3 | Geometry and Nearest Neighbors

> Our brains have evolved to get us out of the rain, find where the berries are, and keep us from getting killed. Our brains did not evolve to help us grasp really large numbers or to look at things in a hundred thousand dimensions. – Ronald Graham

Learning Objectives:

- Describe a data set as points in a high dimensional space.
- Explain the curse of dimensionality.
- Compute distances between points in high dimensional space.
- Implement a K-nearest neighbor model of learning.
- Draw decision boundaries.
- Implement the K-means algorithm for clustering.

Dependencies: Chapter 1

You can think of prediction tasks as mapping inputs (course reviews) to outputs (course ratings). As you learned in the previous chapter, decomposing an input into a collection of features (e.g., words that occur in the review) forms a useful abstraction for learning. Therefore, inputs are nothing more than lists of feature values. This suggests a **geometric view** of data, where we have one dimension for every feature. In this view, examples are points in a high-dimensional space.

Once we think of a data set as a collection of points in high dimensional space, we can start performing geometric operations on this data. For instance, suppose you need to predict whether Alice will like Algorithms. Perhaps we can try to find another student who is most "similar" to Alice, in terms of favorite courses. Say this student is Jeremy. If Jeremy liked Algorithms, then we might guess that Alice will as well. This is an example of a **nearest neighbor** model of learning. By inspecting this model, we'll see a completely different set of answers to the key learning questions we discovered in Chapter 1.

3.1 *From Data to Feature Vectors*

An example is just a collection of feature values about that example, for instance the data in Table 1 from the Appendix. To a person, these features have meaning. One feature might count how many times the reviewer wrote "excellent" in a course review. Another might count the number of exclamation points. A third might tell us if any text is underlined in the review.

To a machine, the **features** themselves have no meaning. Only the **feature values**, and how they vary across examples, mean something to the machine. From this perspective, you can think about an example as being represented by a **feature vector** consisting of one "dimension" for each feature, where each dimenion is simply some real value.

Consider a review that said "excellent" three times, had one excla-

mation point and no underlined text. This could be represented by the feature vector $\langle 3, 1, 0 \rangle$. An almost identical review that happened to have underlined text would have the feature vector $\langle 3, 1, 1 \rangle$.

Note, here, that we have imposed the convention that for **binary features** (yes/no features), the corresponding feature values are 0 and 1, respectively. This was an arbitrary choice. We could have made them 0.92 and -16.1 if we wanted. But 0/1 is convenient and helps us interpret the feature values. When we discuss practical issues in Chapter 5, you will see other reasons why 0/1 is a good choice.

Figure 3.1 shows the data from Table 1 in three views. These three views are constructed by considering two features at a time in different pairs. In all cases, the plusses denote positive examples and the minuses denote negative examples. In some cases, the points fall on top of each other, which is why you cannot see 20 unique points in all figures.

The mapping from feature values to vectors is straighforward in the case of real valued features (trivial) and binary features (mapped to zero or one). It is less clear what to do with **categorical features**. For example, if our goal is to identify whether an object in an image is a tomato, blueberry, cucumber or cockroach, we might want to know its color: is it RED, BLUE, GREEN or BLACK?

One option would be to map RED to a value of 0, BLUE to a value of 1, GREEN to a value of 2 and BLACK to a value of 3. The problem with this mapping is that it turns an unordered set (the set of colors) into an ordered set (the set $\{0, 1, 2, 3\}$). In itself, this is not necessarily a bad thing. But when we go to *use* these features, we will measure examples based on their distances to each other. By doing this mapping, we are essentially saying that RED and BLUE are more similar (distance of 1) than RED and BLACK (distance of 3). This is probably not what we want to say!

A solution is to turn a categorical feature that can take four different values (say: RED, BLUE, GREEN and BLACK) into four binary features (say: IsItRed?, IsItBlue?, IsItGreen? and IsItBlack?). In general, if we start from a categorical feature that takes V values, we can map it to V-many binary indicator features.

With that, you should be able to take a data set and map each example to a feature vector through the following mapping:

- Real-valued features get copied directly.
- Binary features become 0 (for false) or 1 (for true).
- Categorical features with V possible values get mapped to V-many binary indicator features.

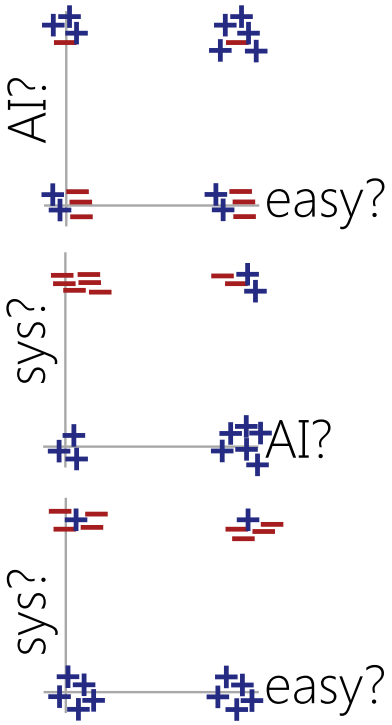


Figure 3.1: A figure showing projections of data in two dimension in three ways – see text. Top: horizontal axis corresponds to the first feature (easy) and the vertical axis corresponds to the second feature (AI?); Middle: horizontal is second feature and vertical is third (systems?); Bottom: horizontal is first and vertical is third. Truly, the data points would like exactly on $(0, 0)$ or $(1, 0)$, etc., but they have been purturbed slightly to show duplicates.

? Match the example ids from Table 1 with the points in Figure 3.1.

? The computer scientist in you might be saying: actually we could map it to $\log_2 V$-many binary features! Is this a good idea or not?

After this mapping, you can think of a single example as a **vector** in a high-dimensional **feature space**. If you have D-many features (after expanding categorical features), then this **feature vector** will have D-many components. We will denote feature vectors as $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_D \rangle$, so that x_d denotes the value of the dth feature of $\boldsymbol{x}$. Since these are vectors with real-valued components in D-dimensions, we say that they belong to the space $\mathbb{R}^D$.

For $D = 2$, our feature vectors are just points in the plane, like in Figure 3.1. For $D = 3$ this is three dimensional space. For $D > 3$ it becomes quite hard to visualize. (You should resist the temptation to think of $D = 4$ as "time" – this will just make things confusing.) Unfortunately, for the sorts of problems you will encounter in machine learning, $D \approx 20$ is considered "low dimensional," $D \approx 1000$ is "medium dimensional" and $D \approx 100000$ is "high dimensional."

? Can you think of problems (perhaps ones already mentioned in this book!) that are low dimensional? That are medium dimensional? That are high dimensional?

3.2 K-Nearest Neighbors

The biggest advantage to thinking of examples as vectors in a high dimensional space is that it allows us to apply geometric concepts to machine learning. For instance, one of the most basic things that one can do in a vector space is compute **distances**. In two-dimensional space, the distance between $\langle 2, 3 \rangle$ and $\langle 6, 1 \rangle$ is given by $\sqrt{(2-6)^2 + (3-1)^2} = \sqrt{18} \approx 4.24$. In general, in D-dimensional space, the **Euclidean distance** between vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is given by Eq (3.1) (see Figure 3.2 for geometric intuition in three dimensions):

$$d(\boldsymbol{a}, \boldsymbol{b}) = \left[\sum_{d=1}^{D} (a_d - b_d)^2 \right]^{\frac{1}{2}} \tag{3.1}$$

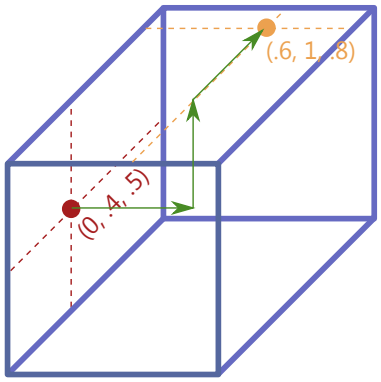


Figure 3.2: A figure showing Euclidean distance in three dimensions. The length of the green segments are 0.6, 0.6 and 0.3 respectively, in the x-, y-, and z-axes. The total distance between the red dot and the orange dot is therefore $\sqrt{0.6^2 + 0.6^2 + 0.3^2} = 0.9$.

? Verify that d from Eq (3.1) gives the same result (4.24) for the previous computation.

Now that you have access to distances between examples, you can start thinking about what it means to learn again. Consider Figure 3.3. We have a collection of training data consisting of positive examples and negative examples. There is a test point marked by a question mark. Your job is to guess the correct label for that point.

Most likely, you decided that the label of this test point is positive. One reason why you might have thought that is that you believe that the label for an example should be similar to the label of nearby points. This is an example of a new form of **inductive bias**.

The **nearest neighbor** classifier is build upon this insight. In comparison to decision trees, the algorithm is ridiculously simple. At training time, we simply store the entire training set. At test time, we get a test example $\hat{\boldsymbol{x}}$. To predict its label, we find the training example $\boldsymbol{x}$ that is most similar to $\hat{\boldsymbol{x}}$. In particular, we find the training

example $\boldsymbol{x}$ that *minimizes* $d(\boldsymbol{x}, \hat{\boldsymbol{x}})$. Since $\boldsymbol{x}$ is a training example, it has a corresponding label, y. We predict that the label of $\hat{\boldsymbol{x}}$ is also y.

Despite its simplicity, this nearest neighbor classifier is incredibly effective. (Some might say *frustratingly* effective.) However, it is particularly prone to overfitting label noise. Consider the data in Figure 3.4. You would probably want to label the test point positive. Unfortunately, it's nearest neighbor happens to be negative. Since the nearest neighbor algorithm only looks at the *single* nearest neighbor, it cannot consider the "preponderance of evidence" that this point should probably actually be a positive example. It will make an unnecessary error.

A solution to this problem is to consider more than just the single nearest neighbor when making a classification decision. We can consider the **K-nearest neighbors** and let them **vote** on the correct class for this test point. If you consider the 3-nearest neighbors of the test point in Figure 3.4, you will see that two of them are positive and one is negative. Through voting, positive would win.

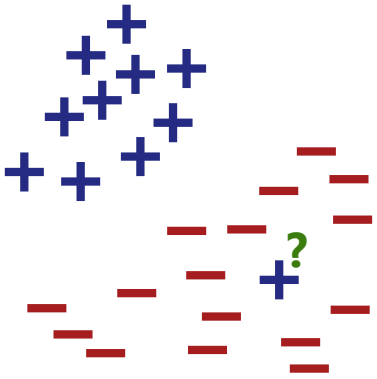


Figure 3.4: A figure showing an easy NN classification problem where the test point is a ? and should be positive, but its NN is actually a negative point that's noisy.

? Why is it a good idea to use an odd number for K?

The full algorithm for K-nearest neighbor classification is given in Algorithm 3.2. Note that there actually is no "training" phase for K-nearest neighbors. In this algorithm we have introduced five new conventions:

1. The training data is denoted by $\mathbf{D}$.
2. We assume that there are N-many training examples.
3. These examples are pairs $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)$. (Warning: do not confuse $\boldsymbol{x}_n$, the nth training example, with x_d, the dth feature for example $\boldsymbol{x}$.)
4. We use [ ]to denote an empty list and $\oplus \cdot$ to append $\cdot$ to that list.
5. Our prediction on $\hat{\boldsymbol{x}}$ is called $\hat{y}$.

The first step in this algorithm is to compute distances from the test point to all training points (lines 2-4). The data points are then sorted according to distance. We then apply a clever trick of *summing* the class labels for each of the K nearest neighbors (lines 6-10) and using the SIGN of this sum as our prediction.

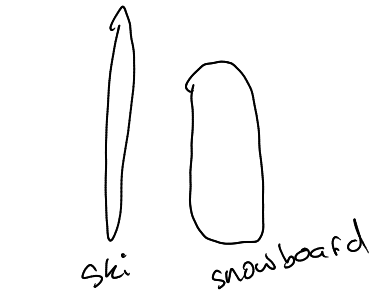
? Why is the sign of the sum computed in lines 2-4 the same as the majority vote of the associated training examples?

The big question, of course, is how to choose K. As we've seen, with $K = 1$, we run the risk of overfitting. On the other hand, if K is large (for instance, $K = N$), then KNN-PREDICT will always predict the majority class. Clearly that is underfitting. So, K is a hyperparameter of the KNN algorithm that allows us to trade-off between overfitting (small value of K) and underfitting (large value of K).

? Why can't you simply pick the value of K that does best on the training data? In other words, why do we have to treat it like a hyperparameter rather than just a parameter.

Algorithm 3 KNN-PREDICT($\mathbf{D}$, K, $\hat{x}$)

```
S ← [ ]
for n = 1 to N do
    S ← S ⊕ ⟨d(x_n, x̂), n⟩          // store distance to training example n
end for
S ← SORT(S)                          // put lowest-distance objects first
ŷ ← 0
for k = 1 to K do
    ⟨dist,n⟩ ← S_k                   // n this is the kth closest data point
    ŷ ← ŷ + y_n                      // vote according to the label for the nth training point
end for
return SIGN(ŷ)                       // return +1 if ŷ > 0 and −1 if ŷ < 0
```



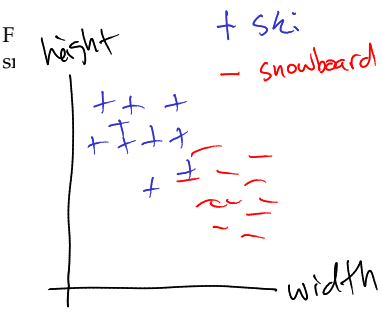


Figure 3.6: Classification data for ski vs snowboard in 2d

One aspect of **inductive bias** that we've seen for KNN is that it assumes that nearby points should have the same label. Another aspect, which is quite different from decision trees, is that all features are equally important! Recall that for decision trees, the key question was *which features are most useful for classification?* The whole learning algorithm for a decision tree hinged on finding a small set of good features. This is all thrown away in KNN classifiers: every feature is used, and they are all used the same amount. This means that if you have data with only a few relevant features and lots of irrelevant features, KNN is likely to do poorly.

A related issue with KNN is **feature scale**. Suppose that we are trying to classify whether some object is a ski or a snowboard (see Figure 3.5). We are given two features about this data: the width and height. As is standard in skiing, width is measured in millimeters and height is measured in centimeters. Since there are only two features, we can actually plot the entire training set; see Figure 3.6 where ski is the positive class. Based on this data, you might guess that a KNN classifier would do well.

Suppose, however, that our measurement of the width was computed in millimeters (instead of centimeters). This yields the data shown in Figure 3.7. Since the width values are now tiny, in comparison to the height values, a KNN classifier will effectively *ignore* the width values and classify almost purely based on height. The predicted class for the displayed test point had changed because of this feature scaling.

We will discuss feature scaling more in Chapter 5. For now, it is just important to keep in mind that KNN does not have the power to decide which features are important.

Figure 3.7: Classification data for ski vs snowboard in 2d, with width rescaled to mm.

MATH REVIEW | VECTOR ARITHMETIC AND VECTOR NORMS

A (real-valued) **vector** is just an array of real values, for instance $\boldsymbol{x} = \langle 1, 2.5, -6 \rangle$ is a three-dimensional vector. In general, if $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_D \rangle$, then x_d is it's dth component. So $x_3 = -6$ in the previous example.

Vector sums are computed pointwise, and are only defined when dimensions match, so $\langle 1, 2.5, -6 \rangle + \langle 2, -2.5, 3 \rangle = \langle 3, 0, -3 \rangle$. In general, if $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$ then $c_d = a_d + b_d$ for all d. Vector addition can be viewed geometrically as taking a vector $\boldsymbol{a}$, then tacking on $\boldsymbol{b}$ to the end of it; the new end point is exactly $\boldsymbol{c}$.

Vectors can be **scaled** by real values; for instance $2\langle 1, 2.5, -6 \rangle = \langle 2, 5, -12 \rangle$; this is called scalar multiplication. In general, $a\boldsymbol{x} = \langle ax_1, ax_2, \ldots, ax_D \rangle$.

The **norm** of a vector $\boldsymbol{x}$, written $||\boldsymbol{x}||$ is its length. Unless otherwise specified, this is its *Euclidean* length, namely: $||\boldsymbol{x}|| = \sqrt{\sum_d x_d^2}$.

3.3 Decision Boundaries

The standard way that we've been thinking about learning algorithms up to now is in the *query model*. Based on training data, you learn something. I then give you a query example and you have to guess it's label.

An alternative, less passive, way to think about a learned model is to ask: what sort of test examples will it classify as positive, and what sort will it classify as negative. In Figure 3.9, we have a set of training data. The background of the image is colored blue in regions that *would* be classified as positive (if a query were issued there) and colored red in regions that *would* be classified as negative. This coloring is based on a 1-nearest neighbor classifier.

In Figure 3.9, there is a solid line separating the positive regions from the negative regions. This line is called the **decision boundary** for this classifier. It is the line with positive land on one side and negative land on the other side.

Decision boundaries are useful ways to visualize the **complexity** of a learned model. Intuitively, a learned model with a decision boundary that is really jagged (like the coastline of Norway) is really complex and prone to overfitting. A learned model with a decision boundary that is really simple (like the bounary between Arizona and Utah) is potentially underfit.

Now that you know about decision boundaries, it is natural to ask: what do decision boundaries for decision trees look like? In order

Figure 3.8:

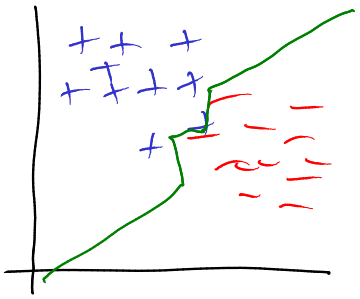


Figure 3.9: decision boundary for 1nn.

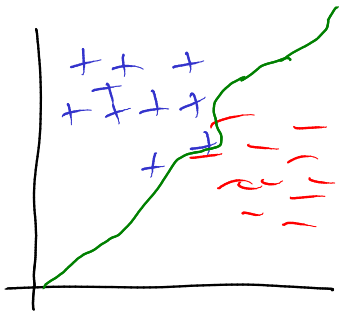


Figure 3.10: decision boundary for knn with k=3.

to answer this question, we have to be a bit more formal about how to build a decision tree on real-valued features. (Remember that the algorithm you learned in the previous chapter implicitly assumed *binary* feature values.) The idea is to allow the decision tree to ask questions of the form: "is the value of feature 5 greater than 0.2?" That is, for real-valued features, the decision tree nodes are parameterized by a feature and a threshold for that feature. An example decision tree for classifying skis versus snowboards is shown in Figure 3.11.

Now that a decision tree can handle feature vectors, we can talk about decision boundaries. By example, the decision boundary for the decision tree in Figure 3.11 is shown in Figure 3.12. In the figure, space is first split in half according to the first query along one axis. Then, depending on which half of the space you look at, it is either split again along the other axis, or simply classified.

Figure 3.12 is a good visualization of decision boundaries for decision trees in general. Their decision boundaries are axis-aligned cuts. The cuts must be axis-aligned because nodes can only query on a single feature at a time. In this case, since the decision tree was so shallow, the decision boundary was relatively simple.

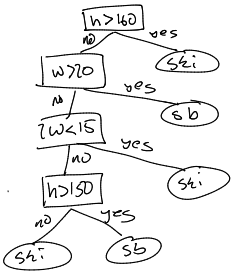


Figure 3.11: decision tree for ski vs. snowboard

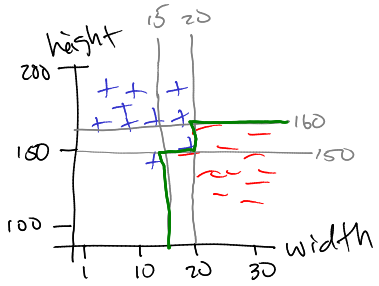


Figure 3.12: decision boundary for dt in previous figure

? What sort of data might yield a very simple decision boundary with a decision tree and very complex decision boundary with 1-nearest neighbor? What about the other way around?

3.4 K-Means Clustering

Up through this point, you have learned all about supervised learning (in particular, binary classification). As another example of the use of geometric intuitions and data, we are going to temporarily consider an **unsupervised learning** problem. In unsupervised learning, our data consists *only* of examples $\boldsymbol{x}_n$ and does *not* contain corresponding labels. Your job is to make sense of this data, even though no one has provided you with correct labels. The particular notion of "making sense of" that we will talk about now is the **clustering** task.

Consider the data shown in Figure 3.13. Since this is unsupervised learning and we do not have access to labels, the data points are simply drawn as black dots. Your job is to split this data set into three clusters. That is, you should label each data point as A, B or C in whatever way you want.

For this data set, it's pretty clear what you should do. You probably labeled the upper-left set of points A, the upper-right set of points B and the bottom set of points C. Or perhaps you permuted these labels. But chances are your clusters were the same as mine.

The *K*-means clustering algorithm is a particularly simple and effective approach to producing clusters on data like you see in Figure 3.13. The idea is to represent each cluster by it's cluster center. Given cluster centers, we can simply assign each point to its nearest

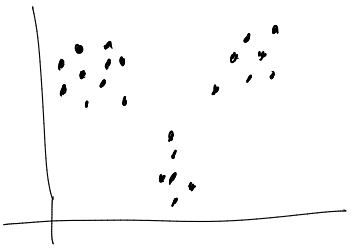


Figure 3.13: simple clustering data... clusters in UL, UR and BC.

center. Similarly, if we know the assignment of points to clusters, we can compute the centers. This introduces a chicken-and-egg problem. If we knew the clusters, we could compute the centers. If we knew the centers, we could compute the clusters. But we don't know either.

The general computer science answer to chicken-and-egg problems is **iteration**. We will start with a guess of the cluster centers. Based on that guess, we will assign each data point to its closest center. Given these new assignments, we can recompute the cluster centers. We repeat this process until clusters stop moving. The first few iterations of the K-means algorithm are shown in Figure 3.14. In this example, the clusters converge very quickly.

Algorithm 3.4 spells out the K-means clustering algorithm in detail. The cluster centers are initialized randomly. In line 6, data point $\boldsymbol{x}_n$ is compared against each cluster center $\boldsymbol{\mu}_k$. It is assigned to cluster k if k is the center with the smallest distance. (That is the "argmin" step.) The variable z_n stores the assignment (a value from 1 to K) of example n. In lines 8-12, the cluster centers are re-computed. First, $\mathbf{X}_k$ stores all examples that have been assigned to cluster k. The center of cluster k, $\boldsymbol{\mu}_k$ is then computed as the mean of the points assigned to it. This process repeats until the centers converge.

An obvious question about this algorithm is: does it converge? A second question is: how long does it take to converge. The first question is actually easy to answer. Yes, it does. And in practice, it usually converges quite quickly (usually fewer than 20 iterations). In Chapter 15, we will actually *prove* that it converges. The question of how long it takes to converge is actually a really interesting question. Even though the K-means algorithm dates back to the mid 1950s, the best known convergence rates were *terrible* for a long time. Here, terrible means exponential in the number of data points. This was a sad situation because empirically we knew that it converged very quickly. New algorithm analysis techniques called "smoothed analysis" were invented in 2001 and have been used to show very fast convergence for K-means (among other algorithms). These techniques are well beyond the scope of this book (and this author!) but suffice it to say that K-means is fast in practice and is provably fast in theory.

It is important to note that although K-means is guaranteed to converge and guaranteed to converge quickly, it is *not* guaranteed to converge to the "right answer." The key problem with unsupervised learning is that we have no way of knowing what the "right answer" is. Convergence to a bad solution is usually due to poor initialization.

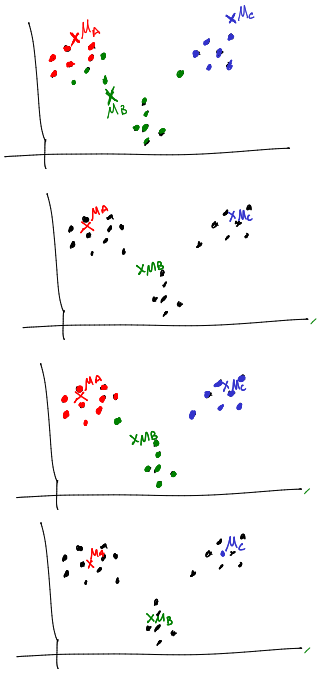


Figure 3.14: first few iterations of k-means running on previous data set

? What is the difference between unsupervised and supervised learning that means that we know what the "right answer" is for supervised learning but not for unsupervised learning?

Algorithm 4 K-MEANS($\mathbf{D}$, K)

```
for k = 1 to K do
    μ_k ← some random location          // randomly initialize center for kth cluster
end for
repeat
    for n = 1 to N do
        z_n ← argmin_k ||μ_k − x_n||    // assign example n to closest center
    end for
    for k = 1 to K do
        X_k ← { x_n : z_n = k }         // points assigned to cluster k
        μ_k ← MEAN(X_k)                 // re-estimate center of cluster k
    end for
until μs stop changing
return z                                // return cluster assignments
```

3.5 *Warning: High Dimensions are Scary*

Visualizing one hundred dimensional space is incredibly difficult for humans. After huge amounts of training, some people have reported that they can visualize four dimensional space in their heads. But beyond that seems impossible.[1]

In addition to being hard to visualize, there are at least two additional problems in high dimensions, both refered to as **the curse of dimensionality**. One is computational, the other is mathematical.

From a computational perspective, consider the following problem. For K-nearest neighbors, the speed of prediction is slow for a very large data set. At the very least you have to look at every training example every time you want to make a prediction. To speed things up you might want to create an *indexing* data structure. You can break the plane up into a grid like that shown in Figure 3.15. Now, when the test point comes in, you can quickly identify the grid cell in which it lies. Now, instead of considering *all* training points, you can limit yourself to training points *in that grid cell* (and perhaps the neighboring cells). This can potentially lead to huge computational savings.

In two dimensions, this procedure is effective. If we want to break space up into a grid whose cells are 0.2×0.2, we can clearly do this with 25 grid cells in two dimensions (assuming the range of the features is 0 to 1 for simplicity). In three dimensions, we'll need $125 = 5{\times}5{\times}5$ grid cells. In four dimensions, we'll need 625. By the time we get to "low dimensional" data in 20 dimensions, we'll need 95,367,431,640,625 grid cells (that's 95 trillion, which is about 6 to 7 times the US national debt as of January 2011). So if you're in 20 dimensions, this gridding technique will only be useful if you have at least 95 trillion training examples.

[1] If you want to try to get an intuitive sense of what four dimensions looks like, I highly recommend the short 1884 book *Flatland: A Romance of Many Dimensions* by Edwin Abbott Abbott. You can even read it online at `gutenberg.org/ebooks/201`.

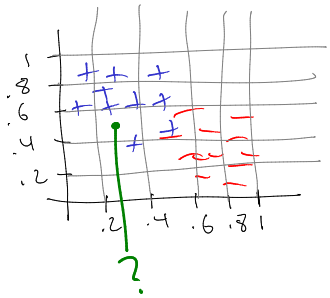


Figure 3.15: 2d knn with an overlaid grid, cell with test point highlighted

For "medium dimensional" data (approximately 1000) dimesions, the number of grid cells is a 9 followed by 698 numbers before the decimal point. For comparison, the number of atoms in the universe is approximately 1 followed by 80 zeros. So even if each atom yielded a googul training examples, we'd still have far fewer examples than grid cells. For "high dimensional" data (approximately 100000) dimensions, we have a 1 followed by just under 70,000 zeros. Far too big a number to even really comprehend.

Suffice it to say that for even moderately high dimensions, the amount of computation involved in these problems is enormous.

In addition to the computational difficulties of working in high dimensions, there are a large number of strange mathematical occurances there. In particular, many of your intuitions that you've built up from working in two and three dimensions just do not carry over to high dimensions. We will consider two effects, but there are countless others. The first is that high dimensional spheres look more like porcupines than like balls.[2] The second is that distances between points in high dimensions are all approximately the same.

? How does the above analysis relate to the number of data points you would need to fill out a full decision tree with D-many features? What does this say about the importance of shallow trees?

Let's start in two dimensions as in Figure 3.16. We'll start with four green spheres, each of radius one and each touching exactly two other green spheres. (Remember that in two dimensions a "sphere" is just a "circle.") We'll place a red sphere in the middle so that it touches all four green spheres. We can easily compute the radius of this small sphere. The pythagorean theorem says that $1^2 + 1^2 = (1 + r)^2$, so solving for r we get $r = \sqrt{2} - 1 \approx 0.41$. Thus, by calculation, the blue sphere lies entirely within the cube (cube = square) that contains the grey spheres. (Yes, this is also obvious from the picture, but perhaps you can see where this is going.)

[2] This result was related to me by Mark Reid, who heard about it from Marcus Hutter.

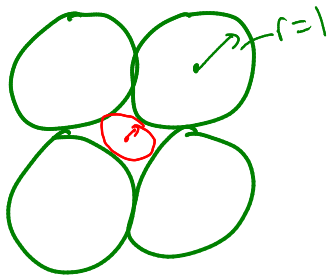


Figure 3.16: 2d spheres in spheres

Now we can do the same experiment in three dimensions, as shown in Figure 3.17. Again, we can use the pythagorean theorem to compute the radius of the blue sphere. Now, we get $1^2 + 1^2 + 1^2 = (1 + r)^2$, so $r = \sqrt{3} - 1 \approx 0.73$. This is still entirely enclosed in the cube of width four that holds all eight grey spheres.

At this point it becomes difficult to produce figures, so you'll have to apply your imagination. In four dimensions, we would have 16 green spheres (called **hyperspheres**), each of radius one. They would still be inside a cube (called a **hypercube**) of width four. The blue hypersphere would have radius $r = \sqrt{4} - 1 = 1$. Continuing to five dimensions, the blue hypersphere embedded in 256 green hyperspheres would have radius $r = \sqrt{5} - 1 \approx 1.23$ and so on.

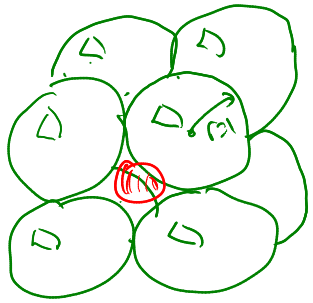


Figure 3.17: 3d spheres in spheres

In general, in D-dimensional space, there will be 2^D green hyperspheres of radius one. Each green hypersphere will touch exactly n-many other hyperspheres. The blue hyperspheres in the middle will touch them all and will have radius $r = \sqrt{D} - 1$.

Think about this for a moment. As the number of dimensions grows, the radius of the blue hypersphere *grows without bound!*. For example, in 9-dimensions the radius of the blue hypersphere is now $\sqrt{9} - 1 = 2$. But with a radius of two, the blue hypersphere is now "squeezing" between the green hypersphere and *touching* the edges of the hypercube. In 10 dimensional space, the radius is approximately 2.16 and it pokes outside the cube.

The second strange fact we will consider has to do with the distances between points in high dimensions. We start by considering random points in one dimension. That is, we generate a fake data set consisting of 100 random points between zero and one. We can do the same in two dimensions and in three dimensions. See Figure **??** for data distributed uniformly on the **unit hypercube** in different dimensions.

Now, pick two of these points at random and compute the distance between them. Repeat this process for all pairs of points and average the results. For the data shown in Figure **??**, the average distance between points in one dimension is about 0.346; in two dimensions is about 0.518; and in three dimensions is 0.615. The fact that these *increase* as the dimension increases is not surprising. The furthest two points can be in a 1-dimensional hypercube (line) is 1; the furthest in a 2-dimensional hypercube (square) is $\sqrt{2}$ (opposite corners); the furthest in a 3-d hypercube is $\sqrt{3}$ and so on. In general, the furthest two points in a D-dimensional hypercube will be $\sqrt{D}$.

You can actually compute these values analytically. Write $\mathcal{U}ni_D$ for the uniform distribution in D dimensions. The quantity we are interested in computing is:

$$avgDist(D) = \mathbb{E}_{\boldsymbol{a} \sim \mathcal{U}ni_D} \Big[\mathbb{E}_{\boldsymbol{b} \sim \mathcal{U}ni_D} \Big[||\boldsymbol{a} - \boldsymbol{b}|| \Big] \Big] \tag{3.2}$$

We can actually compute this in closed form and arrive at $avgDist(D) = \sqrt{D}/3$. Because we know that the maximum distance between two points grows like $\sqrt{D}$, this says that the ratio between average distance and maximum distance converges to $1/3$.

What is more interesting, however, is the *variance* of the distribution of distances. You can show that in D dimensions, the variance is *constant* $1/\sqrt{18}$, *independent of* D. This means that when you look at (variance) divided-by (max distance), the variance behaves like $1/\sqrt{18D}$, which means that the effective variance continues to shrink as D grows[3].

When I first saw and re-proved this result, I was skeptical, as I imagine you are. So I implemented it. In Figure 3.18 you can see the results. This presents a *histogram* of distances between random points in D dimensions for $D \in \{1, 2, 3, 10, 20, 100\}$. As you can see, all of these distances begin to concentrate around $0.4\sqrt{D}$, even for

[3] Brin 1995

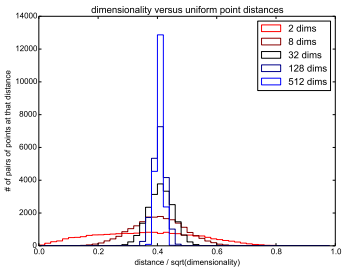


Figure 3.18: histogram of distances in D=2,8,32,128,512

"medium dimension" problems.

You should now be terrified: the only bit of information that KNN gets is distances. And you've just seen that in moderately high dimensions, all distances becomes equal. So then isn't it the case that KNN simply cannot work?

The answer has to be no. The reason is that the data that we get is *not* uniformly distributed over the unit hypercube. We can see this by looking at two real-world data sets. The first is an image data set of hand-written digits (zero through nine); see Section **??**. Although this data is originally in 256 dimensions (16 pixels by 16 pixels), we can artifically reduce the dimensionality of this data. In Figure 3.19 you can see the histogram of average distances between points in this data at a number of dimensions.

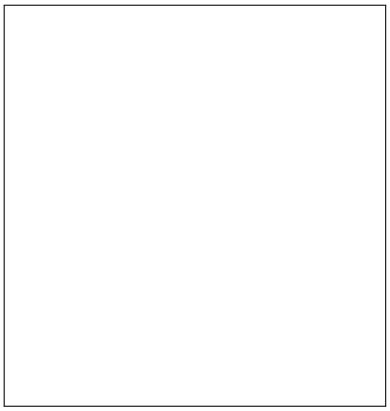


Figure 3.19: `knn:mnist`: histogram of distances in multiple D for mnist

As you can see from these histograms, distances have *not* concentrated around a single value. This is very good news: it means that there is hope for learning algorithms to work! Nevertheless, the moral is that high dimensions are weird.

3.6 Further Reading

TODO further reading