sheet07

December 2, 2017

1 Model Selection

In this programming assignment we examine techniques for model selection on classification and regression tasks. In particular, we first explore the effect of model hyperparameters on the bias and variance of the prediction. In the second part of the assignment we utilize the bias-variance decomposition to perform automatic hyperparameter selection. Several classes and methods are provided in the `utils.py` file:

1.0.1 Datasets

- `utils.Housing()`: This regression dataset is available at http://archive.ics.uci.edu/ml/datasets/Housing and loaded from scikit-learn's inbuilt representation. This data is used for regression. A description of the dataset can be found here http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names. This data is in a `506x13` matrix and the labels in a array of length `506`.
- `utils.Yeast()`: This classification dataset is available at https://archive.ics.uci.edu/ml/datasets/Yeast . This data is used for classification. A description of the dataset can be found here https://archive.ics.uci.edu/ml/machine-learning-databases/yeast/yeast.names. This data is in a `1484x8` matrix and the labels (class probabilities) are in a `1484x7` matrix where `targets[i,j] = 1` if example `i` is of class `j` and `0` otherwise. For example, if we have a dataset of 4 examples which belong to following classes : `[1, 0, 0, 2]` the label matrix would look like this: `T = [[0, 1, 0], [1, 0, 0], [1, 0, 0], [0, 0, 1]]`.

1.0.2 Predictors

We provide two simple classes of predictors, one for regression and one for classification:

- `utils.ParzenRegression`: A regression method based on Parzen window. The hyperparameter corresponds to the scale of the Parzen window. A large scale creates a more rigid model. A small scale creates a more flexible one.
- `utils.ParzenClassification`: A classification method based on Parzen window. The hyperparameter corresponds to the scale of the Parzen window. A large scale creates a more rigid model. A small scale creates a more flexible one. Note that instead of returning a single class for a given data point, it outputs a probability distribution over the set of possible classes.

Each class of predictor implements the following three methods:

- `__init__(self,parameter)`: Create an instance of the predictor with a certain scale parameter.
- `fit(self,X,T)`: Fit the predictor to the data (a set of data points `X` and targets `T`).
- `predict(self,X)`: Compute the output values arbitrary inputs `X`.

1.0.3 Bias Variance Decomposition

As we have seen in the theoretical exercise, there are several possible bias-variance decomposition for different tasks (e.g. classification, or regression).

- `utils.biasVarianceRegression()`: Perform the usual bias-variance decomposition of the mean square error. Reminder: given Y the (random) estimator and T the target, the decomposition is computed as follows:
- $\text{Bias}(Y)^2 = (\mathbb{E}_Y[Y - T])^2$
- $\text{Var}(Y) = \mathbb{E}_Y[Y - \mathbb{E}_Y[Y])^2]$
- $\text{Error}(Y) = \mathbb{E}_Y[(Y - T)^2]$

1.0.4 Sampler

To compute the bias and variance estimates, we require *multiple samples* from the training set for a single set of observation data. To acomplish this, we utilize the `Sampler` class provided. The sampler is initialized with the training data and passed to the method for estimating bias and variance, where its function `sampler.sample()` is called repeatedly in order to fit multiple models and create an ensemble of prediction for each test data point.

1.1 Part 1: Implementing Bias-Variance Decomposition for Classification (20 P)

Implement a function which computes the bias, variance and error given the true labels of the training data and the predicted values. Bias, Variance and Error for classification are defined as:

- $\text{Bias}(Y) = D_{\text{KL}}(T||R)$
- $\text{Var}(Y) = \mathbb{E}_Y[D_{\text{KL}}(R||Y)]$
- $\text{Error}(Y) = \mathbb{E}_Y[D_{\text{KL}}(T||Y)]$

where R is the distribution that minimizes its expected KL divergence from the estimator of probability distribution Y (see the theoretical exercise for how it is computed exactly), and where T is the target class distribution. Note that we consider here the Kullback-Leibler divergence as a measure of classification error, which is commonly done in practice in order to have a smooth objective function.

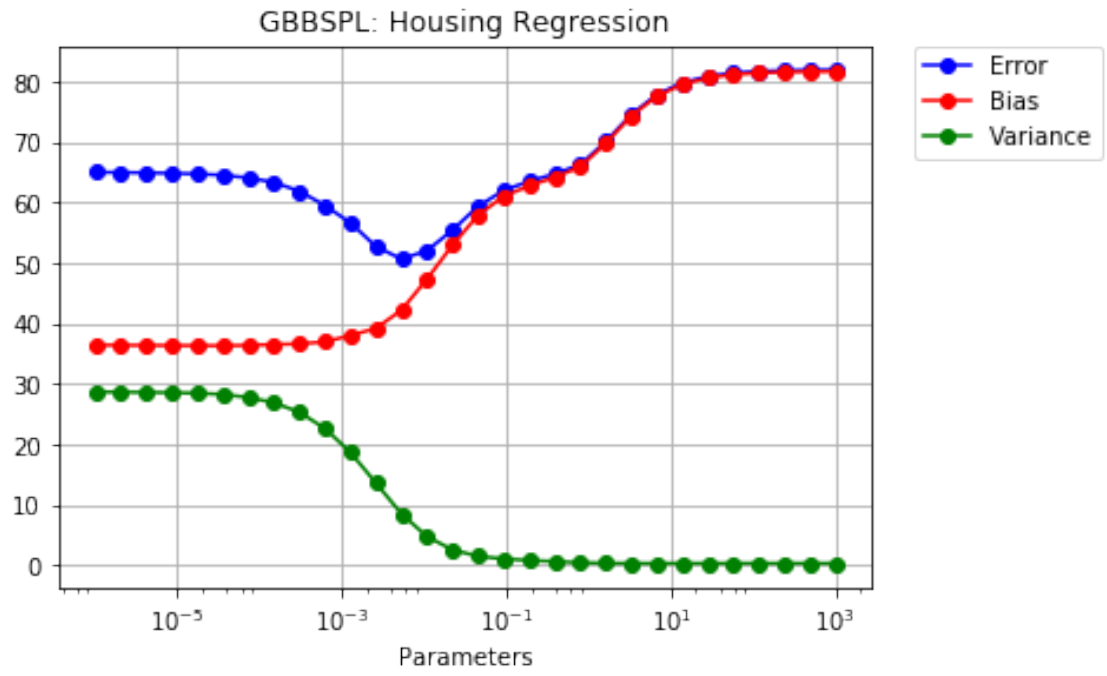
Tasks:

- **Implement the KL-based Bias-Variance Decomposition defined above (10 P)**

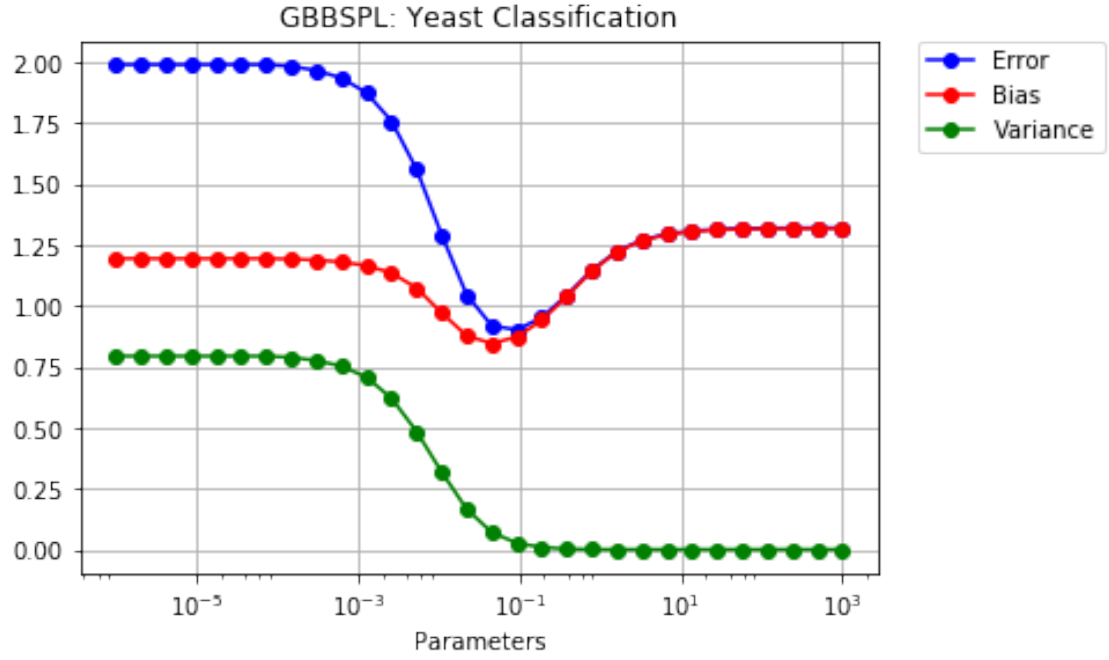
To get started, you can take inspiration from the readily implemented function `utils.biasVarianceRegression()`, which does the following:

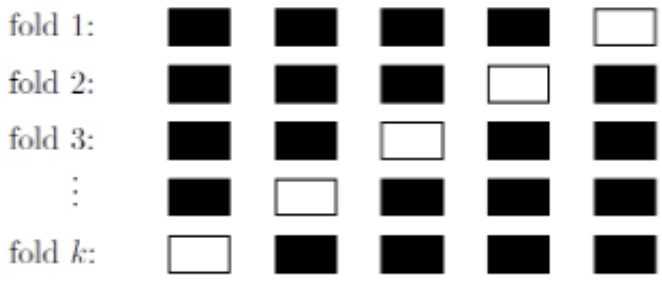
- Iterate for a certain number of times the following:
  - Acquire a subsample of the training data by invoking `sampler.sample()`
  - Using the predictor (which will either be a Parzen Regressor or Parzen Classifier depending on the task), fit the model on the sample and determine the prediction for the observation data (N examples disjoint from the training data). Note that the dimension of the outputs matches the dimension of the targets, so for regression you will get an array of length N and for classification a matrix of shape $N \times$ #classes containing the class distributions.
- Having computed a number of different predictions, determine the bias, variance and error comparing the predictions to the true labels. Check that the decomposition is correct (i.e. bias + variance = error) using an `assert` statement, and return the bias and variance.
- **Once the method is implemented, run Test 1 and Test 2 provided below (10 P)**

```
In [1]: def biasVarianceClassification(sampler, predictor, X, T, nbsamples=25):
            Y = numpy.array([predictor.fit(*sampler.sample()).predict(X) for _ in range(nbsample

            Y_log_mean = numpy.mean(numpy.log(Y), axis=0)

            R_dominator = numpy.sum(numpy.exp(Y_log_mean), axis=1).reshape(Y.shape[1],1)
            R = numpy.exp(Y_log_mean)/R_dominator

            bias     = numpy.mean(numpy.sum(numpy.log(T/R) * T, axis=1))
            variance = numpy.mean(numpy.mean(-numpy.sum(numpy.log(Y/R) * R, axis=2), axis=1))
            error = numpy.mean(-numpy.sum(numpy.log(Y/T) * T, axis=2))

        #     print bias, variance, error

            assert(numpy.abs((bias + variance) / error - 1) < 1e-4)

            return bias, variance

In [2]: ### TEST 1
        import utils,numpy
        %matplotlib inline
        utils.plotBVE(utils.Housing,numpy.logspace(-6,3,num=30),utils.ParzenRegressor,utils.bias
```

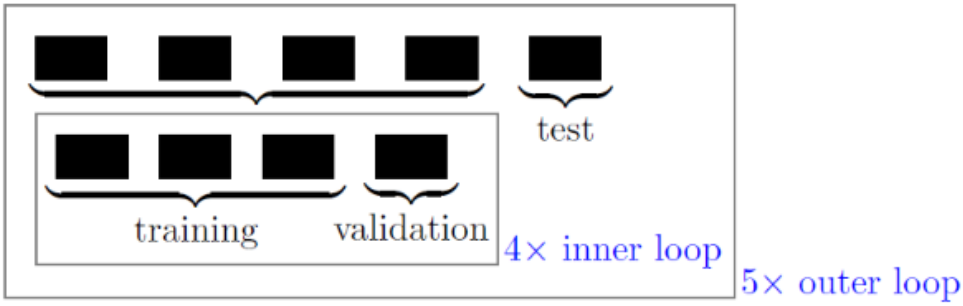


In [3]:
```
### TEST 2
import utils,numpy
%matplotlib inline
utils.plotBVE(utils.Yeast,numpy.logspace(-6,3,num=30),utils.ParzenClassifier,biasVarianc
```





Loop



Nested

1.2 Part 2: Implementing a Parameter Selection Procedure (30 P)

In this part of the exercise, we would like to find what is the best hyperparameter of the model for predicting the Housing regression data. A 5-fold cross-validation procedure is already implemented and that allows to compute error bars.

You need to extend this basic cross-validation procedure by a nested loop of 4-fold cross-validation that selects the best hyperparameters based on some criterion (cost function) to be determined. The nested loop is depicted below:

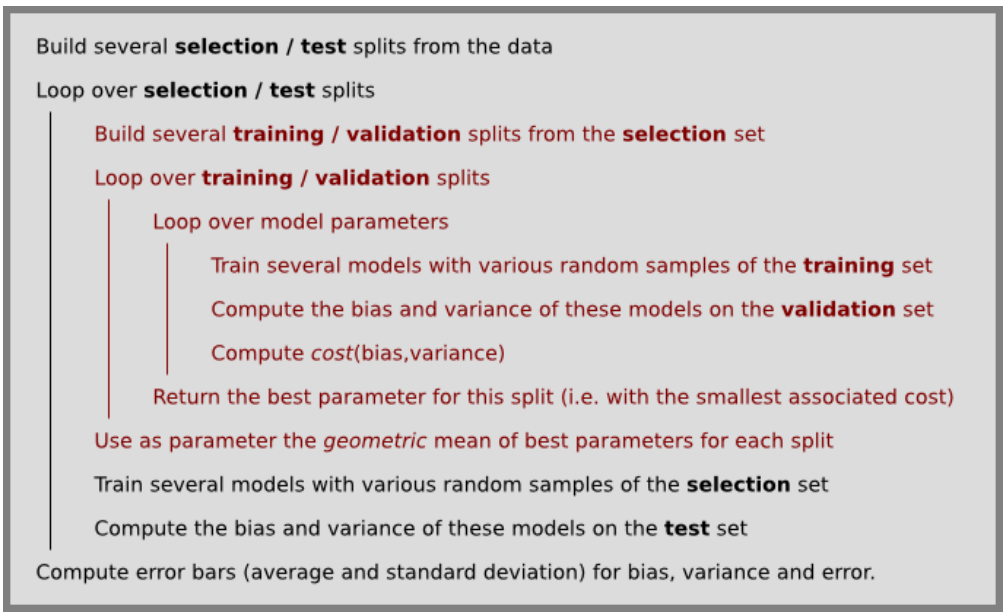
The full procedure for evaluation and hyperparameter selection procedure is shown in the diagram below with the part that you need to implement highlighted in red.

Tasks:

- **Implement the inner loop of 4-fold cross-validation, helping you from the diagram above (20 P)**

For this part, use the following settings:

- Range of parameters to test: 15 parameters logarithmically spaced between 1e-5 and 1e5.
- The returned parameter is the geometric mean of the best parameter found for each split.



Procedure

- The best parameter for each split is the one that minimizes the `costfunction` specified as argument.
- The bias and variance estimates are obtained by sampling 10 times from the training distribution.
- **Verify your implementation by running Test 3 (10 P)**

```
In [4]: import numpy,utils
        from scipy.stats import mstats

        def getbestparameter(Xselect,Tselect,costfunction):

            params = numpy.logspace(-5,5,num=15)

            # Create splits
            splits = [ ([1,2,3],0) , ([0,2,3],1) , ([0,1,3],2) , ([0,1,2],3)]

            best_params = []

            #Loop over train/validation splits
            for inds_train,ind_validation in splits:

                Xtrain = [Xselect[ind] for ind in inds_train]
                Ttrain = [Tselect[ind] for ind in inds_train]

                Xvalidation = Xselect[ind_validation]
                Tvalidation = Tselect[ind_validation]
```

```
            smallest_err_in_cur_split = 999999
            best_param_in_cur_split = 999999

            for param in params:
                # Evaluate bias and variance with this best parameter
                predictor = utils.ParzenRegressor(param)
                sampler   = utils.Sampler(numpy.concatenate(Xtrain,axis=0),numpy.concatenate
                bias,variance = utils.biasVarianceRegression(sampler, predictor, Xvalidation
                if(costfunction(bias, variance) < smallest_err_in_cur_split):
                    best_param_in_cur_split = param
                    smallest_err_in_cur_split = costfunction(bias, variance)
            best_params += [best_param_in_cur_split]
        # geometric mean
        return mstats.gmean(best_params)
```

In [5]:
```
import numpy,utils

def evaluateModel(X,T,costfunction):
    # X: partitioned input
    # T: partitioned targets
    # costfunction: the function for evaluate how good/bad a hyperparameter is

    # Create splits
    splits = [ ([1,2,3,4],0) , ([0,2,3,4],1) , ([0,1,3,4],2) , ([0,1,2,4],3) , ([0,1,2,3

    testbiases,testvariances,testerrors,bestparameters = [],[],[],[]

    #Loop over selection/test splits
    for inds_select,ind_test in splits:

        Xselect = [X[ind] for ind in inds_select]
        Tselect = [T[ind] for ind in inds_select]

        Xtest = X[ind_test]
        Ttest = T[ind_test]

        bestparam = getbestparameter(Xselect,Tselect,costfunction)

        # Evaluate bias and variance with this best parameter
        predictor = utils.ParzenRegressor(bestparam)
        sampler   = utils.Sampler(numpy.concatenate(Xselect,axis=0),numpy.concatenate(Ts
        bias,variance = utils.biasVarianceRegression(sampler,predictor,Xtest,Ttest, nbsa

        testbiases     += [bias]
        testvariances  += [variance]
        testerrors     += [bias+variance]
        bestparameters += [bestparam]
```

```
            # Output results of model evaluation
            print('bias:      %8.5f +/- %8.5f'%(numpy.mean(testbiases),numpy.std(testbiases)))
            print('variance:  %8.5f +/- %8.5f'%(numpy.mean(testvariances),numpy.std(testvariance
            print('error:     %8.5f +/- %8.5f'%(numpy.mean(testerrors),numpy.std(testerrors)))
            print('parameter: %8.5f +/- %8.5f'%(numpy.mean(bestparameters),numpy.std(bestparamet
```

In [6]:
```
### TEST 3

import numpy,utils

costfunctions = [
         ('Parameter Selection Criterion: favor low bias', lambda b,v: 9*b+v),
         ('Parameter Selection Criterion: favor low error',lambda b,v: b+v),
         ('Parameter Selection Criterion: favor low variance',lambda b,v: b+9*v),
]

# Load and partition the data
X,T = utils.Housing()
n = len(X)
X = [X[n*i//5:n*(i+1)//5] for i in range(5)]
T = [T[n*i//5:n*(i+1)//5] for i in range(5)]

print "GBBSPL"
for name,costfunction in costfunctions:
    print('\n\n%s\n'%name)
    evaluateModel(X,T,costfunction)
```

```
GBBSPL


Parameter Selection Criterion: favor low bias

bias:      38.78900 +/-  6.14712
variance:  19.39045 +/-  2.83253
error:     58.17945 +/-  7.10047
parameter:  0.00019 +/-  0.00008


Parameter Selection Criterion: favor low error

bias:      45.39340 +/-  5.91133
variance:   4.42108 +/-  1.34672
error:     49.81448 +/-  6.50931
parameter:  0.00542 +/-  0.00157


Parameter Selection Criterion: favor low variance
```

```
bias:      64.48379 +/- 10.70238
variance:   0.47024 +/-  0.08669
error:     64.95403 +/- 10.63188
parameter:  0.22932 +/-  0.10802
```

In [ ]: