CS273a Homework #4

Machine Learning: Winter 2015

Due: Tuesday February 24th, 2015

Write neatly (or type) and show all your work!

Please remember to turn in at most two documents, one with any handwritten solutions, and one PDF file with any electronic solutions.

Download the provided Homework 4 code, and replace last week's code (several new functions have been added).

Problem 1: Support Vector Machines

In this problem, we'll train a (separable) SVM using a QP solver. Build your separable binary classification data as in the previous homework (no need to do training/validation this time):

```
iris=load('data/iris.txt');      % load the text file
X = iris(:,1:2); Y=iris(:,end); % get first two features
XA = X(Y<2,:); YA=Y(Y<2);       % get class 0 vs 1
```

Now, use either Matlab's `quadprog` or Octave's `qp` function to solve the SVM quadratic program. Unfortunately, `quadprog` is only available in the Optimization toolbox, so if you don't have access to that and can't use the on-campus lab machines, I suggest using Octave. If you don't want to install Octave, you can likely do this problem on an online version of Octave, e.g.,
`http://www.compileonline.com/execute_matlab_online.php` (You'll need to copy and paste your data in.)

Recall that the primal SVM form is:

$$\min_{w,b} \sum_i w_i^2 \qquad \text{s.t.} \qquad y^{(i)}\left(wx^{(i)} + b\right) \geq 1$$

Manipulate this form until it matches a "standard" form used by `qp` or `quadprog`, e.g.,
`quadprog(H,f,A,b,Aeq,Beq,lb,ub)`
and design the necessary matrices `H,f,A,b`, etc. (These will be a bit different for Octave.) Output the resulting linear parameters θ^*, and use your perceptron classifier from last homework to check that it separates the data and plot its decision boundary using `plotClassify2D`.

Now, we'll solve the dual form instead. Compute the Gram matrix of dot products, $K_{ij} = x^{(i)} \cdot x^{(j)}$. Recall that the dual form is:

$$\max_{\alpha \geq 0} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j \, y^{(i)} y^{(j)} K_{ij} \qquad \text{s.t.} \qquad \sum_i \alpha_i y^{(i)} = 0$$

Identify the support vectors from your solution α^*. Verify that your solution is identical, i.e.,

$$\theta^* = \sum_i \alpha_i^* y^{(i)} x^{(i)}$$

and mark the support vectors' locations in your classification boundary plot.

Problem 2: Decision Trees

We'll use the same data as in our earlier homework: In order to reduce my email load, I decide to implement a machine learning algorithm to decide whether or not I should read an email, or simply file it away instead. To train my model, I obtain the following data set of binary-valued features about each email, including whether I know the author or not, whether the email is long or short, and whether it has any of several key words, along with my final decision about whether to read it ($y = +1$ for "read", $y = -1$ for "discard").

x_1 know author?	x_2 is long?	x_3 has 'research'	x_4 has 'grade'	x_5 has 'lottery'	y $\Rightarrow$ read?
0	0	1	1	0	-1
1	1	0	1	0	-1
0	1	1	1	1	-1
1	1	1	1	0	-1
0	1	0	0	0	-1
1	0	1	1	1	1
0	0	1	0	0	1
1	0	0	0	0	1
1	0	1	1	0	1
1	1	1	1	1	-1

In the case of any ties, we will prefer to predict class +1.

(a) Calculate the entropy of the class variable y

(b) Calculate the information gain for each feature x_i. Which feature should I split on first?

(c) Draw the complete decision tree that will be learned from these data.

Problem 3: Decision Trees on Kaggle

We have a Kaggle in-class competition set up for this quarter at
`http://inclass.kaggle.com/c/how-s-the-weather`
In this problem, we will get familiar with Kaggle, download the data, and build a simple regression tree model to predict. You can use the `treeRegress` class provided to build your regression trees.

First, go to Kaggle and create a username (if you don't have one); use your UCI email address, so that you will be able to join our competition.

Note: Kaggle competitions only let you submit a fixed number of predictions per day, usually $\approx 3-5$, so be careful. We'll use a validation split to decide what hyperparameter choices we think are most promising, and upload only one model.

(a) Split out a validation set from your training data examples, and learn a decision tree regressor on the data. To avoid Matlab complaining, specify a max depth of 20, e.g.,

```
dt = treeRegress(Xt,Yt, 'maxDepth',20);
```

(This might take a bit of time; ≈ 2 minutes on my desktop.) Compute its validation MSE.

(b) Now, try varying the maximum depth parameter (`maxDepth`), which forces the tree to stop after at most that many levels. Test values `0, 1, ..., 15` and compare their performance (both training and test) against the full depth. Is complexity increasing or decreasing with the depth cutoff? Identify whether you think the model begins overfitting, and if so, when. If you use this parameter for complexity control, what depth would you select as best?

(c) Now, using high maximum depth ($d = 20$), use `minParent` to control complexity. Try values `2.∧[3:12]=[8,16,...,4096]`. Is complexity increasing or decreasing as `minParent` grows? Identify when (if) the model is starting to overfit, and what value you would use for this type of complexity control.

(d) Using your best complexity control value (either depth or number of parent data), re-train a model on the full data set, predict on the test data, upload it to Kaggle and report its performance.

Problem 4: Ensembles of Trees

Choose **either part** of this question to answer (your choice): a random forest regressor, which is a bagged ensemble of decision trees; **or** an boosted ensemble of regression trees learned with gradient boosting.

In Matlab, you can store vectors of arbitrary objects (such as different learners, even different types of learners) using cell arrays:

```
ens{i} = treeRegress(Xb,Yb,...)  % save ensemble member "i" in a cell array
% ...
predict(ens{i}, Xv,Yv);          % find the predictions for ensemble member "i"
```

Option 1: Random forests:

Random Forests are bagged collections of decision trees, which select their decision nodes from randomly chosen subsets of the possible features (rather than all features). You can implement this easily in `treeRegress` using option (`'nFeatures'`,n), where n is the number of features to select from (e.g., $n = 50$ or $n = 60$ if there are 90-some features); you'll write a for-loop to build the ensemble members, and another to compute the prediction of the ensemble.

(a) Using your validation split, learn a bagged ensemble of decision trees on the training data and evaluate validation performance. (See the pseudocode from lecture slides.) For your individual learners, use little complexity control (depth cutoff 15+, minParent 8, etc.), since the bagging will be used to control overfitting instead. For the bootstrap process, draw the same number of data as in your training set after the validation split ($N' = N$ in the pseudocode). You may find `bootstrapData.m` helpful, although it is very easy to do yourself. Plot the training and validation error as a function of the number of learners you include in the ensemble, for (at least) $1, 5, 10, 25$ learners. (You may find it more computationally efficient to simply learn 25 ensemble members, and then evaluate the results using fewer of them.)

(b) Now choose an ensemble size and repeat on the full training data, make predictions on the test data, and upload to Kaggle. Report your preformance.

Option 2: Gradient boosting:

Gradient boosted trees are boosted collections of decision trees, which are build sequentially to predict the residual error in the current ensemble. You'll write a for-loop to build the ensemble members, and another to compute the prediction of the ensemble.

(a) Using your validation split, learn a gradient boosted ensemble of decision trees on the training data and evaluate validation performance. (See the pseudocode from lecture slides.) For your individual learners, use very strong complexity control (depth cutoff 2–3, or large minParent, etc.), since the boosting process will be adding complexity to the overall learner. Plot the training and validation error as a function of the number of learners you include in the ensemble, for (at least) $1, 5, 10, 25$ learners. (You may find it more computationally efficient to simply learn 25 ensemble members, and then evaluate the results using fewer of them.)

(b) Now choose an ensemble size and repeat on the full training data, make predictions on the test data, and upload to Kaggle. Report your preformance.