University of Toronto Mississauga
Department of Mathematical and Computational Sciences
CSC 411 - Machine Learning and Data Mining, Fall 2019

Assignment 1

Due date: Friday October 11, 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 programs, 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 labelled with the Question number will not be graded. Programs that do not produce output 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 indexing and slicing Numpy arrays. First, indexing begins at 0, not 1. Thus, if `A` is a matrix, then `A[7,0]` is the element in row 7 and column 0. Likewise, `A[0,4]` is the element in row 0 and column 4. Slicing allows large segments of an array to be referenced. For example, `A[:,5]` returns column 5 of matrix `A`, and `A[7,[3,6,8]]` returns elements 3, 6 and 8 of row 7. Similarly, if `v` is a vector, then the statement `A[6,:]=v` copies `v` into row 6 of matrix `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. (30 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 (o) 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 3×4 matrix. Call it `A`. The rows are numbered 0,1,2, and the columns are numbered 0,1,2,3.

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

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

   (d) Add vector `x` to all the rows 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 1×4 matrix. That is, change its shape from (1,4) 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 row 0 of matrix `A` to have the same value as vector `y`.

   (g) Subtract vector `y` from row 2 of matrix `A` and assign the result to row 1 of matrix `A`. (Only row 1 changes.)

   (h) Print the first three columns of matrix `A` as a single matrix.

   (i) Print rows 0 and 2 of matrix `A` as a single matrix.

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

(k) Compute the average of each row of matrix `A`. The result is a 3-dimensional vector.

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

(m) For each column in matrix `A`, sum the elements and square the sum. The result is a 4-dimensional vector.

(n) Using matrix multiplication, compute $\mathtt{A}\mathtt{A}^T$, where $\mathtt{A}^T$ is the transpose of matrix `A`. The result is a 3×3 matrix.

(o) Compute the average of $\mathtt{A}\mathtt{x}^T$. The result is a single real number.

You should use the functions `reshape`, `sum`, `min` 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. (15 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 `myfun(A,B)` that computes $\mathtt{A}\mathtt{A}^T\mathtt{+B}$, where `A` is a $M \times N$ matrix and `B` is a $M \times M$ matrix. Recall that matrix multiplication is defined as follows:

   $$E_{ij} \;=\; \sum_k C_{ik} D_{kj} \tag{1}$$

   where C and D are matrices and $E = CD$ is the matrix product of `C` and `D`. Your program will need to use a triply-nested loop. It should not use any NumPy operations that operate on whole matrices. These include matrix addition and multiplication and whole-matrix assignment statements, such as `C=D`. Instead, you should use loops to operate on matrices one element at a time. For instance, you may use single-element assignment statements, such as `C[i,j]=D[i,j]`. Likewise, do not use the NumPy transpose operation, `A.T`, to compute the transpose of `A`. In fact, for full marks, you should not compute the transpose at all. You may use the NumPy operations `shape` (to determine the dimensions of the input matrices) and `zeros` (to initialize your computations).

   (b) Write down an equation similar to Equation (1) for computing $\mathtt{A}\mathtt{A}^T$ but without computing the transpose of `A`. *i.e.*, the equation should not mention $\mathtt{A}^T$. (This is a math question, not a programming question.) Your function `myfun` in part (a) should implement this equation.

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

   - Use the function `random` in `numpy.random` to create two random matrices, `A` and `B`, of size $M \times N$ and $M \times M$, respectively.

- Use your function `myfun` to compute $\texttt{AA}^T\texttt{+B}$. Call the result `C1`. Use the function `time.time` to measure the execution time of this step. Print out the execution time.
- Use the function `numpy.matmul` to compute $\texttt{AA}^T\texttt{+B}$ in one line of code. Call the result `C2`. 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, `C1-C2`. There are many ways to define the magnitude of a matrix, but for the purpose of this question, we define it to be the sum of the absolute values of the matrix elements. That is, the magnitude of a matrix, A, is $\sum_{ij} |A_{ij}|$. 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 `myfun` 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 $\texttt{AA}^T\texttt{+B}$ is *much* faster than using your `myfun` function.

(d) Run `mymeasure(200,400)` and `mymeasure(1000,2000)`, 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* `myfun` *perform? How many floating-point additions does it perform?* (Depending on your computer, `mymeasure(1000,2000)` could take about half 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. *Linear Least-Squares Regression: theory.* (17 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 output. It has the form

$$y(x) = w_0 + \sum_{m=1}^{M} w_m \phi_m(x) \tag{2}$$

where the ϕ_m are non-linear basis functions. In particular,

$$\phi_m(x) \;=\; \frac{1}{1 + \exp \beta(\alpha_m - x)} \tag{3}$$

is a sigmoid function centered on α_m and compressed horizontally by β. The α_m and β are constants that define the basis functions, and will be specified later. Note that the function y is a linear combination of the basis functions. For more information on linear regression and basis functions, see Chapter 3.1 in Bishop.

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 vector $w = (w_1, ..., w_M)$ and the bias term w_0 that minimize the loss function,

$$l(w_0, w) \;=\; \sum_{n=1}^{N} [t^{(n)} - y(x^{(n)})]^2 \tag{4}$$

where the sum is over all training points, $(x^{(n)}, t^{(n)})$. Although the basis functions are non-linear, we can linearize the problem (and use linear least squares to solve it) by transforming the input from a single real number, x, to a feature vector $z = (z_1, ..., z_M)$, where $z_m = \phi_m(x)$. Then,

$$y(x) = w_0 + \sum_{m=1}^{M} w_m z_m \tag{5}$$

which is linear in z and linear in w. 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 $\partial l(w_0, w)/\partial w$.

The most efficient way to do this is to use matrix and vector operations. The first step is to construct a feature matrix, Z, where each row of Z is a feature vector for a point is our data set. In particular, the n^{th} row of Z is the feature vector for $x^{(n)}$. Note that each data set defines a different feature matrix. For example, we can have a training feature matrix, a testing feature matrix, etc.

In the questions below, you will show that many of the key concepts of linear regression can be represented in matrix/vector notation. In these questions, be sure to get the details of your proofs right. Getting the details right in parts (a) and (b) should be considered as a warm-up for doing parts (c) and (d).

(a) (1 point) Prove that $l(w_0, w) \;=\; \|y - t\|^2$. Here $t = [t^{(1)}, t^{(2)}, ..., t^{(N)}]$ is a vector of the target values for our data set, and $y = [y^{(1)}, y^{(2)}, ..., y^{(N)}]$ is a vector of the predicted values. That is, $y^{(n)} = y(x^{(n)})$.

(b) (4 points) Prove that $y = w_0\vec{1} + Zw$, where y and w are treated as column vectors and $\vec{1}$ is a column vector of 1's.[1]

(c) (8 points) Prove that

$$\frac{\partial l(w_0, w)}{\partial w} \;=\; 2Z^T(y - t)$$

[1]In this assignment, most vectors are treated as column vectors. 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.

(d) (4 points) Prove that

$$\frac{\partial l(w_0, w)}{\partial w_0} = 2\vec{1}^T(y - t)$$

4. *Linear Least-Squares Regression: practice.* (30 points)

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)
```

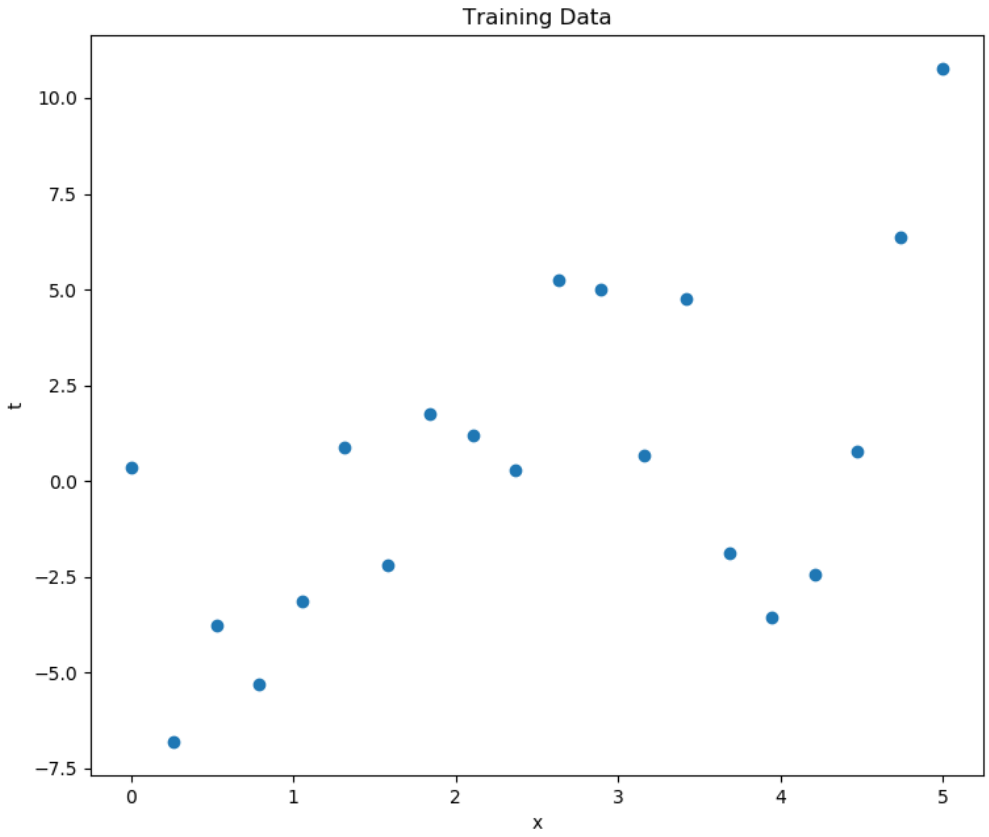
The variable `dataTrain` will now contain the training data, and `dataTest` will contain the test data. Specifically, `dataTrain` is a 20×2 Numpy array, and each row of the array represents a 2-dimensional training point, $(x^{(n)}, t^{(n)})$. Likewise, `dataTest` is an array containing 1000 test points. (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 at most one loop in part (a), no loops in the other parts, and no nested loops at all. Unless specified otherwise, you may assume that the training and testing data are in global variables.

(a) Write a Python function `feature_matrix(X,alpha,beta)` that computes and returns a feature matrix, `Z`, for data set `X`. Here, `X` $= [x^{(1)}, x^{(2)}, ..., x^{(N)}]$ is a vector of real numbers, `alpha` is another vector of real numbers, and `beta` is a single real number. `alpha` and `beta` store the constants $\alpha_1, \alpha_2, ..., \alpha_M$ and β, respectively, in Equation (3). (The function can be written in at most 8 lines of code.)

(b) Write a Python function `plot_basis(alpha,beta)` that plots the basis functions defined by `alpha` and `beta`. The basis functions should all be plotted on a single

Figure 1:



pair of axes using a different color for each basis function. Construct the plot using 1000 equally spaced values of x between `xMin` and `xMax`, where `xMin` is the minimum value of x in the training data, and `xMax` is the largest. You my treat `xMin` and `xMax` as global variables. Use your function `feature_matrix` 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 function `numpy.linspace` useful. (The entire function can be written in 7 lines of code.)

Use `plot_basis` to plot 6 basis functions. In defining the basis functions, use $\beta = 2$, and use values of α_m that are equally spaced between `xMin` and `xMax`. Title the figure, "Question 4(b): 6 basis functions". (Use `plot_basis` to see the effect of changing `alpha` and `beta`. Do not hand in these plots.)

Hint: If you are having trouble evaluating the basis functions with the feature matrix, first evaluate and plot them using any method of your choice (this will take more than 7 lines of code), then once you see what the plots should look like, try using the feature matrix to evaluate the basis functions (using 7 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 `my_fit(alpha,beta)` that uses linear least squares to fit a function of the form (2) to the training data using basis functions defined by `alpha` and `beta`. The function should return the weight vector, w, the bias term, w_0, and the errors, err_{train} and err_{test}. (The entire function can be written in at most 15 lines of code.)

You should use the method `lstsq` in `numpy.linag` to solve the linear least-squares problem. That is, `lstsq(Zaug,t)` computes the weights that minimizes the mean squared training error, where `Zaug` is an augmented feature matrix of the training data, and `t` $= [t^{(1)}, ..., t^{(N)}]$ is the vector of target values. The augmented feature matrix is the feature matrix `Z` computed in part (a) augmented with a leading column of 1's (so it has one more column that `Z`). `lstsq` returns a number of objects. The first one is a weight vector, $(w_0, w_1, ..., w_M)$, that includes the bias term, w_0. (You should *not* use anything else returned by `lstsq`.)

(d) Write a function `plotY(w,w0,alpha,beta)` that plots the function $y(x)$ defined by equation (2) as a red curve. Here, `w` is the weight vector, `w0` is the bias term, and `alpha` and `beta` define the basis functions. Construct the plot using 1000 equally spaced values of x between `xMin` and `xMax`. The function should also plot the training points (as blue dots) in the same figure. Use your function `feature_matrix` to help evaluate $y(x)$ at each of the 1000 values of x. (The entire function can be written in 6 lines of code.) Use the function `scatter` in `matplotlib.pyplot` to plot the training points.

(e) Use `my_fit` to fit a function to the training data using 5 basis functions. In

defining the basis functions, use $\beta = 1$, and use values of α_m that are equally spaced between `xMin` and `xMax`. Use `plotY` to plot the function. Title the figure, "Question 4(e): the fitted function (5 basis functions)". Also, print out the training and test errors. If you have done everything correctly, the plotted function should be smooth and should approximately capture the trend in the training data. Also, the training error should be about 3.7, and the test error should be about 6.1. *Explain why the test error is larger than the training error.*

(f) Repeat part (e) using 12 basis functions. You should find that the function is wigglier, the training error is lower, and the test error is higher. *Explain these results.*

(g) Repeat part (e) using 19 basis functions. (You may get a Python warning, which you can ignore.) You should find that the function is much wigglier and passes through all the training points, the training error is very low (below 10^{-5}), and the test error is very high (over 100). *Explain these results.*

5. *Regularization.* (20 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 (2) that minimizes the regularized loss function

$$\tilde{l}(w_0, w) \;=\; l(w_0, w) + \gamma \sum_{j=1}^{M} w_j^2 \tag{6}$$

where $l(w_0, w)$ is the loss function in Equation (4). 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 γ specifies how important regularization is. The larger γ is, the smaller the optimal weights will be. One of your tasks in this question is to explore the effect of different values of γ 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 γ. 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 (d), 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.

(a) Write a Python function `myfit_reg(alpha,beta,gamma)` that uses regularized least-squares regression to fit a function of the form (2) to the training data using basis functions defined by `alpha` and `beta`, as in Question 4(a). The coefficient of the regularization term is $\gamma =$ `gamma`. The function should return the weight vector, w, the bias term, w_0, and the errors, err_{train} and err_{val}. (The function can be written in at most 14 lines of code, including the 6 lines below.)

You should use the Python class `Ridge` to solve the regularized least-squares problem. It is not provided by Numpy, but by scikit-learn. Specifically, the following code performs ridge regression and computes the optimal weight vector, `w`, and bias term, `w0`:

```
import sklearn.linear_model as lin
ridge = lin.Ridge(gamma)
ridge.fit(Zaug,t)
w = ridge.coef_
w = w[1:]
w0 = ridge.intercept_
```

Here, `t` is the vector of target values, and `Zaug` is the augmented feature matrix, described in Question 4(c). Note that `w0` is given special treatment. This is because it is not included in the regularization term and must be computed somewhat differently.

(b) Use `myfit_reg` to fit a function to the training data using `gamma` $= 10^{-9}$ and 19 basis functions. In defining the basis functions, use $\beta = 1$, and use values of α_m that are equally spaced between `xMin` and `xMax`. Use `plotY` to plot the function. Title the figure, "Question 5(b): the fitted function, 19 basis functions, gamma 10^{-9}". Also, print out the training and validation errors. If you have done everything correctly, the plotted function should be smooth and just a bit bumpier than in Question 4(e). Also, the training error should be about 2.2, and the validation error should be about 5.4.

(c) Repeat part (b) using gamma = 0. The plot should look identical to that of Question 4(g), and the training error should be approximately the same (ie, close to 0).

(d) Define a Python function `best_gamma(alpha,beta)` that uses the validation data to find the best value of the regularization coefficient γ in Equation (6). As before, `alpha` and `beta` specify the basis functions. Your function should do the following:

- Use `myfit_reg(alpha,beta,gamma)` to fit functions to the training data for values of `gamma` spread out over 32 orders of magnitude. Specifically, use values of `gamma` such that $\log_{10}$(`gamma`) = -26, -24, -22, ..., 0, 2, 4.
- 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 γ 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 functions are quite wiggly for small values of `gamma`, almost flat for large values of `gamma`, and fit the data quite well for intermediate values. This illustrates *underfitting* for large values of `gamma`, and *overfitting* for small values of `gamma`. Use the function `suptitle` in `matplotlib.pyplot` to title the entire figure, "Question 5(d): best-fitting functions for log(gamma) = -26, -24, ..., 0, 2, 4". Put the figure into full-screen mode before saving it, to make it large and more readable.
- Plot the training and validation errors versus `gamma`. 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 "gamma", and give the figure the title "Question 5(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 `gamma` increases. The validation error should decrease initially and then begin to increase, reaching a minimum at the optimal value of `gamma`.
- Use `plot(w)` to plot the weights for (i) the smallest value of `gamma` (10^{-26}), (ii) the optimal value of `gamma` and (iii) the largest value of `gamma` (10^4). Title the three plots 'Question 5(d): weights for smallest gamma', 'Question 5(d): optimal weights', 'Question 5(d): weights for largest gamma', respectively. *Describe the main differences in the three curves. Explain how these weights lead to the corresponding plots.*
- Plot the best-fitting function. That is, use `plotY` to plot the function for the value of `gamma` that gives the lowest validation error. In addition, label the axes and give the figure the title, "Question 5(d): best-fitting function (gamma = ??)", filling in the optimal value of `gamma`.
- Print out the optimal values of `gamma` and w_0.
- Compute the test error for the optimal values of `gamma` and w.

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

Note that `best_gamma` requires you to generate a number of plots, most (but not all) using `plotY`. Run `best_gamma` using 19 basis functions. In defining the basis functions, use $\beta = 1$, and use values of α_m that are equally spaced between `xMin` and `xMax`. Hand in all the plots and all the printed values. It should be clear what each piece of output refers to.

6. *Gradient Descent.* (25 points total)

In this question, as in Question 5, you will write a program to 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.

Your program will find the weight vector, w, and bias term, w_0, that minimize the regularized loss function, $\tilde{l}(w, w_0)$, in Equation (6). To do this, you should initialize w and w_0 randomly (*e.g.*, by using `numpy.random.randn`). You should then update them repeatedly using the following two statements:

$$w = w - \lambda \frac{\partial \tilde{l}(w_0, w)}{\partial w}$$

$$w_0 = w_0 - \lambda \frac{\partial \tilde{l}(w_0, w)}{\partial w_0}$$

where the hyper-parameter λ is the learning rate. Executing these two statements is a single iteration of gradient descent. You will need to execute many iterations. You will also have to experiment to find a good learning rate, and you will need to compute the derivatives of $\tilde{l}(w_0, w)$.

(a) (3 points) Derive a matrix/vector expression for $\partial \tilde{l}(w_0, w)/\partial w$ similar to the one given in Question 3(c).

(b) (2 points) Derive a matrix/vector expression for $\partial \tilde{l}(w_0, w)/\partial w_0$ similar to the one given in Question 3(d).

(c) (5 points) Write a Python function `grad_reg(Z,T,w,w0,gamma)` that computes the gradients of the regularized loss function, $\tilde{l}(w, w0)$. Here, using the notation of Questions 3 and 5, `Z` is the feature matrix, `T` $= t$ is the vector of target values, `w` $= w$ is the weight vector, `w0` $= w_0$ is the bias term, and `gamma` $= \gamma$ is the coefficient of the regularization term. Your function should compute and return $\partial \tilde{l}(w_0, w)/\partial w$ and $\partial \tilde{l}(w_0, w)/\partial w_0$ using the formulas you derived in parts (a) and (b), respectively. Do not use any loops. (The function can easily be written in 6 lines of code.)

(d) (15 points) Write a Python program `myfit_reg_gd(alpha,beta,gamma,lrate)` that uses gradient descent to fit a function to the training data. Here, `alpha` and `beta` define the basis functions, `gamma` is the coefficient of the regularization term, and `lrate` is the learning rate. The function should do the following:

- Perform 3,000,000 iterations of gradient descent. (Depending on your computer, this may take about 3 minutes.)
- Compute and record the training and test error after every iteration.
- Use `plotY` to plot the fitted function at nine different time points: after 6^0, 6^1, 6^2, ..., 6^8 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 two plots should be fairly flat with a slight bend in them, the next few plots should be increasingly steep and wavey and should fit the data increasingly well, and the last two plots should appear to fit the data quite well. Title the figure, "Question 6: fitted function as iterations increase". Put the figure into full-screen mode before saving it, to make it large and more readable.
- Use `plotY` to plot the final fitted function in a single figure. Title the figure "Question 6: 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 6: 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 6: 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, and the curve should look flat at the far right end.
- To see that gradient descent has not converged completely even after 3,000,000 iterations, plot the last 1,000,000 training errors (without the test errors). Label the figure, " Question 6: last 1,000,000 training errors". If everything is working correctly, the plot should show the errors decreasing in a curved fashion from left to right.
- Print the final training and test errors after 3,000,000 iterations.
- To check your program, refit the function using `myfit_reg` from Question 5(a). Compute and print the training and test errors for this function (not the validation error). If everything is working correctly, they should be about the same as for gradient descent.
- compute and print the difference in the training errors for gradient descent and `myfit_reg`. If everything is working correctly, the difference should be less than 0.0003.
- Print 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. (When developing your function and searching for a good learning rate, try printing out the training error every 10,000 or 100,000 iterations, but do not hand this in.)

Run your function using 19 basis functions with `beta` $= 1$ and with the α_m spaced uniformly between `xMin` and `xMax`. Also use $\gamma = 0.001$. You will probably find that your function produces a reasonably good fit to the data after 1,000,000 iterations. However, you will need many more iterations to reach the accuracy of `myreg_fit`. If everything is working correctly, you should find that the training error is below 7 after 100,000 iterations, and below 4.7 after 1,000,000 iterations.

Note that `myfit_reg_gd` 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).

137 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: _____