Assignment1

November 3, 2025

1 Assignment 1

This is my submission for Assignment 1 for my Machine Learning module (CS3920). Below is the core functionality of my machine learning algorithm. I opted to use the nearest neighbour (NN) method and a conformal predictor.

[12]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split


iris = load_iris()
iris_label_space = np.array([0,1,2])

ion_X = np.genfromtxt("ionosphere.txt", delimiter=",", usecols=np.arange(34))
ion_y = np.genfromtxt("ionosphere.txt", delimiter=",", usecols=34, dtype="int")
ion_label_space = np.array([1,-1])

iris_X_train, iris_X_test, iris_y_train, iris_y_test =␣
 ↪train_test_split(iris['data'], iris['target'], random_state=2408)
ion_X_train, ion_X_test, ion_y_train, ion_y_test = train_test_split(ion_X,␣
 ↪ion_y, random_state=2408)


def computeEuclideanNorm(vector: np.ndarray) -> float:
    """
        Computes the Euclidean norm of a vector by adding the squares of each␣
 ↪value and square rooting
    """

    total = 0
    for i in range(0,vector.size):
        total += vector[i] * vector[i]

    return np.sqrt(total)
```

```
def calculateEuclideanDistance(v1: np.ndarray, v2: np.ndarray) -> float:
    """
        Calculates the Euclidean distance between two points by computing the␣
 ↪Euclidean norm of the vector distance
    """
    diff = np.subtract(v1, v2)
    return (computeEuclideanNorm(diff))


def computeDistances(sample: np.ndarray, training_set: np.ndarray) -> np.
 ↪ndarray:
    """
        Calculates the distances from the given sample to all other points
    """

    result = np.zeros(len(training_set))
    for i in range(0, len(training_set)):
        result[i] = calculateEuclideanDistance(sample, training_set[i])

    return result


def computeMinimum(a: np.ndarray):
    """
        Calculates the minimum value of an array and returns it, along with its␣
 ↪index
    """

    current_min = np.inf
    min_index = np.inf
    for n in range(a.size):
        if current_min > a[n]:
            current_min = a[n]
            min_index = n

    return current_min, min_index


def computeMaximum(a: np.ndarray):
    """
        Calculates the maximum value of an array and returns it, along with its␣
 ↪index
    """
    current_max = -np.inf
    max_index = -np.inf
    for i in range(a.size):
```

```
        if current_max < a[i]:
            current_max = a[i]
            max_index = i

    return current_max, max_index


# IMPROVED CALCULATE NNs
def calculateNNs(index: int, sample_y: int, y_training_set: np.ndarray,␣
 ↪distances: np.ndarray):
    """
        Calculate the nearest numbers by taking in a matrix of distances
    """
    nn_dist_same = np.inf
    nn_index_same = -1
    nn_dist_diff = np.inf
    nn_index_diff = -1

    for i in range(len(y_training_set)):
        if i == index:
            continue  # skip comparing sample to itself

        dist = distances[index][i]

        if y_training_set[i] == sample_y:
            if dist < nn_dist_same:
                nn_dist_same = dist
                nn_index_same = i
        else:
            if dist < nn_dist_diff:
                nn_dist_diff = dist
                nn_index_diff = i

    return nn_dist_same, nn_index_same, nn_dist_diff, nn_index_diff

def precomputeTrainingDistances(X_train: np.ndarray) -> np.ndarray:
    """
        Precompute pairwise Euclidean distances between all training samples.
    """
    n = len(X_train)
    distances = np.zeros((n, n))

    # compute pairwise distances in training set
    for i in range(n):
        for j in range(i + 1, n):
            dist = calculateEuclideanDistance(X_train[i], X_train[j])
            distances[i][j] = dist
```

```
            distances[j][i] = dist

    return distances


def computeSampleToTrainingDistances(sample: np.ndarray, X_train: np.ndarray)␣
 ↪-> np.ndarray:
    """
        Compute Euclidean distance from one sample to all training samples.
    """

    n = len(X_train)
    dists = np.zeros(n)
    for i in range(n):
        total = 0.0
        for k in range(X_train.shape[1]):
            diff = sample[k] - X_train[i][k]
            total += diff * diff
        dists[i] = np.sqrt(total)

    return dists


def calculateConformityScores(sample_X, sample_y, X_train, y_train,␣
 ↪precomputed_train_dists):
    """
    Calculates conformity scores using precomputed training distances and␣
 ↪manual computation.
    """
    d_sample_to_train = computeSampleToTrainingDistances(sample_X, X_train)

    n = len(X_train) + 1
    distances = np.zeros((n, n))

    # Insert precomputed distances
    for i in range(len(X_train)):
        for j in range(len(X_train)):
            distances[i + 1][j + 1] = precomputed_train_dists[i][j]

    # Add test sample distances
    for i in range(len(X_train)):
        distances[0][i + 1] = d_sample_to_train[i]
        distances[i + 1][0] = d_sample_to_train[i]

    # Augment labels
    y_aug = np.zeros(n, dtype=int)
    y_aug[0] = sample_y
```

```
    for i in range(1, n):
        y_aug[i] = y_train[i - 1]

    # Compute conformity scores manually
    scores = np.zeros(n)
    for i in range(n):
        nn_dist_same, _, nn_dist_diff, _ = calculateNNs(i, y_aug[i], y_aug,␣
 ↪distances)
        if nn_dist_same == 0:
            if nn_dist_diff == 0:
                scores[i] = 0
            else:
                scores[i] = float("inf")
        else:
            scores[i] = nn_dist_diff / nn_dist_same

    return scores


def calculatePValue(scores: np.ndarray):
    """
        Calculate a given p-value given the conformity scores
    """
    test_score = scores[0]
    other_scores = scores[1:]

    rank = 0
    for score in other_scores:
        if score <= test_score:
            rank += 1

    p_value = (rank + 1) / len(scores)

    return p_value


def predict(Y: np.ndarray, sample_X: np.ndarray, X_training_set: np.ndarray,␣
 ↪y_training_set: np.ndarray, precomp_dists):
    """
        Makes a prediction by computing p-values of all possible labels and␣
 ↪selecting the highest one
    """

    Y_size = len(Y)
    p_values = np.zeros(Y_size)

    for i in range (0, Y_size):
```

```
        label = Y[i]
        scores = calculateConformityScores(sample_X, label, X_training_set,␣
↪y_training_set, precomp_dists)
        p_values[i] = calculatePValue(scores)

    _, p = computeMaximum(p_values)

    return Y[p]


def score(predictions: np.ndarray, y_test_set: np.ndarray):
    """
        Score the prediction set against the set of actual labels
    """
    return np.mean(predictions == y_test_set)
```

1.1 Improvements

I noticed using some older implementations of certain algorithms (particularly the NN algorithm) massively impacted running times. Where possible, I tried to implement mitigations to speed up running times.

1.1.1 Nearest Neighbour Algorithm

Below is an old implementation of my nearest neighbour algorithm.

[13]:
```
# OLD IMPLEMENTATION
def calculateNNsOld(sample_X: np.ndarray, sample_y: np.ndarray, X_training_set:␣
↪np.ndarray, y_training_set: np.ndarray):
    nn_dist_same = np.inf
    nn_index_same = np.inf
    nn_dist_diff = np.inf
    nn_index_diff = np.inf

    for i in range(0,len(X_training_set)):
        if y_training_set[i] == sample_y:
            d = calculateEuclideanDistance(sample_X, X_training_set[i])
            if d < nn_dist_same:
                nn_dist_same = d
                nn_index_same = i
        else:
            d = calculateEuclideanDistance(sample_X, X_training_set[i])
            if d < nn_dist_diff:
                nn_dist_diff = d
                nn_index_diff = i

    # print("Nearest (same):", X_training_set[nn_index_same], "Class:",␣
↪y_training_set[nn_index_same], "Distance:", nn_dist_same)
```

```
    # print("Nearest (diff):", X_training_set[nn_index_diff], "Class:",␣
↪y_training_set[nn_index_diff], "Distance:", nn_dist_diff)

    return nn_dist_same, nn_index_same, nn_dist_diff, nn_index_diff
```

I noticed this massively hindered performance as the distances to every other sample were being recalculated every single time, giving $O(n^2)$. I then opted to improve this by using a matrix data structure to store the distances, so that they only need to be computed once, and can be accessed multiple times. This improved running time significantly. Below I will demonstrate an experiment using the old implementation of the `calculateConformityScores` method, and compare it to the new one.

[14]:
```
def calculateConformityScoresOld(sample_X, sample_y, X_training_set,␣
↪y_training_set):
    X_aug = np.concatenate((sample_X.reshape(1, -1), X_training_set), axis=0)
    y_aug = np.concatenate(([sample_y], y_training_set)) # additional sample␣
↪will be the FIRST in augmented set

    scores = np.zeros(len(y_aug))

    for i in range(0, len(y_aug)):
        X_new = np.delete(X_aug, i, axis=0)
        y_new = np.delete(y_aug, i)
        nn_dist_same, _, nn_dist_diff, _ = calculateNNsOld(X_aug[i], y_aug[i],␣
↪X_new, y_new)
        conformity_score = 0
        if nn_dist_same == 0:
            if nn_dist_diff == 0:
                conformity_score = 0
            else:
                conformity_score = np.inf
        else:
            conformity_score =  nn_dist_diff / nn_dist_same

        scores[i] = conformity_score

    return scores
```

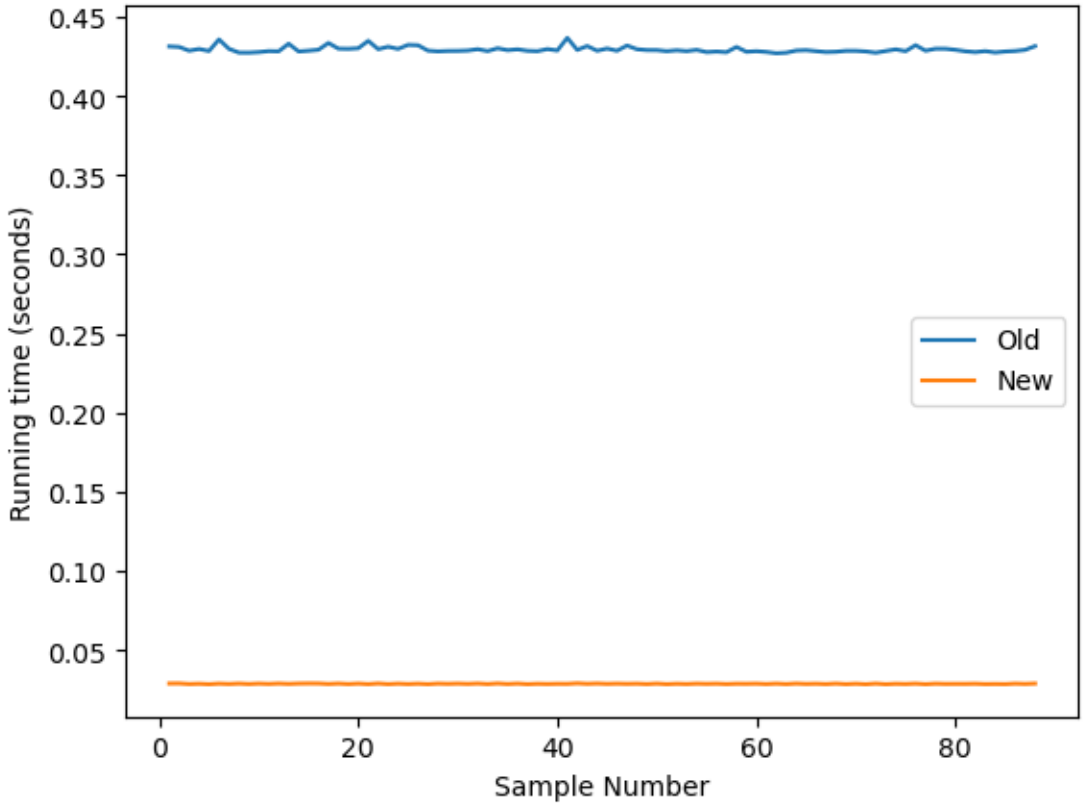
[15]:
```
old_times_ion = np.zeros(len(ion_X_test))
ion_dists = precomputeTrainingDistances(ion_X_train)
for i in range(0, len(ion_X_test)):
    start_old = time.time()
    calculateConformityScoresOld(ion_X_test[i], ion_y_test[i],ion_X_train,␣
↪ion_y_train)
    o = time.time() - start_old
    old_times_ion[i] = o
```

```
new_times_ion = np.zeros(len(ion_X_test))
for j in range(0,len(ion_X_test)):
    start_new = time.time()
    calculateConformityScores(ion_X_test[0], ion_y_test[0], ion_X_train,␣
 ↪ion_y_train, ion_dists)
    n = time.time() - start_new
    new_times_ion[j] = n

x = np.arange(1, len(ion_X_test) + 1)
plt.plot(x, old_times_ion, label="Old")
plt.plot(x, new_times_ion, label="New")
plt.xlabel("Sample Number")
plt.ylabel("Running time (seconds)")
plt.legend()
plt.show()

print("Old implementation completed in average:", np.mean(old_times_ion),␣
 ↪"seconds")
print("New implementation completed in average:", np.mean(new_times_ion),␣
 ↪"seconds")

print("\nIncrease in efficiency of new implementation: {:.2f}%".format((np.
 ↪mean(old_times_ion)/np.mean(new_times_ion)) * 100))
```



```
Old implementation completed in average: 0.42930940877307544 seconds
New implementation completed in average: 0.028653767975893887 seconds

Increase in efficiency of new implementation: 1498.27%
```

1.1.2 Calculating Conformity Scores

An issue that seriously slowed down my program was that I was calculating distances *every time* I wanted to calculate a conformity score. I solved this by precomputing the distances between all samples in the training set, so they only needed to be calculated once. Now, I only need to calculate the distance including the training sample.

[16]:
```python
def calculateConformityScoresOld(sample_X: np.ndarray, sample_y: np.ndarray, X_training_set: np.ndarray, y_training_set: np.ndarray):
    """
        Calculates the conformity score of a sample, using the NN of the same class / NN of different class.
    """

    X_aug = np.concatenate((sample_X.reshape(1, -1), X_training_set), axis=0)
    y_aug = np.concatenate(([sample_y], y_training_set)) # additional sample will be the FIRST in augmented set

    n = len(X_aug)
    scores = np.zeros(n)

    distances = np.zeros((n, n))

    # compute all the distances and store in matrix
    for i in range(0, n):
        for j in range(i + 1, n):
            dist = calculateEuclideanDistance(X_aug[i], X_aug[j])
            distances[i][j] = dist
            distances[j][i] = dist

    # compute nearest neighbours for each to get conformity score
    for i in range(0, n):
        nn_dist_same, _, nn_dist_diff, _ = calculateNNs(i, y_aug[i], y_aug, distances)

        if nn_dist_same == 0:
            if nn_dist_diff == 0:
                scores[i] = 0
            else:
                scores[i] = np.inf
```

```
        else:
            scores[i] =  nn_dist_diff / nn_dist_same

    return scores
```

[17]:
```
def testOldCS(test_X, test_y, train_X, train_y):
    start = time.time()
    for i in range(0, len(test_X)):
        calculateConformityScoresOld(test_X[i], test_y[i], train_X, train_y)
    end = time.time() - start

    return end

def testNewCS(test_X, test_y, train_X, train_y, precomp_dists):
    start = time.time()
    for i in range(0, len(test_X)):
        calculateConformityScores(test_X[i], test_y[i], train_X, train_y,␣
 ↪precomp_dists)
    end = time.time() - start

    return end

iris_dists = precomputeTrainingDistances(iris_X_train)
ion_dists = precomputeTrainingDistances(ion_X_train)

iris_old = testOldCS(iris_X_test, iris_y_test, iris_X_train, iris_y_train)
iris_new = testNewCS(iris_X_test, iris_y_test, iris_X_train, iris_y_train,␣
 ↪iris_dists)

ion_old = testOldCS(ion_X_test, ion_y_test, ion_X_train, ion_y_train)
ion_new = testNewCS(ion_X_test, ion_y_test, ion_X_train, ion_y_train, ion_dists)

print("Old (Iris):", iris_old, "seconds")
print("New (Iris):", iris_new, "seconds")
print("\nOld (Ion): ", ion_old,"seconds")
print("New (Ion): ", ion_new, "seconds")
```

```
Old (Iris): 0.5518229007720947 seconds
New (Iris): 0.1888282299041748 seconds

Old (Ion):  20.582809925079346 seconds
New (Ion):  2.5099782943725586 seconds
```

1.2 Calculating the Score of the Model on the Iris Dataset

[18]:
```
iris_predictions = np.zeros(len(iris_X_test))
iris_dists = precomputeTrainingDistances(iris_X_train)

for i in range(0, len(iris_X_test)):
    iris_predictions[i] = predict(iris_label_space, iris_X_test[i],␣
 ↪iris_X_train, iris_y_train, iris_dists)

s = score(iris_predictions, iris_y_test)

print("Model Score:", s,"\nError Rate:  {:.2f}%".format((1 - s) * 100))
```

```
Model Score: 0.9736842105263158
Error Rate:  2.63%
```

1.3 Calculating the Score of the Model on the Ionosphere Dataset

[19]:
```
ion_predictions = np.zeros(len(ion_X_test))
ion_dists = precomputeTrainingDistances(ion_X_train)

for j in range(0, len(ion_X_test)):
    ion_predictions[j] = predict(ion_label_space, ion_X_test[j], ion_X_train,␣
 ↪ion_y_train, ion_dists)

s = score(ion_predictions, ion_y_test)

print("Model Score:", s,"\nError Rate:  {:.2f}%".format((1 - s) * 100))
```

```
Model Score: 0.8522727272727273
Error Rate:  14.77%
```

1.4 Error Rates vs Sample Size

Here I will perform some experiments to determine how the sample size affects the error rate of the model.

[20]:
```
iris_sizes = np.arange(1, len(iris_X_test) + 1)
ion_sizes = np.arange(10, len(ion_X_test) + 1, 10)

iris_error_rates = []
ion_error_rates = []

def testErrorRates(label_space, X_test, X_train, y_test, y_train, sizes,␣
 ↪precomp_dists):
    error_rates = []
    for size in sizes:
        temp_x_test = X_test[:size]
```

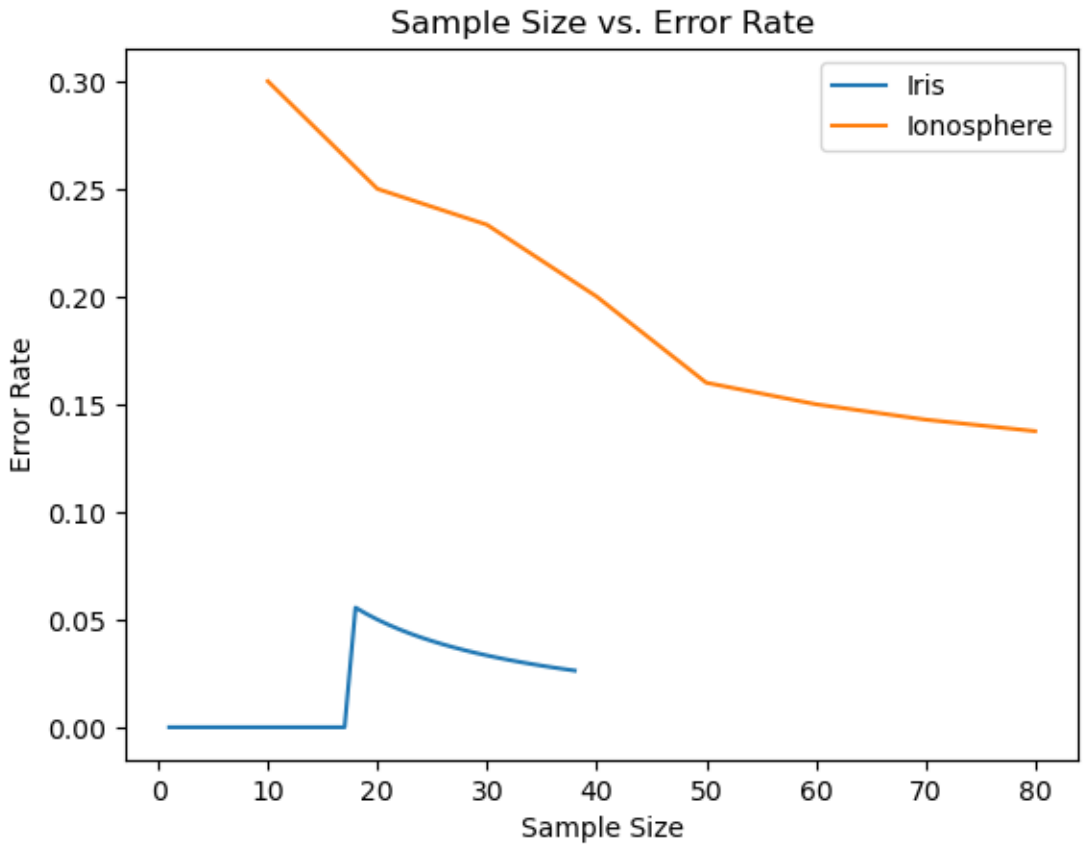
```
        temp_y_test = y_test[:size]
        predictions = np.zeros(size)
        for i in range(0,size):
            predictions[i] = predict(label_space, temp_x_test[i], X_train,␣
 ↪y_train, precomp_dists)

        error = 1 - score(predictions, temp_y_test)
        error_rates.append(error)

    return error_rates

iris_error_rates = testErrorRates(iris_label_space, iris_X_test, iris_X_train,␣
 ↪iris_y_test, iris_y_train, iris_sizes, iris_dists)
ion_error_rates = testErrorRates(ion_label_space, ion_X_test, ion_X_train,␣
 ↪ion_y_test, ion_y_train, ion_sizes, ion_dists)

plt.plot(iris_sizes, iris_error_rates, label="Iris")
plt.plot(ion_sizes, ion_error_rates, label="Ionosphere")
plt.legend()
plt.title("Sample Size vs. Error Rate")
plt.xlabel("Sample Size")
plt.ylabel("Error Rate")
plt.show()
```



1.5 Calculating the Average False P-Values

Before calculating the average false p-values for both datasets, I will first lay out the function I am going to use to do this. The function takes in the X and y training sets, as well as the label space Y and the true label for the sample. The function works on a sample-by-sample basis, so for each sample it calculates the average false p-values. It does this by checking if the label we are currently calculating a p-value for is the *true* label. If so, we can skip it. If not, we calculate the conformity scores, the p_value, and add it to the list of p-values. From this, we simply calculate the average using `np.mean`.

[21]:
```
def calculateAverageFalsePValues(Y, sample_X, X_training_set, y_training_set,␣
↪true_y, precomp_dists):
    Y_size = len(Y)
    false_p_values = []

    for i in range (0, Y_size):
        label = Y[i]
        if label != true_y:
            scores = calculateConformityScores(sample_X, label, X_training_set,␣
↪y_training_set, precomp_dists)
```

```
            p_value = calculatePValue(scores)
            false_p_values.append(p_value)

    return np.mean(false_p_values)
```

We can now apply this function to the Iris dataset.

[22]:
```
total_false_p_values = np.zeros(len(iris_X_test))
for i in range(0,len(iris_X_test)):
    total_false_p_values[i] = calculateAverageFalsePValues(iris_label_space,␣
 ↪iris_X_test[i], iris_X_train, iris_y_train, iris_y_test[i], iris_dists)

print("Total average false p-value:", np.mean(total_false_p_values))
```

```
Total average false p-value: 0.011178388448998605
```

And now the same for the Ionosphere dataset.

[23]:
```
total_false_p_values = np.zeros(len(ion_X_test))
for i in range(0,len(ion_X_test)):
    total_false_p_values[i] = calculateAverageFalsePValues(ion_label_space,␣
 ↪ion_X_test[i], ion_X_train, ion_y_train, ion_y_test[i], ion_dists)

print("Total average false p-value:", np.mean(total_false_p_values))
```

```
Total average false p-value: 0.0625
```

1.6 Calculating the Average Credibility

I will test the average credibility by slightly modifying the `predict` function to just return the maximum p-value.

[24]:
```
def calculateCredibility(Y, sample_X, X_training_set, y_training_set,␣
 ↪precomp_dists):

    Y_size = len(Y)
    p_values = np.zeros(Y_size)
    for i in range (0, Y_size):
        label = Y[i]
        scores = calculateConformityScores(sample_X, label, X_training_set,␣
 ↪y_training_set, precomp_dists)
        p_values[i] = calculatePValue(scores)

    p,_  = computeMaximum(p_values)

    return p

def getAverageCredibility(Y, X_test_set, X_training_set, y_training_set,␣
 ↪precomp_dists):
```

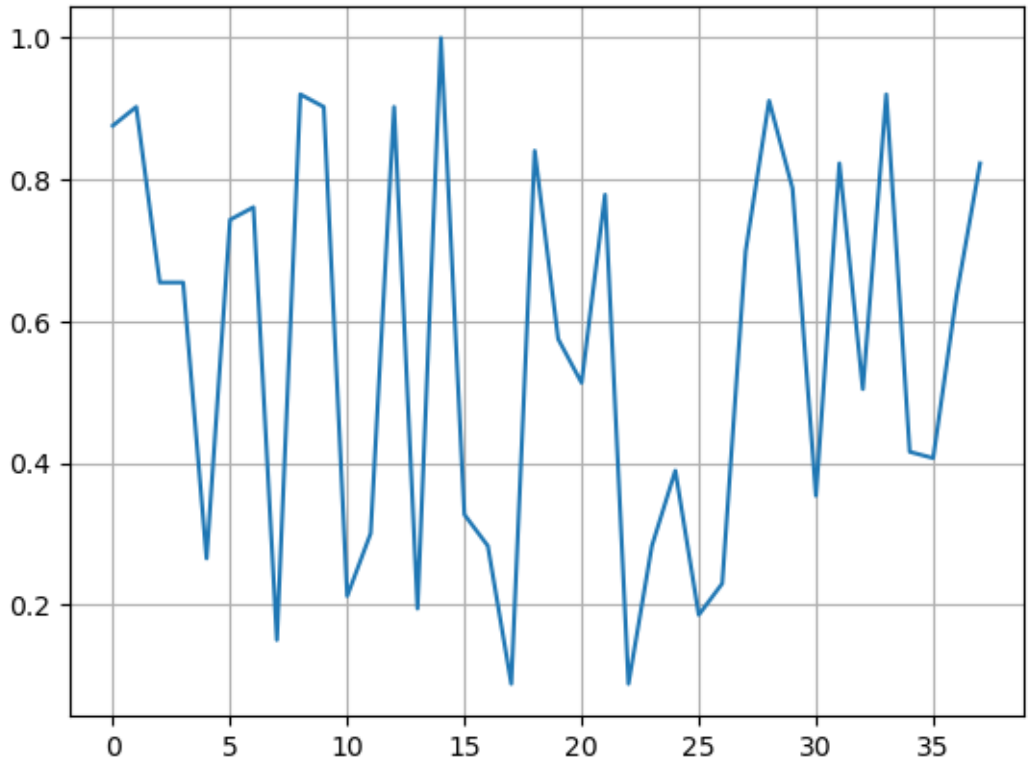
```
    credibilities = np.zeros(len(X_test_set))
    for i in range(0,len(X_test_set)):
        credibilities[i] = calculateCredibility(Y, X_test_set[i],␣
↪X_training_set, y_training_set, precomp_dists)

    x = np.arange(0, len(X_test_set))
    plt.plot(x, credibilities)
    plt.grid(True)
    plt.show()

    print("Average Credibility:", np.mean(credibilities))
```

For the iris dataset, we get the following result:

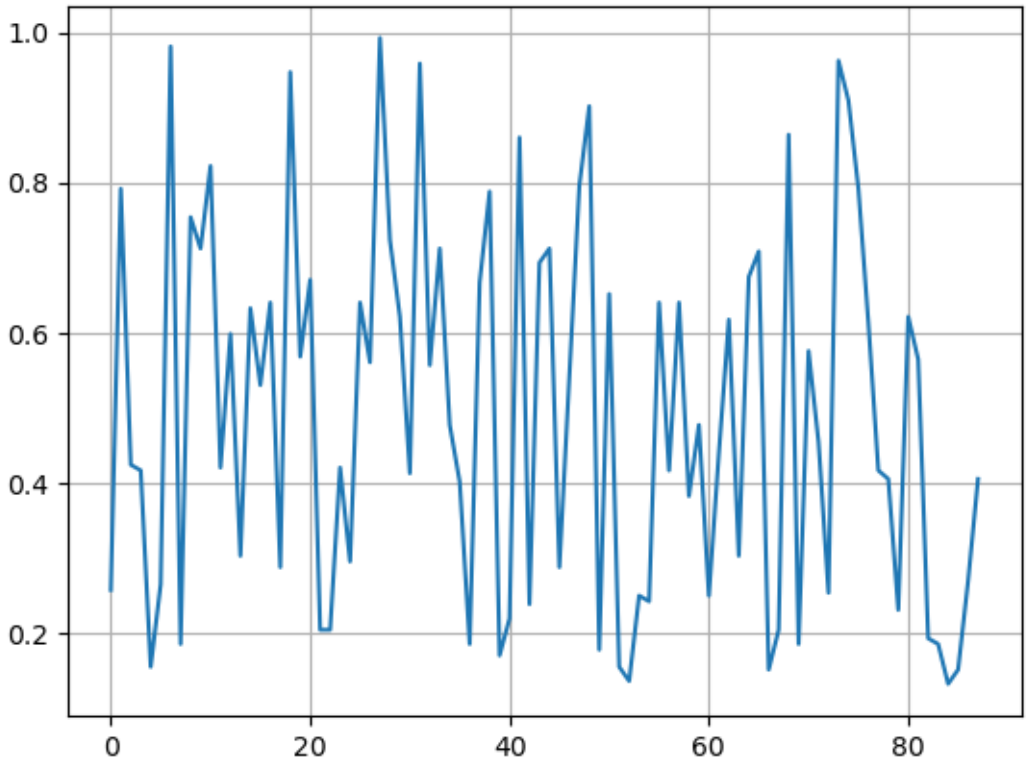
[25]:
```
getAverageCredibility(iris_label_space, iris_X_test, iris_X_train,␣
↪iris_y_train, iris_dists)
```




```
Average Credibility: 0.5607824871914298
```

For the ionosphere dataset, we get the following:

[26]:
```
getAverageCredibility(ion_label_space, ion_X_test, ion_X_train, ion_y_train,␣
↪ion_dists)
```




```
Average Credibility: 0.4921659779614325
```

1.7 Calculating the Average Confidence

Here I will calculate a the model's average confidence

[30]:
```
def calculateConfidence(Y, sample_X, X_training_set, y_training_set,␣
↪precomp_dists):
    Y_size = len(Y)
    p_values = np.zeros(Y_size)
    for i in range (0, Y_size):
        label = Y[i]
        scores = calculateConformityScores(sample_X, label, X_training_set,␣
↪y_training_set, precomp_dists)
        p_values[i] = calculatePValue(scores)

    _, p  = computeMaximum(p_values)
    np.delete(p_values, p) # delete largest value

    s, _ = computeMaximum(p_values) # calculate largest again

    return 1 - s # confidence = 1 - second biggest p value
```

```
def getAverageConfidence(Y, X_test_set, X_training_set, y_training_set,␣
 ↪precomp_dists):
    confidences = np.zeros(len(X_test_set))
    for i in range(0,len(X_test_set)):
        confidences[i] = calculateConfidence(Y, X_test_set[i], X_training_set,␣
 ↪y_training_set, precomp_dists)

    print("Average Confidence:", np.mean(confidences))
    x = np.arange(0, len(X_test_set))
    plt.plot(x, confidences)
    plt.grid(True)
    plt.show()
```

[31]:
```
getAverageConfidence(iris_label_space, iris_X_test, iris_X_train, iris_y_train,␣
 ↪iris_dists)
```

```
Average Confidence: 0.4392175128085702
```