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 useage of LibHIR.
You can get LibHIR at https://github.com/TagineerDai/LibHIR.
Please read the COPYRIGHT before using or modifying LibHIR.

Table of Contents

Quick Start

If you are new to HIR and the task is classification or regression, please try our command-line tool version of LibHIR. We offer a script `prepro.py` to transfer the svm+ data into HIR format data file and initialize the config and model file. Generally, SVM+ data is to concatenate the observed features and contexts with svm format records. For more information, view 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`, which should be copied to the same direct as the program `HIRTrain` and `HIRTest` on Linux, or `HIRTrain.exe` and `HIRTest.exe` on Windows. Run the files with option `-h` and no parameter, to show the usage of them. For further information, view the Command-line Tool Usage section.

Usage: `HIRTrain/HIRTest [option parameter]`

Installation

Linux

On Linux systems, type `make` in the terminal to build the `HIRTrain` and `HIRTest`, add `HIRTrain` and `HIRTest` after the `make` to build the given file.

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

Windows

The executable files `HIRTrain.exe` and `HIRTest.exe` of the given command-line tool are offered. Once you are developing your own program based on LibHIR, please consult `Makefile` to manually build the project for linux platform, or simply use a IDE.

Data Format and Transfer

Under the `data` direction, we offered a sample python code `transformat.py` to build up the svm format data for our sample data movielens. And `prepro.py` under the `script` direction can transfer the data from the svm format into HIR format record, and generate HIR format config file and model files. View the Command-line tool usage for more details.

SVM Format

The HIR model is compatible with the record format of libsvm. The svm format data is the raw input data of the `prepro.py`. Compared with LibSVM or LibFM, our model supports latent vectors of multiple entities, observed featrues and contexts. The auxiliary feature is concatenated after normal svm records and splited by spaces.

The svm data format is shown as follows:

<label> <index$_1$>:<value$_1$> ... <index$_k$>:<value$_k$> <feature$_1$> ... <feature$_k$> . .

Each line is a record instance and is ended by a '\n' character. The target value (The value to be predicted in regression task, or the category label in classification task) is a real value number. In the `<dimension>:<index>` pairs, the entity index are integers, and could be in order or after shuffled. The feature values will be stored as float number. Once there are entities with observed features and contexts of a record, we concatenate them into one feature vector. It will be treated as a special feature vector of arbitrary dimensions, with its values can not be modified by the learning process.

HIR Format

The preprocess script `prepro.py` will load the raw svm+ format file and generate record file, model file, and config file. 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` includes the names of model file, training data and testing data file, the `data specification` section is for the HIR model and records, and the `size of parameter` section includes index and number of each entity. The config format is `[param name] = [param value]`, and line comments in the config are started with a `#`.

It is not suggested that you to modify the parameter in config.txt directly. All the specification of HIR model and HIR record should are generate with the config file by preprocess script. The mismatch of configuration, model and record file will lead to fatal error.

The model.txt contains the features of each entities, the component parameter of the HIR model, such as the tensor layer, the hidden layer, the latent representations, and the target representation. If you are developing your own task with HIR representation, modifing the `prepro.py` and the `HIRModel::ModelLoading` and `HIRModel::ModelSaving` could help.

The record files, namely the `train.txt` and the `text.txt`, are consist with record number, item index list, feature vector list, target value list and predict value list.

User Guide

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

Transformat Script Usage

Once you have python and numpy installed, use the command `python prepro.py [option]` to preprocess the raw svm+ format data. Usage of the options is as follows.

Option	Detail
--rawfile	the svm+ format file name.
--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 layer
--dim	dimensionality of the features, and representations.

The Rtrain is a float in (0,1). And the category, hlayer 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 showed for each epoch, and the model after training will be saved in `model.txt`.

Option	Detail	Value
-r	epoch number	positive integer
-o	objective function	0(ABS), 1(RMSE), 2(LL), 3(GOLD)

Option	Detail	Value
-e	evaluation function	4(ABS), 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 this epoch to any positive integer. The default learning method is Stochastic Gradient Descent. The learning rate (alpha) is a float, which represent the update ratio in the backpropagation process. Both the regression and classification tasks have corresponding objective functions and evaluations, missmatch task and option will cause error. For regression tasks, the optional objective function are the abstract distance(ABS) and the rooted mean squared error(RMSE), and the optional evaluation functions are the mean abstract error(ABS) 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
```

In the HIRTest program, the evaluation function is related to the choice of task. Mismatch of task and evaluation function will cause error. The prediction record will be saved in a given file, `answer.txt` as shown in the sample.

Options	Details	Value
-e	evaluation function	4(ABS), 5(RMSE), 6(AUC), 7(ACC)
-f	predict record file	filename stirng
-h	help	help information

For regression tasks, the predicted value of every record is a double value in a single line, the optional evaluation functions are the mean abstract error(ABS) and the rooted mean squared error(RMSE). For classification tasks, the predicted value of every record is a one-hot vector to distinguish the category, and the optional evaluation functions are the area under the ROC curve(AUC) and the classification accuracy(ACC).

Developer Guide

The network components are declared in the header file `HIR.h`. With the guide of the Makefile, you need to include this file 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 carry out your own task based on LibHIR. The components of LibHIR set fifth below:

HIRConfig

This module loads in and stores the config file in HIR format, and specialized hyperparameters of the model, such as sizes of record numbers. Once some addition settings are to be added into your task, modify the `HIRConfig` class and its construction function, then instantiate it in your code. The compatible setting parameter formats 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);
}
```

The modification of config file could be completed in two ways. The simpler method is to add a record in the `config.txt` directly; while the shortcoming is that it calls for repetition every time after running the preprocess script. Another once for all method 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()
```

After the modification, you could use the static HIRConfig object cfg declared in `HIR.h`. Firstly you have to initialize it by:

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

HIRArgument

This component serves as an argument parser.You could use the arguments defined in LibHIR, add your own options, or define your own parameters.
The following function parse the argument separatly 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 task and indicating corresponding arguments and options. In the sample code, the default argument is 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 could add algernative 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 == ABS_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 regression and classification mission choice. The HIRTrainParam and HIRTestParam in the file are structs, therefore you could define your own parameter struct and declare function with return value of it, and parse the argument of your program. Take the parameter parsing function of `HIRTrain.cpp` as a sample:

```
//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. We could load in and store the records in HIR format with it. You could instantiate the records as follows:

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

The preprocsee script have helped split the raw records into two files as the training to total number ratio. The second parameter 1 indicates using the front part of data, while 0 indicates the latter part.

Members of the HIRRecord class is 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
};
```

Once there are task with predicted value differ from float for regression and one-hot vector for classification, you could modify the declaration of ylist and hat_y. In function `HIRRecord::PredictSaving()`, the hat_y would be saved as a float number or one-hot vector for each record. Rewrite the function to save output once developing your own application.

HIRModel

The module HIRModel contains the parameter and gradient buffer of HIR model, and operations on it. 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;

    //should be made after the cfg and record have done and serve as the HIRModel
class
    std::vector<double*> elist; // param per entity indentity [model.txt]
    std::vector<double*> mylist; // Linear M last-layer [model.txt]
    std::vector<double**> hlist; // param per hidden-layer [model.txt]
    std::vector<double***> Tnet; // Trans with tensor[model.txt]
    std::vector<double*> rlist; //□Tnet+Hmat□□

    //For backpropagation
    double loss; //
    double* dL_y; //                 [cfg.task]
    std::vector<double*> dy_my; //   [cfg.task][dim]
    std::vector<double*>dy_r; //     [cfg.task][dim]
    std::vector<double**> dr_e; //   [cfg.enum] x [dim][dim]
    std::vector<double**> dr_r; //  [size.size() + hnum - 1] x [dim][dim]
    std::vector<double***> dr_T; // [size.size()] x [dim][dim][dim]
    std::vector<double**> dr_H; //  [cfg.hnum] x [dim][dim]

    std::vector<double*> dL_my; //  [cfg.task][dim]
    std::vector<double*>dL_r; //    [size.size() + hnum][dim]
    std::vector<double*> dL_e; //  [cfg.e_num] x [dim]
    std::vector<double***> dL_T; // [size.size()] x [dim][dim][dim]
    std::vector<double**> dL_H; //  [cfg.hnum] x [dim][dim]
    double* feature;
    std::vector<int> size, lidx, ridx;

    HIRModel();
    ~HIRModel();
    HIRModel(HIRConfig cfg);
};
```

When using the HIR model, after the initialize as follows could the value of every component be fetched:

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

The HIR related parameters should not be modified without member function of `HIRModel`, except the `dL_my`, `dy_my`, `dL_y` and `loss`. Feel free to add task related parts to replace the matrix W(`mylist`) between the target representation of HIR, w.r.t. the last item in vector `rlist`, and output of the task target(`hat_y` of `HIRRecord`).

The default optimizer of LibHIR is Stochastic Gradient Descent. Once different learning method is defined, the function `forward_train` and `backward_train` is `HIRTrain.cpp`, and the function `GradForward` and `GradBackward` should be modified. In the training process, the function `forward_train` should be called firstly, with the HIR generating part(part before the target HIR representation) feedforard by calling `HIRModel::GradForward`, and the task related part(part after the HIR representation)) implemented in the origin function. Similarly, the task related part in backpropagation should be handled by `backward_train` and then calling the `HIRModel::GradBackward` to finish the HIR related work. Take the forward propagation as an instance:

```
//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;
}
```

Fucntion `HIRModel::ModelLoading` will load the model component parameters from the file writen by the outmodel function in preprocess, and `HIRModel::ModelSaving` willwrite it back from the HIRModel object, with the specify parameters of the HIRConfig. The `GradClear` deals with the derivation intermediate amount by setting their value to all zeros. Once auxiliary conponents are added in your application, the best practice could be adding in handle codes respectively in these functions, instead of 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();
    //amarkdown-preview://editor/26rgument init
    para_train = parse_train_argument(argc, argv, cfg.task);
    if (para_train.errmsg.size() != 0) {
        printf("Error occured in parsering 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 want to predict with records and trained model, and save the predicted answer of your task, 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 parsering 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 could be found in the paper Collaborative Prediction for Multi-entity Interaction With Hierarchical Representation.

For any questions and comments, please email daiyaxuan2018@outlook.com.