6CCS3ML1 (Machine Learning)

Practical 6: Support vector machines

(Version 1.1)

1 Overview

This practical explores ideas from Lecture 5 on kernel machines. You will use `scikit-learn` to carry out support vector classification and support vector regression. You will do this using kernel functions that are built into `scikit-learn`, and you will also implement your own kernel function. For this practical you will again use datasets from the UCI Machine Learning Repository.

Since you will mainly be re-using ideas that we have met before, in this practical there is no skeleton code to get you started. You are expected to do it all yourself. (Of course, you have examples in code you have been given for previous practicals.)

You are also expected to use the `scikit-learn` documentation to figure out how to use the SVM functions. Of course, they behave much like the classifiers and regression models that we have seen so far, so you have plenty of examples to consult.

2 Getting started

The dataset to use for SVM classification is in the file:

`data_banknote_authentication.txt`

You can download this from KEATS. The dataset is from the UCI Machine Learning Repository banknote authentication dataset:

`https://archive.ics.uci.edu/ml/datasets/banknote+authentication`

As the name suggests, this is data about banknotes, and the classification problem is to identify whether they are genuine or forged. The basis of the data is a series of images, and these were then processed using a Wavlet Transform to obtain a set of features. The machine learning task is to use these features to classify the data.

First, however, you need to load the dataset into a Python program. You have an example of doing this in the code I gave you for Practical 4. Of course you will have to modify it to take account of the fact that the data has a different number of features than the data we used in that practical. You can either read the dataset description at:

`https://archive.ics.uci.edu/ml/datasets/banknote+authentication`

or load the data into a spreadsheet in order to figure out what you need to know about the features.

Once you have the data in your program, I suggest that you downsample it. The original dataset has 1372 entries, and using all of them means that your program will be slow. So I used the `train_test_split` function to randomly select 20% of the data to work with.

3 Support vector classification

Now we come to using SVM classifiers. The `scikit-learn` implementation is documented at:

`http://scikit-learn.org/stable/modules/svm.html`

From this you can see that you use it exactly like any other classifier. First you create a classifier object. Then you fit it to the data. Then you use it to run `score` on a dataset, or to `predict` the outcome for a single example. (If you have forgotten how we do this, look at `classify-iris-simple.py` from Practical 2.) Use the `svm.SVC` classifier.

The main difference between the SVM classifier in `scikit-learn` and the classifiers we have used before is that you get to specify what kind of kernel function to use. As the documentation says, there are four that can be used with `svm.SVC`.

- `linear`: in this case the kernel function is just the dot product of the feature vectors.

  Thus we aren't using the kernel trick, just the ability of an SVM classifier to find the maximum margin separator.

- `rbf`: radial basis function.

- `poly`: polynomial.

  Here you have an additional parameter `degree` which chooses the degree of the polynomial.

- `sigmoid`: sigmoid function.

Note that what `scikit-learn` calls "SVM classification" is what we called "soft margin classification", so there is an option to set the value of the constant C which limits the values of the slack variables. If you don't set C, it defaults to 1, which is fine.

Your job is to:

1. Pick two features from the data set to run the classifier on.

2. Train and test SVM classifiers with each of the four kernel functions: linear, radial basis function, polynomial and sigmoid.

   Since this is a hard problem (you are asking the classifier to distinguish two classes of picture based only on two numbers which summarise the picture), don't be surprised to see quite low accuracy rates. We will come back to this later.

3. For each classifier, plot the decision boundary and the training data.

   For this, I got the plots in Figure 1. Note how non-linear some of the decision boundaries are when mapped back into the original feature space.

   If you don't remember how to produce the decision boundary, take a look at the code in `classify-iris-simple.py`. Remember that you may get memory problems when you try to plot the decision boundaries. This occurs because you are classifying a vary large number of points to create the decision boundaries. (The code sweeps through all the points in the range of the features, classifying each point individually — this is also why the plot is slow.) So reduce the number of points. One easy way to do this is to reduce the value of `plot_step`. You won't get pictures that are as pretty as the ones I got, but at least your code won't crash.

4. Explore different pairs of features to see how accuracy rates and the decision boundaries vary.

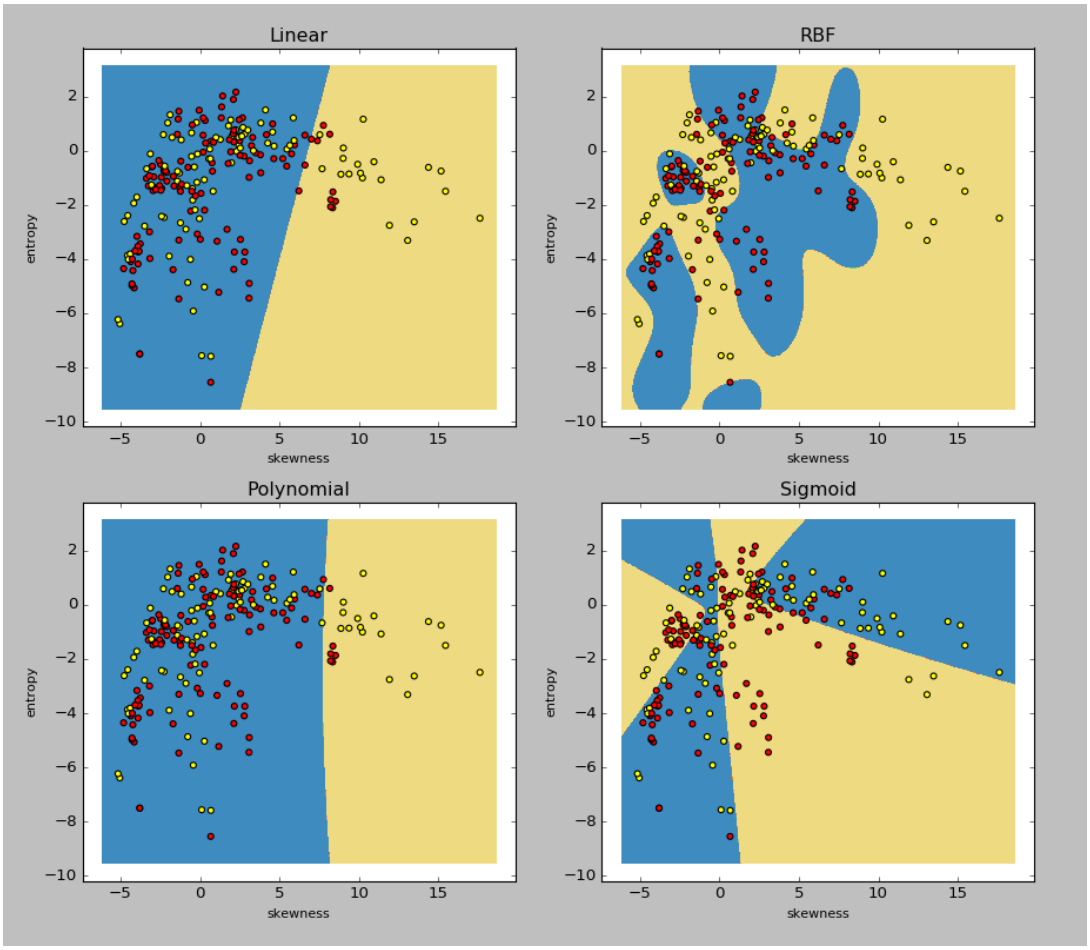

Figure 1: The result of SVM classification on two banknote features.

4 Support vector regression

As we discussed in Lecture 6, SVMs can be used for regression as well as for classification. To test SVM regression we need a different dataset, so download:

`winequality-white.csv`

from KEATS. Each entry in this dataset relates to white variants of the Portuguese "Vinho Verde" wine. The target in this case is an integer (between 0 and 10) which indicates the quality of the wine.

You should:

1. Write a program which loads the Wine Quality data.
2. Run a regression on the data using the four basic kernels supplied by `scikit-learn`.
3. Compute the mean squared error for each of the classifiers (just as you did for Practical 3).

The `scikit-learn` documentation provides information on SVM classification here:

`http://scikit-learn.org/stable/modules/svm.html#regression`

and here:

`http://scikit-learn.org/stable/auto_examples/svm/plot_svm_regression.html`

5 Write your own kernel

`scikit-learn` makes it easy to write your own kernel. You just define a function which implements the kernel. You can then invoke that kernel when you create an instance of an SVM classifer.

The documentation explains how to do this here:

`http://scikit-learn.org/stable/modules/svm.html#using-python-functions-as-kernels`

You should:

1. Use this approach to compute cubic, polynomial and Gaussian kernels.
2. Test an SVM classifier that uses your kernels. How do they perform?

A couple of things to note:

- The function:

  `np.dot( )`

  which takes as arguments two vectors and returns the dot product of the vectors. You will probably want to use this.

- There are two ways to create your own kernel — using Python functions as kernels, and by computing the Gram matrix[1].

  Once you have built your own kernel by writing a Python function, try the Gram matrix method.

  You will find that the Gram matrix method implemented in `scikit-learn` only allows you to test on values that have already been computed. So you cannot have a separate training and test set.

6 More

If you want to try other things:

1. Part of the reason that we got poor performance with the SVM classifier was that we were just using two features.

   Try running the four classifiers on larger feature sets.

   It is still instructive to plot the decision boundaries, but to do that you need to look at just two features at a time. So, to get the full picture, you will need to plot the decision boundary for each pair of features.

[1] A method that pre-computes the kernel function, see `http://www.cs.cornell.edu/courses/cs6787/2017fa/Lecture4.pdf`

2. Run a proper 10-fold cross-validation to establish the kernel with the best accuracy. Clearly you will not be able to use Gram-matrix defined kernels for this exercise.
3. Establish the precision, recall and F1 score for all of the classifiers.

7 Version list

- Version 1.0, January 30th 2020.
- Version 1.1, February 9th 2021.