Binary Classification with k-Nearest Neighbors, Neighborhood-based Classifiers and Single Layer Perceptron Model

Shivchander Sudalairaj
Intelligent Systems (EECS 6036)
University of Cincinnati

October 2020

Abstract

Binary classification is the task of classifying the elements of a set into two groups on the basis of a classification rule. For this experiment, we are considering the Stressed/Not Stressed dataset. The goal of the study is to assess if these individuals are stressed or not after the change. We then build and compare the performance of k-Nearest neighbor, Neighborhood and Single Layer Perceptron models.

1 Performance of kNN and Neighborhood Classifier

1.1 Results

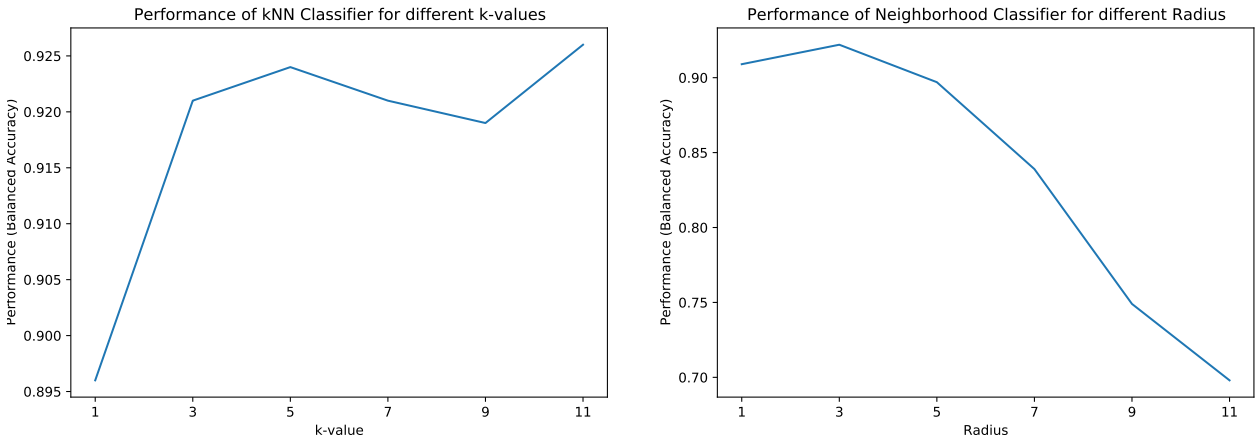

Figure 1.1: *Plot of balanced accuracy based on various k-values for the kNN Classifier*

Figure 1.2: *Plot of balanced accuracy based on various R-values for the Neighborhood-based Classifier*

1.2 Discussion

From Figure 1.1 it can be observed that the optimal k-value for the kNN Classifier is when $k=11$, as the balanced accuracy is the highest for that value. Similarly from Figure 1.2, it can be observed that the optimal R-value for the Neighborhood-based Classifier is when $R=3$.

2 Performance of Perceptron

2.1 Results

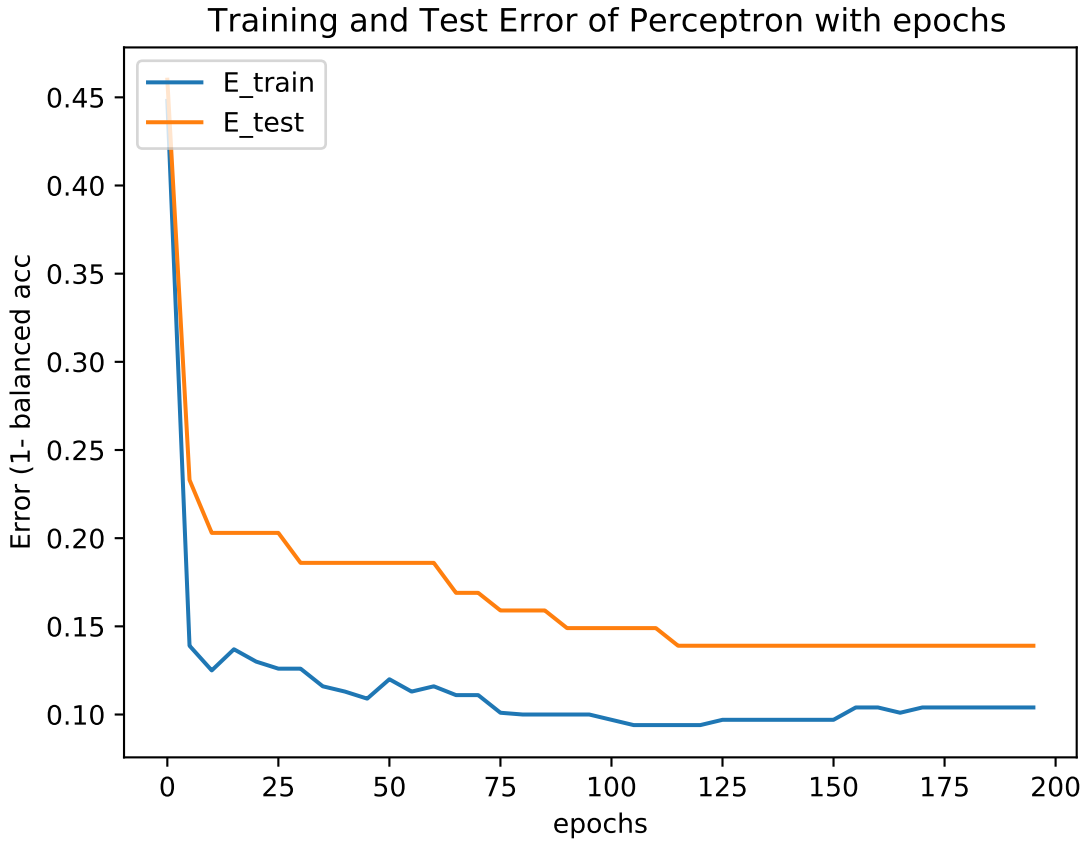

Figure 2.1: *Plot of Error (train, test) based on each epoch for the Perceptron*

2.2 Discussion

When multi-layer perceptron is used, in this case, each hidden unit will get exactly the same signal. E.g. if all weights are initialized to 1, each unit gets signal equal to sum of inputs. If all weights are zeros, which is even worse, every hidden unit will get zero signal. No matter what was the input - if all weights are the same, all units in hidden layer will be the same too. Thus weights for the Perceptron was initialized randomly in order to break the symmetry caused by zero or one initialization. The learning rate was determined after running a grid search on *lr = (0.1, 0.01, 0.001, 0.0001)* and based on the performance, *lr=0.001* was used.

From Figure 2.1 it can be observed that Error rate keeps decreasing as the training progresses and eventually stays constant after 100^{th} epoch. Since both training and testing curves are both decreasing in same trend, we can assure that the perceptron is learning and generalizing well on real world data. It can also be observed that the test error curve flatlines after epoch 100. We can thus stop our training at that epoch.

3 Comparison of Classifiers

Using the best values of parameters for all three classifiers, 9 independent trials with each algorithm was performed. For each trial, 20% of data from each class was set as the test set and the remaining as the training set.

3.1 Performance on Individual Trials

For each trial, each of the model was evaluated using the following metrics: Balanced Accuracy, Precision, Recall, F1 Score.

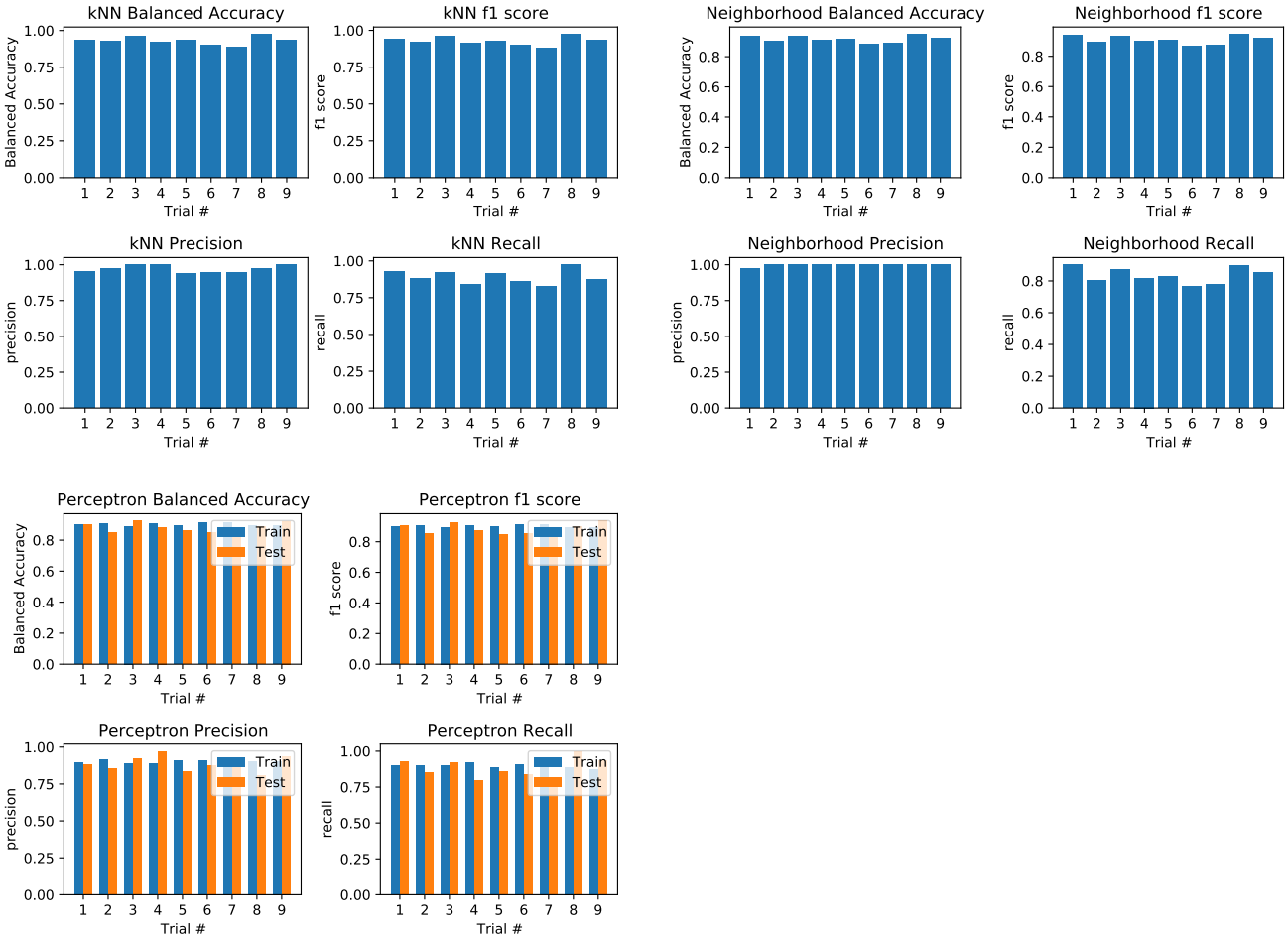

Figure 3.1: *Plot of Metrics of kNN Classifier for various trials*

Figure 3.2: *Plot of Metrics of Neighborhood-based Classifier for various trials*

Figure 3.3: *Plot of Metrics of Perceptron for various trials*

3.2 Average Performance

	kNN	Neighborhood	Perceptron
Balanced Accuracy	0.931 ±0.027	0.916 ±0.022	0.881 ±0.029
Precision	0.97 ±0.025	0.997 ±0.009	0.89 ±0.052
Recall	0.891 ±0.047	0.836 ±0.049	0.882 ±0.068
F1	0.928 ±0.028	0.909 ±0.027	0.883 ±0.033

Table 1: Mean values of balanced accuracy, precision, recall, and F1 score for each algorithm on testing data averaged over all 9 trials, along with the standard deviation for each case.

3.3 Trial-Wise Training Error Time-Series for the Perceptrons

Figure 3.4: *Plot showing the error on the training set plotted against epoch over the duration of training for the 9 trials*

3.4 Mean Training Error for the Perceptron

Figure 3.5: *Plot showing showing the mean training error averaged over the 9 trials plotted against time.*

3.5 kNN Decision Boundary

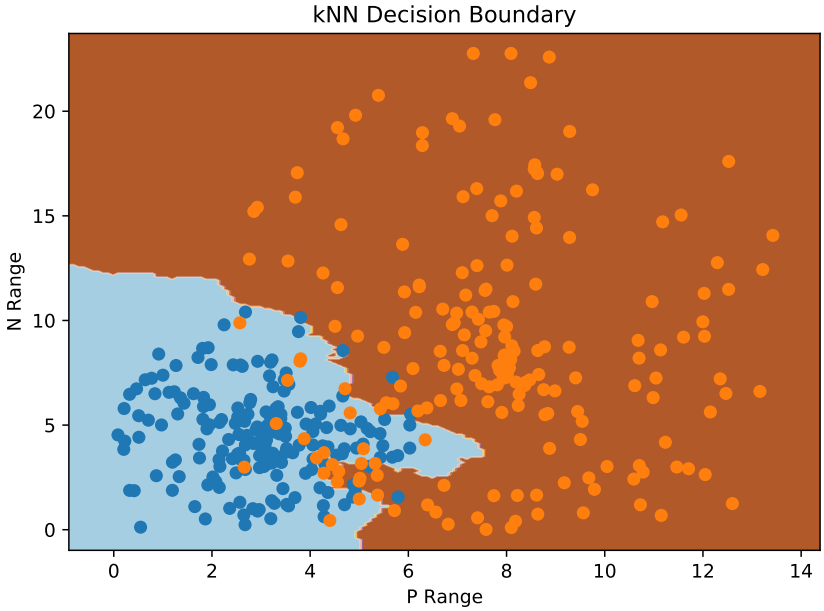

Figure 3.6: *Plot of the P-N feature space indicating the decision boundaries found by the kNN classifier in the best trial*

3.6 Neighborhood-based Decision Boundary

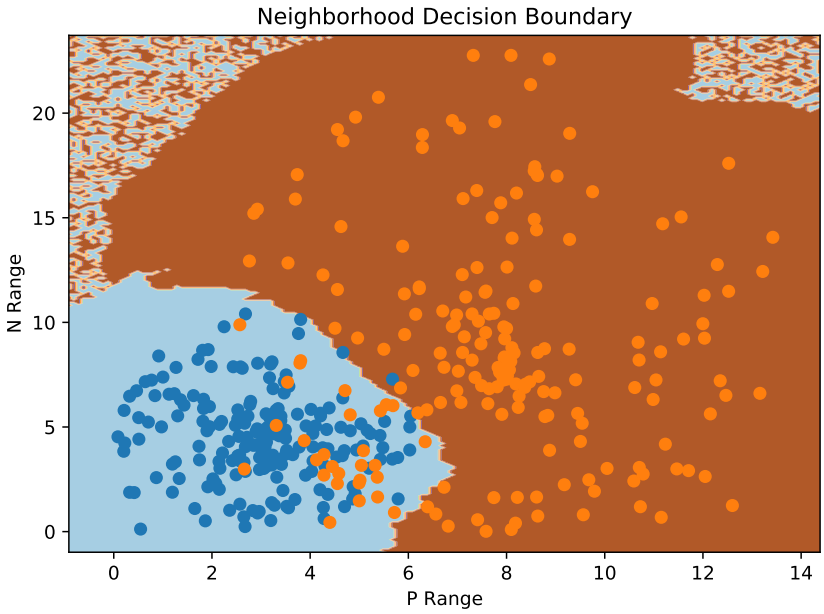

Figure 3.7: *Plot of the P-N feature space indicating the decision boundaries found by the Neighborhood-based classifier in the best trial*

3.7 Perceptron Decision Boundary

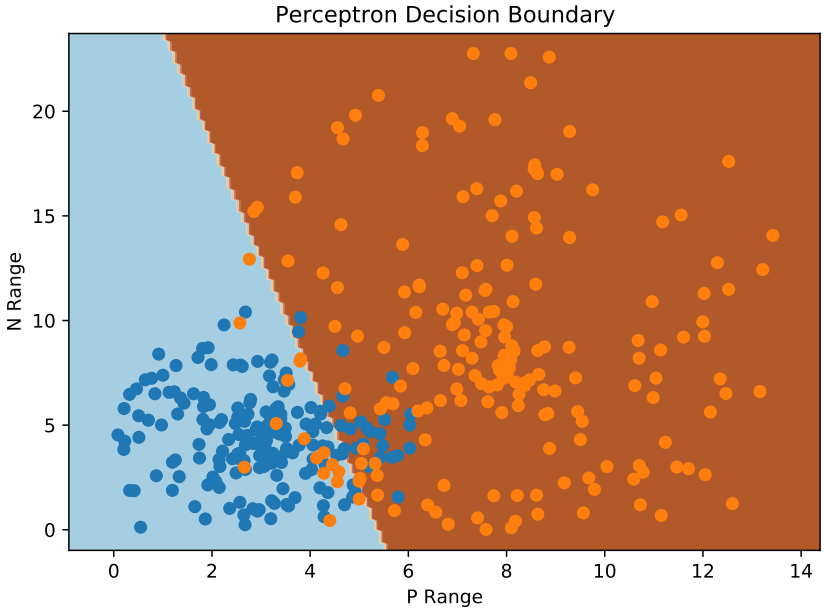

Figure 3.8: *Plot of the P-N feature space indicating the decision boundaries found by the Perceptron in the best trial*

3.8 Analysis of Results

From Figures 3.1, 3.2, 3.3, we can observe that all three models perform fairly well for the Stressed/Not Stressed classification. This is assured by their average performance (Table 1) across 9 random trials. Amongst the three classifiers, kNN has the best balanced accuracy score and F1 score, with close to 93% accuracy. This could be due to the simple implementation or the logic of the kNN algorithm. From Figures 3.6, 3.7 and 3.8, we can observe that the given dataset is fairly easy to separate with a linear classifier; since the classes of data points has a natural separation. And kNN clearly has the best decision boundary for this particular dataset, but it does not generalize well. The boundary is overfit to this dataset, while the Perceptron has a more generalized linear decision boundary.

Both kNN and Neighborhood based classifiers have the advantage of not having a training period. As both the models are lazy learners or instance based learners which uses memory of the train dataset to make predictions on the test set. This also makes adding new data to these models much more simpler.

But these models do not perform well when the dataset size is large, as the cost for calculating the distances to all points will be fairly expensive. These instance based classifiers are also sensitive to noise, outliers can skew the predictions made by the classifier.

Perceptrons learn a more general and linear decision boundary. They do not overfit the data unlike instance based learners. If the data is known to be linearly seperable, perceptrons are guranteed to converge. Perceptrons are much more suited to handle noise in the data unlike kNN or neighborhood based classifiers.

Perceptrons can not learn from data with complex distributions or classes with polynomial decision boundaries. They are restricted to linear discriminants. Perceptrons can not be used for multi-class classification. Unlike Instance based learners like kNN, perceptrons need a training phase which might take a while and be compute costly if the dataset is huge.

kNN classification is simplest to understand for its implementation. It works by measuring the distance between a group of data points defined by the value of k. For problems where all the data points are well defined or contains less non-linearity like the given data, then kNN is a go to algorithm as it is easy to implement and has no complex parameter regulation tasks. So for this particular dataset, kNN is the best suited as it has the best accuracy and F1 score. But it is a poor model considered for larger datasets. As it calculates everything from the ground on for every new prediction. But Perceptrons are recommended as they do not overfit (Figure 3.7 and 3.8) the data and are more generalized than kNN.

4 Code Appendix

For futher reference and code implementation for this paper see : `https://github.com/shivchander/binary-classifiers`

4.1 knn.py

```
#!/usr/bin/env python3
__author__ = "Shivchander Sudalairaj"
__license__ = "MIT"

'''
kNN Classifier
'''
from scipy.spatial import distance


class KNN:
    def __init__(self, X, y):
        self.X = X                              # data features
        self.y = y                              # labels

    def find_neighbor_distances(self, x_test):
        neighbors = []
        for xi, yi in zip(self.X, self.y):
            dist = distance.euclidean(xi, x_test)
            if dist != 0.0:                         # skipping distance from itself
                neighbors.append((dist, yi))

        neighbors.sort(key=lambda tup: tup[0])      # sorting the distance in ascending order
        return neighbors

    def predict(self, x_test, k):
        distances = self.find_neighbor_distances(x_test)          # calc the distance from all o
        # get k nearest neighbors (least distances)
        k_neighbors = distances[:k]
        # get the most frequent label class from nearest neighbors
        output_labels = list(dict(k_neighbors).values())
        predicted_label = max(set(output_labels), key=output_labels.count)
        return predicted_label
```

4.2 neighborhood.py

```
#!/usr/bin/env python3
__author__ = "Shivchander Sudalairaj"
__license__ = "MIT"

'''
Neighborhood based Classifier
```

```
'''
from scipy.spatial import distance
import random


class NeighborhoodClassifier:
    def __init__(self, X, y):
        self.X = X              # data features
        self.y = y              # labels

    def find_neighborhood(self, x_test, R):

        neighborhood = []
        for xi, yi in zip(self.X, self.y):
            dist = distance.euclidean(xi, x_test)
            if dist <= R:                       # lies inside the circular neighborhood
                if dist != 0.0:                 # skipping distance from itself
                    neighborhood.append((dist, yi))

        neighborhood.sort(key=lambda tup: tup[0])  # sorting the distance in ascending order
        return neighborhood

    def predict(self, x_test, R):
        neighbors = self.find_neighborhood(x_test, R)          # Finds all the neighbors in the
        # get the most frequent label class from nearest neighbors
        output_labels = list(dict(neighbors).values())
        if len(output_labels) == 0:                          # there are no neighbors in the neig
            predicted_label = random.randint(0,1)            # making a random prediction
        else:
            predicted_label = max(set(output_labels), key=output_labels.count)
        return predicted_label
```

4.3 perceptron.py

```
#!/usr/bin/env python3
__author__ = "Shivchander Sudalairaj"
__license__ = "MIT"

'''
Perceptron classifier
'''
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
np.random.seed(0)


# Find the min and max values for each column
def dataset_minmax(dataset):
```

```
    minmax = list()
    for i in range(len(dataset[0])):
        col_values = [row[i] for row in dataset]
        value_min = min(col_values)
        value_max = max(col_values)
        minmax.append([value_min, value_max])
    return minmax


# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset):
    minmax = dataset_minmax(dataset)
    for row in dataset:
        for i in range(len(row)):
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])
    return dataset


def train_test_split(X, y, test_size):

    df = pd.DataFrame(X, columns=['P', 'N'])
    df['label'] = y
    # Shuffle dataset
    shuffle_df = df.sample(frac=1)

    # Define a size for your train set
    train_size = int((1-test_size) * len(df))

    # Split your dataset
    train_set = shuffle_df[:train_size]
    test_set = shuffle_df[train_size:]

    X_train = train_set.to_numpy()[:, 0:-1]
    y_train = train_set.to_numpy()[:, -1]
    X_test = test_set.to_numpy()[:, 0:-1]
    y_test = test_set.to_numpy()[:, -1]

    return X_train, X_test, y_train, y_test


def balanced_acc(y_true, y_pred):

    tp, tn, fp, fn = 0, 0, 0, 0

    for y, y_hat in zip(y_true, y_pred):
        if y == y_hat == 1:
            tp += 1
        elif y == y_hat == 0:
            tn += 1
```

```
        elif y_hat == 1 and y == 0:
            fp += 1
        else:
            fn += 1
    sensitivity = tp / (tp + fn)
    specificity = tn / (fp + tn)
    balanced_acc = (sensitivity + specificity) / 2

    return round(balanced_acc, 3)


class Perceptron:
    def __init__(self):
        self.w = None
        self.b = None

    def threshold_function(self, x):
        fx = np.dot(self.w, x)+self.b
        return 1 if fx > 0 else 0

    def predict(self, X):
        y_pred = []
        X = normalize_dataset(X)
        for x in X:
            y_pred.append(self.threshold_function(x))

        return y_pred

    def fit(self, X, y, epochs=100, alpha=0.1, validation_split=0.2, weight_init='random',epoch_
            verbose=True, do_plot=True, do_validation=True, normalize=True, training_error_track

        if normalize:
            X = np.array(normalize_dataset(X))

        if do_validation:
            # train-validation split
            X_train, X_val, y_train, y_val = train_test_split(X, y, test_size= validation_split)
        else:
            X_train = X
            y_train = y

        # weight initialization
        if weight_init == 'random':
            self.w = np.random.rand(X.shape[1])      # random init of weights to braak symmetry
        if weight_init == 'zeros':
            self.w = np.zeros(X.shape[1])
        if weight_init == 'ones':
            self.w = np.ones(X.shape[1])
```

```
self.b = 0                                  # initializing bias to 0
weights_history = []
bias_history = []
train_errors = {}
val_errors = {}
for i in range(epochs):
    for xi, yi in zip(X_train, y_train):
        y_pred = self.threshold_function(xi)
        # weight update
        self.w = self.w + (alpha * (yi - y_pred)) * xi
        # bias update
        self.b = self.b + (alpha * (yi - y_pred)) * 1

    weights_history.append(self.w)
    bias_history.append(self.b)

    if do_validation:
        if i % epoch_checkpoint == 0:        # checkpoint to track error
            _y_train_preds = self.predict(X_train)
            _y_val_preds = self.predict(X_val)
            train_err_i = 1 - balanced_acc(y_train, _y_train_preds)
            val_err_i = 1 - balanced_acc(y_val, _y_val_preds)
            train_errors[i] = train_err_i
            val_errors[i] = val_err_i

            if verbose:
                print("Epoch %d: \n\t Training Error: %0.3f \t Validation Error: %0.3f"%


    if training_error_track:
        if i % epoch_checkpoint == 0:
            _y_preds = self.predict(X_train)
            train_err_i = 1 - balanced_acc(y_train, _y_preds)
            train_errors[i] = train_err_i

if do_validation:
    if do_plot:
        plt.plot(list(train_errors.keys()), list(train_errors.values()), label='E_train'
        plt.plot(list(val_errors.keys()), list(val_errors.values()), label='E_test')
        plt.xlabel('epochs')
        plt.ylabel('Error (1- balanced acc')
        plt.title('Training and Test Error of Perceptron with epochs')
        plt.legend(loc="upper left")
        # plt.xticks(list(train_errors.keys()))
        plt.savefig('figs/perceptron.pdf')
        plt.clf()

if training_error_track:
    return train_errors
```

4.4 main.py

```
#!/usr/bin/env python3
__author__ = "Shivchander Sudalairaj"
__license__ = "MIT"

'''
 Binary classification of Stressed/Not Stressed dataset using kNN, Neighborhood and Perceptron
'''

import re
import numpy as np
from knn import KNN
from neighborhood import NeighborhoodClassifier
import matplotlib.pyplot as plt
from perceptron import Perceptron, train_test_split
from statistics import mean, stdev


def parse_reformat(txt_file):
    X = []
    y = []
    with open(txt_file, 'r') as f:
        for line in f:

            if 'Not Stressed' == line.lstrip().rstrip():
                label = 0
            if 'Stressed' == line.lstrip().rstrip():
                label = 1

            # checking if the line is of the form <float> \t <float>
            if re.search('[+-]?[0-9]+\.[0-9]+\t+[+-]?[0-9]+\.[0-9]+', line.lstrip().rstrip()):
                row = list(map(float, line.rstrip().split('\t')))
                X.append(row)
                y.append(label)

    return X, y


def metrics(y_true, y_pred):
    """
    :param y_true: list of ground truth
    :param y_pred: list of predictions
    :return: dict of metrics {metric: score}
    """
    tp, tn, fp, fn = 0, 0, 0, 0
    metric = {}
    for y, y_hat in zip(y_true, y_pred):
        if y == y_hat == 1:
```

```
            tp += 1
        elif y == y_hat == 0:
            tn += 1
        elif y_hat == 1 and y == 0:
            fp += 1
        else:
            fn += 1

    sensitivity = tp / (tp + fn)
    specificity = tn / (fp + tn)
    balanced_acc = (sensitivity + specificity) / 2
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    f1 = 2 * ((precision * recall) / (precision + recall))

    metric['precision'] = round(precision, 3)
    metric['recall'] = round(recall, 3)
    metric['balanced_acc'] = round(balanced_acc, 3)
    metric['f1'] = round(f1, 3)

    return metric


def classifier(X, y, neighbor_param, classifier_type='knn', X_test = []):
    """
    :param X: feature vector
    :param y: labels
    :param neighbor_param: k value for kNN or R value for Neighborhood
    :param classifier_type: kNN (default), Neighborhood
    :return: predictions
    """
    y_pred = []
    if classifier_type == 'knn':
        model = KNN(X, y)
    if classifier_type == 'neighborhood':
        model = NeighborhoodClassifier(X, y)

    if len(X_test) != 0:
        for xi in X_test:
            y_pred.append(model.predict(xi, neighbor_param))
    else:
        for xi, y_true in zip(X, y):
            y_pred.append(model.predict(xi, neighbor_param))

    return y_pred


def plot_metrics(classifier_metrics, title):
    fig, a = plt.subplots(2, 2)
```

```
    width = 0.35
    x = np.arange(1, 10)
    if title == 'Perceptron':
        a[0][0].bar(x - width/2, [i[0] for i in classifier_metrics['balanced_acc']], width, labe
        a[0][0].bar(x + width/2, [i[1] for i in classifier_metrics['balanced_acc']], width, labe
        a[0][0].legend()
    else:
        a[0][0].bar(x, classifier_metrics['balanced_acc'])
    a[0][0].set_xlabel('Trial #')
    a[0][0].set_ylabel('Balanced Accuracy')
    a[0][0].set_xticks(x)
    a[0][0].set_title(title+' Balanced Accuracy')
    if title == 'Perceptron':
        a[0][1].bar(x - width / 2, [i[0] for i in classifier_metrics['f1']], width, label='Train
        a[0][1].bar(x + width / 2, [i[1] for i in classifier_metrics['f1']], width, label='Test'
        a[0][1].legend()
    else:
        a[0][1].bar(x, classifier_metrics['f1'])
    a[0][1].set_xlabel('Trial #')
    a[0][1].set_ylabel('f1 score')
    a[0][1].set_xticks(x)
    a[0][1].set_title(title + ' f1 score')
    if title == 'Perceptron':
        a[1][0].bar(x - width / 2, [i[0] for i in classifier_metrics['precision']], width, label
        a[1][0].bar(x + width / 2, [i[1] for i in classifier_metrics['precision']], width, label
        a[1][0].legend()
    else:
        a[1][0].bar(x, classifier_metrics['precision'])
    a[1][0].set_xlabel('Trial #')
    a[1][0].set_ylabel('precision')
    a[1][0].set_xticks(x)
    a[1][0].set_title(title + ' Precision')
    if title == 'Perceptron':
        a[1][1].bar(x - width / 2, [i[0] for i in classifier_metrics['recall']], width, label='T
        a[1][1].bar(x + width / 2, [i[1] for i in classifier_metrics['recall']], width, label='T
        a[1][1].legend()
    else:
        a[1][1].bar(x, classifier_metrics['recall'])
    a[1][1].set_xlabel('Trial #')
    a[1][1].set_ylabel('recall')
    a[1][1].set_xticks(x)
    a[1][1].set_title(title + ' Recall')
    plt.tight_layout()
    plt.savefig('figs/'+title+'_metrics.pdf')
    plt.clf()


def print_table(classifier_metrics, title):
    if title == 'Perceptron':
```

```
        precision = [i[1] for i in classifier_metrics['precision']]
        recall = [i[1] for i in classifier_metrics['recall']]
        f1 = [i[1] for i in classifier_metrics['f1']]
        balanced_acc = [i[1] for i in classifier_metrics['balanced_acc']]
    else:
        precision = classifier_metrics['precision']
        recall = classifier_metrics['recall']
        f1 = classifier_metrics['f1']
        balanced_acc = classifier_metrics['balanced_acc']

    print(title)
    print('\t', 'Balanced Accuracy: ', round(mean(balanced_acc), 3), '+/-', round(stdev(balanced
    print('\t', 'F1 score: ', round(mean(f1), 3), '+/-', round(stdev(f1), 3))
    print('\t', 'Precision: ', round(mean(precision), 3), '+/-', round(stdev(precision), 3))
    print('\t', 'Recall: ', round(mean(recall), 3), '+/-', round(stdev(recall), 3))


def q1(X, y):
    """
    :param X: Feature vectors of the dataset
    :param y: labels/ classes
    :return: None - Saves the two figures in the figs directory (should exist)
    """
    print('Solving q1')
    k_nn_performance = {}
    neighborhood_performance = {}
    for n in range(1, 12, 2):
        knn_predictions = classifier(X, y, n, 'knn')
        neighborhood_predictions = classifier(X, y, n, 'neighborhood')

        k_nn_performance[n] = metrics(y, knn_predictions)['balanced_acc']
        neighborhood_performance[n] = metrics(y, neighborhood_predictions)['balanced_acc']

    plt.plot(list(k_nn_performance.keys()), list(k_nn_performance.values()))
    plt.xlabel('k-value')
    plt.ylabel('Performance (Balanced Accuracy)')
    plt.title('Performance of kNN Classifier for different k-values')
    plt.xticks(list(k_nn_performance.keys()))
    plt.savefig('figs/knn_performance.pdf')
    plt.clf()

    plt.plot(list(neighborhood_performance.keys()), list(neighborhood_performance.values()))
    plt.xlabel('Radius')
    plt.ylabel('Performance (Balanced Accuracy)')
    plt.title('Performance of Neighborhood Classifier for different Radius')
    plt.xticks(list(neighborhood_performance.keys()))
    plt.savefig('figs/neighborhood_performance.pdf')
    plt.clf()
```

```
def q2(X, y):
    print('Solving q2')
    model = Perceptron()
    model.fit(X, y, alpha=0.001, weight_init='random', epochs=200, verbose=False, do_plot=True)


def q3(X, y):                               # comparison of kNN, neighborhood, perceptron
    print('Solving q3')
    knn_performance = {'balanced_acc': [], 'precision': [], 'recall': [], 'f1': []}
    neighborhood_performance = {'balanced_acc': [], 'precision': [], 'recall': [], 'f1': []}
    perceptron_performance = {'balanced_acc': [], 'precision': [], 'recall': [], 'f1': []}
    perceptron_error = []
    for i in range(0, 9):
        # splitting the dataset
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

        # testing and scoring kNN and neighborhood
        knn_predictions = classifier(X, y, 11, 'knn', X_test=X_test)
        neighborhood_predictions = classifier(X, y, 3, 'neighborhood', X_test=X_test)
        knn_performance['balanced_acc'].append(metrics(y_test, knn_predictions)['balanced_acc'])
        knn_performance['precision'].append(metrics(y_test, knn_predictions)['precision'])
        knn_performance['recall'].append(metrics(y_test, knn_predictions)['recall'])
        knn_performance['f1'].append(metrics(y_test, knn_predictions)['f1'])

        neighborhood_performance['balanced_acc'].append(metrics(y_test, neighborhood_predictions
        neighborhood_performance['precision'].append(metrics(y_test, neighborhood_predictions)['
        neighborhood_performance['recall'].append(metrics(y_test, neighborhood_predictions)['rec
        neighborhood_performance['f1'].append(metrics(y_test, neighborhood_predictions)['f1'])

        # testing and scoring perceptron
        model = Perceptron()
        perceptron_error.append(model.fit(X_train, y_train, alpha=0.001, weight_init='random', e
                                          do_validation=False, training_error_track=True, epoch_
        y_train_preds = model.predict(X_train)
        y_test_pred = model.predict(X_test)

        perceptron_performance['balanced_acc'].append([metrics(y_train, y_train_preds)['balanced
                                                       metrics(y_test, y_test_pred)['balanced_ac
        perceptron_performance['precision'].append([metrics(y_train, y_train_preds)['precision']
                                                    metrics(y_test, y_test_pred)['precision']])
        perceptron_performance['recall'].append([metrics(y_train, y_train_preds)['recall'],
                                                 metrics(y_test, y_test_pred)['recall']])
        perceptron_performance['f1'].append([metrics(y_train, y_train_preds)['f1'],
                                             metrics(y_test, y_test_pred)['f1']])

    # part a - metric plots
    print('Solving q3 - A')
    plot_metrics(knn_performance, 'kNN')
```

```
plot_metrics(neighborhood_performance, 'Neighborhood')
plot_metrics(perceptron_performance, 'Perceptron')

# part b - printing table
print('Solving q3 - B')
print_table(knn_performance, 'kNN')
print_table(neighborhood_performance, 'Neighborhood')
print_table(perceptron_performance, 'Perceptron')

# part c - trial-wise training error for perceptron
print('Solving q3 - C')
training_errors = []
for i, e in enumerate(perceptron_error):
    plt.plot(list(e.keys()), list(e.values()), label='Trial #'+str(i+1))
    training_errors.append(list(e.values()))
plt.xlabel('epochs')
plt.ylabel('Error (1- balanced acc')
plt.title('Trial-Wise Training Error Time-Series for the Perceptrons')
plt.legend(loc="upper right")
plt.xticks(list(perceptron_error[0].keys()))
plt.savefig('figs/trialwise_error_perceptron.pdf')
plt.clf()

# part d - Perceptron mean training error
print('Solving q3 - D')
mean_errors = list(np.mean(training_errors, axis=0))
std_errors = list(np.std(training_errors, axis=0))
plt.errorbar(list(perceptron_error[0].keys()), mean_errors, std_errors)
plt.xlabel('epochs')
plt.ylabel('Error (1- balanced acc)')
plt.title('Mean and Std Dev of Training Error for the Perceptrons')
plt.xticks(list(perceptron_error[0].keys()))
plt.savefig('figs/mean_error_perceptron.pdf')
plt.clf()

# decision boundaries
# define bounds of the domain
X = np.array(X)
y = np.array(y)
min1, max1 = X[:, 0].min() - 1, X[:, 0].max() + 1
min2, max2 = X[:, 1].min() - 1, X[:, 1].max() + 1
# define the x and y scale
x1grid = np.arange(min1, max1, 0.1)
x2grid = np.arange(min2, max2, 0.1)
# create all of the lines and rows of the grid
xx, yy = np.meshgrid(x1grid, x2grid)
# flatten each grid to a vector
r1, r2 = xx.flatten(), yy.flatten()
r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
```

```
# horizontal stack vectors to create x1,x2 input for the model
grid = np.hstack((r1, r2))
grid_list = grid.tolist()
# make knn predictions for the grid
print('Solving q3 - E')
yhat_knn = np.array(classifier(X, y, 11, 'knn', X_test=grid_list))
# make neighborhood predictions for the grid
print('Solving q3 - F')
yhat_neighborhood = np.array(classifier(X, y, 3, 'neighborhood', X_test=grid_list))
# make perceptron predictions for the grid
print('Solving q3 - G')
yhat_perceptron = np.array(model.predict(grid))

# reshape the predictions back into a grid
zz_knn = yhat_knn.reshape(xx.shape)
zz_neighborhood = yhat_neighborhood.reshape(xx.shape)
zz_perceptron = yhat_perceptron.reshape(xx.shape)

# part e - knn decision boundary
# plot the grid of x, y and z values as a surface
plt.contourf(xx, yy, zz_knn, cmap='Paired')
for class_value in range(2):
    # get row indexes for samples with this class
    row_ix = np.where(y == class_value)
    # create scatter of these samples
    plt.scatter(X[row_ix, 0], X[row_ix, 1], cmap='Paired')
plt.title('kNN Decision Boundary')
plt.xlabel('P Range')
plt.ylabel('N Range')
plt.savefig('figs/knn_decision_boundary.pdf')
plt.clf()

# part f - neighborhood decision boundary
# plot the grid of x, y and z values as a surface
plt.contourf(xx, yy, zz_neighborhood, cmap='Paired')
for class_value in range(2):
    # get row indexes for samples with this class
    row_ix = np.where(y == class_value)
    # create scatter of these samples
    plt.scatter(X[row_ix, 0], X[row_ix, 1], cmap='Paired')
plt.title('Neighborhood Decision Boundary')
plt.xlabel('P Range')
plt.ylabel('N Range')
plt.savefig('figs/neighborhood_decision_boundary.pdf')
plt.clf()

# part f - perceptron decision boundary
# plot the grid of x, y and z values as a surface
plt.contourf(xx, yy, zz_perceptron, cmap='Paired')
```

```
    for class_value in range(2):
        # get row indexes for samples with this class
        row_ix = np.where(y == class_value)
        # create scatter of these samples
        plt.scatter(X[row_ix, 0], X[row_ix, 1], cmap='Paired')
    plt.title('Perceptron Decision Boundary')
    plt.xlabel('P Range')
    plt.ylabel('N Range')
    plt.savefig('figs/perceptron_decision_boundary.pdf')
    plt.clf()


if __name__ == '__main__':
    import time
    start_time = time.time()
    X, y = parse_reformat('data/HW2_data.txt')
    q3(X, y)
    q1(X, y)
    q2(X, y)
    print("--- %s seconds ---" % (time.time() - start_time))
```