CS5793: Artificial Intelligence II
Assignment 3

Prof. Christopher Crick

1 Linear regression: polynomial basis functions

Obtain the file "`crash.txt`" from D2L. This file contains measurements from a crash test dummy during an NHSA automobile crash test. Column 0 measures time in milliseconds after the crash event and column 1 represents the acceleration measured by a sensor on the dummy's head. The file is formatted according to Numpy's default specification for textual data (spaces delineate columns, newlines delineate rows). Thus it can easily be read into a matrix via the following:

```
data = np.loadtxt('crash.txt')
```

Divide this data into a training and test set of equal size – even-numbered rows to the training set and odd-numbered rows to test. Fit sets of L polynomial basis functions to the training set, determining their linear weight vector $\hat{w}$ using the normal equation $(\Phi^T\Phi)\hat{w} = \Phi^T t$. t is an N-dimensional vector of training responses t_n, while Φ is an $N \times M$ matrix of basis function outputs: $\Phi_{ij} = \phi_j(x_i), \phi_j(x) = x^j$. Try $L = 1, \ldots, 20$. For each value of L, compute the maximum likelihood RMS error between the actual data and the model's prediction, on both the training and test sets. Plot these errors.

Hint: Use the normal equation as shown above and solve it using `np.linalg.solve`, rather than rewriting it and using `np.linalg.inv`. Inverting a matrix can lead to numerical instability, so it should be avoided when possible.

Plot the training data and the output of the model function with the lowest RMS on the training data. To do this, use `np.linspace` to generate values on the time axis and the model function to compute acceleration axis values. Do the same for the best fit on the test set.

2 Linear regression: radial basis functions

Now we turn to radial basis functions. Perform the same procedure as with the polynomial fit, except use radial basis functions $\phi_j(x) = \exp\left\{-\frac{(x-\mu_j)^2}{2\sigma^2}\right\}, j = 1, \ldots, L$. The function centers μ_j should be evenly spaced across the range of x (about 0-60), and the standard deviation σ of the functions should be the distance between one basis function mean and the next. Investigate $L = 5, 10, 15, 20, 25$, computing and plotting RMS on training and test data, and plotting the best-fit model for the training and test sets, just as in the last problem.

3 Bayesian MAP estimates

ML estimates become very unstable as the order of the model grows. This can be addressed by adopting a Bayesian prior, which is expressed by the parameters β (an estimate of the inverse variance of the noise in the system) and α (an estimate of the inverse variance between the actual generative function in the real world and our choice of model). By eyeballing the noise in the data plot, it looks like $\sigma = 20$ is a reasonable estimate of its standard deviation. This corresponds to $\beta = \frac{1}{\sigma^2} = 0.0025$. Using this as an estimate of β, consider 100 candidate values for α, spaced logarithmically between 10^{-8} and 10^0 (`alpha = np.logspace(-8,0,100)`). Using $(\Phi^T\Phi + \frac{\alpha}{\beta}I_M)\hat{w} = \Phi^T t$, applied to the training set and using $L = 50$ radial basis functions, find the value of α with the best performance on the test set and plot the output of this model function, along with the data.

4 Logistic regression

Download the Iris dataset from `http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data` . The Numpy `loadtxt` function assumes that everything is a float, but here the category is given as a string. Here is one way to load it, converting the strings to numbers:

```
def flower_to_float(s):
    d = {'Iris-setosa':0.,'Iris-versicolor':1.,'Iris-virginica':2.}
    return d[s]

irises = np.loadtxt('iris.data',delimiter=',',converters={4:flower_to_float})
```

You aren't done massaging the data, though. Separate the matrix into separate data and label arrays. Then turn the label array from an $N \times 1$ array with different numbers delineating different classes into an $N \times K$ membership array, where each class corresponds to having a one in the appropriate column, and zero everywhere else – a structure often called a "one-hot" array. Furthermore, add a column of ones to the data – this will allow you to have a bias term $\phi_0(x_n)$. Then divide the data and labels in half to make a training and test set (make sure there are proportional numbers of each class in each set).

Your job will be to implement a function $f(w)$ which you can pass to `scipy.optimize.minimize` for gradient descent, as well as an initial w vector (which can be an array of ones, since we don't have any better idea of where to start). We are going to use linear transformations of our data (no nonlinear basis functions) for simplicity, which means that $\phi(x_i) = [1, x_{i1}, x_{i2}, x_{i3}, x_{i4}]$, four data dimensions and a bias term. The maximum likelihood probability of class k is:

$$p(t_{nk} = 1|x_n, w) = \frac{\exp(w_k^T\phi(x_n))}{\sum_{l=1}^K \exp(w_l^T\phi(x_n))} \tag{1}$$

$w_k \in \mathbb{R}^M$ are the weights associated with class k, and $w \in \mathbb{R}^{KM}$ is a single-dimensional vector with the weights for all classes. Why represent it this way? The input w to the function $f(w)$ is assumed by the `minimize` function to be a single vector, so we have to play along. That means you will have to massage w in your code. w contains 15 elements, one for each data column (including the bias vector) for each class. In the equation above, w_k and w_l are only 5 elements long, so depending on the class, you are using $w[0:5]$ (class 1), $w[5:10]$ (class 2), etc.

We also need a Gaussian prior in order to deal with potential degenerate behavior.

$$p(w) = \mathcal{N}(w|0, \alpha^{-1} I_{KM}) \propto \exp\left(-\frac{\alpha}{2} w^T w\right) \tag{2}$$

You may pick any α that you find reasonable. The best one you found in the previous problem wouldn't be a terrible choice; it might be off by an order of magnitude or two, but not more. And with Bayesian priors, that is often close enough. With this in mind, the objective function whose minimum is the MAP estimate of the weight vector w becomes

$$f(w) = -\log p(w) - \sum_{n=1}^{N} \log p(t_n|x_n, w) = \frac{\alpha}{2} w^T w - \sum_{n=1}^{N} \left[\sum_{k=1}^{K} t_{nk} w_k^T \phi(x_n) - \log \sum_{l=1}^{K} \exp(w_l^T \phi(x_n)) \right] \tag{3}$$

Implement this function and pass it in as an argument to `minimize`. The object returned by `minimize` has a number of attributes; the one you want is named `x`, but it is your best-fit posterior estimate of the vector w.

```
w_init = np.ones(15)
w_hat = minimize(f, w_init).x
```

Now, you can compute the class posterior probabilities of your test set using the softmax function.

$$s_k(z) = \frac{\exp(z_k)}{\sum_{l=1}^{K} \exp(z_l)} \tag{4}$$

where $z_k = w_k^T \phi(x_n)$, and x_n is a member of your test set. Classify the test set point according to the highest probability produced by softmax, and report the overall classification accuracy.