Computational Machine Learning, Fall 2015
Homework 4: stochastic gradient algorithms

Due: Tuesday, November 24th, 2015, before 11:59pm (submit via email)
Preparation:

- install the software packages and material related to the Scikit-Learn lecture available here `https://github.com/amueller/nyu_ml_lectures`
- read Chapter 14 in [2] on gradient descent, stochastic gradient descent, and variants thereof

Instructions: submit the link to your GitHub repository including your Jupyter notebooks.

1 Prologue

The purpose of the homeworks is to provide guidance for you to implement *yourself* useful machine learning algorithms, and use the real-world data related to your project as a playground to use your own codes and observe their performance.

In this homework, you will first implement linear support vector machines, using regular gradient descent and using stochastic gradient descent. Then, you will implement a new feature representation and use it on the real-world data related to your project. All the support Python code needed was presented in the Scikit-Learn lectures. Take advantage of this assignment to make progress on your project write-up.

It is essential that you play by the rules and implement *your own codes*, genuinely and pouring your heart into it. *Should you have any question on this homework assignment, please feel free to post it on Piazza.*

2 Linear Support Vector Machines

In this section, you will work on Linear Support Vector Machines (Linear SVMs), described in [2]. You will first implement Linear SVMs using regular gradient descent for small sample sizes. Then, you will implement Linear SVMs using stochastic gradient descent, for large sample sizes.

2.1 Linear SVMs with gradient descent

First of all, read carefully Chapter 14 in [2], as well as:

`http://scikit-learn.org/stable/modules/sgd.html`

We consider here linear support vector machines (Linear SVMs). Let $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, +1\}$ be the training set. Linear SVMs learn a classifier $g(x) = \text{sign}(f(x))$ with $f(x) = w^T x + b$, where (w, b) are solutions of the convex optimization problem

$$\min_{w,b} \|w\|^2 + \frac{C}{n} \sum_{i=1}^{n} \ell_{\text{hinge}}(y_i, f(x_i)) \tag{1}$$

and for any $y, t \in \mathbf{R}$ we have

$$\ell_{\text{hinge}}(y, t) := \max(0, 1 - yt) \ . \tag{2}$$

Two remarks are in order, before you implement your own algorithm to solve the Linear SVM convex optimization problem.

2.1.1 Making our job easier

Intercept First, the Linear SVM convex optimization problem involves two variables, resp. the weight vector w and the intercept (aka bias) b. Solving an optimization involving only one variable would be more convenient. A simple strategy consists in assuming that the training set is properly re-centered, so that there is no need for an intercept, and we assume beforehand that $b = 0$. While such a strategy is rather effective for *balanced datasets*, that is training sets that have roughly equal number of datapoints for each class, it may fall short for unbalanced datasets.

Another common strategy for linear methods is to add an extra dimension to x that is always a fixed value, say 1. For $i = 1, \ldots, n$, we replace x_i by $(x_i, 1)$. Now, we have a training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^{d+1} \times \{-1, +1\}$. We learn a classifier $g(x) = \text{sign}(f(x))$ with $f(x) = w^T x$, where w is a solution of

$$\min_{w \in \mathbf{R}^{d+1}} \|w\|^2 + \frac{C}{n} \sum_{i=1}^{n} \ell_{\text{hinge}}(y_i, f(x_i)) \ . \tag{3}$$

We shall use this strategy from now on. Still, we shall always assume that the *training set was re-centered and re-normalized before running any algorithm.*

Smoothed hinge loss Second, the empirical risk term of the Linear SVM convex optimization problem is non-differentiable, since the linear hinge loss is non-differentiable. Dealing with non-differentiable optimization problems is beyond the scope of the course. I strongly recommend M. Overton's course next semester for those interested. So we need a way to circumvent the non-differentiability of the hinge loss.

Here is a simple and effective strategy. The hinge loss is non-differentiable because of its "kink" when $yt = 1$. We just need to smooth this kink, that is replace that part of the function with a smooth one. Define the huberized hinge loss[1] or smoothed hinge loss as

$$\ell_{\text{huber-hinge}}(y, t) := \begin{cases} 0 & \text{if} \quad yt > 1 + h \\ \frac{(1+h-yt)^2}{4h} & \text{if} \quad |1 - yt| \leq h \\ 1 - yt & \text{if} \quad yt < 1 - h \end{cases} \tag{4}$$

where h is a parameter to set, typically between 0.01 and 0.5

[1]Named after Peter J. Huber, who used a similar smoothing strategy in the context of regression.

From now on, we shall consider the smooth convex optimization problem

$$\min_{w \in \mathbf{R}^{d+1}} F(w) := \|w\|^2 + \frac{C}{n} \sum_{i=1}^{n} \ell_{\text{huber-hinge}}(y_i, f(x_i)) \,. \tag{5}$$

Plot on a same plot, with different colors, the misclassification error loss, the (regular) hinge loss, and the huberized hinge loss. Explain how the huberized hinge loss relates to the regular hinge loss and to the misclassification error loss.

Keep in mind that a surrogate loss is meant to be computationally tractable upper-bounding function to the original loss (here the misclassification error loss). Since we want to *minimize the objective*, and since we are interested in the classification error performance, we do not really care which loss function we choose. All we are interested in is how tight an upper-bound is our surrogate loss function with respect the loss we actually care about, that is the misclassification error loss. The huberized hinge loss is almost as tight an upper-bound as the hinge loss. Yet, the huberized hinge loss is differentiable, whereas the hinge loss is not. The huberized hinge loss is nicer than hinge loss, since we can compute gradients to optimize the objective.

Write a mathematical proof that the huberized hinge loss is differentiable (hint: the huberized hinge loss is defined piecewise). Write the analytic expression(s) of the gradient of the huberized hinge loss. Is the gradient huberized hinge loss Lipschitz-continuous over a particular domain? (hint: the huberized hinge loss is defined piecewise) If the gradient huberized hinge loss is Lipschitz-continuous, write a mathematical proof. Then, give an upper-bound on the Lipschitz-continuous parameter L, and write the corresponding mathematical proof. If gradient huberized hinge loss is not Lipschitz-continuous, write a mathematical proof. For all these questions, use Chapters 13-14 in [2] as reference for basic properties of differentiable and Lipschitz-continuous functions.

2.1.2 Analytic expressions

Define $X \in \mathbf{R}^{n \times d+1}$ be examples matrix (design matrix), where the i'th row of X is x_i. Let $y = (y_1, \ldots, y_n)^T \in \mathbf{R}^{n \times 1}$ be a the labels (or responses/outputs).

compute_obj
compute_grad

Write the objective function $F(w)$ as an matrix/vector expression, without using an explicit summation sign. Then, write down an expression for the gradient of F.

In our search for a w that minimizes F, suppose we take a step from w to $w+\eta\Delta$, where $\Delta \in \mathbf{R}^{d+1}$ is a unit vector giving the direction of the step, and $\eta \in \mathbf{R}$ is the length of the step. Write down an approximate xpression for $F(w + \eta\Delta) - F(w)$. Then, write down the expression for updating θ in the gradient descent algorithm. Let η be the step size. Write a function `compute_obj`, to compute $F(w)$ for a given w. Write a function function `compute_grad`, to compute $\nabla_w F$.

2.1.3 Numerical checks

For many optimization problems, coding up the gradient correctly can be tricky. Luckily, there is a nice way to numerically check the gradient calculation. This only allows you to find out gross mistakes in your analytic calculations. *This is not a mathematical proof.*

If $F : \mathbf{R}^d \to \mathbf{R}$ is differentiable, then for any direction vector $\Delta \in \mathbf{R}^d$, the directional derivative of F at w in the direction Δ is given by:

$$\lim_{\varepsilon \to 0} \frac{F(w + \varepsilon\Delta) - F(\theta - \varepsilon\Delta)}{2\epsilon}$$

We can approximate this derivative by choosing a small value of $\varepsilon > 0$ and evaluating the quotient above. We can get an approximation to the gradient by approximating the directional derivatives in each coordinate direction and putting them together into a vector. See for details

`http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization`

Write a function `grad_checker`. (Optional) Write a generic version of `grad_checker` that will work for any objective function. For instance, you could consider ridge regression (regularised least-squares) and Linear SVMs with squared hinge loss with $\ell_{\text{hinge}}(y,t) := \max(0, 1-yt)^2$. The function should take as parameters a function that computes the objective function and a function that computes the gradient of the objective function.

2.1.4 Gradient Descent

Write a function `my-gradient-descent` that implements the gradient descent algorithm with a constant step-size η described in Sec. 14.1 of [2]. The function is initialised at $w_0 = 0$. The function takes as input η and the maximum number of iterations maxiter. The function returns the output w_T with $T =$ maxiter.

The gradient descent function `my-gradient-descent` should look at the optimization objective F as black-box. In other words, the objective function that gradient descent minimises should not be hard-coded inside the gradient descent function. The gradient descent function should only require `compute_obj` and `compute_grad` as sub-routines[2]. Therefore, as long as the user provides `compute_obj` and `compute_grad` sub-routines, the gradient descent function can work with any convex differentiable objective.

Generate a synthetic data for binary classification. Each class is modelled as a Gaussian distribution, with 500 examples for training and 500 for testing. You get can inspiration from the synthetic data generated in

`http://scikit-learn.org/stable/auto_examples/classification/plot_lda_qda.html#example-classification-plot-lda-qda-py`

Feel free to play with the parameters of the Gaussian distributions (mean, covariance). Make sure the two classes have sufficient overlap. The overlap is controlled by the distance of the means of the two Gaussians.

Normalize your data. You will use here Linear SVM with huberized hinge loss, trained using your gradient descent algorithm. Write a function `my-svm` for Linear SVM, that can used for training (by calling `my-gradient-descent`) and testing. Set $C = 1$ and maxiter $= 1000$. Run experiments for various values of the fixed step-size η. You can take values $0.1 * 1.1^k$ for $k = 1, 2, \ldots$. Visualise the linear separation learned by your Linear SVM. Color differently the two sides of the separating hyperplane. Plot the objective function vs the iterations, as the gradient descent algorithm proceeds. Plot the misclassification error on the training set vs the iterations, and plot the the misclassification error on the testing set vs the iterations. What do you observe? How many iterations do you need to optimise F? How many iterations do you need to reach your best performance in terms of misclassification error on the testing set? What seems to be the best

[2]In the mathematical optimization literature, this is usually referred to as a *first-order oracle*, since the optimization algorithm can query $F(w)$ and $\nabla F(w)$ (first-order derivative) for any w.

Implement backtracking line search
kwargs

setting for η as far as the optimization of the objective function is concerned. Same question but for the misclassification error on the testing set.

The goal of the gradient descent algorithm is to *decrease* the objective, that is have $F(w_{t+1}) < F(w_t)$ where $w_{t+1} = w_t - \eta \nabla F(w_t)$. Implement backtracking line search (google it). Profile your code, optimise in terms of speed. Now, how does it compare to the best fixed step-size you found in terms of number of iterations? How does the extra time to run backtracking line search at each step compare to the time it takes to compute the gradient?

Thanks to backtracking line-search, the problem of setting the step-size is solved. There are two parameters that need to be set: C and maxiter. Setting maxiter = 1000 is somewhat arbitrary, since this is unrelated to the classification performance. You can choose a stopping criterion based on optimization considerations. The optimization/training process can be stopped when no further progress with respect to the objective F seems to be possible, that is when $\|\nabla F\| < \epsilon$, with typically $\epsilon = 10^{-3}$. Still, this is unrelated to the classification performance. You can choose a stopping criterion based on generalization performance considerations. The optimization/training process can be stopped when no further progress with respect to the misclassification error on a validation set seems to be possible, that is when over a short window the misclassification error did not improve (decrease) by a significant factor. Denoting Perf(t) misclassification error on the validation set at iteration t, this translates into stopping the algorithm when $\text{Perf}(t) > \rho \min_{u=t-10,t-9,\ldots,t-1} \text{Perf}(u)$, with typically $\rho = 0.9$.

Add several options to `my-svm` that allow the user to choose between the different stopping criteria: i) maximum number of iterations; ii) optimization-based criterion; iii) generalization-performance-based criterion. For option iii), the validation set is just a random sub-sample of the training set. Run a cross-validation grid-search on C to find the optimal value. Run experiments for that value of C. Comment on your results.

At this point, you have your own implementation of a Linear SVM. Let us compare it with Scikit-Learn's `sklearn.svm.LinearSVC`. Compare your best performance in terms of misclassification error to the best performance after cross-validation of `sklearn.svm.LinearSVC`. Change your synthetic data by moving the two means closer to each other (not to much, of course), to have a strong overlap. Run experiments with `my-svm` and `sklearn.svm.LinearSVC`. Compare generalization performance.

2.2 Stochastic Gradient Descent

Generate a synthetic data for binary classification. Each class is modelled as a Gaussian distribution, with 10^7 examples for training and 10^7 for testing. Run `my-svm` on this dataset. How long does it take to make one iteration?

When the training data set is large, evaluating the gradient of the loss function can take a long time, since it requires looking at each training example to take a single gradient step. In this case, stochastic gradient descent (SGD) can much more effective. In SGD, the gradient of the risk is approximated by a gradient at a single example. The algorithm sweeps through the whole training set one by one, and performs an update for each training example individually. One pass through the data is called an *epoch*. Note that each epoch of SGD touches as much data as a single step of batch gradient descent. It is also important that as we cycle through the training examples, they are in a random order, drawn independently for each epoch.

Write a function `my-sgd` that implements stochastic gradient, as described in the lecture slides. The function is initialised at $w_0 = 0$. The function takes as input η_0 and t_0 and a stopping criterion along the same lines as for the regular gradient descent. Only the counterparts of stopping criteria i) and iii) apply here. The function returns the last iterate w_T.

Plot the value of the objective function (or the log of the objective function if that is more clear) as a function of epoch for several values of η_0 and t_0. Write a "warm-up" function that selects the best value of η_0 and t_0 on a sub-sample. Best here is in terms of generalization performance (misclassification error). Then the stochastic gradient algorithm uses these values η_0 and t_0 to work with the large sample.

Profile your code, improve its speed. Generate a large synthetic dataset. Estimate the amount of time it takes on your computer for a single epoch of SGD. Compare SGD and gradient descent, if your goal is to minimize the total number of epochs (for SGD) or steps (for batch gradient descent), on several synthetic datasets you generates. Which would you choose? If your goal were to minimize the total time, which would you choose?

Launch experiments for several synthetic datasets in R^d, say with $d = 1,024$, with varying number of examples, different overlap between the distributions, different covariance matrices, etc. Set up "races" between your two optimization algorithms (regular gradient descent, stochastic gradient). The winner of the race is the algorithm that achieves the quickest the best generalisation performance on the testing set. For each synthetic dataset, you can count how many times a particular algorithm won the race over several independent replications. To define a situation, you can only use n, d, C, since the other parameters are unknown to the user for real-world datasets. Extract from your experimental results a simple empirical "rule of thumb" that, for any dataset[3], characterized by n (the number of training examples), d (the dimension of the data), C (the regularization parameter), automatically decides which algorithm to use.

3 Project

The previous sections allowed you to write your own Linear SVM. Equipped with these codes and Scikit-Learn's Linear SVM, you can now implement a second approach for your project. Last time, you used k-nearest neighbor classifier. Now you can use your own Linear SVM.

Please use the project-specific instructions you received by email to work on this section.

Document and write down your experiments: settings, results, plots, etc. Should you have any question on this homework assignment, please feel free to post it on Piazza.

4 Feedback

1. Approximately how long did it take to complete this assignment?
2. Is there a concept you feel you still don't quite grasp? Detail what remains unclear.
3. Is there a terminology/concept that has been used multiple times during the course, but you still have no idea what it means?
4. Is there a concept you would love to see covered during the course? (deep learning is not eligible; you can just take my course next semester)

[3]You cannot use the other parameters since they are unknown to the user for real-world datasets.

References

[1] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1027–1035, 2007.

[2] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, 2014. (document), 2, 2.1, 2.1.1, 2.1.4