k-nearest neighbors and kd-trees

CS 360 Machine Learning

Week 2, Day 1

January 30, 2024

Contents

1 Python style and implementation notes

Hopefully at this point you're fairly familiar with good coding style, but here are a few quick reminders and tips:

- It's important to remember to decompose code into natural functions, i.e., to use logical breaks based on the represented ideas.
- When thinking about variable names, it'll be useful to include variable types in the name itself to help you keep track.
- Recall that Python uses snake_case not CamelCase to represent variables.
- Be sure not to import based on, e.g., `from numpy import *` and instead use `import numpy as np` because you can end up with name conflicts and confusion!

Each file should have a block header at the top that includes name, course, date, and a description of hte overall file. Each function should also has docstrings and appropriate comments. Each block of code and inline comments should have comments as needed; a good rule of thumb is that if you would need the comment in order to remember and understand what's going on in this part of the code, you should include it. Especially now that you're a 300-level student, you should be sure to take care with your code and use good style!

The main function is the driver of the whole program, and it relies on helper functions throughout the code. One way to create this type of structure in your programs is to use top-down-design (TDD). In these cases, you design the main function first pretending that you already have the other functions you'll need available. While doing this, you'll think about the details that are abstracted below into smaller tasks, and name them, and these will become your helper functions. Ideally, you should also stub out those function– this means they should return the correct type so your code runs, but doesn't do the correct task yet.

Then, you do the implementation starting at the lowest level–these won't depend on anything, so you can implement and even test them without the rest of the code. This is called "bottom up" implementation. Now is the time to practice these good software engineering practices so you don't duplicate unneeded code or effort!

2 Overfitting

You may have seen this in CS 260 before, but it's important as we move to more advanced machine learning techniques. See textbook Figure 1-23 for an example. You'll see that it has data points and is attempting to fit a model (high degree polynomial) to the given data. But there's too much complexity in the model relative to the phenomena you're trying to describe. The textbook defines *overfitting* as what "happens when the model is too complex relative to the amount and noisiness of the training data." It's useful to think of this as relative to the underlying phenomena you're trying to describe. Another way to think about overfitting is that you're essentially memorizing the data and not learning to generalize. Usually the solution is to reduce the complexity, but you can also get more training data or attempt to reduce the noise in the training data. In this figure, what would have helped us is if we had more training data in the larger x value ranges.

Underfitting is the opposite of overfitting—you had the opportunity to learn something but didn't. In this case, the capabilities of the model were not enough to learn the phenomena.

One example from CS 260 (and see textbook Figure 1-23 or slides) was where there was a pattern from the data that wasn't being captured with a line (underfitting) and when the model was changed to use a polynomial of degree 10 too much was captured (overfitting), while attempts to model the data using a polynomial of degree 3 worked well. We determined this was the best degree for the model polynomial in CS 260 based on an elbow plot, but we could also see this by evaluating the data on the test set. Both underfit and overfit models will have high error on the *test* data.

One common pattern observed in ML is that as model complexity goes from low to high, the error on the *training* data goes from high to low–it's really just memorizing the data, but the error on the *test* data starts high, goes low, and then goes high again as the model is overfitting. Obviously, we want to end up with low test set error! This is a central idea in all of machine learning.

You can decompose the error into error that comes from noise in the data versus the model. Suppose you had two observations with the same features but one has one label and the other has a different label – that's an inherent limitation of the data. We'll try to separate the data error versus the model error later in this class.

Is it wrong to use the test data to determine the model complexity? Yes – remember our edict: **never touch the test data**! So how do we solve this? We create a *validation* data set – a third data set in addition to the training and test sets. All these *hyperparameters* – specific parameters that control the model such as the polynomial's degree in the example above – should be chosen based on the validation dataset.

3 k-nearest neighbors (kNN)

Why would we be interested in finding a point's nearest neighbor in a set of points? We can cast any classification problem we want to solve as a nearest neighbor problem – we can just borrow the label from the closest training data point into the test data. We might also want to find a nearest neighbor because we can use it for filling in missing data – we could find the nearest neighbor in other dimensions and fill in the feature value from that other dimension. It can also be used as a subroutine for some clustering methods.

Let me do a quick example with kNN for binary classification in two dimensions (i.e. with two features). Suppose we have some points that are positive examples and some that are negative examples. Suppose that we are given a new test point with two known features and we want to know what its resulting label is. We can plot that point and see what its nearest neighbor is – we might find that the new point is near to a negative example and predict that its label is negative. Suppose that we had a test point with an unknown label near to the decision boundary? We could average a few more examples (the k in kNN) and use the average (or a voting scheme) to determine the label.

This is often called *naïve kNN* because it's looking at each other point to determine what training data point is closes to the test data. This algorithm takes in training points and a query point and returns the label:

```
knn(points, q):
   for p in points:
       if d(p, q) < min}
          min = d(p, q)
          nn = p
```

I may want to keep track of both the minimum distance and the neighbor associated with that. This is the general pseudocode for the naïve algorithm – you just go through all the points and consider their distance to your query point.

What's the runtime of this algorithm? Suppose that we have n training points, then the runtime is $O(n)$. The distance calculation may not be trivial depending on the number of dimensions we have (d) but ignoring the number of dimensions for now, we can think of this as linear. (Including the distance calculation and including the number of dimensions, this is $O(nd)$.) Suppose that I have m test points and n and m are of a similar order, then we end up with something close to quadratic quickly. We'd like something faster, especially when handling large datasets. (So far, this is just for the nearest neighbor algorithm for $k = 1$. What about higher values of k? We'll think about how to extend this pseudocode to higher k values in the lab.)

k-nearest neighbors (kNN) creates implicit decision boundaries; separations of regions of the feature space based on the class label and the chosen value of k (see Figure 1). These aren't formal functions, but are useful for thinking about what's going on with the data space. One formalized way of thinking about these is as Voronoi diagrams, and the decision boundary and Voronoi diagrams depend on the distance metric chosen. The same point may have different nearest neighbors under different distance metrics chosen. The distance metric we think about commonly is the Euclidean distance, but distance walking in a city has a metric known as Manhattan distance, and there are other distance metric options as well. In our lab we'll use Euclidean distance.

Formally, suppose that we have points a and b in d dimensions where $a = (a_1, a_2, ...a_d)$ and $b = (b_1, b_2, ...b_d)$, then Euclidean distance is defined as:

$$d_E(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_d - b_d)^2}$$

and the Manhattan distance is defined as:

$$d_M(a, b) = |a_1 - b + 1| + |a_2 - b_2| + ... + |a_d - b_d|$$

Considering the example in Figure 1 with the Euclidean distance, you can think about each (very small light grid) dot of Figure 1 as a hypothetical test set which we're coloring based on the label it would have received. Looking at the left of that figure, you can see that there's a point all on its own with the region around it colored blue, so we can tell that $k = 1$ in this example, and if you only look at the nearest neighbor ($k = 1$) you'll often find that you're overfitting to the noise by using only that closest neighbor.

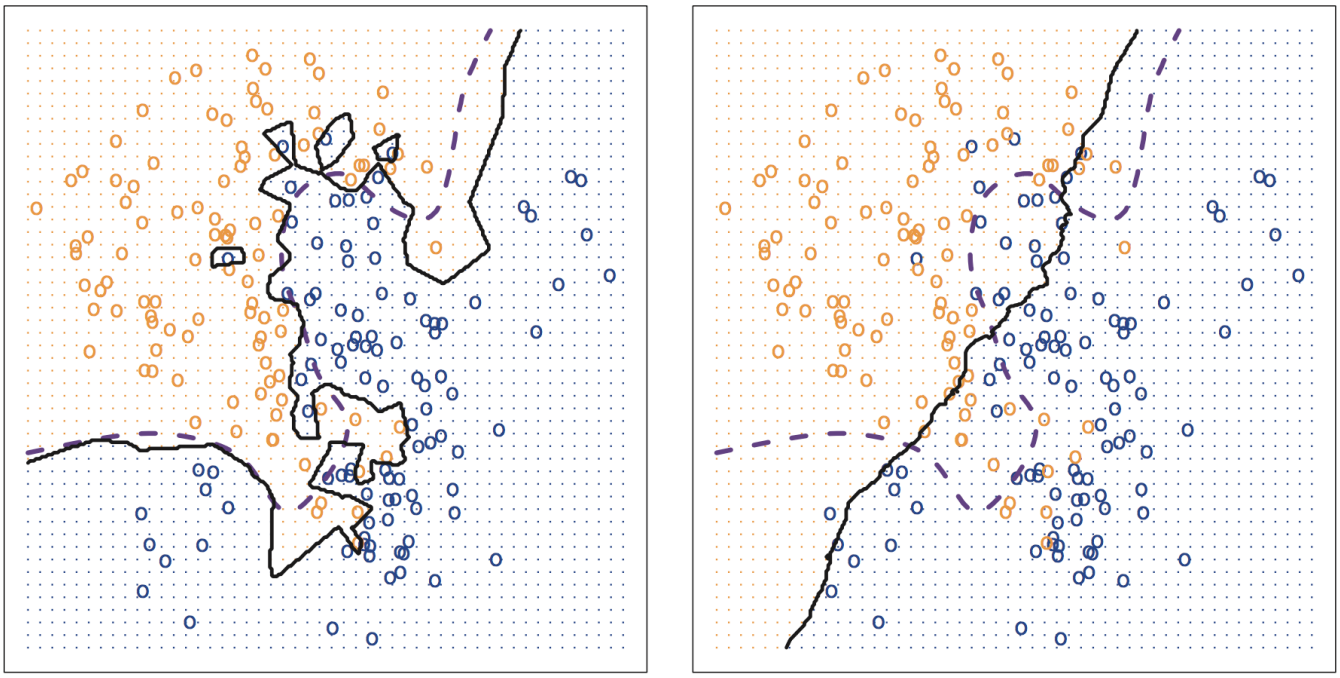

Figure 1: k-nearest neighbors example with decision boundaries shown in orange and blue, and points in the training set shown as orange or blue circles. Possible decision boundaries, based on different choices of k, are shown as dashed or solid lines.

The right is underfitting. At the very extremes you could even choose $k = n$, and then you'd just be returning the majority class. On the right we actually chose $k = 100$. It's not looking closely enough at the training data, but rather returning broad trends like averages in the data.

We can also see that this is very inefficient – why look at the points far in the lower right of the space, if you're considering a query point in the upper left? The intuition behind this next algorithm is that we can partition the space like we would a tree.

3.1 *kd*-trees

In order to do something sublinear for repeated queries, we might recall binary search trees and other trees can achieve this. In this case, we'll use *kd*-trees. We'll use the example in Figure 2 as a running example. Points A through G are the training data. Without seeing the test data at all, we can build a tree structure that will be useful in labeling the test data. Again, here we just have two dimensions ($d = 2$). We can set up a recursive algorithm to create the tree. This algorithm will take the points and the depth of the tree (initialized to 0) - we'll use the `mod` operator to cycle through the dimensions. It doesn't actually matter what dimension you start with, but by convention we'll start with $d = 0$ or the x-axis. For two dimensions, the *dim* will alternate between 0 and 1. In the below, *depth* will always be increasing and represents the depth in the constructed tree.

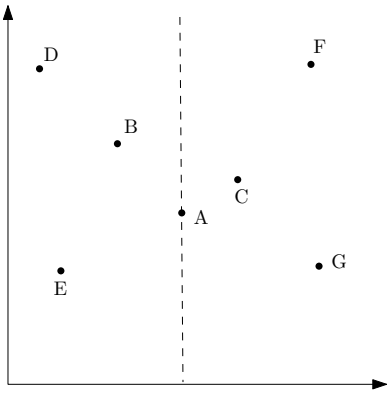

Figure 2: A running example set of points for the *kd*-tree.

```
kdtree (points, depth):
     dim = depth \% d
     sort points and take the median
     make Node object at median
     if even, more on the left
     node.point = median
     node.left = kdtree(points on the left of the median, depth + 1)
     node.right = kdtree(points on the right, depth +1)
     return ?
```

The above shows *incomplete* pseudocode for creating a *kd*-tree. Note that the above will also need a base case and a return value.

Imagine that we flatten (project) all the points onto the x-axis, then the order along that axis is: $DEBACFG$. This is the order of the points sorted based on $dim = 0$. Now I want to find a point to divide based on to create the tree. Suppose that I took the whole space and just divided it in two based on the space - I might end up with more points on one side of the tree than the other. But if we use median (as opposed to the middle based on the coordinates) then we can end up with a more balanced tree. In this case, we'll divide the space based on the median point in dimension 0, so A becomes the root of the tree (see Figure 3).

If there's an even number of points, we'll arbitrarily choose to have more on the left because we'd like there to be a node for each training points; this means that the left child will be filled before your right child, which is a useful convention. When we recursively split, we will recurse on the points on the left of the median (or the right). In future dimensions (e.g., the y-axis), these will actually become up and down, but we'll stick with the terms left and right. Note that for our recursive call, we need to be sure to return something as well.

Continuing our example, after splitting so that A is the root node based on taking the median along an ordering of the points along the x-axis (formally a projection on the x-axis), we now go to the next dimension ($dim = 1$). Looking at that order, we find that in increasing order based on y, we have the points EBD on the left of the original dividing line. Our median here is B, so B becomes the left child of A. Continuing E becomes the left child of B and D becomes its right child.

On the other side of the recursive call from A, we order the points in y-axis order GCF and find that the right child of A is C. Continuing, G becomes the left child of C and F becomes its right child. The final tree and corresponding space partitioning is shown in Figure 4.

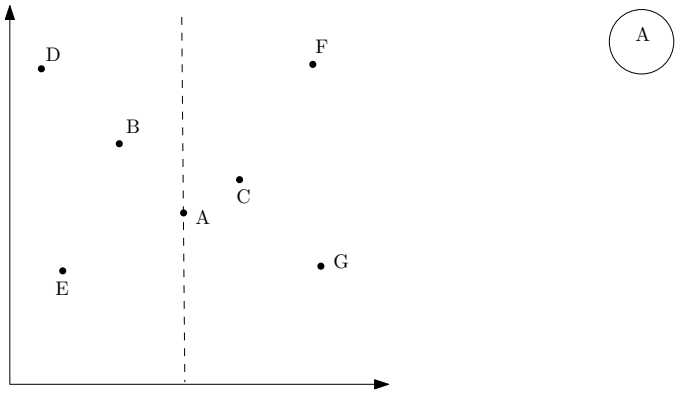

Figure 3: A running example set of points for the *kd*-tree, with the first split line creating root node A.

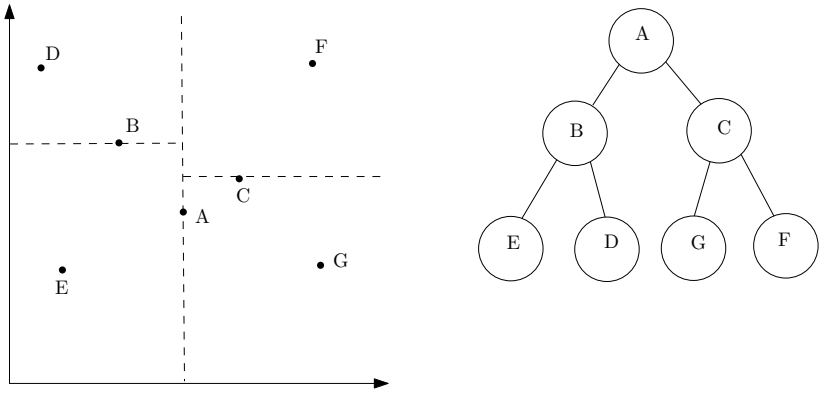

Figure 4: A running example set of points for the *kd*-tree, with the full tree and space partitioning shown.

So now that we've built this tree, how do we use it? Suppose that my query point is close to F. How could I determine what its nearest neighbor was programmatically? First I compare to A and see whether its x-value is more or less than A's x-value. It's greater, so I go to the right. Then I compare the query point's y-value to C's y-value and see if it's greater or less, since it's greater I go to F. Our query point is close to F, so we can return F.

However this algorithm is incorrect! There are a number of special cases we need to consider. What if the query point is on the dividing line? What if the query point is on the other side of a boundary but still closer to a point on the other side? At each point in our traversal of the tree, we actually need to calculate and compare the distance of the query point to the distance to the training point at a node. And in some cases, you'll need to travel down the other side of the recursive call as well because based on the distance comparison there's a chance the relevant closest point, based on the current minimum distance, is on the other side of the dividing boundary. So we always compare the current minimum distance to the distance between the query point and the dividing boundary – we can think about this as a circle with a radius of the current minimum distance centered at the query point intersecting with the dividing boundary.

As practice, try making a *kd*-tree on the below training points (Figure 5), starting as is convention with the x-axis.

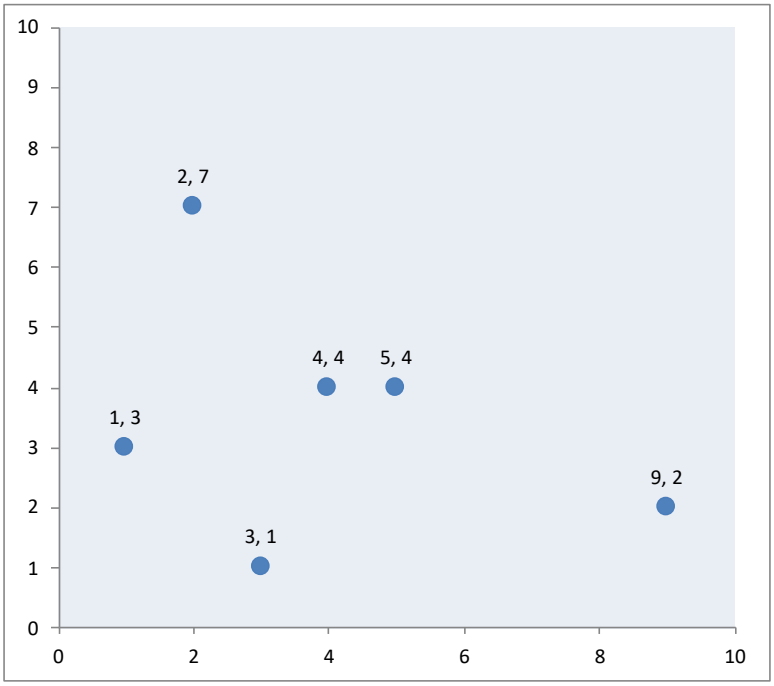

Figure 5: *kd*-tree example initial training points