

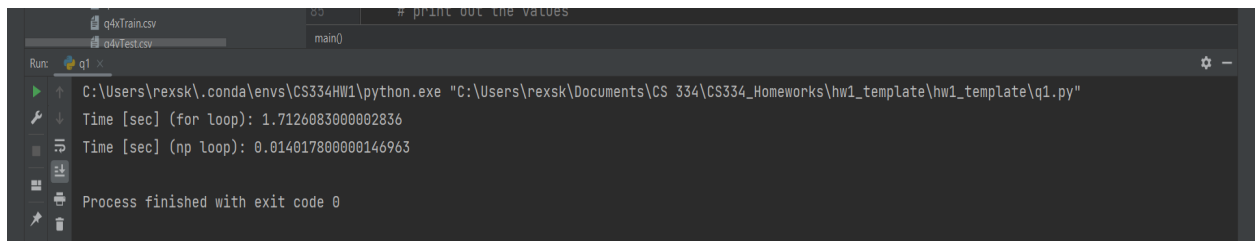
"""

THIS CODE IS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING CODE WRITTEN BY OTHER STUDENTS OR LARGE LANGUAGE MODELS SUCH AS CHATGPT.
Tommy Skodje

I collaborated with the following classmates for this homework:
None

"""

1. d.



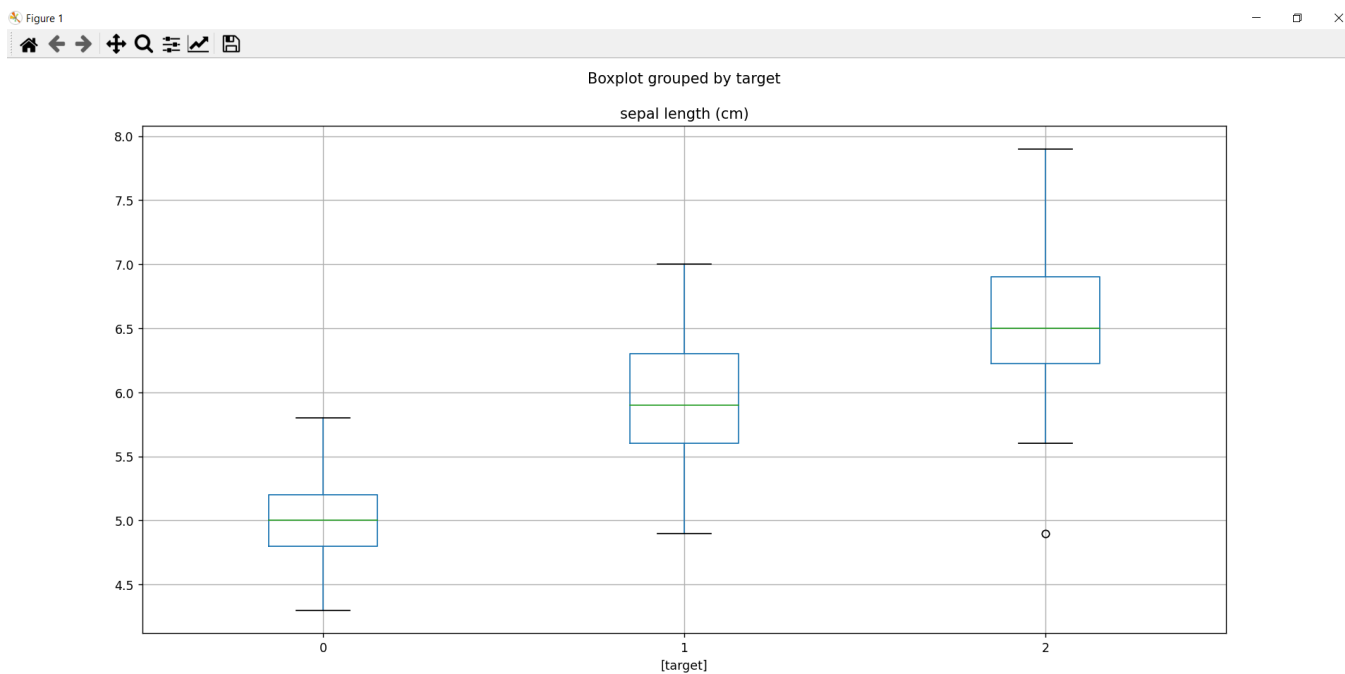
```
q4xTrain.csv      83      # print out the values
q4xTest.csv       main()

Run: q1 x
C:\Users\rexsk\.conda\envs\CS334HW1\python.exe "C:\Users\rexsk\Documents\CS 334\CS334_Homeworks\hw1_template\hw1_template\q1.py"
Time [sec] (for loop): 1.7126083000002836
Time [sec] (np loop): 0.014017800000146963
Process finished with exit code 0
```

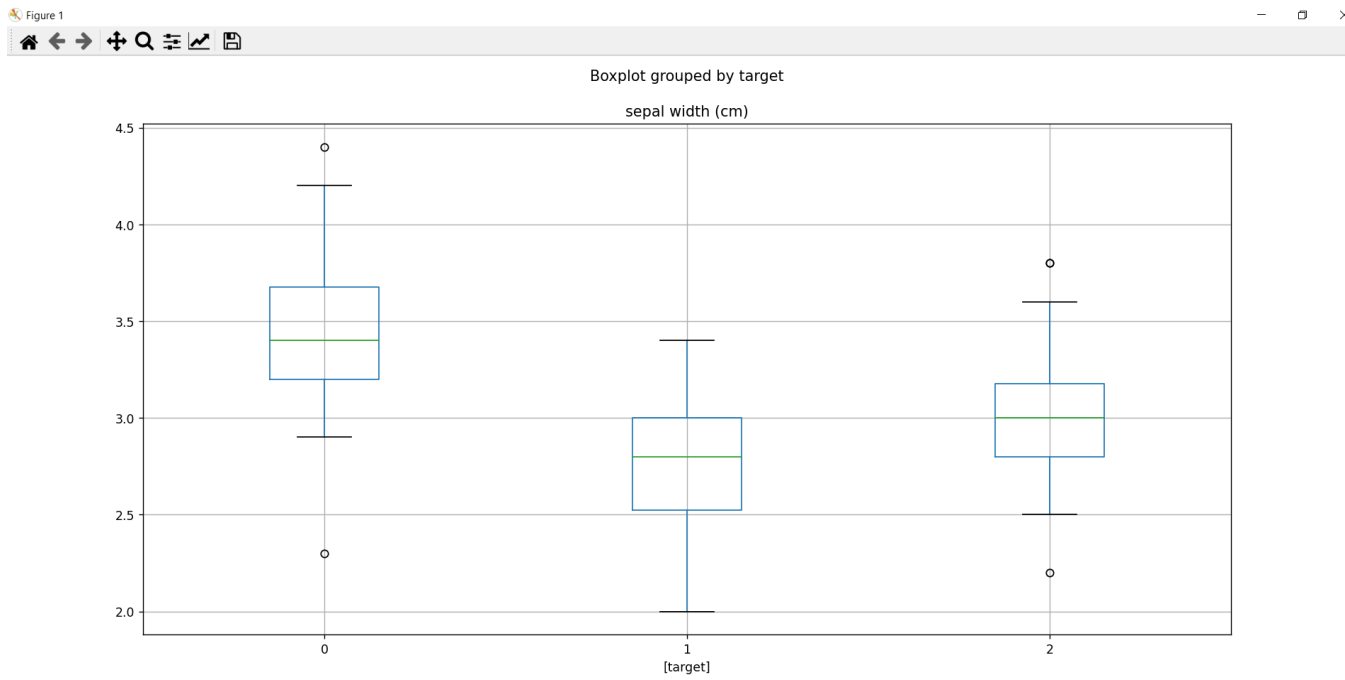
As seen in the screenshot above, the for loop ran in 1.7126083000002836 seconds while the vectorized approach ran in 0.014017800000146963 seconds. This means that the vectorized approach ran about 1.6985905 seconds faster than the for loop approach.

2. b.

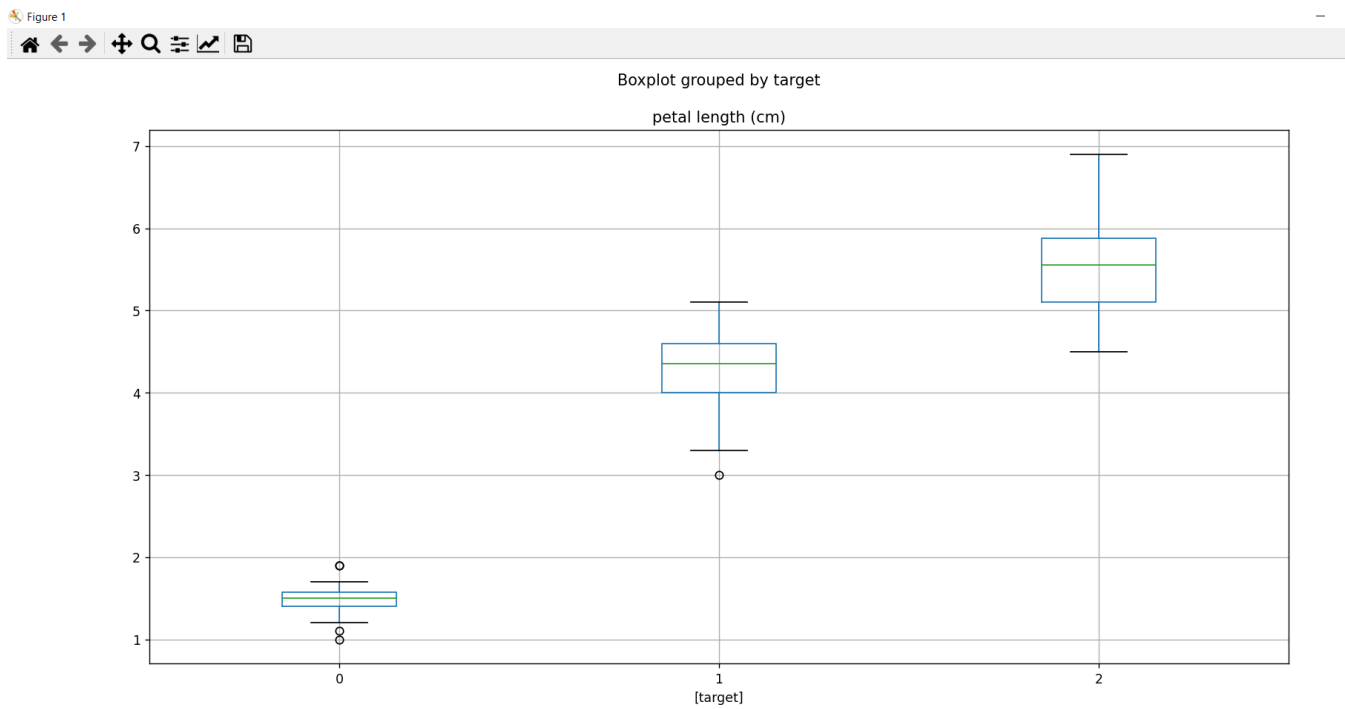
Sepal Length Boxplot



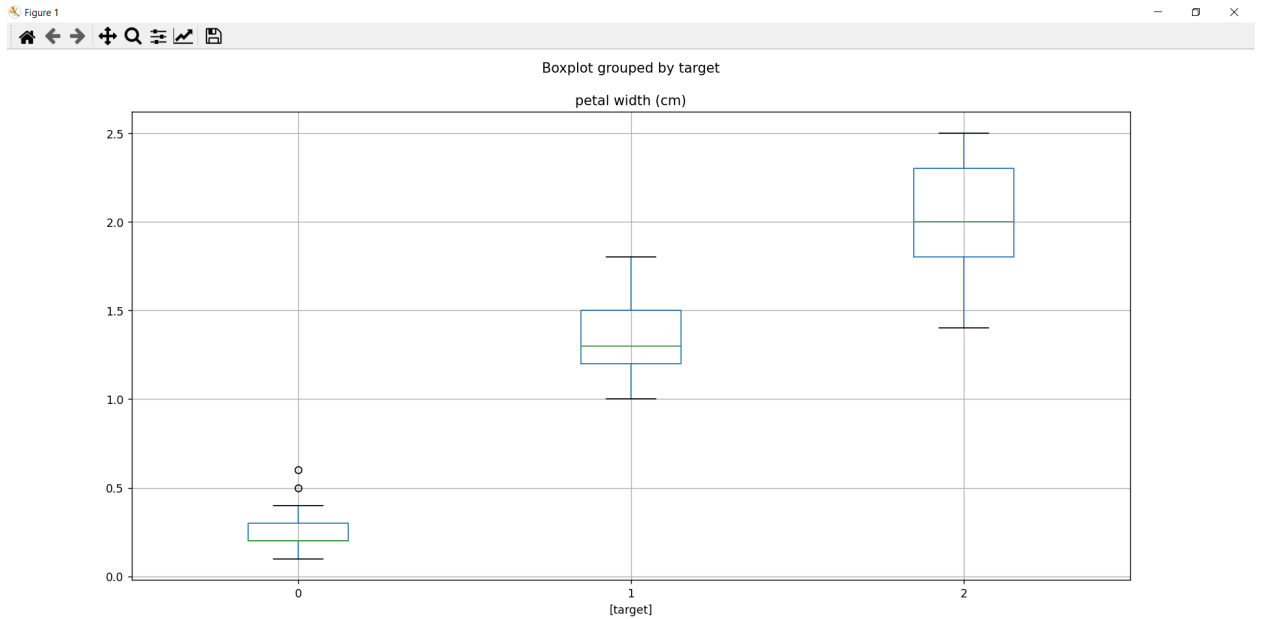
Sepal Width Boxplot



Petal Length Boxplot

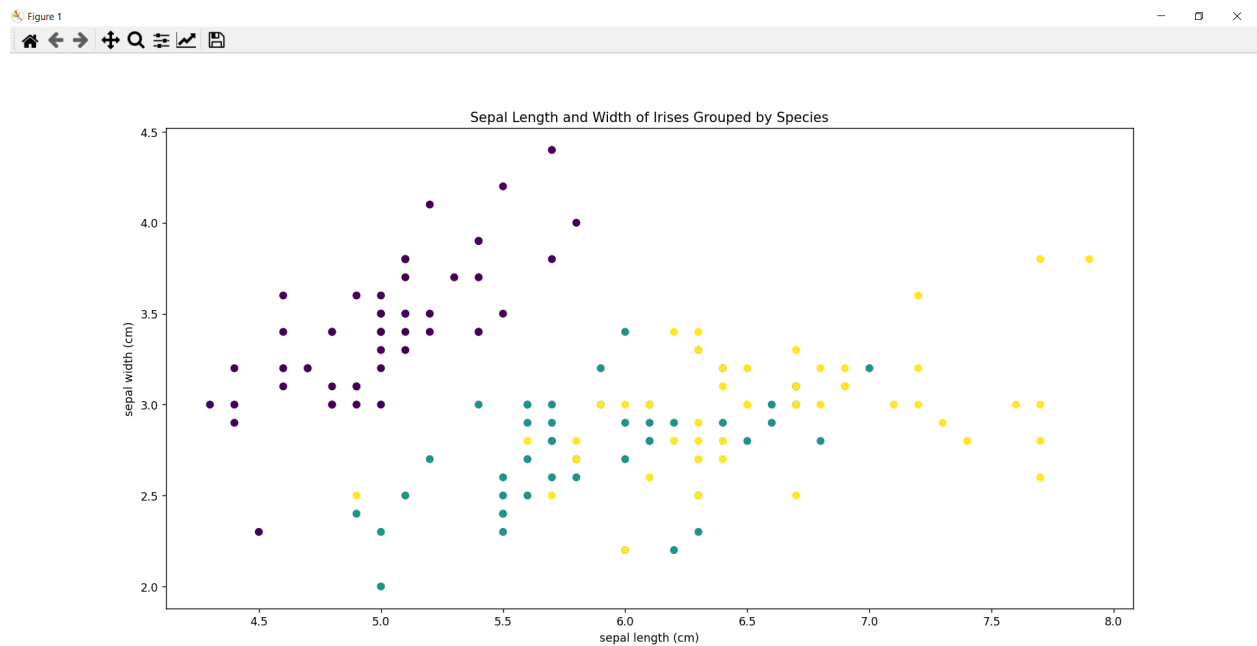


Petal Width Boxplot

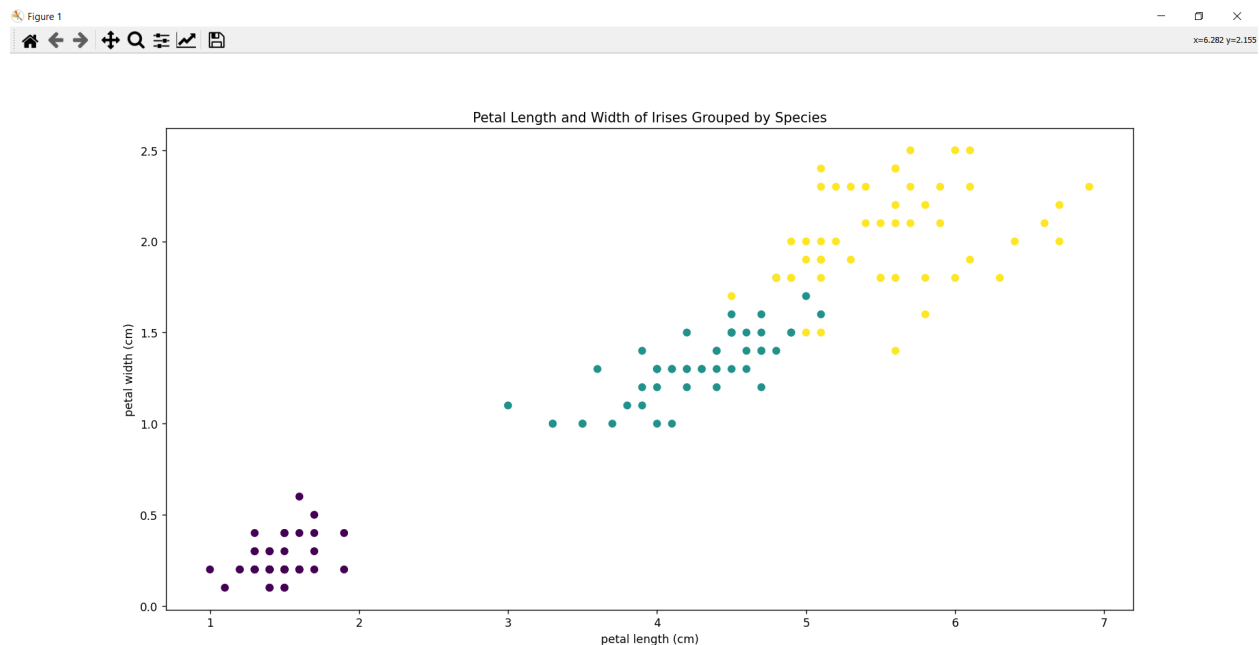


2. c.

Sepal Plot



Petal Scatterplot



2. d. Group 0 (Sentosa Irises), tend to have the shortest sepal length at an average of around 5 cm, group 1 (Versicolour Irises) have an average length of about 6 cm, and group 2 have the largest average sepal length at about 6.5 cm. As for sepal width, group 0 has an average width of just under 3.5 cm, group 1 has an average of about 2.75 cm, and group 2 has an average sepal width of about 3.0 cm.

Looking solely at the sepals of the irises, from the scatterplot we can see that the divisions between the groups are:

- Group 0 (purple) tends to have sepal lengths of under 6.0 cm and sepal widths of over 3.0 cm.
- Group 1 (green) has sepal lengths usually somewhere between 5.0 cm and 6.5 cm and sepal widths under 3.0 cm.
- Group 2 (yellow) tends to have sepal lengths over 6.0 cm and sepal widths between 2.5 cm and 3.5 cm.

However, group 1 and group 2 are very mixed together at some points, so it is necessary to consider petal length and petal width as well.

Group 0 has both the shortest and narrowest petals on average, with about 1.5 cm and 0.25 cm respectively. Group 1 is in the middle for both petal length and width, with 4.5 cm and 1.25 cm on average for each. Group 2 has both the longest and widest petals on average, with 5.5 cm and 2.0 cm.

We can see this trend continue on the scatterplot, with there being clear distinctions between each group.

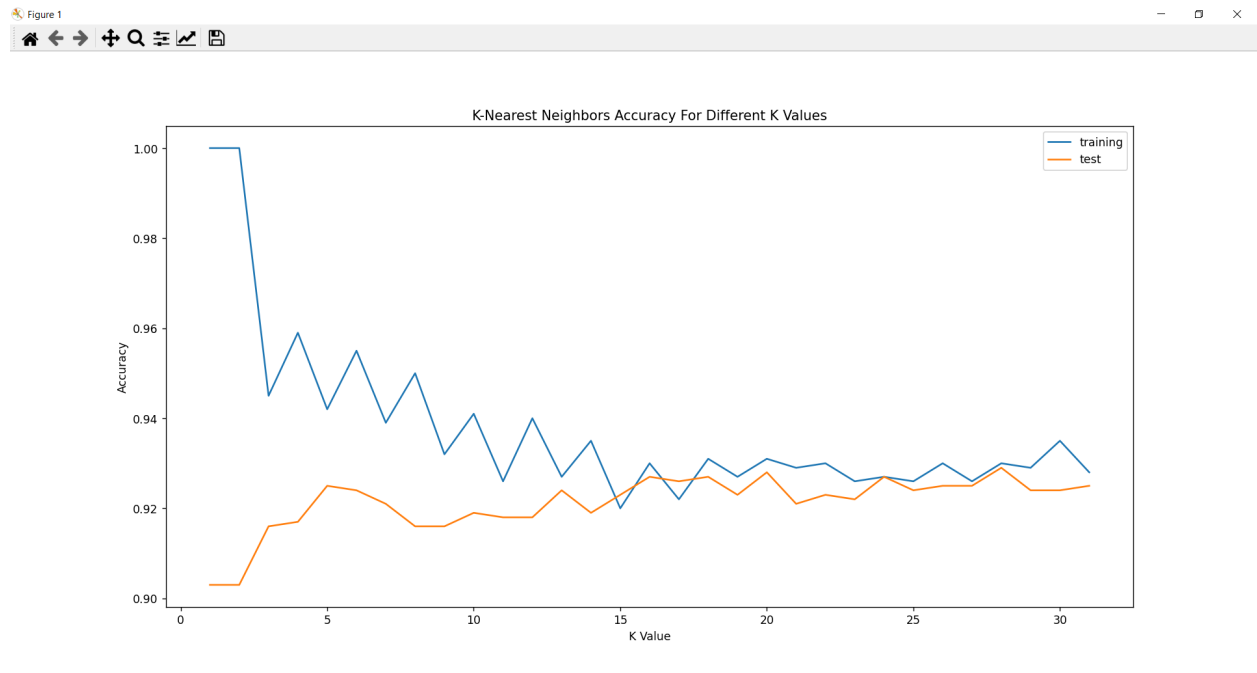
- Group 0 has petals less than about 2.0 cm in length and 0.75 cm in width.
- Group 1 has petal lengths between about 3.0 cm and 5.0 cm, and petal widths between about 1 cm and 1.5 cm.
- Group 2 has petal lengths greater than about 5.0 cm and wider than 1.5 cm for most data points.

After analyzing the lengths and widths of both sepals and petals, some “rules” that could classify the species types could be.

1. If an iris has petals less than 2.0 cm in length and 0.75 cm in width, and also sepals less than 6.0 cm in length and greater than 3.0 cm in width, it is likely a *Sentosa* Iris (group 0).
2. If an iris has petal lengths between 3.0 cm and 5.0 cm, petal widths between 1 cm and 1.5 cm, sepal lengths between 5.0 cm and 6.5 cm, and sepal widths under 3.0 cm, then it is likely a *Versicolour* Iris (group 1)
3. If an iris has petals greater than 5.0 cm in length and greater than 1.5 cm in width, and sepals greater than 6.0 cm in length and between 2.5 cm and 3.5 cm in width, then it is likely a *Virginica* Iris (group 2)
4. If an Iris does not fit into any of these groups, find the closest group in terms of all categories (petal length, petal width, sepal length, sepal width)

3. c. Given an array of predicted values and the true labels, and setting k to 1, the percentage of correctly classified samples for the training data was 100%, and the percentage of correctly classified samples for the test data was 90.3%.

3. d.



The plot above shows the accuracy for training and test data for k values of between 1 and \sqrt{n} . As seen in the chart above, the training accuracy for K-Nearest Neighbors is a perfect 100% for $k = 1$ and $k = 2$, but drops off drastically afterwards. The accuracy stabilizes after around $k = 20$. In contrast, the training data starts at a relatively low accuracy just above 90%, and increases steadily, peaking just before $k = 30$.

3. e. The predict function implemented in question 3b starts with the nested for loop shown below:

```
for test_row in self.test_data:
    distances = []
    for training_row in self.training_data:
        # Find the Euclidean distance for each data point
        sum_of_squared_differences = np.sum((training_row -
        test_row) ** 2)
        euc_distance = np.sqrt(sum_of_squared_differences)
        distances.append(euc_distance)
```

There are n test rows and n training rows, so the time taken for these lines of code will grow at a rate of $O(n^2)$

The operations of instantiating the distances list, taking the square root of the sum of squared differences, and appending the Euclidean Distance to the distances list all take constant time, so they do not factor into the overall big O time complexity of the algorithm.

Calculating the sum of squared differences is dependent on how many features (d) there are. The sum will grow linearly in relation to the number of features since each addition takes constant time. Since this takes place within the nested for loop, the time complexity for this portion of code is $O(n^2 d)$.

The next portion of the algorithm is as follows:

```
# Find the k-nearest neighbors for the test point
sorted_indices = np.argsort(distances)
k_nearest_indices = sorted_indices[:self.k]
# Find the labels for the k-nearest neighbors
k_nearest_labels = []
for j in range(self.k):
    k_nearest_labels.append(self.training_labels[k_nearest_indices[j]])

# Find which label appears the most often within k_nearest_labels
# seen_labels has format label:count to keep track of labels already seen previously.
seen_labels = {}
for j in range(len(k_nearest_labels)):
    if k_nearest_labels[j] in seen_labels:
        seen_labels[k_nearest_labels[j]] = seen_labels[k_nearest_labels[j]] + 1
    else:
        seen_labels[k_nearest_labels[j]] = 1

prediction = max(seen_labels, key=seen_labels.get)
yHat.append(prediction)
```

This entire portion of the algorithm takes place inside the outermost for loop. The operations within the for loops seen above take constant time no matter the values of n or d .

The 2 for loops in the code above both loop for the length of k, and don't rely on n or d to determine how long they run. Each doesn't have any further nesting, so they both grow linearly in relation to k. Therefore, the time complexity for this portion of code is $O(k)$.

The code above takes place entirely within the outermost for loop of the code, which runs with $O(n)$ time. Therefore, the time complexity of my implementation of the predict function in question 3b is $O(n(k)nd) = O(n^2kd)$.

4. d. Evaluating the accuracy of the model as a function of k outputs the following results, where n = no preprocessing, s = standard scale, m = min max scale, and i = with irrelevant features. I evaluated up to k = 21 (which is close to sqrt n) increasing k by intervals of 4.

When k = 1: n = 0.8583, s = 0.8396, m = 0.8813, i = 0.8562

When k = 5: n = 0.8666, s = 0.8646, m = 0.8813, i = 0.8562

When k = 9: n = 0.8689, s = 0.8646, m = 0.829, i = 0.8646

When k = 13: n = 0.8583, s = 0.8625, m = 0.8771, i = 0.8688

When k = 17: n = 0.8604, s = 0.8625, m = 0.8875, i = 0.8604

When k = 21: n = 0.8604, s = 0.8625, m = 0.8833, i = 0.8667

Analyzing the results, we can see that standard scale preprocessing causes accuracy to decrease for small values of k, but as k approaches sqrt(n), the accuracy of the model increases after standard scale is applied. Min max scaling provides a positive boost to accuracy of about 2-3% no matter the value of k. Finally, irrelevant features seem to have a very minimal effect on the dataset, as the drop in accuracy is less than 1% for all values of k, and the accuracy strangely even increases for some k values.