Non-parametric Regression

Global Fit vs. Local Fix
The techniques we have looked at so far fit a single function to all the data – they create a global fit. However, sometimes data relationships may be different in different regions of the data. In these cases, fitting locally provides better predictions.

1 Nearest Neighbor Regression (1NN)

This is a very simple algorithm that works well when there is lots of data.

- Simply choose the observation in the data that is closest to the input value as the predicted value.

Given a dataset of pairs; $(x_1, y_1), (x_2, y_2), \ldots\ (x_N, y_N)$ and a query parameter x_q;

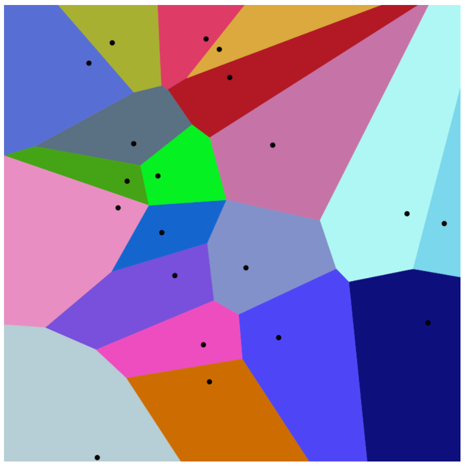
1. Find the x_i in the data set that is closest to x_q.

$$x_{NN} \leftarrow \min_i distance(x_i, x_q)$$

2. Predict $y_q = y_{NN}$

The distance metric is a key to making this work well.

This can be visualized as a Voronoi Tesselation where the (x_i,y_i) space is broken into areas, each with one (x_i,y_i) point and where the area around the point is includes all points that are nearest to it (and so not nearer to any other point).



Example Voronoi Diagram using Euclidean Distance from Wikipedia

Distance Metrics

1D Euclidian Distance is simple;

$$distance(x_i, x_q) = |x_i - x_q|$$

Euclidean Distance in Multiple Dimensions

In multiple dimensions $\vec{x}_i$ and $\vec{x}_q$ are feature vectors of length D. The Euclidean distance is the square root of the sum of the squared differences in each dimension.

$$euclidean\ distance(\vec{x}_i, \vec{x}_q) = \sqrt{(\vec{x}_i[1] - \vec{x}_q[1])^2 + \cdots + (\vec{x}_i[d] - \vec{x}_q[d])^2}$$

$\vec{x}_i$	is the vector of D features for the ith observation in the dataset.
$\vec{x}_i[1]$	is the first feature value (first column) in the ith row of N observations
$\vec{x}_i[d]$	is the last feature value (last column) in the ith row of N observations
$\vec{x}_q$	is the vector of D features for the query.
$\vec{x}_q[1]$	is the first feature value (first column) in query vector.
$\vec{x}_q[d]$	is the last feature value (last column) in query vector.

Scaled Euclidean Distance

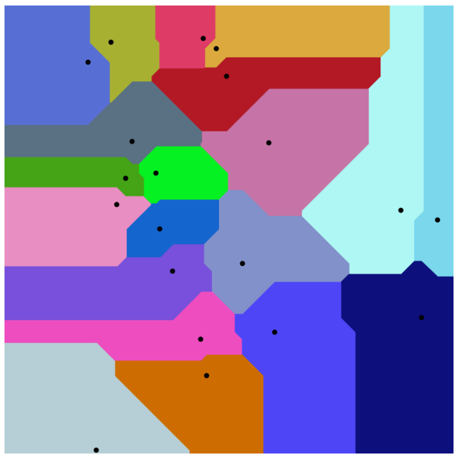
Here we add a scaling factor for each feature's distance as a vector $\vec{a}$ of length D, the number of features. This allows us to weigh important features more in the distance metric;

$$scaled\ euclidean\ distance(\vec{x}_i, \vec{x}_q) = \sqrt{\vec{a}_1(\vec{x}_i[1] - \vec{x}_q[1])^2 + \cdots + \vec{a}_d(\vec{x}_i[d] - \vec{x}_q[d])^2}$$

There are many of other distance metrics

- Manhattan Distance
- Hamming Distance
- Cosine Similarity
- Mahalonobis distance (standard deviations from the mean)
- Rank based distance
- Correlation based distance

Here is a Voronoi Diagram using the same data as the one above, but using Manhattan distance as the distance metric.


Example Voronoi Diagram using Manhattan Distance from Wikipedia

```
1NN Algorithm

1. Query is $x_q$
2. Initialize distance to nearest neighbor $\delta_{NN}$ to infinity
   and the nearest neighbor to null.
   - $\delta_{NN} = \infty$
   - NN = null
3. For i in 1..N
   - compute $\delta = distance(x_i, x_q)$
   - if $\delta < \delta_{NN}$
     - $\delta_{NN}$
     - $y_{NN} = y_i$
4. return $(x_q, y_{NN})$
```

1NN has limitations.

- It requires dense data
  In places where data is sparse, predictions are not as good.
- It is sensitive to noise in the data
  If the observed data is noisy, the resulting predictions are very noisy (they resemble an over-fitted function).

k-NN, K Nearest Neighbors Regression

This is a simple extension of 1NN. Rather than find a single closest observation, if finds the closest K neighbors to the query and use their average as the prediction.

Given a dataset of pairs; $(x_1, y_1), (x_2, y_2), \ldots\ (x_N, y_N)$ and a query parameter x_q;

1. find the k nearest neighbors to x_q, $(x_{NN_1}, y_{NN_1}), (x_{NN_2}, y_{NN_2}), \dots\ (x_{NN_k}, y_{NN_k})$ where the distance to the kth nearest neighbor, $distance(x_{NN_k}, x_q)$, is the smallest distance in the set, such that any x_i that is not in the dataset has $distance(x_i, x_q) \geq distance(x_{NN_k}, x_q)$
2. Predict the average $\hat{y}_q = \frac{1}{k}(y_{NN_1} + y_{NN_2} + \cdots + y_{NN_k}) = \frac{1}{k}\sum_{j=1}^{k} y_{NN_j}$

```
k-NN Algorithm

1. Query is $x_q$, k is the number of nearest neighbors.
2. Initialize a sorted queue of length k with first k
   neighbors sorted by increasing distance from $x_q$ (so the
   last one in the list is the farthest from $x_q$).
   - for j in 1..k insert $x_j$ into $NN_j$
     ordered by increasing distance
3. Initialize the nearest neighbor distance (the distance
   that all k nearest neighbors are within from the query)
   to the distance between the query and the last neighbor
   in the ordered nearest neighbors list.
   - $\delta_{NN} = distance(NN_k, x_q)$
3. For i in k+1..N
   - compute $\delta = distance(x_i, x_q)$
   - if $\delta < \delta_{NN_k}$              # point ($x_i$,$y_i$) is closer than
                                 # farthest nearest neighbor
     - remove $NN_k$ from the sorted queue
     - insert ($x_i$,$y_i$) into queue, ordered by $\delta$.
4. return $\hat{y}_q = \frac{1}{k}\sum_{j=1}^{k} y_{NN_j}$         # return mean distance of
                                 # nearest k neighbors
```

Results

- This tends to smooth out issues in noisy data. There are still discontinuities, but not radical swings that seem like over-fitting like we had with noisy data and the 1-NN algorithm.
- There are boundary effects at either end of our data where we are with k data points of either end of the data. In these two regions, we get a constant fit (based on the k points at the end of the data).
- There are discontinuities as points move in our out of the k window. This create fits where a very small change in x can results in a large change in y, which seem unrealistic for many applications. For instance, a 1 square foot change in floor space could result in unexpectedly large change in predicted price, which would not make sense.

Weighted k Nearest Neighbors

Neighbors are weighted based on their closeness to the query point. If a point is very close, it is similar to the query, so we want that to have more influence on the prediction. If a point is farther away, it is less similar, so we want it to have less influence on the prediction.

So generally, we can multiply each point by a weight factor that depends upon their distance to the query, add all the resulting terms and divide this sum by the total of the weights involved to get our prediction;

$$\hat{y}_q = \frac{c_{qNN1} + c_{qNN2} + \cdots + c_{qNNk}}{\sum_{j=1}^{k} c_{qNNj}}$$

So we want the we want the weight to be larger is the distance is smaller and we want the weight to be smaller if the distance is larger. So would could use 1/distance(x_i,x_q) as the weight to get this effect.

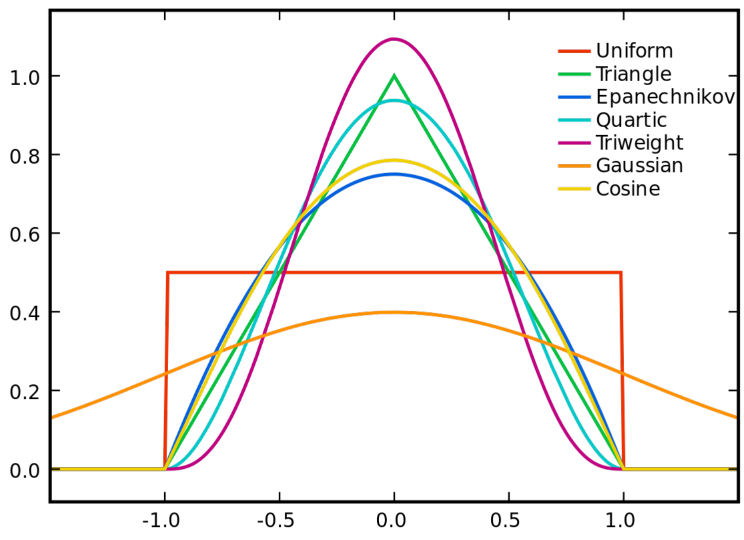
$$c_{qNNj} = \frac{1}{distance(x_i, x_q)}$$

Kernels

More generally, we can use an isotropic Kernel that is a function of the distance between a point and the query point.

$$c_{qNNj} = Kernel_{\lambda}\left(\left|x_{NN_j} - x_q\right|\right)$$

This Wikipedia article discussed Kernels generally. Here is an illustration from the article of various isotropic Kernels.



All of these kernel have a **bandwidth parameter** λ that determines how quickly the function decays towards zero. In the illustration above, λ ranges from -1 to 1. Within the range of $-\lambda..\lambda$, weights are greater than zero; they decay to zero outside this range, except in the case of the Gaussian Kernel, the weights only approach zero and don't go to zero.

$$Gaussian\ Kernel_\lambda\left(\left|x_{NN_j} - x_q\right|\right) = e^{-\left(x_{NN_j}-x_q\right)^2/\lambda}$$

With the other isotropic Kernels, outside of the range $-\lambda..\lambda$, the weights drop to zero; these are called bounded kernels.

Kernel Regression (Nadaraya-Watson Kernel Weighted Average)
This is related to k Nearest Neighbors when using a Kernel. In Kernel regression, we apply a weight to every data point in the data set.

$$\hat{y}_q = \frac{\sum_{i=1}^{N} c_{q_i} y_i}{\sum_{i=1}^{N} c_{q_i}} = \frac{\sum_{i=1}^{N} distance(x_i, x_q) \times y_i}{\sum_{i=1}^{N} distance(x_i, x_q)}$$

Overall this provides a smoother result than k-Nearest Neighbors regression.

Typically the choice of kernel has less effect than the choice of the tuning parameter λ. If the tuning parameter is too small, we can see effects from over-fitting (high variance). If the tuning parameter is too large, we see over-smoothing and higher bias.

Choosing the bandwidth Parameter λ
This is the same as we have done before. We can use a validation set to validate our choice of λ, or we could use Cross Validation if we don't have enough data for a validation set.

Local Fits

The simplest global model for a fit is the constant model. In particular, we take all points and average them and use that average as our prediction. This can be reformulated into a form more like kernel regression;

$$\hat{y}_q = \frac{1}{N}\sum_{i=1}^{N} y_i = \frac{\sum_{i=1}^{N} c y_i}{\sum_{i=1}^{N} c}$$

c is some constant weight that is the same for all points. This is like the boxcar kernel where the bandwidth is the entire input space.

If effect, a Kernel Regression calculates a constant fit in a local region of the data. The region is based on the query point and the choice of the bandwidth parameter. This is known a locally weight averages.

Locally Weighted Linear Regression

Rather than fit a constant, we can fit a line or a polynomial locally.

Relative to a local constant fit

- local linear fit reduces bias at the boundaries with only a small increase in variance
- local quadratic fit helps capture curvature in the interior, but increases variance and does not help at the boundaries
- with sufficient data, local polynomials of odd degree dominate those of even degree

Generally a local linear fit should be the default choice.

Non-parametric Approaches

- Flexible - Lots of choices for what is fitted and how much data is used to fit
    - Constant fit; 1NN, kNN, Kernel Regression
    - Locally Weighted Linear Regression and other locally weighted polynomials

- Other things like splines, trees, locally weighted structural regression models

- Makes very few assumptions about f(x)
- **Complexity of the fit can grow with the number of observations N**

Limiting Behavior with increasing the number of observations

Noiseless data

- In the limit of an infinite amount of noiseless data, the Mean Squared Error MSE for 1NN goes to zero. Because MSE is the sum of bias^2 + variance, this means that both bias and variance go to zero.
- This is NOT true for a parametric model – there will always be some bias even with infinite data because a parametric model with a finite number of coefficients will be limited to some inherent shape and so will not be able to perfectly model arbitrary data. For example, a quadratic has an inherent shape that cannot fit all sets of arbitrary data, even if the data is noiseless.

Noisy data

- With noisy data, 1NN will have wild variations; it is over-fit, so it has very high variance. This is true even as the amount of data approaches infinite data (if that data is noisy).
- In the limit of an infinite amount of noisy data, the MSE for kNN goes to zero IF k is large enough.
- A parametric fit on noisy data will still have bias even as the amount of data approaches infinity.

So NN and Kernel Regression work very well with large data sets. However, things are not so good when D (number of dimensions) is very large or N (number of observations) is small.

- NN and Kernel Regression as sensitive to the amount of data available. In particular, if the data is sparse in a locale then local over-fitting happens.
- This can happen if N is small.
- This can happen if D is very large because the amount of data required for good coverage increases exponentially with the number of dimensions. So for good coverage N = O(exp(d)) data points.

When N is small relative to D, so sparseness is a problem, then parametric models become useful.

Complexity of NN search

When using a naïve brute force search

- given one query point x_q

- We must scan through all N observations and calculate the distance(x_i,x_q) in order to find a nearest neighbor.
- For 1NN, this is O(N) complex.
- for kNN, this is O(N*log(k)) complex (assuming we use an ordered queue to hold the k neighbors).

Remember that NN Regression works best if there are lots of data points, so when N is huge the actual predictions are good, but the computational costs are extremely high.

We will learn how to mitigate this complexity in the Clustering and Retrieval course.

kNN for Classification

For instance, classifying email as spam or not spam. If we have a set of labeled emails (spam or not spam), and we have some sort of distance measure (or it's inverse, a similarity measure) for emails, then given an unlabeled email, we can find it's k nearest neighbors by applying the distance/similarity metric to the query email and the labeled emails. Once we have the k nearest (most similar) neighbors, we can vote based on the neighbor's labeling. Perhaps this is a majority vote, but other voting schemes could be used.