ME4 Machine Learning - Tutorial 7

In Lecture 7 we studied non-parametric techniques, including estimating probability through Parzen windows and performing nearest neighbour classification. In this tutorial you will use Python and Scikit-learn to put this into practice, estimating density for some distributions and using this to perform some classification. This should reinforce some of the concepts covered in the lecture.

1 Density estimation with Parzen windows

Use a normal distribution to define a set 40 of samples, $\mu = 0$, $\sigma = 1$, in a column (note that to make a column like this the shape should be [40,1]). Ideally look up how to do this yourself (it should only be a single line). The correct answer is at the end of the sheet if you are struggling. We will then try to estimate the distribution from these samples, using Parzen windows. This requires the KernelDensity function in scikit-learn:

```
from sklearn.neighbors import KernelDensity
...
kde = KernelDensity(kernel='tophat', bandwidth=1).fit(X)
```

where X is the array containing the samples. We have set the kernel to the top hat kernel, and the bandwidth can be adjusted to different values. Having set up the kernel density estimator, we can pull out the function at a set of test points in x and convert into probability p:

```
x_sample = np.linspace(-5, 5, 1000)
p = np.exp(kde.score_samples(x_sample.reshape(len(x_sample), -1)))
```

Note that the score_samples() function requires the x samples to be in a 2D array, hence the reshape(), and it outputs log probability, so we need to use the np.exp() function to convert to probability.

Plot the probability as a function of x, for bandwidths of 0.2, 1.0 and 4.0. Try some others and see how they look. Which do you think is the best? Which would you select? Look up the equation for the underlying normal distribution function (you did this in tutorial 2, so feel free to use code from that) and plot that on the same axes to see how these compare.

Test out a Gaussian function instead (kernel='gaussian'). Try the three different bandwidths to confirm the behaviour. Do you think this is always better than the top hat (square) approach?

2 Window plotting

You should be able to modify your code to plot just the underlying window which was used, centred at x=0. Do this for the 'exponential' window, bandwidth=0.5. [Hint: as shown in the lecture, the final probability distribution is a superposition of many windows together, coming from each point in the distribution. To just get one window, you can just do one point.] Code is given at the bottom if you get really stuck, but try yourself first.

3 2D function classification

Generate the circular distribution with n = 200 – you will need to utilise the same function from previous tutorials. We will now attempt to perform classification based on the probability distributions which are produced for the two classes. Split X into two parts, corresponding to each class:

```
X, y = gen_circular_distribution(200)
X1 = X[y == 0, :]
X2 = X[y == 1, :]
```

Use KernelDensity() to fit both of these with a Gaussian window of bandwidth 1.

Define an image grid of 200 × 200 from -10 to 10 in each dimension (as we have done many times through the course), sample the probability for each class across this grid, and generate an image of the class by selecting the class which has the highest probability. [Note that this probability is the likelihood, not the posterior, but it is sufficient for our illustrations.] What do you think will happen as you reduce the bandwidth? Try this and see.

4 Nearest neighbour classification

Keeping the same parameters as for part 3 above, we now want to use the KNeighboursClassifier.

```
from sklearn.neighbors import KNeighborsClassifier
...

near = 1
neigh = KNeighborsClassifier(n_neighbors=near)
neigh.fit(X, y)
```

As with other Scikit-learn classifiers, the predict() function will produce the classification - apply this to a grid to generate an image of the classification regions. Do a scatter plot of the data on top to help you visualise it. How can you adjust your code to reduce the variance you are likely to see?

5 Estimate density manually (Optional)

So far we have used a lot of standard functions from scikit-learn. It is possible to estimate probability density manually by dividing the domain up into large grid squares and counting how many points we get in each. This is something which will be good programming practice if you want to get into Python a bit more, but should not be seen as an essential part of the course.

Define a distribution - use gen_circular_distribution() as before, with $n = 2000$. Define a grid of 20 × 20 squares from -10 to 10 in each dimension (note - just use linspace for this and do not worry about the boundary square extending outside the range). Estimate the probability distribution across each of these squares (hint: it is inefficient to loop through every square and then check which points lie within - find another way). Halve the size of the squares and have 40×40. View the result. Double the size and have 10×10. Based on the 20×20 squares, test what happens when you half and double n.

6 Reference code

Part 1:

```
X = np.random.normal(loc=0, scale=1, size=[40, 1])
```

NB this assumes you've imported numpy as np.

Part 2:

```
#Fit with a single point:
kde = KernelDensity(kernel='exponential', bandwidth=0.5).fit(np.zeros([1, 1]))
#sample the function (assuming x_sample has been generated
# with linspace to sweep through the x values to be evaluated)
p_sing = np.exp(kde.score_samples(x_sample.reshape(len(x_sample), -1)))
```