Instructor
Prof. Volkan Cevher

Head TA
Ya-Ping Hsieh

Homework Exercise-1 (for Lectures 4–7)

This homework covers the lectures 4, 5, 6 and 7 in which we consider binary classification by the linear support vector machine (SVM), and you will compute the SVM classifier by the gradient descent and accelerated gradient methods. You will also explore how algorithmic enhancements, such as backtracking linesearch and adaptive restart strategies, improve the numerical efficiency. In the second part, you will implement the Newton and quasi-Newton methods to compute the SVM classifier. Finally, you will implement three versions of stochastic gradient descent methods.

1 Problem statement

The support vector machine is one of the most popular approaches to *linear binary classification*. Given a training data set of n points of the form

$$(\mathbf{a}_1, b_1), \ldots, (\mathbf{a}_n, b_n)$$

where $\mathbf{a}_i \in \mathbb{R}^p$ and $b_i \in \{-1, +1\}$ indicates the class to which $\mathbf{a}_i$ belongs. The goal of linear binary classification is to learn a hyperplane in $\mathbb{R}^p$, based on the training data set, which separates the points with label $+1$ and those with label -1. Notice that this goal cannot be achieved perfectly if the labels in the training data set are noisy, e.g., each of the label is incorrect with probability $\varepsilon \in (0, 1/2)$ independently, and the ratio of incorrectly classified points in another *test data set* becomes an important performance measure.

The empirical risk minimization (ERM) principle in Lecture 2 suggests that we can consider the hyperplane that minimizes the empirical $0-1$ loss on the training set. That is, we can compute $(\mathbf{x}^\star, \mu) \in \mathbb{R}^p \times \mathbb{R}$ defined by

$$(\mathbf{x}^\star, \mu^\star) \in \operatorname*{arg\,min}_{(\mathbf{x},\mu)\in\mathbb{R}^p\times\mathbb{R}} \left\{ \ell_{0-1}(\mathbf{x}, \mu) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\{b_i(\mathbf{a}_i^T\mathbf{x}+\mu)\le 0\}}(\mathbf{x}, \mu) \right\}, \tag{1}$$

where

$$\mathbf{1}_{\{b_i(\mathbf{a}_i^T\mathbf{x}+\mu)\le 0\}}(\mathbf{x}, \mu) = \begin{cases} 1 & \text{if } b_i(\mathbf{a}_i^T\mathbf{x}+\mu) \le 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then $\{\mathbf{y} : \mathbf{y}^T\mathbf{x}^\star + \mu^\star = 0, \mathbf{y} \in \mathbb{R}^p\}$ is a separating hyperplance. We note that ℓ_{0-1} is not convex. To overcome the computational issue, ℓ_{0-1} is usually replaced by an approximation called the *Hinge loss*:

$$\ell_{\mathrm{H}}(\mathbf{x}, \mu) = \frac{1}{n}\sum_{i=1}^{n} \max\Big(1 - b_i(\mathbf{a}_i^T\mathbf{x}+\mu), 0\Big),$$

and (1) could be approximated by the Hinge loss minimization problem, plus a regularization term, $\frac{\lambda}{2}\|\mathbf{x}\|^2$. This formulation is known as linear support vector machine. Note that ℓ_{H} is not continuously differentiable, thus we want to consider a slightly different loss function. One possible replacement is modified Huber loss:

$$\ell_{\mathrm{MH}}(\mathbf{x}, \mu) = \frac{1}{2n}\sum_{i=1}^{n} g_i(\mathbf{x}, \mu)$$

such that

$$g_i(\mathbf{x}, \mu) = \begin{cases} 0, & b_i(\mathbf{a}_i^T\mathbf{x}+\mu) > 1+h, \\ (1+h-b_i(\mathbf{a}_i^T\mathbf{x}+\mu))^2/4h, & |1-b_i(\mathbf{a}_i^T\mathbf{x}+\mu)| \le h, \\ 1-b_i(\mathbf{a}_i^T\mathbf{x}+\mu), & b_i(\mathbf{a}_i^T\mathbf{x}+\mu) < 1-h. \end{cases}$$

where h is a parameter to choose. As $h \to 0$, modified Huber loss approaches Hinge loss. Finally, define the objective function f as

$$f(\mathbf{x}, \mu) := \ell_{\mathrm{MH}}(\mathbf{x}, \mu) + \frac{\lambda}{2}\|\mathbf{x}\|^2. \tag{2}$$

In this homework, we will learn a separating homogeneous half-space by modified Huber loss minimization. Specifically, we will consider the case when $\mu = 0$, solve the following SVM problem with modified Huber loss

$$\mathbf{x}^\star \in \arg\min_{\mathbf{x}} f(\mathbf{x}, 0), \tag{3}$$

and use $\{\mathbf{y} : \mathbf{y}^T\mathbf{x}^\star = 0, \mathbf{y} \in \mathbb{R}^p\}$ as the separating hyperplane. We will write $f(\mathbf{x})$ for $f(\mathbf{x}, 0)$ for simplicity.

Note: without any more explications, matrix norm in the following is understood as Frobenius norm, 0 and 1 represent zeros and ones vectors or matrices.

PROBLEM 1: (10 POINTS) - GEOMETRIC PROPERTIES OF THE OBJECTIVE FUNCTION f

Given $\mathbf{x} \in \mathbb{R}^p$, we denote by $\mathbf{I}_L$, $\mathbf{I}_Q$ and $\mathbf{I}_0$ $n \times n$ matrices such that $\mathbf{I}_L(i,i) = 1$ if $b_i(\mathbf{a}_i^T\mathbf{x}+\mu) < 1-h$, $\mathbf{I}_Q(i,i) = 1$ if $|1-b_i(\mathbf{a}_i^T\mathbf{x}+\mu)| \leq h$, $\mathbf{I}_0(i,i) = 1$ if $b_i(\mathbf{a}_i^T\mathbf{x}+\mu) > 1+h$ and 0 otherwise. Set $\mathbf{A} := [\mathbf{a}_1, \cdots, \mathbf{a}_n]^T$ and $\mathbf{b} := [b_1, \cdots, b_n]^T$. For convenience, we also set $\tilde{\mathbf{A}} := [b_1\mathbf{a}_1, \cdots, b_n\mathbf{a}_n]^T$.

(a) (5 points) Do necessary calculations to show that

$$\nabla f(\mathbf{x}) = \lambda\mathbf{x} + \frac{1}{4nh}\tilde{\mathbf{A}}^T\mathbf{I}_Q[\tilde{\mathbf{A}}\mathbf{x} - (1+h)\mathbf{1}] - \frac{1}{2n}\tilde{\mathbf{A}}^T\mathbf{I}_L\mathbf{1}$$

and then explain that ∇f is L-Lipschitz continuous with $L = \lambda + \frac{1}{4nh}\|\mathbf{A}^T\| \cdot \|\mathbf{A}\|$.

Hint: Observe that $L = 0$ for the linear part, i.e. when $b_i(\mathbf{a}_i^T\mathbf{x}+\mu) < 1-h$. Hence, it is sufficient to compute L for the quadratic region, i.e. $|1 - b_i(\mathbf{a}_i^T\mathbf{x}+\mu)| \leq h$, by assuming that all the samples $\mathbf{a}_i$ lie in that region. Use the fact that $\|\mathbf{A}\| = \|\tilde{\mathbf{A}}\|$, $\|\mathbf{A}^T\| = \|\tilde{\mathbf{A}}^T\|$, and

$$\lambda\mathbf{x} + \frac{1}{4nh}\tilde{\mathbf{A}}^T\mathbf{I}_Q[\tilde{\mathbf{A}}\mathbf{x} - (1+h)\mathbf{1}] = \lambda\mathbf{x} + \frac{1}{4nh}\tilde{\mathbf{A}}^T[\tilde{\mathbf{A}}\mathbf{x} - (1+h)\mathbf{1}]$$

(b) (3 points) Suppose that $\mathbf{I}_Q = \mathbb{I}$ where $\mathbb{I}$ is the identity matrix. Explain why f is twice-differentiable at $\mathbf{x}$ and do necessary calculations to show that

$$\nabla^2 f(\mathbf{x}) = \lambda\mathbb{I} + \frac{1}{4nh}\mathbf{A}^T\mathbf{A}. \tag{4}$$

Note that f is not twice differentiable at $\mathbf{x}$ if $\mathbf{I}_Q = \mathbb{I}$ does not hold. However, we can still use the following

$$\nabla^2 f(\mathbf{x}) = \begin{cases} \lambda\mathbb{I} + \frac{1}{4nh}\mathbf{A}^T\mathbf{A}, & \text{if } f \text{ is twice-differentiable at } \mathbf{x}, \\ \mathbf{0}, \text{otherwise.} \end{cases}$$

as Hessian in Newton method.

(c) (2 points) Show that f is λ-strongly convex.

2 Numerical methods for linear support vector machine

Our aim in the remainder of this homework is to implement different optimization algorithms for solving the regularized support vector machine problem (3). We will re-use the breast-cancer dataset of the Recitation 1 consisting of $n = 546$ samples, each with $p = 10$ features. In all experiments, $\lambda = 0.0001$. You can write the codes either in Matlab or Python. Here are the descriptions of the codes:

Matlab	Python	Description
`compute_error.m`	`commons.compute_error`	compute errors
`Oracles.m`	`commons.Oracles`	return function values, gradients, stochastic gradients, Hessian
`GD.m`	`algorithms.GD`	Gradient Descent
`GDstr.m`	`algorithms.GDstr`	Gradient Descent (strongly convex)
`AGD.m`	`algorithms.AGD`	Accelerated Gradient Descent
`AGDstr.m`	`algorithms.AGDstr`	Accelerated Gradient Descent (strongly convex)
`LSGD.m`	`algorithms.LSGD`	Gradient Descent with line search
`LSAGD.m`	`algorithms.LSAGD`	Accelerated Gradient Descent with line search
`AGDR.m`	`algorithms.AGDR`	Accelerated Gradient Descent with restart
`LSAGDR.m`	`algorithms.LSAGDR`	Accelerated Gradient Descent (line search + restart)
`QNM.m`	`algorithms.QNM`	Quasi-Newton Method
`NM.m`	`algorithms.NM`	Newton Method
`SGD.m`	`algorithms.SGD`	Stochastic Gradient Descent
`SAG.m`	`algorithms.SAG`	Stochastic Averaging Gradient
`SVR.m`	`algorithms.SVR`	Stochastic Gradient Descent (Variance Reduction)
`SVM.m`	`SVM.py`	Main code

General guides:

- Your are required to complete the missing parts in the codes, indicated by markers *YOUR CODES HERE*.
- The scripts `SVM.m` (Matlab) and `SVM.py` (Python) are to test your results. For instance, change `GD_option` = 1 if you want to test `GD` and similarly to the other algorithms. By default these values are 0.
- Put the resulting figures and record the resulting ratios of incorrectly classified points in the test data set in your report.
- For Python coders: in provided codes, an 1D- vector of length m in Matlab is understood to be an array of size $m \times 1$ while it is understood to be a Numpy array of shape $(m,)$. If you choose to code in Python, you might have to have some reshape operations when you work with Quasi-Newton method.

2.1 First order methods for linear support vector machine

The optimality condition of (3) is

$$\nabla f(\mathbf{x}^\star) = \lambda \mathbf{x}^\star + \frac{1}{4nh}\tilde{\mathbf{A}}^T \mathbf{I}_Q[\tilde{\mathbf{A}}\mathbf{x}^\star - (1+h)\mathbf{1}] - \frac{1}{2n}\tilde{\mathbf{A}}^T \mathbf{I}_L \mathbf{1} = \mathbf{0}. \tag{5}$$

Condition (5) is in fact the necessary and sufficient condition for $\mathbf{x}^\star$ to be optimal for (3). We can equivalently write this condition as

$$\mathbf{x}^\star = \mathbf{x}^\star - \mathbf{B}\nabla f(\mathbf{x}^\star) \tag{6}$$

for any symmetric, positive definite matrix $\mathbf{B}$. This is a fixed-point formulation of (3), which will be used to develop gradient and Newton-type methods.

PROBLEM 2: (45 POINTS) - FIRST ORDER METHODS FOR SVM

Notice that choosing $\mathbf{B} = \alpha\mathbb{I}$ with $\alpha > 0$ in the formulation (6) suggests using the gradient descent method to solve (3). In this problem, you will implement various variants of gradient descent algorithm. We have defined 3 input arguments (see the documentations in `Oracles.m` and `commons.Oracles` to know how to use them in your codes):

1. **fx** : The function that characterizes the objective to be minimized;
2. **gradf** : The function that characterizes the gradient of the objective function **fx**;
3. **parameter**: The structure (Matlab) or the dictionary (Python) that includes the fields `maxit` (number of iterations), `x0` (initial estimate), `strcnvx` (strongly convex constant) and `Lips` (Lipschitz constant).

(a) (5 points) The main ingredients of the gradient descent algorithm are the descent direction (or search direction) $\mathbf{d}^k$ and step-size α_k. In this part, we consider the gradient descent algorithm with constant step size $1/L$, i.e., $\alpha_k = 1/L$ for any $k = 0, 1, \dots$. Recall the L computed from Problem 3.(a).

Implement this algorithm by completing `GD.m` or `algorithms.GD`.

Note that the objective function is strongly convex. Therefore, we can select the constant step-size α as $2/(L+\lambda)$ to get a faster convergence rate. Implement this variant of the gradient descent method, by completing the code in `GDstr.m` or `algorithms.GDstr`.

(b) (10 points) We can accelerate the above gradient descent algorithm as follows ($t_0 = 1$):

$$\begin{cases} \mathbf{x}^{k+1} & := \mathbf{y}^k - \alpha_k \nabla f(\mathbf{y}^k), \\ t_{k+1} & := \frac{1}{2}(1 + \sqrt{1 + 4t_k^2}) \\ \mathbf{y}^{k+1} & := \mathbf{x}^{k+1} + \frac{t_k - 1}{t_{k+1}}(\mathbf{x}^{k+1} - \mathbf{x}^k). \end{cases}$$

Details of this algorithm can be found in Lecture 5.

Implement this algorithm with constant step size $\alpha = 1/L$, by completing the missing parts in `AGD.m` or function `AGD` in `algorithms.AGD`.

Note that the objective function is strongly convex. Therefore, we can use the accelerated gradient algorithm for strongly convex objectives to get faster convergence. This variant can be summarized as follows:

$$\begin{cases} \mathbf{x}^{k+1} & := \mathbf{y}^k - \alpha_k \nabla f(\mathbf{y}^k), \\ \mathbf{y}^{k+1} & := \mathbf{x}^{k+1} + \frac{\sqrt{L} - \sqrt{\lambda}}{\sqrt{L} + \sqrt{\lambda}}(\mathbf{x}^{k+1} - \mathbf{x}^k). \end{cases}$$

Implement this variant of the accelerated gradient method by completing the code in `AGDstr.m` or `algorithms.AGDstr`.

(c) (15 points) We can obtain better performance by considering a line search procedure, which adapts the step-size α_k to the local geometry. The line-search strategy to determine the stepsize α_k for the standard GD

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha_k \nabla f(\mathbf{x}^k).$$

is the follow: At the step k, let $\mathbf{x}^k$ be the current iteration and $\mathbf{d}^k = -\nabla f(\mathbf{x}^k)$ be a given descent direction, and perform:

- Set $L_0 = L$.
- At each iteration, set $L_{k,0} = \frac{1}{2}L_{k-1}$, where k is the iteration counter.
- Using a for loop, find the minimum integer $i \geq 0$ that satisfies $f\left(\mathbf{x}^k + \frac{1}{2^i L_{k,0}}\mathbf{d}^k\right) \leq f(\mathbf{x}^k) - \frac{1}{2^{i+1}L_{k,0}}\|\mathbf{d}^k\|^2$.
- Set $L_k = 2^i L_{k,0}$ and use the step-size $\alpha_k := \frac{1}{L_k}$ (i.e., use the new estimate that you have used in the line-search: $\mathbf{x}^k + \frac{1}{2^i L_{k,0}}\mathbf{d}^k$).

Complete the missing parts in the files `LSGD.m` or `algorithms.LSGD` in order to implement gradient descent with line-search.

We now incorporate a line-search with accelerated gradient method in (b) as follow: at step k, one has the current iterations $\mathbf{x}^k$ together with *intermediate variable* $\mathbf{y}^k$ and its corresponding direction $\mathbf{d}^k = -\nabla f(\mathbf{y}^k)$. **Note that the intermediate variable is then used in the gradient step and hence the line-search will be performed on it to determine the stepsize**.

- Perform a line-search strategy as above with respect to $\mathbf{y}^k$ and direction $\mathbf{d}^k$ to determine the step-size α_k as follows:
  - Set $L_0 = L$.
  - At each iteration, set $L_{k,0} = \frac{1}{2}L_{k-1}$, where k is the iteration counter.
  - Using a for loop, find the minimum integer $i \geq 0$ that satisfies $f\left(\mathbf{y}^k + \frac{1}{2^i L_{k,0}}\mathbf{d}^k\right) \leq f(\mathbf{y}^k) - \frac{1}{2^{i+1}L_{k,0}}\|\mathbf{d}^k\|^2$.
  - Set $L_k = 2^i L_{k,0}$ and use the step-size $\alpha_k := \frac{1}{L_k}$.
- Update the next iterations:

$$\begin{cases} \mathbf{x}^{k+1} & := \mathbf{y}^k - \alpha_k \nabla f(\mathbf{y}^k), \\ t_{k+1} & := \frac{1}{2}(1 + \sqrt{1 + 4\frac{L_k}{L_{k-1}}t_k^2}) \\ \mathbf{y}^{k+1} & := \mathbf{x}^{k+1} + \frac{t_k - 1}{t_{k+1}}(\mathbf{x}^{k+1} - \mathbf{x}^k). \end{cases}$$

Complete the missing parts in the files `LSAGD.m` or `algorithms.LSAGD` in order to implement accelerated gradient descent with line-search.

(d) (15 points) The accelerated gradient method is non-monotonic, so it can be oscillatory, i.e. $f(\mathbf{x}^{k+1}) \nleq f(\mathbf{x}^k)$ for all $k \geq 0$. To prevent such behavior, we can use the so-called adaptive restart strategy. Briefly, one such strategy can be explained as follows: At each iteration, whenever $\mathbf{x}^{k+1}$ is computed, we evaluate $f(\mathbf{x}^{k+1})$ and compare it with $f(\mathbf{x}^k)$:

- If $f(\mathbf{x}^k) < f(\mathbf{x}^{k+1})$, restart the iteration, i.e., recompute $\mathbf{x}^{k+1}$ by setting $\mathbf{y}^k := \mathbf{x}^k$ and $t_k := 1$;
- Otherwise, let the algorithm continue.

This strategy requires the evaluation of the function value at each iteration, which increases the computational complexity of the overall algorithm. Implement the adaptive restart strategy which uses the function values for the accelerated gradient algorithm with constant step-size $\alpha_k = 1/L$ by completing the files `AGDR.m` or function `AGDR` in `algorithms.AGDR`.

Incorporate the line-search, acceleration and function values restart by completing `LSAGDR.m` or function `LSAGDR` in `algorithms.LSAGDR`.

Hint: Note that while the restart is executed on $\mathbf{x}^k$, the line-search strategy is executed on $\mathbf{y}^k$ and the direction $\mathbf{d}^k = -\nabla f(\mathbf{y}^k)$ to determine the stepsize and hence, use line-search each time you encounter an intermediate variable $\mathbf{y}^k$.

2.2 Second order method for SVM

In this problem, you will implement two second-order methods to solve SVM including Newton method and Quasi-Newton method. We have defined 4 input arguments (see the documentations in `Oracles.m` and `commons.Oracles` to know how to use them in your codes):

1. **fx** : The function that characterizes the objective to be minimized;
2. **gradf** : The function that characterizes the gradient of the objective function **fx**;
2. **hessf** : The function that characterizes the hessian of the objective function **fx**;
4. **parameter**: The structure (Matlab) or the dictionary (Python) that includes the fields `maxit` (number of iterations), `x0` (initial estimate), `strcnvx` (strongly convex constant) and `Lips` (Lipschitz constant).

PROBLEM 3: (20 POINTS) - SECOND ORDER METHODS FOR SVM

(a) (10 points) The Newton method can be briefly described as follows (see Lecture 6 for more details):

- **Compute the search direction:** The Newton search direction can be computed by:

$$\mathbf{d}_n^k := \arg\min_{\mathbf{d}} \{\nabla f(\mathbf{x}^k)^T \mathbf{d} + (1/2)\mathbf{d}^T \nabla^2 f(\mathbf{x}^k)\mathbf{d}\} = -\nabla^2 f(\mathbf{x}^k)^{-1} \nabla f(\mathbf{x}^k). \tag{7}$$

  Since the computation of the inverse of the Hessian is costly, one can implement some linear algebra routines to solve the following linear system:

$$\nabla^2 f(\mathbf{x}^k)\mathbf{d}_n^k = -\nabla f(\mathbf{x}^k). \tag{8}$$

  This linear system is positive definite. Therefore, we can use the conjugate gradient method to solve it.

- **Compute the step size:** The step-size α_k can be computed via a line-search procedure as in the standard gradient method.
  - Set an initial estimate of an appropriate step-size α_0 (use $\alpha_0 = 10$), and choose $\kappa := 0.1$;
  - At each iteration, set $\alpha_{k,0} = \theta\, \alpha_{k-1}$, where k is the iteration counter (use $\theta = 64$);
  - Using a for loop, find the minimum integer $i \geq 0$ that satisfies $f\left(\mathbf{x}^k + \frac{\alpha_{k,0}}{2^i}\mathbf{d}^k\right) \leq f(\mathbf{x}^k) + \kappa \frac{\alpha_{k,0}}{2^{i+1}} \nabla f(\mathbf{x}^k)^T \mathbf{d}^k$;
  - Take $\alpha_k = \frac{\alpha_{k,0}}{2^i}$.

- **Update the iteration:** Given $\mathbf{d}_n^k$ and α_k, we can update the estimate as

$$\mathbf{x}^{k+1} := \mathbf{x}^k + \alpha_k \mathbf{d}_n^k.$$

Implement the Newton method by completing `NM.m` or `algorithms.NM`. You can use the built-in MATLAB function `pcg` (preconditioned conjugate gradient method) or function `spsolve` found in `scipy.sparse.linalg.linsolve` for Python to solve (8).

(b) (10 points) Since the number of data points n is usually much larger than the number of parameters p in (3), evaluating the full Hessian $\nabla^2 f(\mathbf{x}^k)$ is expensive. To overcome this bottleneck, we can apply the quasi-Newton method, where $\nabla^2 f(\mathbf{x}^k)$ is approximated by a positive definite matrix $\mathbf{H}_k$, which can be computed via the BFGS (Broyden - Fletcher - Goldfarb - Shanno) update:

$$\mathbf{H}_{k+1} := \mathbf{H}_k - \frac{(\mathbf{H}_k\mathbf{s}_k)(\mathbf{H}_k\mathbf{s}_k)^T}{\mathbf{s}_k^T\mathbf{H}_k\mathbf{s}_k} + \frac{\mathbf{v}_k\mathbf{v}_k^T}{\mathbf{s}_k^T\mathbf{v}_k},$$

where $\mathbf{s}_k := \mathbf{x}^{k+1} - \mathbf{x}^k$, $\mathbf{v}_k := \nabla f(\mathbf{x}^{k+1}) - \nabla f(\mathbf{x}^k)$ and $\mathbb{H}_0 = \mathbb{I}$, the identity matrix. However, in order to avoid the evalution of (8), we can directly work with $\mathbf{B}_k = \mathbf{H}_k^{-1}$ using the following update rule:

$$\mathbf{B}_{k+1} := \mathbf{B}_k - \frac{(\mathbf{B}_k\mathbf{v}_k)(\mathbf{B}_k\mathbf{v}_k)^T}{\mathbf{v}_k^T\mathbf{B}_k\mathbf{v}_k} + \frac{\mathbf{s}_k\mathbf{s}_k^T}{\mathbf{s}_k^T\mathbf{v}_k}.$$

Use this scheme with $\mathbf{B}_0 := \mathbb{I}$ to compute the quasi-Newton direction, $\mathbf{d}_n^k := -\mathbf{B}_k\nabla f(\mathbf{x}^k)$, and complete `QNM.m` or `algorithms.QNM` to implement the quasi-Newton method for solving (3) with line-search as in NM method to determine the step-size α_k.

2.3 Stochastic gradient method for SVM

In this problem, you will implement three versions of stochastic gradient descent method to solve SVM. We have defined 4 input arguments (see the documentations in `Oracles.m` and `commons.Oracles` to know how to use them in your codes):

1. **fx** : The function that characterizes the objective to be minimized;
2. **gradf** : The function that characterizes the gradient of the objective function **fx**;
2. **gradfsto** : The function that characterizes the stochastic gradient of the objective function **fx**;
4. **parameter**: The structure (Matlab) or the dictionary (Python) that includes the fields `maxit` (number of iterations), `x0` (initial estimate), `strcnvx` (strongly convex constant), `Lmax` (see definition below) and `no0functions` (number of functions).

PROBLEM 4: (25 POINTS) - STOCHASTIC GRADIENT METHODS FOR SVM

To use the stochastic gradient descent, we recast (3) as follow

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}\underbrace{\left\{\frac{1}{2}g_i(\mathbf{x},0) + \frac{\lambda}{2}\|\mathbf{x}\|^2\right\}}_{f_i(\mathbf{x})}. \tag{SVM}$$

Let $\mathbf{1}_{\{predicate\}} = 1$ if the predicate is true, and 0 otherwise. Similarly as in Problem 1, we can deduce that

$$\nabla f_i(\mathbf{x}) = \lambda\mathbf{x} + \mathbf{1}_{\{|1-b_i a_i^T\mathbf{x}|\le h\}}\frac{1}{4h}\mathbf{a}_i(\mathbf{a}_i^T\mathbf{x} - b_i(1+h)) - \mathbf{1}_{\{b_i a_i^T\mathbf{x}<1-h\}}\frac{1}{2}b_i\mathbf{a_i}.$$

Consider the following stochastic gradient update: at the iteration k, pick $i_k \in \{1,\dots,n\}$ uniformly at random and define

$$\mathbf{x}^{k+1} := \mathbf{x}^k - \alpha_k\nabla f_{i_k}(\mathbf{x}^k). \tag{SGD}$$

(a) (5 points) Show that $\nabla f_{i_k}(\mathbf{x})$ is an unbiased estimation of $\nabla f(\mathbf{x})$. Explain why ∇f_{i_k} is Lipschitz continuous with $L(f_{i_k}) = \frac{1}{4h}\|\mathbf{a}_{i_k}\|^2 + \lambda$. As a hint, recall how we upper bounded L in Problem 1. Set $Lmax = \max_{i\in\{1,\dots,n\}} L(f_i)$

(b) (5 points) We can use the standard stochastic gradient descent (SGD) to solve (SVM). Complete the following codes in Matlab: `SGD.m` or `algorithms.SGD` using the following stepsize rule $\alpha_k = \frac{1}{k}$.

(c) (7 points) Consider the following stochastic averaging gradient method (SAG) to solve (SVM):

$$\begin{cases}\text{pick } i_k \in \{1,\dots,n\} \text{ uniformly at random}\\ \mathbf{x}^{k+1} := \mathbf{x}^k - \frac{\alpha_k}{n}\sum_{i=1}^n \mathbf{v}_i^k,\end{cases}$$

where

$$\mathbf{v}_i^k = \begin{cases}\nabla f_i(\mathbf{x}^k) & \text{if } i = i_k,\\ \mathbf{v}_i^{k-1} & \text{otherwise.}\end{cases}$$

Complete the following codes in Matlab: `SAG.m` or `algorithms.SAG` using the stepsize $\alpha_k = \frac{1}{16Lmax}$ and $\mathbf{v}^0 = \mathbf{0}$.

(d) (8 points) We can get faster convergence rate for SGD by using the following version of SGD with variance reduction:

$$\begin{cases}\tilde{\mathbf{x}} = \mathbf{x}^k, \mathbf{v}^k = \nabla f(\tilde{\mathbf{x}}), \tilde{\mathbf{x}}^0 = \tilde{x}\\ \text{For } l = 0,\dots,q-1:\\ \quad \text{Pick } i_l \in \{1,\dots,n\} \text{ uniformly at random}\\ \quad \mathbf{v}^l = \nabla f_{i_l}(\tilde{\mathbf{x}}^l) - \nabla f_{i_l}(\tilde{\mathbf{x}}) + \mathbf{v}^k\\ \quad \tilde{\mathbf{x}}^{l+1} := \tilde{\mathbf{x}}^l - \gamma\mathbf{v}^l\\ \mathbf{x}^{k+1} = \frac{1}{q}\sum_{l=0}^{q-1}\tilde{\mathbf{x}}^{l+1}.\end{cases}$$

Complete the following codes in Matlab `SVR.m` or the following codes in Python `algorithms.SVR` with the following rules: $\gamma = 0.01/Lmax$ and $q = [1000 * Lmax]$, i.e., q is the integer part of $1000 * Lmax$.

3 Guidelines for the preparation and the submission of the homework

Work on your own. Do not copy or distribute your codes to other students in the class. Do not reuse any other code related to this homework. Here are few warnings and suggestions for you to prepare and submit your homework.

- This homework is due at 9:00AM, 5th of November, 2018.
- Submit your work before the due date. Late submissions are not allowed and you will get 0 point from this homework if you submit it after the deadline.
- Your final report should include detailed answers and it needs to be submitted in PDF format. The PDF file can be a scan or a photo. Make sure that it is readable. The results of your simulations should also be presented in the final report with clear explanations.
- You can use MATLAB or Python for the coding exercises. We provide some incomplete MATLAB and Python scripts. You need to complete the missing parts to implement the algorithms. Depending on your implementation, you might also want to change some written parts and parameters in the provided codes. In this case, indicate clearly your modifications on the original code, both inside the codes with comments, and inside your written report. Note that you are responsible for the entire code you submit.
- Your codes should be well-documented and they should work properly. Make sure that your code runs without errors. If the code you submit does not run, you will not be able to get any credits from the related exercises.
- Compress your codes and your final report into a single ZIP file, name it as `ee556_2018_hw1_NameSurname.zip`, and submit it through the moodle page of the course.
- Discussing concepts with other students is OK; however, each homework exercise should be attempted and completed individually. You should write your report on your own, with your own words and understanding. Your reports and your codes will be checked thoroughly. Reports with overly similar unjustified statements will be considered as copying, and copying and cheating will not be tolerated. The first time results in zero point for the corresponding homework, and the second time results in zero point for the whole course.