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:

Datasets

- **`utils.Housing():`** This regression dataset is available at http://archive.ics.uci.edu/ml/datasets/Housing (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 (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 (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 (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]].`

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.

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:
  - $\mathrm{Bias}(Y)^2 = (\mathbb{E}_Y[Y - T])^2$
  - $\mathrm{Var}(Y) = \mathbb{E}_Y[Y - \mathbb{E}_Y[Y])^2]$
  - $\mathrm{Error}(Y) = \mathbb{E}_Y[(Y - T)^2]$

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.

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:

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

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 match the dimension of the targets, so for regression you will get an array of length N and for classification a matrix of shape $N \times \#\mathrm{classes}$ 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 [ ]:
```
def biasVarianceClassification(sampler, predictor, X, T, nbsamples=25):
    # --------------------------------
    # TODO: REPLACE BY YOUR CODE
    # --------------------------------
    import solutions
    return solutions.biasVarianceClassification(sampler, predictor, X, T, nbsa
mples=nbsamples)
    # --------------------------------
```

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

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

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.
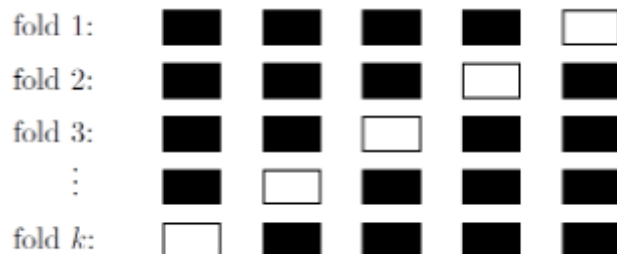
You need to extend this basic cross-validation procedure by a nested loop of 4-fold cross-validation that selects the best hyperparameters based on some criterion (cost function) to be determined. The nested loop is depicted below:
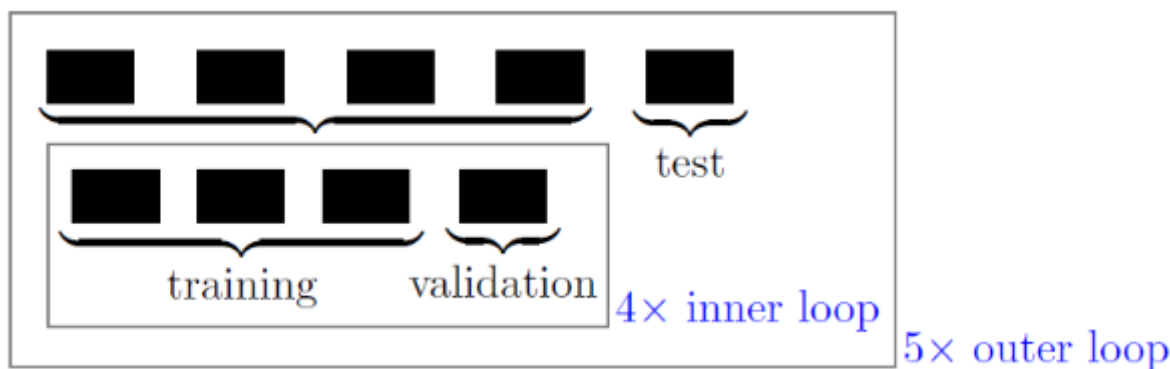
The full procedure for evaluation and hyperparameter selection procedure is shown in the diagram below with the part that you need to implement highlighted in red.

Build several **selection / test** splits from the data

Loop over **selection / test** splits

> Build several **training / validation** splits from the **selection** set
>
> Loop over **training / validation** splits
>
> > Loop over model parameters
> >
> > > Train several models with various random samples of the **training** set
> > >
> > > Compute the bias and variance of these models on the **validation** set
> > >
> > > Compute *cost*(bias,variance)
> >
> > Return the best parameter for this split (i.e. with the smallest associated cost)
>
> Use as parameter the *geometric* mean of best parameters for each split
>
> Train several models with various random samples of the **selection** set
>
> Compute the bias and variance of these models on the **test** set

Compute error bars (average and standard deviation) for bias, variance and error.

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.
   - 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 [ ]:  import numpy,utils

         def getbestparameter(Xselect,Tselect,costfunction):
             # -------------------------------
             # TODO: REPLACE BY YOUR CODE
             # -------------------------------
             import solutions
             return solutions.getbestparameter(Xselect,Tselect,costfunction)
             # -------------------------------
```

```
In [ ]:  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],4) ]

             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.conc
         atenate(Tselect,axis=0))
                 bias,variance = utils.biasVarianceRegression(sampler,predictor,Xtest,T
         test, nbsamples=20)

                 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(testb
         iases)))
             print('variance:  %8.5f +/- %8.5f'%(numpy.mean(testvariances),numpy.std(te
         stvariances)))
             print('error:     %8.5f +/- %8.5f'%(numpy.mean(testerrors),numpy.std(teste
         rrors)))
             print('parameter: %8.5f +/- %8.5f'%(numpy.mean(bestparameters),numpy.std(b
         estparameters)))
```

In [ ]:
```python
### 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 "YourGroupName"
for name,costfunction in costfunctions:
    print('\n\n%s\n'%name)
    evaluateModel(X,T,costfunction)
```