Gradient Descent Based Classifier

In this programming assignment, you will implement a gradient-based classifier for a classification problem with numerical predictive attributes and a discrete classification attribute.

Classifier

The classifier is the same as in problem set 5, and we will use the same notations. The learning algorithm passes a set of c exemplar vectors to the classifier, and the classifier classifies a test point according to the closest exemplar vector.

However, we will modify how the classification algorithm works. First, we will modify the objective function to charge a "cost" only to misclassified points, as is done in the perceptron model. Let G be a tuple of exemplar vectors $G = \langle \vec{g}_1 \ldots \vec{g}_c \rangle$. For a given training example Y where $Y.C = v$, let $\vec{r}_G(Y)$ be the closest exemplar to Y (assume that there are no ties). As in problem set 5, let $\vec{u}(Y)$ be the vector of predictive attributes and let $v = Y.C$, the classification attribute. Thus, if $\vec{r}_G(Y) = \vec{g}_v$, then G is correcly classifying Y, otherwise it is incorrectly classifying Y. Therefore the quantity

$$d^2(\vec{g}_v, \vec{u}(Y)) - d^2(\vec{r}_G(Y), \vec{u}(Y))$$

is zero if G correctly classifies Y and positive if it does not.

We therefore can associate an error cost with Y for classifier G:

$$\text{Cost}_G(Y) = \min(M, d^2(\vec{g}_v, \vec{u}(Y)) - d^2(\vec{r}_G(Y), \vec{u}(Y)))$$

The quantity M is externally provided. The point of putting a maximum on the cost is to make sure that distant outliers do not have too much of an influence.

The objective function for G with respect to a given training set T is the sum of the costs of the instances in Y: $O_T(G) = \sum_{Y \in T} \text{Cost}_G(Y)$.

The classification algorithm is to use gradient descent over this objective function to find a local minimum.

It is easily seen that the objective function is zero if and only if all points in the training set are correctly classified; and that it is continuous and piecewise differentiable.

The negative gradient of O with respect to an exemplar vector g_v is computed as follows:

Let $P_{G,v}(T)$ be the set of all instances Y in T such that $Y.C = v$ but $\vec{r}_G(Y) \neq \vec{g}_v$ and $\text{Cost}_G(Y) < M$; the points that are labelled v but misclassified, and have a cost less than M.

Let $Q_{G,v}(T)$ be the set of all instances Y in T such that $Y.C \neq v$ but $\vec{r}_G(Y) = \vec{g}_v$ and $\text{Cost}_G(Y) < M$; the points that are classified as v but whose label is something else, and and have a cost less than M.

Then the negative gradient of O with respect to $\vec{g}_v$ is given by:

$$-\vec{\nabla}_{g_v}(O) = 2 \cdot (\sum_{Y \in P_{G,v}(T)} \vec{u}(Y) - \vec{g}_v) + 2 \cdot (\sum_{Y \in Q_{G,v}(T)} \vec{r}_G(Y) - \vec{u}(Y))$$

Intuitively, you want to move $\vec{g}_v$ toward data points that should be labelled v but are not, and away from data points that are labelled v but should be labelled something else. However, if a data point is too far from its proper exemplar, then we give it up (for the time being) as hopeless, and don't consider it in the calculations.

Therefore the gradient descent classification algorithm works like this:

```
function gradDescent(training set T; real stepSize; real epsilon; real M) {
    initialize vectors $\vec{g}_1 \dots \vec{g}_c$ in $\mathbb{R}^k$ arbitrarily;          % exemplar points
    previousCost= infinity;
    PrevCost = infinity;
    PrevAccuracy = computeAccuracy($\vec{g}_1 \dots \vec{g}_k, T$);
    loop {
        TotalCost = 0.0;
        for (v = 1 to c) $\vec{n}_v = \vec{0}$;                          % negative gradient
        for (each datapoint $Y$ in $T$) {
            $v = Y.C$
            find $\vec{g}_w$ closest to $\vec{u}(Y)$;
            if ($w \neq v$) {
                Cost = distSquared($\vec{g}_v, \vec{u}(Y)$) − distSquared($\vec{u}(Y) - \vec{g}_w$);
                if (Cost < M) {
                        $\vec{n}_v$ += $\vec{u}(Y) - \vec{g}_v$;
                        $\vec{n}_w$ += $\vec{g}_w - \vec{u}(Y)$;
                        TotalCost += Cost;
                        }
                else TotalCost += M;                          % end if
            }                                                 % end if
        }                                                     % end for
        if (TotalCost < epsilon) return $\vec{g}_1 \dots \vec{g}_c$;          % Close enough.
        if (TotalCost > (1 − epsilon) * PrevCost) return $\vec{g}_1 \dots \vec{g}_c$; % Descent has run out of steam
        for (v = 1 to c) $\vec{h}_v = \vec{g}_v + stepSize * \vec{n}_v$;
        NewAccuracy= computeAccuracy($\vec{h}_1 \dots \vec{h}_k, T$);
        if (NewAccuracy < PrevAccuracy) return $\vec{g}_1 \dots \vec{g}_k$;
        for (v = 1 to c) $\vec{g}_v = \vec{h}_v$;
        PrevCost = TotalCost;
        PrevAccuracy = NewAccuracy
    }                                                         % end loop
}                                                             % end gradDescent
```

Program

The values of parameters `stepSize`, `epsilon` and `M` will be given in command line arguments, as specified below. Another command line argument will be the number of iterations of random restart to run.

The driver program will call `gradDescent` a number of times, with different initializations of the vectors $\vec{g}_1 \dots \vec{g}_c$. On the first pass, they will be initialized to be the centroid of their respective categories, as in problem 1. On the remaining pass, they will be initialized randomly. For each attribute, find the minimum and maximum value in the training set; then sample uniformly between the minimum and maximum.

After the classifier has returned, the fraction of points in the training set classified correctly is computed. (This class of classifiers is too inexpressive for overfitting to be a concern; unless the training set is extremely small, the percentage correct on the training set and on the test set will be pretty much the same.)

Input

The training set should be in a CSV (comma separated values) file. You may assume that there are no null values. The predictive attributes are floating point numbers, and the values of the classification attributes are single lower-case letters 'a' through 'z'. (There are not more than 26 values.) Thus, the sample data in the problem set would be presented in the following input file:

```
1.0,1.0,2.0,a
2.0,1.0,1.0,a
2.0,0.0,1.0,a
0.0,2.0,1.0,b
3.0,2.0,0.0,b
3.0,3.0,0.0,c
0.0,3.0,0.0,c
3.0,2.0,1.0,c
0.0,3.0,3.0,c
```

Command line arguments

Your program `classify` should take five or six command line arguments, in sequence: the name of the file for the training set; the values of `stepSize`, `epsilon` and `M`; the number of random restarts, and, optionally "`-v`" to indicate verbose output.

Thus in Java, the command line call might be `java classify training.csv 0.1 0.01 0.2 100`.

Output

In compact output, just output the fraction of the training set correctly classified in each run, and, at the end, the best accuracy obtained in any run.

In verbose output, output a trace of the sequence of values of the exemplar vectors, and of the successive values of the accuracy.

Example sets

I will produce three synthetic data sets for you to try running your program on.

The first will be the toy training set in problem set 5. I will also generate the correct output for some reasonable setting of the parameters, with `N` = `0` (i.e. just starting from the centroids).

The second will be a data set that is fairly close to being linearly separable; the categories will correspond to spatial regions that are nearly polyhedral, though not perfectly, and there will be some amount of noise (points that are randomly labelled). I would expect the algorithm to do pretty well over this data set.

The third will be a data set that is nowhere near being linearly separable; the categories will correpond to regions that are far from convex, or disconnected, plus there will be noise. The classifier cannot do *well* on this data set; the question is, will it do better than chance?