Nearest Neighbour Classifier

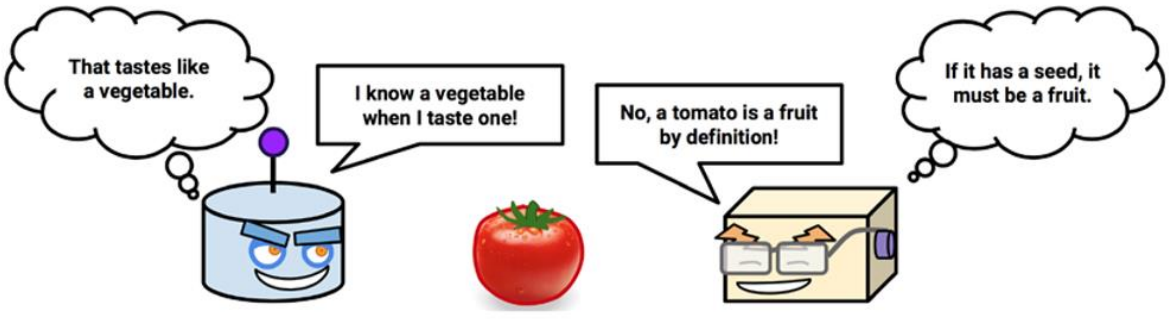
Lazy Learning – Classification Using Nearest Neighbors

An interesting new type of dining experience has been appearing in cities around the world. Patrons are served in a completely darkened restaurant by waiters who move carefully around memorized routes using only their sense of touch and sound. The allure of these establishments is the belief that depriving oneself of visual sensory input will enhance the sense of taste and smell, and foods will be experienced in new ways. Each bite provides a sense of wonder while discovering the flavors the chef has prepared.

Can you imagine how a diner experiences the unseen food? Upon first bite, the senses are overwhelmed. What are the dominant flavors? Does the food taste savory or sweet? Does it taste similar to something eaten previously? Personally, I imagine this process of discovery in terms of a slightly modified adage: if it smells like a duck and tastes like a duck, then you are probably eating duck.

This illustrates an idea that can be used for machine learning—as does another maxim involving poultry: "birds of a feather flock together." Stated differently, things that are alike are likely to have properties that are alike. Machine learning uses this principle to classify data by placing it in the same category as similar or "nearest" neighbors.

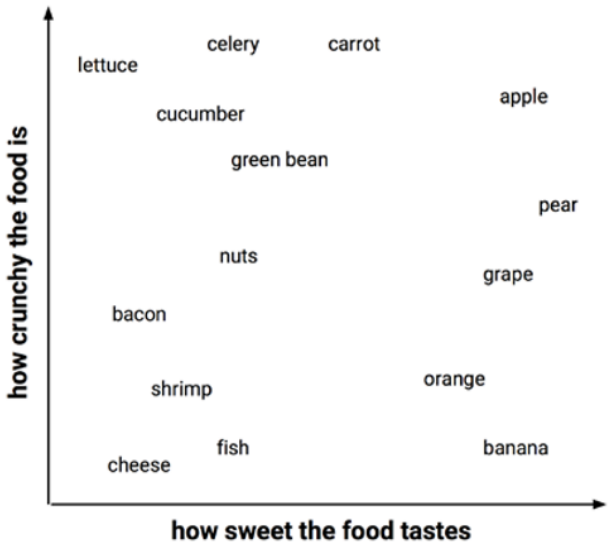
d



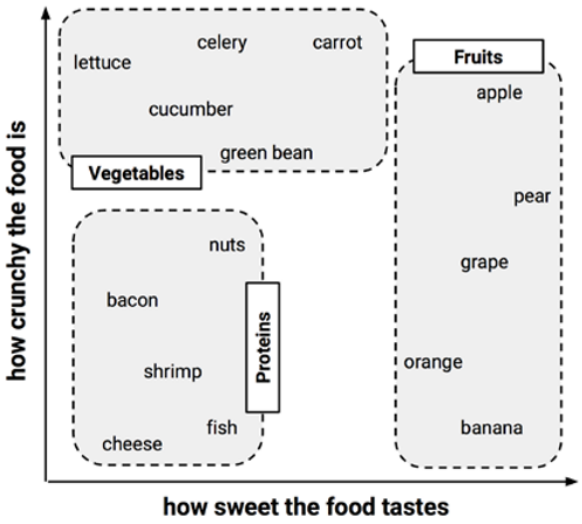
To illustrate this process, let's revisit the blind tasting experience described in the introduction. Suppose that prior to eating the mystery meal we had created a dataset in which we recorded our impressions of a number of ingredients we tasted previously. To keep things simple, we rated only two features of each ingredient. The first is a measure from 1 to 10 of how crunchy the ingredient is and the second is a 1 to 10 score of how sweet the ingredient tastes. We then labeled each ingredient as one of the three types of food: fruits, vegetables, or proteins. The first few rows of such a dataset might be structured as follows:

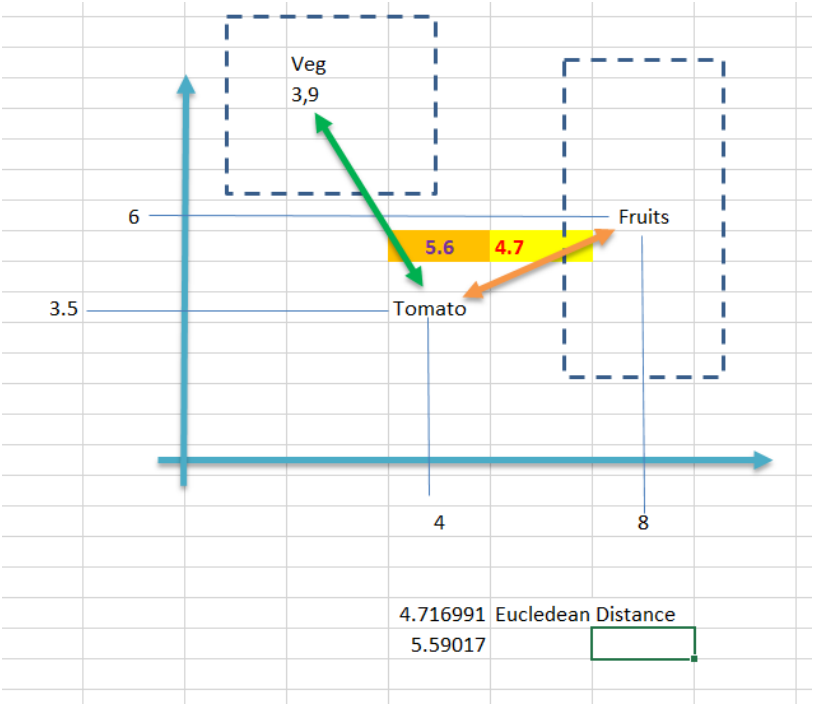
Ingredient	Sweetness	Crunchiness	Food type
apple	10	9	fruit
bacon	1	4	protein
banana	10	1	fruit
carrot	7	10	vegetable
celery	3	10	vegetable
cheese	1	1	protein

The k-NN algorithm treats the features as coordinates in a multidimensional feature space. As our dataset includes only two features, the feature space is two-dimensional. We can plot two-dimensional data on a scatter plot, with the x dimension indicating the ingredient's sweetness and the y dimension, the crunchiness. After adding a few more ingredients to the taste dataset, the scatter plot might look similar to this:

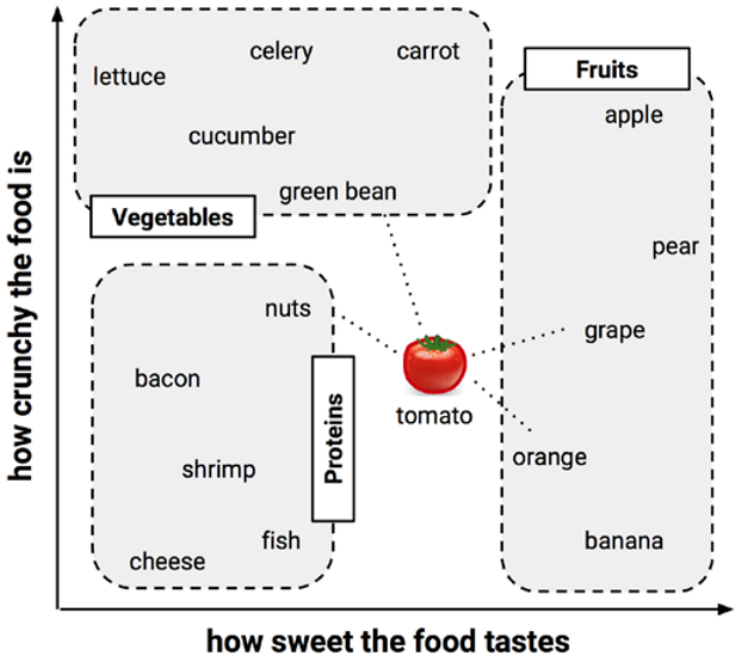


Did you notice the pattern? Similar types of food tend to be grouped closely together. As illustrated in the next diagram, vegetables tend to be crunchy but not sweet, fruits tend to be sweet and either crunchy or not crunchy, while proteins tend to be neither crunchy nor sweet:





Suppose that after constructing this dataset, we decide to use it to settle the age-old question: is tomato a fruit or vegetable? We can use the nearest neighbor approach to determine which class is a better fit, as shown in the following diagram:



Measuring similarity with distance

Locating the tomato's nearest neighbors requires a **distance function**, or a formula that measures the similarity between the two instances.

There are many different ways to calculate distance. Traditionally, the k-NN algorithm uses **Euclidean distance**, which is the distance one would measure if it were possible to use a ruler to connect two points, illustrated in the previous figure by the dotted lines connecting the tomato to its neighbors.

Euclidean distance is measured "as the crow flies," implying the shortest direct route. Another common distance measure is Manhattan distance, which is based on the paths a pedestrian would take by walking city blocks. If you are interested in learning more about other distance measures, you can read the documentation for R's distance function (a useful tool in its own right), using the `?dist` command.

Euclidean distance is specified by the following formula, where p and q are the examples to be compared, each having n features. The term p_1 refers to the value of the first feature of example p, while q_1 refers to the value of the first feature of example q:

$$\text{dist}(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$$

The distance formula involves comparing the values of each feature. For example, to calculate the distance between the tomato (*sweetness* = 6, *crunchiness* = 4), and the green bean (*sweetness* = 3, *crunchiness* = 7), we can use the formula as follows:

$$\text{dist}(\text{tomato, green bean}) = \sqrt{(6 - 3)^2 + (4 - 7)^2} = 4.2$$

In a similar vein, we can calculate the distance between the tomato and several of its closest neighbors as follows:

Ingredient	Sweetness	Crunchiness	Food type	Distance to the tomato
grape	8	5	fruit	*sqrt((6 - 8)^2 + (4 - 5)^2) = 2.2*
green bean	3	7	vegetable	*sqrt((6 - 3)^2 + (4 - 7)^2) = 4.2*
nuts	3	6	protein	*sqrt((6 - 3)^2 + (4 - 6)^2) = 3.6*
orange	7	3	fruit	*sqrt((6 - 7)^2 + (4 - 3)^2) = 1.4*

To classify the tomato as a vegetable, protein, or fruit, we'll begin by assigning the tomato, the food type of its single nearest neighbor. This is called 1-NN classification because $k = 1$. The orange is the nearest neighbor to the tomato, with a distance of 1.4. As orange is a fruit, the 1-NN algorithm would classify tomato as a fruit.

If we use the k-NN algorithm with $k = 3$ instead, it performs a vote among the three nearest neighbors: orange, grape, and nuts. Since the majority class among these neighbors is fruit (two of the three votes), the tomato again is classified as a fruit.

Choosing an appropriate k

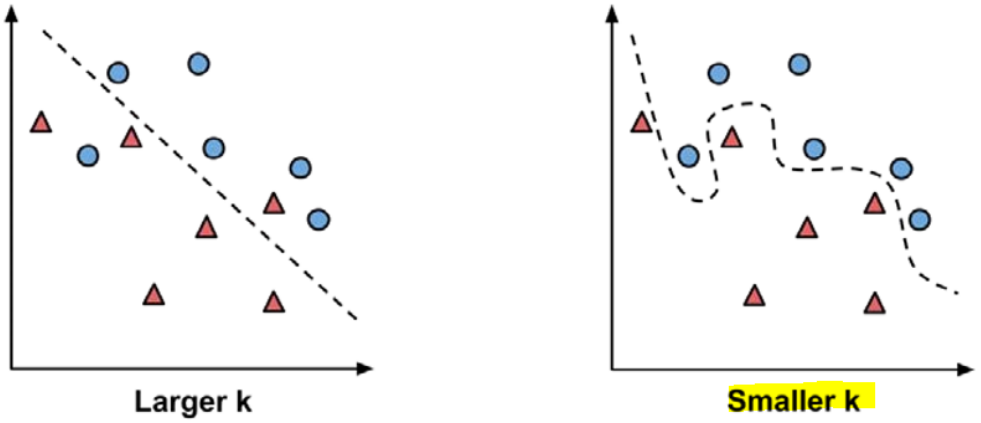
The decision of how many neighbors to use for k-NN determines how well the model will generalize to future data. The balance between overfitting and underfitting the training data is a problem known as **bias-variance tradeoff**. Choosing a large k reduces the impact or variance caused by noisy data, but can bias the learner so that it runs the risk of ignoring small, but important patterns.

Suppose we took the extreme stance of setting a very large k, as large as the total number of observations in the training data. With every training instance represented in the final vote, the most common class always has a majority of the voters. The model would consequently always predict the majority class, regardless of the nearest neighbors.

On the opposite extreme, using a single nearest neighbor allows the noisy data or outliers to unduly influence the classification of examples. For example, suppose some of the training examples were accidentally mislabeled. Any unlabeled example that happens to be nearest to the incorrectly labeled neighbor will be predicted to have the incorrect class, even if nine other nearest neighbors would have voted differently.

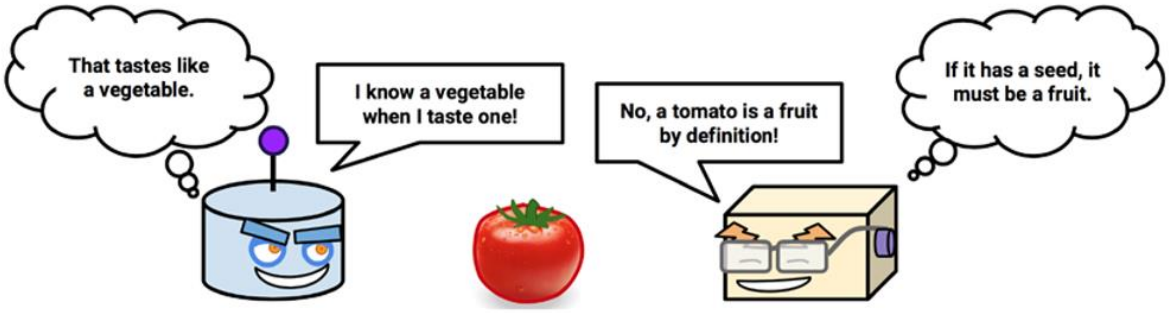
Obviously, the best k value is somewhere between these two extremes.

The following figure illustrates, more generally, how the decision boundary (depicted by a dashed line) is affected by larger or smaller k values. Smaller values allow more complex decision boundaries that more carefully fit the training data. The problem is that we do not know whether the straight boundary or the curved boundary better represents the true underlying concept to be learned.



In practice, choosing k depends on the difficulty of the concept to be learned, and the number of records in the training data. One common practice is to begin with k equal to the square root of the number of training examples. In the food classifier we developed previously, we might set $k = 4$ because there were 15 example ingredients in the training data and the square root of 15 is 3.87.

However, such rules may not always result in the single best k. An alternative approach is to test several k values on a variety of test datasets and choose the one that delivers the best classification performance. That said, unless the data is very noisy, a large training dataset can make the choice of k less important. This is because even subtle concepts will have a sufficiently large pool of examples to vote as nearest neighbors.



Why is the k-NN algorithm lazy?

Classification algorithms based on the nearest neighbor methods are considered **lazy learning** algorithms because, technically speaking, no abstraction occurs. The abstraction and generalization processes are skipped altogether, and this undermines the definition of learning, proposed in *Chapter 1, Introducing Machine Learning*.

Under the strict definition of learning, a lazy learner is not really learning anything. Instead, it merely stores the training data verbatim. This allows the training phase, which is not actually training anything, to occur very rapidly. Of course, the downside is that the process of making predictions tends to be relatively slow in comparison to training. Due to the heavy reliance on the training instances rather than an abstracted model, lazy learning is also known as **instance-based learning** or **rote learning**.