Honor Code ¶

THIS CODE IS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING CODE WRITTEN BY OTHER STUDENTS - Eric Gu

1.

Time [sec] (for loop): 4.1875

Time [sec] (np loop): 0.03125

np Loop is much faster, I have added a main print statement that will show the difference, this one is specifically

4.15625 seconds faster

2.

For a set of rules in order to figure out classification of species type, it seems that

For Setosas: sepal width seems to be within 2.8-4.5, while sepal length is 4-5.8, and petal width & length are low with width being lower than .5 generally, and petal length being lower than 2 but greater than 1

If the range of each parameter is within these boundaries, it should be classified as a setosa

We can apply the logic above for each other species, so I will not got into the specific measurements for the other species. The method for the other species is similar, with all the classifications being within the range of measurements of the specific species, and to match a species based off of that.

If the species in question does not fit within any specific classification type, we can use an algorithm to fit it with the ranges that are most similar to those of the other species, to provide a guess

This should be sufficient!

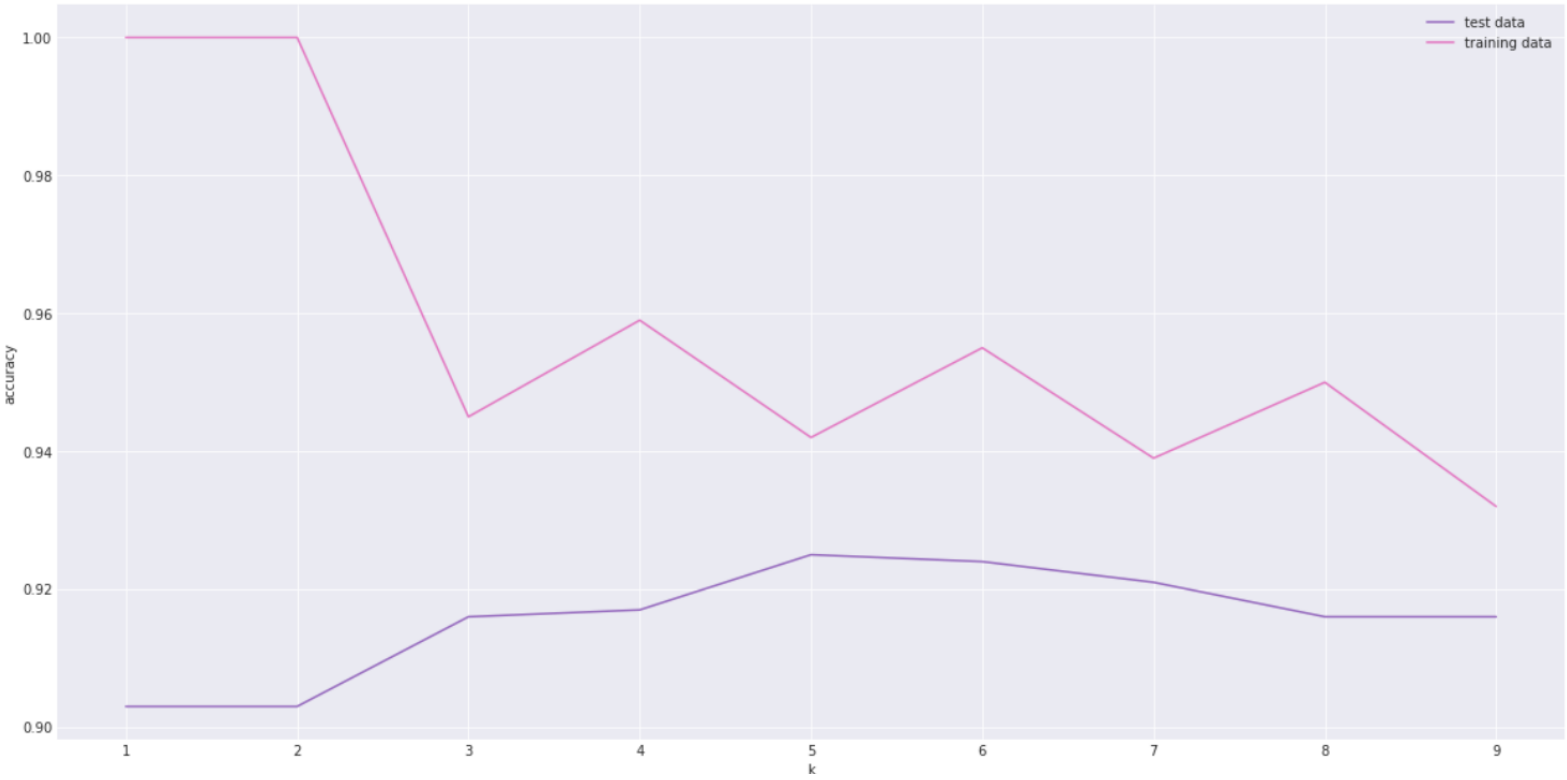
3.

The predict function implemented includes a method call of another helper method, in terms of training size (n), the number of features (d), and the number of neighbors (k):

```
def predict(self, xFeat):
    myHat = [self.iterative_predict(i) for i in xFeat.to_numpy()]
    return myHat

def iterative_predict(self, x):
    differences = np.array([euclidean_distance(x, x_train) for x_train in self.matrix.to_numpy()])
    samples = np.argsort(differences)
    nearest_labels = [self.array[i] for i in samples[:self.k]]
    max_common = Counter(nearest_labels).most_common(1)
    return max_common[0][0]
```

We can see that for the training size n, we have to iterate through the entire training size, and compare it to every other row in the training size. This means the runtime of training size with its comparisons to the features is at least O(n * d), which is what it takes for distances. Then, in order to calculate for the least distance, for every n we must do it k times, meaning that we have runtime of O(n * k). Thus, the total runtime will be O(kn + dn), so O((k + d)(n)). If k and d are both constants, then we can change it to O(n).



4.

Here is a graph of k from 1 to 51, with the different types of processing:


Unprocessed
Standard
Min Max
Noise
Accuracy
0.89
0.88
0.87
0.86
0.85
0.84
0.83
0
6
12
18
24
30
36
42
48
k

For the different types of pre-processing:

After a few nodes, the accuracy of specific processing is pretty similar, but we notice drastic changes about the k= 8-10 mark where we can see clearer winners/losers.

Standardization:

Outperforms clearly in accuracy after about k = 8, but starts to lose out about after k = 33 to min-max range. The best kind of accurate pre-processing for the k-values in general i.e. 1 to 20, but min-max seems better for later k values. Sometimes gets beaten by the others in the beginning, but it is generally rare that it is in last place, except near the end of after k = 33. Seems standardization is the best for early/mid values, before 33.

Min-Max range:

With lesser k values, especially in the beginning, min-max range is very inaccurate (for the first few k it is the least accurate). However, after about k = 33, it becomes the most accurate. After k = 20 it is in second place in accuracy, with standardization being the best. Therefore, min-max seems the best for very high k-values.

Irrelevant Features:

Performs a lot like unprocessed, and then the same at about 40. Either way, it is clear that with the added noise, it rarely helped the accuracy. Whenever it did help the accuracy, it is just out of unlikely probability. Probably not good to use in general. This performs similar, and sometimes worse than unprocessed, so we should prefer the other 2 that do much better.

The sensitivity to data:

The sensivity of irrelevant features in the behavior seems similar to unprocessed overall from k = 1 to 40, but seems generally less accurate in the beginning, and continues to be that way, oscillating here and there and going a bit above unprocessed, until it merges with unprocessed. After about 40, it the sensitivity to data seems less and less sensitive with the merge with unprocessed after about 40ish. Either way, it is sensitive enough that irrelevant features change the accuracy by quite a few percentage in multiple cases, so it is sensitive enough that we don't want this noise. However, it is not as sensitive to the point of completely throwing all accuracy off.