CS 334 — Homework 1 Solutions

Jack Anstey

Problem 1: Numerical Programming

a) See q1.py

b) See q1.py

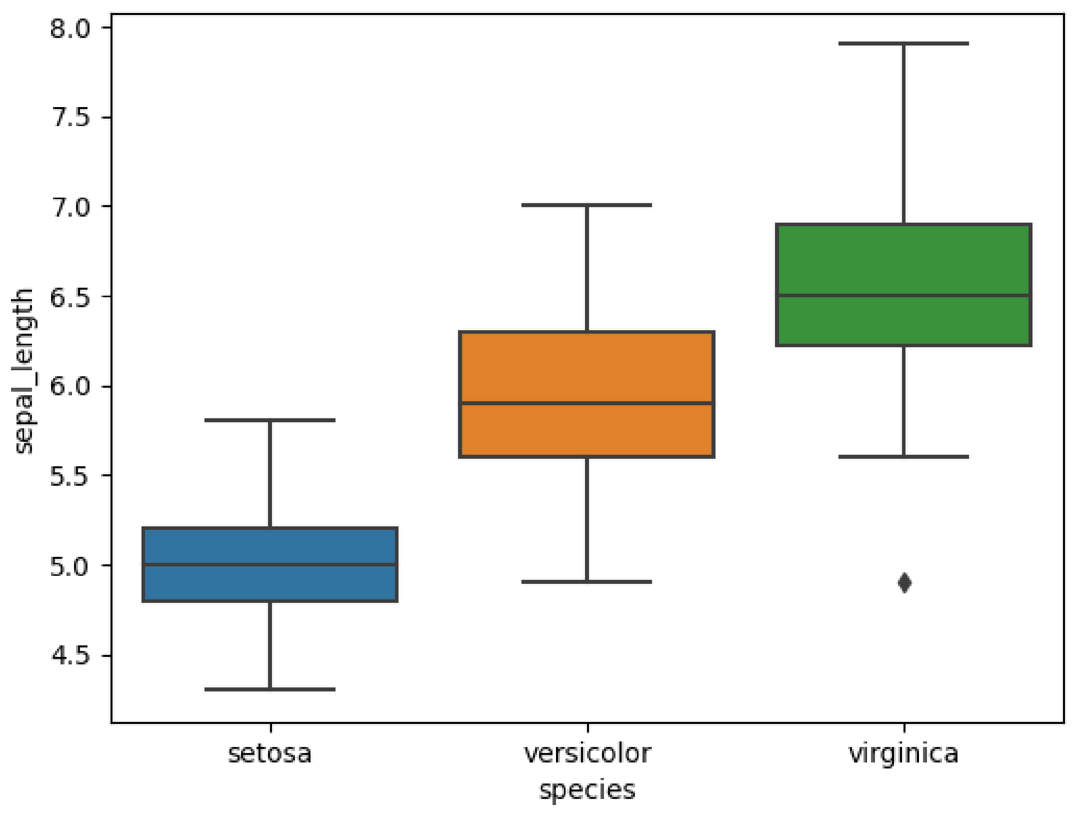
c) See q1.py

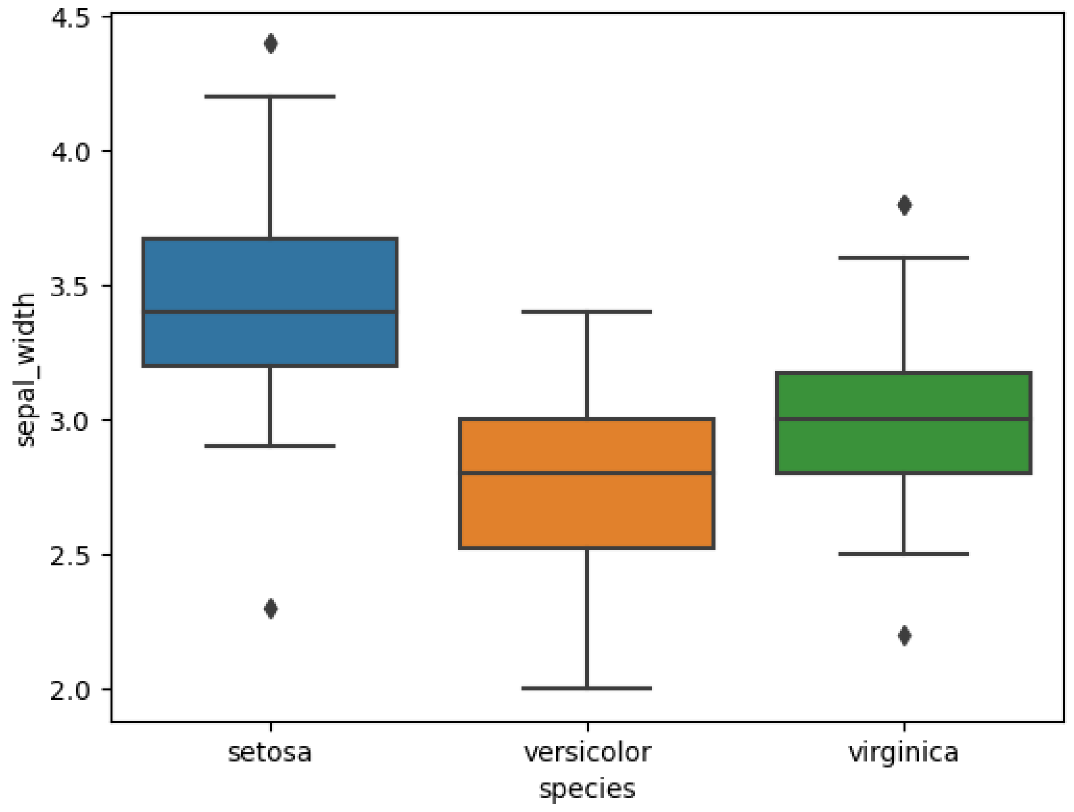
d) The vectorized approach was significantly faster than the first approach. On my laptop, the first approach took $\approx$1.783 seconds, while the np version took $\approx$0.002 seconds. That is $\approx$912.718 times faster!

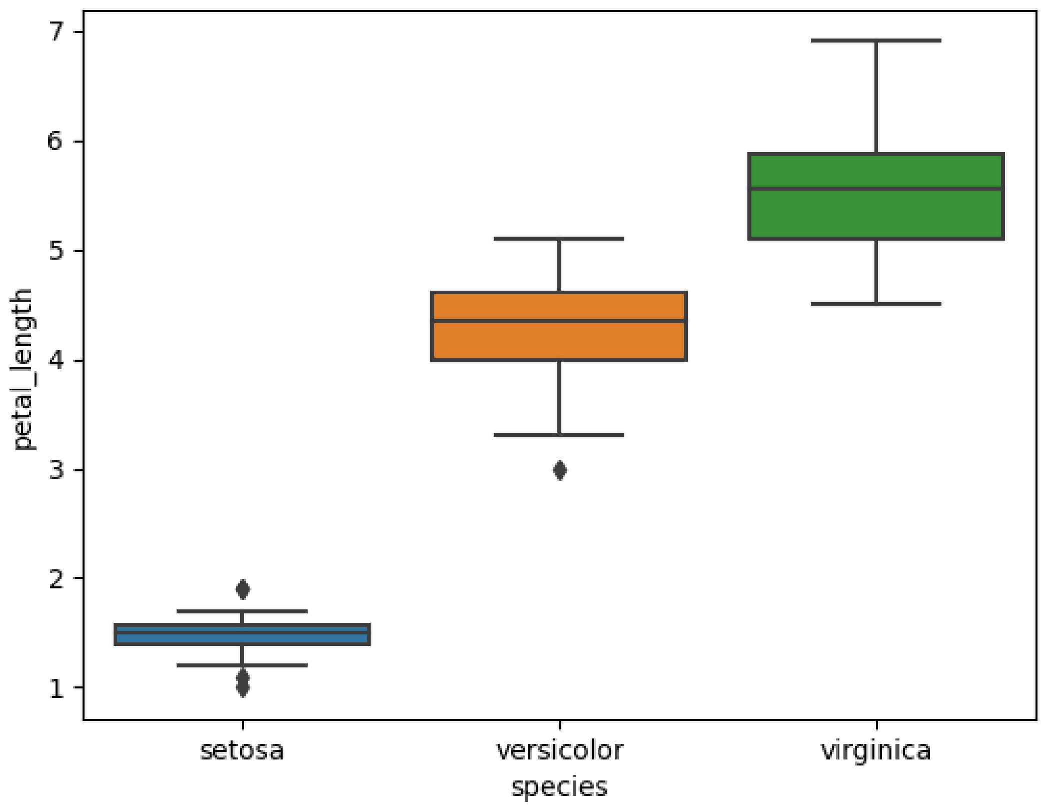
Problem 2: Visualization Exploration

a) See q2.py

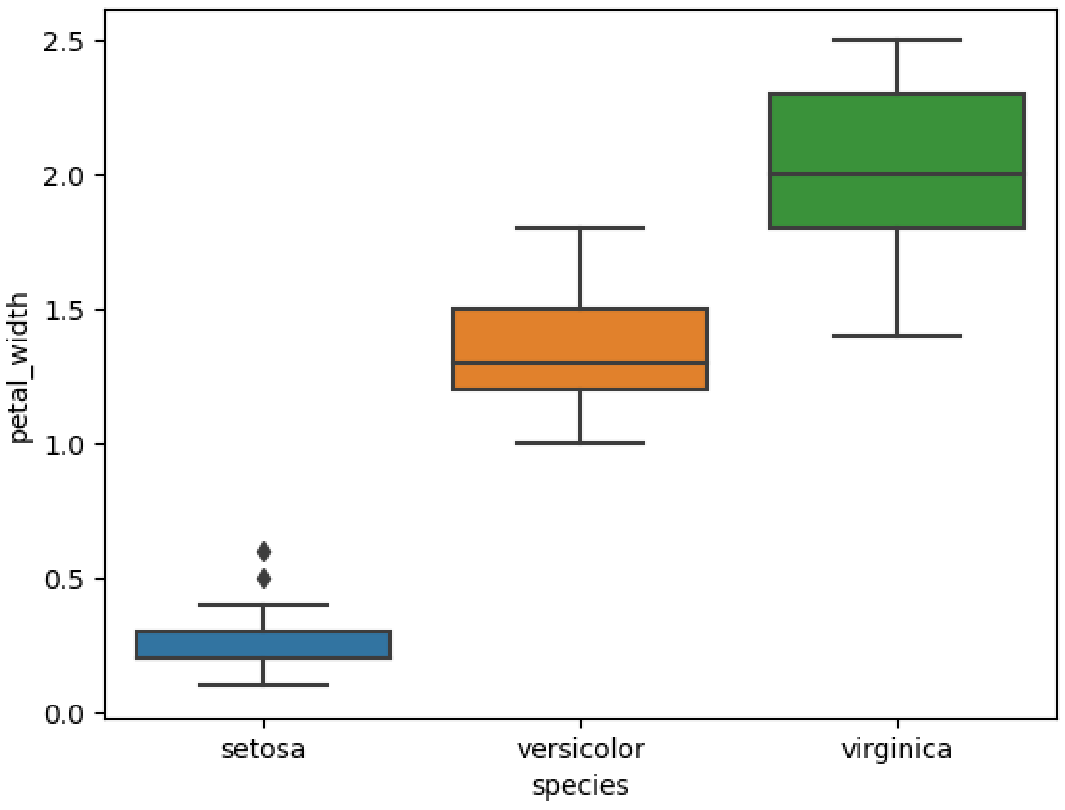
b) The boxplots are as follows:





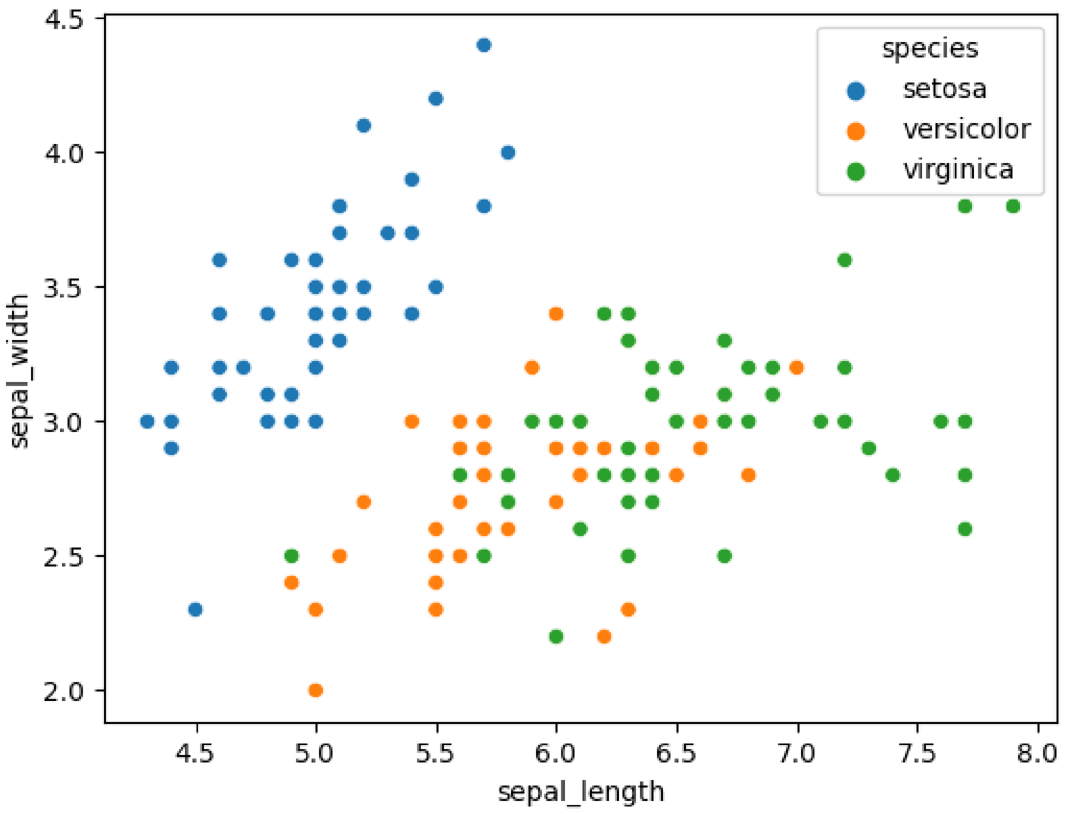


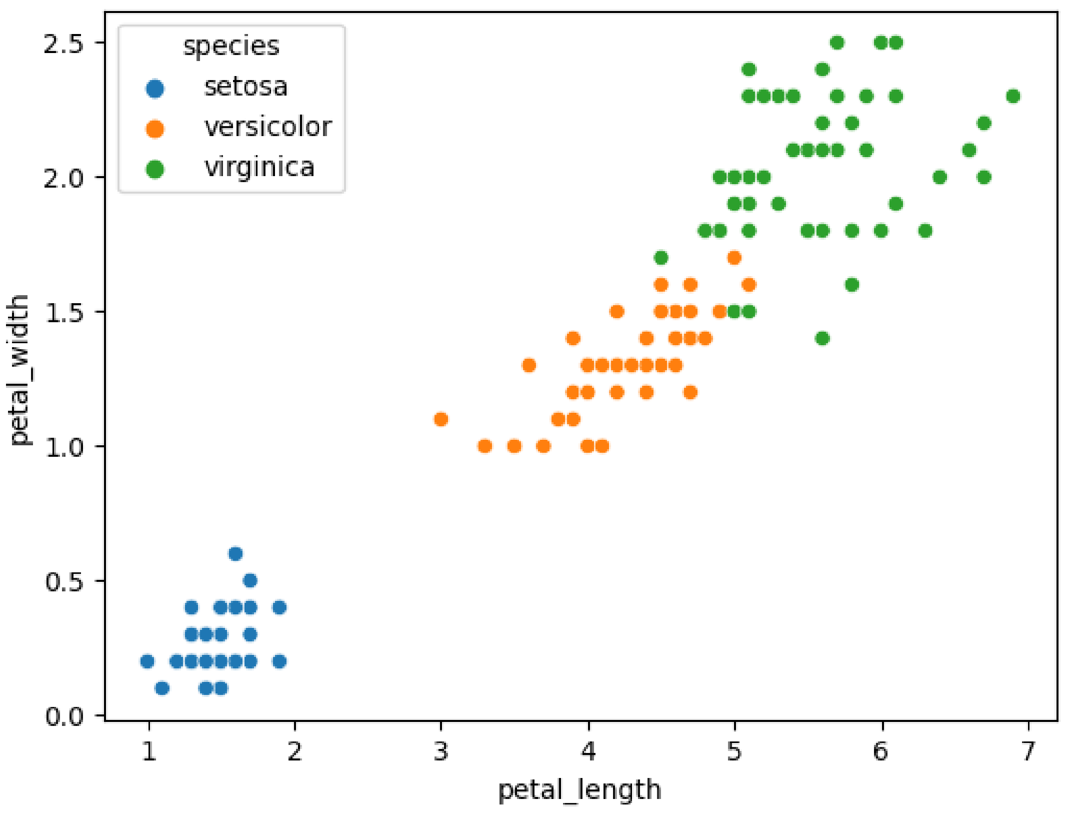
7
6
5
4
3
2
1
petal_length
setosa
versicolor
virginica
species



2.5
2.0
1.5
1.0
0.5
0.0
petal_width
setosa
versicolor
virginica
species

c) The scatterplots are as follows:





d) Using a combination of all 6 graphs that were created during this problem, we can create a set of criteria to classify each species of Iris that was observed. One of the clearest groupings is found in the last graph, petal length vs. petal width. Here, it is extremely easy to identify setosa vs the other species. It exclusively has a petal length and width that is significantly smaller than any other species, therefore, if the petal length is below 2.5 and the width is below 0.75, we can be extremely confident that it is setosa. If we for some reason need to be even more confident, we can look at the sepal length and sepal width scatterplot. If we draw a line from (4, 2.25) to (6.25, 4.5), anything that is to the left of that line is a setosa from the dataset we have. Therefore, if both of these conditions are met we can be confident that we can classify it as a setosa species of iris.

What about the other species, versicolor and virginica? These are a little more difficult to suss out, so we will use all the graphs to help answer this question. We can see in the petal length vs. petal width scatterplot that there are clearly two groups, though the differences aren't as distinct as setosa. Versicolor tends to have a shorter overall length and width than virginica, though they both overlap around the 5 length and 1.6 width mark. So if the length and width aren't both short like versicolor, or both long like virginica, how can we tell the difference? We need to use the sepal length and width graphs to figure it out.

The scatterplot is even more muddled, so using the boxplots are more useful here. Outside of outliers, we see that virginica for the majority of the time has a larger sepal length than versicolor. This is also true for sepal width, though it isn't at the same majority level nor does it have the same difference in size as the sepal length. Combining all of these statistics at the same time is our best bet in making a classification difference between versicolor and virginica then. If the petal length and the petal width are in between the two groups, then if the sepal length and widths are larger, we can classify with confidence that it's virginica, and vice versa for versicolor.

Problem 3: K-NN Implementation

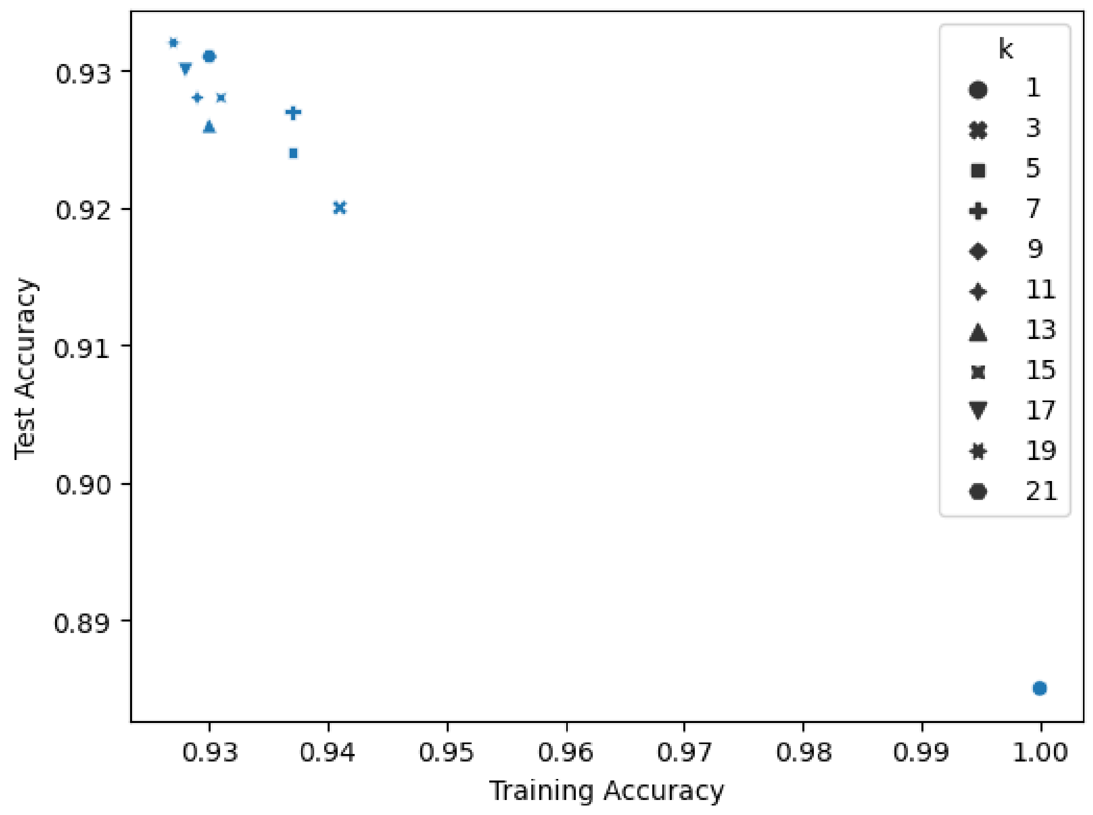
a) See knn.py

b) See knn.py

c) See knn.py

d) For a k=1, I found the training accuracy to be 100% and the test accuracy to be 85.5%.

e) For figuring out the training accuracy as a function of k, I chose to run my K-NN algorithm in a for loop, where each iteration was equal to k. I skipped all even numbers since you would never purposely choose an even k because that only introduces ties and therefore reduces accuracy. My K-NN algorithm is expensive, so I only chose to test values of k ranging from 1 to 21 so it would not run for an extreme length of time.



Ideally we want, for some value of k, a datapoint that is in the upper right corner of the scatterplot. This would mean that we achieved 100% accuracy for both the training dataset and the test dataset. We also prefer a higher test dataset accuracy over a higher training accuracy since we already know the values of the training dataset, but still don't want it to be too low.

For our given training and test data, a k of 19 is shown to have the best balance of accuracy. It has the highest test accuracy at over 93% with a near 93% accuracy for the training data too. Both of these statistics are solid and I would therefore choose a k of 19 if I were to get more test data.

f) There is a lot that goes into the predict method to make it scale for any given dataset, which influences how expensive a method predict is. First is the input, which is xFeat, which is a mxd matrix. We also have the training size to contend with, which is a nxd matrix. I have a for loop that iterates for each tuple of xFeat, so it runs m times. Within that, I have another for loop that runs for features $d-2$, as the training set always has an index and label feature which is not needed for the immediate prediction steps. After that loop (but still in the first loop), I have a loop that iterates through the entirety of the training set, so n number of times. Here, the distance is calculated for each and every feature. This requires an additional loop which runs a total of $d-2$ times (same explanation as before).

After this nested loop we have a loop that runs for length k to get the nearest neighbors from the points found previously. Lastly, I have a loop that runs for the number labels the data has. All of our data set's only had 2 labels, but this cannot be assumed universally so I will call the number of labels l.

In all, that gives us a runtime of $O(m((d-2)+n*(d-2)+k+l))$. We can simplify this to be $O(m*(n*d+k+l))$.

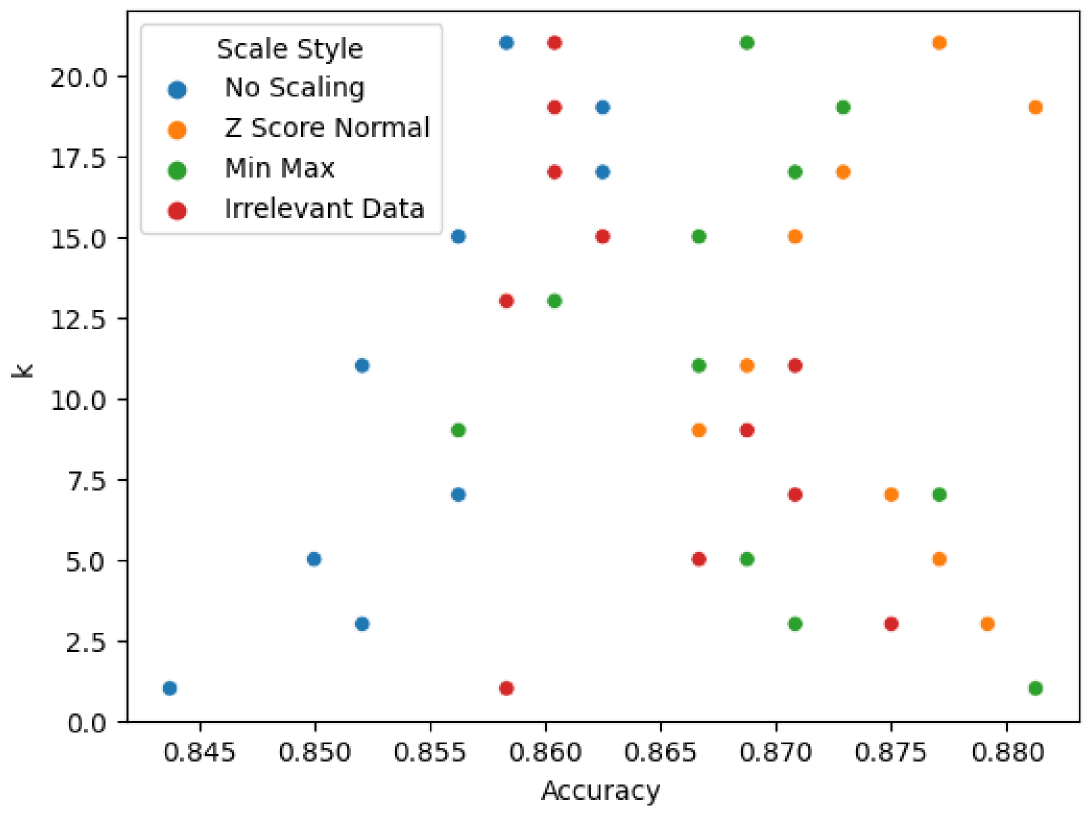
Problem 4: K-NN Performance

a) See q4.py

b) See q4.py

c) See q4.py

d) Just like in problem 3, I chose to have my k go from 1 to 21. Any longer would have significantly increased the runtime which is already long at roughly 15 minutes. For the scatterplot, the x-axis is accuracy and the y-axis is the k used. The hue defines the scale style and is labeled on the plot itself.



The k and scale have a massive effect on the accuracy of the K-NN algorithm as seen in the scatter plot, falling as low as just above 84.5% accuracy (which isn't very good). The largest variance occurred when using irrelevant data as the accuracy was generally poor and only mildly better as k sat between 3 and 11. This makes sense since the data is made up: it isn't a reflection of the actual content of the dataset and the K-NN isn't able to easily ignore it, even at high levels of k.

No scaling at all, surprisingly, does the worst. It improves for higher levels of k (to a point) as expected, but it is surprising that it's even worse than adding irrelevant data.

Min-max scaling looks much better, though it has extreme highs and lows. k=1 had surprisingly (and concerningly) the greatest accuracy, while any k=9 had the worst accuracy, even worse than any given k with irrelevant data. The accuracy at higher k's such as 15, 19, and 21 all had much better results with accuracy around 87% to 87.5%.

Lastly, setting our data to a normal curve, funnily enough, has our results mimicking that of a normal curve. At both high and low k's we achieve the best accuracy seen out of these scaling options, topping the charts at a near 90% accuracy value.

Medium-sized k's bring our accuracy down, though nothing like the lows of min-max or irrelevant data.

The biggest conclusion that I have gathered from this plot is that while scale matters to a significant degree, the absolute biggest factor on performance is the value of k that you choose. Z Score and Min-max scaling can be helpful and while we saw z score outperform min-max here, I would want to test both out if I were to use a different training dataset in the future. Irrelevant data, while having literally no meaning to the dataset, also *improved* performance. Ideally, I would want to include as many features as possible, scaled, to get the best of both worlds. This would help improve the accuracy of the K-NN model for any value of k, which is a powerful addition.