Introduction to Machine Learning (67577)

Exercise 4
Gradient-Based Learning

Second Semester, 2025

Contents

Submission

Please make sure to follow the general submission instructions available on the course website. In addition, for the following assignment, submit a single `ex4_ID.tar` file containing:

- An `Answers.pdf` file with the answers for all theoretical and practical questions (include plotted graphs *in* the PDF file).
- The following python files (without any directories): `gradient_descent.py`, `learning_rate.py` , `modules.py`, `logistic_regression.py` and `gradient_descent_investigation.py`

The `ex4_ID.tar` file must be submitted in the designated Moodle activity prior to the date specified *in the activity*.

- Plots included as separate files will be considered as not provided.

1 Theoretical Part

1.1 Convex optimization

1. Let $f_1, \dots, f_m : C \to \mathbb{R}$ be a set of convex functions and $\gamma_1, \dots, \gamma_m \in \mathbb{R}_+$. Prove from definition that $g(\mathbf{u}) = \sum_{i=1}^{m} \gamma_i f_i(\mathbf{u})$ is a convex function.
2. Give a counterexample for the following claim: Given two functions $f, g : \mathbb{R} \to \mathbb{R}$, define a new function $h : \mathbb{R} \to \mathbb{R}$ by $h = f \circ g$. If f and g are convex then h is convex as well.

1.2 Sub-gradients for Soft-SVM Objective

The Soft-SVM objective, though convex, is not differentiable in all of its domain due to the use of the hinge-loss. Therefore, to implement a sub-gradient descent solver for this problem we must first describe sub-gradients of the objective.

3. Given $\mathbf{x} \in \mathbb{R}^d$ and $y \in \{\pm 1\}$. Show that the hinge loss is convex in $\mathbf{w}, b$. That is, define

$$f(\mathbf{w}, b) := \ell^{hinge}_{\mathbf{x},y}(\mathbf{w}, b) = \max\left(0, 1 - y(\mathbf{x}^\top \mathbf{w} + b)\right)$$

   and show that f is convex in $\mathbf{w}, b$.

4. Deduce some sub-gradient of the hinge loss function $g \in \partial \ell^{hinge}_{\mathbf{x},y}(\mathbf{w}, b)$.
5. Let $f_1, \dots, f_m : \mathbb{R}^d \to \mathbb{R}$ be a set of convex functions and $\mathbf{g}_k \in \partial f_k(\mathbf{x})$ for all $k \in [m]$ be sub-gradients of these functions. Define $f : \mathbb{R}^d \to \mathbb{R}$ by $f(\mathbf{x}) = \sum_{i=1}^{m} f_i(\mathbf{x})$. Show that $\sum_k \mathbf{g}_k \in \partial \sum_k f_k(\mathbf{x})$.

6. Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m} \subseteq \mathbb{R}^d \times \{\pm 1\}$ be a sample and define $f : \mathbb{R}^d \to \mathbb{R}$ by:

$$f(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \ell^{hinge}_{\mathbf{x}_i, y_i}(\mathbf{w}, b) + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

   Find a sub-gradient of f for any $\mathbf{w}$.

2 Practical Part

Write the necessary code in the files specified in the questions.

2.1 Gradient Descent

In the following section you will implement a generic Gradient Descent algorithm, and explore and visualize its performance on different objective functions. To assist you with the implementation please start by reading the documentation of the gradient_descent.py file and following the steps as described below

Learning Rate: The `GradientDescent` class, when initialized, receives a *learning rate strategy* in the form of a `BaseLR` instance. Read the documentation of the `BaseLR` base class in the _learning_rate.py file and then implement two learning rate strategies in the :

- Constant (Fixed) Learning Rate (i.e. $\eta_t = \eta$) - `FixedLR` class in the learning_rate.py file

Objective Functions (Modules): When running the `GradientDescent.fit` function it receives an instance derived from the `BaseModule` class. This class defines the generic abstract form of any objective to be minimized using graident descent. Its two main functions are used to compute the value of the function and the derivative of the function at a given point of interest. Read the documentation of the BaseModule base class in the base_module.py file.

Implement the `L2` and `L1` modules in the _methods.modules.py file. Note that both these modules ignore any passed inputs in the `compute_output` and `compute_jacobian` functions and simply use the `weights` defined in the base class. Some of the other modules will be implemented later.

Gradient Descent Algorithm Implement the `GradientDescent` class in the `descent_methods .gradient_descent.py` file.

- Note that when instantiating a `GradientDescent` object a `callback` can be passed. This will be used to investigate different properties of the algorithm's run and will be specified in the questions below.
- Implementation must support several solution types, one of which is the average of $\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(t)}$. In your implementation do not store all solutions to avoid wasting memory.

2.1.1 Comparing Fixed learning rates

We begin with investigating the GD convergence over the L1 and L2 objectives using fixed learning rates. In the `gradient_descent_investigation.py` file implement the function `compare_fixed_learning_rates` as specified in function documentation:

- Implement the `get_gd_state_recorder_callback` function as specified in function documentation. This function returns a "fresh" callback function and lists for losses and weights throughout the GD iterations.
- Minimize the `L1` and `L2` modules for each of the following fixed learning rates $\eta \in \{1, 0.1, 0.01, 0.001\}$, setting the initial starting point (i.e. the initial value of the module's weights) to $\mathbf{w}_0 = \left(\sqrt{2}, \frac{e}{3}\right)$

  - Notice that the The `L2` module actually implements the **squared** L2 norm.

Of note, all the objective functions we saw so far depended on the training data $\mathbf{X},\mathbf{y}$. In this section, we minimize a function that ignores the given training data (like all regularization modules). Modules implemented later in this exercise will use the training data.

Then, answer the following questions:

1. Plot the descent path for each of the settings described above (you can use the `plot_descent_path`). Add below the plots for $\eta = 0.01$ and explain the differences seen between the `L1` and `L2` modules.
2. following the previous question describe two phenomena that you have seen in the descent path of the ℓ_1 objective when using GD and a fixed learning rate.
3. For each of the modules, plot the convergence rate (i.e. the norm as a function of the GD iteration) for all specified learning rates. Explain your results
4. What is the lowest loss achieved when minimizing each of the modules? Explain the differences

2.2 Minimizing Regularized Logistic Regression

In the following part you will use your implementation of Gradient Descent to solve a regularized logistic regression optimization problem. Implement the following as described below:

- Implement the `LogisticModule` in the _methods.modules.py file, as described in class documentation.
  - In the `compute_output`, you should return the negative log-likelihood

$$f(\mathbf{w}) = -\frac{1}{m}\log\left(\prod_i P(Y = y_i \mid X = \mathbf{x}_i, \mathbf{w})\right)$$

    recall derivations seen in class and recitations, as well as in the course book.
  - In the `compute_jacobian`, you should return the derivative of the objective above $\frac{\partial f(\mathbf{w})}{\partial \mathbf{w}}$.
  - Recall the calculations done in recitations 6 and 10.
- Implement the `RegularizedModule` in the _methods.modules.py file, as described in class documentation. This module receives two generic modules to be used as the fidelity and regularization terms. For example: `RegularizedModule(LogisticModule(), L1())`.
- Implement the `LogisticRegression` class in the `{logistic_regression.py file as specified in class documentation. This class should wrap the usage of your gradient descent implementation on the LogisticModule, RegularizedModule, L1 and L2.`
- `When initializing the model's weights sample from` $\mathbf{w} \sim \mathcal{N}\left(0, \frac{1}{d}I\right)$. `This is equivalent to sampling from the standard Gaussian distribution and dividing by` $\sqrt{d}$.

R Notice that the `LogisticRegression` class does not create an instance of the `GradientDescent` class. Instead, it receives it as a *dependency* in the constructor. As such, your `LogisticRegression` implementation is open for future extensions and support of different solvers. This is one of the 5 SOLID coding principles - if you wish to write good code and have a good design - be SOLID.

Then, load the South Africa Heart Disease dataset (SAheart.data), split it to train- and test sets (80% train) and answer the following questions:

5. Using your implementation, fit a logistic regression model over the data. Use the `predict_proba` to plot an ROC curve. You can use sklearn's `metrics.roc_curve` function and the code provided in Lab 04.
6. Which value of α achieves the optimal ROC value according to the criterion below. Using this value of α^* what is the model's test error?

$$\alpha^* = \text{argmax}_\alpha\{\text{TPR}_\alpha - \text{FPR}_\alpha\}$$

7. Fit an ℓ_1-regularized logistic regression by passing `penalty="l1"` when instantiating a logistic regression estimator
   - Set $\alpha = 0.5$
   - Use your previously implemented cross-validation procedure to choose λ
   - After selecting λ repeat fitting with the chosen λ and $\alpha = 0.5$ over the entire train portion.

   What value of λ was selected and what is the model's test error?
   When fitting the model you can (but don't have to) set the parameters as follows:
   - Use `max_iter=20,000` and `lr=1e-4`.
   - When searching for the optimal λ:
     - Search the following values $\lambda \in \{0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1\}$.
     - Use $\alpha = 0.5$ as the cutoff.