CS274a Homework #3

Introduction to Machine Learning: Fall 2013

Due: Thursday October 31st, 2013

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 3 code, and replace last week's code (a few functions have been updated).

Problem 1: Linear Regression

For this problem we will explore linear regression and the creation of additional features.

(a) Load the "`data/mcycle80.txt`" data set, and split it into 75% / 25% training/test. The first column `data(:,1)` is the scalar feature (x) values; the second column `data(:,2)` is the target value y for each example.

(b) Use the provided `linearRegress` class to create a linear regression predictor of y given x. You can plot the resulting function by simply evaluating the model at a large number of x values, `xs`:

```
lr = linearRegress( Xtr, Ytr );  % create and train model
xs = [0:.01:2]';                 % densely sample possible x-values
ys = predict( lr, xs );          % make predictions at xs
```

Plot the training data along with your learned prediction function in a single plot. Also calculate and report the mean squared error in your predictions on the training and test data.

(c) Try fitting $y = f(x)$ using a polynomial function $f(x)$ of increasing order. Do this by the trick of adding additional polynomial features before constructing and training the linear regression object. You can do this easily yourself; you can add a "constant" feature and a quadratic feature of Xtr with

```
Xtr2 = [Xtr, Xtr.^2];
```

(You could potentially also add the all-ones feature, but this is currently done automatically in the learner.) A function "`fpoly`" is also provided to more easily create such features:

```
XtrP = fpoly(Xtr, degree, false); % create polynomial features up to given degree, no "1" feature
[XtrP, M,S] = rescale(XtrP);      % often a good idea to scale the features
lr = linearRegress( XtrP, Ytr );  % create and train model
XteP = rescale(Xte, M,S);         % apply same scaling transformation to Xtest
```

This snippet also shows a useful feature transformation framework – often we wish to apply some transformation (possibly data-dependent, like scaling) to the features. Ideally, we should then be able to apply this same transform to new, test data when it arrives, so that it will be treated in exactly the same way as the training data. "Feature transform" functions like rescale output their settings, (here, `M,S`), so that they can be reused on subsequent data.

Train models of degree $d = 1, 3, 5, 7, 10, 18$ and (1) plot their learned prediction function $f(x)$ and (2) their training and test errors (on a log scale, `semilogy`). For (1), don't forget to also expand and scale the features of `xs` using `fpoly` and `rescale`. Also, you may want to save the original axes of your plot and re-apply them to each subsequent plot for consistency. For (2), plot the resulting training and test errors as a function of polynomial degree.

(d) The `linearRegress` class can also take a regularization parameter such as I discussed in class. Using $d = 18$, try evaluating at different values of α and plot the training and test MSE. It is typical to explore α in a logarithmic scale, so try $\alpha \in [1e{-}6 \ldots 10]$, e.g., `logspace(-6,1,15)`.

Problem 2: Perceptron Algorithm

In this problem, we'll explore a basic perceptron algorithm on separable and non-separable data. In the next problem, you will extend this implementation to optimize over smooth loss functions.

We'll start by building two binary classification problems, one separable and the other not:

```
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
XB = X(Y>0,:); YB=Y(Y>0);       % get class 1 vs 2
```

(a) Show the two classes in a scatter plot and verify that one is linearly separable.

(b) Write (fill in) the function `@perceptClassify/plot2DLinear.m` so that it plots the two classes of data in different colors, along with the decision boundary (a line). Include the listing of your code in your report.

(c) Build a linear classifier using the basic perceptron algorithm I gave in class:

```
pc = perceptClassify(Xp,Yp, step,nIter);
% you can explore the various options and code
```

Run it on each data set (`Xp = XA` and `Xp=XB`). Using your previous function, the code should plot both the data points and the decision boundary at each iteration, along with the misclassification rate as a function of iteration. When you have decided on some good parameter values, show the error rate plots for both data sets and comment on their behavior and convergence.

(d) You can also set the perceptron weights manually. Instead of running the perceptron algorithm, try computing the coefficients of a linear *regression* of the data. Use these coefficients within your classifier (`setWeights`) and compare its performance.

(e) On the linearly separable data, add another point at $x = (-10, 10)$ with class $y = -1$. Try both the linear regression and perceptron algorithm again. Now how do they compare? What happened?

Problem 3: Logistic Regression (linear classifier with logistic loss)

In this problem you will extend the perceptron classifier code from Problem 1 to optimize the logistic negative log-likelihood loss on a "soft" logistic function prediction.

(a) Copy and rename your `perceptClassify` object class to `logisticClassify`.

(b) Adapt your `train.m` and `predict.m` functions to use stochastic gradient descent on the logistic loss function. For logistic regression, it is easier to use classes 0 vs 1 (compared to using classes +1 vs -1, in the perceptron code). In our notation, let $z = \theta x^{(i)}$ is the linear response of the perceptron, and σ is the standard logistic function

$$\sigma(z) = \big(1 + \exp(-z)\big)^{-1}.$$

The (regularized) logistic negative log likelihood loss function is then

$$J(\theta) = \frac{1}{m} \sum_j -y^{(j)} \log \sigma(\theta x^{(j)T}) \; - \; (1 - y^{(j)}) \log(1 - \sigma(\theta x^{(j)T})) \; + \; \alpha \sum_i \theta_i^2$$

where $y^{(j)}$ is either 0 or 1.

(c) Derive the gradient of the regularized negative log likelihood, and give it in your report. (You will need this in your gradient descent code.) Your stochastic gradient descent code should mirror that of the perceptron algorithm, except you will also need (1) a stopping criterion, for example when J stops changing by less than a small tolerance, and (2) you may want a stepsize that is dependent on the iteration, for example decreasing harmonically, 1/(# of iterations). For this problem, use $\alpha = 0$; you will use it later. In your report, please include the functions that you wrote (at minimum, `train.m`, but possibly a few small helper functions as well).

(d) To test your algorithm, you may want to start with a simple 1D data set:

```
X1 = [0.05 0.24 0.28 0.53 0.61 0.48 0.58 0.76 0.80 0.95]';
Y1 = [ 0    0    0    0    0   1    1    1    1    1]';
lc = logisticClassify(X1,Y1 ... );
```

You can use some of the plotting functions from the perceptron classifier to help you visualize the evolution of your classifier. Specifically, `plot1DLinear` plots the data along with the linear response function θx^T; you can also add the logistic response $\sigma(\theta x^T)$ to visualize how closely it matches the true data. For debugging, I suggest starting with an extremely small step size and verifying that the prediction function evolves in the correct direction. For your report, plot the data points (x, y) along with the final learned linear response θx^T and logistic probability $\sigma(\theta x^T)$ on the same plot.

(e) Run your algorithm on both sets `XA,YA` and `XB,YB` from the previous problem. Plot the decision boundary as your algorithm converges. Again, comment on their behavior, comparing to the behavior of the basic perceptron algorithm. Include the final classifier boundary (and data) in your report.

(f) In your code, at each full iteration of your algorithm (after each time through all the data), compute both your misclassification rate and also the value of the logistic NLL loss $J(\theta)$. Plot them both as a function of the iteration on `XB,YB` and compare their behavior.

(g) You can also use feature expansion to increase the number of features available for your linear classifier.

```
XtrP = fpoly(Xtr, degree, false); % create polynomial features up to given degree, no "1" feature
[XtrP, M,S] = rescale(XtrP);      % often a good idea to scale the features
```

Build your logistic regression classifier on data XB, YB using polynomial features of degree 3. You will be unable to visualize the decision boundary as a line, so show the evolution of your loss function J and use this to debug and set the step size appropriately, etc. Compute your final training and test performance and comment on the results.

(h) Although your linear classifier uses more than two features, since the *original* data are still two-dimensional, you can visualize the classifier boundary in those two dimensions. To do so, define an "implicit function" `transform(x)` that applies the polynomial feature expansion and scaling operations to a two-dimensional input feature vector, and use `plotClassify2D`:

```
transform = @(x) rescale( fpoly(x,degree,false), M,S);  % save transform as implicit function
plotClassify2D( learner, Xtr,Ytr, transform);           % plot decision boundary with xform
```

Comment on the resulting decision boundary. What is its analytic form?

(i) **(For fun, not graded)** You can also use other feature expansion functions. The function `fkitchensink` implements a number of random feature creation methods, for example:

```
[XtrF W] = fkitchensink( Xtr, nFeat, 'sigmoid');  % generate nFeat random transformed features
XteF     = fkitchensink( Xte, nFeat, 'sigmoid',W);  % apply same transforms to Xtest
```

(note that XtrF does not contain the original features anymore). Try building a classifier using these new features (for some values of nFeat) and visualize their classification boundaries. How do they do?

Problem 4: High dimensional data

Now, let's try a classification problem that actually lives in a higher dimension. Load the `mnist` data, consisting of images of handwritten digits:

```
mnist=load('data/mnist.txt');     % load the text file
Y=mnist(:,1); X = mnist(:,2:end); % note different order of data
keep = (Y==0 | Y==8);             % create binary classification problem between zeros and eights
Y=Y(keep); X=X(keep,:);
```

To visualize what these data look like you can look at the image of data point j with:

```
imagesc( reshape(X(j,:), [28 28])' ); colormap(1-gray(256));
```

As usual, split the data into training and validation.

(a) Build a linear classifier between these two classes. Report what settings you decided on and why, and show your best performance (error rate and J), along with the convergence plot of the error rate and J for these settings. Compare your training and test error rates.

(b) Here, we have 784 features and only about 400 data points, so although our error rates don't indicate overfitting (because the problem is pretty easy), in other cases we might be concerned. An option is to reduce the number of features (and hence the number of parameters). A simple idea in feature reduction is to use random linear projections of the original features; this is implemented in the function `fproject.m`. (You may want to scale the data after projection.) So, for example,

```
[XtP P] = fproject(Xt, nFeat);
[XtP M S] = rescale(XtP);
XvP = rescale( fproject(Xt,nFeat,P), M,S);  % apply to validation data
```

would construct projected and scaled features.

Re-run your predictor after projecting down to a mere two features and compute training and test error. Repeat using 10 features. Comment on your results.