CS4619: Artificial Intelligence 2

Nearest-Neighbours for Regression

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]:
%load_ext autoreload
%autoreload 2
%matplotlib inline

In [2]:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

k-Nearest Neighbours for Regression

Linear Regression is an example of **parametric learning**: it ‘chooses’ from a **restricted** set of hypotheses (models) by finding parameters that give a good fit to an unrestricted training set. This is often effective. But we might suspect that we could learn a better model if we had an unrestricted set of hypotheses.

In this lecture, we look at **nearest-neighbour models** for regression. These give us simple examples of **nonparametric learning**, where there is no restriction on the models: the more complicated the data, the more complicated the hypotheses.

The key assumption of nearest-neighbour models for regression is that \( x \)'s value for \( y \) will be similar to \( x \)'s neighbours' values for \( y \). If this assumption holds, then to predict \( x \)'s value for \( y \), we find \( x \)'s neighbours and base our prediction on their value for \( y \).

How many neighbours should we use? The simplest method is to fix a value, designated \( k \). This means that \( x \)'s value for \( y \) is obtained from the \( y \)-values of the \( k \) nearest neighbours of \( x \). Hence, these algorithms are often referred to as **k-nearest-neighbours** methods, abbreviated to **kNN**. We will look first at the case where \( k = 1 \), and then at the case where \( k > 1 \). But first we consider how to measure the distance between two examples.

Distance
Let \( \mathbf{x} \) be one vector of feature values and \( \mathbf{x}' \) be another. Simplest is to measure their Euclidean distance:

\[
d(\mathbf{x}, \mathbf{x'}) = \sqrt{(x_1 - x'_1)^2 + (x_2 - x'_2)^2 + \ldots + (x_n - x'_n)^2}
\]

or, more concisely:

\[
d(\mathbf{x}, \mathbf{x'}) = \sqrt{\sum_{j=1}^{n} (x_j - x'_j)^2}
\]

It has a nice vectorized implementation in numpy Python:

```python
In [3]: def dist(x, xprime):
   ...:     return np.sqrt(np.sum((x - xprime)**2))
   ...

# Example
my_house = np.array([107, 3, 2])
your_house = np.array([120, 3, 1])
dist(my_house, your_house)
```

```
Out[3]: 13.038404810405298
```

**Thoughts About Measuring Distances**

Questions:

- (Easy) For finding nearest neighbors, we don't really need to take the square root. Why not?
- (Difficult) Does the example bring to mind a problem with using definitions of distance like this one on features like these?

In passing, let's mention that there are other ways to define distance. In the scikit-learn documentation, you might see Minkowski distance being mentioned. It is a generalization of Euclidean distance:

\[
d(\mathbf{x}, \mathbf{x'}) = (\sum_{j=1}^{n} |x_j - x'_j|^p)^{1/p}
\]

When \( p = 1 \), we get Manhattan distance; when \( p = 2 \), we get Euclidean distance.

**1NN**

Suppose we have a training set \( \mathbf{X} \). We want to predict \( \hat{y} \) for feature vector \( \mathbf{x} \). For each \( \mathbf{x}^{(i)} \in \mathbf{X} \), we compute \( d(\mathbf{x}, \mathbf{x}^{(i)}) \). We find the \( i \) such that \( \mathbf{x}^{(i)} \in \mathbf{X} \) has the smallest distance to \( \mathbf{x} \). Then, our prediction is \( \mathbf{y}^{(i)} \).

A minor complication is what to do when there is a tie, i.e. two or more examples share the same smallest distance to \( \mathbf{x} \). In this case, we might just take the first one that we found (which is what the Python does, below) or we might choose one of them at random.

**My Python Implementation of 1NN**

First, let's read in the CorkA property dataset:
Use pandas to read the CSV file

def = pd.read_csv("dataset-corkA.csv")

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df[['flarea', 'bdrms', 'bthrms']].values
y = df['price'].values

Here's a function that implements 1NN:

```python
In [5]: def NN_predict(x):
    ...:     dists = [dist(x, xi) for xi in X]
    ...:     index = np.argmin(dists)
    ...:     return y[index]
```

Let's make a 1NN prediction for \[
\begin{bmatrix}
114 \\
3 \\
2
\end{bmatrix}
\] . It turns out that its nearest neighbour in the CorkA property dataset is \[
\begin{bmatrix}
115.2 \\
4 \\
2
\end{bmatrix}
\] , so we predict a selling price of 385:

```python
In [6]: # Example prediction
    ...: NN_predict(np.array([[114, 3, 2]]))
```

```
Out[6]: 385
```

**scikit-learn Implementation of 1NN**

As usual, we don't really need to write it for ourselves:

```python
In [7]: from sklearn.neighbors import KNeighborsRegressor
    ...: estimator = KNeighborsRegressor(n_neighbors = 1)
    ...: estimator.fit(X, y)
    ...: # Example prediction
    ...: estimator.predict([[114, 3, 2]])
```

```
Out[7]: array([ 385.])
```

Questions:

- What do you think the fit method is doing? Or, to put this another way, where is all the work taking place?
- You can change which distance function it uses. The defaults are p=2, metric='minkowski', i.e. Euclidean. How would you edit the code above to use Manhattan distance?
A problem arises with 1NN if the training set contains noise. If the values of the features or dependent variable are incorrect, they will lead to incorrect predictions. This motivates use of $k > 1$.

In kNN in general, we find the $k$ examples in the training set whose distance from $x$ is smallest and we base our prediction on their $y$-values. Typical values for $k$ might be 3, 5, 10 or 20, although there may be scenarios where much larger values (100, say) would be used. With $k > 1$, the impact of noisy examples in the training set is reduced. If $k$ is 'too small', noisy examples may still have too great an impact. If $k$ is 'too big', you are no longer exploiting any structure in the training data (although we will see below that distance-weighting can help reduce this problem).

**Parameters versus Hyperparameters**

$k$ is a kind of parameter, like $\beta$ in linear regression. But there's a big difference: it's the job of your linear regression learning algorithm to come up with the value of $\beta$, whereas the value of $k$ is not set by the kNN algorithm — its value is fed into the algorithm. Let's make this distinction:

- **Parameters**: variables whose values are set by the learning algorithm — such as $\beta$ in linear regression.
- **Hyperparameters**: variables whose values are not set by the learning algorithm, rather they are supplied to the learning algorithm — such as $k$ in kNN.

**Question:**

- Give an example of a hyperparameter that we saw when studying linear regression.

If the learning algorithm does not set them, how will we choose the values for hyperparameters? Specifically, how do we choose the value of $k$?

Often, we'll just make a guess — hopefully, an informed guess. E.g. we know that a value for $k$ between 3 and 20 has often worked well on other datasets. In an up-coming lecture, we will re-visit this topic and see that there are also automatic methods that can help us to choose the values of hyperparameters.

**Unweighted kNN**

The simplest approach is to report the mean of the neighbours' $y$-values:

$$\hat{y} = \frac{\sum_{x^{(i)} \in NN} y^{(i)}}{k}$$

where $NN$ is the set of $x$'s $k$ nearest-neighbours.

As before, if there are ties for $k$th place, this code just takes the ones that the sorting places first in the ordering:
In [8]: ```python
def kNN_predict(x, k):
    dists = [dist(x, xi) for xi in X]
    indexes = np.argsort(dists)[:k]
    return np.mean(y[indexes])
```

If we use 3NN to predict for \( \mathbf{x} = \begin{bmatrix} 114 \\ 3 \\ 2 \end{bmatrix} \) again, we find the three nearest neighbors are
\[
\begin{bmatrix}
115.2 \\
4 \\
2
\end{bmatrix},
\begin{bmatrix}
112.4 \\
3 \\
2
\end{bmatrix},
\begin{bmatrix}
112.4 \\
3 \\
2
\end{bmatrix}
\]
and so we predict the mean,
\[
\hat{y} = \frac{385 + 225 + 225}{3} = 278.33:
\]

In [9]: ```python
# Example prediction
kNN_predict(np.array([[114, 3, 2]]), k = 3)
```

Out[9]: 278.33333333333331

A better implementation would avoid the sorting, which could become expensive for a large training set. Instead, it would insert the distances into a priority-ordered queue whose length was constrained to be \( k \).

But the best thing is to use scikit-learn:

In [10]: ```python
estimator = KNeighborsRegressor(n_neighbors = 3)
estimator.fit(X, y)
# Example prediction
estimator.predict([[114, 3, 2]])
```

Out[10]: array([ 278.33333333])

**Distance-weighted kNN**
In unweighted kNN, each neighbour contributes equally to the prediction. An obvious variant is to give greater weight to closer neighbours: the closer you are, the more you contribute to the computation of the mean. We need a weighted mean:

\[
\hat{y} = \frac{\sum_{x^{(i)} \in NN}(w^{(i)} \times y^{(i)})}{\sum_{x^{(i)} \in NN} w^{(i)}}
\]

The weights \( w^{(i)} \) should be inversely proportional to the distances, so we could use the following:

\[
w^{(i)} = \frac{1}{dist(x, x^{(i)})}
\]

Let's use distance-weighted 3NN to predict for \( x = \begin{bmatrix} 114 \\ 3 \\ 2 \end{bmatrix} \). We get the same neighbours as above. But we need their distances from \( x \), which, to two decimal places, are 1.56, 1.6 and 1.6, respectively. The weights are \( 1/1.56 = 0.641, 1/1.6 = 0.625 \) and \( 1/1.6 = 0.625 \). And the prediction is the weighted average:

\[
\hat{y} = \frac{0.641 \times 385 + 0.625 \times 225 + 0.625 \times 225}{0.641 + 0.625 + 0.625} = 279.2
\]

We can easily do this in scikit-learn by specifying weights='distance':

```
In [11]: estimator = KNeighborsRegressor(n_neighbors=3, weights='distance')
estimator.fit(X, y)
# Example prediction
estimator.predict([[[114, 3, 2]]])
```

```
Out[11]: array([[ 279.19021761]])
```

### Fixing a Problem

We defined the weights \( w^{(i)} \) to be inversely proportional to the distances:

\[
w^{(i)} = \frac{1}{dist(x, x^{(i)})}
\]

But this gives the possibility of division by zero: it will arise if there is ever an example in the training set whose distance from \( x \) is zero (presumably, exact matches). One solution to this is to adjust the denominator:

\[
w^{(i)} = \frac{1}{0.001 + dist(x, x^{(i)})}
\]

Somewhat unbelievably, scikit-learn has not dealt gracefully with this problem:

```
In [12]: # Example prediction
estimator.predict([[[101.9, 3, 3]]])
```

```
Out[12]: array([ nan])
```

Instead, we can supply our own function that takes in an array of distances and returns an array of weights. My inv_distances function does this in a vectorized way:
In [13]:
def inv_distances(dists):
    return 1 / (0.0001 + dists)

estimator = KNeighborsRegressor(n_neighbors = 3, weights = inv_distances)
estimator.fit(X, y)

# Example prediction
estimator.predict([[114, 3, 2], [101.9, 3, 3]])

Out[13]: array([ 279.1901632,  180.04828505])

Shepard's Method

If you're using distance-weighting then, instead of basing the prediction \( \hat{y} \) on just the \( k \) nearest neighbours of \( \mathbf{x} \), you could base it on all the examples in the training set. In other words, use \( k = m \). Distant examples will have little effect on the prediction. The disadvantage is that it will run more slowly.

Let's try it:

In [14]:
m, n = X.shape

estimator = KNeighborsRegressor(n_neighbors = m, weights = inv_distances)
estimator.fit(X, y)

# Example prediction
estimator.predict([[114, 3, 2], [101.9, 3, 3]])

Out[14]: array([ 249.12766093,  180.12020134])

Advantages of Nearest-Neighbours Methods

- Instead of fitting a global model to the training set, nearest neighbour methods estimate the target function in a local fashion: they construct a different approximation to the target function for each \( \mathbf{x} \) for which a prediction is sought. This is advantageous when the target function is complex. Other methods construct a single hypothesis that has to perform well across the whole space of examples. This makes kNN (especially with \( k > 1 \) and distance weighted) a very effective prediction method for many practical problems.
- Most learning algorithms only work with features whose values are numeric. kNN can work on features whose values are not numeric — provided a suitable distance function can be defined.
- It is easy for kNN to incrementally incorporate new training examples: if new training examples become available, you simply add them to the training set — that's it! By contrast, many other learning algorithms would have to re-compute their model from scratch.

Disadvantages of Nearest-Neighbours Methods
- The cost of kNN prediction can be high.
- KNN methods often suffer from the **curse of dimensionality**.

We look at both these disadvantages — and ways of overcoming them — in the next two subsections.

### The Cost of kNN Prediction

In kNN prediction, pretty much all the work takes place on-demand, at prediction time. In the way we have presented it so far, every time it makes a prediction, the kNN algorithm must compute the distance between \( x \) and *every* \( x^{(i)} \) in the training set. For a time, it was thought that this confined kNN methods to datasets with few training examples (small \( m \)).

But there are solutions that can allow kNN to be used more widely: to use clever data structures that make it possible to find the nearest neighbours without computing the distance to every training example.

One such data structure is the *kd*-tree. The training examples are stored in the leaves of a tree. The storage policy ensures that examples that are close to each other (according to the distance function) are stored either at the same leaf or at nearby leaves. Interior nodes of the tree contain tests that examine the features and, based on the outcome, direct you to the appropriate subtree. Given \( x \), walking from root to leaves, as directed by the tests, takes you to neighbours in time that is logarithmic in the number of examples. (It is not necessarily so cheap and easy to find the \( k \) nearest neighbours: you might have to 'backup' the tree to visit other subtrees.) *kd*-trees work best, though, for low dimensional data (small \( n \)).

Another such data structure is the ball tree, which is based on constructing nested hyper-spheres. Building a ball tree is more costly than building a *kd*-tree. But ball trees find neighbours quickly, even for high dimensional data.

By default, scikit-learn will choose the data structure for you. But if you prefer, you can specify it explicitly using the algorithm parameter, whose values are 'auto' (default), 'ball_tree', 'kd_tree' or 'brute', e.g.:

```
In [15]: estimator = KNeighborsRegressor(n_neighbors = 1, algorithm = 'kd_tree')
estimator.fit(X, y)

# Example prediction
estimator.predict([[114, 3, 2]])
```

```
Out[15]: array([ 385.])
```

For more details, see the on-line documentation ([http://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbor-algorithms](http://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbor-algorithms)).

Another way to reduce the cost of nearest neighbour methods is to edit the training set, i.e. to delete examples so that \( m \) is smaller. Obviously, the key is in choosing the right examples to delete: ones which do little damage to prediction accuracy. We will not look at these ideas any further in this module.

### The Curse of Dimensionality
Typically, the distance measure is computed across all the features and (other things being equal) treats them as equally important. But some may be less important than others — or even irrelevant. (By contrast, in some sense, methods such as linear regression discover the relative importances of the features.) Using these features in the distance calculation can have negative effects.

Consider a problem in which there are 30 features but only two are relevant. Two examples might be identical on the two relevant features. But their distance, when all 30 features are used in the calculation, might be very high. The neighbours found by such a distance function will then not be ones that are truly predictive of the dependent variable, resulting in poor predictions.

The more features there are, the more likely it is that situations like this will arise. This is the so-called curse of dimensionality (where dimensionality refers to \( n \), the number of features). For a time, it was thought that this confined kNN methods to datasets with few features (small \( n \)), preferably where all of them are of roughly equal importance.

But there is a solution that can allow kNN to be used more widely: to use feature weights in the distance function. These weights capture the relative importances of each feature (or even, when zero, they select which features are irrelevant):

\[
d(x, x') = \sqrt{\frac{\sum_{j=1}^{n} w_j \times (x_j - x'_j)^2}{\sum_{j=1}^{n} w_j}}
\]

Be clear: these feature weights \((w_j)\) are quite different from the distance weights \((w^{(i)})\) that we used earlier.

Where will the feature weights come from? It’s possible that your domain expert can supply them up-front. But, in machine learning, we always tend to prefer to use the data to help us to find them. They are further examples of hyperparameters and so they can be set in the same way as \( k \) (see the up-coming lecture).