When do we use KNN algorithm?

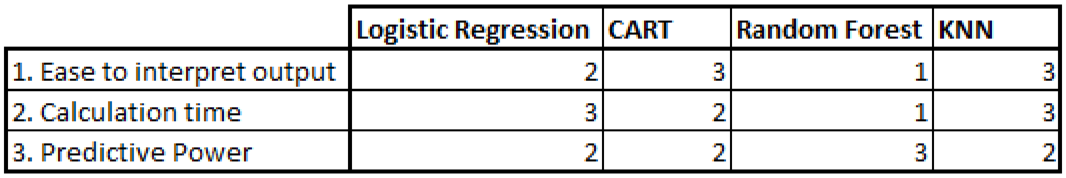
KNN can be used for both classification and regression predictive problems. However, it is more widely used in classification problems in the industry. To evaluate any technique, we generally look at 3 important aspects:

1. Ease to interpret output

2. Calculation time

3. Predictive Power

Let us take a few examples to place KNN in the scale:

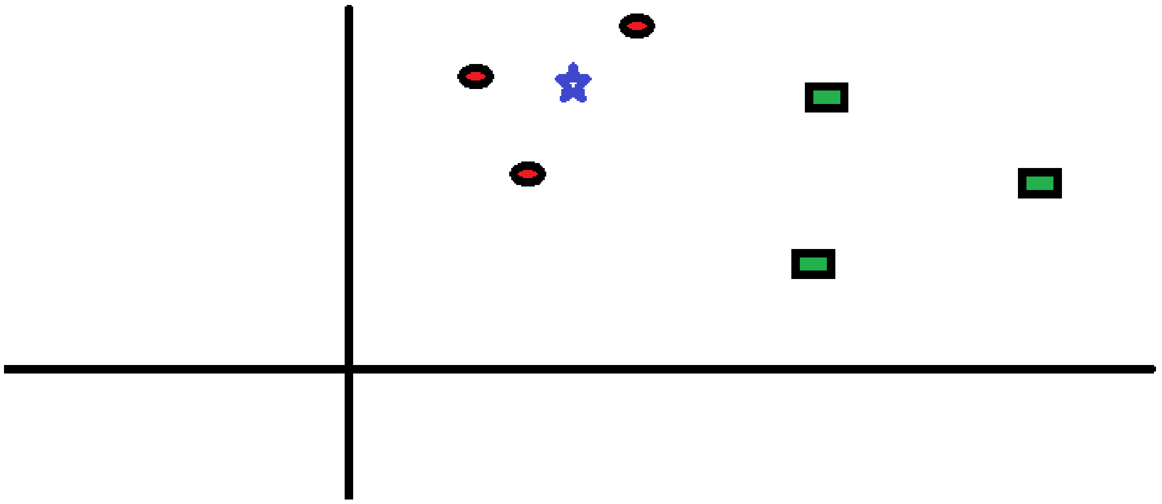


	Logistic Regression	CART	Random Forest	KNN
1. Ease to interpret output	2	3	1	3
2. Calculation time	3	2	1	3
3. Predictive Power	2	2	3	2

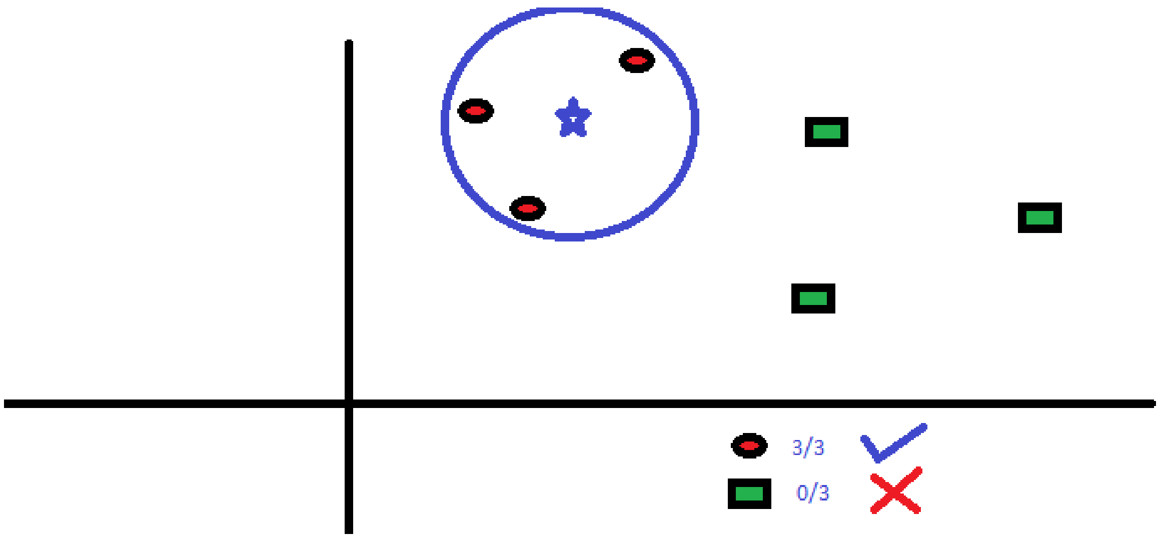
KNN algorithm fairs across all parameters of considerations. It is commonly used for its easy of interpretation and low calculation time.

How does the KNN algorithm work?

Let’s take a simple case to understand this algorithm. Following is a spread of red circles (RC) and green squares (GS):



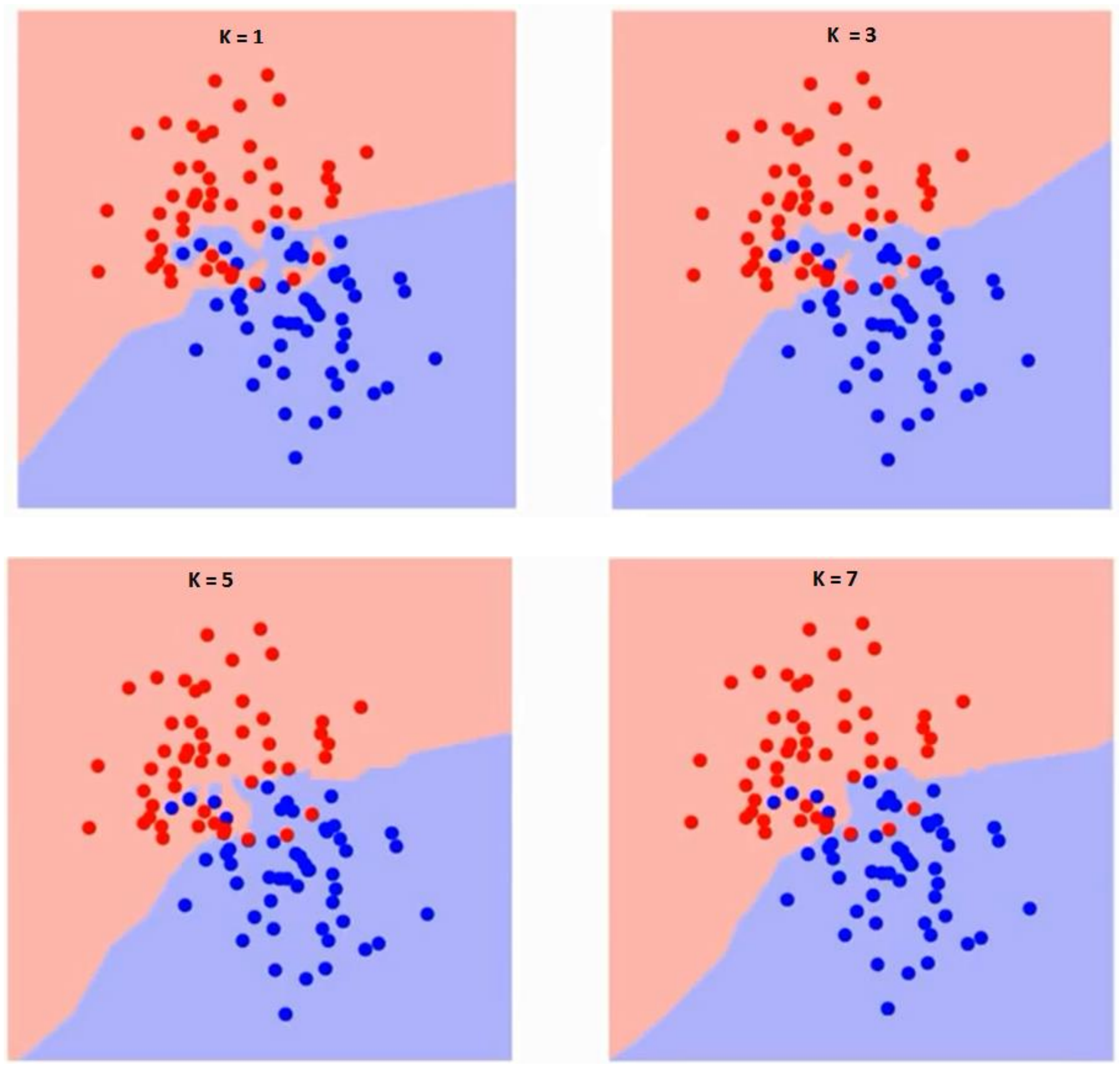
You intend to find out the class of the blue star (BS) . BS can either be RC or GS and nothing else. The “K” is KNN algorithm is the nearest neighbors we wish to take vote from. Let’s say K = 3. Hence, we will now make a circle with BS as center just as big as to enclose only three datapoints on the plane. Refer to following diagram for more details:



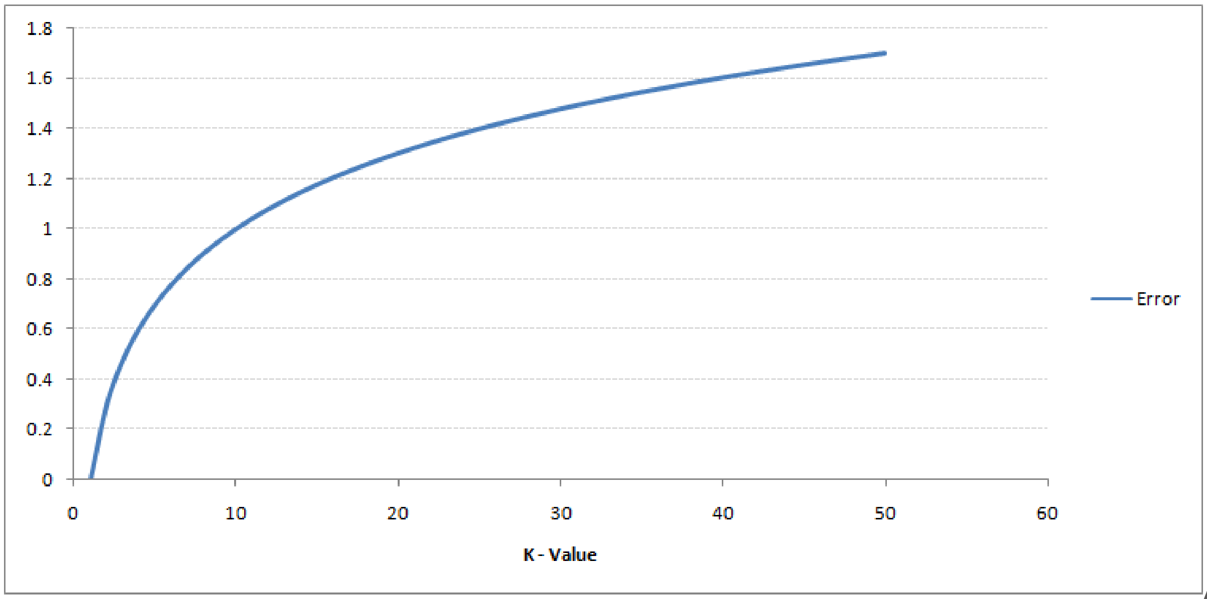
The three closest points to BS is all RC. Hence, with good confidence level we can say that the BS should belong to the class RC. Here, the choice became obvious as all three votes from the closest neighbor went to RC. The choice of the parameter K is very crucial in this algorithm. Next we will understand what are the factors to be considered to conclude the best K.

How do we choose the factor K?

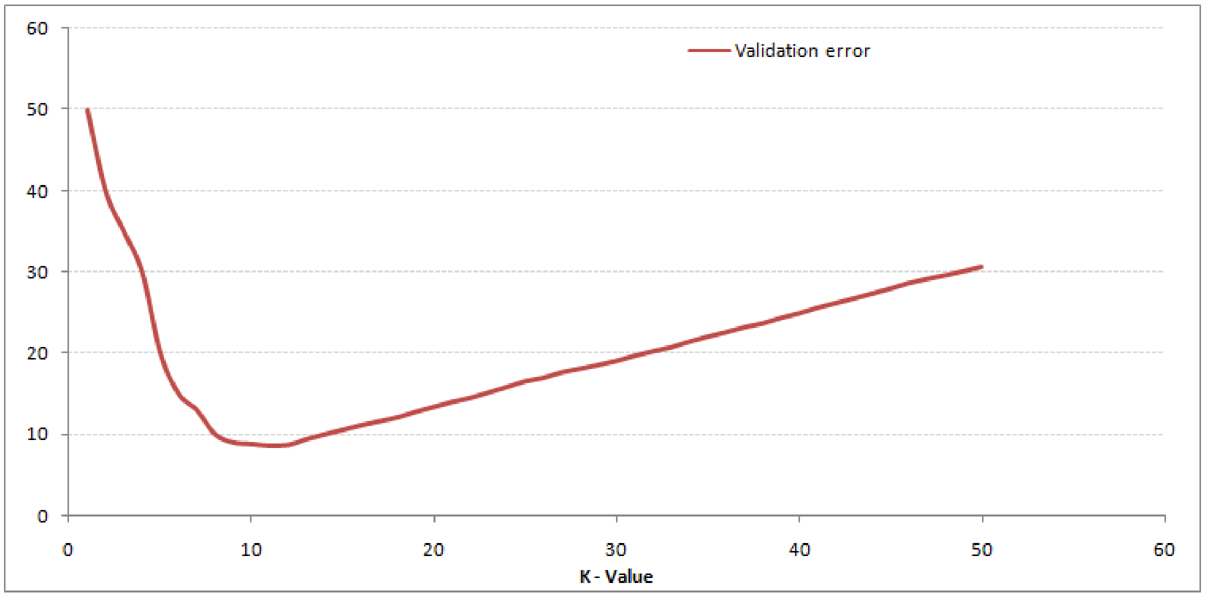
First let us try to understand what exactly does K influence in the algorithm. If we see the last example, given that all the 6 training observation remain constant, with a given K value we can make boundaries of each class. These boundaries will segregate RC from GS. The same way, let’s try to see the effect of value “K” on the class boundaries. Following are the different boundaries separating the two classes with different values of K.



If you watch carefully, you can see that the boundary becomes smoother with increasing value of K. With K increasing to infinity it finally becomes all blue or all red depending on the total majority. The training error rate and the validation error rate are two parameters we need to access on different K-value. Following is the curve for the training error rate with varying value of K :



As you can see, the error rate at K=1 is always zero for the training sample. This is because the closest point to any training data point is itself.Hence the prediction is always accurate with K=1. If validation error curve would have been similar, our choice of K would have been 1. Following is the validation error curve with varying value of K:



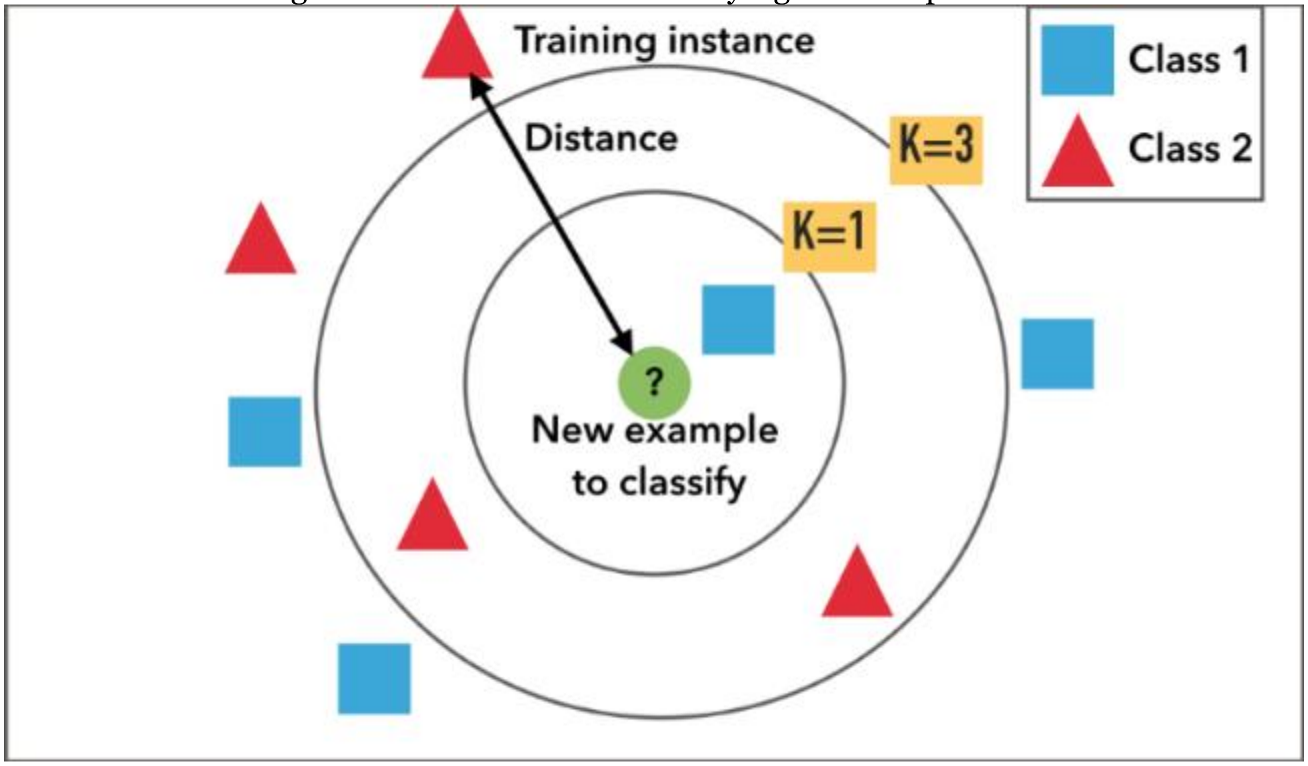
This makes the story more clear. At K=1, we were overfitting the boundaries. Hence, error rate initially decreases and reaches a minima. After the minima point, it then increase with increasing K. To get the optimal value of K, you can segregate the training and validation from the initial dataset. Now plot the validation error curve to get the optimal value of K. This value of K should be used for all predictions.

Breaking it Down – Pseudo Code of KNN

We can implement a KNN model by following the below steps:

1. Load the data
2. Initialize the value of k
3. For getting the predicted class, iterate from 1 to total number of training data points
    1. Calculate the distance between test data and each row of training data. Here we will use Euclidean distance as our distance metric since it's the most popular method. The other metrics that can be used are Chebyshev, cosine, etc.
    2. Sort the calculated distances in ascending order based on distance values
    3. Get top k rows from the sorted array
    4. Get the most frequent class of these rows
    5. Return the predicted class

KNN Algorithm is based on **feature similarity**: How closely out-of-sample features resemble our training set determines how we classify a given data point:



Example of k-NN classification. The test sample (inside circle) should be classified either to the first class of blue squares or to the second class of red triangles. If k = 3 (outside circle) it is assigned to the second class because there are 2 triangles and only 1 square inside the inner circle. If, for example k = 5 it is assigned to the first class (3 squares vs. 2 triangles outside the outer circle).

KNN can be used for **classification**—the output is a class membership (predicts a class—a discrete value). An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. It can also be used for **regression**—output is the value for the object (predicts continuous values). This value is the average (or median) of the values of its k nearest neighbors.

A few Applications and Examples of KNN

- Credit ratings—collecting financial characteristics vs. comparing people with similar financial features to a database. By the very nature of a credit rating, people who have similar financial details would be given similar credit ratings. Therefore, they would like to be able to use this existing database to predict a new customer's credit rating, without having to perform all the calculations.
- Should the bank give a loan to an individual? Would an individual default on his or her loan? Is that person closer in characteristics to people who defaulted or did not default on their loans?
- In political science—classing a potential voter to a "will vote" or "will not vote", or to "vote Democrat" or "vote Republican".
- More advance examples could include handwriting detection (like OCR), image recognition and even video recognition.

Some pros and cons of KNN

Pros:

- No assumptions about data—useful, for example, for nonlinear data
- Simple algorithm—to explain and understand/interpret
- High accuracy (relatively)—it is pretty high but not competitive in comparison to better supervised learning models

- Versatile—useful for classification or regression

Cons:

- Computationally expensive—because the algorithm stores all of the training data
- High memory requirement
- Stores all (or almost all) of the training data
- Prediction stage might be slow (with big N)
- Sensitive to irrelevant features and the scale of the data

Quick summary of KNN

The algorithm can be summarized as:

1. A positive integer k is specified, along with a new sample
2. We select the k entries in our database which are closest to the new sample
3. We find the most common classification of these entries
4. This is the classification we give to the new sample

A few other features of KNN:

- KNN stores the entire training dataset which it uses as its representation.
- KNN does not learn any model.
- KNN makes predictions just-in-time by calculating the similarity between an input sample and each training instance.

The **k-Nearest-Neighbors (kNN)** method of classification is one of the simplest methods in machine learning, and is a great way to introduce yourself to machine learning and classification in general. At its most basic level, it is essentially classification by finding the most similar data points in the training data, and making an educated guess based on their classifications. Although very simple to understand and implement, this method has seen wide application in many domains, such as in **recommendation systems**, **semantic searching**, and **anomaly detection**.

As we would need to in any machine learning problem, we must first find a way to represent data points as **feature vectors**. A feature vector is our mathematical representation of data, and since the desired characteristics of our data may not be inherently numerical, preprocessing and feature-engineering may be required in order to create these vectors. Given data with **N** unique features, the feature vector would be a vector of length **N**, where entry **I** of the vector represents that data point's value for feature **I**. Each feature vector can thus be thought of as a point in **R^N**.

Now, unlike most other methods of classification, kNN falls under **lazy learning**, which means that there is **no explicit training phase before classification**. Instead, any attempts to generalize or abstract the data is made upon classification. While this does mean that we can immediately begin classifying once we have our data, there are some inherent problems with this type of algorithm. We must be able to keep the entire training set in memory unless we apply some type of reduction to the data-set, and performing classifications can be computationally expensive as the algorithm parse through all data points for each classification. **For these reasons, kNN tends to work best on smaller data-sets that do not have many features.**

There are two important decisions that must be made before making classifications. One is the value of **k** that will be used; this can either be decided arbitrarily, or you can try **cross-validation** to find an optimal value. The next, and the most complex, is the **distance metric** that will be used.

There are many different ways to compute distance, as it is a fairly ambiguous notion, and the proper metric to use is always going to be determined by the data-set and the classification task. Two popular ones, however, are **Euclidean distance** and **Cosine similarity**.

Euclidean distance is probably the one that you are most familiar with; it is essentially the magnitude of the vector obtained by subtracting the training data point from the point to be classified.

$$E(x,y) = \sqrt{\sum_{i=0}^{n}(x_i - y_i)^2}$$

General formula for Euclidean distance

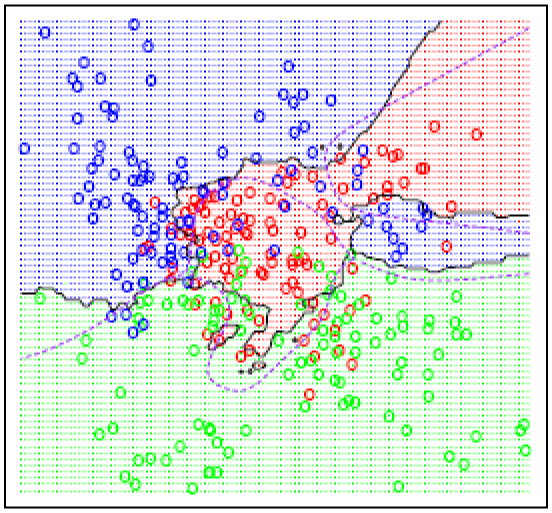
Another common metric is Cosine similarity. Rather than calculating a magnitude, Cosine similarity instead uses the difference in direction between two vectors.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

General formula for Cosine similarity

Choosing a metric can often be tricky, and it may be best to just use cross-validation to decide, unless you have some prior insight that clearly leads to using one over the other. For example, for something like word vectors, you may want to use Cosine similarity because the direction of a word is more meaningful than the sizes of the component values. Generally, both of these methods will run in roughly the same time, and will suffer from highly-dimensional data.

After doing all of the above and deciding on a metric, the result of the kNN algorithm is a decision boundary that partitions **R^N** into sections. Each section (colored distinctly below) represents a class in the classification problem. The boundaries need not be formed with actual training examples—they are instead calculated using the distance metric and the available training points. By taking **R^N** in (small) chunks, we can calculate the most likely class for a hypothetical data-point in that region, and we thus color that chunk as being in the region for that class.



This information is all that is needed to begin implementing the algorithm, and doing so should be relatively simple. There are, of course, many ways to improve upon this base algorithm. Common modifications include weighting, and specific preprocessing to reduce computation and reduce noise, such as various algorithms for feature extraction and dimension reduction. Additionally, the kNN method has also been used, although less-commonly, for regression tasks, and operates in a manner very similar to that of the classifier through averaging.