CS 240 : Lab 3
Stochastic Gradient Descent and Regularization

TAs : Harshvardhan Agarwal, Pulkit Agarwal

Instructions

- This lab delves into the essentials of stochastic optimization, Bayesian parameter estimation, and regularization methods to enhance model robustness.
- This lab will be **graded**. The weightage of each question is provided in this PDF.
- Please read the problem statement and the submission guidelines carefully.
- All code fragments need to be written within the `TODO` blocks in the given Python files. Do not change any other part of the code.
- **Do not** add any *print* statements to the final submission since the submission will be evaluated automatically.
- For any doubts or questions, please contact either the TA assigned to your lab group or one of the 2 TAs involved in making the lab.
- The submission will be evaluated automatically, so stick to the naming conventions and follow the directory structure strictly.
- The deadline for this lab is **Monday, 29 January, 5 PM**.
- The submissions will be checked for plagiarism, and any form of cheating will be appropriately penalized.

The directory structure should be as follows (nothing more, nothing less). **Since your submission will go through an automated grading tool, if these structures are not properly maintained, it may not be graded properly (even if the program is correct)**.

```
cs240_rollno_lab3 /
     |- q1 /
         |- q1.py
     |- q2 /
         |- q2.py
     |- q3 /
         |- q3.py
```

After creating your directory, package it into a tarball: **cs240_rollno_lab3.tar.gz**

Command to generate tarball:
`tar -czvf cs240_rollno_lab3.tar.gz cs240_rollno_lab3/`

1 Question 1: Ridge Regression [55 marks]

Ridge regression is a regularization method that is used to analyze any data that suffers from multicollinearity. Multicollinearity is identified when two or more predictor variables in a regression model have a high correlation. In the presence of multicollinearity, it becomes challenging to determine the individual effect of each predictor variable on the dependent variable. The coefficients of the correlated variables may become unstable, and it may be difficult to assign unique contributions to each variable (Interested students can read this to further understand multicollinearity and its effects, for this lab, we won't be focusing on this aspect).

1.1 Ridge Regression as MAP Estimate

As discussed in the lectures and the previous lab, the setup that we have is as follows:

- **Dataset:** $\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{n}$, where $\mathbf{x}^{(i)}$ is an input data point and $y^{(i)}$ is the corresponding observed output. Each input data point is a d-dimensional vector $\mathbf{x}^{(i)} \in \mathbb{R}^d$, and the outputs are scalars $y^{(i)} \in \mathbb{R}$.
- **Weights:** $w \in \mathbb{R}^d$, where d is the dimension of each data point.
- **Hypothesis Function:** $h_w(\mathbf{x}) = w^\top \mathbf{x}$ is the hypothesis function to be optimized to predict the output for any point $\mathbf{x}$ in the d-dimensional space.

In class, we studied the Maximum Aposteriori (MAP) estimate. When the MAP estimate is applied to linear regression with a Gaussian prior on the weight vector w, the objective function obtained is equivalent to that for Ridge Regression. This can be expressed as

$$\textbf{Ridge Loss:} \quad \mathcal{L}(w) = \|y - \mathbf{X}w\|_2^2 + \lambda \|w\|_2^2$$

Here λ is a hyperparameter that determines the strength of the regularization. This is generally tuned with the help of different techniques like grid search and cross-validation (interested students can read more here).

1.2 Tasks to be completed

We have provided a `dataset.csv` file which consists of a sample dataset. We have also divided the dataset into train and test splits. In the file `q1.py`, you are expected to:

- Find the closed-form solution of the weight vector for ridge regression. Use only the train split to find the weight vector. Code in the function `create_regression_weights`. [25 marks]
- Find the mean squared error (MSE) on the test set corresponding to a given weight vector (which is provided as an argument to the function `generate_test_error`). [25 marks]
- Find the optimum value of the hyperparameter λ using grid search over the space `[0,2)` (given in the file). Here, the optimum λ corresponds to the one with the lowest MSE on the test set. [5 marks]
- Plot the test set errors versus λ for the above range. [Ungraded]

To run the file, simply use the command line argument: `python3 q1.py`

2 Question 2 : Lasso Regression [20 marks]

Another commonly used regularization technique is called lasso regression (also known as $L1$ regularization). This incorporates a penalty term based on the absolute values of the regression coefficients. This penalty encourages sparsity in the model, effectively leading some coefficients to become exactly zero. Lasso regression is particularly useful for feature selection, where it automatically selects a subset of the most relevant features. Similar to the discussion above, lasso regression can also be interpreted from a Bayesian perspective as an MAP estimate. In this interpretation, the $L1$ regularization term in Lasso corresponds to a Laplace prior on the weight vector. The objective function for Lasso Regression can be expressed as

$$\textbf{Lasso Loss:} \quad \mathcal{L}(w) = \frac{1}{n}\|y - \mathbf{X}w\|_2^2 + \lambda\|w\|_1$$

2.1 Code

To run the file use the following command line:

```
python q2.py --dataset <dataset01.csv> --seed <seed_value> --eta <learning_rate>
```

The gradient descent submission will be evaluated for different datasets with different seeds (these seeds are used to initialise d-dim weight tensors passed to the Gradient descent algorithm) and different learning rates η. The code has been divided into several functions for your convenience. The following are the functions present in the code file:

- `train test split()`: This function generates an 80:20 train:test split given a dataset. The significance of the train:test split will be discussed later in the class. You do not need to edit this function.
- `lasso_loss()`: Write the code to calculate Lasso Loss in this function.
- `lasso_loss_derivative()`: Write the code to compute gradient of the Lasso-Loss function.(You may need basic matrix calculus for this question).
- `train_model()`: This function does a gradient descent based on the `lasso_loss` and `lasso_loss_derivative`.

In the file `q2.py`, you are expected to:

- Implement `lasso_loss()` [10 marks]
- Implement `lasso_loss_derivative()` [10 marks]

Note: For implementing the derivative of L_1 regularization $||w||_1$, use `torch.sign()` function.

2.2 Further Reading (Ungraded)

Regularization encompasses a range of techniques to correct for overfitting in machine learning models. As such, regularization is a method for increasing a model's generalizability, i.e., its ability to produce accurate predictions on new datasets. Interested students can read more about regularization in machine learning here.

3 Question 3: Stochastic Gradient Descent [25 marks]

Stochastic Gradient Descent (SGD) is an optimization algorithm widely employed in machine learning for training models. Unlike traditional Gradient Descent (GD), which computes the gradient over the entire dataset in each iteration, SGD updates model parameters based on the gradient of the cost function with respect to a single randomly chosen data point. This inherent stochasticity introduces variability in each iteration, allowing SGD to escape local minima and converge more rapidly, especially in high-dimensional and noisy datasets.

In this question, you will have to implement both GD and SGD.[1] The procedures take in `X` and `y` values, `eta` (step size) and `total_steps` while returning an array of `w_values` (`w` after each update step). We will later plot these values on a contour diagram to build intuition on how the algorithms work.

Algorithm 1 Gradient Descent $(X, Y, eta, total\_steps)$

$w \leftarrow zero\_vector()$
$w\_values \leftarrow [w]$
$step \leftarrow 0$
while $step < total\_steps$ **do**
  $dw \leftarrow \nabla f(X, Y, w)$
  $w = w - eta * dw$
  append w to w_values
  $step \leftarrow step + 1$
end while
return w_values

Algorithm 2 Stochastic Gradient Descent $(X, Y, eta, total\_steps)$

$w \leftarrow zero\_vector()$
$w\_values \leftarrow [w]$
$step \leftarrow 0$
while $step < total\_steps$ **do**
  $dw \leftarrow \nabla f(X[step], Y[step], w)$ ▷ $X[step], y[step]$ represents X, y values of $step$-th datapoint
  $w = w - eta * dw$
  append w to w_values
  $step \leftarrow step + 1$
end while
return w_values

3.1 Code

To run the file use the following command line:

```
python3 q3.py --dataset <dataset_file> --eta <learning_rate> --steps <total_steps>
```

Get the data from file `data.csv`, containing columns x_1, x_2 and y.

The gradient descent submission will be evaluated for different datasets and learning rates `eta`. The code has been divided into several functions for your convenience. You only have to make changes in the code file within `### YOUR CODE BEGINS HERE ###` and `### YOUR CODE ENDS HERE ###`.

In the file `q3.py`, you are expected to:

- Implement GD according to Algorithm 1. [5 marks]

[1] It is okay to reuse your earlier code on GD, though you are supposed to implement using a slightly different stopping condition here. The purpose of GD implementation along with SGD is to observe the difference between the two.

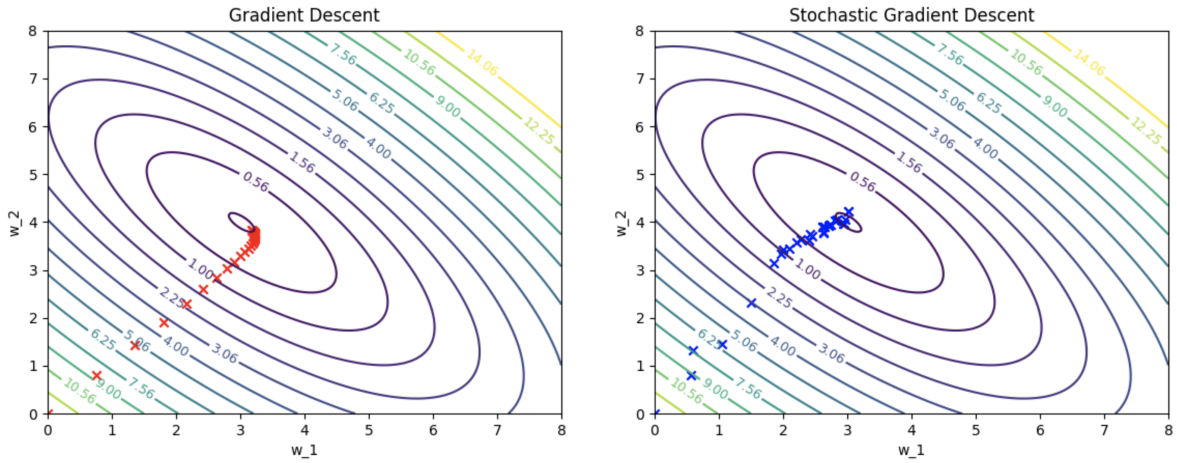
- Implement SGD according to Algorithm 2. [20 marks]
- Compare generated plots `contour_plot_GD.png` and `contour_plot_SGD.png`. [Ungraded]

Note: As mentioned in the pseudo-code above, for i-th step of SGD, we are using the i-th datapoint from the given dataset. Please stick to this convention.

3.2 Contour Plots (Ungraded)

The generated contour plots show how w gets updated in each step of descents. In GD, the entire dataset is utilized to calculate the average cost function gradient in each iteration, resulting in a smoother and more direct convergence path. On the other hand, SGD updates the weights based on a randomly chosen data point or a small batch of data points, making it computationally efficient and potentially faster to converge, although its path is more erratic due to the frequent updates.

You can mess around with different values of `eta` and `total_steps` to observe the difference between updates of GD and SGD.



The plots shown here correspond to `eta` = 0.2 and `total_steps` = 30.