Chapter 3: Geometry and Nearest Neighbors

As we learned in the previous readings, when you have an input that you are trying to analyze to release an output, you could break it down into a colleciton of features. This suggests a **geometric view** of the data where we have on dimension for every feature.

Now that we have them as a geometric view, we can even perform geometric operations on them as well.

We can also compare the predictions that we have. For instance, if Ahmed and Yousef have similar preferences of courses, and we are trying to predict whether Ahmed will like Algorithm. But if Ahmed will like Algorithms, then Yousef will probably will, and this is called a **nearest neighbor**.

3.1 From Data to Feature Vectors

An example is just a collection of feature values of that example, and this includes a student writing the word "excellant" in a review of a course.

Another feature include the number of exclamation points in the review. Another includes any text that is underlined.

But to a machine learning algorithm, the **features** themselves have no meaning, but the **featuers values** have meanings to the algorithm and how they vary from one example to the other.

Thus, from this prospective, we can say that those feature values can be represented in a vector called **feature vector** where each dimension is just a feature value.

For example, a student with 3 excellant words, 4 exclamation points, and no underlined text has the following vector form: $\langle 3, 4, 0 \rangle$ and so on as you can imagine.

Binary features , on the other hand, are unique because they only take two distinct values such as 'Yes' or 'No' and 0/1 and so on and we will talk about this later.

But it is hard to convert features values that are of **catogarical features** type from features to vector form.

For example, if we are determining whether an image is tomato, blueberry, or cucumber, or cockroach, do we classify them based on their colors: red, blue, green, or black?

One might say that we can map the color red to 0, blue to 1, green to 2, and black to 3. But this is not a good way to do this because if we are going to represent them geomtrically using vectors, then we are saying that that category has an order which it does not.

For example, when we use these feature, 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). But, this is probably not what we want to say!

A better solution is to turn this categorical feature into four different binary values: (is it blue?, is it red?, is it green?, is it black?).

Therefore, in general, if we have a categoral feature that takes V-values, then we can turn into V-many binary indicator values.

Therefore, we have the following things which can be used to map each example to a feature vector:

- Real-valued features get copied directly as we have seen with the review example.
- Binary features become 0 for false and 1 for true.
- categorical features with V possible values get mapped into V-many binary indicator features.

Therefore, now you can think of a single example as a **vector** in a **vector space**.

If you have a category feature and you expanded it into V-many binary indicator, then this will be stored in a vector in this form: $x = \langle x_1, x_2, ..., x_D \rangle$ which means that the vector space that we are working with will be based on how many D's there are. It is a D-dimension based vector space based on the number of categorical features there are when they are expanded.

3.2: K-Nearest Neighbors

So, the biggest advantage to thinking of examples as vectors is that you can apply geometric concepts to machine learning.

For example, one of the things that you can do in a vector space is compute **distances**. In two-dimesnional space, the distance between $\langle 2, 3\rangle$ and $\langle 6, 1\rangle$ is: $\sqrt{(2-6)^2 + (3-1)^2} \approx 4.24$.

But, in general, in any D-dimensional vector space, the **Euclidean distance** between vectors a and b is given by the following formula:

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

Now that we have all that, consider a vector space where we have a bunch of positive signs and a bunch of negative signs and one question mark sign among them. That question mark happens to be among the positive sign bunch, and you are asked to guess the sign of that question mark. What would you say?

You will probably say it is a positive sign, and that is a form of **inductive bias**. The **nearest neighbor** classifier is based on this insight.

What you do it simple: you graph the training set and you are given a test set which consists of one member $\hat{x}$. You are trying to predict what $\hat{x}$ so you try to find the closest member of the training set or example x to $\hat{x}$.

So, you basically find the number x that basically minimizes $d(x, \hat{x})$ and that is the intution behind the nearest neighbor classifier.

Despite its simplicity, the nearest neighbor classifier is incredibly effective. However, there is one issue with this.

Suppose you have a bunch of positive signs clustered together and there is one negative sign among those positive charges. Suppose

there is a question mark right next to the negative sign, what would the nearest neighbor classifier say?

It would say that the negative sign should be in place of that question mark! However, this is not right since the surrounding signs are positive which is why we have **k-nearest neighbors**.

If you have 3-nearest neighbors, then we would get 2 positive charges and one negative charge, which means that the question mark will be positive charge because the majority of the closest 3 neigbors are positive.

Before, we write out the algorithm, we have the following convention:

- The training data is denoted by D.
- We assume there are N-many training examples.
- These examples are pairs: $(x_1, y_1), (x_2, y_2)...., (x_N, y_N)$.
- Do not confuse x_N which denotes the Nth example with x_d which denotes the dth feature.
- We also use [] to denote to an empty list and use $\oplus$ to append to the list.
- Our prediction on $\hat{x}$ is $\hat{y}$.

Therefore, we have the following algorithm:

Algorithm 1 KNNN Predict(D, k, $\hat{x}$)

procedure MyProcedure
  $S \leftarrow [\ ]$
  for $n = 1$ **to** N **do**
    $S \leftarrow S \oplus \langle d(x, \hat{x}), n \rangle$ ▷ store distance to the array
  end for
  $S \leftarrow SORT(S)$ ▷ sort from lowest to highest
  $\hat{y} \leftarrow 0$
  for $k = 1$ **to** K **do** ▷ Depends on K: if we want to get the 3 (K = 3) nearest neigbor
    $\langle dist, n \rangle \leftarrow S_k$
    $\hat{y} \leftarrow \hat{y} + y_n$ ▷ Adding the *labels* then assigning
  end for
  return SIGN($\hat{y}$) ▷ return +1 if $\hat{y} > 0$ and otherwise -1
end procedure

One flaw with the nearest neighbor classifier is that it treats all features the same weight. Meaning, if your example had very few relevant features and a lot of irrelevant features, then KNNN will preform poorly and this is, as mentioned before, is part of **inductive bias**.

Another thing is **feature scale**. If you are trying to decide whether a picture is a floor or a sky, and you are measuring pictures based on their width and height in cm, then KNNN would be fine.

However, if you switched the width to mm, then all the data points will be clustered together which means that KNNN will preform poorly because it will say that width feature is same as height and will judge mostly on height.

3.3: Decision Bounderies

When you ask yourself: What sort of examples from the test set will be positive or negative, you are essentially saying that there will be regions in the vector space that *would* be positive and some that *would* be negative.

There will also be a solid line seperating the two regions from each other, and that is called **decision boundary**. On one side is the positive side and the other is the negative side.

The decision boundary allows us to simplify the complexity of the model. If you have a jagged line, then you would be *overfitting* the model and if you have a really simple line then you would be *underfitting* the model.

3.5 Warning: High Dimensions Are Scary

If you have a lot of dimensions, they will be very hard to visualize for humans. Additionally and more importantly, two issues will be born if we were to use KNNN on very large dimensions and those are called **the curse of dimensionality**: computational and mathematical.

In terms of computationality, it will take *a lot* of time to try to compute the distance of each example in the training set. One solution to this is to divide up the graphing grid into little squares and once you confront with a test data, you calculate which grid that set will be in and then calculate the all of the distances of each example in *that* grid.

For example, if you have a 0.2 x 0.2 grid cell and you are working with 2-dimensional vector space, we can have 25 total grid cells (assuming the range to be from 0 to 1 for simplicity). If you have 3-dimensions, then it is 125 = 5 x 5 x 5, and so on.

Each device controller has a