EECE6036 - Homework 2

Wayne Stegner

October 8, 2020

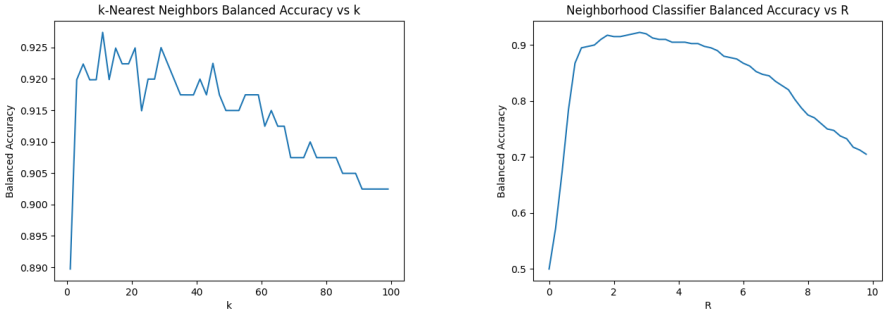
1 Problem 1

1.1 Problem Summary

In this problem, two spatial-neighborhood classifiers are investigated to classify people as either "Stressed" or "Not Stressed" based on previous and current income. The k-Nearest Neighbors (KNN) classifies based on the k nearest points, while the Neighborhood algorithm classifies based on the points within a radius R. In this implementation of the Neighborhood algorithm, ties in the distribution of classes in the neighborhood will default to "Not Stressed" The goal of this problem is to find optimal k and R for these algorithms when applied to the given dataset.

1.2 Results

The balanced accuracy metrics for KNN and Neighborhood classifiers are shown in Fig. 1a and Fig. 1b respectively. The values of k were from 1 to 99 to get a better idea of the scalability of k, using only the odd numbers to avoid ties in the class distribution of the neighbors. The values of R were from 0 to 10 incrementing by 0.2 to observe the behavior of R over a large range. The optimal value for k is 11 with a balanced accuracy of 92.74%, and the optimal value for R is 2.8 with a balanced accuracy of 92.25%.



(a) (b)

Figure 1: Balanced accuracy for KNN (a) and Neighborhood Classifier (b).

1.3 Discussion

KNN is slightly more accurate than the Neighborhood method. However, KNN has the advantage that it will always have a neighborhood with a consistent size and no ties (if k is odd). This dataset contains some sparse regions which may yield few or no neighbors for some points, but increasing R to accommodate for those regions will cause large neighborhoods in the dense regions to hold disproportionately many points.

1.4 Conclusion

The optimal k value is 11 and the optimal R value is 2.8. Both algorithms classify this dataset to over 92%.

2 Problem 2

2.1 Problem Summary

In this problem, the same dataset is classified using a single perceptron using a simple training algorithm. For this problem, the dataset was split into 80% training and 20% testing. The purpose of this problem is to try different learning rates, training epochs, and weight initialization strategies.

2.2 Results

The error (1 - balanced accuracy) for both training and testing after training for 500 at a learning rate of 0.0001 is shown in Fig. 2. The final training error is 0.108, and the final testing error is 0.070. Because this dataset is 2-dimensional, it was simple to choose a range of weights that would go across the dataset.

To get the desired general position, the bias weight is initialized on a uniform distribution between -10 and -50, while the input weights are initialized on a uniform distribution between 2 and 5. The axis intercepts can be calculated as $\frac{-w_{bias}}{w_{input}}$, which will be between 5 and 25 with the selected ranges. This range will put the initial decision boundary close to the data.

The learning rate and training epochs were found empirically. The learning rate was started out as 0.1, then adjusted down by a factor of 10 until the error curve smoothed out and settled down. The number of epochs was set based on the learning rate to observe the error bottom out, or continue to oscillate for the higher learning rates.

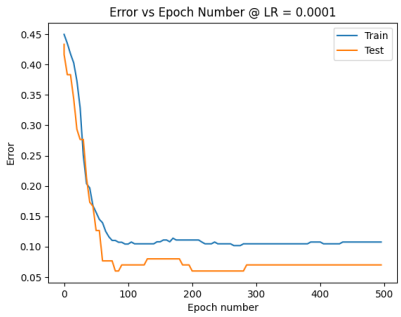


Figure 2: Training and testing error for a single perceptron.

2.3 Discussion

These results indicate that the perceptron is successfully training on the data, because the error decreases over time and stays low. Training appears to be finished around 100 epochs, but it is safer to train a bit longer (150 epochs) just in case the initial decision boundary is worse.

2.4 Conclusion

The perceptron should be trained with a learning rate of 0.0001 for 150 epochs to ensure full training.

3 Problem 3

3.1 Problem Summary

The purpose of this problem is to compare the performance of the three classifiers using the optimal parameters from Problems 1 and 2. This will be done over nine trials, in which each algorithm will classify a random 80% train/20% test partition of the dataset. For each trial, the perceptron has a random initialization of the weights as described in Problem 2. Performance metrics are measured for each trial, including balanced accuracy, precision, recall, and F1 score.

3.2 Results

3.2.1 Performance on Individual Trials

The performance metrics for the individual trials of the KNN, Neighborhood, and perceptron classifiers are shown in Fig. 3, Fig. 4, and Fig. 5 respectively.

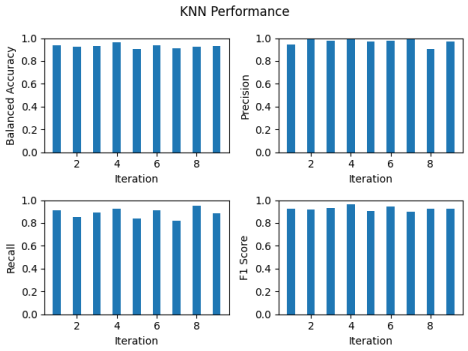


Figure 3: Performance metrics for the KNN classifiers.

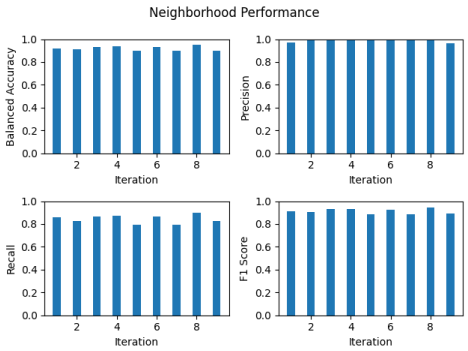


Figure 4: Performance metrics for the Neighborhood classifiers.

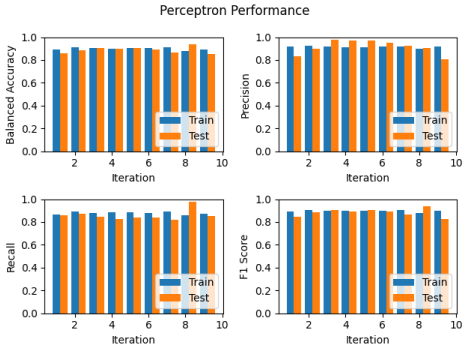


Figure 5: Performance metrics for the perceptron classifiers.

3.2.2 Average Performance

Tab. 1 shows the average and standard deviation performance metrics for the classifiers aggregated over all nine trials.

Table 1: Average performance of the classifiers.

	KNN	Neighborhood	Perceptron
Balanced Accuracy	0.929 ± 0.016	0.920 ± 0.018	0.890 ± 0.025
Precision	0.971 ± 0.030	0.993 ± 0.014	0.915 ± 0.058
Recall	0.887 ± 0.041	0.845 ± 0.034	0.859 ± 0.044
F1 Score	0.926 ± 0.018	0.912 ± 0.021	0.884 ± 0.031

3.2.3 Trial-Wise Training Error for the Perceptrons

The trial-wise training error for each of the nine perceptrons over the duration of training is shown in Fig. 6.

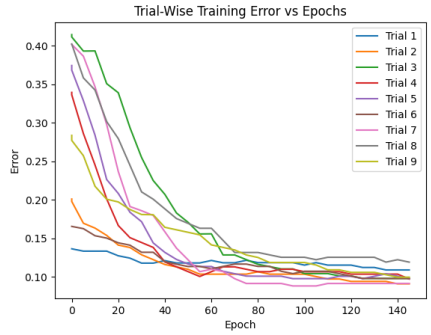


Figure 6: Trial-wise training error of perceptrons over training duration

3.2.4 Mean Training Error for the Perceptron

The mean and standard deviation training error for each of the nine perceptrons over the duration of training is shown in Fig. 7.

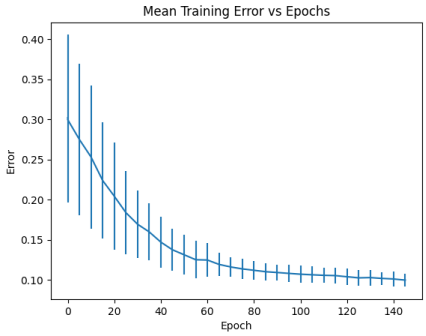


Figure 7: Mean training error of perceptrons over training duration.

3.2.5 Best KNN Decision Boundary

The decision boundary for the KNN classifier with the highest balanced accuracy is shown in Fig. 8. This boundary was calculated by sampling the input space with 100 divisions in both the N range (0 to 23) and the P range (0 to 14).

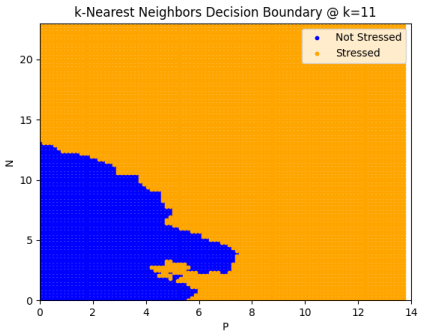


Figure 8: Decision boundary for the best KNN classifier.

3.2.6 Best Neighborhood Classifier Decision Boundary

The decision boundary for the Neighborhood classifier with the highest balanced accuracy is shown in Fig. 9. This boundary was calculated using the same method as the KNN. The blue regions in the upper left and right corners are an artifact of the tie resolution method, which is to default to a "Not Stressed" classification.

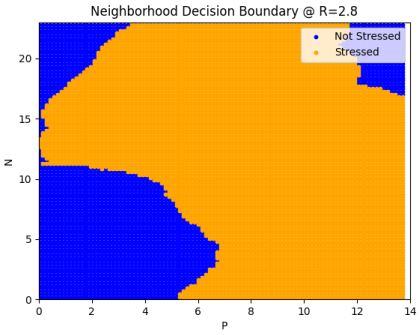


Figure 9: Decision boundary for the best Neighborhood classifier.

3.2.7 Perceptron Decision Boundary

The decision boundary for the perceptron unit with the highest balanced accuracy is shown in Fig. 10. This boundary was calculated based on the weights of the trained unit.

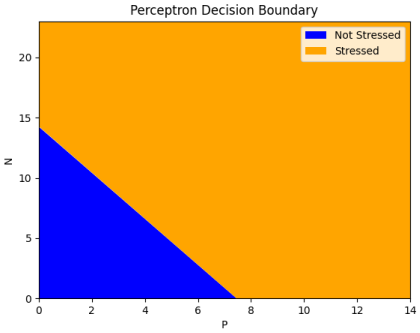


Figure 10: Decision boundary for the best perceptron unit.

3.2.8 Analysis of Results

These algorithms all exhibit the ability to classify the given dataset, with each of them having balanced accuracies of near or just above 90% averaged over all nine trials, shown in Tab. 1. The standard deviations are all relatively low, indicating that all of the trials had similar ability to classify the testing set. Therefore, it is safe to say that all of the classification algorithms have the ability to classify this particular dataset.

These algorithms each have pros and cons. The KNN classifier is advantageous in that it has the ability to generate a nonlinear decision boundary, as shown in Fig. 8. Additionally, it has the guarantee that each point will have enough neighbors to make a classification. On the negative side, the

KNN decision boundary can over-fit. Additionally, because each point is guaranteed to have neighbors, these neighbors might be spatially distant for outliers, which may lead to a classification that does not make sense.

The Neighborhood classifier, like the KNN classifier, has the ability to produce nonlinear decision boundaries, as shown in Fig. 9. In contrast to the KNN, the Neighborhood classifier guarantees that all neighbors are spatially close. However, because neighbors are based on proximity, points with either equal distributions of classes in the neighbors or no neighbors at all may be misclassified. For this implementation, the default for those cases was a classification of "Not Stressed," which causes all outliers to be in that class. With datasets with a mix of dense and sparse data points, it will be challenging to find a radius where all sparse points have neighbors, but the dense points do not have extraneous amounts of neighbors.

The perceptron can only produce a linear decision boundary as shown in Fig. 10, which can be viewed as both a pro and a con. On the positive side, the linear decision boundary is simpler than the KNN and Neighborhood decision boundaries, and achieves comparable results. However, the negative side is that it will only work for linearly separable data. Another negative side is that it takes time to train the unit, while KNN and Neighborhood do not. However, the classification calculation by the perceptron is much simpler, because it only has to do a few multiplications and summations, rather than calculating Euclidean distance for every single reference point.

While each of the classifiers exhibit reasonably good performance, based on this study I will recommend the KNN for this particular dataset. The KNN outperforms the Neighborhood classifier in every metric presented in Tab. 1 except for precision, and the precision of the KNN was only about 2% worse for the KNN. While the linear decision boundary of the perceptron allows for quick classification, the perceptron performed the worst in every metric. Additionally, this dataset is not strictly linearly separable, and there is significant overlap between the two classes that cannot be captured by a single perceptron.

3.3 Discussion

The KNN, Neighborhood, and perceptron classifiers all were able to classify this dataset. They each have pros and cons, but in the end the KNN is probably the best classifier for this dataset. That being said, the recommendation may change if more data is acquired which suggests that the KNN was merely over-fit to this dataset, and a linear decision boundary is more accurate. In that case, the perceptron would be better.

3.4 Conclusion

While all three algorithms were able to classify the dataset, the KNN is the most recommended classifier for this particular case.

A Code

A.1 dataset.py

```
################################################################################
# Imports
################################################################################
import numpy as np


class Dataset:
    def __init__(
            self,
            data_file = 'data.txt',
            classes = ['Not Stressed', 'Stressed'],
            shuffle_data = False,
    ):
        self.data_file = str(data_file)
        self.classes = classes
        self.importDataFile()
        if shuffle_data:
            self.shuffleData()

    def isFloat(self, test_str):
        '''Quick check if a string is a float value
        Parameters:
        -----------
            test_str : str
                String to test for float cast compatibility
        Returns:
            bool
                True if the string can cast to a float, else false
        '''
        try:
            float(test_str)
            return True
        except ValueError:
            return False

    def importDataFile(self):
        '''Imports data from the txt format Dr. Minai presented
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
        print(f'Importing text data from {self.data_file}')
        self.data_points = list()
        # Read data from txt file
        with open(self.data_file) as data:
            class_id = 0
            for line in data.readlines():
                line = line[:-1] # Strip off newline
                # Check if line is new class definition
                if line in self.classes:
                    class_id = self.classes.index(line)
                    continue
                # Check if line is a set of data points
                line = line.split()
                if (len(line) == 2) and self.isFloat(line[0]) and self.isFloat(line[1]):
                    # Each data point will be [P, N, Truth]
                    new_item = np.array(
                        (float(line[0]), float(line[1]), float(class_id)),
                        dtype=np.float64
                    )
                    self.data_points.append(new_item)
        num_data_points = len(self.data_points)
        self.data_points = np.array(self.data_points, dtype=np.float64)

    def resizeDataPartitions(self, num_train_points):
        '''Size dataset partitions
        Parameters:
        -----------
            num_train_points : int
                Number of train data points
                The rest will be test points
        Returns:
        --------
            None
        '''
        num_test_points = len(self.data_points) - num_train_points
```

```
        self.train = np.empty((num_train_points, 3), dtype=np.float64)
        self.test = np.empty((num_test_points, 3), dtype=np.float64)

    def partitionAllTrain(self):
        '''Format entire dataset to training
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
        # Allocate train and test numpy arrays
        num_train_points = len(self.data_points)
        self.resizeDataPartitions(num_train_points)
        # Iterate through data points
        for i in range(len(self.data_points)):
            self.train[i] = self.data_points[i]

    def partitionOneTest(self, skip_idx):
        '''Format all but one data point to training
        Parameters:
        -----------
            skip_idx : int
                Index of testing data point
        Returns:
        --------
            None
        '''
        num_train_points = len(self.data_points) - 1
        self.resizeDataPartitions(num_train_points)
        # Iterate through data points except for skip_idx
        for i in range(len(self.data_points)):
            if i == skip_idx:
                self.test[0] = self.data_points[i]
                continue
            item_idx = i
            if i > skip_idx:
                item_idx -= 1
            self.train[item_idx] = self.data_points[i]

    def partitionXTrain(self, train_portion):
        '''Format a portion of the dataset to training
        Parameters:
        -----------
            train_portion : float
                Portion of data to make train
        Returns:
        --------
            None
        '''
        num_train_points = int(train_portion * len(self.data_points))
        self.resizeDataPartitions(num_train_points)
        # Iterate through data_points and partition
        for i in range(len(self.data_points)):
            if i < num_train_points:
                self.train[i] = self.data_points[i]
            else:
                item_idx = i - num_train_points
                self.test[item_idx] = self.data_points[i]

    def shuffleData(self):
        '''Shuffle data in self.data_points
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
        np.random.shuffle(self.data_points)


if __name__ == '__main__':
    from matplotlib import pyplot as plt
    import pathlib
    CODE_DIR = pathlib.Path(__file__).parent.absolute()
    ROOT_DIR = CODE_DIR.parent
    DATA_FILE = CODE_DIR.joinpath('data.txt')
    IMG_DIR = ROOT_DIR.joinpath('images')
    IMG_DIR.mkdir(mode=0o775, exist_ok=True)
    # Plot data points
    dataset = Dataset(data_file = DATA_FILE)
    dataset.partitionAllTrain()
```

```
    not_stressed = {'p': list(), 'n': list()}
    stressed = {'p': list(), 'n': list()}
    for point in dataset.train:
        if point[2] == 0: # Not stressed
            not_stressed['p'].append(point[0])
            not_stressed['n'].append(point[1])
        else: # Stressed
            stressed['p'].append(point[0])
            stressed['n'].append(point[1])
    plt.scatter(not_stressed['p'], not_stressed['n'], label='Not Stressed')
    plt.scatter(stressed['p'], stressed['n'], label='Stressed')
    plt.xlabel('P')
    plt.ylabel('N')
    plt.legend()
    data_plot = IMG_DIR.joinpath('dataset.png')
    plt.savefig(str(data_plot))
```

A.2 classifier.py

```
import numpy as np
import matplotlib.pyplot as plt

class Classifier():
    '''Abstract class for classifiers for Homework 2
    Used to calculate the accuracy metrics
    '''

    def __init__(self):
        '''Helps with autocomplete of parameters
        '''
        self.resetStats()

    def resetStats(self):
        '''Reset the true/false pos/neg stats
        '''
        self.true_pos = 0
        self.true_neg = 0
        self.false_pos = 0
        self.false_neg = 0

    def getSens(self):
        '''Return sensitivity (recall, true positive rate)
        '''
        return float(self.true_pos) / (self.true_pos + self.false_neg)

    def getSpec(self):
        '''Return specificity (selectivity, true negative rate)
        '''
        return float(self.true_neg) / (self.false_pos + self.true_neg)

    def getPPV(self):
        '''Return positive predictive value (precision)
        '''
        return float(self.true_pos) / (self.true_pos + self.false_pos)

    def getNPV(self):
        '''Return negative predictive value
        '''
        return float(self.true_neg) / (self.true_neg + self.false_neg)

    def getAcc(self):
        '''Return accuracy (unbalanced)
        '''
        total_true = self.true_pos + self.true_neg
        total_false = self.false_pos + self.false_neg
        return float(total_true) / (total_true + total_neg)

    def getBalAcc(self):
        '''Return balanced accuracy
        '''
        return (self.getSens() + self.getSpec()) / 2.0

    def getF1(self):
        '''Return F_1 score
        '''
        return 2 / ((1/self.getPPV()) + (1/self.getSens()))

    def getFbeta(self, beta):
        '''Return F_beta score
        '''
        beta2 = beta * beta
        return (1 + beta2) / (1/(beta2 * self.getPPV()) + (1/self.getSens()))
```

```
    def scoreResult(self, pred, truth):
        '''Classify results as true/false pos/neg
        Parameters:
        -----------
            pred : int
                Predicted class
            truth : int
                Truth class
        Returns:
        --------
            None
        '''
        if truth == 0:
            if pred == 0:
                self.true_neg += 1
            else:
                self.false_pos += 1
        else:
            if pred == 0:
                self.false_neg += 1
            else:
                self.true_pos += 1

    def drawDecision(
            self,
            div = 100,
            plot_name = 'knn_dec_bound.png',
            plot_title = 'Decision Boundary',
            max_p = 14,
            max_n = 23,
        ):
        '''Draw decision boundary
        Parameters:
        -----------
            div : int
                Number of divisions for the axis with maximal value
                In this case, the N axis
        Returns:
        --------
            None
        '''
        p_step = max_p / div
        n_step = max_n / div
        p_steps = int(max_p / p_step)
        n_steps = int(max_n / n_step)
        stressed_p = list()
        stressed_n = list()
        not_stressed_p = list()
        not_stressed_n = list()
        for p_idx in range(p_steps):
            for n_idx in range(n_steps):
                p = p_idx * p_step
                n = n_idx * n_step
                # test_points[p_step][n_step][0] = p
                # test_points[p_step][n_step][1] = n
                test_point = np.array([p, n], dtype=np.float64)
                pred = self.predict(test_point)
                if pred == 0.0:
                    not_stressed_p.append(p)
                    not_stressed_n.append(n)
                if pred == 1.0:
                    stressed_p.append(p)
                    stressed_n.append(n)
        plt.figure()
        plt.scatter(not_stressed_p, not_stressed_n, c='blue', s=10,
                label='Not Stressed')
        plt.scatter(stressed_p, stressed_n, c='orange', s=10, label='Stressed')
        plt.title(plot_title)
        plt.xlabel('P')
        plt.ylabel('N')
        plt.legend(loc='upper right')
        plt.xlim([0, max_p])
        plt.ylim([0, max_n])
        plt.savefig(str(plot_name))
        plt.close()
```

A.3 knn.py

```
################################################################################
# Imports
################################################################################
from dataset import Dataset
from classifier import Classifier
```

```
import numpy as np
from matplotlib import pyplot as plt


class KNN(Classifier):
    def __init__(
            self,
            dataset,
            k = 11,
    ):
        self.k = k
        self.dataset = dataset
        self.resetStats()

    def setK(self, k):
        '''Set value of k
        Parameters:
        -----------
            k : int
                Value of k
        Returns:
        --------
            None
        '''
        self.k = k

    def getDistance(self, x, y):
        '''Get Euclidean distance between 2 points
        Parameters:
        -----------
            x, y : np.array of float
                Points to calculate distance between
        Returns:
        --------
            float
                Euclidean distance between x and y
        '''
        # What a one-line beauty
        return np.sqrt(np.sum(np.square(x - y)))

    def predict(self, test):
        '''Predict a value for test based on dataset.train points
        Parameters:
        -----------
            test : np.array of float
                Point to predict class of
        Returns:
        --------
            int (or maybe float, haven't decided yet)
                Class prediction for test
        '''
        distances = list()
        for i in range(len(self.dataset.train)):
            # Store truth and distance
            new_point = dict()
            new_point['truth'] = self.dataset.train[i][2]
            new_point['dist'] = self.getDistance(test, self.dataset.train[i][:2])
            distances.append(new_point)
        # Sort distances and only keep the lowest k
        distances.sort(key=lambda x: x['dist'])
        distances = distances[:self.k]
        closest_classes = {0.: 0, 1.: 0}
        for d in distances:
            closest_classes[d['truth']] += 1
        return max(closest_classes, key = lambda x: closest_classes[x])

    def evalAll(self):
        '''Predict on all data points in dataset.data_points
        For question 1
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
        self.resetStats()
        # Iterate through all data points as the test points
        for i in range(len(self.dataset.data_points)):
            self.dataset.partitionOneTest(i)
            test_point = self.dataset.test[0][:2]
            test_truth = self.dataset.test[0][2]
            test_pred = self.predict(test_point)
            self.scoreResult(test_pred, test_truth)
```

```
    def evalTest(self):
        '''Predict on all data points in the test set
        For question 3
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
        self.resetStats()
        # Iterate through all data points as the test points
        for p in self.dataset.test:
            test_point = p[:2]
            test_truth = p[2]
            test_pred = self.predict(test_point)
            self.scoreResult(test_pred, test_truth)


if __name__ == '__main__':
    '''
    Fulfils problem 1 part 1
    '''
    # Seed RNG for repeatability
    np.random.seed(69420)
    # Additional imports
    import pathlib
    # File locations
    CODE_DIR = pathlib.Path(__file__).parent.absolute()
    ROOT_DIR = CODE_DIR.parent # Root project dir
    IMG_DIR = ROOT_DIR.joinpath('images')
    IMG_DIR.mkdir(mode=0o775, exist_ok=True)
    DATA_IN_FILE = CODE_DIR.joinpath('data.txt')
    DATA_OUT_DIR = ROOT_DIR.joinpath('data')
    DATA_OUT_DIR.mkdir(mode=0o775, exist_ok=True)
    DATA_OUT_FILE = DATA_OUT_DIR.joinpath('knn_acc')
    BAL_ACC_PLOT = IMG_DIR.joinpath('knn_bal_acc.png')
    # Get dataset
    dataset = Dataset(data_file = DATA_IN_FILE, shuffle_data = True)
    knn = KNN(dataset, k=1)
    # Iterate through k values
    k_vals = range(1, 100, 2)
    bal_acc = list()
    for k in k_vals:
        knn.setK(k)
        knn.evalAll()
        bal_acc.append(knn.getBalAcc())
        print(f'k = {k}\tBalanced accuracy: {knn.getBalAcc()}')
    max_k = k_vals[np.argmax(bal_acc)]
    print(f'Maximum accuracy: {np.max(bal_acc)}\tk = {max_k}')
    # Plot data points
    plt.figure()
    plt.plot(k_vals, bal_acc)
    plt.title('k-Nearest Neighbors Balanced Accuracy vs k')
    plt.xlabel('k')
    plt.ylabel('Balanced Accuracy')
    plt.savefig(str(BAL_ACC_PLOT))
    plt.close()
    # Get accuracy at optimal k
    knn.setK(max_k)
    knn.evalAll()
    with open(str(DATA_OUT_FILE), 'w') as data_f:
        data_f.write(f'bestk = {max_k}\n')
        knn.evalAll()
        acc = 100 * knn.getBalAcc()
        data_f.write(f'acc = {acc:0.2f}\n')
```

A.4 neighborhood.py

```
################################################################################
# Imports
################################################################################
from dataset import Dataset
from classifier import Classifier
import numpy as np
from matplotlib import pyplot as plt


class Neighborhood(Classifier):
    def __init__(
            self,
            dataset,
```

```
        R = 1,
):
    self.R = R
    self.dataset = dataset
    self.resetStats()

def setR(self, R):
    '''Set value of R
    Parameters:
    -----------
        R : int
            Value of R
    Returns:
    --------
        None
    '''
    self.R = R

def getDistance(self, x, y):
    '''Get Euclidean distance between 2 points
    Parameters:
    -----------
        x, y : np.array of float
            Points to calculate distance between
    Returns:
    --------
        float
            Euclidean distance between x and y
    '''
    return np.sqrt(np.sum(np.square(x - y)))

def predict(self, test):
    '''Predict a value for test based on dataset.train points
    Parameters:
    -----------
        test : np.array of float
            Point to predict class of
    Returns:
    --------
        int (or maybe float, haven't decided yet)
            Class prediction for test
    '''
    distances = list()
    for i in range(len(self.dataset.train)):
        # Store truth and distance
        new_point = dict()
        new_point['truth'] = self.dataset.train[i][2]
        new_point['dist'] = self.getDistance(test, self.dataset.train[i][:2])
        distances.append(new_point)
    # Sort distances and only keep the lowest R
    distances = [x for x in distances if x['dist'] <= self.R]
    closest_classes = {0.: 0, 1.: 0}
    for d in distances:
        closest_classes[d['truth']] += 1
    # TODO - If classes are equal it returns 0.0
    return max(closest_classes, key = lambda x: closest_classes[x])

def evalAll(self):
    '''Predict on all data points in dataset.data_points
    For question 1
    Parameters:
    -----------
        None
    Returns:
    --------
        None
    '''
    self.resetStats()
    # Iterate through all data points as the test points
    for i in range(len(self.dataset.data_points)):
        self.dataset.partitionOneTest(i)
        test_point = self.dataset.test[0][:2]
        test_truth = self.dataset.test[0][2]
        test_pred = self.predict(test_point)
        self.scoreResult(test_pred, test_truth)

def evalTest(self):
    '''Predict on all data points in the test set
    For question 3
    Parameters:
    -----------
        None
    Returns:
    --------
```

```
            None
        '''
        self.resetStats()
        # Iterate through all data points as the test points
        for p in self.dataset.test:
            test_point = p[:2]
            test_truth = p[2]
            test_pred = self.predict(test_point)
            self.scoreResult(test_pred, test_truth)


if __name__ == '__main__':
    '''
    Fulfils problem 1 part 2
    '''
    # Seed RNG for repeatability
    np.random.seed(69420)
    # Additional imports
    import pathlib
    # File locations
    CODE_DIR = pathlib.Path(__file__).parent.absolute()
    ROOT_DIR = CODE_DIR.parent # Root project dir
    IMG_DIR = ROOT_DIR.joinpath('images')
    IMG_DIR.mkdir(mode=0o775, exist_ok=True) # Create images dir if needed
    BAL_ACC_PLOT = IMG_DIR.joinpath('neighborhood_bal_acc.png')
    DATA_IN_FILE = CODE_DIR.joinpath('data.txt')
    DATA_OUT_DIR = ROOT_DIR.joinpath('data')
    DATA_OUT_DIR.mkdir(mode=0o775, exist_ok=True)
    DATA_OUT_FILE = DATA_OUT_DIR.joinpath('neighborhood_acc')
    # Get dataset
    dataset = Dataset(data_file = DATA_IN_FILE, shuffle_data = True)
    neighborhood = Neighborhood(dataset, R=1)
    # Iterate through R values
    R_vals = np.arange(0, 10, 0.2)
    bal_acc = list()
    for R in R_vals:
        neighborhood.setR(R)
        neighborhood.evalAll()
        bal_acc.append(neighborhood.getBalAcc())
        print(f'R = {R:0.1f}\tBalanced accuracy: {neighborhood.getBalAcc()}')
    max_r = R_vals[np.argmax(bal_acc)]
    print(f'Maximum accuracy: {np.max(bal_acc)}\tr = {max_r:0.1f}')
    # Plot data points
    plt.plot(R_vals, bal_acc)
    plt.title('Neighborhood Classifier Balanced Accuracy vs R')
    plt.xlabel('R')
    plt.ylabel('Balanced Accuracy')
    plt.savefig(str(BAL_ACC_PLOT))
    # Get accuracy at optimal R
    neighborhood.setR(max_r)
    neighborhood.evalAll()
    with open(str(DATA_OUT_FILE), 'w') as data_f:
        data_f.write(f'bestr = {max_r:0.1f}\n')
        neighborhood.evalAll()
        acc = 100 * neighborhood.getBalAcc()
        data_f.write(f'acc = {acc:0.2f}\n')
```

A.5 perceptron.py

```
################################################################################
# Imports
################################################################################
from dataset import Dataset
from classifier import Classifier
import numpy as np
from matplotlib import pyplot as plt


class Perceptron(Classifier):
    def __init__(
            self,
            dataset,
            num_inputs = 2,
            learning_rate = 0.1,
    ):
        self.num_inputs = num_inputs
        self.learning_rate = learning_rate
        self.init_weights()
        self.dataset = dataset
        self.resetStats()
        self.resetAcc()

    def resetAcc(self):
```

```
        '''Reset accuracy metrics
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
        self.train_acc = list()
        self.test_acc = list()
        self.epoch_nums = list()

    def init_weights(self):
        '''Initialize weights for the perceptron
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
        self.weights = np.empty(self.num_inputs + 1, dtype=np.float64)
        # Initialize bias weight
        self.weights[0] = np.random.uniform(-10, -50)
        # Initialize input weights
        for i in range(len(self.weights) - 1):
            self.weights[i+1] = np.random.uniform(2, 5)

    def get_weights(self):
        '''Return weights array
        Parameters:
        -----------
            None
        Returns:
        --------
            np.array of float64
                Array of weight values
        '''
        return self.weights

    def predict(self, input_val):
        '''Predict a value for test based on self.weights
        Parameters:
        -----------
            input_val : np.array of float64
                Point to predict class of
        Returns:
        --------
            int (or maybe float, haven't decided yet)
                Class prediction for test
        '''
        # Concatenate bias value
        input_val = np.concatenate(([1], input_val))
        score = np.sum(input_val * self.weights)
        if score > 0:
            return 1
        return 0

    def trainSingleVal(self, input_val):
        '''Change weights based on a single point
        Parameters:
        -----------
            input_val : np.array of float64
                Point to learn on, ending with truth value
        Returns:
        --------
            None
        '''
        point = input_val[:-1]
        truth = input_val[-1]
        # Calculate weight change based on pred and point
        pred = self.predict(point)
        change_scalar = self.learning_rate * (truth - pred)
        self.weights += change_scalar * np.concatenate([[1], point])

    def trainEpoch(self):
        '''Change weights based on the training set
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
```

```
        for point in self.dataset.train:
            self.trainSingleVal(point)

    def train(self, epochs=10):
        '''Change weights for a desired number of epochs
        Parameters:
        -----------
            epochs : int
                Amount of epochs to train
        Returns:
        --------
            None
        '''
        # Log initial accuracies
        self.resetAcc()
        self.logAll(0)
        for e in range(epochs):
            # Perform epoch training
            self.trainEpoch()
            if (e % 5) == 0:
                self.logAll(e)

    def evalTrain(self):
        '''
        '''
        self.resetStats()
        for point in self.dataset.train:
            p = point[:-1]
            truth = point[-1]
            self.scoreResult(self.predict(p), truth)

    def evalTest(self):
        '''
        '''
        self.resetStats()
        for point in self.dataset.test:
            p = point[:-1]
            truth = point[-1]
            self.scoreResult(self.predict(p), truth)

    def logAll(self, epoch_num):
        '''Evaluate test and train dataset partitions
        Parameters:
        -----------
            None
        Returns:
        --------
            None
        '''
        self.evalTrain()
        self.train_acc.append(self.getBalAcc())
        self.evalTest()
        self.test_acc.append(self.getBalAcc())
        self.epoch_nums.append(epoch_num)

    def plotError(self, plot_name = 'epoch_error.png'):
        plt.figure()
        plt.plot(self.epoch_nums, np.subtract(1, self.train_acc))
        plt.plot(self.epoch_nums, np.subtract(1, self.test_acc))
        plt.legend(['Train', 'Test'])
        plt.title(f'Error vs Epoch Number @ LR = {self.learning_rate}')
        plt.xlabel('Epoch number')
        plt.ylabel('Error')
        plt.savefig(str(plot_name))
        plt.close()

    def drawDecision(
            self,
            div = None,
            plot_name = 'knn_dec_bound.png',
            plot_title = 'Decision Boundary',
            max_p = 14,
            max_n = 23,
        ):
        '''Draw decision boundary
        Parameters:
        -----------
            div : int
                Number of divisions for the axis with maximal value
                In this case, the N axis
        Returns:
        --------
            None
        '''
```

```
        p_axis_point = -self.weights[0]/self.weights[1]
        n_axis_point = -self.weights[0]/self.weights[2]
        plt.figure()
        plt.fill([0, 0, p_axis_point], [0, n_axis_point, 0], 'blue')
        plt.fill(
            [0, 0, max_p, max_p, p_axis_point],
            [n_axis_point, max_n, max_n, 0, 0],
            'orange'
        )
        plt.title(plot_title)
        plt.xlabel('P')
        plt.ylabel('N')
        plt.xlim([0, max_p])
        plt.ylim([0, max_n])
        plt.legend(['Not Stressed', 'Stressed'])
        plt.savefig(str(plot_name))
        plt.close()

if __name__ == '__main__':
    '''
    Fulfils problem 2
    '''
    # Seed RNG for repeatability
    np.random.seed(80085)
    # Additional imports
    import pathlib
    # File locations
    CODE_DIR = pathlib.Path(__file__).parent.absolute()
    ROOT_DIR = CODE_DIR.parent # Root project dir
    IMG_DIR = ROOT_DIR.joinpath('images')
    IMG_DIR.mkdir(mode=0o775, exist_ok=True)
    DATA_IN_FILE = CODE_DIR.joinpath('data.txt')
    DATA_OUT_DIR = ROOT_DIR.joinpath('data')
    DATA_OUT_DIR.mkdir(mode=0o775, exist_ok=True)
    DATA_OUT_FILE = DATA_OUT_DIR.joinpath('perceptron_err')
    ERR_PLOT = IMG_DIR.joinpath('perceptron_err.png')
    # Parameters
    LEARNING_RATE = 0.0001
    EPOCHS = 500
    # Get dataset
    dataset = Dataset(data_file = DATA_IN_FILE, shuffle_data = True)
    perc = Perceptron(dataset, learning_rate = LEARNING_RATE)
    perc.dataset.partitionXTrain(0.8)
    perc.train(epochs = EPOCHS)
    perc.plotError(ERR_PLOT)
    with open(str(DATA_OUT_FILE), 'w') as data_f:
        # Log learning rate and epochs
        data_f.write(f'learning_rate = {LEARNING_RATE}\n')
        data_f.write(f'epochs = {EPOCHS}\n')
        # Log train accuracy
        perc.evalTrain()
        train_err = 1 - perc.getBalAcc()
        data_f.write(f'train_err = {train_err:0.3f}\n')
        # Log test accuracy
        perc.evalTest()
        test_err = 1 - perc.getBalAcc()
        data_f.write(f'test_err = {test_err:0.3f}\n')
```

A.6 problem_3.py

```
################################################################################
# Imports
################################################################################
from dataset import Dataset
from knn import KNN
from neighborhood import Neighborhood
from perceptron import Perceptron
import numpy as np
from matplotlib import pyplot as plt
import pathlib


################################################################################
# Best Metrics / Constants
################################################################################
K = 11
R = 2.8
LEARNING_RATE = 0.0001
EPOCHS = 150
# File locations
CODE_DIR = pathlib.Path(__file__).parent.absolute()
ROOT_DIR = CODE_DIR.parent # Root project dir
IMG_DIR = ROOT_DIR.joinpath('images')
```

```
IMG_DIR.mkdir(mode=0o775, exist_ok=True)
KNN_PERF = IMG_DIR.joinpath('knn_performance.png')
KNN_DEC_BOUND = IMG_DIR.joinpath('knn_dec_bound.png')
NEIGHBOR_DEC_BOUND = IMG_DIR.joinpath('neighborhood_dec_bound.png')
NEIGHBOR_PERF = IMG_DIR.joinpath('neighborhood_performance.png')
PERC_DEC_BOUND = IMG_DIR.joinpath('perceptron_dec_bound.png')
PERC_PERF = IMG_DIR.joinpath('perceptron_performance.png')
TRIAL_ERR = IMG_DIR.joinpath('trial_wise_error.png')
MEAN_ERR = IMG_DIR.joinpath('mean_error.png')
DATA_IN_FILE = CODE_DIR.joinpath('data.txt')
DATA_OUT_DIR = ROOT_DIR.joinpath('data')
DATA_OUT_DIR.mkdir(mode=0o775, exist_ok=True)
AVG_PERF_TAB = DATA_OUT_DIR.joinpath('avg_perf.csv')


################################################################################
# Problem 3
################################################################################
# Function definitions
def logMetrics(classifier, key):
    '''Log performance metrics for the given classifier
    Parameters:
    -----------
        classifier : Classifier
            Classifier object to get metrics from
            Should be KNN, Neighborhood, or Perceptron
        key : str
            Key for list to store metrics in
    '''
    bal_acc[key].append(classifier.getBalAcc())
    precision[key].append(classifier.getPPV())
    recall[key].append(classifier.getSens())
    f1[key].append(classifier.getF1())

def bestDecision(classifier, key, plot_name, plot_title):
    if classifier.getBalAcc() > maxAcc[key]:
        print('* New best accuracy')
        maxAcc[key] = classifier.getBalAcc()
        classifier.drawDecision(plot_name=plot_name,
                plot_title=plot_title)

def plotPerf(keys, plot_name, plot_title = '', legend = None):
    bar_width = .4
    x = np.arange(1, 10)
    plt.figure()
    plt.suptitle(plot_title)
    # Balanced Accuracy
    plt.subplot(221)
    plt.xlabel('Iteration')
    plt.ylabel('Balanced Accuracy')
    plt.ylim([0, 1])
    for i, key in enumerate(keys):
        plt.bar((i * bar_width) + x, bal_acc[key], width = bar_width)
    # Precision
    plt.subplot(222)
    plt.xlabel('Iteration')
    plt.ylabel('Precision')
    plt.ylim([0, 1])
    for i, key in enumerate(keys):
        plt.bar((i * bar_width) + x, precision[key], width = bar_width)
    # Recall
    plt.subplot(223)
    plt.xlabel('Iteration')
    plt.ylabel('Recall')
    plt.ylim([0, 1])
    for i, key in enumerate(keys):
        plt.bar((i * bar_width) + x, recall[key], width = bar_width)
    # F1
    plt.subplot(224)
    plt.xlabel('Iteration')
    plt.ylabel('F1 Score')
    plt.ylim([0, 1])
    for i, key in enumerate(keys):
        plt.bar((i * bar_width) + x, f1[key], width = bar_width)
    if legend != None:
        for idx in range(221, 225):
            plt.subplot(idx)
            plt.legend(legend, loc='lower right')
    plt.tight_layout()
    plt.savefig(str(plot_name))
    plt.close()

# Seed RNG for repeatability
np.random.seed(69420)
```

```
# Store results for later graphing
bal_acc = {
    'knn': list(),
    'neighborhood': list(),
    'perc_train': list(),
    'perc_test': list(),
}
precision = {
    'knn': list(),
    'neighborhood': list(),
    'perc_train': list(),
    'perc_test': list(),
}
recall = {
    'knn': list(),
    'neighborhood': list(),
    'perc_train': list(),
    'perc_test': list(),
}
f1 = {
    'knn': list(),
    'neighborhood': list(),
    'perc_train': list(),
    'perc_test': list(),
}
train_err = list()
maxAcc = {
    'knn': 0,
    'neighborhood': 0,
    'perceptron': 0,
}
# Test 9 versions of the dataset/algorithms
for i in range(9):
    print(f'* * * Starting iteration {i+1} of 9 * * *')
    dataset = Dataset(data_file=DATA_IN_FILE, shuffle_data=True)
    dataset.partitionXTrain(0.8)
    print(f'Starting KNN')
    knn = KNN(dataset, k=K)
    knn.evalTest()
    logMetrics(knn, 'knn')
    bestDecision(knn, 'knn', KNN_DEC_BOUND,
            f'k-Nearest Neighbors Decision Boundary @ k={knn.k}')
    print(f'Starting Neighborhood')
    neighborhood = Neighborhood(dataset, R=R)
    neighborhood.evalTest()
    logMetrics(neighborhood, 'neighborhood')
    bestDecision(neighborhood, 'neighborhood', NEIGHBOR_DEC_BOUND,
            f'Neighborhood Decision Boundary @ R={neighborhood.R}')
    print(f'Starting Perceptron')
    perceptron = Perceptron(dataset, learning_rate=LEARNING_RATE)
    perceptron.train(epochs=EPOCHS)
    perceptron.evalTrain()
    logMetrics(perceptron, 'perc_train')
    perceptron.evalTest()
    logMetrics(perceptron, 'perc_test')
    train_err.append(np.subtract(1, perceptron.train_acc))
    bestDecision(perceptron, 'perceptron', PERC_DEC_BOUND,
            f'Perceptron Decision Boundary')
# Individual trial performance
plotPerf(['knn'], KNN_PERF, 'KNN Performance')
plotPerf(['neighborhood'], NEIGHBOR_PERF, 'Neighborhood Performance')
plotPerf(['perc_train', 'perc_test'], PERC_PERF, 'Perceptron Performance',
        legend=['Train', 'Test'])
# Average performance
metrics = [bal_acc, precision, recall, f1]
labels = ['Balanced Accuracy', 'Precision', 'Recall', 'F1 Score']
keys = ['knn', 'neighborhood', 'perc_test']
with open(str(AVG_PERF_TAB), 'w') as csv:
    csv.write(',KNN,Neighborhood,Perceptron\n')
    for metric, metric_l in zip(metrics, labels):
        write_str = f'{metric_l},'
        for i in range(len(keys)):
            mean = np.mean(metric[keys[i]])
            std = np.std(metric[keys[i]])
            write_str += f'${mean:0.3f} \pm {std:0.3f}$'
            if i == (len(keys) - 1):
                write_str += '\n'
            else:
                write_str += ','
        csv.write(write_str)
# Trial-wise training error
plt.figure()
plt.xlabel('Epoch')
plt.ylabel('Error')
```

```
plt.title('Trial-Wise Training Error vs Epochs')
for err in train_err:
    plt.plot(perceptron.epoch_nums, err)
plt.legend([f'Trial {i}' for i in range(1, 10)], loc='upper right')
plt.savefig(str(TRIAL_ERR))
plt.close()
# Mean training error
mean_err = []
std_err = []
for i in range(len(train_err[0])):
    points = []
    for j in range(len(train_err)):
        points.append(train_err[j][i])
    mean_err.append(np.mean(points))
    std_err.append(np.std(points))
plt.figure()
plt.errorbar(perceptron.epoch_nums, mean_err, yerr=std_err)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Mean Training Error vs Epochs')
plt.savefig(str(MEAN_ERR))
plt.close()
```