CS273a Homework #2

Introduction to Machine Learning: Winter 2015

Due: Tuesday January 20th, 2015

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

Problem 1: Linear Regression

For this problem we will explore linear regression, the creation of additional features, and cross-validation.

(a) Load the "`data/curve80.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. For consistency in our results, **don't** reorder (shuffle) the data (they're already in a random order), and use the first 75% of the data for training.

(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:.05:10]';                % densely sample possible x-values: note transpose!!!
ys = predict( lr, xs );          % make predictions at xs
```

Plot the training data along with your prediction function in a single plot. Also calculate and report the mean squared error in your predictions on both 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 quadratic feature of Xtr with

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

(You can also add the all-ones constant feature in a similar way, but this is currently done automatically within the learner's train function.) A function "`fpoly`" is also provided to more easily create such features:

```
XtrP = fpoly(Xtr, degree, false); % create poly features up to given degree; no "1" feature
[XtrP, M,S] = rescale(XtrP);      % it's often a good idea to scale the features
lr = linearRegress( XtrP, Ytr );  % create and train model

% now, apply the same polynomial expansion & scaling transformation to Xtest:
XteP = rescale( fpoly(Xte,degree,false), M,S);
```

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, unseen 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 are written to 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 (error values on a log scale, e.g., `semilogy`). For (1),

remember that your learner has now been trained on the polynomially expanded features, and so is expecting `degree` features to be input. So, don't forget to also expand and scale the features of `xs` using `fpoly` and `rescale`. You can do this manually as in the code snippet above, or you can think of this as a "feature transform" function Phi, eg.,

```
Phi = @(x) rescale( fpoly(x,degree,false), M,S);   % defines an "implicit function" Phi(x)
% parameters "degree", "M", and "S" are memorized at the function definition

% Now, Phi will do the required feature expansion and rescaling:
YhatTrain = predict( lr, Phi(Xtr) );    % predict on training data
YhatTest  = predict( lr, Phi(Xte) );    % predict on test data
% etc.
```

Also, you may want to save the original axes of your plot and re-apply them to each subsequent plot for consistency. You can do this by, for example:

```
plot( ... );           % A plot whose scale I like, such as a plot of the data points alone
ax = axis;             % get the axes of the plot: [xmin, xmax, ymin, ymax]
hold on; plot( ... ):  % Something else that may be on a very different scale
axis(ax);              % restore the axis ranges of the 1st plot
```

For (2), plot the resulting training and test errors as a function of polynomial degree.

Problem 2: Cross-validation

In the previous problem, you decided what degree of polynomial fit to use based on performance on some test data. (Technically, since you knew the answers to these data's targets and could use them to evaluate performance at different degrees, I would probably call them validation data instead.) Let's now imagine that you did not have access to the target values of the test data you held out in the previous problem, and wanted to decide on the best polynomial degree.

Of course, we could simply repeat the exercise, further splitting `Xtr` into a training and validation split, and then assessing performance on the validation data to decide on a degree. But it is often more effective to use cross-validation to estimate the optimal degree.

Cross-validation works by creating many such training/validation splits, called folds, and using all of these splits to assess the "out-of-sample" (validation) performance by averaging them. You can do a 5-fold validation test, for example, by:

```
nFolds = 5;
for iFold = 1:nFolds,
  [Xti,Xvi,Yti,Yvi] = crossValidate(Xtr,Ytr,nFolds,iFold);  % take ith data block as validation
  learner = linearRegress(...   % TODO: train on Xti, Yti , the data for this fold
  J(iFold) = ...                % TODO: now compute the MSE on Xvi, Yvi and save it
end;
% the overall estimated validation performance is the average of the performance on each fold
mean(J)
```

Using this technique on your training data `Xtr` from the previous problem, find the 5-fold cross-validation MSE of linear regression at the same degrees as before, $d = 1, 3, 5, 7, 10, 18$. Plot the cross-validation error (with semilogy, as before) as a function of degree. Which degree has the minimum cross-validation error? How does its MSE estimated from cross-validation compare to its MSE evaluated on the actual test data?

Problem 3: Kaggle

TBD...