B1 Engineering Computation: Project B (2020-21)

Optimization Methods for Linear Regression

Course webpage: `http://mpawankumar.info/teaching/b1.html`

1 Introduction

Regression refers to the task of estimating a function that maps a multivariate input to a real-valued output. In this project, we will restrict ourselves to linear regression, where the function being estimated is a linear function of the input. As an illustrative example, consider a city district $\mathbf{x}$. We are interested in predicting the median value of owner-occupied homes in this district. We will represent all the features that we believe are relevant to make this prediction as $\Phi(\mathbf{x})$. In other words, $\Phi(\mathbf{x})$ is a vector whose elements are scalar values that denote quantities such as district $\mathbf{x}$'s per capita crime rate, or pupil-teacher ratio in its schools.

In order to solve the above challenging problem, we will make use of a data set of city districts for which we know the ground-truth median home values. Formally, we have access to a training data set $\mathcal{D} = \{(\Phi(\mathbf{x}_i), y_i), i = 1, \cdots, n\}$ where $\Phi(\mathbf{x}_i)$ are the features of a district $\mathbf{x}_i$ and y_i is its median home value. Using the training data set $\mathcal{D}$, we will estimate a vector $\mathbf{w}$ such that $\mathbf{w}^\top \Phi(\mathbf{x})$ predicts the home value for $\mathbf{x}$. In other words, $\mathbf{w}$ is the set of parameters that performs linear regression from the input features $\Phi(\mathbf{x})$ to the output score y.

One natural way of estimating $\mathbf{w}$ is to minimize the error between the prediction and the ground-truth score over the training data set. In addition, we may also want to incorporate any prior knowledge we have about $\mathbf{w}$ when estimating it. In the remainder of the document, we will consider different types of error functions between the prediction and the ground-truth (commonly referred to as loss functions in the machine learning literature), as well as different forms of prior knowledge. For each choice, we will design an appropriate learning objective whose variables will correspond to the parameters $\mathbf{w}$. The computational task is to design and code appropriate optimization algorithms that minimize the learning objective in order to provide the best estimate of $\mathbf{w}$. Once we have obtained the parameters $\mathbf{w}$, we will be able to use it to make predictions for previously unseen test samples (new city districts).

2 Data Set

Before you start coding, get familiar with the data set that we will use, namely, the Boston Housing Data Set. The data set is available for download at the following URL:

`http://lib.stat.cmu.edu/datasets/boston`

It consists of 506 rows where each row corresponds to a sample (a city district). The first 13 entries of each row denote the features for the corresponding district. The final entry is the housing price we wish to predict. Download the data set and store it in a format that is easy to handle with Matlab (for example, in .mat or .txt formats). In each task, we will be randomly splitting the samples into training and testing data sets. We will use the training data set to estimate the parameters $\mathbf{w}$. We will use the test data set to evaluate the quality of the parameters.

Tip: Avoid the temptation to hard code values such as the number of features or the number of samples in your program. Make sure your code is as general as possible, that is, it can be used to perform linear regression on any given data set. This will allow you to explore other regression tasks once you have finished the basic tasks of the project. There are plenty more learning objectives and data sets to explore, and a good report should include some interesting extensions.

3 Least Squares Regression

We start by considering a simple learning objective, namely the mean-squared difference between the prediction and the ground-truth over the training data set $\mathcal{D}$. Formally, the learning objective of *least-squares regression* is defined as follows:

$$\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (\mathbf{w}^\top \Phi(\mathbf{x}_i) - y_i)^2. \tag{1}$$

A significant advantage of the above learning objective is that it admits a closed form solution for the best estimate of parameters $\mathbf{w}$. Specifically, it can be verified that if we take the gradient of the above learning objective and set it to $\mathbf{0}$ in order to find a stationary point (in this case, the minimum), we obtain a linear system of equations of the form

$$\mathbf{A}\mathbf{w} = \mathbf{b} \Longrightarrow \mathbf{w} = \mathbf{A}^{-1}\mathbf{b}. \tag{2}$$

In other words, operationalizing least-squares regression only requires us to find the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ corresponding to the given training data set. The actual optimization effort is minimal since it only requires a single matrix inversion followed by a matrix-vector product. We do need to take care of situations where the matrix $\mathbf{A}$ is not invertible, either by taking its pseudo-inverse, or by adding a small constant to its diagonal elements.

Task 1: Least-squares regression.

- Write a function `random_split`, which takes as its input a data set `D` (in matrix form), and a scalar value `frac` that lies between 0 and 1. Note that we will make the assumption that each row of `D` represnts one sample. The last element of each row is the value we wish to predict, and the remaining elements are the feature vector. The function should split the data set into two: `frac` fraction of the data set should become the training data set `train_D` and the remaining should become the test data set.

  ```
  [train_D,test_D] = random_split(D, frac)
  ```

  Make sure that the split is random. You may find Matlab's in-built function `randperm` useful for this purpose.

- Write a function `lsq_regression`, which takes as its input `train_D` and returns the best estimate of $\mathbf{w}$ by (i) forming the appropriate matrix $\mathbf{A}$; (ii) forming the appropriate vector $\mathbf{b}$; and (iii) computing the closed form solution of $\mathbf{w}$. You may use Matlab's inbuilt function `inv` to compute the inverse of $\mathbf{A}$.

  ```
  w = lsq_regression(train_D)
  ```

- Write a function `compute_mean_squared_error`, which takes as its input `test_D` and `w`, and returns the mean-squared error over the test data set.

  ```
  mse = compute_mean_squared_error(test_D,w)
  ```

- Try several different values of `frac`. For each value, repeat the above steps multiple times, each time generating a different random split of the data set. Report the mean and standard deviations of all the statistics you are computing.

As the Boston Housing Data Set is small, you may find that the variations in runtime and test accuracy

do not vary much as you change `frac`. As an optional subtask, you may wish to try this on a significantly larger data set.

4 Ridge Regression

You may notice that the parameters estimated using least squares regression perform (significantly) better on the training data set compared to the test data set. This is expected. After all, we are estimating the parameters by explicitly minimizing the loss over the training data set. However, we could perhaps improve the performance on the test data set (which is what we care about) by incorporating some prior knowledge about the parameters. A simple example of such a prior knowledge could be that the magnitude of the parameters should not be very large. This knowledge can be incorporated by introducing the square of the ℓ_2 norm of the parameters into the learning objective. Formally, we define the *ridge regression* learning objective as follows:

$$\min_{\mathbf{w}} \frac{\lambda}{2}||\mathbf{w}||_2^2 + \frac{1}{n}\sum_{i=1}^{n}(\mathbf{w}^\top \Phi(\mathbf{x}_i) - y_i)^2. \tag{3}$$

The term λ is the relative importance of the *regularization term* (that is, the squared ℓ_2 norm term) compared to the mean-squared loss over the training objective. Given a value of λ, ridge regression retains the main computational advantage of least-squares regression. In other words, it admits a closed-form solution since setting its gradients to zero also results in a linear system of equations.

How can we find an appropriate value of λ? Note that we cannot minimize the above objective function with respect to both $\mathbf{w}$ and λ, since this would set $\lambda = 0$, thereby reducing the problem to least-squares regression. Furthermore, we cannot set λ such that the resulting parameters performs best on the test data set since, in practice, we will not have access to the ground-truth score of the test samples. Instead, we split the data set $\mathcal{D}$ with known ground-truth scores into two parts: (i) $\mathcal{D}_{train}$, the training data set, which is used to estimate the parameters $\mathbf{w}$ for different values of λ; and (ii) $\mathcal{D}_{val}$, the validation data set, which is used to estimate the goodness of different values of λ.

In more detail, we restrict λ to belong to a set of values, for example,

$$\lambda \in \{1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3\}. \tag{4}$$

For each putative value of λ, we estimate $\mathbf{w}$ using the training data set, and then use it to measure the error on the validation data set. We pick the value of λ which performs the best on the validation data

set. Fixing λ to this value, we estimate the final set of parameters using the entire data set $\mathcal{D}$.

Task 2: Ridge Regression

- Split the entire data set into two using the previously coded `random_split` function.

  ```
  [trainval_D,test_D] = random_split(D, frac)
  ```

- Using the same `random_split` function, split the first data set into a training data set and a validation data set.

  ```
  [train_D,val_D] = random_split(trainval_D,frac2)
  ```

  Note that `frac2` need not be the same as `frac`. Typically, `frac2` is set to 0.8.

- Write a function `ridge_regression`, which takes as its input `train_D` and `lambda` and returns the best estimate of **w** by (i) forming the appropriate matrix **A**; (ii) forming the appropriate vector **b**; and (iii) computing the closed form solution of **w**. You may use Matlab's inbuilt function `inv` to compute the inverse of **A**.

  ```
  w = ridge_regression(train_D,lambda)
  ```

- Identify the best value of `lambda` by computing the mean-squared error on the validation data set.

- Find the best estimate of **w** by training on the entire data set `trainval_D` using the value of `lambda` found in the previous step. Use this **w** to find the mean-squared error over the test data set. How does this compare with the accuracy of least squares regression?

- **(Optional)** You may wish to generate multiple splits of the entire data set, and find the value of `lambda` that provides the best performance *on average* over the multiple splits. Perhaps you could use 5 different splits in such a way that each sample is present in the validation data set exactly once.

5 Gradient Descent for Smoothed Functions

So far we have considered mean-squared loss functions and squared ℓ_2 norm regularization due to their inherent advantage of admitting closed form solutions. However, these are not the only options that make sense for regression. For example, we can measure the error between the prediction and the ground-truth using the absolute difference. Similarly, we can regularize the parameters using their ℓ_1 norm (which has been shown to provide a sparser parameter vector). In other words, the following objective function is perfectly valid for regression:

$$\min_{\mathbf{w}} f(\mathbf{w}) \equiv \frac{\lambda}{2}||\mathbf{w}||_1 + \frac{1}{n}\sum_{i=1}^{n} |\mathbf{w}^\top \Phi(\mathbf{x}_i) - y_i|. \tag{5}$$

The above objective function does not admit a closed form solution. In fact, it is not even differentiable. Hence, we will not even be able to use gradient descent to minimize it. While there are ways of handling non-smooth functions of this type, in this project (at least, in this part of the project), we will take a much simpler approach, namely, approximating non-smooth functions with smooth alteratives.

Pointwise Maximum of Linear Functions. Consider the following function:

$$g(\mathbf{w}) = \max_i \{\mathbf{a}_i^\top \mathbf{w} + b_i\}, \text{ where } i \in \{1, \cdots, m\}. \tag{6}$$

Given a value of $\mathbf{w}$, one can evaluate the function by first computing $\mathbf{a}_i^\top \mathbf{w} + b_i$ for each $i = 1, \cdots, m$, and then picking the maximum value. The above function does not have gradients everywhere (try to draw an example where $\mathbf{w}$ is 1-dimensional).

Smoothed Version. We can replace $g(\mathbf{w})$ with an approximation as follows:

$$g_\tau(\mathbf{w}) = \tau \log\left(\sum_{i=1}^{m} \exp\left(\frac{1}{\tau}(\{\mathbf{a}_i^\top \mathbf{w} + b_i\})\right)\right), \tag{7}$$

where τ is a user-specified positive constant. It can be verified that as $\tau \to 0+$, $g_\tau \to g$. Unlike g, the function g_τ is differentiable everywhere for a reasonable value of τ. Thus, for the sake of efficient optimization, we can replace g with g_τ.

Numerical Instability. The use of $\exp$ and $\log$ can create numerical instability issues when evaluating g_τ or its gradients. To overcome this, it is common to evaluate g_τ using the following *log-sum-exp*

trick:

$$g_\tau(\mathbf{w}) = g(\mathbf{w}) + \tau \log \left(\sum_{i=1}^{m} \exp \left(\frac{1}{\tau} (\{\mathbf{a}_i^\top \mathbf{w} + b_i\} - g(\mathbf{w})) \right) \right), \tag{8}$$

Since $\mathbf{a}_i^\top \mathbf{w} + b_i \leq g(\mathbf{w})$ for all $i \in \{1, \cdots, m\}$, the argument of the $\exp$ function will not be very large. Therefore, any errors caused by $\exp$ will not be substantial.

The learning objective $f(\mathbf{w})$ can be reformulated as a sum of pointwise maximum of linear functions as follows:

$$f(\mathbf{w}) = \min_{\mathbf{w}} \frac{\lambda}{2} \sum_{j=1}^{d} \max\{w_j, -w_j\} + \frac{1}{n} \sum_{i=1}^{n} \max\{\mathbf{w}^\top \Phi(\mathbf{x}_i) - y_i, y_i - \mathbf{w}^\top \Phi(\mathbf{x}_i)\}, \tag{9}$$

where d is the dimensionality of $\mathbf{w}$. We can now replace f with its smoothed approximation f_τ, and minimize it using gradient descent. Specifically, we initialize $\mathbf{w}$ as $\mathbf{w}_0$ (for example, a vector whose elements are all equal to 0). We then iteratively update the parameters as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla f_\tau(\mathbf{w}_t), \tag{10}$$

where η is an optimization hyperparameter called the step-size and $\nabla f_\tau(\mathbf{w}_t)$ is the gradient of f_τ evaluated as $\mathbf{w}_t$. We stop the iterations based on a convergence criterion (for example, the change in the value of the objective function f_τ from one iteation to the next goes below a certain threshold). You will find it very helpful to plot the objective function value over the iterations in order to figure out an appropriate step-size η and an appropriate convergence criterion. Use Matlab's in-built `plot` function for this purpose.

Task 3: Smoothed ℓ_1 Regression

- Generate the training, validation and test data sets as before.

```
[trainval_D,test_D] = random_split(D, frac)
[train_D,val_D] = random_split(trainval_D, frac2)
```

- Write a function `smoothed_l1_regression`, which takes as its input `train_D` and `lambda` and returns the best estimate of **w** by performing gradient descent on the function f_τ. Choose a small value of τ to ensure that the approximation is accurate, but make sure that your code is numerically stable.

```
w = smoothed_l1_regression(train_D,lambda)
```

- Identify the best value of `lambda` by computing the mean-absolute error on the validation data set using a function `compute_mean_abs_error`.

```
mae = compute_mean_abs_error(val_D,w)
```

  Note that we are using a different loss function for evaluation, since we are using a different loss function for training.

- Find the best estimate of **w** by training on the entire data set `trainval_D` using the value of `lambda` found in the previous step. Use this **w** to find the mean-absolute error over the test data set.

- **(Optional)** As in the previous section, you may wish to generate multiple splits of the entire data set, and find the value of `lambda` that provides the best performance *on average* over the multiple splits.

- **(Optional)** You may wish to start with a large value of τ, minimize f_τ and then use this as an initialization for a new problem with a reduced value of τ. By gradually annealing the value of τ instead of starting with a very small one, you may obtain a more numerically stable and/or efficient algorithm.

6 Stochastic Optimization for Large-Scale Data Sets

One of the main computational deficiencies of gradient descent for linear regression is that the amount of time required to compute the gradient scales linearly with the size of the training data set. When the data set becomes large (in practice, it is common to use data sets with millions of samples), gradient descent becomes infeasible. Luckily, there is a more suitable alternative based on stochastic approximation of the learning objective. Consider picking one training sample, indexed by $k \in \{1, \cdots, n\}$, from a uniform distribution over all the training samples. We now define a stochastic approximation of the learning objective as

$$f^k(\mathbf{w}) = \min_{\mathbf{w}} \frac{\lambda}{2}||\mathbf{w}||_1 + |\mathbf{w}^\top \Phi(\mathbf{x}_k) - y_k|. \tag{11}$$

It can be shown that the expected value of f^k with respect to the uniform distribution over the training samples is exactly equal to the original learning objective f. Thus, instead of computing the gradient of f_τ, we compute the gradient of the smoothed version of f^k, which we denote by f^k_τ. In more detail, we initialize the parameter $\mathbf{w}$ as $\mathbf{w}^0$. At each iteration t, we pick a training sample indexed by k_t uniformly at random, and update the parameter as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla f^{k_t}_\tau(\mathbf{w}_t). \tag{12}$$

Note that, unlike gradient descent, the step-size is no longer a constant, but depends on the iteration number. In particular, it has been shown that one is guaranteed to convergence to a solution that is close to the globally optimal solution if the step-sizes form a diminshing, divergent sequence. In other words, we need to choose η_t such that

$$\lim_{t\to\infty} \eta_t = 0, \lim_{T\to\infty} \sum_{t=1}^{T} \eta_t = \infty. \tag{13}$$

You have seen several examples of diminshing and divergent sequences in P1. Choose any such sequence to implement stochastic optimization for smoothed ℓ_1 regression (e.g. $\eta_t = 1/t$). A reasonable convergence criterion would be to stop the optimization once η_t is below a certain threshold. You may find it helpful to measure the true objective function value after every T iterations to check if you are making some progress.

Task 4: Stochastic Smoothed ℓ_1 Regression

- Generate the training, validation and test data sets as before.

```
[trainval_D,test_D] = random_split(D, frac)
[train_D,val_D] = random_split(trainval_D, frac2)
```

- Write a function `stochastic_smoothed_l1_regression`, which takes as its input `train_D` and `lambda` and returns the best estimate of **w** by performing stochastic gradient descent. Choose a small value of τ to ensure that the approximation is accurate.

```
w = stochastic_smoothed_l1_regression(train_D,lambda)
```

- Identify the best value of `lambda` by computing the mean-absolute error on the validation data set.

```
mae = compute_mean_abs_error(val_D,w)
```

- Find the best estimate of **w** by training on the entire data set `trainval_D` using the value of `lambda` found in the previous step. Use this **w** to find the mean-absolute error over the test data set.

- **(Optional)** As in the previous section, you may wish to generate multiple splits of the entire data set, and find the value of `lambda` that provides the best performance *on average* over the multiple splits. You may also wish to anneal the value of τ for increased numerical stablity and/or efficiency.

7 Linear Programming Formulation

So far we have only considered unconstrained optimization for linear regression. One can also reformulate the problem $\min_{\mathbf{w}} f(\mathbf{w})$ as a linear program, which can allow us to obtain the globally optimal solution without resorting to smoothed approximations. Specifically, we can estimate $\mathbf{w}$ using the fol-

lowing linear program:

$$\begin{aligned} \min_{\mathbf{w},\boldsymbol{\nu},\boldsymbol{\xi}} \quad & \frac{\lambda}{2}\sum_{j=1}^{d}\nu_j + \frac{1}{n}\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & \nu_j \geq w_j, \\ & \nu_j \geq -w_j, \\ & \xi_i \geq \mathbf{w}^\top\Phi(\mathbf{x}_i) - y_i, \\ & \xi_i \geq y_i - \mathbf{w}^\top\Phi(\mathbf{x}_i). \end{aligned} \tag{14}$$

Convince yourself that the above problem is (i) equivalent to minimizing $f(\mathbf{w})$; and (ii) is a linear program (Hint: The objective function is being minimized with respect to ν_j, so ν_j will take the smallest value that satisfies the constraints). An additional advantage of a linear programming formulation is that one can incorporate more complex prior knowledge about the parameters, such as the non-negativity of some or all of its elements, as additional linear constraints.

Task 5: Linear Programming for ℓ_1 Regression

- Generate the training, validation and test data sets as before.

```
[trainval_D,test_D] = random_split(D, frac)
[train_D,val_D] = random_split(trainval_D, frac2)
```

- Write a function `lp_l1_regression`, which takes as its input `train_D` and `lambda` and returns the best estimate of **w** by solving an appropriate linear program. You may use Matlab's inbuilt linear programming solver `linprog` for this purpose.

```
w = lp_l1_regression(train_D,lambda)
```

- Identify the best value of `lambda` by computing the mean-absolute error on the validation data set.

```
mae = compute_mean_abs_error(val_D,w)
```

- Find the best estimate of **w** by training on the entire data set `trainval_D` using the value of `lambda` found in the previous step. Use this **w** to find the mean-absolute error over the test data set.

- Compare the accuracy and the runtime of the linear programming based solver with (stochastic) gradient descent.

- **(Optional)** As in the previous section, you may wish to generate multiple splits of the entire data set, and find the value of `lambda` that provides the best performance *on average* over the multiple splits.

8 Extensions

Now that you have coded the basic steps, you may want to go back and revisit the optional steps in the previous tasks. Or perhaps you want to try linear regression on a data set of your own? Or improve the speed of stochastic gradient descent by considering mini-batches instead of a single sample at each iteration? Or perform variance reduction using more sophisticated methods such as SVRG? Or read about subgradients and how they can be used to minimize non-smooth functions, and code them up? There are no restrictions on the type of extensions you can explore as part of the project. So have fun researching the topic.