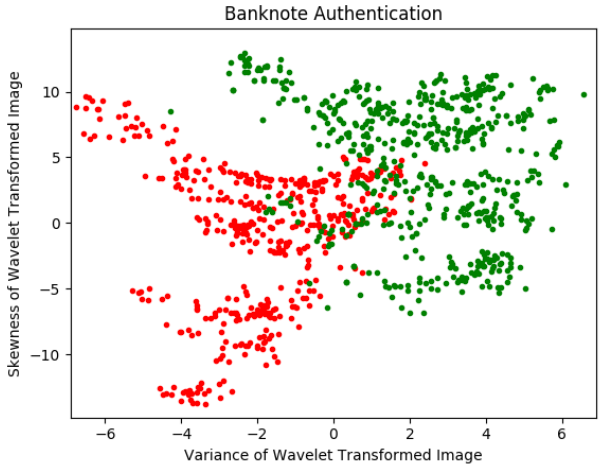
k-Nearest Neighbor Classification

© Sam Scott, Mohawk College, 2019

The Basic Assumption

In the scatterplot below, green data points represent authentic banknotes and red data points are counterfeit. The data has two numeric features (Skewness and Variance). Notice how the green and red data points form overlapping clusters or "neighbourhoods".

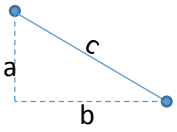


The basic idea behind k-Nearest Neighbour (kNN) classification is that when you get a new example, you should decide what label it gets by looking at its "nearest neighbours" from the training examples. The *k* parameter controls how many nearest neighbours will be consulted. Whatever class label is most common among the *k* nearest neighbours is chosen as the prediction for the new example.

In the banknote data, we would expect the approach to work well in the areas that are entirely red or green. In overlapping areas, kNN is unlikely to achieve 100% accuracy, but if more of the nearest neighbours are labelled "authentic", it's more likely that the new example should be labelled "authentic" as well.

Computing the Euclidean Distance

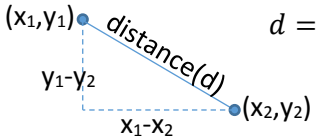
In two dimensions, you can use the Pythagorean Theorem to compute the distance between two points. The result is known as the Euclidean Distance.



$$c^2 = a^2 + b^2$$

$$c = \sqrt{a^2 + b^2}$$

The Pythagorean Theorem



$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Euclidean Distance (2D)

This procedure for computing Euclidean Distance is to square the differences in each dimension (feature), add them up, and take the square root. Euclidean distance measurement scales up to 3 or more dimensions, even though it's not possible to visualize it after 3 dimensions.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 + \cdots}$$

Euclidean Distance (3D+)

Pros and Cons of kNN

Pros: Accurate, copes well with outliers.

Cons: Resource use (time and memory)

Normalizing Your Data

Often the range of possible values for a feature will allow it to dominate the distance calculation. In the data shown here (representing fake user profiles on a dating app), one feature (Number of Frequent Flier Miles) ranges from 0 to 50000 or so, while another (Hours of Reality TV per week) ranges from 0 to 2. To see why this might be an issue, consider the data for Sally and Sandy shown on the right.

	Air Miles	Reality TV
Sally	25000	2
Sandy	50000	1

Relatively speaking both features differ by the same amount (one is half the value of the other). But in the distance calculation, Air Miles makes a much bigger contribution.

$$d = \sqrt{(50000 - 25000)^2 + (1 - 2)^2} = \sqrt{625\,000\,000 + 1} = \sqrt{625\,000\,001}$$
$$= 25000.00002$$

In fact, no amount of difference in TV viewing is likely to make very much difference here. If Sandy also watched 2 hours per week, the distance between them would drop to 25000, which is a difference of less than 0.00000001%. The solution is to **normalize** your data before you use it.

How to Normalize

Normalizing the data means squeezing every feature down into the range 0 to 1. For each feature, you find the minimum (**min**) value in the training set and the range (**r**), then replace each value (**v**) with the normalized value $\mathbf{v}_{\mathbf{norm}}$=(**v**-**min**)/**r**. This can be done for your entire data set in a single line if you are using Numpy effectively.

The table at right shows the data for Sally and Sandy after normalizing, assuming that the range of air miles is 0 to 50000 and the range of ice cream is 0 to 2.

	Air Miles	Reality TV
Sally	0.5	1
Sandy	1	0.5

Now the contribution to the distance calculation is the same for both features.

$$d = \sqrt{(1 - 0.5)^2 + (0.5 - 1)^2} = \sqrt{0.25 + 0.25} = \sqrt{0.5} \cong 0.707$$

In this case, if Sally ate 2 litres per week, the distance between them would drop to 0.5, which is a difference of 30%. In the normalized data, TV viewing matters!

Weighted Voting

It's usually a good idea to experiment with different values for *k*. But some values of *k* might lead to tie votes. If every vote has the same weight, you have no choice but to choose randomly among the winners. One way to avoid ties is to weight each vote using the inverse of the distance for each point. For example, a neighbour that is 2 units away from a new data point only gets a vote worth 0.5 (because 1/2 is the inverse of 2).

Weighted voting can also help when you have an unbalanced data set. For example, suppose you have a training set consisting of all news stories published in 2016 and you want to determine which news stories were about earthquakes. The ratio of non-earthquake to earthquake stories is going to be high. Maybe it's 10000 to 1. This is going to be like trying to find a needle in a haystack. For k-Nearest Neighbour, there will be so few earthquake stories, the nearest 3, 5, or 7 might always be majority non-earthquake. Weighted voting can help with this by making the closest examples worth a lot more.

Distance Calculations

Euclidean Distance is not the only way to measure how far apart two items are in multidimensional space. Another choice is **Manhattan Distance**. This is a measure of distance that is easier to compute and sometimes leads to better performance. Both Euclidean Distance and Manhattan Distances can also be seen as special cases of a family of measurements known as the **Minkowski Distance**.

Manhattan Distance

In a city with a grid layout (like Manhattan), the distance between two points is not a straight line. To get from the corner of 5th and 7th to the corner of 10th and 1st, you have to travel 5 blocks in one direction and 6 blocks in another for a total of 11 blocks. So the total distance is just the sum of the difference in each direction.

The Manhattan Distance between two points is the absolute value of the sum of the difference across all feature values. Suppose you have *n* features, your training example is (t_1, t_2, t_3, …, t_n), and your new data point is (p_1, p_2, p_3, …, p_n). Then the Manhattan Distance is:

$d_{manhattan} = \sum_{i=1}^{n} |t_i - p_i|$ ← Note that |x| denotes the absolute value of x.

Minkowski Distance

The Minkowski Distance is a generalization of Euclidean Distance. Pick an integer *p* to use as an exponent. Suppose if you have *n* features, your training example is (t_1, t_2, t_3, …, t_n), and your new data point is (e_1, e_2, e_3, …, e_n). Then the Minkowski Distance is:

$d_{minkowski} = (\sum_{i=1}^{n} |t_i - e_i|^p)^{\frac{1}{p}}$

Recall that when you raise a number to the exponent 1/2, you are taking the square root.

When p = 2, the Minkowski Distance is Euclidean. When p = 1, it's Manhattan.

Exercises

1. Go to knn_example.py and modify the code so that it loads the wine dataset and then runs the algorithm 10 times with a different testing and training split each time. Report the average accuracy over 10 runs.

2. Experiment with different combinations of weighting, p values, and normalizing vs. not normalizing on the wine dataset. Which combination gives you the highest accuracy? Which of these parameters has the biggest impact?

3. SKLearn also has a RadiusNeighborsClassifier. Read the sklearn documentation to see what it does. Then change your classifier so that you are using this one. How well does it do compared to kNN?