of breast cancer using Novel RBF kernel with PSO**

```
import pandas as pd
import math
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import datetime


class RBFN:
    def __init__(self, df, weights, biased=1, threshold=0):
        self.df = df
        self.weights = weights
        self.biased = biased
        self.threshold = threshold
        self.no_of_inputs = len(df)
        self.no_of_pattern = len(df)

        self.learning_rate = {"weight": 0.5, "centre": 0.1}
        self.no_of_variables=1
        self.population_size=4
        self.intial_weights=0.5
        self.c1=1
        self.c2=1

        self.r1=0.1
        self.r2=0.2
        self.n_particles=len(x_train)
        self.intial_velocities=[0,0,0,0]
        self.intial_particles_pos=weights
        self.p_best = self.intial_particles_pos
        self.g_best=''
        self.updated_fitness=[0]*self.n_particles
        self.updated_particles_pos=[0]*self.n_particles
        self.updated_p_best=[0]*self.n_particles
        self.update_velocities=[0]*self.n_particles
        self.calculated_output=[0]*self.n_particles
        self.rbf_matrix=[]

    def input_preprocess(self,x_train):
        self.input_data = []
```

```
        # input_values = list(self.input_data_dict.values())
        input_values = [list(x.values()) for x in list(x_train.iloc[:, :-1].to_dict().values())]
        for i in range(len(input_values)):
            if i==0:
                for value in input_values[i]:
                    self.input_data.append([value])
            else:
                for j in range(len(input_values[i])):
                    self.input_data[j] = self.input_data[j] + [input_values[i][j]]
        self.x_scalar = StandardScaler()
        print(self.input_data)
        self.input_data = list(map(tuple, self.x_scalar.fit_transform(self.input_data)))

        # self.input_data = list(map(tuple, self.input_data))
        # print(self.input_data)
        self.centres = self.input_data.copy()
        # print(self.centres)

    def preprocess(self,x_train,y_train):
        # print([list(x.values()) for x in list(self.df.iloc[:, :-1].to_dict().values())])
        # self.input_data_dict = {x:list(y.values()) for x,y in self.df.iloc[:,:-1].to_dict().items()}
        self.output_data = list(y_train)
        self.input_data = []
        # input_values = list(self.input_data_dict.values())
        input_values = [list(x.values()) for x in list(x_train.iloc[:, :-1].to_dict().values())]
        for i in range(len(input_values)):
            if i==0:
                for value in input_values[i]:
                    self.input_data.append([value])
            else:
                for j in range(len(input_values[i])):
                    self.input_data[j] = self.input_data[j] + [input_values[i][j]]
        self.x_scalar = StandardScaler()
        print(self.input_data)
        self.input_data = list(map(tuple, self.x_scalar.fit_transform(self.input_data)))

        # self.input_data = list(map(tuple, self.input_data))
```

```
80          # print(self.input_data)
81          self.centres = self.input_data.copy()
82          # print(self.centres)
83
84      def generate_accuracy_report(self, actual_values,
    predicted_values):
85          acc_score = accuracy_score(actual_values,
    predicted_values)
86          print("Accuracy:", acc_score)
87          output_labels = list(set(actual_values))
88          conf_matrix = confusion_matrix(actual_values,
    predicted_values,
89                                         labels=output_labels)
90          print(conf_matrix)
91
92          tn, fp, fn, tp = conf_matrix.ravel()
93          testing_conf_mat = {
94              'True Positive': tp,
95              'False Negative': fn,
96              'False Positive': fp,
97              'True Negative': tn
98          }
99          print(testing_conf_mat)
100
101         conf_report = classification_report(actual_values,
    predicted_values, labels=output_labels)
102         print(conf_report)
103         conf_report_data = [z for z in [[y.strip() for y in x.
    strip().split("  ") if len(y) > 0] for x in str(conf_report
    ).split("\n")] if len(z) > 0]
104         conf_report_data[0].insert(0, "")
105         conf_report_data[3].insert(1, "")
106         conf_report_data[3].insert(2, "")
107
108         output_data = {}
109         for i in range(len(conf_report_data)):
110             if i == 0:
111                 for label in conf_report_data[i]:
112                     output_data[label] = []
113             else:
114                 for j in range(len(conf_report_data[i])):
115                     output_data[conf_report_data[0][j]].append(
    conf_report_data[i][j])
116
117         output_data[conf_report_data[0][0]].extend(["", ""] +
```

```
    list(testing_conf_mat.keys()) + ["", "Accuracy"])
118         output_data[conf_report_data[0][1]].extend(["", ""] +
    list(testing_conf_mat.values()) + ["", acc_score])
119         for i in range(2, len(conf_report_data[0])):
120             output_data[conf_report_data[0][i]].extend([""]*8)
121
122         out_df = pd.DataFrame(data=output_data)
123         dt = datetime.datetime.now()
124         out_df.to_excel(f"Reports/accuracy_report_{dt.timestamp
    ()}.xlsx", index=None)
125
126
127
128     def print(self):
129         print("Weights: ", self.weights)
130         print("Centres: ", self.centres)
131
132     def update_velocity(self, x, v, p_best, g_best, c0=1, c1=1,
     w=0.5):
133
134             x = np.array(x)
135             v = np.array(v)
136
137             r1 = 0.1
138             r2 = 0.2
139             p_best = np.array(p_best)
140             g_best = np.array(g_best)
141             # print(x)
142             # print(v)
143             # print("p_best:",p_best)
144             # print("g_best:",g_best)
145             new_v = w*v + c0 * r1 * (p_best - x) + c1 * r2 * (
    g_best - x)
146             new_v = round(new_v,2)
147             return new_v
148
149     def update_position(self, x, v):
150
151             x = np.array(x)
152             v = np.array(v)
153
154             new_x = x + v
155             new_x = round(new_x,2)
156             return new_x
157
```

```
158     def update_velocity(self, x, v, p_best, g_best, c0=1, c1=1,
        w=0.5):
159
160             x = np.array(x)
161             v = np.array(v)
162
163             r1 = 0.1
164             r2 = 0.2
165             p_best = np.array(p_best)
166             g_best = np.array(g_best)
167             # print(x)
168             # print(v)
169             # print("p_best:",p_best)
170             # print("g_best:",g_best)
171             new_v = w*v + c0 * r1 * (p_best - x) + c1 * r2 * (
    g_best - x)
172             new_v = round(new_v,2)
173
174             return new_v
175
176     def calculate_fitness(self,output_data,velocities,
    rbf_matrix):
177
178             sum=0
179
180             for i in range(len(velocities)):
181               sum += (velocities[i]*rbf_matrix[i])
182
183             new_fitness=(output_data-sum)**2
184             new_fitness=round(new_fitness,2)
185
186             return new_fitness
187
188     def compare(self,old,new,old_postion,new_postion):
189             # print("old new fitness:",old,new)
190             # print("old new postion:",old_postion,new_postion)
191             result=[]
192             for i in range(len(old)):
193               min_index = np.argmin([old[i],new[i]])
194               if min_index == 0:
195                 result.append(old_postion[i])
196               elif min_index == 1:
197                 result.append(new_postion[i])
198
199             return result
```

```
200
201
202     def after_updated_weight_calculation(self,updated_p_best,
    rbf_values):
203         print("Updated Weights:",updated_p_best)
204         updated_val = 0
205         calculated_output=[]
206         for j in range(len(rbf_values)):
207
208             for i in range(len(updated_p_best)):
209               updated_val += (updated_p_best[i]*rbf_values[j][i
    ])
210
211             if updated_val > self.threshold:
212                 calculated_output.append(1)
213             else:
214                 calculated_output.append(0)
215
216
217         return calculated_output
218
219     def calculate_error():
220         e_sum = 0
221         for e_i in self.error_vector:
222             e_sum += e_i
223         return math.pow(e_sum, 2) / self.no_of_pattern
224
225
226
227     def optimize(self, maxiter=5):
228
229
230         for _ in range(maxiter):
231
232           print("Iteration ",_)
233           if _ != 0:
234             self.intial_fitness=self.updated_fitness
235             self.intial_particles_pos=self.
    updated_particles_pos
236             self.p_best=self.updated_p_best
237             self.intial_velocities=self.update_velocities
238
239           for i in range(self.n_particles):
240
241             v=self.intial_velocities[i]
```

```
242             #Update velocity
243 
244             self.update_velocities[i] = self.update_velocity(
    self.intial_particles_pos[i], v, self.p_best[i], self.
    g_best)
245 
246             #Update particle position
247             self.updated_particles_pos[i] = self.
    update_position(self.intial_particles_pos[i], self.
    intial_velocities[i])
248 
249         print("Updated velocity:",self.update_velocities)
250         print("Updated Position:",self.updated_particles_pos)
251 
252 
253         for i in range(self.n_particles):
254           self.updated_fitness[i]=self.calculate_fitness(self
    .output_data[i],self.update_velocities,self.rbf_matrix[i])
255 
256         print("Updated fitness:",self.updated_fitness)
257         gbest_index = np.argmin(self.updated_fitness)
258 
259         self.g_best = self.updated_particles_pos[gbest_index]
260         print("Updated gbest:",self.g_best)
261 
262         self.updated_p_best=self.compare(self.intial_fitness,
    self.updated_fitness,self.intial_particles_pos,self.
    updated_particles_pos)
263         print("Updated pbest:",self.updated_p_best)
264 
265         self.calculated_output=self.
    after_updated_weight_calculation(self.updated_p_best,self.
    rbf_matrix)
266         print("Network Output",self.calculated_output)
267         print("Original Output:",self.output_data)
268         self.error_vector = []
269         for i, j in zip(self.calculated_output, self.
    output_data):
270             self.error_vector.append(abs(i-j))
271 
272         # self.error = calculate_error()
273         print("Network Error:",self.error_vector)
274         e_sum = 0
275         for e_i in self.error_vector:
276           e_sum += e_i
```

```
277             error= math.pow(e_sum, 2) / self.no_of_pattern
278             print("Network error after Iteration:",error)
279 
280     def distance1(self,x, y):
281             return math.sqrt(math.pow(y[0] - x[0], 2) + math.
    pow(y[1] - x[1], 2))
282 
283     def norm1(self,x, y):
284             return math.pow((y[0] - x[0]), 2) + math.pow((y[1]
    - x[1]), 2)
285 
286     def rbf1(self,x, t, spr):
287             sigma=(2 * math.pow(spr, 2))
288             if sigma == 0.0:
289                 rbf=1
290             else:
291                 rbf=math.exp((-1 * self.norm1(x, t)) / sigma)
292             return rbf
293 
294     def spread1(self,t_i, neighbours):
295         sum = 0
296         # print(t_i,neighbours)
297         for t_k in neighbours:
298             norm_val = self.norm1(t_i, t_k)
299             sum += norm_val
300         return math.sqrt(sum / len(neighbours))
301 
302 
303 
304 
305 
306     def get_neighbours1(self,t_i, t_list):
307             if t_i in t_list:
308                 t_list.remove(t_i)
309             distances = dict()
310             for t_k in t_list:
311                 distances[tuple(t_k)] = self.distance1(t_i, t_k
    )
312             # print(distances)
313             distances = {k:v for k,v in distances.items() if v
    <= min(distances.values())}
314             return distances
315 
316     def calculate_test_data(self):
317 
```

```
318         self.calculated_output = []
319         self.input_rbf_matrix = {}
320         self.spr_values = []
321         self.output_val=[]
322         # print("No: ", len(self.centres))
323         for i in range(len(self.centres)):
324             # print("i: ", i)
325             # print("No inside: ", len(self.centres))
326             t_i = self.centres[i]
327             neighbours = self.get_neighbours1(t_i, self.
    input_data.copy())
328             # print("neighbours - ", neighbours)
329             spr = self.spread1(t_i, neighbours)
330             # print("spread - ", spr)
331             self.spr_values.append(spr)
332             rbf_values = []
333             for j in range(len(self.input_data)):
334                 x_val = self.input_data[j]
335                 # print(x_val,t_i,spr)
336                 rbf_val = self.rbf1(x_val, t_i, spr)
337                 # rbf_val=round(rbf_val,2)
338 
339                 if self.input_rbf_matrix.get(x_val) is not None
    :
340                     self.input_rbf_matrix[x_val].append(rbf_val
    )
341                 else:
342                     self.input_rbf_matrix[x_val] = [rbf_val]
343                 rbf_values.append(rbf_val)
344 
345 
346             self.rbf_matrix.append(rbf_values)
347 
348             output_val = 0
349 
350             for wei, rbf_val in zip(self.weights, rbf_values):
351                 output_val += (wei * rbf_val)
352                 # self.output_val.append(output_val)
353             # else:
354             #     output_val += self.biased
355             self.output_val.append(output_val)
356             if output_val > self.threshold:
357                 self.calculated_output.append(1)
358             else:
359                 # print("Output: 0")
```

```
                self.calculated_output.append(0)
        # print(self.input_rbf_matrix)
        print("Testing Network Output",self.calculated_output)
        return self.calculated_output



    def predict(self,x_test):
        self.input_preprocess(x_test)
        result=self.calculate_test_data()

        return result

    def train(self):
        def distance(x, y):
            return math.sqrt(math.pow(y[0] - x[0], 2) + math.
    pow(y[1] - x[1], 2))

        def norm(x, y):
            return math.pow((y[0] - x[0]), 2) + math.pow((y[1]
    - x[1]), 2)

        def rbf(x, t, spr):
            sigma=(2 * math.pow(spr, 2))
            if sigma == 0.0:
                rbf=1
            else:
                rbf=math.exp((-1 * norm(x, t)) / sigma)
            return rbf

        def get_neighbours(t_i, t_list):
            if t_i in t_list:
                t_list.remove(t_i)
            distances = dict()
            for t_k in t_list:
                distances[tuple(t_k)] = distance(t_i, t_k)
            # print(distances)
            distances = {k:v for k,v in distances.items() if v
    <= min(distances.values())}
            return distances

        def spread(t_i, neighbours):
            sum = 0
            # print(t_i,neighbours)
            for t_k in neighbours:
```

```
                norm_val = norm(t_i, t_k)
                sum += norm_val
            return math.sqrt(sum / len(neighbours))

        self.calculated_output = []
        self.input_rbf_matrix = {}
        self.spr_values = []
        self.output_val=[]
        # print("No: ", len(self.centres))
        for i in range(len(self.centres)):
            # print("i: ", i)
            # print("No inside: ", len(self.centres))
            t_i = self.centres[i]
            neighbours = get_neighbours(t_i, self.input_data.
copy())
            # print("neighbours - ", neighbours)
            spr = spread(t_i, neighbours)
            # print("spread - ", spr)
            self.spr_values.append(spr)
            rbf_values = []
            for j in range(len(self.input_data)):
                x_val = self.input_data[j]
                # print(x_val,t_i,spr)
                rbf_val = rbf(x_val, t_i, spr)
                # rbf_val=round(rbf_val,2)

                if self.input_rbf_matrix.get(x_val) is not None
:
                    self.input_rbf_matrix[x_val].append(rbf_val
)
                else:
                    self.input_rbf_matrix[x_val] = [rbf_val]
                rbf_values.append(rbf_val)


            self.rbf_matrix.append(rbf_values)

            output_val = 0

            for wei, rbf_val in zip(self.weights, rbf_values):
                output_val += (wei * rbf_val)
                # self.output_val.append(output_val)
            # else:
            #     output_val += self.biased
            self.output_val.append(output_val)
```

```
            if output_val > self.threshold:
                self.calculated_output.append(1)
            else:
                # print("Output: 0")
                self.calculated_output.append(0)
        # print(self.input_rbf_matrix)
        print("Network Output",self.calculated_output)
        # print("--------------------RBF Matrix
    ---------------------")
        # for key, value in self.input_rbf_matrix.items():
        #     print(key, "->", value)
        # print
    ("-------------------------------------------------")

     def calc_error_vector(self):
        def calculate_error():
            e_sum = 0
            for e_i in self.error_vector:
                e_sum += e_i
            return math.pow(e_sum, 2) / self.no_of_pattern

        self.is_trained = True
        self.error_vector = []
        for i, j in zip(self.calculated_output, self.
    output_data):
            self.error_vector.append(abs(i-j))
            if i-j!=0:
                self.is_trained = False
        # print("----------------Error Vector--------------")
        # print(self.error_vector)
        # print("------------------------------------------")
        if not self.is_trained:
            self.error = calculate_error()
        print("Network Total Error:",self.error)

        self.intial_velocities=[round(i * 0.1,2) for i in self.
    weights]
        print("Intial velocity",self.intial_velocities)

        self.intial_fitness=[]
        for output, network_ouput in zip(self.output_data, self
    .output_val):

                fitness_val=(output-network_ouput)**2
                fitness_val=round(fitness_val,2)
```

```
                self.intial_fitness.append(fitness_val)

        print("Intial Fitness value:",self.intial_fitness)
        min_index = np.argmin(self.intial_fitness)
        # print("asdsadsd",min_index)
        self.g_best=self.p_best[min_index]

        print("Intial gbest",self.p_best[min_index])









import pandas as pd


df = pd.read_excel(r'/home/pragnakalpl7/Documents/Dialogflow/
    Classification/RBFN_24_Sep_2021/RBFN_16_Oct_2021/Inputs/
    input_9.xlsx')

df_testing = df.sample(frac=0.2)
df_training = df.drop(df_testing.index)


x_train = df_training.iloc[:, :-1]
y_train = df_training.iloc[:, -1]

data = [[0,0,0], [0,1,1], [1,0,1],[1,1,0]]

# df = pd.DataFrame(data, columns=['X', 'Y','Output'])
# print(df)
weights = [-2.3,1.7,1.3,-0.5]

import numpy as np
sampl = np.random.uniform(low=0, high=1, size=(699,))
# print(sampl)
sampl = np.round(sampl, decimals=2)
# print(sampl)
# print(list(sampl))

weights=list(sampl)


```

```
biased = -2.7

algo = RBFN(df, weights, biased=biased)


algo.preprocess(x_train,y_train)
# algo.print()

epoch = 1
i=0
while i<epoch:
    algo.train()
    algo.calc_error_vector()
    algo.optimize()

    # algo.print()
    # print("Trained: ", algo.is_trained)
    if algo.is_trained:
        print("Total Iterations: ", (i+1))
        break
    # else:
    #     algo.backpropagate()
        # algo.print()
    i+=1


predicted_values = algo.predict(df_testing.iloc[:, :-1])

actual_values = list(df_testing.iloc[:, -1])

print(predicted_values)

algo.generate_accuracy_report(actual_values, predicted_values)
```

Listing 13: Novel RBF kernel with PSO