CPSC-540 **Machine Learning** **2013**

Homework # 3

NAME:____

Signature:____

STD. NUM: ____

1 Random forests

In this question you will implement the training procedure for random forests. There are a few ways to go about this, but the one you will be using is more or less the original algorithm from Leo Breiman.

In Breiman's algorithm, each tree in the forest is trained using a bootstrapped sample of the training data. Bootstrapping means that each tree is trained on a slightly different version of the data, which helps ensure the trees are uncorrelated. This is explained in the pseudo-code in the lectures and also in great detail in the random forests chapter of the book by Hastie, Friedman and Tibshirani, which is freely available online (the link to it is on the main course webpage).

Each tree is built recursively, starting from the root. At each node a randomized search procedure is applied to determine how the node should be split. When the best split is found, the data in the node are partitioned according to the chosen split, and the procedure is applied recursively to split the left and right children. Splitting continues until no acceptable split can be found or a maximum depth is reached.

The split procedure works by choosing `n_trials` dimensions at random to search along (we will stick to axis aligned splits to keep things simple). In each of these dimensions, it chooses a set of thresholds and computes the information gain from splitting along each dimension at each threshold. For each dimension, it should try enough thresholds to ensure that every possible partition is checked (choosing the thresholds to be halfway between pairs of data points sorted along the current dimension is a good way to ensure this).

The split procedure chooses the (dimension, threshold) pair which gives the largest information gain over all the candidates. Additionally, the split procedure must respect the parameter `n_min_leaf` by refusing to create a split where either of the children contains fewer than `n_min_leaf` data points.

Your task is to implement the split procedure described here using the skeleton code provided at:

```
http://cs.ubc.ca/~mdenil/hidden/simpleforests/simpleforests.zip
```

The provided code handles building each tree using a bootstrapped sample of the training data and also handles making predictions once the forest has been built. What is missing is the body of the `_find_split_parameters` function in `rf/builder.py`. This function is called when splitting each node to determine which (dimension, threshold) should be used to split the data. You must implement this function.

There are several tests included in the root directory titled `test_*.py`. You can run them individually by running the file directly, e.g.

```
# python test_forestfire.py
```

or you can run

```
# nosetests
```

to run them all. If you have implemented the split procedure correctly then all of these tests will pass.

Running the synthetic examples (`test_synthetic_classification.py` and `test_synthetic_regression.py`) will generate images visualizing the predictions of the forest as well as the predictions of each tree. Example images which show the intended output from these tests are also included with the code for comparison.

You should submit a printout of rf/builder.py with the code you wrote. Also submit a printout of the result of running

```
# nosetests -vs
```

in the project directory. This will verify all the unit tests pass with the correct accuracy.

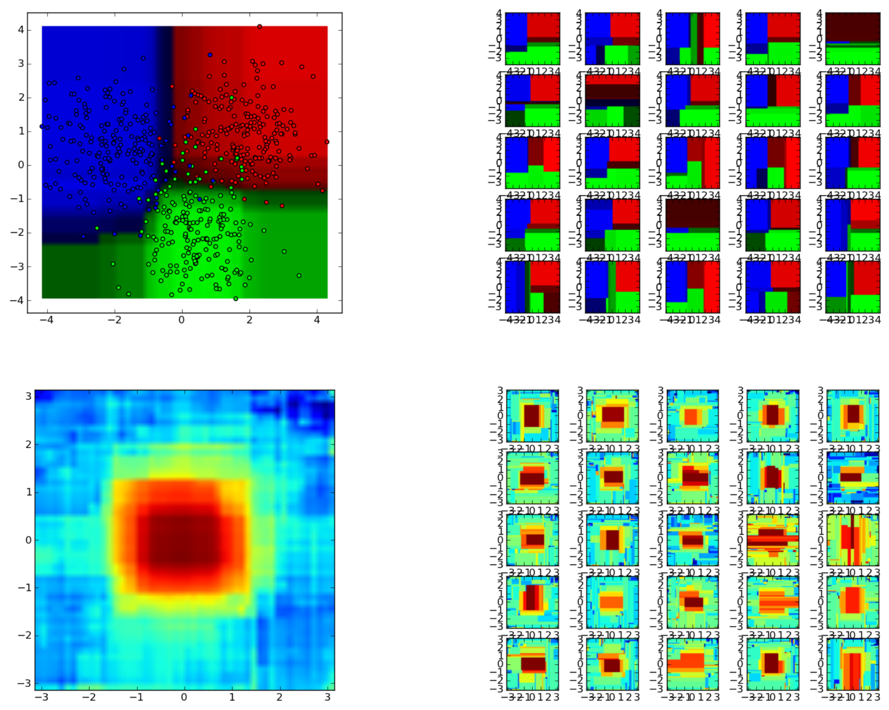

Figure 1: **Top row:** Expected output from `synthetic_classification_example.py`. The forest prediction is shown on the left, and the individual tree predictions on the right. **Bottom row:** Expected output from `synthetic_regression_example.py`. The forest prediction is shown on the left, and the individual tree predictions on the right.

2 Bayesian optimization with Gaussian processes

In this question you will implement a Bayesian optimization procedure using Gaussian processes (GPs). In order to do this we will provide you with starter code that you can download from

`http://cs.ubc.ca/~hoffmann/bayesopt.zip`

After unzipping, you will find the file `bayesopt.py`, which is where all your modifications for this assignment will go.

2.1 Structure of the provided code

In the given code we provide a Gaussian process class `GP` such that (once the modifications detailed in the next subsection are done!) one can perform the following:

```
gp = GP(kernel, sigma)
gp.add_data(xtrain, ytrain)
mu, s2 = gp.posterior(xtest)
```

The first line above creates the GP prior, given by a kernel function (which we'll get to momentarily) and the standard deviation of the noise. The second line adds training data to the GP, which must be in the form of an (n, d)-array `xtrain` and an n-vector `ytrain`. After adding this data we can then obtain the posterior mean and variance for a collection of test points `xtest` which must also be an (n, d)-array (note that this n can be different from the n used for training).

The kernel function must be a function which takes two arrays `x1` and `x2` of size (n, d) and (m, d) and returns an (n, m)-array of all the pairwise kernel evaluations. We also allow for the kernel to take optional hyperparameters, see for example the definition of the squared-exponential kernel

```
def sqexp_kernel(x1, x2, ell=1.0, sf2=1.0):
    ...
```

where `ell` and `sf2` control the behavior of this kernel. When initializing the Gaussian process, any additional named parameters will be passed to the kernel. For example to use a squared-exponential kernel with $\ell = 2.0$ we could initialize a `GP` as

```
gp = GP(sqexp_kernel, sigma, ell=2.0)
```

We also provide a function to perform Bayesian optimization with GPs, `gpopt`, over a discrete set of points. Given a noisy function `f` we can optimize this function as follows:

```
candidates = ... # some set of points we want to optimize at
gp = GP(sqexp_kernel, sigma=sigma)
xopt = gpopt(f, gp, acq, candidates)
```

The returned value `xopt` will be a sequence of points the algorithm "thinks are optimal" at every iteration, i.e. `xopt[-1]` is the optimum found by the algorithm. The acquisition function `acq` is at the heart of Bayesian optimization and should be something of the form

```
def acq(gp, candidates, ...):
    ...
```

This function should take a `GP` instance and a set of points `candidates` and return a vector or *index* such that the highest index value corresponds to the next point that should be selected. See the expected improvement, `gpei`, acquisition function for an example. Note also, that like the kernel functions defined above any additional named parameters passed to `gpopt` will be passed to the acquisition function. I.e. to use expected improvement, but with $\xi = 0.8$ we can run

```
gpopt(f, gp, gpei, candidates, xi=0.8)
```

2.2 Implementation

You will have to modify this file in all the locations marked with a comment of the form

```
# IMPLEMENT ME: ...
```

In particular, your task will consist of the: (1) implement the squared exponential kernel; (2) implement the posterior computations for Gaussian processes; (3) implement the GP-UCB acquisition function. We will detail each of these tasks below.

The code for implementing the Gaussian process is all contained within the first "block" of code, at the top of the file. The first step is to implement a squared-exponential kernel function,

$$k(x, x') = \sigma_f^2 \exp(-0.5\|x - x'\|_2^2/\ell).$$

We have provided starter code of the form

```
def sqexp_kernel(x1, x2, ell=1.0, sf2=1.0):
    d = dist.cdist(x1, x2, 'sqeuclidean')
    k = ...
    return k
```

We have also provided a call to the `cdist` function which should return an $m \times d$ matrix of the pairwise squared distances between `x1` and `x2`. You must then use this to implement the rest of the kernel.

Next, you must implement the posterior updates for the Gaussian process itself. Later on in the code you will find a section of code inside the `posterior(...)` function that looks like

```
s2 -= np.diag(Kk.T.dot(k))
mu = ...
```

Here, `mu` and `s2` are the mean and variance terms for each point passed in. We have provided code to compute the variance, and you must compute the mean. Note also that you can make use of `self.X`, `self.y`, and `self.K` to compute this term, where respectively these represent the set of training inputs, training outputs, and the kernel matrix.

At this point the GP code should be fully implemented, and if you run the code (from ipython you can just type `%run bayesopt.py`) it should produce the plot shown in Figure 2, which is fitting a noisy sin function. The code that implements this demo is `demo_gp()` and is called from the very bottom of the file inside the `__main__` section.

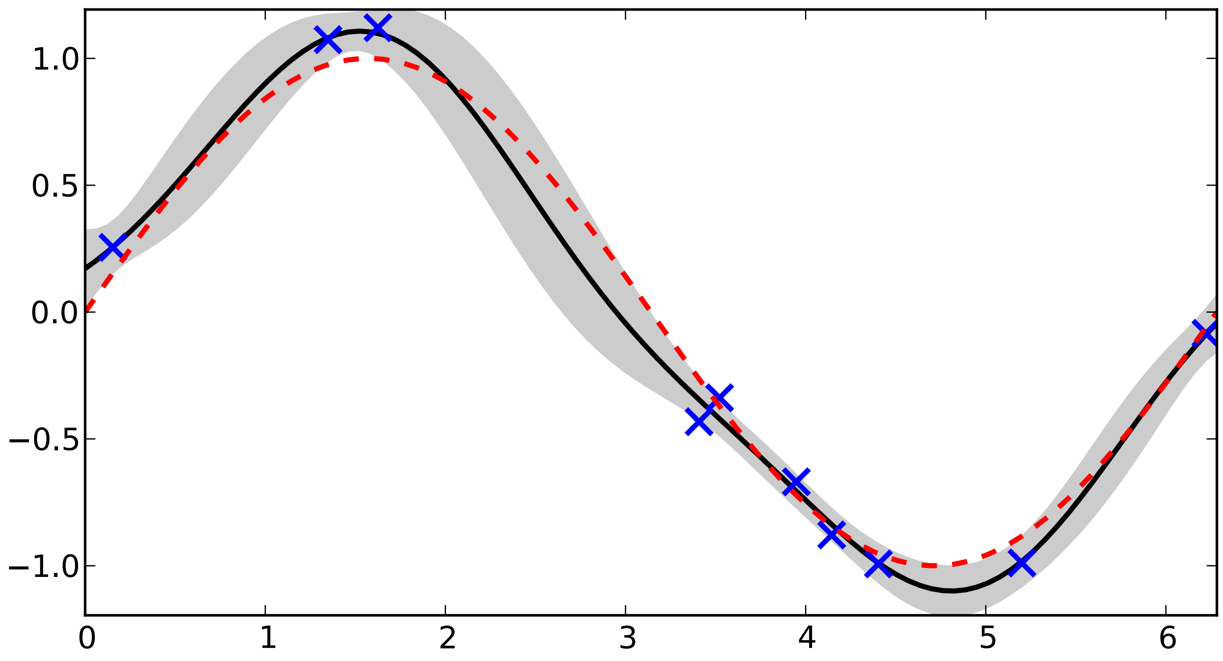

Figure 2: Using a Gaussian process to fit a noisy sin function.

Finally, the second block of code in this file implements Bayesian optimization. We have provided for you a function `gpei(...)` which computes the *expected improvement* acquisition function. Your task is to implement the *upper confidence bound* (UCB) acquisition function as discussed in class. If we let $\mu(x)$ and $\sigma^2(x)$ denote the posterior mean and variance evaluated at x then at iteration t we can write the UCB acquisition function as

$$\text{ucb}_t(x) = \mu(x) + \xi \log(t)\sigma(x).$$

Your task is to implement this function, which can be found with the following code skeleton:

```
def gpucb(gp, x, xi=0.1):
    ...
```

Once this has been implemented the `demo_gp()` function call at the end of the file can be replaced with `demo_bo1()`, which utilizes Bayesian optimization to find the maximum of the same noisy sin function. Running this code just as before should result in a plot similar to that of Figure 3.

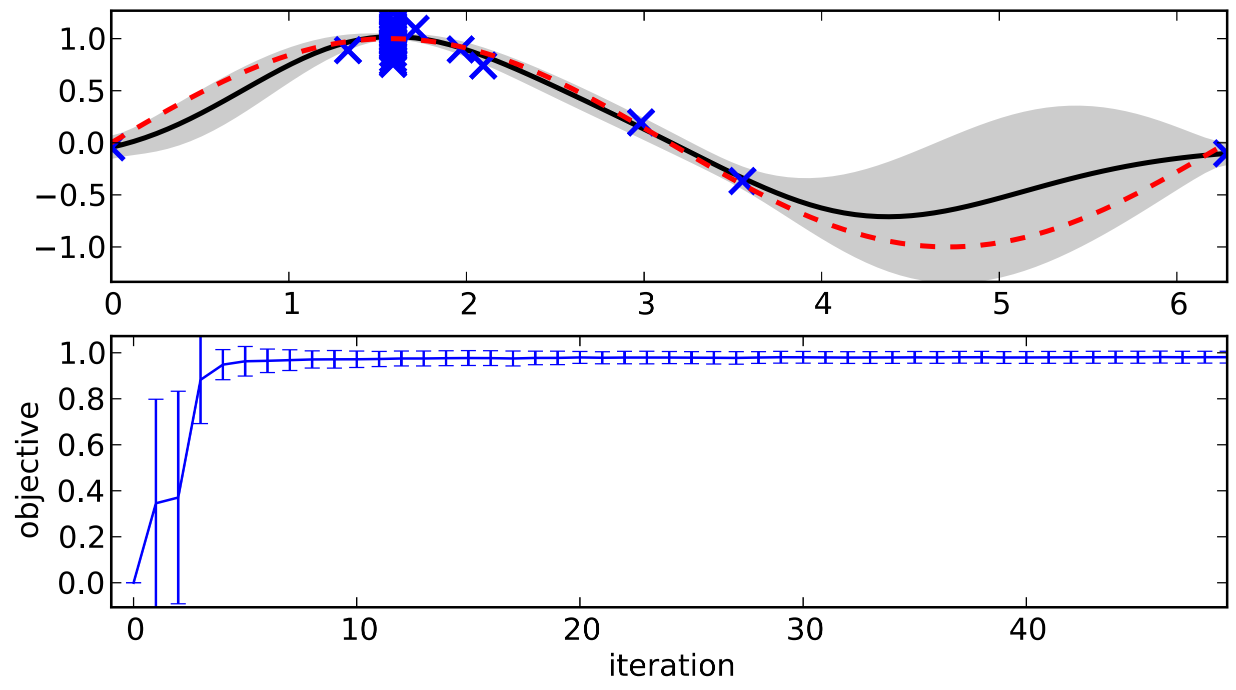

Figure 3: Using a Bayesian optimization to optimize the noisy sin function. The top plot shows an example GP at the end of the optimization process, and the bottom plot shows the result of the optimization averaged over 30 runs.

Turn in your code along with your reproduction of the two plots produced above.

3 Gradient and Hessian for logistic regression

Derive the expressions for the gradient and Hessian of logistic regression presented in class.

Loss function $J(\theta) = -\sum_{i=1}^{n} [\, y_i \log \pi_i + (1-y_i) \log(1-\pi_i)]$

where $\pi_i = \frac{1}{1+e^{-x_i^T\theta}}$ $\pi_i \in R,\ y_i \in R,\ x_i \in R^{n\times d},\ \theta \in R^d$

$n = $ # training data $d = $ # features.

To compute gradient and Hessian of $J(\theta)$, we compute gradient of $\log \pi_i$, $\log(1-\pi_i)$, π_i first.

$$\log \pi_i{}'_{\theta} = -\log(1+e^{-x_i^T\theta})'_{\theta} = -\frac{1}{1+e^{-x_i^T\theta}} \cdot e^{-x_i^T\theta} \cdot (-x_i) = (1-\pi_i)x_i$$

$$\log(1-\pi_i)'_\theta = \left(\log \frac{e^{-x_i^T\theta}}{1+e^{-x_i^T\theta}}\right)'_\theta = \left(-x_i^T\theta\right)'_\theta + \left(\log \pi_i\right)'_\theta = -x_i + (1-\pi_i)x_i$$

$$= -\pi_i x_i$$

$$(\pi_i)'_\theta = \pi_i(1-\pi_i)\cdot x_i$$

Therefore

$$\nabla_\theta J(\theta) = -\sum_{i=1}^n \left[y_i(1-\pi_i)x_i + (1-y_i)(-\pi_i x_i)\right] = \sum_{i=1}^n (\pi_i - y_i)x_i = \underbrace{X^T(\pi - y)}_{\in R^d}$$

$$H_\theta = \nabla_\theta^2 J(\theta) = \left(\sum_{i=1}^n (\pi_i - y_i)x_i\right)'_\theta = \underbrace{\sum_{i=1}^n \pi_i(1-\pi_i)\cdot x_i\cdot x_i^T}_{\in R^{d\times d}}$$