Quick Start Guide for CLOP

Amir Reza Saffari Azar Alamdari
Institute for Computer Graphics and Vision
Graz University of Technology, Graz, Austria
amir@ymer.org

Isabelle Guyon
Clopinet Enterprise, 955 Creston road
Berkeley, CA 94708, USA
isabelle@clopinet.com

May 16, 2006

Contents

1 Introduction

The main goal of this guide is to provide the *Model Selection Game* participants a quick starting point for using *Challenge Learning Object Package* (CLOP). This document will not cover information about the game, goals, rules, and etc, for those information please refer to the game's official website at `http://clopinet.com/isabelle/Projects/modelselect/challenge/`.

1.1 What is CLOP?

CLOP is a software package, which contains several ready-to-use machine learning algorithms to be used during the game by participants. It is based on Spider package [1] from Department of Empirical Inference for Machine Learning and Perception, Max Planck Institute for Biological Cybernetics, Tuebingen, Germany. CLOP has more algorithms provided by challenge organizers compared to Spider and it runs on MATLAB `http://www.mathworks.com`, but it doesn't depend on any particular toolbox of MATLAB. CLOP together with this manual can be downloaded from `http://clopinet.com/isabelle/Projects/modelselect/Clop.zip`.

Briefly, to provide a sufficient toolbox for the challenge, we did the following modifications to the original Spider package:

[1] `http://www.kyb.tuebingen.mpg.de/bs/people/spider/index.html`

- The entire spider is provided, but for the purpose of participating to the game, you are allowed to build models only from the following list of CLOP objects:
  - bias
  - chain
  - ensemble
  - gentleboost
  - gs
  - kridge
  - naive
  - neural
  - normalize
  - pc_extract
  - relief
  - rf
  - rffs
  - shift_n_scale
  - standardize
  - subsample
  - svc
  - svcrfe
  - s2n

  This list can also be obtained by typing `whoisclop` at the MATLAB prompt. Please refer to Section 4 for details.

- Some of the CLOP objects were not part of the original Spider; some have been modified (methods overloaded) for several reasons, including providing a simpler hyperparameterization (hiding some hyperparameters as *private*), returning always a discriminant value as output, providing more efficient implementations, and providing good default values to all hyperparameters.

For a valid challenge submission, please refrain from using other Spider learning objects, except using cross-validation objects, which allow you searching for the best model, but do not affect the final model. In addition, you should not modify the code of the CLOP objects, except if you find an error (please let us know) or make a change that does not affect the end result, but affects computational efficiency (we should be able to reproduce your results, given the model and the same set of parameters that you used). You can verify that your object is a valid CLOP object with the function `isclop`.

1.2 How to install CLOP?

First of all you should obtain CLOP from the website mentioned above, and save it into any directory that you prefer. Then you have to open and extract the zip file using any archiving software of your convenient, into any directory that you want to install the CLOP. For the rest of this manual we will call this destination directory, where the extracted files are stored, as MyProjects/CLOP/ and we will refer to MyProjects/ as root directory. For your case, you have to replace it with the path to your CLOP directory. Now you have CLOP installed on your computer. If you are running CLOP under Linux OS, you might need to compile some of our models, please refer to Section **??**.

The followings are the directory structures of CLOP and a brief description of what is inside:

1. MyProjects/CLOP/challenge_objects/: Challenge objects provided by the organizers.
2. MyProjects/CLOP/sample_code/: Sample code and lots of different functions written for the game.
3. MyProjects/CLOP/spider/: Original Spider is located here.

1.3 What I need to run CLOP?

Since CLOP runs under MATLAB, you have to have MATLAB running on your computer. Additionally, in order to use CLOP for this game you will need to download the datasets from `DATASET'S WEBSITE LINK`. For compatibility with the current settings inside the sample code, we suggest you to have your datasets extracted into the following directory: MyProjects/Data/. Of course you can choose other places, but then you have to modify manually some parts of the main.m program which is located inside the sample_code directory, see bellow for more information.

1.4 How to run CLOP?

We will describe first how to run a sample program using CLOP. Start MATLAB. Now change the current directory to where the sample code is located (in our case MyProjects/CLOP/sample_code/) using:

```
>> cd MyProjects/CLOP/sample_code/
```

and run main.m program:

```
>> main
```

If you have the datasets in MyProjects/Data/, this probably will run the main program successfully. You should notice from the MATLAB command window that the program displays some license agreement terms first and then loads a specific dataset and trains some algorithms on it, and finally tests the trained models and saves the results. If you experienced some problems check that the directory structure and path are correct.

1.5 Compilation of SVC

The `svc` model is originally based on a C code, so depending on your machines configuration, there might be a need for compilation. We have provided precompiled versions for Windows and they usually run without problems. But for Linux, you need to compile them again. The source code for `svc` is located in CLOP/challenge_objects/packages/libsvm-mat-2.8-1. For Linux users, there are two different Makefiles: Makefile_orig is the one which was provided by the authors of the SVM package, and Makefile_amir is what I used on my machine to compile it. The only difference is these two files is the name of the C compiler. So you might need to go to one of these files and change environmental variables in the beginning of the file according your system's settings and which version of C compiler you have installed. For Windows users, you can run make.m to compile this object again, if needed. There is a README.TXT file which describes installation procedure in more details.

1.6 More Details on Objects and Classes

Spider and CLOP both use object oriented programming style provided in MATLAB, which are called *class*es. If you do not know anything about MATLAB objects and/or object oriented programming, don't be scared away. You can learn how to use CLOP from examples, and in principle you will not need to deal with objects and classes. But you may definitely benefit from reading the (short) MATLAB help on objects. Briefly:

- An object is a structure (i.e. has data members), which has a number of programs (or methods) associated to it. The methods modify eventually the data members.

- The methods of an object *myObject* are stored in a directory named @myObject, which must be in your MATLAB path if you want to use the object (e.g. call addpath).

- One particular method, called *constructor* is named myObject. It is called (with eventually some parameters) to create a new object. For example: `>> myObjectInstance = myObject(someParameters);`

- Once an instance is created, you can call a particular method. For example: `>> myResults = myObjectMethod(myObjectInstance, someOtherParameters);`

- Note that myObjectInstance should be the first argument of myObjectMethod. Matlab *knows* that because myObjectInstance is an instance of myObject, it must call the method myObjectMethod found in the directory @myObject. This allows methods overloading (i.e. calling methods the same name for different objects.)

- Inheritance is supported in MATLAB, so an object may be derived from another object. A child object inherits from the methods of its parents. For example: `>> myResults = myParentMethod(myObjectInstance, someOtherParameters);` In that case, the method myParentMethod is found in the parent directory @myObjectParent, unless of course it has been overloaded by a method of the same name found in @myObject.

Refer to MATLAB's help for more information about classes and objects at `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/index.html`. Some useful functions for dealing with classes and objects in MATLAB are:

- **isa**: checks the class type
- **class**: returns the class
- **methods**: returns all the methods
- **struct**: lets you examine the data members
- **fieldnames**: returns a cell array with the field names

2 Sample Program

2.1 What is inside the main.m program?

The main program is written to provide participants with an easy-to-use template of how to utilize CLOP and build their own model with it [2]. Here we describe what are the different parts of this program and how to modify them for your own preferences.

1. **Initialization** This part of the code specifies some initial values for different variables which will be used through out the code:
    - It starts with cleaning the variables from workspace, cleaning the command window, and closing all figures. If you don't want to loose your available data and figures, remove this part of the code.
    - The next section defines the directory structure of your system. It is assumed that you want to have the following directories for your data and results, which is well designed to keep track of different activities and results required for a valid submission:

[2] With *model*, here we mean any combination of algorithms that one might define to apply to a dataset. For example the sequence of normalizing the data, and then classifying it using a neural network can be described as a model.

  - my_root: root directory where everything is there,
    Example: MyProjects/.
  - data_dir: data directory where datasets are,
    Example: MyProjects/Data/.
  - code_dir: code directory where CLOP is located,
    Example: MyProjects/CLOP/.
  - resu_dir: directory where results will be stored,
    Example: MyProjects/Results/.
  - zip_dir: directory where zip files will be stored,
    Example: MyProjects/Zipped/.
  - model_dir: directory where models will be stored,
    Example: MyProjects/Models/.
  - score_dir: directory where model scores will be stored,
    Example: MyProjects/Scores/.

- ForceOverWrite defines if you want the system to overwrite your previous results (if they are available in results, models, and zip directories). The default value is 1, which will not disturb you for overwriting questions, but be careful that in this case there is always risk of loosing valuable information and results.
- DoNotLoadTestData defines if you want the system to load the test set or not. This is used for cases where the size of test set is very large and loading it to memory will not be efficient for training phase. The default value is 1, which will not load the test sets.
- MergeDataSets defines if you want the system to merge training and validation sets. This is useful when you have labels for validation sets and you want to use them as extra training samples. The default value is 0, which will not merge the training and validation sets.
- FoldNum defines if you want the system to do a k-fold cross-validation or not, where k=FoldNum. The default value is 0, which will not perform cross-validation.
- The next step is to define which datasets you want to train, you can define this in the dataset cell array. For starting it is a better idea to start with just one dataset and check how your models are performing on it. In addition note that not always a single model is a good choice for all datasets.
- In order to keep your models separate from your current code, by default in training and testing section the program will call another function named model_examples.m with a user chosen model name and the dataset. You can easily define there which algorithms you would like to try over datasets under the chosen name, without a need to change the code in main program. This procedure will facilitate the model selection process too. For more information about model_examples.m, please refer to Section 4. You can define your model names in the modelset cell array.

- Now that you have defined your preferences, the system tries to generate and add the current directories (and their subsequent directories) to MATLAB's search path. This will finalize the initialization section.

2. **Train/Test** Here the code will start a loop for training and testing your models over different datasets you have specified above. Each loop consists of the following steps:

   - Loading the dataset and creating data structures: Since the original datasets provided from challenge website are stored in text files, this section will load all of them into proper data structures suitable for MATLAB. Because you will need these variables in your later experiments again and the process of loading them from text files usually takes longer time compared to loading variables from a MATLAB save file (.mat format), it also saves the data structures in the same data locations. The next time that you run the code over the same dataset, this section will automatically check whether the .mat format is available or not, and in the case of existing file it will load data from that .mat file, resulting in a very low loading time. Please refer to Section 3 for more information about the data variables.
   - Looping over different models: With the `modelset` array you can define several models to be tested over different datasets. This part of the code starts a loop over those models, which includes the following parts:
     - First of all, it calls `model_examples.m` function with the model and dataset names, in order to get back a valid CLOP or Spider model. Please refer to Section 4 for more information on how to define valid models.
     - If you have defined the algorithm to perform a cross-validation step, the system starts this step and trains the model FoldNum times.
     - Now that the program has the data and model, it is time to train the algorithms specified within the model, and obtain the training results. This is done simply by calling `train` function with model and training data.
     - After training, it will calculate the balanced error rate of the trained model, followed by computing a very tentative guess for test BER.
     - Now that the model is trained over the training set, it is time for testing it over validation and test sets, this task is accomplished easily by calling `test` function with model and proper data set.
     - The next step is to save the results into specified directories in appropriate official format which can be sent directly to the challenge website. It follows by saving the models too, which are needed for a valid final entry of the game.

3. **Make Archive** The finishing part of the code is to make a zip archive of all necessary files needed for submission to the challenge website for verification.

As it can be seen from these steps, the main program contains almost everything that a participant needs to enter the competition and produce proper results. We suggest you to make a backup copy of this program, and then modify different parts of it according to your interests. Since this is a model selection competition, you may want to replace the simple k-fold cross-validation provided as example by a more elaborate model selection strategy to select an optimum model or set of hyperparameters.

3 Data Structure

The data object that has been used in CLOP, consists of the following fields (let's name the data variable `D`):

1. `D.train`: training data is stored here.
2. `D.valid`: validation data is stored here.
3. `D.test`: test data is stored here.

Each of these fields has two additional subfields, called `.X` and `.Y`. In `.X` field the raw data are stored in a matrix format with example arranged in rows and features in columns. The `.Y` field contains the labels for the corresponding set of examples in a one dimensional vector format. Note that if the labels are not provided in the datasets (always for test set and just for validation set before release of validation labels), then this field will be an empty vector. Additionally one can get statistics about the data using function `data_stats(D)`.

4 Defining Models

For the purpose of the game, a valid *model* is defined as a combination of learning objects from a predefined list (type `whoisclop` at the MATLAB prompt to get a the full list of allowed CLOP learning objects; to check that a particular object is a valid CLOP object, type `isclop(object)`).

A typical model usually (but not necessarily) consists of the following parts[3]:

1. Preprocessing
2. Feature Selection
3. Classification

[3] Note that the two first steps should not necessarily be in this order. In particular, since feature selection changes the number of features, normalization, which is a preprocessing method, may need to be done/redone after feature selection

4. Postprocessing

The simplest model for our challenge can be just a classifier. Defining a model is a very simple task within CLOP framework. For example the following code makes a neural network classifier available with its default hyperparameters:

```
>> myClassifier = neural
```

and this code defines a linear support vector machine classifier with a shrinkage (regularization) value of 0.1:

```
>> myClassifier = svc({'shrinkage=0.1'})
```

Note the way that a hyperparameter value (shrinkage in this case) is passed to the object constructor (svc in this case). This is the general method to assign different values for hyperparameters rather than their default values. If you want to know what are the hyperparameters of each model, you can simply type `>> help OBJECT_NAME` in MATLAB command window, where `OBJECT_NAME` is the name of a model. This will bring up information available for each model together with the list of hyperparameters used inside the model.

In general, there are two types of hyperparameters for each model: one is named as *public* and the other is *private*. For the competition, only the public hyperparameters can be tuned. The method to do that is via the constructor, as showed in examples. To find out which hyperparameters are public, use `default(OBJECTNAME)` in MATLAB command window. It is also possible to set the hyperparameters directly (outside of the constructor), but do this *at your own risks* since this may generate inconsistencies.

4.1 How to combine different models?

There exist two different ways to combine several models with each other. The first one is serial combination where outputs of each model is fed to the inputs of the next model. For example suppose that we want to normalize the data and then classify it with a neural network. This is a serial combination of algorithms where output of normalization step is supposed to be an input for neural network classifier. This type of combinations can be done easily using `chain` object. The following is the sample code you would need for the example described above:

```
>> myModel = chain({normalize , neural})
```

Now when you train or test `myModel` with an input data, first the `normalize` algorithm will operate on the data and then it will pass the resulted data to `neural` object to classify it:

```
>> [Outputs, myModelTrained] = train(myModel, D.train)
```

In this example it is supposed that `D.train` contains the training data. The `myModelTrained` is the resulted model after training, while `Outputs` contains information about predicted labels and other output variables. For more information about training and testing models, please refer to Section 5.

The other way of combining different algorithms is to combine different models with each other in a parallel style. This is usually known as *ensemble* methods in machine learning, and the goal is often to combine outputs of different learning algorithms to improve the classification performances. For example we want to have two classifiers, one neural network and one naive Bayes, trained on the same data, and then add their outputs to create the final results. The following code will generate the desired model and train it on `D.train` set:

```
>> myNeural = chain({normalize, neural({'units=3', 'balance=1'})})
>> myNaive = naive
>> myClassifier = ensemble({myNeural, myNaive})
>> [Outputs, myModelTrained] = train(myClassifier, D.train)
```

Note that for ensemble methods there are more sophisticated algorithms to train and combine outputs of different classifiers, like *bagging* and *boosting*. We consider those methods as individual classifiers and they will be described in next sections in details.

Note that the nth model of a chain or ensemble can be easily accessed with the curly bracket notation. For example in the following model, we want to access to the `neural` object, which is the 2nd element in chain and 3rd in ensemble object:

```
>> myChain = chain({normalize, ensemble({svc, kridge, neural}), bias})
       chain
      {
      1:normalize center=0
      2:ensemble
      3:bias option=1
      }
>> C{2}{3}
      neural units=10 shrinkage=1e-014 balance=0 maxiter=100
```

4.2 Preprocessing Methods

The following section shows different preprocessing models available within CLOP for participants.

4.2.1 standardize

- **Description:** Standardization of the features (the columns of the data matrix are divided by their standard deviation; optionally, the mean is first subtracted if center=1). Note that a lot of methods benefit from this preprocessing, particularly neural networks.
- **Hyperparameters:** $center \in \{0, 1\}$
- **Default Values:** $center = 1$
- **Example:** `>> myModel = chain({normalize({'center=1'}) , naive})`

4.2.2 normalize

- **Description:** Normalization of the lines of the data matrix (optionally the mean of the lines is subtracted first if center=1). Some methods benefit from this preprocessing, particularly the polynomial kernel methods. It is sometimes best to normalize after feature selection or both before and after.
- **Hyperparameters:** $center \in \{0, 1\}$
- **Default Values:** $center = 0$
- **Example:** `>> myModel = chain({standardize({'center=1'}) , naive})`

4.2.3 shift_n_scale

- **Description:** Performs this transformation globally on the data matrix $X = (X - offset)/scale$, while offset and factor are set as hyperparameters, or subject to training. Optionally performs in addition $log(1 + X)$.
- **Hyperparameters:** $offset \in [-\infty, \infty]$, $factor \in [0^+, \infty]$, take_log $\in \{0, 1\}$
- **Default Values:** $offset = min(X)$, $factor = max(X - offset)$, take_log $=$ 0
- **Example:** `>> myModel = chain({shift_n_scale({'take_log=1'}) , naive})`

4.2.4 pc_extract

- **Description:** Extract f_max number of features with principal component analysis.
- **Hyperparameters:** f_max $\in [0, \infty]$
- **Default Values:** f_max $= \infty$
- **Example:** `>> myModel = chain({pc_extract({'f_max=50'}) , naive})`

4.2.5 subsample

- **Description:** Make a subset of p_max number of the training patterns. It is possible to specify which patterns should be included in the resulting subset, by giving additional input to the `subsample` function which contains the indexes of those patterns. With balance hyperparameter you can specify whether the resulting subset should be a balanced set according to the number of class members or not. Note that subsampling can be combined with the ensemble object to implement bagging, for example[4]:

```
 for k=1:100
  baseModel{k}=chain({subsample({'p_max=1000', 'balance=1'}), kridge});
 end
 myModel = chain({standardize, ensemble(baseModel, 'signed_output=1'),
bias});
```

- **Hyperparameters:** p_max $\in [0, \infty]$, $balance \in \{0, 1\}$
- **Default Values:** p_max $= \infty$, $balance = 0$
- **Example:** `>> myModel = chain({subsample({'p_max=100'}) , naive})`

4.3 Feature Selection Methods

The following section shows different feature selection models available within CLOP for participants. The following notation for hyperparameters is commonly used for all feature selection algorithms:

- `f_max` defines the maximum number of features to be selected using the target model.
- `w_min` defines a threshold on the ranking criterion W of the target model. If $W(i) <=$ w_min, the feature i is eliminated. W is vector with non-negative values, so a negative value of w_min means all the features are kept.

4.3.1 s2n

- **Description:** Signal-to-noise ratio coefficient for feature ranking. This method ranks features with the ratio of the absolute difference of the class means over the average class standard deviation. This criterion is similar to the Fisher criterion, the Ttest criterion, and the Pearson correlation coefficient. It can be thought of as a *linear univariate* feature ranking method. The top ranking features are selected and the new data matrix returned. The hyperparameters can be changed after construction of the object to allow users to vary the number of features without retraining.

[4] Note that this does not take advantage of the *out-of-bag examples* to compute an estimate of the test error, but can be considered as an approximation to bagging algorithm

- **Hyperparameters:** f_max $\in [0, \infty]$, w_min $\in [-\infty, \infty]$
- **Default Values:** f_max $= \infty$, w_min $= -\infty$
- **Example:** `>> myModel = chain({s2n({'f_max=100'}) , naive})`

4.3.2 relief

- **Description:** This method ranks features with the Relief coefficient. Relief is a method based on the nearest neighbors scoring features according to their relevance, in the context of others. It is a non-linear multivariate feature ranking method. In our implementation, it is slow for large numbers of patterns because we compute the entire distance matrix. We chunk it if it is very large, to avoid memory problems. The top ranking features are selected and the new data matrix returned. The hyperparameters can be changed after construction of the object to allow users to vary the number of features without retraining. `k_num` defines the number of neighbors in the Relief algorithm.
- **Hyperparameters:** f_max $\in [0, \infty]$, w_min $\in [-\infty, \infty]$, k_num $\in [0, \infty]$
- **Default Values:** f_max $= \infty$, w_min $= -\infty$, k_num $= 4$
- **Example:** `>> myModel = chain({relief({'f_max=100', 'k_num=5'}) , naive})`

4.3.3 gs

- **Description:** Forward feature selection with Gram-Schmidt orthogonalization. This is a forward selection method creating nested subsets of complementary features. The top ranking features are selected and the new data matrix returned. Note that if you want to change the value of `f_max` after training, it cannot be set to a larger number than the number `f_max` used for training (or it will be chopped at f_max).
- **Hyperparameters:** f_max $\in [0, \infty]$
- **Default Values:** f_max $= \infty$
- **Example:** `>> myModel = chain({gs({'f_max=100'}) , naive})`

4.3.4 rffs

- **Description:** Random Forest used as feature selection filter. The *child* argument, which may be passed in the argument array, is an `rf` object, with defined hyperparameters. If no child is provided, an `rf` with default values is used.
- **Hyperparameters:** f_max $\in [0, \infty]$, w_min $\in [-\infty, \infty]$, child

- **Default Values:** f_max = ∞, w_min = $-\infty$, child=rf
- **Example:** `>> myModel = chain({rffs({'w_min=0.2'}) , naive})`

4.3.5 svcrfe

- **Description:** Recursive Feature Elimination filter using SVC. This is a backward elimination method creating nested subsets of complementary features. The *child* argument, which passed in the argument array, is an `svc` object, with defined hyperparameters. If no child is provided, a linear `svc` with default values is used.
- **Hyperparameters:** f_max $\in [0, \infty]$, child
- **Default Values:** f_max = ∞, child=svc
- **Example:** `>> myModel = chain({svcrfe({'f_max=100'}) , naive})`

4.4 Classification Methods

The following section shows different classification models available within CLOP for participants. The following notation for hyperparameters is commonly used for all classification algorithms:

- For kernel methods, the general kernel equation is

  $$k(x_1, x_2) = (coef0 + x_1.x_2)^{degree} exp(-gamma \|x_1 - x_2\|^2)$$

  Note that (.) is vector dot product operator. Some examples of mostly used kernels are:

  1. *Linear Kernel:* $degree = 1, coef0 = 0, gamma = 0$, $k(x_1, x_2) = (x_1.x_2)$
  2. *Polynomial degree N Kernel:* $degree = N, coef0 = a, gamma = 0$, $k(x_1, x_2) = (a + x_1.x_2)^N$
  3. *RBF Kernel:* $degree = 0, coef0 = 0, gamma = \gamma$, $k(x_1, x_2) = exp(-\gamma \|x_1 - x_2\|^2)$

- `units` usually defines the number of substructures needed for each algorithm. For example, in the case of a `neural` object, it defines the number of hidden neurons, while in boosting methods, it is the number of weak learners to be used in the algorithm.
- `balance` is a flag used to specify if the algorithm should balance the number of class members before training using subsampling method.
- `shrinkage` defines usually the regularization parameter used in each algorithm.

4.4.1 kridge

- **Description:** Kernel ridge regression. This object trains a regression learning machine using the least square loss and a weight-decay or *ridge* specified by the *shrinkage* parameter.
- **Hyperparameters:** coef0 $\in [0, \infty]$, degree $\in [0, \infty]$, gamma $\in [0, \infty]$, shrinkage $\in [0, \infty]$, balance $\in \{0, 1\}$
- **Default Values:** coef0 $= 1$, degree $= 1$, gamma $= 0$, shrinkage $= 1e-14$, balance $= 0$
- **Example:** `>> myModel = chain({normalize , kridge({'coef0=0.1', 'degree=2'})})`

4.4.2 svc

- **Description:** Support vector classifier. This object trains a *2-norm* SVC, i.e. the shrinkage parameter is similar to the ridge in kridge. There is no box constraint (bound on the alphas).
- **Hyperparameters:** coef0 $\in [0, \infty]$, degree $\in [0, \infty]$, gamma $\in [0, \infty]$, shrinkage $\in [0, \infty]$
- **Default Values:** coef0 $= 0$, degree $= 1$, gamma $= 0$, shrinkage $= 1e - 14$
- **Example:** `>> myModel = chain({normalize , svc({'degree=0', 'gamma=0.1', 'shrinkage=0.1'})})`

4.4.3 naive

- **Description:** Naive Bayes classifier. Two separate implementations are made for binary and continuous variables (the object switches automatically). For binary variables, the model is based on frequency counts; for continuous variable, the model is a Gaussian classifier.
- **Hyperparameters:** None
- **Default Values:** None
- **Example:** `>> myModel = chain({normalize , naive})`

4.4.4 neural

- **Description:** Neural networks classifier. Two layer neural network (a single layer of hidden units). The shrinkage corresponds to a weight decay or the effect of a Bayesian Gaussian prior. `units` is the number of hidden neurons, and `maxiter` defines the number of training epochs.
- **Hyperparameters:** units $\in [0, \infty]$, maxiter $\in [0, \infty]$, shrinkage $\in [0, \infty]$, balance $\in \{0, 1\}$

- **Default Values:** units = 10, maxiter = 100, shrinkage = $1e - 14$, balance = 0
- **Example:** `>> myModel = chain({normalize , neural({'units=5'})})`

4.4.5 rf

- **Description:** Random Forest classifier. This object builds an ensemble of tree classifiers with 2 elements of randomness: (1) each tree is trained on a randomly drawn *bootstrap* subsample of the data (approximately 2/3 of the examples); (2) for each node, the feature to split the node is selected among a random subset of all features. `units` is the number of trees, and `mtry` defines the number of candidate feature per split.
- **Hyperparameters:** units $\in [0, \infty]$, mtry $\in [0, \infty]$
- **Default Values:** units = 100, mtry = $\sqrt{\text{Feat\_Num}}$
- **Example:** `>> myModel = chain({normalize , rf({'units=200'})})`

4.4.6 gentleboost

- **Description:** This object builds an ensemble of classifiers (called weak learners) by sequentially adding weak learners trained on a subsample of the data. The subsamples are biased toward the examples misclassified by the previous weak learner. Gentleboost is a variant of the adaboost algorithm, which is less sensitive to data outliers because it puts less weight on misclassified examples and weighs the weak learners evenly. The base classifier is defined separately and can be any of the classification methods defined above. `units` is the number of weak learners, `subratio` defines the ratio of subsampling compared to the original dataset, and `rejNum` is the number of different trials to get a weak learner before stopping the whole iteration, if the weighted error of a weak learner is over 0.5.
- **Hyperparameters:** units $\in \{1, 2, ..., \infty\}$, subratio $\in [0, 1]$, rejNum $\in \{1, 2, ..., \infty\}$, balance $\in \{0, 1\}$
- **Default Values:** units = 5, subratio = 0.9, rejNum = 3, balance = 1
- **Example:**
  ```
  >> myBase = naive
  >> myModel = chain({normalize , gentleboost(myBase, {'units=10'})})
  ```

4.5 Postprocessing Methods

The following section shows different postprocessing models available within CLOP for participants.

4.5.1 bias

- **Description:** Bias optimization. It calculates a threshold value that will be applied to the outputs of classifier, optimizing several factors listed bellow. The `option` can be one of following items:
  1. minimum of the BER.
  2. break-even-point between sensitivity and specificity.
  3. average of the two previous results if they do not differ a lot, otherwise zero.
  4. values that gives the same fraction of positive responses on the test data than on the training data (transduction).
- **Hyperparameters:** $option \in \{1, 2, 3, 4\}$
- **Default Values:** $option = 1$
- **Example:** `>> myModel = chain({normalize({'center=1'}) , naive, bias({'option=2'})})`

4.6 Model Selection Methods

In the `main.m` example script we provide, we illustrate how the Spider can be used to perform a selection of the CLOP models. In the script, we use the Spider object cv, which implements the traditional k-fold cross-validation algorithm. Since this is a model selection contest, we encourage you to develop your own algorithms for this purpose. The main goal of developing CLOP was to provide the participants with a diverse set of learning algorithms that should be sufficient to achieve competitive results. By restricting them to the use of CLOP models, they can focus on model selection rather than spending time tuning their own classification algorithms. So use this opportunity and develop your ideas for better model selection systems. This may include developing better:

- hyperparameter search strategies
- model architectures (by combining CLOP modules with combinations of chains and ensembles)
- model assessment (e.g. cross-validation, complexity penalization, etc.)

The Spider provides a few objects implementing some of these tasks:

- `param`: this object allows you to specify ranges of parameter values
- `cv`: this object implements k-fold cross-validation
- `gridsel`: this object integrates the functions of param and cv to perform model selection with grid search

It is noteworthy that the model selection game is as much an *ensemble method* game as it is a *model selection* game, since hyperparameter selection can to a large extent be circumvented by using an ensemble of methods.

Note that objects like r2w2_sel and bayessel are specific to the Spider implementation of SVMs, which is different from the one of CLOP. r2w2_sel concerns feature selection. Please do not use it since we restricted the feature selection methods to the ones provided in this manual. bayessel adjust the C hyperparameter in 1-norm SVM. This is not useful to CLOP users, since in CLOP, we provide only a 2-norm SVM.

4.7 How to use **model_examples.m**?

In order to keep the model creation separate from the main code that might be used by participants, we use another program which is called model_examples.m. Using this scheme, one can easily define lots of different models without a need to modify the main.m program. This style is not necessary but we recommend you to follow the proposed structural system.

In model_examples.m, the first section is used to checks empty calls of this function. Next, there exist several proposed models and methods which some of them are common for all datasets, and some has specific instructions to deal with different characteristics of different datasets. Check this section carefully to get a general idea of how to create and write your own models. Note that the proposed models are not optimal for this challenge, and we have created them just to show different capabilities of CLOP. In order to be competitive in the challenge, you would need to create your models and tune their parameters according to any model selection strategy that you have in your mind.

5 Training and Testing

After you have defined your models, you can easily train them using train command as bellow:

```
>> myModel = chain({normalize({'center=1'}) , naive, bias({'option=2'})})
>> [TrainOutputs, myModelTrained] = train(myModel, D.train)
```

In this example, we assume that there is a data object named D already available in the workspace which contains training data and labels. The train function returns the original model with tuned parameters in myModelTrained, while TrainOutputs will have predicted labels as .X subfield, (TrainOutputs.X) and original target training labels as .Y subfield, (TrainOutputs.Y). The TrainOutput object can later be used to evaluate the performance of training phase, like computing the BER, AUC, and etc.

After training a model, we usually are interested in testing the algorithm over unseen data examples. This can be easily done with test function as follows:

```
>> TestOutputs = test(myModelTrained, D.test)
```

Now the `TestOutput` has the predicted labels again in `.X` subfield, but since the labels for test sets will not be provided, the `.Y` field will be empty. The `TestOutput.X` can be passed further to other functions such as `save_outputs` to generate output files which are suitable to send directly to online website.

6 Credits

The organization of this competition was a team effort to which many have participated. We are particularly grateful to Olivier Guyon (MisterP.net) who implemented the back-end of the web site. The front-end follows the design of Steve Gunn (University of Southampton), formerly used for the NIPS 2003 feature selection challenge. We are thankful to Bernd Fischer (ETH Zurich) for administering the computer resources. Other advisors and beta-testers are gratefully acknowledged: Gideon Dror, Joachim Buhmann, Yoshua Bengio (University of Montreal), Asa Ben-Hur (Colorado State university), Lambert Schomaker (University of Groningen), and Vladimir Vapnik (NEC, Princeton). The Challenge Learning Object Package (CLOP) is based on code to which many people have contributed: The creators of the spider: Jason Weston, Andre Elisseeff , Gikhan Bakir , Fabian Sinz. The developers of the packages attached: Chih-Chung Chang and Chih-JenLin Jun-Cheng (LIBSVM), Chen, Kuan-Jen Peng, Chih-Yuan Yan, Chih-Huai Cheng, and Rong-En Fan (LIBSVM Matlab interface), Junshui Ma and Yi Zhao (second LIBSVM Matlab interface), Leo Breiman and Adele Cutler (Random Forests), Ting Wang (RF Matlab interface), Ian Nabney and Christopher Bishop (NETLAB), Thorsten Joachims (SVMLight), Ronan Collobert (SVM Torch II), Jez Hill, Jan Eichhorn, Rodrigo Fernandez, Holger Froehlich, Gorden Jemwa, Kiyoung Yang, Chirag Patel, Sergio Rojas. This project is supported by the National Science Foundation under Grant N0. ECS-0424142. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Predicant biosciences, Microsoft, and Unipen provided additional support permitting to grant prizes to the winners.