

# Documentation of LibHIR

LibHIR is an open source library implementing the model, Hierarchical Interaction Representation (HIR), to provide a collaborative prediction under the scenarios with multiple entities, features and contexts. It can be implemented for different tasks, i.e., regression, classification, ranking and so on. This document explains the usage of LibHIR. You can get LibHIR from <https://github.com/TagineerDai/LibHIR>.

Please read the [COPYRIGHT](#) before using or modifying LibHIR.

## Table of Contents

- Quick Start
- Installation
- Data Format and Transfer
  - SVM Format
  - HIR Format
- User Guide
  - Transform Script Usage
  - HIRTrain Usage
  - HIRTest Usage
- Developer Guide
- Additional Information

## Quick Start

If you are new to HIR and the task is classification or regression, please try the command-line tool version of LibHIR. The script `prepro.py` transfers the data with the svm format into the HIR format, and initializes the config and the model files. Generally, the data has observed features and contexts. For more information, please refer to the Data Format and Transfer section.

Usage: `prepro.py --rawfile filename.txt`

The preprocessing script generates four files, including `model.txt`, `config.txt`, `train.txt` and `test.txt`. These files should be copied to the same directory of the programs `HIRTrain` and `HIRTest` on Linux, or that of `HIRTrain.exe` and `HIRTest.exe` on Windows. If we run the files with option `-h`, it will show the usage of these files. For further information, please view the section of user guide.

Usage: `HIRTrain/HIRTest [option parameter]`

## Installation

### Linux

On Linux systems, please type `make` in the terminal to build the `HIRTrain` and `HIRTest` files.

To uninstall LibHIR, please use the command `make clean`.

## Windows

The executable files `HIRTrain.exe` and `HIRTest.exe` of the corresponding command-line tool are offered. Once you develop a program based on LibHIR, please consult `Makefile` about how to manually build the project on Linux platform, or how to simply use an IDE.

## Data Format and Transfer

Under the `data` directory, we offer a python script `transform.py` to prepare the svm format based on the sample data, i.e., movielens. And `prepro.py` under the `script` directory can transfer the data from the svm format into the HIR format, and generate config and model files. Please view the section of user guide for more details.

### SVM Format

The LibHIR toolkit is compatible with the data format of libsvm, i.e., the svm format. If there exist features of entities and contextual information, these auxiliary features and contexts are concatenated after records, and are splited by spaces. The data with this format is the raw input of the `prepro.py`, and the format is shown as follows:

```
<label> <index1>:<value1> ... <indexk>:<valuek> <feature1> ... <featurek> . .
```

Each line is a record and ended by the '\n' character. The label is a real value in a regression task, or a one hot category label in a classification task. In the `<index>:<value>` pairs, the indexes of entities are integers. The feature values can be stored in float. If there exist observed features of entities and contexts of a record, we concatenate them into a feature vector. This vector can be with arbitrary dimensionality, and cannot be updated by the learning process.

### HIR Format

The preprocess script `prepro.py` can load the data with the svm format and generate record file, model file, and config file. Please use the option `-h` or `--help` to view the usage of the options and parameters.

The config file `config.txt` contains three sets of parameters. The `filenames` are names of the model file, training data file and testing data file, the `data specification` section is for the HIR model and records, and the `size of parameter` section includes indices and number of entities. The config format is `[param name] = [param value]`, and line comments in the config are started with a `#`.

All specifications of the HIR model and HIR records can are generated by the config file. Modifying the parameters in `config.txt` directly is not suggested. The mismatch between configuration, model and record files will lead to the fatal error.

The `model.txt` contains latent features of all entities, the parameters of the HIR model, such as the

tensor layer, the hidden layer, the joint representations, and the final representation. If you are developing your own task with the HIR representation, please modify the `prepro.py`, the `HIRModel::ModelLoading` and the `HIRModel::ModelSaving`.

The record files are the data sets, i.e., the `train.txt` and the `test.txt`.

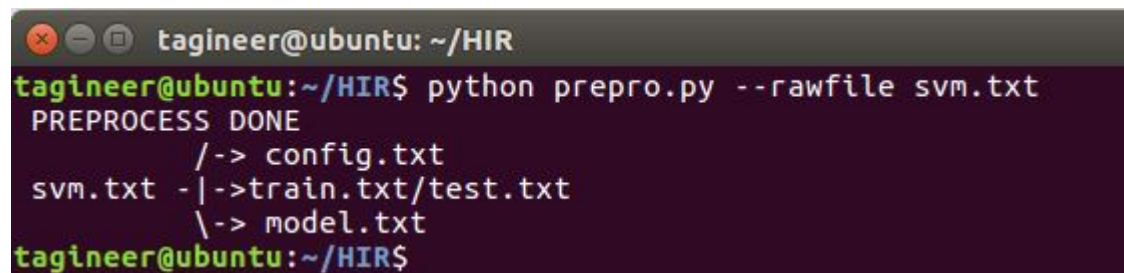
## User Guide

The command-line tool offers implementation of the basic regression and classification tasks based on the HIR representation.

### Transform Script Usage

sample:

```
python prepro.py --rawfile svm.txt
```

A terminal window with a dark background and light-colored text. The prompt is 'tagineer@ubuntu: ~/HIR'. The command 'python prepro.py --rawfile svm.txt' has been executed. The output shows 'PREPROCESS DONE' followed by a series of file operations: '/-> config.txt', 'svm.txt -|->train.txt/test.txt', and '\-> model.txt'. The prompt returns to 'tagineer@ubuntu: ~/HIR\$'.

```
tagineer@ubuntu: ~/HIR
tagineer@ubuntu:~/HIR$ python prepro.py --rawfile svm.txt
PREPROCESS DONE
      /-> config.txt
svm.txt -|->train.txt/test.txt
      \-> model.txt
tagineer@ubuntu:~/HIR$
```

Once you have installed python and numpy, the command `prepro.py` can be used to preprocess the data with the svm format. The usage of the options are as follows.

Option	Detail
--rawfile	the name of svm format data.
--Rtrain	the ratio of split. Rtrain = training : total record
--category	the type of task. 1--regression 2--biClass 3..n--multiClass
--hlayer	number of the hidden layers
--dim	dimensionality of feature and representation.

The Rtrain is a float value in (0,1). And the category, layer and dim are integers.

### HIRTrain Usage

Sample:

```
HIRTrain -r 100 -o 1 -a 0.001 -e 5
```

In the HIRTrain program, the HIR model is specified by the config file `config.txt` and trained by using the data in `train.txt`. The loss will show at each epoch, and the model after training process will be saved in `model.txt`.

Option	Detail	Value
-r	number of epoch	positive integer
-o	objective function	0(MAE), 1(RMSE), 2(LL), 3(GOLD)
-e	evaluation function	4(MAE), 5(RMSE), 6(AUC), 7(ACC)
-a	learning rate	float number in (0,1)
-m	learning method	0(SGD)
-h	help	the help information

You can assign a positive integer to the number of epoch. The default learning method is Stochastic Gradient Descent. The learning rate (alpha) is a float value, which represents the update ratio in the backpropagation process. Both the regression and classification tasks have corresponding objective functions and evaluations, the mismatch between tasks and options will cause error. For regression tasks, the optional objective functions are the abstract distance (MAE) and the rooted mean squared error (RMSE), and the optional evaluation functions are the mean abstract error (MAE) and the rooted mean squared error (RMSE). For classification tasks, the optional objective functions are the log-loss (LL) and the zero-one loss (GOLD), and the optional evaluation functions are the area under ROC curve (AUC) and the mean accuracy (ACC).

## HIRTest Usage

Sample:

```
HIRTest -e 5 -f answer.txt
```

```

C:\Windows\system32\cmd.exe
D:\HIR>HIRTrain.exe -r 10 a -0.1 -o 1 -e 6 -m 10
=== LOSS ===
epoch 0 : 3.43359
epoch 1 : 3.29508
epoch 2 : 3.13272
epoch 3 : 2.94048
epoch 4 : 2.68631
epoch 5 : 2.32134
epoch 6 : 1.82314
epoch 7 : 1.06864
epoch 8 : 0.706825
epoch 9 : 0.6464
Save at model.txt
HIR train ended.
Press Enter to exit.

D:\HIR>HIRTest.exe -e 6
The evaluation result is: 0.505803
Predict answer is saved at predict.txt
HIR test ended.
Press Enter to exit.

D:\HIR>

```

In the HIRTest program, the evaluation function depends on the task. Mismatch between the task and the evaluation function will cause error. The predicted result can be saved in a given file, e.g., `answer.txt` in the sample code.

Options	Details	Value
-e	evaluation function	4(MAE), 5(RMSE), 6(AUC), 7(ACC)
-f	prediction file	filename string
-h	help	help information

For regression tasks, the predicted result of each record is a double value, the optional evaluation functions are the mean abstract error (MAE) and the rooted mean squared error (RMSE). For classification tasks, the predicted value of each record is a one-hot vector, and the optional evaluation functions are the area under the ROC curve (AUC) and the classification accuracy (ACC).

## Developer Guide

With the guide of the Makefile, you need to include the header file `HIR.h` in your C++ source files, then compile and link as introduced in the `Makefile`. The sample codes `HIRTrain.cpp` and `HIRTest.cpp` are references on how to develop your own task based on LibHIR. The five modules of LibHIR are introduced as follows.

### HIRConfig

This module loads and stores the config file, and specifies hyperparameters of the model, such as the number of records. Once some additional settings need to be added in your task, please modify the `HIRConfig` class and the corresponding construction function, and then instantiate it. The compatible formats of parameters are integer, float, string and vector.

```
//HIRConfig.h
class HIRConfig {
public:
    //file name
    std::string mfname = "model.txt";
    std::string tfname = "train.txt";
    std::string efname = "test.txt";
    // Data specification
    int rtrain = 0;
    int rtest = 0;
    int fwid = 0;
    int task = 1;//regression default
    int e_num = 0;
    int hnum = 1;
    int dim = 10;
    // Size of parameter
    std::vector<int> maxe, idxe, size;
    HIRConfig();
    ~HIRConfig();
};
```

```
//HIRConfig.cpp
HIRConfig::HIRConfig() {
    HIRParser parser = HIRParser("config.txt");
    parser.get("rtrain", rtrain);
}
```

```

    parser.get("rtest", rtest);
    parser.get("fwid", fwid);
    parser.get("task", task);
    parser.get("dim", dim);
    parser.get("hnum", hnum);
    parser.get("enum", e_num);
    parser.get("maxe", maxe);
    parser.get("idxe", idxe);
    parser.get("size", size);
}

```

There are two ways to modify the config file. The simpler one is to add an additional attribution in the `config.txt` directly, and the shortcoming is that the file should be modified every time after running the preprocess script. The other one is to modify the `outconfig()` function in the script `prepro.py`.

```

def outconfig(cfname):
    global rtrain, rtest, fwid, task, dim, hnum
    global entity, imax, index, size

    # finish the config.txt in format given
    config = open(cfname, 'w')
    s = """###Filename
modelFile = model.txt
trainRecord = train.txt
testRecord = test.txt

### Data specification
# record number [integer]
rtrain = {0}
rtest = {1}
# feature width [vector]
fwid = {2}
# target scale [integer]
task = {3}
# model specification
model = 0
dim = {4}
hnum = {5}
enum = {6}
""".format(rtrain, rtest, fwid, task, dim, hnum, entity)
    config.write(s)
    s = """
### Size of parameter
# max entity [vector]
maxe = {0}
# entity feature index [vector]
idxe = {1}
# size of network [vector]
size = {2}""".format(listprint(imax, ','), listprint(index, ','), listprint(size, ','))
    config.write(s)
    config.close()

```

Then, you can use the static HIRConfig object `cfg` declared in `HIR.h`. Firstly, you have to initialize it by using:

```
//HIRTrain.cpp
//config init
cfg = HIRConfig();
```

## HIRArgument

This module serves as an argument parser. You can use the arguments defined in LibHIR, add your own options, or define your own parameters. The following function parse the argument for sample code `HIRTrain.cpp` and `HIRTest.cpp` respectively.

```
HIRTrainParam parse_train_argument(int argc, char** argv, int task);
HIRTestParam parse_test_argument(int argc, char **argv, int task);
```

The parameter `task` is to distinguish different tasks and indicate corresponding arguments and options. In the sample code, the default arguments are initialized as follows:

```
//HIRArgument.cpp
if (task == 1) { //Regression
    param.obj = RMSE_L;
    param.evl = RMSE_E;
}
else { //Classification
    param.obj = GOLD_L;
    param.evl = ACC_E;
}
```

You can add alternative branches to handle the customized task. Besides, incompatible option and task should be treated carefully as follows:

```
//HIRArgument.cpp
// in function HIRTrainParam parse_train_argument(int argc, char **argv, int task);
case 'e':
    evl = atoi(argv[i]);
    if ((task == 1 && (evl == MAE_E || evl == RMSE_E)) ||
        (task > 1 && (evl == AUC_E || evl == ACC_E)))
        param.evl = evl;
    else (param.errmsg.push_back("Argument Parser--The parameter evaluation method
invalid.));
    break;
```

The task param is related to the chosen task, regression or classification. The `HIRTrainParam` and `HIRTestParam` are structs, therefore you could define your own struct of parameters and declare function with return value, and parse the argument. Take the parameter parsing function of

HIRTrain.cpp as an example.

```
//HIRTrain.cpp
HIRTrainParam para_train;
para_train = parse_train_argument(argc, argv, cfg.task);
int evaluation_type = para_train.evl;
```

## HIRRecord

The `HIRRecord` serves as the data management module. It can load and store the records. You can instantiate the `HIRRecord` function as follows:

```
//HIRTrain.cpp
train = HIRRecord(cfg, 1);
//HIRTest.cpp
test = HIRRecord(cfg, 0);
```

The preprocess script splits the raw records into two files. The second parameter 1 indicates using the front part of data, while 0 indicates the latter part.

Members of the `HIRRecord` class are listed as follows:

```
class HIRRecord {
public:
    HIRRecord();
    ~HIRRecord();
    HIRRecord(HIRConfig cfg, int istrain);
    void predictSave(std::string pfname, HIRConfig cfg);
    //int Evaluate(HIRConfig cfg, HIRModel model, int eva_mtd);
    //int PredictSaving(std::string filename);
    int rec, train;
    std::string filename;
    std::vector<int*> ilist; // index per record
    std::vector<double*> flist; // feature per record
    double** ylist; // target value
    double** hat_y; // predicted value of the model
};
```

If predicted results of a new task do not have the format of float or one-hot vector, you can modify the declaration of `ylist` and `hat_y`. In function `HIRRecord::PredictSaving()`, the `hat_y` can be saved as a float value or one-hot vector for each record. Please rewrite the function to save prediction when developing your own task.

## HIRModel

The module `HIRModel` contains parameters, the gradient buffer and operations of HIR model. The declaration of `HIRModel` is as follows.



```
//HIRModel.cpp
class HIRModel {
public:
    int ModelLoading(HIRConfig cfg);
    int ModelSaving(HIRConfig cfg);
    int GradInit(HIRConfig cfg);
    int GradClear(HIRConfig cfg);
    int GradForward(HIRConfig cfg, int* index_list, double* feature);
    int GradBackward(HIRConfig cfg, HIRTrainParam para, int* index_list);
    int ismat, dim;
    ...
};
```

The value of every component can be fetched after the initialization as follows.

```
//model init
HIR = HIRModel(cfg);
HIR.GradInit(cfg);
```

The parameters of HIR can be modified by using the member function of `HIRModel`. You can replace the matrix `mylist` with other components which are compatible to the defined output.

The default optimizer of LibHIR is Stochastic Gradient Descent. Once different learning methods are defined, the `forward_train` and `backward_train` functions in `HIRTrain.cpp`, and the `GradForward` and `GradBackward` functions in `HIRModel.cpp` should be modified. In the training process, the function `forward_train` should be called firstly, the feedforward process is finished within the HIR model (before the final representation) by calling `HIRModel::GradForward`, and the left part of the task (after the final representation) is implemented in the origin function. Similarly, the task related part in backpropagation should be handled by `backward_train` and then called `HIRModel::GradBackward` to finish the backpropagation of residual and update of HIR parameters. Take the forward propagation as an example:

```
//HIRTrain.cpp
int forward_train(int index, HIRConfig cfg) {
    int* eids = train.ilist.at(index);
    int x, y, z;
    int xx, yy, zz, lh, lr;
    double ztemp, ytemp, * vx, * vy, * vr, *** T, ** M;
    xx = zz = cfg.dim;
    eids[0]; eids[1]; eids[2]; eids[3];

    //representation generating
    if (cfg.fwid == 0)
        HIR.GradForward(cfg, train.ilist.at(index), nullptr);
    else
        HIR.GradForward(cfg, train.ilist.at(index), train.flist.at(index));

    //M forward
    lr = cfg.hnum + cfg.size.size() - 1;
    vx = HIR.rlist.at(lr);
```

```

yy = cfg.task;
for (y = 0; y < yy; y++) {
    vy = HIR.mylist.at(y);
    train.hat_y[index][y] = dotMul(yy, vx, vy);
}
return 0;
}

```

The `HIRModel::ModelLoading` function can load components of HIR model from the file written by the function `outmodel` in `prepro.py`, and `HIRModel::ModelSaving` can write it back from the `HIRModel` object, with the specify parameters of the `HIRConfig`. The `GradClear` deals with the gradients by setting their values to zero. Once auxiliary model structures are added to your application, adding codes respectively in these functions is better than implementing the operation in your main training loop.

## HIRTrain and HIRTest

The complete training process is as follows:

```

//HIRTrain.cpp
int main(int argc, char**argv) {
    //config init
    cfg = HIRConfig();
    //amardown-preview://editor/26rgument init
    para_train = parse_train_argument(argc, argv, cfg.task);
    if (para_train.errmsg.size() != 0) {
        printf("Error occured in parsing arguments.\n");
        return 0;
    }
    //record init
    train = HIRRecord(cfg, 1);
    //model init
    HIR = HIRModel(cfg);
    HIR.GradInit(cfg);
    std::cout << "=== LOSS ===" << std::endl;
    std::vector<int> v;
    for (int i = 0; i < train.rec; ++i) {
        v.push_back(i);
    }
    int seed;
    for (int epoch = 0; epoch < 100; epoch++) {
        // forward
        // obtain a time-based seed:
        seed = std::chrono::system_clock::now().time_since_epoch().count();
        std::shuffle(v.begin(), v.end(), std::default_random_engine(seed));

        for (int index = 0; index < train.rec; index++) {
            HIR.GradClear(cfg);
            forward_train(v.at(index), cfg);
            backward_train(v.at(index), cfg);
        }
        // evaluate
        double evl = evaluate(para_train.evl, train.rec, train.ylist, train.hat_y);
    }
}

```

```

        std::cout << "epoch " << epoch <<" : " << evl << std::endl;
    }
    HIR.ModelSaving(cfg);
    printf("HIR train ended.\nPress Enter to exit.");
    getchar();
    return 0;
}

```

If your plan to predict the test records with the trained model, and save the prediction results, refer to the `HIRTest.cpp`:

```

//HIRTest.cpp
int main(int argc, char**argv) {
    //config init
    cfg = HIRConfig();
    //model init
    HIR = HIRModel(cfg);
    HIR.GradInit(cfg);
    //argument init
    para_test = parse_test_argument(argc, argv, cfg.task);
    if (para_test.errmsg.size() != 0) {
        printf("Error occured in parsing arguments.\n");
        return 0;
    }
    //record init
    test = HIRRecord(cfg, 0);
    // forward
    for (int index = 0; index < test.rec; index++) {
        HIR.GradClear(cfg);
        forward_test(index, cfg);
    }
    // evaluate
    double evl = evaluate(para_test.evl, test.rec, test.ylist, test.hat_y);
    std::cout << "The evaluation result is: " << evl << std::endl;
    test.predictSave(para_test.pfname, cfg);
    printf("HIR test ended.\nPress Enter to exit.");
    getchar();
    return 0;
}

```

## Additional Information

Further information of the HIR model can be found in the paper [Collaborative Prediction for Multi-entity Interaction With Hierarchical Representation](#).

For any questions and comments, please email [daiyaxuan2018@outlook.com](mailto:daiyaxuan2018@outlook.com).