University of Toronto Mississauga
Department of Mathematical and Computational Sciences
CSC 411 - Machine Learning and Data Mining, Fall 2017

Assignment 1

Due date: Tuesday October 10, 11pm.
No late assignments will be accepted.

The material you hand in should be legible (either typed or *neatly* hand-written), well-organized and easy to mark, including the use of good English. In general, short, simple answers are worth more than long, complicated ones. Unless stated otherwise, all answers should be justified.

All computer problems are to be done in Python with the NumPy, SciPy and scikit-learn libraries. Your programs should be properly commented and easy to understand. Hand in all the programs you are asked to write. They should be stored in a single file called `source.py`. We should be able to import this file as a Python module and execute your programs. For generating plots, you may find the SciPy functions `plot` and `figure` in `matplotlib.pyplot` useful, as well as the function `numpy.linspace`. 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 can use `numpy.matmul(A,B)`. Also, whenever possible, *do not use loops*, which are very slow in Python. Instead, use Numpy's vector and matrix operations, which are much faster and can be executed in parallel. If you haven't already done so, please read the NumPy tutorial on the course web page.

You should hand in four files: the source code of all your Python programs, a pdf file of all the requested program output, answers to all the non-programming questions (scanned hand-writing is fine), 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. The four 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.

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.

1. 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 `mymult(A,B)` that multiplies two matrices, `A` and `B`. Recall that if `C` is the product matrix, then the ij^{th} element of `C` is given by

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

   Your function 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 B) and `zeros` (to initialize C).

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

   - Use the function `rand` in `numpy.random` to create two random matrices, A and B, of size $I \times K$ and $K \times J$, respectively.
   - Use the function `numpy.matmul` to multiply A and B. Call the result $C1$.
   - Use the function `time.time` to measure the execution time of `numpy.matmul`.
   - Print out the execution time.
   - Use your function `mymult` to multiply A and B. Call the result $C2$
   - Use the function `time.time` to measure the execution time of `mymult`.
   - Print out the execution time.
   - Compute and print out the magnitude of the difference between $C1$ and $C2$:

     $$\sum_{ij}(C1_{ij} - C2_{ij})^2$$

     Do *not* use iteration to compute this magnitude. Instead, use only NumPy operations. You may find `numpy.sum` useful.

   If your function `mymult` is working correctly, the last step should produce a very small number. You should also find that `numpy.matmul` is much faster than `mymult`.

   (c) Run `mymeasure(1000,50,100)` and `mymeasure(1000,1000,1000)`, 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 `mymeasure` perform.

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.

2. *Linear Regression: theory.*

In the rest of this assignment, you will fit a polynomial to data using linear least-squares regression. That is, you will estimate the weights $w = (w_0, w_1, ..., w_M)$ in the function

$$y(x) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M$$

In particular, you will find the w that minimizes the loss function,

$$l(w) \;=\; \sum_n [t^{(n)} - y(x^{(n)})]^2$$

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.

In this question, you will formulate and solve the linear least-squares problem analytically in terms of matrices and vectors. Central to this is the *data matrix*, Z, which has N rows and $M+1$ columns, that is, one row for each training point and one column for each weight. The n^{th} row consists of powers of $x^{(n)}$. Specifically, $Z_{nm} = [x^{(n)}]^m$ for $m = 0, 1, ..., M$. Note that the first column of Z consists entirely of 1's. In addition, let $t = [t^{(1)}, t^{(2)}, ..., t^{(N)}]$ be the vector of target values, and let $\hat{y} = [y(x^{(1)}), y(x^{(2)}), ..., y(x^{(N)})]$ be the vector of predicted target values. In this assignment, most vectors, including t, $\hat{y}$ and w, are treated as column vectors.[1]

(a) Prove that $\hat{y} = Zw$. Thus, once the data matrix, Z, is computed, the polynomial can be evaluated at each data point by doing matrix multiplication.

(b) Prove that $l(w) = \|Zw - t\|^2$, where in general, $\|v\|$ denotes the magnitude of a vector, v. That is, if $v = (v_1, ..., v_N)$, then $\|v\|^2 = \sum_n v_n^2$.

(c) Prove that

$$\frac{\partial l(w)}{\partial w_m} \;=\; -2\sum_n [t^{(n)} - y(x^{(n)})] Z_{nm}$$

(d) Prove that

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

Hint: recall that t and $\hat{y}$, as well as $\partial l(w)/\partial w$, are column vectors.

(e) By setting $\partial l(w)/\partial w = 0$, we can solve for the w that minimizes $l(w)$. This is slightly complicated because Z is not in general a square matrix and therefore has no inverse. However, $Z^T Z$ *is* a square matrix, of dimensions $M+1 \times M+1$. Moreover, it is almost always invertible as long as $n >= M+1$, *i.e.*, as long as there is at least one training point for each weight we are trying to learn. With this in mind, prove that the optimal value of w is given by $w = (Z^T Z)^{-1} Z^T t$.

[1]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.

3. *Linear Regression: practice.*

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

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

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 below.

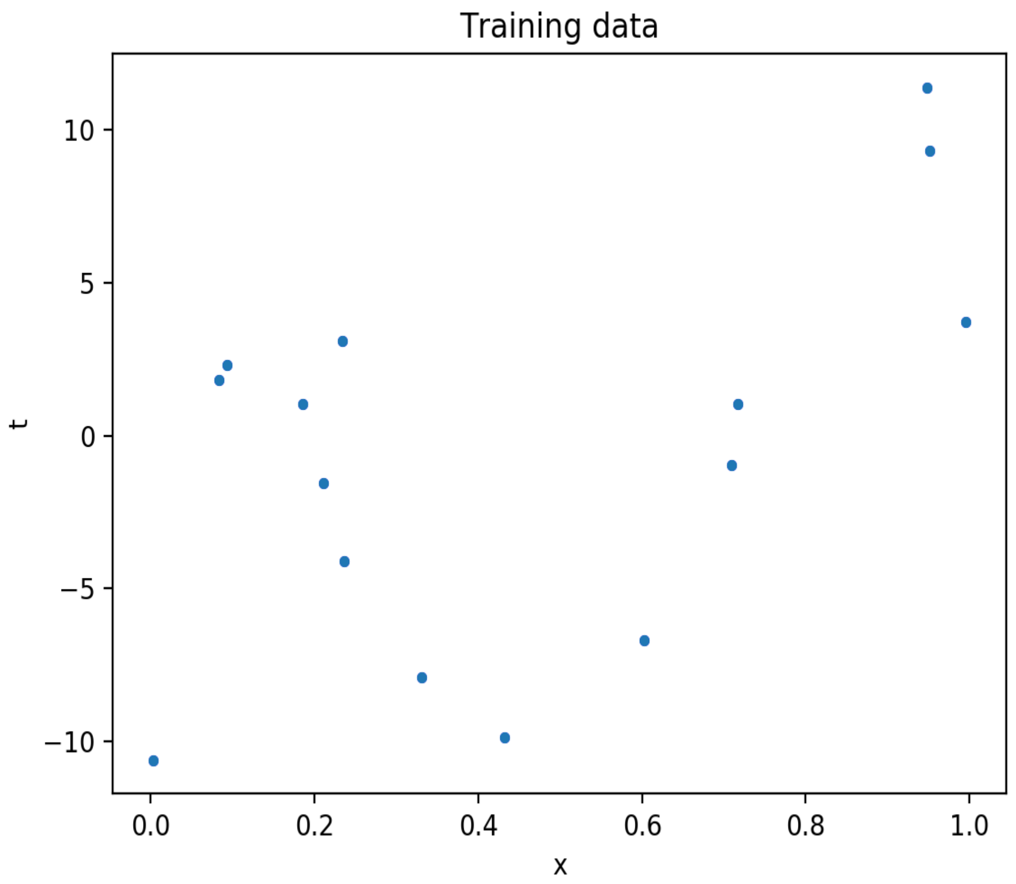
In this question and in the rest of the assignment, to evaluate a polynomial at a set of points, you should compute the data matrix, Z, and use the results of Question 2(a). In particular, do *not* use the function `numpy.polyval` or anything similar.

(a) Write a Python function `dataMatrix(X,M)` that computes and returns a data matrix, `Z`, for the problem of fitting a polynomial of degree M to data set `X`. Here, `X` $= [x^{(1)}, x^{(2)}, ..., x^{(N)}]$ is a vector of real numbers.

(b) Write a Python function `fitPoly(M)` that fits a polynomial of degree M to the training data using linear least-squares. The function should return the weight vector, w, and the errors, err_{train} and err_{test}. You should use the method `lstsq` in `numpy.linag` to solve the linear least-squares problem. That is, `lstsq(Z,t)` computes the value of w that minimizes $\|Zw - t\|^2$. `lstsq` returns a number of objects. The first one is the weight vector, w. That is $w = result[0]$ where $result =$ `lstsq`(Z, t), or simply $w =$ `lstsq`$(Z, t)[0]$. To compute the errors, you may find the functions `numpy.sum` and `numpy.mean` useful.

(c) Write a function `plotPoly(w)` that plots the polynomial $y(x)$ defined by w as a red curve. Construct the plot using 1000 equally spaced values of x between 0

Figure 1:



and 1. The function should also plot the training points (as blue dots) in the same figure. Use the function `plot` in `matplotlib.pyplot` to do the plotting and the function `ylim` to limit the y-axis to values between 15 and -15. You may also find the function `numpy.linspace` useful.

(d) Write a function `bestPoly` that finds the degree of the polynomial that best fits the data. Specifically, this function should do the following:

- Use `fitPoly(M)` to fit polynomials to the training data for M = 0,1,2,...,15.
- Use `plotPoly` to plot each of these fitted polynomials. Use `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. You should find that the polynomials are straight lines for M = 0,1, then gradually become more curvy as M increases, fit the data pretty well for some intermediate values of M, then become more and more wiggly as M increases even more, becoming totally wild when M is near 15, and pass through every data point exactly when M = 15. This illustrates *underfitting* for small values of M, and *overfitting* for large values of M.
- 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 3: 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 polynomial. That is, use `plotPoly` to plot the polynomial for the value of M that gives the lowest test error. In addition, label the axes and give the figure the title "Question 3: best-fitting polynomial (degree = M)", 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 `bestPoly` requires you to generate a number of different plots, most (but not all) using `plotPoly`. Hand in all the plots and all the printed values. It should be clear what each piece of output refers to.

4. *Regularization.*

In this problem you will write a Python program to fit a high-degree polynomial 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 the function

$$y(x) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M$$

that minimizes the loss function

$$l(w) \;=\; \sum_n [t^{(n)} - y(x^{(n)})]^2 + \alpha \sum_{j=1}^{M} w_j^2$$

where the first sum is over all training points, $(x^{(n)}, t^{(n)})$. The second sum 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 polynomial, $y(x)$, 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 Question 2. 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 2, 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.

(a) Write a Python function `fitRegPoly(M,alpha)` that fits a polynomial of degree M to the training data using regularized least-squares where the weight 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. 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(Z,t)
w = ridge.coef_
w[0] = ridge.intercept_
```

Here, `Z` is the data 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) Define a Python function `bestRegPoly()` that finds the polynomial of degree 15 that best fits the data with regularization. In particular, this function should do the following:

- Use `fitRegPoly(M,alpha)` to fit polynomials to the training data for M = 15 and for values of `alpha` spread out over 16 orders of magnitude. Specifically, use values of `alpha` such that $\log_{10}$(`alpha`) = -13, -12, -11, ..., 1, 2.
- Use `plotPoly` to plot each of these fitted polynomials. 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 polynomials are very 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`.
- 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 4: 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 tend to decrease initially and then begin to increase, reaching a minimum at the optimal value of `alpha`.
- Plot the best-fitting polynomial. That is, use `plotPoly` to plot the polynomial for the value of `alpha` that gives the lowest validation error. In addition, label the axes and give the figure the title "Question 4: best-fitting polynomial (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 `bestRegPoly` requires you to generate a number of different plots, most (but not all) using `plotPoly`. Hand in all the plots and all the printed values. It should be clear what each piece of output refers to.

5. *Gradient Descent.*

In this question, as in Question 4, you will fit a polynomial 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 4.

(a) Prove that for $m \geq 1$,

$$\frac{\partial l(w)}{\partial w_m} = 2\alpha w_m - 2\sum_n [t^{(n)} - y(x^{(n)})] Z_{nm}$$

(b) Write down a formula for $\partial l(w)/\partial w_0$. Justify your answer without giving a proof.

(c) Write a Python program `regGrad(Z,t,w,alpha)` that computes and returns the gradient of the regularized loss function, *i.e.*, the vector $\partial l(w)/\partial w$. Here, `Z` is a data matrix, `t` is a vector of target values, `w` is a weight vector, and `alpha` is the coefficient of the regularization term.

(d) Write a Python program `fitPolyGrad(M,alpha,lrate)` that uses gradient descent to fit a polynomial of degree `M` to the training data. 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

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

where $\partial l(w)/\partial w = [\partial l(w)/\partial w_0,\ \partial l(w)/\partial w_1,\ \cdots\ \partial l(w)/\partial w_M]$. The hyperparameter 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 `M` = 15 and the optimal value of `alpha` that you found in Question 4. Your program should then compute the same value of `w`, and the same training and test error, as in Question 4. You will probably find that your function produces a reasonably good fit to the data after one million iterations. However, you will need many more iterations to reach the accuracy of the answer in Question 4. In addition, your function should do the following:

- Perform 10,000,000 iterations of gradient descent. (This may take a long time, so you may want to print something every 10,000 iterations, to monitor the execution.)
- Compute and record the training and test error every 1000 iterations. You may find the function `numpy.mod` useful.

- Plot the recorded training and test errors. 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 5: training and test error v.s. time". If everything is working correctly, the training error should decrease as the number of iterations increases.
- Use `plotPoly` to plot the fitted polynomial at eight different time points: after 10^0, 10^1, 10^2, ..., 10^7 iterations. Use `subplot` to arrange all eight plots on a 3x3 grid in a single figure, with iterations increasing from left to right and top to bottom.
- Use `plotPoly` to plot the final fitted polynomial in a single figure. Title the figure "Question 5: fitted polynomial".
- 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 4. (If not, try using a higher learning rate.)

Note that `fitPolyGrad` requires you to generate a number of different plots, most (but not all) using `plotPoly`. Hand in all the plots and all the printed values. It should be clear what each piece of output refers to.

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: Sayantan Chattopadhyay
(Underline your last name)

Student number: 1000818145

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: [illegible]