University of Toronto Mississauga
Department of Mathematical and Computational Sciences
CSC 411 - Machine Learning and Data Mining, Fall 2018

Assignment 1

Due date: Monday October 15, 11:59pm.
No late assignments will be accepted.

As in all work in this course, 20% of your grade is for quality of presentation, including the use of good English, properly commented and easy-to-understand programs, and clear proofs. In general, short, simple answers are worth more than long, complicated ones. Unless stated otherwise, all answers should be justified. The TA has a limited amount of time to devote to each assignment, so what you hand in should be legible (either typed or *neatly* hand-written), well-organized and easy to evaluate. (An employer would demand no less.) All computer problems are to be done in Python with the NumPy, SciPy and scikit-learn libraries.

Hand in five files: The source code of all your programs (functions and script) in a single Python file, a pdf file of figures generated by the progams, a pdf file of all printed output, a pdf file of answers to all the non-programming questions (scanned hand-writing is fine, but *not* photographs), and a scanned, signed copy of the cover sheet at the end of the assignment. Be sure to indicate clearly which question(s) each program and piece of output refers to. All the Python code (functions and script) for a given question should appear in one location in your source file, along with a comment giving the question number. All material in all files should appear in order; *i.e.*, material for Question 1 before Question 2 before Question 3, etc. It should be easy for the TA to identify the material for each question. In particular, all figures should be titled, and all printed output should be identified with the Question number. The five files should be submitted electronically as described on the course web page. In addition, if we run your source file, it should not produce any errors, it should produce all the output that you hand in (figures and print outs), and it should be clear which question each piece of output refers to. *Output that is not labled with the Question number will not be graded.*

I don't know policy: If you do not know the answer to a question (or part), and you write "I don't know", you will receive 20% of the marks of that question (or part). If you just leave a question blank with no such statement, you get 0 marks for that question.

No more questions will be added.

Tips on Scientific Programming in Python

If you haven't already done so, please read the NumPy tutorial on the course web page. Be sure to read about slicing and indexing Numpy arrays. For example, if `A` is a matrix, then `A[:,5]` returns the 5th column, and `A[7,[3,6,8]]` returns the 3rd, 6th and 8th elements in row 7. Similarly, if `v` is a vector, then the statement `A[6,:]=v` copies `v` into the 6th row of `A`. Note that if `A` and `B` are two-dimensional numpy arrays, then `A*B` performs *element-wise* multiplication, *not* matrix multiplication. To perform matrix multiplication, you should use `numpy.matmul(A,B)`. Whenever possible, *do not use loops*, which are very slow in Python. In particular, avoid iterating over the elements of a large vector or matrix. Instead, use numpy's vector and matrix operations, which are much faster and can be executed in parallel. For example, if `A` is a matrix and `v` is a column vector, the `A+v` will add `v` to every column of `A`. Likewise for rows and row vectors. Also, the functions `sum` and `mean` in `numpy` are useful for summing or averaging over all or part of an array. Many NumPy functions that are defined for single numbers can be passed lists, vectors and matrices instead. For example, $f([x_1, x_2, ..., x_n])$ returns the list $[f(x_1), f(x_2), ..., f(x_n)]$. The same is true for many user-defined functions. The term `numpy.inf` represents infinity. It results from dividing by 0 in numpy. It can also result from overflow (*i.e.*, from numbers that are too large to represent in the computer, like 10^{1000}). The term `numpy.nan` stands for "not a number", and it results from doing 0/0, inf/inf or inf-inf in numpy. For generating and labelling plots, the following SciPy functions in `matplotlib.pyplot` are needed: `plot`, `xlabel`, `ylabel`, `title`, `suptitle` and `figure`. You can use Google to conveniently look up SciPy functions. e.g., you can google "numpy matmul" and "pyplot suptitle".

1. (28 points total) This simple warm-up question illustrates Numpy's facilities for indexing and computing with arrays without using loops. In each question below, you should use at most one assignment statement and one print statement (in addition to printing the question number). In questions (h) to (n) you should use one print statement and *no* assignment statements. Each question has a simple solution. Do not use any loops. (2 points each.) Your code should look like this:

```
import numpy as np
import numpy.random as rnd

print '\n\nQuestion 1'
print     '----------'

print '\nQuestion 1(a):'
A = ...
print A

print '\nQuestion 1(b):'
x = ...
print x

print '\nQuestion 1(c):'
B = ...
print B
```

   (a) Construct a random 4×3 matrix. Call it `A`.

   (b) Construct a random 4-dimensional column vector (that is, a 4×1 matrix). Call it `x`.

   (c) Reshape `A` into a 2×6 matrix. Call the result `B`. `A` itself does not change.

   (d) Add vector `x` to all the columns of matrix `A`. Call the resulting matrix `C`. `C` has the same dimensions as `A`.

   (e) Reshape `x` so that is a 4-dimensional vector instead of a 4×1 matrix. That is, change its shape from (4,1) to (4). Call the resulting vector `y`. (Note that `y` is neither a column vector nor a row vector. We say it has rank 1.)

   (f) Change column 0 of matrix `A` to have the same value as vector `y`.

   (g) Add vector `y` to column 2 of matrix `A` and assign the result to column 0 of matrix `A`. (Only column 0 changes.)

   (h) Print the first two columns of matrix `A`.

   (i) Print rows 1 and 3 of matrix `A`.

   (j) Compute the sum of each column of matrix `A`. The result is a 3-dimensional vector.

(k) Compute the maximum of each row of matrix `A`. The result is a 4-dimensional vector.

(l) Compute the average of all the elements in matrix `A`. The result is a single real number.

(m) Compute the log of the square of each element in matrix `A`. The result is a matrix with the same dimensions as `A`.

(n) Using matrix multiplication, compute $\mathtt{A}^T\mathtt{x}$, where $\mathtt{A}^T$ is the transpose of matrix `A`. The result is a 3-dimensional column vector (i.e., a 3×1 matrix).

You should use the functions `reshape`, `sum`, `max` and `mean` in `numpy`, as well as the function `random` in `numpy.random`. The expression `A.T` computes the transpose of matrix `A` (as does the Numpy function `transpose`). You may also find the Numpy function `shape` useful. You will have to look these functions up in the Numpy manual (simply google them) and read their specifications carefully.

2. (12 points) This simple warm-up question is meant to illustrate the vast difference in execution speed between iteration in Python (which is slow) and matrix operations in Numpy (which are fast).

   (a) Write a Python function `cube(A)` that computes the cube of a square matrix, `A`, *i.e.* $\mathtt{A}^3$ `= A*A*A`, where `*` denotes matrix multiplication. Recall that matrix multiplication is defined as follows:

   $$C_{ij} \;=\; \sum_k A_{ik} B_{kj}$$

   where A and B are matrices and $C = A * B$. Your program will need to use a triply-nested loop. It should not use any NumPy operations other than `shape` (to determine the dimensions of `A`) and `zeros` (to initialize your computations).

   (b) Write a Python function `mymeasure(N)` to measure execution speed. Specifically, the function should do the following:

   - Use the function `random` in `numpy.random` to create a `NxN` random matrix, `A`.
   - Use the function `numpy.matmul` to compute $\mathtt{A}^3$ in one line of code. Call the result `Cube1`. Use the function `time.time` to measure the execution time of this step. Print out the execution time.
   - Use your function `cube` to compute $\mathtt{A}^3$. Call the result `Cube2`. Use the function `time.time` to measure the execution time of this step. Print out the execution time.
   - Compute and print out the magnitude of the difference matrix, `Cube1`$-$`Cube2`. There are many ways to define the magnitude of a matrix, but for the purpose of this question, we define it to be the largest element in the absolute value of the matrix. That is, the magnitude of a matrix, A, is $\max_{ij} |A_{ij}|$. Do *not* use iteration to compute this magnitude. Instead, use only NumPy operations. You can do this in one line of code.

If your function `cube` is working correctly, the last step should produce a very small number (much less than 10^{-5}), which is due to numerical error. You should also find that using `numpy.matmul` to compute $\mathtt{A}^3$ is *much* faster than using your `cube` function.

(c) Run `mymeasure(200)` and `mymeasure(2000)`, and hand in the printed results. Be sure it is clear which measurement each printed value refers to. In each case, how many floating-point multiplications does `cube` perform. (Depending on your computer, `mymeasure(2000)` could take over an hour to compute. If your computer is slow, you may want to let it run over night.)

Because loops and iteration in Python are so slow, your programs in the rest of this assignment should avoid using them. Instead, you should use matrix and vector operations in NumPy wherever possible.

3. *Kernel Linear Regression: theory.* (20 points total)

In the rest of this assignment, you will fit a function to data using linear least-squares regression. As described in class, the data consists of a set of pairs, $(x^{(1)}, t^{(1)})$, ... $(x^{(N)}, t^{(N)})$, where each $x^{(n)}$ is an input and each $t^{(n)}$ is a target value. Each pair $(x^{(n)}, t^{(n)})$ is called a data point, though for brevity we will sometimes refer to $x^{(n)}$ as a data point (instead of as the input of a data point). In general, $x^{(n)}$ can be a vector, but in this assignment, it will simply be a real number.

The function you will fit to the data takes a real-number, x, as input and returns a real number, $y(x)$, as ouput. It has the form

$$y(x) = w_0 + \sum_{m=1}^{M} w_m \phi(x - s^{(m)}) \tag{1}$$

where $\phi(x) = \exp\left(-x^2/2\sigma^2\right)$ is called the kernel function, and $s^{(1)}, ..., s^{(M)}$ are special real-numbers called *support vectors*, since they are used to define (or support) the fitted function.[1] Note that the kernel function has the shape of a normal (or Gaussian) distribution with a peak at $x = 0$ and a standard deviation of σ. If we let $\psi_m(x) = \phi(x - s^{(m)})$, then the function y is simply a linear combination of $\psi_1, ..., \psi_M$. That is, $y(x) = w_0 + \sum_m w_m \psi_m(x)$ for all x. Also, each $\psi_m(x)$ has the shape of a normal distribution with a peak at $x = s^{(m)}$. The ψ_m are called *basis functions*. Note that each basis function corresponds to a support vector. In particular, ψ_m is defined by $s^{(m)}$. For more information on linear regression and basis functions, see Chapter 3.1 in Bishop.

In an advanced method known as Support Vector Machines (which we may study towards the end of the course), the support vectors are chosen using a sophisticated optimization procedure. However, in this assignment, we will simply use M randomly chosen training points as support vectors, where $M \leq N$. In particular, since I have randomly shuffled the training points, we will simply use the first M training points,

[1] In a more general seting, x and the $s^{(m)}$ can be vectors, and even text objects, not just real numbers.

that is, $s^{(m)} = x^{(m)}$. Your job is to find the weights $w_0, w_1, ..., w_M$ so that the function $y(x)$ best fits the data. In particular, you will find the w that minimizes the loss function,

$$l(w) \;=\; \sum_{n=1}^{N} [t^{(n)} - y(x^{(n)})]^2 \tag{2}$$

where the sum is over all training points, $(x^{(n)}, t^{(n)})$. In this problem, each $x^{(n)}$ is a single real number, not a vector.

Later, you will formulate and solve the linear least-squares problem using gradient descent and you will compute the training and test errors. To do this, you will need to evaluate the fitted function, and you will need to compute a number of gradients, including $\partial l(w)/\partial w$, where $w = (w_0, w_1, ..., w_M)$. Central to this is the notion of a *kernel matrix*, K, which has N rows and $M+1$ columns, that is, one row for each data point and one column for each weight. Specifically, $K_{nm} = \exp\left[-(x^{(n)} - s^{(m)})^2/2\sigma^2\right]$, where $x^{(n)}$ is a data point and $s^{(m)}$ is a support vector. In addition, K has a 0 column consisting entirely of 1s. That is, $K_{n0} = 1$ for $n = 1, .., N$. This is to accomodate the bias term, w_0. Note that each data set defines a different kernel matrix. For example, we can have a training kernel matrix, a testing kernel matrix, etc. A kernel matrix has a simple interpretation. In particular, each row can be thought of as a feature vector; that is, row n is a feature vector for $x^{(n)}$, where the basis function ψ_m defines the m^{th} feature. In this sense, the basis functions convert each data point into a feature vector. Ordinary linear least squares is then applied to the feature vectors.

In the questions below, be sure to get the details of the proofs right, even in parts (a) and (b), which might seem easy. Getting the details right in parts (a) and (b) should be considered as a warm-up for doing part (c).

(a) (4 points) Show how to use the kernel matrix to compute $y(x^{(n)})$ for $n = 1, ..., N$. Express your answer in terms of matrix-vector multiplication. Prove your answer mathematically.

(b) (3 points) Prove that

$$\frac{\partial \|w\|^2}{\partial w} \;=\; 2w$$

where $\|w\|^2 = \sum_m w_m^2$.

(c) (13 points) Prove that

$$\frac{\partial l(w)}{\partial w} \;=\; 2K^T(Kw - t)$$

where $t = [t^{(1)}, t^{(2)}, ..., t^{(N)}]$ is the vector of target values. Here (and in the questions below), most vectors, including t, w and $\partial l(w)/\partial w$, are treated as column vectors.[2]

4. *Kernel Linear Regression: practice.* (30 points)

[2]Be warned, however, that when programming, some `numpy` functions will only work as expected if they are given 1-dimensional vectors, not column or row vectors, which `numpy` views as 2-dimensional matrices.

In this question you will write a Python program to fit a function to data using linear least-squares regression. You will also be computing the mean squared training and test errors, given by:

$$err_{train} = \sum_{n=1}^{N_{train}} [t^{(n)} - y(x^{(n)})]^2 / N_{train}$$
$$err_{test} = \sum_{n=1}^{N_{test}} [t^{(n)} - y(x^{(n)})]^2 / N_{test}$$

where the two sums are over the training data and test data, respectively, and N_{train} and N_{test} are the number of training and test points, respectively.

The first step is to download the file `data1.pickle.zip` from the course web site. Uncompress it (if your browser did not do so automatically). The file contains training and test data. Next, start the Python interpreter and import the `pickle` module. You can then read the file `data1.pickle` with the following command in Python 2.x:

```
with open('data1.pickle','rb') as f:
    dataTrain,dataTest = pickle.load(f)
```

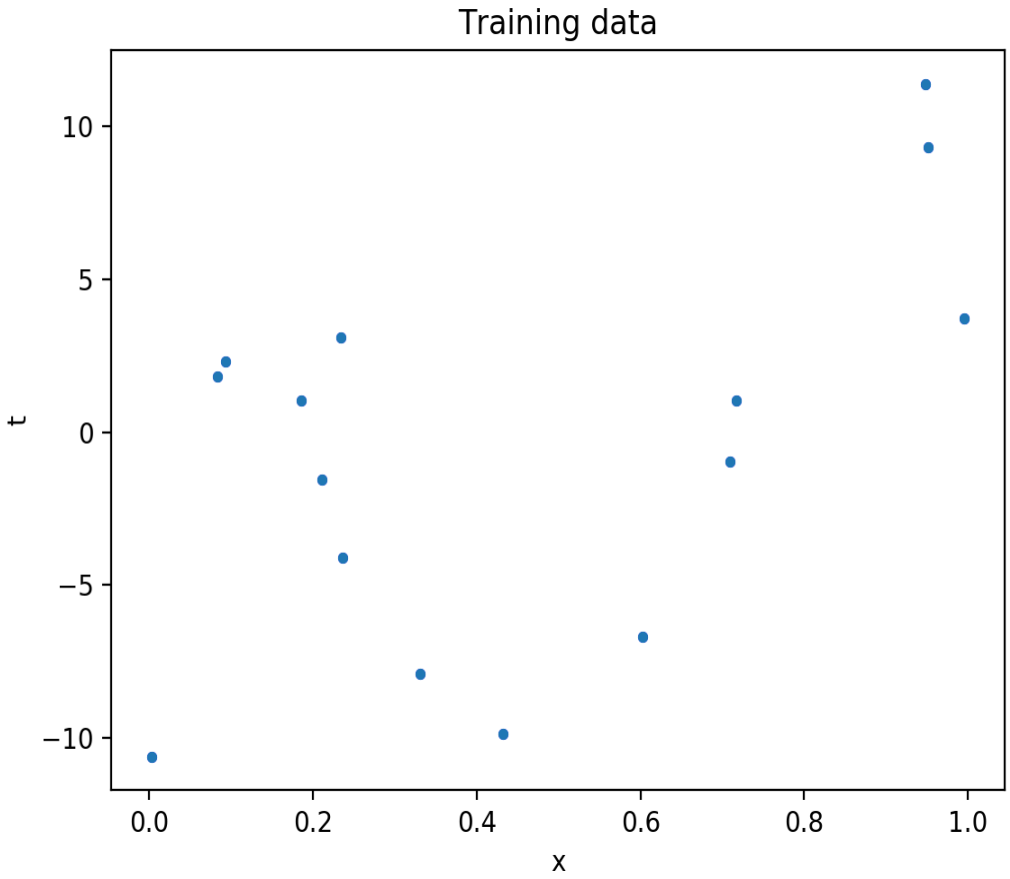
The variable `dataTrain` will now contain the training data, and `dataTest` will contain the test data. Specifically, `dataTrain` is a 15×2 Numpy array, and each row of the array represents a 2-dimensional training point, $(x^{(n)}, t^{(n)})$. Likewise, `dataTest` is an array containing 1000 test points. (If you have trouble reading the data file, it may be because you are using Python 3.x instead of 2.x.) The training data is illustrated in the scatter plot in Figure 1.

In answering the questions below, you should minimize your use of Python loops and use Numpy matrix or vector operations instead. For full marks, you should use a total of at most two loops (and no nested loops) in this question. Unless specified otherwise, you may assume that the training and testing data are in global variables. In the end, you will use training points as support vectors, as described above.

(a) Write a Python function `kernelMatrix(X,S,sigma)` that computes and returns a kernel matrix, `K`, for data set `X` with respect to support vectors `S`. Here, `X` $= [x^{(1)}, x^{(2)}, ..., x^{(N)}]$ is a vector of real numbers, `S` $= [s^{(1)}, s^{(2)}, ..., s^{(M)}]$ is another vector of real numbers, and `sigma` is a single real number representing the standard deviation, σ, of the kernel function.

(b) Write a Python function `plotBasis(S,sigma)` that plots the basis functions defined by the support vectors in `S` and by $\sigma =$ `sigma`. The basis functions should all be plotted on a single pair of axes using a different color for each basis function. Construct the plot using 1000 equally spaced values of x between 0 and 1. Use your function `kernelMatrix` to evaluate the basis functions at each of the 1000 values of x. Label the horizontal axis "x" and the vertical axis "y". Use the function `plot` in `matplotlib.pyplot` to do the plotting. You may find the

Figure 1:



function `numpy.linspace` useful. (The entire function can be written in under 10 lines of code.)

Use `plotBasis` to plot 5 basis functions using the first 5 training points as support vectors. Use `sigma` = 0.2. Title the figure, "Question 4(b): some basis functions with sigma = 0.2". If you have done everything correctly, then each basis function should look like a Gaussian and have a maximum value of 1, as shown in Figure 2, which shows three randomly chosen basis functions with `sigma` = 0.2. For comparison, Figure 3 shows basis functions for the same three support vectors but with `sigma` = 0.1.

Hint: If you are having trouble evaluating the basis functions with the kernel matrix, first evaluate and plot them using any method of your choice (this will take more than 10 lines of code), then once you see what the plots should look like, try using the kernel matrix to evaluate the basis functions (using fewer than 10 lines of code). Note that this is a standard approach to solving any problem of the form, "do X using method Y". *i.e.*, first do X using a method of your choice, then do it using method Y. (In general, when I ask you to use a particular method to solve a problem it is usually a test of your understanding or implementation of the method and a warm-up exercise for later problems. It is also far faster than using loops. That is certainly the case here.)

(c) Write a Python function `myfit(S,sigma)` that uses linear least squares to fit a function of the form (1) to the training data using the support vectors in `S`. The function should return the weight vector, w, and the errors, err_{train} and err_{test}. (The entire function can be written in at most 10 lines of code.)

You should use the method `lstsq` in `numpy.linag` to solve the linear least-squares problem. That is, `lstsq(K,t)` computes the value of w that minimizes the mean squared training error, where `K` is the kernel matrix and `t` = $[t^{(1)}, ..., t^{(N)}]$ is a vector of target values. `lstsq` returns a number of objects. The first one is the weight vector, w.

(d) Write a function `plotY(w,S,sigma)` that plots the function $y(x)$ defined by equation (1) as a red curve. Here, `w` is the weight vector, `S` contains the support vectors, and σ = `sigma`. Construct the plot using 1000 equally spaced values of x between 0 and 1. The function should also plot the training points (as blue dots) in the same figure. Use your function `kernelMatrix` to help evaluate $y(x)$ at each of the 1000 values of x. Use the function `ylim` to limit the y-axis to values between 15 and -15. (The entire function can be written in at most 10 lines of code.)

(e) Use `myFit` to fit a function to the training data using the first 5 training points as support vectors. Use `sigma` = 0.2. Use `plotY` to plot the function. Title the figure, "Question 4(e): the fitted function (5 basis functions)". If you have done everything correctly, the plotted function should be smooth and should approximately capture the trend in the data.

(f) Write a function `bestM(sigma)` that finds the number of basis functions needed to best fit the test data. Specifically, this function should do the following:

- Use `myfit(M,sigma)` to fit functions to the training data using M = 0,1,2,...,15

Figure 2:

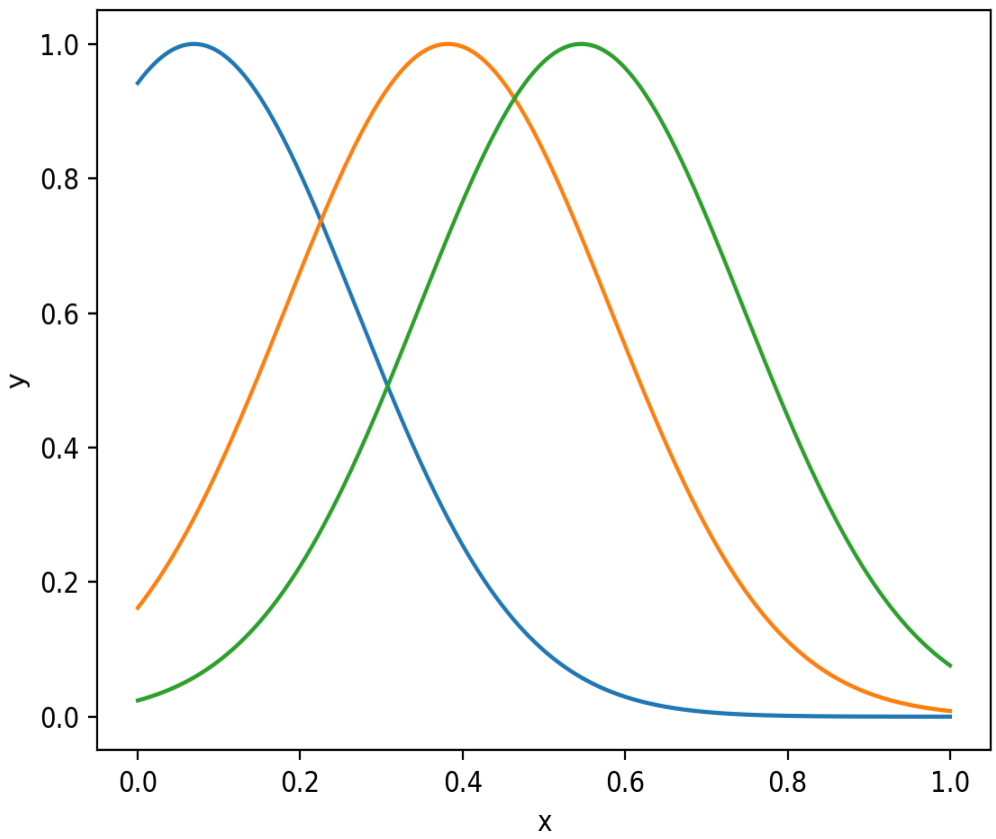
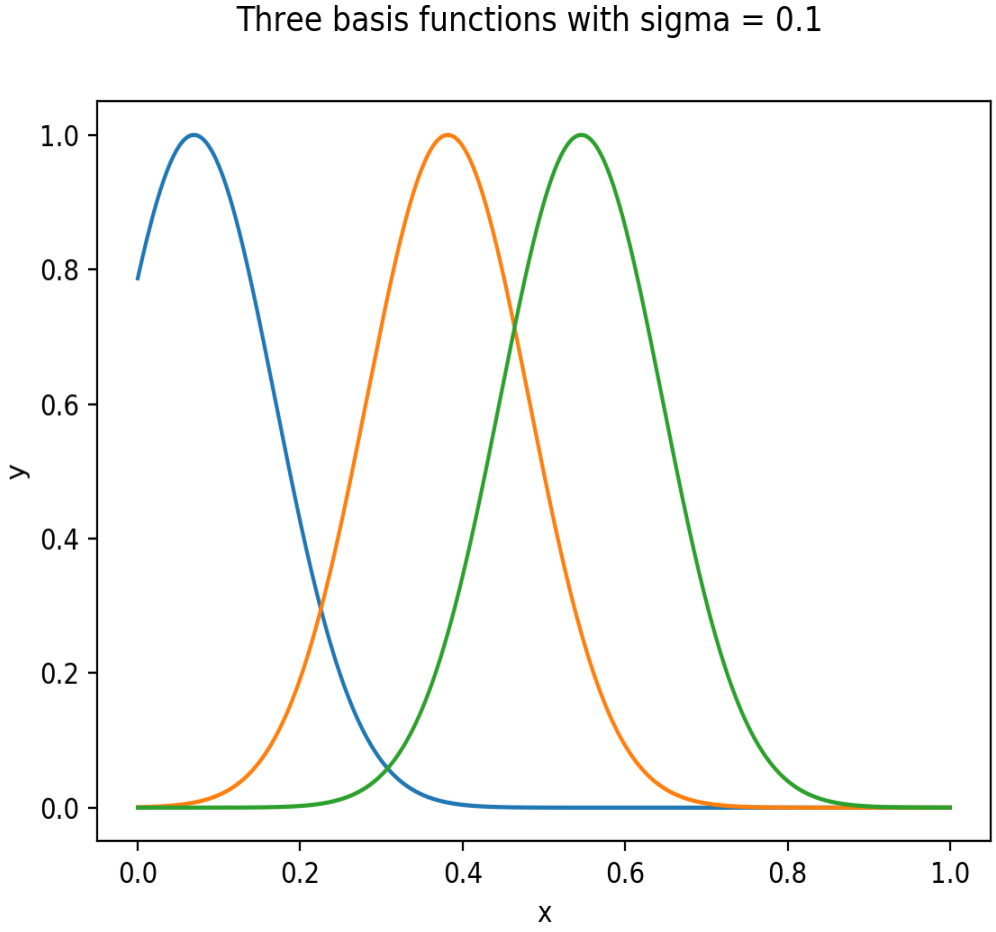

Three basis functions with sigma = 0.2
1.0
0.8
0.6
0.4
0.2
0.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x

Figure 3:


Three basis functions with sigma = 0.1
1.0
0.8
0.6
0.4
0.2
0.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x

basis functions. Use the first M training points as support vectors for the basis functions.

- Use `plotY` to plot each of these fitted functions. Use the function `subplot` in `myplotlib.pyplot` to arrange all 16 plots on a 4x4 grid in a single figure, with M increasing from left to right and top to bottom. (Be sure to call `subplot` outside of `plotY`, and do not call `figure` inside `plotY`.) You should find that the function is a horizontal line for M = 0, then gradually becomes more curvy as M increases, fits the data pretty well for some intermediate values of M, then becomes more and more wiggly as M increases even more, becoming totally wild when M is near 15, and passes through every data point exactly when M = 15. This illustrates *underfitting* for small values of M, and *overfitting* for large values of M. Use the function `suptitle` in `matplotlib.pyplot` to title the entire figure, "Question 4(f): best-fitting functions with 0-15 basis functions". Put the figure into full-screen mode before saving it, to make it more readable.
- Plot the training and test errors as a function of M. Plot them on a single pair of axes using blue for the training error and red for the test error. Limit the vertical axis to a maximum error of 250. Label the axes and give the figure the title "Question 4(f): training and test error". You should find that the test error is usually, if not always, greater than the training error. Also, the training error should decrease as M increases, reaching zero when M=15. The test error should tend to decrease initially and then begin to increase, becoming extremely large when M is near 15. This illustrates that both training and test error are high during underfitting, while training error is low and test error is high during overfitting.
- Plot the best-fitting function. That is, use `plotY` to plot the function for the value of M that gives the lowest test error. In addition, label the axes and give the figure the title "Question 4(f): best-fitting function (M basis functions)", filling in the correct value of M.
- Print out the optimal values of M and w.
- Print out the training and test errors for the optimal values of M and w. You should find that training error < test error.

Note that `bestM` generates a number of different plots, most (but not all) using `plotY`. Run `bestM` using `sigma` = 0.2. Hand in all the plots and all the printed values. It should be clear what each piece of output refers to.

5. *Regularization.* (15 points)

In this problem you will write a Python program to fit a function, $y(x)$, to data using regularized linear least-squares regression (also called Ridge regression). That is, you will estimate the weight vector $w = (w_0, w_1, ..., w_M)$ in Equation (1) that minimizes the regularized loss function

$$\tilde{l}(w) \;=\; l(w) + \alpha \sum_{j=1}^{M} w_j^2 \tag{3}$$

where $l(w)$ is the loss function in Equation (2). The sum over j is the regularization term, and its purpose is to prevent overfitting by preventing the weights from becoming too large. Notice that the bias weight w_0 is not included in the regularization term. This is because w_0 simply controls the height of the fitted function, $y(x)$, and this height does not cause overfitting no matter how large it is.

The coefficient α specifies how important regularization is. The larger α is, the smaller the optimal weights will be. One of your tasks in this question is to explore the effect of different values of α and to learn the optimal value. To do this, you will use a set of *validation data* in addition to training and test data. The training data is used to learn the values of parameters, like the w_j, while the validation data is used to learn the values of hyper-parameters, like α. The testing data is used to estimate the accuracy of the final, learned system. The mean squared validation error is

$$err_{val} = \sum_{n=1}^{N_{val}} [t^{(n)} - y(x^{(n)})]^2 / N_{val}$$

where the sum is over the validation data, and N_{val} is the number of points in the validation data.

In this question, you will use the same training data as in Question 4. However, you will use a new set of testing data as well as a set of validation data. To obtain them, download and uncompress the file `data2.pickle.zip` from the course web site. You can then read the file with the following command in Python 2.x:

```
with open('data2.pickle','rb') as f:
    dataVal,dataTest = pickle.load(f)
```

The variable `dataVal` will now contain the validation data, and `dataTest` will contain the new test data. Unlike Question 4, which provided a plentiful amount of testing data to give very accurate estimates of testing error, this question uses validation and test sets that are comparable in size to the training set, which is a more realistic scenario.

In answering the questions below, you should minimize your use of Python loops and use Numpy matrix or vector operations instead. For full marks, you should use no loops at all, except in part (c), where it is convenient to use one loop (but no nested loops). Unless specified otherwise, you may assume that the training, validation and test data are in global variables. In the end, you will use training points as support vectors, as described above.

(a) Write a Python function `regFit(S,sigma,alpha)` that uses regularized least-squares regression to fit a function of the form (1) to the training data using the support vectors in `S`. The coefficient of the regularization term is $\alpha =$ `alpha`. The function should return the weight vector, w, and the errors, err_{train} and err_{val}.

You should use the Python class `Ridge` to solve the regularized least-squares problem. It is provided not by Numpy, but by scikit-learn. Specifically, the following code performs ridge regression and computes the optimal weight vector, `w`:

```
import sklearn.linear_model as lin
ridge = lin.Ridge(alpha)
ridge.fit(K,t)
w = ridge.coef_
w[0] = ridge.intercept_
```

Here, `K` is the kernel matrix and `t` is the vector of target values. Note that `w[0]` is given special treatment. This is because it is not included in the regularization term and must be computed somewhat differently.

(b) Use `regFit` to fit a function to the training data using all 15 training points as support vectors, `sigma` = 0.2 and `alpha` = 1. Use `plotY` to plot the function. Title the figure, "Question 5(b): the fitted function (alpha=1)". If you have done everything correctly, the plotted function should approximately capture the trend in the data, but should be smoother and flatter than the best-fitting function in Question 4(f).

(c) Define a Python function `bestAlpha(S,sigma)` that uses the validation data to find the best value of the regularization coefficient α in Equation (3). As before, `S` contains the support vectors. In particular, your function should do the following:

- Use `regFit(S,alpha)` to fit functions to the training data for values of `alpha` spread out over 16 orders of magnitude. Specifically, use values of `alpha` such that $\log_{10}$(`alpha`) = -12, -11, -10, ..., 0, 1, 2, 3.
- Use `plotY` to plot each of these fitted functions. Again, use `subplot` to arrange all 16 plots on a 4x4 grid in a single figure, with `alpha` increasing from left to right and top to bottom. You should find that the functions are quite wiggly for small values of `alpha`, almost flat for large values of `alpha`, and fit the data quite well for intermediate values. This illustrates *underfitting* for large values of `alpha`, and *overfitting* for small values of `alpha`. Title the entire figure, "Question 5(c): best-fitting functions for log(alpha) = -12,-11,...,1,2,3". Put the figure into full-screen mode before saving it, to make it more readable.
- Plot the training and validation errors versus `alpha`. Instead of using `plot`, use the function `semilogx` in `matplotlib.pyplot` to draw the curves. This gives a logarithmic scale on the horizontal axis. Plot both curves on a single pair of axes using blue for the training error and red for the validation error. Label the vertical axis "error", and the horizontal axis "alpha", and give the figure the title "Question 5(c): training and validation error". You should find that the validation error is usually, if not always, greater than the training error. Also, the training error should increase as `alpha` increases. The validation error should decrease initially and then begin to increase, reaching a minimum at the optimal value of `alpha`.
- Plot the best-fitting function. That is, use `plotY` to plot the function for the value of `alpha` that gives the lowest validation error. In addition, label the axes and give the figure the title, "Question 5(c): best-fitting function (alpha = ??)", filling in the optimal value of `alpha`.

- Print out the optimal values of `alpha` and w.
- Compute the test error for the optimal values of `alpha` and w.
- Print out the training, validation and test errors for the optimal values of `alpha` and w. You should find that training error $<$ validation error $<$ test error.

Note that `bestAlpha` requires you to generate a number of plots, most (but not all) using `plotY`. Run `bestAlpha` using all 15 training points as support vectors and `sigma` $= 0.2$. Hand in all the plots and all the printed values. It should be clear what each piece of output refers to.

6. *Cross Validation.* (25 points)

Question 5 introduced the idea of minimizing validation error as a way of estimating hyper-parameters like α. Unfortunatley, simple validation does not give good estimates of validation error unless the validation set is fairly large. In Question 5, we used 20 validation points and 15 training points, 35 points in all. This sacrifices a lot of data for validation that could have been used for training. The problem is we want to use as much of our precious data as possible for training (to get a good estimate of the parameters, w), but we also want to hold back a lot of data for use in validation (to get an accurate estimate of validation error, so we can choose a good value for the hyper-parameter, α).

Cross validation is an attempt to get the best of both worlds. It is more complicated than simple validation, but uses more of the data for training and gives a better estimate of validation error with the remaining data. There are several versions of cross validation, and in this question, we will be using one of the simplest, called *k-fold cross validation*. 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.

Notice that training is done K times, once for each fold; and each time, most of the data (K-1 folds) is used for training. Moreover, all K folds (ie, all of the data) is eventually used for validation too, giving a better estimate of validation error. To keep things simple, we will assume in this assignment that the number of training points is a multiple of K, so that all folds have the same size. You can read more about cross validation on pages 32 and 33 of Bishop and on Wikipedia.

In this question, we will not use the training data that was used in earlier questions. Instead, we will use the validation data from Question 5 as training data (since we want 20 training points). We will continue to use the test data from Question 5 as test data.

As always, in answering the questions below, you should minimize your use of Python loops and use Numpy matrix or vector operations instead. For full marks, you should use a total of at most 2 loops (and no nested loops) in this question.

(a) Randomly reorder the validation data from Question 5 and use it as training data in the questions below. Plot this new training data in a scatter plot similar to Figure 1. Label the axes, and title the figure, "Question 6(a): Training data for Question 6". It should show 20 blue points. You may find the function `shuffle` in `numpy.random` useful.

(b) Define a Python function `cross_val(K,S,sigma,alpha,X,Y)` that uses `K`-fold cross validation while fitting a function to the data in `X` and `Y`. The fitted function has the form given by Equation (1), where `S` contains the support vectors and $\sigma =$ `sigma`. The function should be fit using regularized least-squares regression with a regularization coefficient of $\alpha =$ `alpha`.

`X` and `Y` should be divided into `K` folds of equal size for cross validation. (You may assume that the number of points in `X` and `Y` is a multiple of `K`). `X` contains the input values, $x^{(n)}$, in Equation (2), while `Y` contains the target values, $t^{(n)}$. During cross validation, `cross_val` will generate `K` training errors and `K` validation errors, resulting from the `K` attempts to fit a function to the data. Each validation error is estimated on one fold, and each training error is estimated on the remaining `K`-1 folds. `cross_val` should return two `K`-dimensional vectors: a vector of the training errors, and a vector of the validation errors.

(c) Perform 5-fold cross validation by applying `cross_val` to the training data you generated in part (a). Use `alpha` = 1.0, `sigma` = 0.2 and use the first 10 training points as support vectors, `S`. On a single pair of axes, plot the training errors (in blue) and the validation errors (in red). Label the horizontal axis "fold", and the vertical axis "error". Title the figure, "Question 6(c): training and validation errors during cross validation". You should find that both curves are jagged. (If either curve is perfectly straight, your code is incorrect.) If you were unlucky in your random reordering in part (a), then parts of the validation curve may be extremely spikey. Finally, print out the mean training and mean validation errors. You should find that the mean validation error is greater than the mean training error.

(d) Define a Python function `bestAlphaCV(K,S,sigma,X,Y)` that uses `K`-fold cross validation on the data in `X` and `Y` to find the best value of the hyper-parameter α in Equation (3). As usual, `S` contains the support vectors. In particular, `bestAlphaCV` should do the following:

- Use `cross_val(K,S,sigma,alpha,X,Y)` to perform cross validation for values of `alpha` spread out over 16 orders of magnitude. Specifically, use values of `alpha` such that $\log_{10}$(`alpha`) = -11, -10, ..., 0, 1, 2, 3, 4.
- for each value of α, compute the mean training and validation errors.
- Plot the mean training and validation errors versus `alpha` using a log scale on the horizontal axis, as in Question 5(c). Plot both curves on a single pair of axes using blue for the training error and red for the validation error. Label the vertical axis "error", and the horizontal axis "alpha", and give the figure the title "Question 6(d): training and validation error". You should find that the validation error is usually, if not always, greater than the training error.

Also, the training error should increase as `alpha` increases. The validation error should decrease initially and then begin to increase, reaching a minimum at the optimal value of `alpha`. (However, if you were unlucky in your random reordering in part (a), then parts of the validation curve may be extremely spikey.)

- Record the optimal value of α, that is, the value that leads to the lowest mean-validation-error during cross validation.
- Using the optimal value of α, use `regFit(S,alpha)` to fit a function to the data in `X` and `Y`, *i.e.*, using all the data in `X` and `Y` as training data without without holding any data back for validation. We will call this the best-fitting function. Let w be the weight vector of this function. We will call this the optimal value of w.
- Compute the test error of the best-fitting function, using the test data from Question 5.
- Compute the training error of the best-fitting function, using all of `X` and `Y` as the training data.
- Plot the best-fitting function. Label the axes, and give the figure the title, "Question 6(d): best-fitting function (alpha = ??)", filling in the optimal value of `alpha`.
- Print out the optimal values of `alpha` and w.
- Print out the training, mean-validation and test errors for the best-fitting function. You should find that training error is the smallest of the three.
- Sometimes the mean validation error is bigger than the test error. Explain why this happens.

Note that `bestAlphaCV` requires you to generate a number of plots. Run `bestAlphaCV` using 5-fold cross validation on the training data you created in part (a). Use the first 15 training points as support vectors, and use `sigma` = 0.2. Hand in all the plots and all the printed values. It should be clear what each piece of output refers to.

7. *Gradient Descent.* (25 points)

In this question, as in Question 5, you will fit a function to data with regularized least-squares regression. This time, however, instead of using a built-in optimization method (like `Ridge`), you will write your own method based on gradient descent. Use the same training and testing data as in Question 5. (Reminder: If your functions in Question 6 reset the training data, you will have to set it back again.)

Write a Python program `fitRegGD(S,sigma,alpha,lrate)` that fits a function to the training data using the support vectors in `S`. Here, `alpha` is the coefficient of the regularization term, and `lrate` is the learning rate. Your program should compute the optimal weight vector, `w`, using gradient descent. That is, `w` should first be initialized to a random value (*e.g.*, by using `numpy.random.randn`). It should then be updated

repeatedly using the statement

$$w \;=\; w - \lambda \frac{\partial \tilde{l}(w)}{\partial w}$$

where $\tilde{l}(w)$ is the regularized loss function given in Equation (3), and the hyper-parameter λ is the learning rate. You will have to experiment to find a good learning rate. Try using values that differ by a factor of ten (e.g., 10, 1, 0.1, 0.01, 0.001, ...) and monitor the training error. If the training error is `inf` or `nan`, or if it tends to increase with each iteration, then try using a lower learning rate. However, to achieve fast convergence, you will want to use as large a learning rate as possible.

Run your function using all 15 training points as support vectors. Use `sigma` $= 0.2$ and the optimal value of `alpha` that you found in Question 5. Your program should then compute the same value of `w`, and the same training and test error as in Question 5. You will probably find that your function produces a reasonably good fit to the data after a few thousand iterations. However, you will need many more iterations to reach the accuracy of the answer in Question 5. In addition, your function should do the following:

- Perform 100,000 iterations of gradient descent.
- Compute and record the training and test error after every iteration.
- Use `plotY` to plot the fitted function at nine different time points: after 4^0, 4^1, 4^2, ..., 4^8 iterations. Use `subplot` to arrange all nine plots on a 3x3 grid in a single figure, with iterations increasing from left to right and top to bottom. If everything is working correctly, the first plot should be fairly flat and a bit wavey, the next few plots should fit the data increasingly well, and the last few plots sould appear to fit the data quite well. Title the figure, "Question 7: fitted function as iterations increase". Put the figure into full-screen mode before saving it, to make it more readable.
- Use `plotY` to plot the final fitted function in a single figure. Title the figure "Question 7: fitted function".
- Plot the recorded training and test errors on a single pair of axes. The vertical axis of the plot is error, and the horizontal axis is number of iterations. Plot the training error as a blue curve, and the test error as a red curve. Label the axes. Label the figure, "Question 7: training and test error v.s. iterations". If everything is working correctly, both errors should decrease extremely rapidly (almost vertically), then quickly level out and be almost horizontal for most of the plot. Also, the test error curve should be above the training error curve almost everywhere (and certainly at the right end).
- It may appear from the previous plot that gradient descent converges to the final answer almost right away. To see that this is not the case, replot the training and test errors using a log scale on the horizontal axis. Label the figure, "Question 7: training and test error v.s. iterations (log scale)". If everything is working correctly, both errors should decrease in a gently waving curve from left to right.

- To see that gradient descent has not converged completely even after 100,000 iterations, plot the last 10,000 training errors. Label the figure, " Question 7: last 10,000 training errors". If everything is working correctly, the plot should show the errors streadily decreasing from left to right.
- Let `w` be the optimal weight vector computed by gradient descent after the 100,000 iterations above. Print the final values of training error, test error, and `w`. If everything is working correctly, they should be approximately the same as in Question 5. (If not, try using a higher learning rate.)
- To check your program, recompute the optimal weight vector using `regFit` from Question 5(a). Call this weight vector `w2`. If everything is working properly, `w` and `w2` should be almost identical.
- Compute the magnitude of the difference vector `w - w2`, where magnitude is defined in the last item of Question 2(b). Print the magnitude. It should be a small number. In fact, by choosing the learning rate properly, the magnitude should be less than 10^{-4}. You should try to achieve this.
- Print the learning rate.
- Print the value of `alpha`.

Note that `fitRegGD` requires you to generate a number of different plots. Hand in all the plots and all the printed values. It should be clear what each piece of output refers to. For full marks, your program should have only one loop (and no nested loops).

155 points total

University of Toronto Mississauga
CSC 411 - Machine Learning and Data Mining

Cover sheet for Assignment 1

Complete this page and hand it in with your assignment.

Name: _____
(Underline your last name)

Student number: _____

I declare that the solutions to Assignment 1 that I have handed in are solely my own work, and they are in accordance with the University of Toronto Code of Behavior on Academic Matters.

Signature: _____