**K Nearest Neighbors classifier**

**Introduction**

The KNN algorithm is a robust and versatile classifier that is often used as a benchmark for more complex classifiers such as Artificial Neural Networks (ANN) and Support Vector Machines (SVM). Despite its simplicity, KNN can outperform more powerful classifiers and is used in a variety of applications such as economic forecasting, data compression and genetics..

**What is KNN?**

Let’s first start by establishing some definitions and notations. We will use

*X* to denote a *feature* (aka. predictor, attribute) and *y*

to denote the *target* (aka. label, class) we are trying to predict.

KNN falls in the **supervised learning** family of algorithms. Informally, this means that we are given a labelled dataset consiting of training observations (*x*,*y*)and would like to capture the relationship between x And *y* More formally, our goal is to learn a function *h*:*X*→*Y* so that given an unseen observation *x*

*h*(*x*) can confidently predict the corresponding output *y*.

The KNN classifier is also a **non parametric** and **instance-based** learning algorithm.

* **Non-parametric** means it makes no explicit assumptions about the functional form of h, avoiding the dangers of mismodeling the underlying distribution of the data. For example, suppose our data is highly non-Gaussian but the learning model we choose assumes a Gaussian form. In that case, our algorithm would make extremely poor predictions.
* **Instance-based** learning means that our algorithm doesn’t explicitly learn a model. Instead, it chooses to memorize the training instances which are subsequently used as “knowledge” for the prediction phase. Concretely, this means that only when a query to our database is made (i.e. when we ask it to predict a label given an input), will the algorithm use the training instances to spit out an answer.

It is worth noting that the minimal training phase of KNN comes both at a *memory cost*, since we must store a potentially huge data set, as well as a *computational cost* during test time since classifying a given observation requires a run down of the whole data set. Practically speaking, this is undesirable since we usually want fast responses.

How does KNN work?

In the classification setting, the K-nearest neighbor algorithm essentially boils down to forming a majority vote between the K most similar instances to a given “unseen” observation. Similarity is defined according to a distance metric between two data points. A popular choice is the Euclidean distance given by

but other measures can be more suitable for a given setting and include the Manhattan, Chebyshev and Hamming distance.

More formally, given a positive integer K, an unseen observation *x* and a similarity metric *d*, KNN classifier performs the following two steps:

* It runs through the whole dataset computing *d*    between *x*    and each training observation. We’ll call the K points in the training data that are closest to *x*    the set ** A  . Note that K is usually odd to prevent tie situations.
* It then estimates the conditional probability for each class, that is, the fraction of points in **    with that given class label. (Note *I*(*x*) is the indicator function which evaluates to 1 1 when the argument *x*   is true and 0   otherwise) gets assigned to the class with the largest probability.

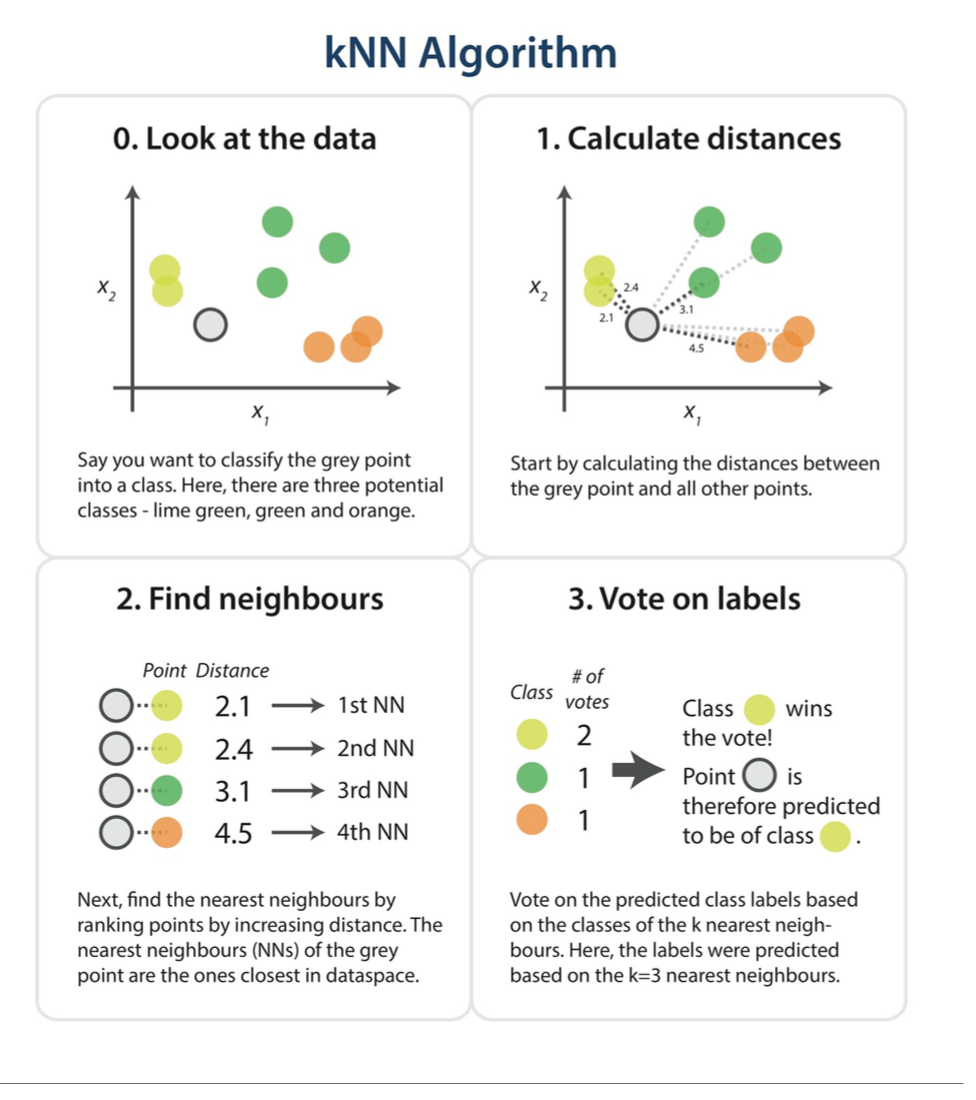
Pros and Cons of KNN

**Pros**

As you can already tell from the previous section, one of the most attractive features of the K-nearest neighbor algorithm is that is simple to understand and easy to implement. With zero to little training time, it can be a useful tool for off-the-bat analysis of some data set you are planning to run more complex algorithms on. Furthermore, KNN works just as easily with multiclass data sets whereas other algorithms are hardcoded for the binary setting. Finally, as we mentioned earlier, the non-parametric nature of KNN gives it an edge in certain settings where the data may be highly “unusual”.

**Cons**

One of the obvious drawbacks of the KNN algorithm is the computationally expensive testing phase which is impractical in industry settings. Note the rigid dichotomy between KNN and the more sophisticated Neural Network which has a lengthy training phase albeit a **very fast** testing phase. Furthermore, KNN can suffer from skewed class distributions. For example, if a certain class is very frequent in the training set, it will tend to dominate the majority voting of the new example (large number = more common). Finally, the accuracy of KNN can be severely degraded with high-dimension data because there is little difference between the nearest and farthest neighbor.



**Assignment :**

* Implement your own simple KNN classifier using python, (Don’t use any build in functions)
* Use provided train and test file yeast\_train.txt,yeast\_test.txt
* Each record in dataset contain feature values are separated by  commas, and the last value on each line is the class label
* If there is a tie in the class predicted by the *k*-nearest neighbors, then  among the classes that have the same number of votes, the tie should  be broken in favor of the class comes first in the Train file.
* Use Euclidean distance to compute distances between instances.
* Report accuracy on testing data when k=1,2,3....9.
* As output, your programs should print the value of *k* used for the test  set on the first line, each output line should list the predicted class  label, and actual class label.
* Also output the number of correctly classified test instances, and the  total number of instances in the test set & Accuracy.
* Example :

 k value : 3

 Predicted class : POX Actual class : CYT

.

.

.

Number of correctly classified instances : 238 Total number of instances : 445

Accuracy : 0.5348314606741573