CSE 250B: Machine learning **Fall 2016**

Worksheet 6 — Algorithms for regression and classification

This worksheet covers the following topics:

- Least-squares and regularized least-squares regression.
- Isotonic regression.
- Variants of the perceptron algorithm.

1. *Least-squares regression.* Suppose we have data $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \in \mathbb{R}^p \times \mathbb{R}$ and we wish to find a vector $w \in \mathbb{R}^p$ such that $y^{(i)} \approx w \cdot x^{(i)}$. This is a *regression problem.*

   A common loss function to use in this context is:

   $$L(w) = \sum_{i=1}^{n} (y^{(i)} - w \cdot x^{(i)})^2.$$

   It is useful to express this function in matrix-vector form. Let X be the $n \times p$ matrix whose rows are the $x^{(i)}$, and let y be the p-dimensional vector whose ith entry is $y^{(i)}$.

   (a) Check that $L(w) = \|y - Xw\|^2$.

   (b) Obtain an expression for the first derivative, $\nabla L(w)$, in terms of X and y.

   (c) By setting this derivative to zero, obtain a closed-form solution for the optimal w. You can assume that the $p \times p$ matrix $X^T X$ is invertible (otherwise, its *Moore-Penrose pseudoinverse* can be used).

   (d) Now use the same methodology to obtain a closed-form solution to the *regularized least-squares regression* problem, given by loss function:

   $$L(w) = \left(\sum_{i=1}^{n} (y^{(i)} - w \cdot x^{(i)})^2 \right) + \lambda \|w\|^2.$$

   Here λ is some constant. This formulation helps avoid overfitting when p is large.

   Note on classification versus regression: we typically think of regression problems as those where the $y^{(i)}$ are continuous-valued, and classification problems as those where the $y^{(i)}$ are discrete. But we could use least-squares (or regularized least-squares) regression to solve a classification problem, where for instance $y^{(i)}$ just takes on two possible values, -1 and $+1$.

2. *Isotonic regression.* In a *line fitting* problem, we have a data set consisting of pairs $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^2$ and we want to draw a line through them. More precisely, we want to find parameters $a, b \in \mathbb{R}$ such that $f(x) = ax + b$ passes as close as possible to the points. We have already seen a least-squares formulation of this.

   In *isotonic regression*, we allow a more general function f. It doesn't have to be a line: it just needs to be monotonically increasing, that is, $f(x) \geq f(x')$ whenever $x \geq x'$.

(a) Here is a training set of six points (x_i, y_i):

$$(4, 20),\ (2, 5),\ (5, 9),\ (3, 7),\ (1, 10),\ (6, 12).$$

Plot these points, and sketch a function $f : \mathbb{R} \to \mathbb{R}$ that is monotonically increasing and that passes through *as many of these points as possible.*

Let's sort the data points so that $x_1 \leq x_2 \leq \cdots \leq x_n$. Monotonicity then means

$$f(x_1) \leq f(x_2) \leq \cdots \leq f(x_n).$$

In fact, we can choose any $f(x_i)$ values that meet this requirement; and we can fill in the rest of the f-curve by, say, linearly interpolating between these points.

How shall we evaluate candidate functions f? In part (a), we used the loss function

$$L_o(f) \ = \ \#\text{ of training points that } f \text{ does not pass through.}$$

Finding the optimal such f is called the *longest increasing subsequence* problem in computer science, and can be solved efficiently. However, we typically prefer to use a different, *least-squares* loss.

Here is a least-squares formulation of our problem: given $x_1 \leq x_2 \leq \cdots \leq x_n$ and corresponding values $y_1, \ldots, y_n$, find $f_1, f_2, \ldots, f_n \in \mathbb{R}$ such that $f_1 \leq f_2 \leq \cdots \leq f_n$ and such that the squared loss

$$L(f) = \sum_{i=1}^{n} (y_i - f_i)^2$$

is minimized. (Here f_i corresponds to $f(x_i)$.)

(b) Show that this is a convex optimization problem.

An elegant approach to solving this problem is the *pool adjacent violators* algorithm. It starts by simply setting $f_i = y_i$ for all i, and then repeatedly fixes any monotonicity violations: any time it finds $f_i > f_{i+1}$, it resets both of them to the average of f_i, f_{i+1} and merges points x_i and x_{i+1}.

Here is the algorithm, given a set of x values and their corresponding $y(x)$.

- Let S be the sorted list of x-values
- For all x in S:
  - Set $f(x) = y(x)$
  - Assign weight $w(x) = 1$
- While there adjacent values $x < x'$ in S with $f(x) > f(x')$:
  - Remove x' from S and set a pointer from it to x
  - Let $f(x) = \frac{w(x)f(x)+w(x')f(x')}{w(x)+w(x')}$
  - Let $w(x) = w(x) + w(x')$

At the end, each of the original x-points either lies in the list S, in which case it receives value $f(x)$, or leads to some $\tilde{x}$ in list S by following pointers, in which case it receives value $f(\tilde{x})$.

(c) Run this algorithm on the small data set of six points from part (a). What values of f does it yield?

3. *Voting Perceptrons.* The perceptron algorithm, as presented in class, will not converge when given data that is not linearly separable.

   One work-around is to simply halt the algorithm after a fixed number (say, T) of passes through the data, and to output the vector w at that time. But this has a drawback: it might be that an update occurred right before termination, and caused the final w to point in a poor direction.

   A better idea is to combine the various w's obtained during the training process. In particular, a vector w that survived for quite a long time is likely to be good. This is the motivation for the *voted perceptron* algorithm. Given data $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \in \mathbb{R}^p \times \{-1, 1\}$, it operates as follows:

   - $\ell = 1$, $c_\ell = 0$
   - $w_1 = 0$
   - Repeat T times:
     - Randomly permute the data points
     - For $i = 1$ to n:
       - If $(x^{(i)}, y^{(i)})$ is misclassified by w_ℓ:
         - $w_{\ell+1} = w_\ell + y^{(i)} x^{(i)}$
         - $c_{\ell+1} = 1$, $\ell = \ell + 1$
       - Else:
         - $c_\ell = c_\ell + 1$

   At the end of this process, we have a collection of linear separators $w_1, w_2, \ldots, w_\ell$ as well as their "survival times" $c_1, \ldots, c_\ell$. To classify a new point x, we take the weighted majority vote:

$$\text{sign}\left(\sum_{j=1}^{\ell} c_j \,\text{sign}(w_j \cdot x)\right).$$

   An alternative rule is the simpler *averaged perceptron*: predict $\text{sign}(w \cdot x)$, for

$$w = \sum_{j=1}^{\ell} c_j w_j.$$

   (a) Implement the voted perceptron algorithm and try it out on the small 2-d data set `data1.txt`. (Remember to add a constant-valued feature to the data.) Try $T = 10$ or thereabouts. Show the final decision boundary. Is it linear?

   (b) [Optional] After a few passes through the data, the number of classifiers ℓ might grow inconveniently large. Can you devise a good way to downsample them so that at most L different values of w are ever stored? Give pseudocode, and redo the previous problem (part (b)) using your technique. Try a higher value of T to assess how well your downsampling is working.

   (c) [Optional] Give pseudocode for the averaged perceptron that avoids keeping track of all w's seen (it should maintain at most two w's). Show the decision boundary it achieves on the small data set, with $T = 10$.

4. *Kernelized perceptron.* Implement the kernel perceptron algorithm for the quadratic and RBF kernels. Show the boundaries obtained for the small data sets `data1.txt` and `data2.txt`. For the RBF kernel, show boundaries for a few settings of the scale parameter σ.