CSC 2545, Spring 2017
Kernel Methods and Support Vector Machines

Assignment 3

Due on Sunday April 16 at 11pm.
No late assignments will be accepted.

The material you hand in should be legible, well-organized and easy to mark, including the use of good English. Short, simple answers and proofs will receive more marks than long, complicated ones. Up to 20% of your mark will be for presentation. Unless stated otherwise, you must justify your answers. All computer problems are to be done in Python with Scikit-learn and should be properly commented and easy to understand. These programs should all be stored in a single file called `programs.py`. We should be able to import this file as a Python module and execute your programs.

You should hand in three files: `programs.py`, the source code of your Python program; `output.pdf`, a file of all the requested program output; and `answers.pdf`, a file of answers to all the non-programming questions (scanned hand-writing is fine, as long as it is clear and legible). The three files should be submitted electronically as described on the course web page.

No more questions will be added

1. (10 points) Question 2.6 on page 56 of the text.

2. (10 points) Question 13.4 on page 423 of the text.

3. (20 points) The Representer Theorem considers prediction functions of the form $\tilde{f}(x) = f(x) + b$, where $f \in RKHS$ and $b \in \mathbb{R}$. Here, the bias term is a simple constant, b. We can extend this to more-general bias terms that include prior knowledge about how bias depends on x. For example, given functions ψ_1 and ψ_2, we might consider prediction functions of the form $\tilde{f}(x) = f(x) + b_1\psi_1(x) + b_2\psi_2(x)$, where $b_1, b_2 \in \mathbb{R}$. With this in mind, answer Question 4.3 on page 123 in the text.

4. (28 points total) This problem generalizes Question 5 in Assignment 2. That is, we consider optimization problems of the form

$$\min_{\beta,b} \sum_{i,j} \beta_i \beta_j k(x_i, x_j)/2 + C \sum_i R[y_i f(x_i)] \tag{1}$$

where $f(x) = \sum_i \beta_i k(x, x_i) + b$ and R is not-necessarily the hinge loss function. Instead, we assume only that R is decreasing and convex (like the hinge loss). To keep things simple, we also assume that nothing is degenerate; that is, R is strictly convex and twice

differentiable (and therefore $R''(z) > 0$), the Gram matrix of the training data, K, is strictly positive-definite, and equation (1) has a minimum. Here, $K_{ij} = k(x_i, x_j)$, and x_i and x_j are training points. As usual, we assume K is symmetric.

(a) (7 points) Show that β_i has the same sign as y_i.

(b) (3 points) Show that if x_i is badly misclassified, then $|\beta_i|$ is large.

(c) (3 points) Show that if x_i is on the correct side of the decision boundary by a wide margin, then $|\beta_i|$ is small.

(d) (3 points) What properties of a soft-margin SVM do the above three properties generalize?

(e) (5 points) A SVM has the property that $\sum_i y_i \alpha_i = 0$ (Equation (7.39) on page 205 of the text). Derive the analogous property for learning machines defined by Equation (1) above.

(f) (7 points) Show that $\sum_{ij} \beta_i \beta_j k(x_i, x_j) = \sum_i \beta_i f(x_i)$ at the solution to equation (1). Use this to interpret the minimization problem.

5. (20 points) In this question, you will train an SVM to classify images of handwritten digits. There are ten different digits (0 to 9), so you will be training a multi-class SVM, not a binary SVM (see page 211 in the text).

To start, download the MNIST data file from the course web page. The file is called `mnist.pickle.zip`. Uncompress it (if your browser did not automatically do it). Start the Python interpreter and import the `pickle` module. You can then read the file `mnist.pickle` with one of the two commands below, the first for Python 2.x, and the second for Python 3.x. (`'rb'` opens the file for reading in binary.) If you have any difficulty openning the file, try downloading the uncompressed version.

```
with open('mnist.pickle','rb') as f:
    data = pickle.load(f)

with open('mnist.pickle','rb') as f:
    data = pickle.load(f,encoding='latin1')
```

The variable `data` will now contain a Python dictionary with two keys, called `training` and `testing`. `data['training']` contains a tuple with 10 fields, numbered 0 through 9. Likewise for `data['testing']`. Each field contains a 2-dimensional array having 784 columns. Each row of each array represents a hand-written digit. For example, each row of the array in field 7 represents the digit 7 (i.e., each row of `data['training'][7]`, and each row of `data['testing'][7]`). Each training array represents about 6,000 digits, and each testing array represents about 1,000 digits.

Although each digit is stored as a row vector with 784 components, it actually represents an array of pixels with 28 rows and 28 columns ($784 = 28 \times 28$). Each pixel is stored as a floating-point number, but has an integer value between 0 and 255 (i.e., the values representable in a single byte).

To view a digit, you must first convert it to a 28×28 array using the function `numpy.reshape`. You can check the dimensions of a NumPy array by using the function `numpy.shape`. (Alternatively, given a NumPy array, `A`, you can use `A.shape` and `A.reshape`.) To display a 2-dimensional array as an image, you can use the function `imshow` in `matplotlib.pyplot`. To see an image in black-and-white, add the keyword argument `cmap='Greys'` to `imshow`. To remove the smoothing and see the 784 pixels clearly, add the keyword argument `interpolation='nearest'`. Try displaying a few digits.

Your job is to fit an SVM with a RBF kernel to the training data and then estimate its accuracy on the testing data. Unlike assignments 1 and 2, in which we could generate large amounts of test data, here (as in most real problems) we have a relatively small amount of test data, so you cannot use it to choose the hyper-parameters of the SVM and the kernel (i.e., C and γ). Otherwise, you would be using the test data to learn the hyper-parameters and to test them, which leads to overly-optimistic estimates of accuracy. Instead, you will use cross validation to choose the hyper-parameters, and reserve the test data until after all the learning has been completed.

In validation, the SVM is fit to a (randomly chosen) portion of the training data, and the rest of the training data (called validation data) is used to estimate the accuracy of the trained SVM. One chooses the hyperparameters that give the lowest error on the validation data. Finally, these hyperparameters are used to fit an SVM to the entire training set, and the test data is used to estimate its accuracy.

Cross validation is a variation on this. It is more complicated but gives a better estimate of the validation error. The idea is to randomly partition the training data into K subsets, called folds. One of the folds is used as validation data and the remaining K-1 folds as training data. This is done in K different ways, using each of the K folds in turn as validation data. This gives K different estimates of the validation error. The average of these K errors is called the cross-validation error. The values of the hyper parameters that give the lowest cross-validation error are then chosen to train the final SVM. This method is called K-fold cross validation. You can read more about cross validation on page 217 of the text, on Wikipedia, and at the following scikit-learn web page:

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

You can use the function `cross_val_score` in `sklearn.model_selection` to perform k-fold cross validation. The keyword argument `cv` specifies the number of folds. For example, `cv=5` would perform 5-fold cross validation. Note that `cross_val_score` does not generate a *random* partition of the training data into K folds. To achieve this, you should randomly permute the training data before using `cross_val_score`. This will ensure that the data in each fold comes from the same distribution.

What to do: Write Python programs to carry out the following tasks:

(a) The training and test data read in from the file `mnist.pickle` is not in the form required by the functions for training and testing a classifier in scikit-learn. Transform the data into the correct form.

(b) Using K-fold cross validation, find the values of C and γ that give the smallest cross-validation error on the MNIST training data. Print out and hand in these values. Also print out and hand in the list of validation errors returned by `cross_val_score` and their average (the cross-validation error). You should be able to achieve a cross-validation error of less than 0.02.

(c) Fit an SVM on the entire training set using the hyper parameters found in part (b). Compute and hand in the training error.

(d) Evaluate the SVM on the test data. Compute and hand in the test error.

(e) Choose a random sample of 36 (different) test images that the SVM misclassifies. Display the images in a single figure in a 6×6 grid pattern. Hand in the figure.

The grid search in part (b) above is the computationally most-intensive part of this question. You will need to perform a grid search over a wide range of values for C and γ. Moreover, for each combination of C and γ, you will need to perform K-fold cross validation, which involves training an SVM K times on K large data sets, as well as testing it K times. Since you have no idea a priori what the best values of C and γ are, you will have to try a wide range of values, spanning many orders of magnitude. If you have access to a parallel computer, you can do the grid search in parallel by putting different combinations of C and γ on different cpu's. If you only have access to a single cpu, here are some other ways to reduce the search time:

(a) Do the search in two phases. In phase one, do a coarse-grained grid search over many orders of magnitude. In phase two, do a more fine-grained search over a much smaller range centered on the best values of C and γ that you found in phase one. (Only after phase two is complete do you fit an SVM to all the training data, and estimate its accuracy on the test data.)

(b) To speed up the coarse-grained search, use only a portion of the training data. In particular, use the data for only two digits, instead of all ten digits. This way, you can train a binary SVM, which is much faster than training a multi-class SVM (which involves training many binary SVMs).

(c) To speed up the coarse search even more, do not use K-fold cross validation. Instead, do simple validation. i.e., split the training data randomly into a validation set and a (smaller) training set. This way, you train an SVM only once (on the small training set) and test it only once (on the validation set) for each combination of C and γ. You can use the function `train_test_split` in `sklearn.model_selection` to randomly split the training data into two smaller sets.

(d) Only in phase two, do you need to use K-fold cross validation on all the training data for all ten digits, since only then are you trying to get an accurate estimate of C and γ.

(e) Use a small value of K, such as K=3 (the default), for the fine-grained search in phase two.

(f) The cost of training and testing goes up dramatically when C and γ are far from their optimal values. This is because most training points end up on or inside the margins,

so the number of support vectors goes up dramatically. To avoid this problem, the fine-grained search in phase two should consider only a narrow range of values, at most one order of magnitude. You may even have to conduct multiple fine-grained searches over narrow ranges.

(g) Think carefully about what a reasonable value of γ might be. The RBF kernel is a similarity measure, but if gamma is too large, then no image will be similar to any other image. Conversely, if γ is too small, then all the images will be highly similar to each other. Neither situation will give you good predictions. It will also greatly increase the time required for training and testing (because of the large number of support vectors). To find a reasonable range of values for γ, recall that each image has 784 pixels and that each pixel has a value between 0 and 255.

You may have to run your grid search(es) for several hours or all night. To estimate the total execution time of the grid search, you can use the function `time` in the module `time` to measure the execution time of each call to `cross_val_score`.