ME4 Machine Learning - Tutorial 5

Lecture 5 covered some of the theory behind Support Vector Machines (SVMs). In this tutorial you will use Python and Scikit-learn to perform classification via SVMs to get practical experience of the theory covered in the lecture. You will generate some test datasets, and you will also be able to explore the different kernel types discussed in the lecture to see what effect these have on the classification performance.

1 Set up the environment

We will be using some functions for this tutorial, some of which you may have developed earlier in the course. There are two options for accessing these here. If using Colab, you should start from the interactive notebook [tutorial5.ipynb] given - import this into Colab. If using a more traditional IDE, such as PyCharm, use the file [tools.py] provided, put it in your working directory and add the following line to your script

```
from tools import *
```

This will import the functions from tools.py.

2 XOR dataset

We will try to perform classification on an XOR dataset (as shown in the lecture). This is a classic example which cannot be classified by a simple linear discriminator. Generate a dataset using the 'gen_xor_distribution()' function in tools.py with a parameter of 400. Do a scatter plot to check that it looks sensible. It would be a good idea to make sure that this is producing the same dataset each time it is run, so you may wish to add 'np.random.seed(0)' before calling the function.

Import the necessary support vector machine library functions:

```
from sklearn.svm import SVC
```

then you can define a support vector machine using SVC:

```
svm = SVC(C=0.01, gamma='auto', kernel='poly', degree=2)
```

Note that here we're using polynomial basis functions, degree 2 (matching what I did in the lecture). Use svm.fit() to fit the training XOR dataset to this data (note - look up the syntax in the online documentation) then use it to predict classification values at every point in a grid of 200 $\times$ 200 pixels from -4 to 4 in each direction - plot these. Also plot the scattered training data on top of this.

The SVC has several methods for extracting useful analysis information. SVC.decision_function() can be used to calculate the decision function throughout; its form matches the classification output from SVC.predict(), so can be plotted in exactly the same way. Plot contours at -1, 0 and 1, corresponding to the margin positions and the decision boundary itself; this can be added to the same plot with:

```
Z = np.reshape(svm.decision_function(Xgrid), [npx, npy])
ax.contour(x1line, x2line, Z, colors='k', levels=[-1, 0, 1],
                              alpha=0.5, linestyles=['--', '-', '--'])
```

where x1line and x2line are the two coordinates for which the grid is defined and Z is the decision function, reshaped to match the coordinates, Xgrid is the combination of the gridding vectors, defined via numpy.meshgrid() as in previous tutorials, and npx and npy are both 200 for the number of pixels in x and y respectively. In this plot, the two dashed lines define the extent of the margin. The solid line is the classification boundary itself.

We now want to plot which points are the support vectors.

```
sv = svm.support_vectors_
```

gives the support vectors. Plot these points on the scatter plot too with black crosses (marker="x", c="#000000"). You should notice that several of the support vectors lie within the margin region (although in this case, none cross the boundary itself). Why do some of these points cross the margin edge? Based on what we looked at in the lecture, what could you do to stop this happening? Note that we will find out about the specific scikit-learn parameters in a minute.

Now try out different functions. First of all see what happens with linear mapping (kernel = 'linear') for this system. Does this behaviour match what you'd expect? Why? The degree=2 parameter for the polynomial kernel should be self-explanatory – try switching to a fourth-order polynomial instead. What do you think about the result?

Then try varying C (how hard the boundary is) although note that there is a different definition here from what we discussed in the lecture - small C allows large overlaps, while large C fits the data very tightly (hard boundary) – this is the reciprocal behaviour of what we were considering. For C=1000 with kernel='poly', degree=2, you should see that there are just the support vectors needed to define the margin boundaries. This works well with well separated data, but could be more tricky if the data overlaps. What happens to bias and variance when C is very large?

Also try kernel='rbf' - this is a very common choice - the radial basis functions. Again, adjust the C value and see the effect on the resulting decision surface and the support vectors.

3 Circular dataset

In tutorial 1 we optionally defined a circular dataset. This is another example which is a challenge to segment via linear classifiers. The functions provided include 'gen_circular_distribution' which will generate the distribution for you. Use 200 points, and fit a SVM to the data. Again, try some different kernel functions and parameters to see how they perform. Note that you should plot across the range -10 to 10 in each dimension if plotting.

3.1 Performance evaluation

Select the rbf kernel and increase the number of points for the distribution to 500. We will now demonstrate how overfitting will behave as well as learning how to implement the k fold cross-validation approach (discussed in lecture 3).

We will use the k fold approach to test the performance for several different parameters. Scikit-learn already has tools to achieve this built in. You can set this up with:

```
from sklearn.model_selection import KFold

kf = KFold(n_splits=5, shuffle=True)
```

n_splits defines the number of splits, and shuffle means that the data points are shuffled before being split, i.e. the points are randomly allocated to each split. We can then loop through the different splits as follows:

```
for train_index, test_index in kf.split(X):
        X_train = X[train_index]
        y_train = y[train_index]
        X_test = X[test_index]
        y_test = y[test_index]

        #use X_train, y_train to train the SVM
        #...
        #use svm.predict() to predict the output for the test data set
        #...
        #loop through to compare the test data output to what it should be
        #       and obtain the fraction of correct classifications)
        #...
        #do the same prediction and performance assessment performance
        #       with the training data
        #...
```

Write code for the commented sections where indicated by '...'. When you loop through to compore the data, you may wish to reuse code from one of the earlier tutorials; we did something similar in tutorial 2. You should average your five sets of performance values together.

More detail about the code

The for loop extracts pairs of indices from the kf.split() function. We get 5 sets of these - corresponding to the number of splits identified - train_index and test_index are set to these. If you want to check the sizes of these, you should see that train_index has 80% of the indices, and test_index the remaining 20%. These indices will be different parts for each loop. We use the indices to define the training and testing datasets in the first four lines within the loop.

Training on the training dataset should be straightforward, and predicting with the predict() function similarly. This should output a variable comparable to y_test, and you should compare each element by looping through.

The last part, assessing the performance with the training data is less necessary, however, it is useful to help judge overfitting and whether there is any discrepance between the training fitting and the performance with the test dataset.

Observe the results for training and testing data when C = 0.03, C = 0.2, C = 2, C = 20. What would you expect to happen to these errors as C varies? What does this mean for bias and variance?

If you have time and want to investigate more, you can try to plot a graph as a function of C. In order to investigate the behaviour of the performance for both the training and testing data fully, you can add another loop around the outside of this to vary C automatically; it is wise to use a power law to sample, e.g.:

```
C_array = np.power(10, np.linspace(-1.5, 1.5, 8))
```

This will generate eight of values between $10^{-1.5}$ and $10^{1.5}$. Produce a plot of the two values as a function of C. Does this match your expectation? Based on this, what would you set your value of C to be? Identify where bias and variance would be high and low respectively.