**KNN from scratch**

Go to S://Public/CS/KNN for this document.

Take a look at the toy example.

*from sklearn.neighbors import KNeighborsClassifier*

*import numpy as np*

*X\_train = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])*

*y\_train = np.array([1,1,1,0,0,0])*

*neighbor = KNeighborsClassifier(n\_neighbors=3)*

*neighbor.fit(X\_train, y\_train)*

*X\_test = np.array([[1, 0], [-2, -2]])*

*print(neighbor.predict(X\_test))*

Output:j

[0,1]

X\_train is a 2D array, or matrix. In this example, we have 6 samples, each of which consist of 2 features.

y\_train is an array of target class corresponding to X\_train.

You might be wondering why X is in uppercase and y is in lowercase. In Linear Algebra, we usually denote matrices by uppercase letters and vectors by lowercase letters.

Then you call the predict method with a test set(X\_test). The output is a list of predicted classes.

Calculating k-NN by Hand

Let's calculate k-NN by hand using the above example. In my experience, hand calculation really helps to understand the algorithm. Unlike most algorithms, k-NN does nothing at fit. All the work happens at predict. The first test sample is [1,0]. To measure the similarity, we simply calculate the difference for each feature and add them up. This is called Manhattan distance. The Manhattan distance between vector p and q can be written as follows:

d(p,q)=∑i=1n|pi−qi|

The first training sample is [-1, -1]. So the Manhattan distance is

|−1−1|+|−1−0|=2+1=3

Notice the absolute value notation. If you don't take the absolute value, positive value and negative value cancel out.

Similarly, the Manhattan distances of the rest of the training data are 4, 6, 1, 2, 4, respectively.

K = 3 in this example, so we pick the 3 nearest neighbors.

The smallest value means the nearest, so the nearest neighbor is [1,1] with distance = 1.

Its corresponding class is 0. The second and third nearest neighbors are [2,1], [-1,-1], whose classes are 0, 1.

We have two 0's and one 1 so the prediction is 0.

**Part 2: A Simple k-NN**

The Initial Code

*class KNeighborsClassifier(object):*

*def \_\_init\_\_(self, n\_neighbors=5):*

*self.n\_neighbors = n\_neighbors*

*def fit(self, X, y):*

*self.X = X*

*self.y = y*

*return self*

*def \_predict\_one(self, test):*

*return 1*

*def predict(self, X):*

*return [self.\_predict\_one(i) for i in X]*

For now, \_\_init\_\_ takes one parameter, n\_neighbors. We just save it to the instance variable. As I said before, fit doesn't do anything in k-NN. Again, we just save X and y to the instance variables for later use. predict in scikit-learn predicts not just 1 instance, but multiple instances. So we have \_predict\_one for predicting 1 instance. predict calls \_predict\_one for each instance and puts them in a list. By the way, I am using the term data, sample, and instance loosely. They pretty much mean the same thing here. For now, \_predict\_one always returns 1, but let's test the code.

*import numpy as np*

*X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])*

*y = np.array([1,1,1,0,0,0])*

*neighbor = KNeighborsClassifier()*

*neighbor.fit(X, y)*

*print(neighbor.predict(np.array([[1, 0], [-2, -2]])))*

Output: [1,1]

It is quite common to initialize the class and fit in 1 line.

*neighbor = KNeighborsClassifier().fit(X, y)*

Now let's work on \_predict\_one.

**Manhattan Distance**

**Exercise 1**

The first thing you have to do is calculate distance. The method \_distance takes two numpy arrays data1, data2, and returns the Manhattan distance between the two. This shouldn't be that hard, so I want you to write it by yourself. If you have to highlight space below “here is my solution”/

Hint: This can be done in 1 line of code:

*def \_distance(self, data1, data2):*

*"""returns Manhattan distance"""*

*# Your code here*

*neighbor = KNeighborsClassifier()*

*print(neighbor.\_distance(np.array([-1, -1]),np.array([1, -2])))*

Your code should print 3.

Solution

Here is my solution:

def \_distance(self, data1, data2):

return sum(abs(data1 - data2))

Getting K nearest neighbors

Now that you have distance, we can get K nearest neighbors. To do that, first we naively sort the samples. If you have 1000 examples and just want get, say, 3 smallest values, it is unnecessary to sort them all. But our goal is to understand how k-NN works, not write efficient code:

*def \_predict\_one(self, test):*

*distances = sorted((self.\_distance(x, test), y) for x, y in zip(self.X, self.y))*

*return distances*

It's important to sort a pair of distance and a target class. Otherwise you won't be able to find a corresponding class value. When sorting tuples or lists instead of numbers, sorted function sorts by the first element. The first element of our tuple is distance, so we are good. I'm breaking \_predict\_one method temporarily; it no longer returns a number. But that's OK. Let's test it:

*X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])*

*y = np.array([1,1,1,0,0,0])*

*neighbor = KNeighborsClassifier().fit(X, y)*

*print(neighbor.\_predict\_one(np.array([1, 0])))*

Output:

[(1, 0), (2, 0), (3, 1), (4, 0), (4, 1), (6, 1)]

It works! It matches what we calculated by hand earlier.

As a side note, when applying a function to a list/dictionary comprehension, you don't need square brackets or curly braces. Using them would make it ugly and hard to read:

distances = sorted([(self.\_distance(x, test), y) for x, y in zip(self.X, self.y)])

Once we sorted the distances, we can take the first k elements:

*def \_predict\_one(self, test):*

*distances = sorted((self.\_distance(x, test), y) for x, y in zip(self.X, self.y))*

*neighbors = distances[:self.n\_neighbors]*

*return neighbors*

**Uniform Weights**

**Exercise 2**

After we take the k nearest neighbors, we no longer care about their distance. What we care is how many of them belong to each class. So \_compute\_weights method changes distances to 1. This is fairly straightforward. Again, this can be done is 1 line of code:

*def \_compute\_weights(self, distances):*

*"""computes uniform weights"""*

*# Your code here*

*neighbor = KNeighborsClassifier()*

*print(neighbor.\_compute\_weights(np.array([(1, 0), (2, 0), (3, 1)])))*

Output: [(1, 0), (1, 0), (1, 1)]

Solution

Here is my solution:

def \_compute\_weights(self, distances):

return [(1, y) for d, y in distances]

Now we call the method with neighbors as the argument:

*def \_predict\_one(self, test):*

*distances = sorted((self.\_distance(x, test), y) for x, y in zip(self.X, self.y))*

*neighbors = distances[:self.n\_neighbors]*

*weights = self.\_compute\_weights(neighbors)*

*return weights*

You can shorten the code a little bit as long as it's readable:

*weights = self.\_compute\_weights(distances[:self.n\_neighbors])*

Predicting 1 Instance

At this point, our k-NN class looks like this:

*import numpy as np*

*class KNeighborsClassifier(object):*

*def \_\_init\_\_(self, n\_neighbors=5):*

*self.n\_neighbors = n\_neighbors*

*def fit(self, X, y):*

*self.X = X*

*self.y = y*

*return self*

*def \_distance(self, data1, data2):*

*return sum(abs(data1 - data2))*

*def \_compute\_weights(self, distances):*

*return [(1, y) for d, y in distances]*

*def \_predict\_one(self, test):*

*distances = sorted((self.\_distance(x, test), y) for x, y in zip(self.X, self.y))*

*weights = self.\_compute\_weights(distances[:self.n\_neighbors])*

*return weights*

*def predict(self, X):*

*return [self.\_predict\_one(i) for i in X]*

We need to group the weights by class so that we can count them. To do that, we can use defaultdict:

*from collections import defaultdict*

*def \_predict\_one(self, test):*

*distances = sorted((self.\_distance(x, test), y) for x, y in zip(self.X, self.y))*

*weights = self.\_compute\_weights(distances[:self.n\_neighbors])*

*weights\_by\_class = defaultdict(list)*

*for d, c in weights:*

*weights\_by\_class[c].append(d)*

*return weights\_by\_class*

*X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])*

*y = np.array([1,1,1,0,0,0])*

*neighbor = KNeighborsClassifier().fit(X, y)*

*print(neighbor.\_predict\_one(np.array([1, 0])))*

Output:

defaultdict(<class 'list'>, {0: [1, 1, 1], 1: [1, 1]})

**Exercise 3**

We are almost done. Given the weights\_by\_class, we can predict the class. As always, this can be done in 1 line. The example we've been using is binary classification, but your code should also work with multiclass classification:

*def \_predict\_one(self, test):*

*distances = sorted((self.\_distance(x, test), y) for x, y in zip(self.X, self.y))*

*weights = self.\_compute\_weights(distances[:self.n\_neighbors])*

*weights\_by\_class = defaultdict(list)*

*for d, c in weights:*

*weights\_by\_class[c].append(d)*

*# Your code here*

*X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])*

*y = np.array([1,1,1,0,0,0])*

*neighbor = KNeighborsClassifier().fit(X, y)*

*print(neighbor.\_predict\_one(np.array([1, 0])))*

Output: This should print 0

Solution

Here is mine:

*return max((sum(val), key) for key, val in weights\_by\_class.items())[1]*

There is quite a lot going on in this line. Some people might prefer to break it down to multiple lines, which is totally fine:

*counts = [(sum(val), key) for key, val in weights\_by\_class.items()]*

*majority = max(counts)*

*return majority[1]*

First we count how many neighbors belong to each class using sum, then take the majority with max. Just like sorted, when dealing with tuples, max looks at the first element by default. Finally, you take the second element of the tuple, which is the predicted class.

That's it!

*import numpy as np*

*from collections import defaultdict*

*class KNeighborsClassifier(object):*

*def \_\_init\_\_(self, n\_neighbors=5):*

*self.n\_neighbors = n\_neighbors*

*def fit(self, X, y):*

*self.X = X*

*self.y = y*

*return self*

*def \_distance(self, data1, data2):*

*return sum(abs(data1 - data2))*

*def \_compute\_weights(self, distances):*

*return [(1, y) for d, y in distances]*

*def \_predict\_one(self, test):*

*distances = sorted((self.\_distance(x, test), y) for x, y in zip(self.X, self.y))*

*weights = self.\_compute\_weights(distances[:self.n\_neighbors])*

*weights\_by\_class = defaultdict(list)*

*for d, c in weights:*

*weights\_by\_class[c].append(d)*

*return max((sum(val), key) for key, val in weights\_by\_class.items())[1]*

*def predict(self, X):*

*return [self.\_predict\_one(i) for i in X]*

Let's test it.

*X\_train = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])*

*y\_train = np.array([1,1,1,0,0,0])*

*neighbor = KNeighborsClassifier(n\_neighbors=3)*

*neighbor.fit(X\_train, y\_train)*

*X\_test = np.array([[1, 0], [-2, -2]])*

*print(neighbor.predict(X\_test))*

The output looks as you expect: [0,1]

**Part 3: Some Improvements**

If you look at the scikit-learn documentation, you see that KNeighborsClassifier takes a lot of parameters and has many more methods.

Distance-Weighted k-NN

The way we pick a winner from k nearest neighbors is simply take the majority. But what happens in this case?

*X = np.array([[1, 1], [4, 4], [5, 5]])*

*y = np.array([1,0,0])*

*neighbor = KNeighborsClassifier(n\_neighbors=3).fit(X, y)*

*print(neighbor.\_predict\_one(np.array([0, 0])))*

According to our code, it will produce 0 because we have 2 neighbors of class 0 whereas we have only 1 neighbor of class 1. But those two neighbors of class 0 are further away. Intuitively, 1 is more likely to be correct in this case. So we would like to take the distances into account, giving closer points more weight. To do that, we take the inverse of a distance(1/d). There is also 1/(d2), which implies that further points matter even less than just the inverse, which makes the choice of K matters less. I don't know which one is better, but scikit-learn chooses the former, so I will stick to that.

First we need to add a new parameter, weights. The default is uniform.

*def \_\_init\_\_(self, n\_neighbors=5, weights='uniform'):*

*self.n\_neighbors = n\_neighbors*

*self.weights = weights*

**Exercise 4**

Let's make some change to \_compute\_weights. I only left the main part for the exercise.

*def \_compute\_weights(self, distances):*

*if self.weights == 'uniform':*

*return [(1, y) for d, y in distances]*

*elif self.weights == 'distance':*

*# Your code here*

*raise ValueError("weights not recognized: should be 'uniform' or 'distance'")*

*neighbor = KNeighborsClassifier()*

*print(neighbor.\_compute\_weights(np.array([(1, 0), (2, 0), (3, 1)])))*

Output:

[(1.0, 0), (0.5, 0), (0.3333333333333333, 1)]

Solution

This should be fairly straightforward:

*return [(1/d, y) for d, y in distances]*

Now this is classified as 1 instead of 0:

*X = np.array([[1, 1], [4, 4], [5, 5]])*

*y = np.array([1,0,0])*

*neighbor = KNeighborsClassifier(n\_neighbors=3, weights='distance').fit(X, y)*

*print(neighbor.\_predict\_one(np.array([0, 0])))*

There is one problem though. What happens if some distance is 0? You will get division by 0 error. Distance = 0 means two samples have exactly the same value on every feature. This might happen sometimes, especially if the sample size is big. Or, if you fit and predict with the same X to measure the training set accuracy, you will definitely get the error on every single instance(If you don't know why, pause and think about it).

So when at least one distance is 0, we will forget about weighted distance and assign 1 to distance = 0 and 0 to everything else. In other words, we compute uniform weights on those samples whose distance is 0.

**Exercise 5**

Let's implement the improved version of \_compute\_weights:

*def \_compute\_weights(self, distances):*

*if self.weights == 'uniform':*

*return [(1, y) for d, y in distances]*

*elif self.weights == 'distance':*

*# Your code here*

*raise ValueError("weights not recognized: should be 'uniform' or 'distance'")*

*neighbor = KNeighborsClassifier(weights='distance')*

*print(neighbor.\_compute\_weights([(0, 1),(0, 1),(3, 0),(0, 0)]))*

Output:

[(1, 1), (1, 1), (1, 0)]

Solution

Here is mine:

*matches = [(1, y) for d, y in distances if d == 0]*

return matches if matches else [(1/d, y) for d, y in distances]

First it collects all the samples whose distance is 0 and assigns 1. If the matches contains something, it returns it. Otherwise, it just returns what we had previously.

**Euclidean Distance**

We used Manhattan distance to calculate the distance between the two points. Another popular distance is Euclidean Distance. To calculate it, we square the difference for each feature, add them up, and take the square root of it.

Let's add Euclidean distance to our code.

The first thing we need to do is add a new parameter p to \_\_init\_\_:

*def \_\_init\_\_(self, n\_neighbors=5, p=2):*

*self.n\_neighbors = n\_neighbors*

*self.weights = weights*

*self.p = p*

When p = 1, Manhattan distance is used, and when p = 2, Euclidean distance. The default is 2. You might think why we use numbers instead of something like 'manhattan' and 'euclidean' as we did on weights. The reason for this is that Manhattan distance and Euclidean distance are the special case of Minkowski distance. For arbitrary p, Minkowski distance is used in scikit-learn

**Exercise 6**

Just like Manhattan distance, if you follow the formula, you will be fine. Hint: sqrt in math module does not work on numpy array. You can use numpy's own sqrt instead.

*def \_distance(self, data1, data2):*

*"""1: Manhattan, 2: Euclidean"""*

*if self.p == 1:*

*return sum(abs(data1 - data2))*

*elif self.p == 2:*

*# Your code here*

*raise ValueError("p not recognized: should be 1 or 2")*

*neighbor = KNeighborsClassifier(p=2)*

*print(neighbor.\_distance(np.array([-1, -1]),np.array([1, -2])))*

The output should be 2.2360679775, which is

Solution

*return np.sqrt(sum((data1 - data2)\*\*2))*

**Score method**

**Exercise 7**

We have one more method to write which is score. It's just the mean accuracy of the given test data. It calls predict, compares the output to y, and returns what fraction of them got right. Hint: You don't need list comprehension for this:

*def score(self, X, y):*

*# Your code here*

*X\_train = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])*

*y\_train = np.array([1,1,1,0,0,0])*

*neighbor = KNeighborsClassifier().fit(X\_train, y\_train)*

*X\_test = np.array([[1, 0], [-2, -2]])*

*y\_test = np.array([0, 0])*

*print(neighbor.score(X\_test, y\_test))*

The output should be 0.5

Solution

*return sum(self.predict(X) == y) / len(y)*

Now we have the improved version of k-NN.

*import numpy as np*

*from collections import defaultdict*

*class KNeighborsClassifier(object):*

*def \_\_init\_\_(self, n\_neighbors=5, weights='uniform', p=2):*

*self.n\_neighbors = n\_neighbors*

*self.weights = weights*

*self.p = p*

*def fit(self, X, y):*

*self.X = X*

*self.y = y*

*return self*

*def \_distance(self, data1, data2):*

*"""1: Manhattan, 2: Euclidean"""*

*if self.p == 1:*

*return sum(abs(data1 - data2))*

*elif self.p == 2:*

*return np.sqrt(sum((data1 - data2)\*\*2))*

*raise ValueError("p not recognized: should be 1 or 2")*

*def \_compute\_weights(self, distances):*

*if self.weights == 'uniform':*

*return [(1, y) for d, y in distances]*

*elif self.weights == 'distance':*

*matches = [(1, y) for d, y in distances if d == 0]*

*return matches if matches else [(1/d, y) for d, y in distances]*

*raise ValueError("weights not recognized: should be 'uniform' or 'distance'")*

*def \_predict\_one(self, test):*

*distances = sorted((self.\_distance(x, test), y) for x, y in zip(self.X, self.y))*

*weights = self.\_compute\_weights(distances[:self.n\_neighbors])*

*weights\_by\_class = defaultdict(list)*

*for d, c in weights:*

*weights\_by\_class[c].append(d)*

*return max((sum(val), key) for key, val in weights\_by\_class.items())[1]*

*def predict(self, X):*

*return [self.\_predict\_one(x) for x in X]*

*def score(self, X, y):*

*return sum(1 for p, t in zip(self.predict(X), y) if p == t) / len(y)*

**Back to the Iris Flower dataset**

Let's try our k-NN on a more realistic example. You can import a handful of classic datasets directly from scikit-learn's datasets module.

*from sklearn import datasets*

*from sklearn.cross\_validation import train\_test\_split*

*iris = datasets.load\_iris()*

*X\_train, X\_temp, y\_train, y\_temp = \*

*train\_test\_split(iris.data, iris.target, test\_size=.3)*

*X\_validation, X\_test, y\_validation, y\_test = \*

*train\_test\_split(X\_temp, y\_temp, test\_size=.5)*

*neighbor = KNeighborsClassifier().fit(X\_train, y\_train)*

*print(neighbor.score(X\_train, y\_train))*

*print(neighbor.score(X\_validation, y\_validation))*

Using train\_test\_split function from cross\_validation module, it first splits the data in the ratio 70:30, then splits the latter in half. Now we have 70% for the training set, 15% for the validation and test set.

Next it initializes k-NN with no special configuration and fits it. Finally, it prints the training set accuracy and the validation set accuracy to see if the model is overfitting. Since the validation set contains only 30 samples, the result varies a lot. Sometimes I got 100% and sometimes 90%. The most common one seems to be .967, meaning it got only one sample wrong. I've tried the various combinations of the 3 parameters, but the default setting is hard to beat in this dataset.

Once you find the good parameters, you can get the true score on the test set.

*print(neighbor.score(X\_test, y\_test))*

I encourage you to play with the code and see how changing each parameter affects the accuracy.