ISyE4133: Advanced Optimization

Project #2: Predicting Cancer Diagnoses

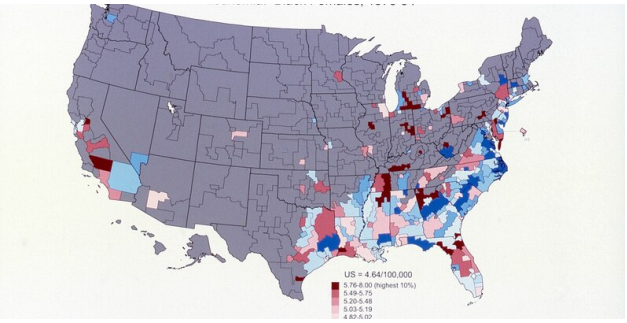
Checkpoint: *May 30 (11:59pm EST)* Project: Jun 11 (11:59pm EST)

Problem Description

Machine learning develops data-driven methods to predict future outcomes. It employs statistics to describe models, data to adapt these models to real-world scenarios, and optimization to find the best way to configure the model to the data. In this problem, we will consider two models for predicting the annual number of cancer diagnoses in the US. Our dataset can be downloaded from:

`https://www.kaggle.com/datasets/varunraskar/cancer-regression`.

In particular, we will use the dataset in the file `cancer_reg.csv`, which contains several attributes such as income level, demographics, and percentage of married households to predict the number of diagnoses or deaths from cancer. However, like many real-world datasets, this dataset is not perfect: it contains missing or invalid data as well as extraneous data. A visualization of the dataset is shown below.



In this project, we will get our "hands dirty" and work with real data, build models, and develop algorithms to turn data into predictions. At the same time, we will see what optimization techniques are required to get this models to perform well, as well as understand the limitations of these models. Students will:

1. Work with semi-real world data and learn how to parse, clean, and process them for data analysis.
2. Basic familiarity with implementing linear and nonlinear models, as well as training them within `PyTorch` to best fit models to data.
3. Initial practice with hyperparameter tuning, which allows for better empirical training of models.
4. Exposure to basic hardware accelerators (e.g., GPUs) to speedup training.

Submission Instructions

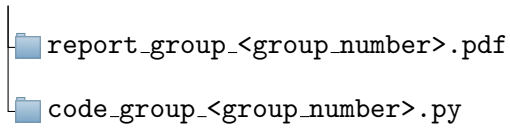
- Please upload:

    1. A single **typed** PDF report with your answers (when applicable) to the questions below. Moreover, all plots in your report must be digitally created with a clear and appropriate title, x and y-axis label, several x and y-axis tick labels[1], and a legend when appropriate. Your submission should also contain a brief paragraph explaining the contribution of each member.
    2. Python code. Your code must be able to run if we directly copy and paste the contents into a Jupyter/Colab notebook. **Code that does not run may automatically receive a zero for the code portion**[2]. You can assume when I run the code, I have a local copy of `cancer_reg.csv` and basic Python packages (e.g., `numpy`, `pandas`, `matplotlib`) and necessary packages to run PyTorch.

- Outside references, including generative AI are allowed. But the submitted work must be one's own original work and effort. Use of outside reference must be properly cited, i.e., name of the reference, a link to the reference (if possible), and how the reference was used. Collaborations are only permitted within the group.

File Directory Structure

Your submission should consist of one PDF and one Python file. I will assign each group a number when they submit the project checkpoint. When archiving your Python files and preparing your PDF report, please use the following file directory structure for your submission:



For example, `report_group_<group_number>.pdf` should be submitted as `report_group_1.pdf` if your group number is 1. You may lose a small number of points if your submission does not adhere to this format.

[1] Matplotlib will usually have these automatically generated.
[2] I will try to be lenient for small bugs due to issues between running code on different machines.

Details

- Use the dataset described in the "Problem Description" section. In particular, we will work with the file `cancer_reg.csv`. A description of each column in this CSV file can be found in the provided Kaggle link. Use `pandas` to parse and manipulate this dataset (see Part 1 for more details).
- In your code, use comments to identify which codes correspond to answering which questions. This will make it easier for staff to grade the coding portion. Code that is written poorly and does not have comments to guide reader may thus lose points due to low code "read-ability". **Note**: You can complete your work in a Colab notebook and convert to a Python file afterwards. Text cells are automatically converted to multi-line comments.
- Grading is roughly 50% report (includes correctness and presentation, i.e., writing is mostly clear and not too many grammatical errors), 45% for code (includes ability to run, correct, and "read-able", i.e., should be able to be read and understood by someone who has not seen your code before and should not contain excess comments nor commented out code), and 5% for completing the project checkpoint.
- If you feel some group members did not sufficiently contribute to the project and caused the overall quality to go down and do not know how to resolve it, you can privately email the instructor to find alternative grading guidelines.

To get you started, you may refer to the following starter code with hints:

https://colab.research.google.com/drive/1flo0v2zzah4-2UA3Ro5e84VlEDAQZO_F?usp=sharing.

Questions

1. (30 points) This question is about downloading, inspecting, parsing, and cleaning data. Download the cancer dataset from Kaggle and extract the `cancer_reg.csv` file. Then using `pandas`, complete the following steps:
   (a) The dataset has several attributes, which can be split into labels, data/features, and neither. Suppose we want our label to be the "Average number of cancer cases diagnosed annually" and we want data/features containing a *single* numerical value that is not directly related to the number of deaths or diagnoses of cancer. Explain which attributes you will then remove, and for each give a short (at most one sentence) explanation.
   (b) Extract the label and data/features into a vector `y` and matrix `X`, respectively. `X` should have the same number of rows as elements in `y`.
   (c) Print out how many values are missing or invalid values (i.e., NaNs) in `X`.
   (d) Apply data imputation to those missing/invalid values in `X` automatically using Pandas. To do so, read the following documentation (https://tinyurl.com/mr4ph7n6) and decide which method can best adaptively fill in missing data. Explain your rationale for this choice, and apply it. If there are still any missing data after this first round of data imputation, manually fill in the data using your own method/statistic, and explain the rationale behind it.

(e) Split the data-label pair into a training-testing split with a ratio of 80:20. Use the function `train_test_split` from `scikit-learn`.

2. (10 points) Formulate the ridge regression problem as an optimization problem. In particular, use the *mean square error* as the objective, where we divide by the objective by the number of data points (this coincides with PyTorch's MSE loss function: https://docs.pytorch.org/docs/stable/generated/torch.nn.MSELoss.html). Clearly define and explain the meaning for the variables, objectives, constraints (if any), and data. Then find the "theoretically best step size" for this problem when solving using gradient descent (GD).

3. (20 points) Solve the optimization problem with GD, i.e., using full gradients. Implement the algorithm in PyTorch using the step size from Part 2. Limit the run to 5000 training iterations, and compute the testing error (mean square error, or MSE) every 100 iterations. Include in your report the total runtime and plot the testing error progression similarly to in class, where the y-axis is on a log scale. This will serve as our baseline.

   Hint: Regularization can be incorporated in the optimizer with `weight_decay` (see: https://docs.pytorch.org/docs/stable/generated/torch.optim.SGD.html).

4. (20 points) We will now use a nonlinear model. Repeat Part 3 but only run for 2000 iterations and use a 5-layer neural network, where

   - The first layer, which is also the input layer, is linear with output dimension 64
   - The second layer is a `ReLU` activation function
   - The third layer is a linear layer where both the input and output dimension are 64
   - The fourth layer is a `ReLU` activation function
   - The fifth (and final) layer is a linear layer with output dimension 1

   Use the default parameters for everything, and do not include regularization in the optimizer. When you finish, you may notice the performance is not good. Instead of including a plot, inspect the testing error, and explain what you see. Provide an explanation for the phenomena you see.

   Hint: https://machinelearningmastery.com/exploding-gradients-in-neural-networks/.

5. (30 points) To improve performance, we will apply a very basic hyperparameter tuning in this problem. Using the same model from Part 4 but only for a shorter training duration of 500 iterations, exhaustively try all combinations of the following:

   - **Optimizer**: Between `SGD` and `Adam` (see: https://docs.pytorch.org/docs/stable/optim.html). Make sure these methods are accessing the full gradient.
   - **Initialization**: Use either `normal` and `orthogonal` to initialize the weights (see: https://docs.pytorch.org/docs/stable/nn.init.html). In both cases, set the bias to zero.
   - **Step size**: Three step sizes (i.e., learning rates): the default value, a choice at least 10X larger than the default value, and a choice at least 10X smaller than the default value.
   - **Gradient clipping/normalization**: Using gradient clipping (with `max_norm=1.0`) and no clipping (see: https://docs.pytorch.org/docs/stable/generated/torch.nn.utils.clip_grad_norm_.html).

In total, there are 24 different configurations to choose from. Enumerate through all possible configurations with a single for loop (**Hint**: use the Python package `itertools` to help form all combinations). After running all 24 configuration, report the final testing error from each (you do not need to evaluate the testing error every 100 iterations like in Part 3). Which hyperparameters seemed to have the most impact of performance and which did not?

6. (5 points) Repeat Part 4 with the best set of hyperparameters from the previous part. Also plot the convergence like we did in Part 3. Compare the testing error convergence to Part 3. Is it better or worse?

7. (10 points) There are many other potential hyperparameters we could tune. Find at least 5 other hyperparameters in PyTorch we could tune (and a couple details about what they do). Provide links to documentation detailing each of your listed hyperparameters. Also, name a tuning library that can automate the tuning process for us.

8. (10 points) To speedup performance, repeat Part 6 but now implement with stochastic gradient descent (SGD) as described in class. **Since SGD involves randomness, seed your algorithm before every run to ensure consistent performance**. Compare the runtime difference between GD and SGD as well as their convergence.

9. (5 points) Another way to speedup performance is to use hardware accelerators. Similar to the previous part, repeat Part 6 but run on a GPU (e.g., a free T4 GPU on Google Colab). Compare the runtime between CPU GD and GPU GD.

10. (10 points) Looking at the all the methods we considered, we notice none of these methods are able to achieve 0 training nor testing error. In your own words, describe some reasons for why this may be the case.