CS178 Homework #2

Machine Learning & Data Mining: Winter 2017

Due: Friday January 27th, 2017

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[:,0]` is the scalar feature (x) values; the second column `data[:,1]` 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 and the rest for testing:

```
X = data[:,0]
X = X[:,np.newaxis]    # code expects shape (M,N) so make sure it's 2-dimensional
Y = data[:,1]          # doesn't matter for Y
Xtr,Xte,Ytr,Yte = ml.splitData(X,Y,0.75)  # split data set 75/25
```

(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 = ml.linear.linearRegress( Xtr, Ytr );  # create and train model
xs = np.linspace(0,10,200);         # densely sample possible x-values
xs = xs[:,np.newaxis]               # force "xs" to be an Mx1 matrix (expected by our code)
ys = lr.predict( xs );              # make predictions at xs
```

Plot the training data along with your prediction function in a single plot. Print the linear regression coefficients (`lr.theta`) and check that they match your plot. Finally, 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 = np.zeros( (Xtr.shape[0],2) )  # create Mx2 array to store features
Xtr2[:,0] = Xtr[:,0]        # place original "x" feature as X1
Xtr2[:,1] = Xtr[:,0]**2     # place "x^2" feature as X2
# Now, Xtr2 has two features about each data point: "x" and "x^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 "`ml.transforms.fpoly`" is also provided to more easily create such features. Note, though, that the resulting features may include extremely large values – if $x \approx 10$, then e.g., x^{10} is extremely large. For this reason (as is often the case with features on very different scales) it's a good idea to rescale the features; again, you can do this manually yourself or use a provided `rescale` function:

```
# Create polynomial features up to "degree"; don't create constant feature
#   (the linear regression learner will add the constant feature automatically)
XtrP = ml.transforms.fpoly(Xtr, degree, bias=False);

# Rescale the data matrix so that the features have similar ranges / variance
XtrP,params = ml.transforms.rescale(XtrP);
#  "params" returns the transformation parameters (shift & scale)

# Then we can train the model on the scaled feature matrix:
lr = ml.linear.linearRegress( XtrP, Ytr );    # create and train model

# Now, apply the same polynomial expansion & scaling transformation to Xtest:
XteP,_ = ml.transforms.rescale( ml.transforms.fpoly(Xte,degree,false), params);
```

This snippet also shows a useful feature transformation framework – often we wish to apply some transformation to the features; in many cases the desired transformation depends on the data (such as rescaling the data to unit variance). 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, `params = (mu,sig)`, a tuple containing the mean and standard deviation used to shift and scale the data), 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 (plot the 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 (columns) 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.,

```
# Define a function "Phi(X)" which outputs the expanded and scaled feature matrix:
Phi = lambda X: ml.transforms.rescale( ml.transforms.fpoly(X, degree,False), params)[0]
# the parameters "degree" and "params" are memorized at the function definition

# Now, Phi will do the required feature expansion and rescaling:
YhatTrain = lr.predict( Phi(Xtr) );    # predict on training data
YhatTest  = lr.predict( 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. (Otherwise, high-degree polynomials may look "flat" due to some extremely large values.) You can do this by, for example:

```
plt.plot( ... );  # A plot whose scale I like, such as a plot of the data points alone
ax = plt.axis();  #  get the axes of the plot: [xmin, xmax, ymin, ymax]
plt.plot( ... );  # Something else that may be on a very different scale
plt.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 when training is reasonably fast, it can be 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 in range(nFolds):
  Xti,Xvi,Yti,Yvi = ml.crossValidate(Xtr,Ytr,nFolds,iFold);  # take ith data block as validation
  learner = ml.linear.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
print np.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, $d = 1, 3, 5, 7, 10, 18$ (or more densely, if you prefer). 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?