EE512: Machine Learning

Homework 1

Due 7:04pm, 22nd April, 2020

1. Use A4 papers for handwritten answers. Scan your responses and submit pdfs. Name your documents like "Q1.pdf ", "Q2.pdf " etc.
2. Make sure to read the questions carefully before attempting them.
3. You are allowed to discuss with fellow students and with TAs for general advice, but you must submit your own work.
4. Plagiarism will be not tolerated.
5. The write-ups must be very clear, to-the-point, and presentable. We will not just be looking for correct answers rather highest grades will be awarded to write-ups that demonstrate a clear understanding of the material. Write your solutions as if you were explaining your answer to a colleague. Style matters and will be a factor in the grade.
6. Submit jupyter notebooks titled as "Q1.ipynb" , "Q2.ipynb" and "Q5.ipynb" in google classroom. You may use jupyter notebook "markdown" cells to give comments on your results. All notebooks and pdfs should be in a folder. Name this folder with your roll number e.g. HW2 MSDS19001

Q.1) **Document Classification:**

In this problem you will explore the use of Naive Bayes classification applied to a classic text processing problem. Specifically one of the first usages of the Naive Bayes approach concerned what is known as the author attribution problem. Here we will tackle a particularly famous instance: who wrote the Federalist Papers.

Federalist Papers were a series of essays written in 1787-1788 meant to persuade the citizens of the state of New York to ratify the Constitution and which were published anonymously under the pseudonym "Publius". In the later years authors were revealed as Alexander, Hamilton, John Jay and James Madison. However there is disagreement as to who wrote which essays. Hamilton wrote a list of which essays he had authored only days before being killed in duel with then Vice President

Aaron Burr. Madison wrote his own list many years later which is in conflict with Hamilton's list on 12 of the essays. Since by this point the two (who were close friends) had become bitter rivals, historians have long been unsure as to the reliability of both lists.

We will try to settle this dispute using simple Naive Bayes classifier. You will need to download the documents which are in the file **fedpapers_split.txt** as well as some starter code in **fedpapers.py**, both attached with the Homework. The file **fedpapers.py** load the documents and build a "bag of words" representation of each document. Your task is to complete the missing portions of the code and determine your best guess as to who wrote each of the twelve disputed essays.

Submit your code along with your answers as to how many essays you think were written by Hamilton and how many were by Madison. (Note: There isn't actually a verifiably correct answer here, but there is an answer that has gained broad acceptance among historians.)

10 marks

Q.2) **Gradient Descent:**

In this problem you will explore implementation of three different solvers for logistic regression. To get started download **lr-gd.py** attached with Homework.

(a) We will begin by implementing logistic regression using standard *gradient descent* for performing the maximum likelihood estimation step. Review the lecture slides and make sure that you understand what **lr-gd.py** is doing. Test out the file by experimenting a bit with various parameters — especially the "step size" or learning rate α — to obtain a good convergence rate. I would recommend trying a wide variety starting with $\alpha \approx 1$ and ranging to $\alpha \ll 1$. You should feel free to play around with stopping criteria than those provided. For this problem, all you need to do is to report the value of α you used and number of iterations required for convergence for those parameters.

(b) Using the notes, adapt **lr-gd.py** to implement Newton's method for solving this problem. This method requires computing both the gradient and the Hessian each iteration.

$$\frac{\Delta^2 l(\theta)}{\partial \theta} = -\sum_{i=1}^{N} \tilde{x_i}\tilde{x_i}^T g(\theta^T \tilde{x_i})(1 - g(\theta^T \tilde{x_i}))$$

where g is defined as in the notes.(You may verify this by yourself but you don't need to do.) [Take extra care in implementation making sure that you have the correct signs in your algorithm — remember we are trying to minimize the *negative log-likelihood*. If you algorithm did not converges, this is a likely suspect. Note also that in the notes I assume that the $\tilde{x_i}$ are the column vectors, but in python code they are treated as row vectors.]

Just as reminder in python A*B computes the element-wise multiplication, while A.dot(B) computes the matrix multiplication. You may also wan't to compute a matrix inverse of A using np.linalg.inv(A).

(c) We will implement yet another variant of *gradient descent* called *stochastic gradient descent* to perform the optimization step in logistic regression. In standard gradient descent, in order to compute the gradient we must compute the sum

$$\sum_{i=1}^{n} \tilde{x_i}(y - g(\theta^T \tilde{x_i})) \tag{1}$$

In the dataset we're using here, this is not much of a challenge, but if our dataset is extremely massive (i.e. if n is extraordinarily large) then even simply computing the gradient can be computationally intractable. One possibility in this case in known as *stochastic gradient descent*, and consist of simply selecting on $\tilde{x}_i$ at random and treating $\tilde{x}_i(y - g(\theta^T \tilde{x}_i))$ as rough approximation to (1) and proceeding with the standard gradient descent approach as before. This will in general require more iterations, but each iteration can be more cheaper when n is large, so can be very effective when dealing with large datasets. Implement a version of stochastic gradient descent by writing a new function to replace **grad_desc** in **lr-gd.py**. Submit you code with assignment and report number of iterations required for convergence. Compute this result from part (a). You may find the command **np.random.permutation(n)** to be of use.

(d) Finally, try computing the results from part (a),(b) and (c) applied to much larger dataset. Try changing n_samples to 1000 000 . You will want to comment out the plotting portion of the code for this part(or at least the part that plots the training data). Record running time and number of iterations for all three algorithms when applied to this dataset. Note that in order to run traditional gradient descent on this large dataset, you will need to make α much smaller (think about why this would be the case). If you see warnings regarding divided by zero, you have set α to be too large. Also note that you can get the system time in Pyhton bu adding **import time** to your file and then using the **time.time()** command.

10 marks

Q.3) **Loss Functions:**

Suppose we have a prediction problem where the target t corresponds to an angle, measured in radians. A reasonable loss function we might use is

$$\mathcal{L}(y, t) = 1 - \cos(y - t)$$

Suppose we make predictions using a linear model

$$y = \boldsymbol{x}\boldsymbol{w} + b$$

where $\boldsymbol{w}$ is D-dimensional column vector and $\boldsymbol{x}$ is D-dimensional row vector.

As usual cost is average loss over training set.

$$\mathcal{E} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y^{(i)}, t^{(i)})$$

Derive a sequence of vectorized mathematical expressions for the gradient of the cost with respect to $\boldsymbol{w}$ and b. Assume that inputs are arranged in a $N \times D$ design matrix $\boldsymbol{X}$ with one row per training example i.e. there are N examples and each example is D-dimensional row-vector. The expressions should be something you can translate to Python program without requiring a for-loop.

You answer should be like this:

$$\boldsymbol{y} = \ldots$$

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{y}} = \dots$$
$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} = \dots$$
$$\frac{\partial \mathcal{E}}{\partial b} = \dots$$

Here $\boldsymbol{y}$ is N-dimensional column vector. You can use $\sin(\boldsymbol{A})$ to denote sin function applied elementwise to $\boldsymbol{A}$. Also remember that $\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}}$ can be written as follows;

$$\boldsymbol{w} = [w_1, w_2, \dots, w_D]^T$$

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} = \begin{bmatrix} \frac{\partial \mathcal{E}}{\partial w_1} \\ \frac{\partial \mathcal{E}}{\partial w_2} \\ \dots \\ \frac{\partial \mathcal{E}}{\partial w_D} \end{bmatrix}$$

5 marks

Q.4) **Backprop:**

Consider a neural network with N input units, N output units and K hidden units. The activations are computed as follows:

$$\boldsymbol{z} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$$

$$\boldsymbol{h} = \sigma(\boldsymbol{z})$$
$$\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}$$

where σ denotes the logistic function applied elementwise. The cost will involve both $\boldsymbol{h}$ and $\boldsymbol{y}$:

$$\mathcal{E} = \mathcal{R} + \mathcal{S}$$

$$\mathcal{R} = \boldsymbol{r}^T\boldsymbol{h}$$

$$\mathcal{S} = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{s}\|_2$$

for given $\boldsymbol{r}$ and $\boldsymbol{s}$ vectors.

- Draw the computation graph relating $\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{h}, \boldsymbol{y}, \mathcal{R}, \mathcal{S}, \mathcal{E}$
- Derive backprop equations for computing $\bar{x} = \frac{\partial \mathcal{E}}{\partial \boldsymbol{x}}$. You may use σ' to denote the derivative of logistic function (so you don't need to write it explicitly.)

10 marks

Q.5) **Implementation of Neural Network for Regression:**

In this problem, you are required to do regression using 3-layer neural network $f_\theta(.)$ where θ represents the set of learnable parameters. Input feature vector $\boldsymbol{x} \in \mathbb{R}^2$ is 2-dimensional and target variable $y \in \mathbb{R}$ is a scaler formed by a non-linear combination of input features. $\boldsymbol{x} = [x_1, x_2]^T$

$$y = x_1^2 + x_2^3 + x_1 x_2$$

Goal is to train a neural network using $(\boldsymbol{x}, y)$ pairs such that, after training, whenever we give an unseen $\boldsymbol{x}_{test}$ to neural network, it should return $\hat{y} = f_\theta(\boldsymbol{x}_{test})$ that is close enough to true value y_{test}. Training data and starter code is attached with homework.

You are **not allowed** to use high-level APIs like Keras, Pytorch or Tensorflow etc. Use **numpy** package to implement given neural network.

Tasks:

1. Implement feed forward using equations given below.

$$z_1 = W_1 x + b_1$$

$$h_1 = \sigma(z_1)$$

$$z_2 = W_2 h_1 + b_2$$

$$h_2 = \sigma(z_2)$$

$$\hat{y} = W_3 h_2 + b_3$$

$$loss = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

where N is batch size.

2. Derive expressions for gradients of learnable parameters with respect to loss of neural network using backpropagation on paper. We are using sigmoid as activation function.
3. Implement backpropagation using results from 2nd step.
4. Using training data and mean-squared-error (MSE) loss, train neural network.
5. Using test data x_{test}, predict $\hat{y}_{test}$
6. Show predicted $\hat{y}_{test}$ and ground truth y_{test} on same plot.

Question 5 continues on the next page...

7. Report $\|\hat{\boldsymbol{y}}_{test} - \boldsymbol{y}_{test}\|_2$.

25 marks
