CSCI-SHU360 Machine Learning Homework 3
Due: Oct 30th 11:59 PM, 2024 (GMT+8)
100 points

Instructions

- **Collaboration policy:** Homework must be done individually, except where otherwise noted in the assignments. "Individually" means each student must hand in their own answers, and you must write and use your own code in the programming parts of the assignment. It is acceptable for you to collaborate in figuring out answers and to help each other solve the problems, and you must list the names of students you discussed this with. We will assume that, as participants in an undergraduate course, you will be taking the responsibility to make sure you personally understand the solution to any work arising from such collaboration.
- **Online submission:** You must submit your solutions online on the course **Gradescope** site (you can find a link on the course Brightspace site). You need to submit (1) a PDF that contains the solutions to all questions to the Gradescope **HW3 PaperWork** assignment (**including the questions asked in the programming problems**), (2) `x.py` or `x.ipynb` files for the programming questions to the Gradescope **HW3 CodeFiles** assignment. We recommend that you type the solution (e.g., using LaTeX or Word), but we will accept scanned/pictured solutions as well (clarity matters).
- **Generative AI Policy:** You are free to use any generative AI, but you are required to document the usage: which AI do you use, and what's the query. You are responsible for checking the correctness.
- **Late Policy: No late submission is allowed.**

Before you start: this homework only has programming problems. You should still have all questions answered in the write-up pdf. Also note that some sub-problems are still essentially math problems and you need to show detailed derivations.

1 Programming Problem: Logistic Regression [40 points]

In this problem, you will implement the gradient descent algorithm and mini-batch stochastic gradient descent algorithm for multi-class logistic regression from scratch (which means that you cannot use built-in logistic regression modules in scikit-learn or any other packages). Then you will be asked to try your implementation on a hand-written digits dataset to recognize the hand-written digits in given images.

We provide an ipython notebook "logistic_regression_digits.ipynb" for you to complete. You need to complete the scripts below "TODO" (please search for every "TODO"), and submit the completed ipynb file. In your write-up, you also need to include the plots and answers to the questions required in this session.

In this problem, you will implement a logistic regression model for multi-class classification. Given features of a sample x, a multi-class logistic regression produces class probability (i.e., the probability of the sample belonging to class k)

$$\Pr(y = k|x; W, b) = \frac{\exp(xW_k + b_k)}{\sum_{j=1}^{c} \exp(xW_j + b_j)} = \frac{\exp(z_k)}{\sum_{j=1}^{c} \exp(z_j)}, \quad \forall\ k = 1, 2, \cdots, c \tag{1}$$

where c is the number of all possible classes, model parameter W is a $d \times c$ matrix and b is a c-dim vector, and we call the c values in $z = xW + b$ as the c logits associated with the c classes. Binary logistic regression model ($c = 2$) is a special case of the above multi-class logistic regression model (you can figure out why by yourself with a few derivations). The predicted class $\hat{y}$ of x is

$$\hat{y} = \arg \max_{k=1,2,\cdots,c} \Pr(y = k|x; W, b). \tag{2}$$

For simplicity of implementation, we can extend each x by adding an extra dimension with value 1, i.e., $x \leftarrow [x, 1]$, and accordingly add an extra row to W, i.e, $W \leftarrow [W; b]$.

After using the extended representation of x and W, the logits z become $z = xW$. Logistic regression solves the following optimization for maximum likelihood estimation.

$$\min_W F(W) \text{ where } F(W) = \frac{1}{n} \sum_{i=1}^{n} -\log[\Pr(y = y_i|x = X_i; W)] + \frac{\eta}{2}\|W\|_F^2, \tag{3}$$

where we use a regularization similar to the ℓ_2-norm in ridge regression, i.e., the Frobenius norm $\|W\|_F^2 = \sum_{j=1}^{c} \|W_{\cdot,j}\|_2^2 = \sum_{j=1}^{c} \sum_{i=1}^{d} (W_{i,j})^2$. Note that $W_{\cdot,j}$ is the j^{th} column of W.

1. *[15 points]* Derive the gradient of $F(W)$ w.r.t. W, i.e., $\nabla_W F(W)$, and write down the gradient descent rule for W. Hint: compute the gradient w.r.t. to each W_j for every class j first.

2. *[10 points]* Below "2 batch gradient descent (GD) for Logistic regression", implement the batch gradient descent algorithm with a constant learning rate. To avoid numerical problems when computing the exponential in the probability $\Pr(y = k|x; W, b)$, you can use a modification of the logits z', i.e.,

$$z' = z - \max_j z_j. \tag{4}$$

   Explain why such a modification could avoid potential numerical problems and show that the overall result is unchanged by applying such a trick.

   When the change of objective $F(W)$ comparing to $F(W)$ in the previous iteration is less than $\epsilon = 1.0e-4$, i.e., $|F_t(W) - F_{t-1}(W)| \leq \epsilon$, stop the algorithm. Please record the value of $F(W)$ after each iteration of gradient descent.

   Please run the implemented algorithm to train a logistic regression model on the randomly split training set. We recommend to use $\eta = 0.1$. Try three different learning rates $[5.0e-3, 1.0e-2, 5.0e-2]$, report the final value of $F(W)$ and training/test accuracy in these three cases, and draw the three convergence curves (i.e., $F_t(W)$ vs. iteration t) in a 2D plot.

3. *[5 points]* Compare the convergence curves: what is the advantages and disadvantages of large and small learning rates?

4. *[5 points]* Below "4 stochastic gradient descent (SGD) for Logistic regression", implement the mini-batch stochastic gradient descent (SGD) for logistic regression. You can reuse some code from the previous gradient descent implementation.

   Tuning hyper-parameters is critical to get good models. For the mini-batch SGD algorithm, the hyper-parameters consist of 1) the initial learning rate, 2) the learning rate schedule, or how do we change the learning rate over time, 3) the number of epochs to train (an epoch corresponds to training the model with every data point once; though theoretically SGD samples data points with replacement, in practice, the widely adopted approach is to random shuffle the dataset to form mini-batches, let the model learn from all data points once, and repeat until convergence), and 4) the mini-batch size.

   We suggest using the following automatic Tuning for the learning rate schedule and number of epochs. For the learning rate schedule, record the objective $F(W)$ over epochs, and if the objective has not become better than 10 epochs ago, half the learning rate. For the number of epochs, or stopping

criteria, continue training until $F(W)$ has not get better than 20 epochs ago.

You can start by using an initial learning rate of $1.0e-2$ and a mini-batch size of 100 for this problem. You can discard the last mini-batch of every epoch if it is not full. Please remember to record the value of $F(W)$ after each epoch and the final training and test accuracy.

Run your code for different mini-batch sizes: [10, 50, 100]. Report the final value of $F(W)$ and final training/test accuracy, and draw the three convergence curves ($F_t(W)$ vs. epoch t) in a 2D plot.

5. *[5 points]* Compare the convergence curves: do they (logistic regression with the three different batch sizes) show the same convergence speed, when the same initial learning rate is used? For different batch sizes, you may need to tune the initial learning rate. In general, the rule of thumb is to scale the learning rate linearly with the batch size. Please draw the new convergence curves after tuning the learning rate in a 2D plot. What is the difference compared to the old convergence curves? Can you give some mathematical explanations based on the SGD you implemented?

2 Programming Problem: Lasso [60 points]

Before you start: This problem could be time-consuming and please start early! It may take **several hours** to train your model depending on your implementation.

We provide an ipython notebook "lasso.ipynb" for you to complete. In your terminal, please go to the directory where this ipynb file is located, and run command "jupyter notebook". A local webpage will be automatically opened in your web browser. Click the above file to open the notebook. You may also use an IDE such as vscode.

You need to complete everything below each of the "TODO"s that you find (please search for every "TODO"). Once you have done that, please submit the completed ipynb file as part of your included .zip file.

In your write-up, you also need to include the plots and answers to the questions required in this session. Please include the plots and answers in the pdf file in your solution.

Recall that for lasso, we aim to solve:

$$\text{argmin}_{\theta,\theta_0} F(\theta,\theta_0) \quad \text{where} \quad F(\theta,\theta_0) = \frac{1}{2}\sum_{i=1}^{n}(\langle x^{(i)},\theta\rangle + \theta_0 - y^{(i)})^2 + \lambda\sum_{j=1}^{m}|\theta_j|. \tag{5}$$

where λ is the hyperparameter to control the regularization. Here, $x^{(i)}$ is an m-dimensional data point with the corresponding label $y^{(i)}$, θ is the weight vector of m dimensions and θ_0 is a scalar offset. Note that the objective is slightly different from the one given in the lecture, with an extra $1/2$ in front of the MSE.

Remarks: Do not include θ_0 in the computation of precision/recall/sparsity. However, do not forget to include it when you compute the prediction produced by the lasso model, because it is one part of the model.

1. *[15 points]* Implement the coordinate descent algorithm to solve the lasso problem in the notebook.

   We provide a function "DataGenerator" to generate synthetic data in the notebook. Please read the details of the function to understand how the data are generated. In this problem, you need to use $n = 50, m = 75, \sigma = 1.0, \theta_0 = 0.0$ as input augments to the data generator. Do not change the random seed for all the problems afterwards.

   Stopping criteria of the outer loop: stop the algorithm when either of the following is fulfilled: 1) the number of steps exceeds 100; or 2) no element in θ changes more than ϵ between two successive iterations of the outer loop, i.e., $\max_j |\theta_j(t) - \theta_j(t-1)| \leq \epsilon$, where the recommended value for $\epsilon = 0.01$, where $\theta_j(t)$ is the value of θ_j after t iterations.

   At the beginning of the lasso, use the given initialization function "$\theta, \theta_0 = \text{Initialw(X, y)}$" to initialize θ and θ_0 by the least square regression or ridge regression.

You can try different values of λ to make sure that your solution makes sense to you (Hint: DataGenerator gives the true θ and θ_0).

Solve lasso on the generated synthetic data using the given parameters and report indices of non-zero weight entries. Plot the objective value $F(\theta, \theta_0)$ v.s. coordinate descent step. The objective value should always be non-increasing.

2. *[5 points]* Implement an evaluation function in the notebook to calculate the precision and recall of the non-zero indices of the lasso solution with respect to the non-zero indices of the true vector that generates the synthetic data. Precision and recall are useful metrics for many machine learning tasks. For this problem in specific,

$$\text{precision} = \frac{|\{\text{non-zero indices in } \hat{\theta}\} \cap \{\text{non-zero indices in } \theta^*\}|}{|\{\text{non-zero indices in } \hat{\theta}\}|}; \tag{6}$$

$$\text{recall} = \frac{|\{\text{non-zero indices in } \hat{\theta}\} \cap \{\text{non-zero indices in } \theta^*\}|}{|\{\text{non-zero indices in } \theta^*\}|}, \tag{7}$$

where θ^* is the θ in true model weight, while $\hat{\theta}$ is the θ in lasso solution.

You also need to report the sparsity (the number of nonzero entries) of $\hat{\theta}$, and the RMSE of the training data (you may check the definition from HW2 Prob 5). Note that a solution can have high precision with low recall (e.g., the solution contains only one correct non-zero index while the true vector contains many) and vice versa. Report the precision and recall of your lasso solution from the previous problem.

3. *[10 points]* Vary λ and solve the lasso problem multiple times. Choose 50 evenly spaced λ values starting with $\lambda_{max} = \|(y - \bar{y})X\|_\infty$ ($\bar{y}$ is the average of elements in y, and $\|a\|_\infty = \max_j a_j$), and ending with $\lambda_{min} = 0$. Plot the precision v.s. λ and recall v.s. λ curves on a single 2D plot. Briefly explain the plotted pattern and curves. On top of this, try to have fun with λ and play with this hyperparameter, explore, discover, and tell us what you have discovered.

   Draw a "lasso solution path" for each entry of θ in a 2D plot. In particular, use λ as the x-axis, for each entry θ_i in θ achieved by lasso, plot the curve of θ_i vs. λ for all the values of λ you tried similar to the plot we showed in class from the Murphy text (in your case, there are 50 points on the curve). Draw such curves for all the m entries in θ within a 2D plot, use the same color for the 5 features in DataGenerator used to generate the data, and use another very noticeably distinct color for other features. If necessary, set a proper ranges for x-axis and y-axis, so you can see sufficient detail.

   Now change the noise's standard deviation $\sigma = 10.0$ when using "DataGenerator" to generate synthetic data, draw the lasso solution path again. Compare the two solution path plots with different σ, and explain their difference. Be complete, and clear.

4. *[5 points]* Use the synthetic data generation code with different parameters: $(n = 50, m = 75), (n = 50, m = 150), (n = 50, m = 1000), (n = 100, m = 75), (n = 100, m = 150), (n = 100, m = 1000)$ (keeping other parameters the same as in Sub-Problem 1). Vary λ in the same way as in the previous question (Sub-Problem 3), and find the λ value that can generate both good precision and recall for each set of synthetic data points.

   For each case, draw the "lasso solution path" defined in Sub-Problem 3.

5. *[25 points]* **This question is challenging, requiring major changes to your previous implementation as well as significant training time.** Run lasso to predict review stars on Yelp by selecting important features (words) from review comments. We provide the data in hw3_data.zip. You can unzip the file and use the provided function "DataParser" in the notebook to load the data. There are three files: "star_data.mtx", "star_labels.txt", "star_features.txt". The first file stores a matrix market matrix and DataParser reads it into a scipy csc sparse matrix, which is your data matrix X. The second file contains the labels, which are the stars of comments on Yelp, is your y. The third file contains the names of features (words). For the last two txt files, you can open them in an editor and take a look at their contents.

The sparse data X has size 45000×2500, and is split into the training set (the first 30000 samples), validation set (the following 5000 samples) and the test set (the last 10000 samples) by "DataParse". Each column corresponds to a feature, i.e., a word appearing in the comments. Your mission is to solve lasso on the training set, tune the λ value to find the best RMSE on the validation set, and evaluate the performance of the obtained lasso model on the test set.

Important to read before you start: Here, we are dealing with a sparse data matrix. Most numpy operations for dense matrices you used for implementing lasso in Sub-Problem 1 cannot be directly applied to sparse matrices here. You can still use the framework you got in Sub-Problem 1, but you need to replace some dense matrix operations (multiply, dot, sum, slicing, etc.) by using sparse matrix operations from "scipy.sparse" (please refer to `https://docs.scipy.org/doc/scipy/reference/sparse.html` for details of sparse matrix operations).

The sparse matrix format here aims to help you make the algorithm more efficient in handling sparse data. Do not try to directly transform the sparse matrix X to a dense one by using "X.todense()", since it will waste too much memory. Instead, try to explore the advantages of different sparse matrix types (csc, coo, csr) and avoid their disadvantages, which are listed under each sparse matrix type in the above link. You can change the format of a sparse matrix X to another one by using (for example) "X.tocsc()" if necessary, but do not use it too often. For some special sparse matrix operations, it might be more efficient to write it by yourself. We provide an example "cscMatInplaceEleMultEveryRow" in the notebook. You can use it, or modify it for your own purpose.

This will be a good practice for you to think about how to write an efficient ML algorithm. Try to avoid building new objects inside the loop, or computing anything from scratch. You can initialize them before the loop start, and use the lightest way to update them in the loop. Note any operation that seems "small" inside the loop could possibly lead to expensive computations, considering the total number of times it will be executed.

You can use "if" to avoid unnecessary operations on zeros. Do not loop over matrices or vectors if not necessary: use matrix or batch operations provided by numpy or scipy instead. Try to use the most efficient operation when there are many choices reaching the same result. If you write an inefficient code here, running it will take extremely longer time. During debugging, timing each step or operation in the loop will help you figure out which step takes longer time, and you can then focus on how to accelerate it.

Run this experiment may take 1-3 hours, depending on your implementation, please start early. Before you leave it running by itself, make sure that the timing results indicate a reasonable finishing time, and remember to save intermediate results to avoid restarting from the very beginning caused by program crashes. You need to finish:

- Explain how do you modify your implementation to make the code more efficient compared to the "naive" implementation you did for Sub-Problem 1. Compare the computational costs of every coordinate descent iteration in terms of n (number of data points) and m (number of dimensions).
- Plot the training RMSE (on the training set) v.s. λ values and validation RMSE (on the validation set) v.s. λ values on a 2D plot. Use the definition of λ_{max} in sub-problem 3 and run experiments on multiple values of λ. You can reduce the number of different λ values to 20. You can also increase the minimal λ to be slightly larger than 0 such that $0 \leq \lambda_{min} \leq 0.1\lambda_{max}$. These two changes will save you some time.
- Plot the lasso solution path defined in Sub-Problem 3.
- Report the best λ value achieving the smallest validation RMSE you find on the validation set, and report the corresponding test RMSE (on the test set).
- Report the top-10 features (words in comments) with the largest magnitude in the lasso solution w when using the best λ value, and briefly explain if/why they are meaningful.