LibEDM
Users’ Manual

Tables and Figures

LibEDM is an open-source library developed in C++ and aims to provide a uniform platform for developing and evaluating ensemble/pruning related algorithms and methods. It can also serve as a toolkit for applying these techniques to real-world problems.

LibEDM is implemented in the context of classification, and it should also be suitable for regression analysis after some modifications.

1. Environment Requirements and file descriptions

LibEDM can work with Visual C++ 2005 (VC2003, VC2008 should also work) for MS-Windows, or GNU and Intel C++ compiler for Linux. It should work on any platform that supports ISO C++, such as UNIX.

Directory	File	Description
/	COPYRIGHT	Copyright declaration for LibEDM
	Classifier.h	Base class of all classifiers.
	DataSet(.h, .cpp)	Base class (Common functions) for instances reading, validating and manipulating.
	Ensemble(.h, .cpp)	Base class for all ensemble methods.
	EnsemblePruner(.h, .cpp)	Base class for all ensemble pruning methods.
	IncrementalClassifier.h	Base class for all classifiers that can learn incrementally.
	IncrementalEnsemble.h	Base class for all ensemble-based incremental classifiers, which create ensemble on incoming data, which is either a data trunk or a data stream.
	IncrementalTrunkEnsemble.h	Base class for all trunk-based incremental ensembles that creates at most a single base classifier on all data of each incoming trunk.
	LibEDM.dsp	Workplace for working with VC++ 6.0.
	LibEDM.vcproj	Solution configuration for working with VC++ 8.0 and above.
	makefile	Make file for making LibEDM in Linux.
	Obj.h	Base class for all classes in LibEDM.
	Prediction(.h, .cpp)	Manipulation of prediction results.
	svm(.h, .cpp)	Source code of Libsvm.

classifiers/	bpnn(.h, .cpp)	Trainer for back propagation neural network.
	b-svm(.h, .cpp)	Trainer for SVM, it's an encapsulation of libsvm.
	C45(.h, .cpp)	Trainer for C4.5, it's a re-implementation of Quilan's C4.5 decision tree.
	GaussNaiveBayes(.h, .cpp)	Trainer for naive Bayes base on Gauss distribution assumption.
	NaiveBayes(.h, .cpp)	Trainer for Naïve Bayes base on continuous attributes discretization.
ensembles/	AdaBoost(.h, .cpp)	AdaBoost: ensemble generating based on Boosting.
	Bagging(.h, .cpp)	Bagging: ensemble generating by bootstrap re-sampling of training data.
	CustomEnsemble(.h, .cpp)	An ensemble method allowing user manipulation for the base classifier.
FileFormats/	ArffData(.h, .cpp)	Reading data set from ARFF format data file (one file each data set).
	UCIData(.h, .cpp)	Reading data sets from UCI format data files (one names file and one data file for each data set).
IncrementalEnsembles/	ACE(.h, .cpp)	ACE ensemble-based incremental learning algorithm.
	AWE(.h, .cpp)	AWE ensemble-based incremental learning algorithm.
	FCAE(.h, .cpp)	FACE ensemble-based incremental learning algorithm.
	SEA(.h, .cpp)	SEA ensemble-based incremental learning algorithm.
	Win (h, cpp)	Sliding window.
pruners/	cluster(.h, .cpp)	Ensemble pruning based on clustering.
	FS(.h, .cpp)	Ensemble pruning based on a greedy strategy (forward selection).
	Gasen(.h, .cpp)	Ensemble pruning based on the Genetic Algorithm.
	MDSQ(.h, .cpp)	Ensemble pruning based on aggregation ordered by vector distances.
	OrientOrder(.h, .cpp)	Ensemble pruning based on aggregation ordered by vector angels.
	PMEP(.h, .cpp)	Ensemble pruning based on pattern mining.
	SelectAll(.h, .cpp)	Ensemble pruning: All base classifiers are selected.
	SelBest(.h, .cpp)	Ensemble pruning: select the best base classifier.
utilities	CrossValidate.h	Encapsulation for cross validation.
	DateTime(.h, .cpp)	A transplantable encapsulation for time and date manipulating.

	Ga(.h, .cpp)	Encapsulation for Genetic Algorithm.
	RandSequence(.h, .cpp)	Transfer an array into random sequence.
	Statistics(.h, .cpp)	Functions for Friedman and Bergmann-Hommel statistical test.
	zString(.h, .cpp)	Routines to manipulating strings.
examples/		Example programs (see example description of each chapter).

Table 1-1 Directory structure of LibEDM

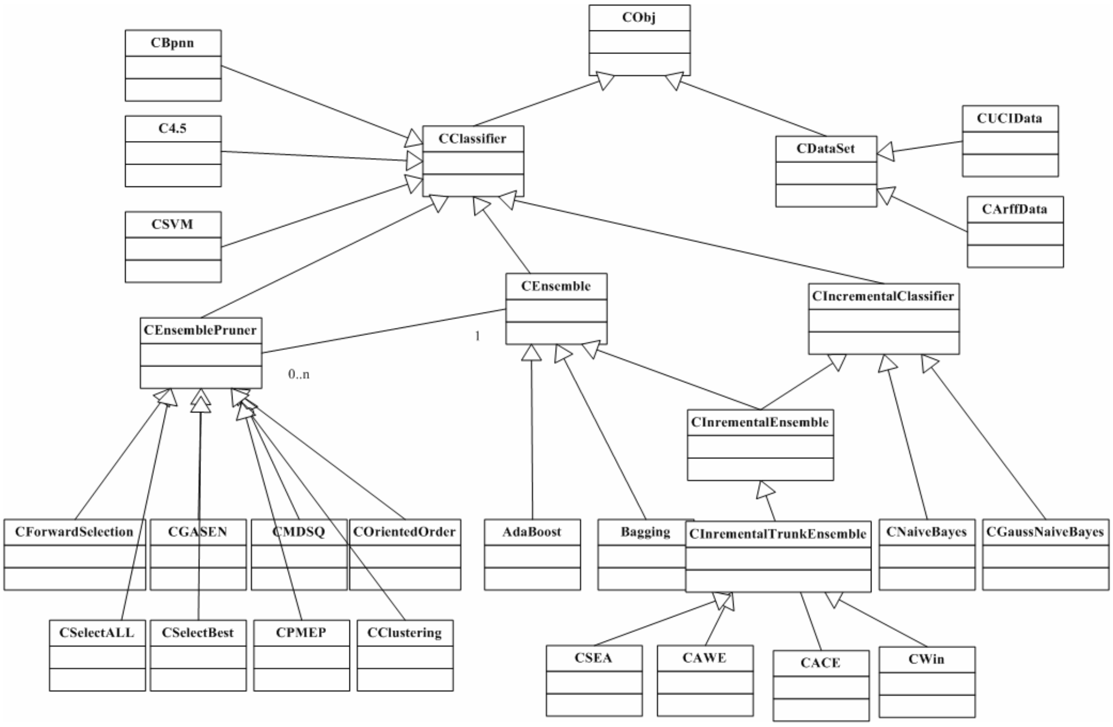


Figure 1-2 Hierarchy for classes of LibEDM

1.1. Compiling and Examples

To compile LibEDM, open *kensm.sln* in Microsoft Visual C++ 2005 and execute the *build* command in the *building* menu for MS-Windows; or execute the *make* command for Linux in the directory of LibEDM. After compiling, a library (named *libEDM.lib* on Windows or *libEDM*.so on Linux) will be created. To use LibEDM, users should add the LibEDM library to their projects and make sure all LibEDM source directories are in their including paths. The header paths for each kind of classes are listed below:

Name	**Description**
/	All base classes of LibEDM.

/classifiers	Classes for base classifiers.
/ensembles	Classes for ensembles
/FileFormats	Classes for supporting different data file formats.
/IncrementalEnsembles	Classes for ensembles that can be trained incrementally.
/pruners	Classes for ensemble pruners.
/utilities	Utility classes for Cross-Validation, statistical tests, Genetic Algorithm, string manipulating and date-time manipulating.

Table 1-3 Header file paths

Examples are included in LibEDM. To compile them, entering *examples* directory and use *make* command for Linux/UNIX (LibEDM must be built before building examples), or opening the workplace *examples.sln* in MS-VC++ for Windows. Detailed description for these examples can be found in corresponding chapters:

Name	**Source file**	**Description**	**Chapter**
dataset	example1.cpp	Reading and extracting from data files.	6.3
classifier	example2.cpp	Creating a BPNN from training data files.	10.5
classifier1	example2.1.cpp	Restoring a BPNN from archives.	10.5
Ensemble	example3.cpp	Creating a Bagging ensemble from training data files, then using it to predict a test data set.	12.4
Pruner	example4.cpp	Creating a Bagging ensemble and pruning it through Selecting-Best method, then using the pruned ensemble to predict.	16.9
Pruner1	example4.1.cpp	Creating a Bagging ensemble and pruning it through Selecting-Best and MDSQ separately. Because both these pruning methods can share all base-classifiers' prediction results for the validation set, so total pruning-time for all these methods can be saved.	16.9
Statistic	example5.cpp	Statistical comparison of multiple classifier methods by Friedman test and Bergmann-Hommel test.	17.7.4
CrossValidate	example6.cpp	Testing RPROP, a BP-neural-network training method, on an ARFF-format data file, by using three-folder cross-validation.	17.4.3
IncEnsemble	example7.cpp	Training an ACE ensemble incrementally and use it to predict.	14.3.3

Table 1-4 Examples

2. Glossary

1. Learner: Machine learning algorithms used to train models (i.e. classifier in LibEDM), such as BPNN (back propagation neural network), SVM, C4.5 decision tree and Naive Bayes, etc.;
2. Incremental classifier: Models that can be trained incrementally, which can adapt to the change of input data.
3. Data (Instances): Formatted data used by learners to create models, usually stored in files;
4. Label: For classification task, each instance belongs to a type. Each type is represented as a label. Label may be unknown for new instances.
5. (Attribute) Value: An instance is represented as a collection of values for all attributes.
6. Base classifier: Models trained on instances, which will be used to compose a ensemble;
7. Ensemble: Combination of more than one models that can work together to predict.
8. Ensemble pruning: Procedures to reduce some base classifiers from ensemble so as to improve the efficiency and performance of ensemble.
9. Validation set: Used by most pruning methods, works as a basis, on which base classifiers are evaluated, compared and selected/pruned.

3. Base class for all: CObj

class CObj

CObj is the base class for all the other classes of LibEDM. It encapsulates common functions that are needed by all the objects in LibEDM.

3.1. Requirements

```
#include "Obj.h"
using namespace LibEDM;
```

3.2. Members

Information	
GetName	Return name for this object.
GetCreatingTime	Return the time for creating the object

Variables	
Name	Name of the object
CreatingTime	Time for creating
Ref	Number of objects that are currently referencing this object

3.2.1. CObj::GetName

Get name of the object.

```
string GetName() const
```

Parameters

Return Value

Name of the object.

Remarks

3.2.2. CObj::GetCreatingTime

Get the time used to create the object.

```
string GetCreatingTime() const
```

Parameters

Return Value

Time creating the object.

Remarks

Creating time is useful when comparing time performance for different algorithms, such as base classifier creating, ensemble creating, ensemble pruning and predicting.

4. Error: CError

class CError

CError is throwed when some kinds of fatal errors happened and must be processed immediately .

4.1. Requirements

```
#include "Obj.h"
using namespace LibEDM;
```

4.2. Members

Information	
CError	Construct a error object.
Variables	
Description	Description for this error.
Code	Code number for this error, each object may have same error number.
Level	Not used.

4.2.1. CError::CError

```
CError(const string &Desc,const int &Code, const int &Level)
```

Parameters

Description

Description for this error.

Code

Code number for this error, each object may have same error number.

Return Value

Name of the object.

Remarks

4.2.2. CError:: Description

Get a creating time of the object.

```
        string   Description;
```

Return Value

Description of this error.

Remarks

5. Instances manipulation: CDataSet

class CDataSet : public CObj

In class CDataSet, common functions for instances inputing and outputing, manipulation and transformation are implemented. After a CDataSet object is constructed, data can then be loaded into the memory, and the data memory will be kept until the object is destructed.

User can build CDataSet objects directly from data files of supported formats (C4.5, CSV and ARFF, etc.), or users can read data from databases or other medias into memory through their own codes then constrcut CDataSet Objects from these data in memory.

Mostly data is stored in files. For example, in C4.5 or CSV format data is stored in two file, i.e. the description file and the real data file, while in the ARFF format both the description and the data is in the same file. In all these formats, controls characters are used to indicates meaning of each word, each field and each line. Users should avoid to use these characters in names or values. In all these formats, each line describe an attribute, an instance or the class labels.

Control Characters	**Description**
:	(C4.5 format) In a description file, as the delimiter of a attribute name and its corresponding description(type and/or values for nominal values)
.	End of line
(CR/LF)	End of line
,	Delimiter of values
(space)	Delimiter of values
(TAB)	Delimiter of values
;	Delimiter of values (Comment a line when it is a leading character)
|	Comment
"…"	Quatoed string
?	A unknow label or value

Table 5-1 Control characters in the data/description file

LibEDM do not process instances with unknown values at present, so these instances will be removed during construction of the data object. But instances with unknown labels are allowed to exist in data object when it is only used for predicting. If a instance with unknow lable are appeared in a training set, it is just skipped.

For each file format, a subclass need to be inherited from CDataSet, in which the function that reading from real data files and/or description files should be implemanted. So if users want to build data object from files, use a proper subclass of CDataSet corresponding to your file formats.

5.1. Requirements

```
#include <string>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <istream>
#include <sstream>
#include <climits>


#include <cstring>
```

```
#include <cassert>
#include <cstdarg>
#include <ctime>
using namespace std;
#include "CObj.h"
#include "DataSet.h"
#include "RandSequence.h"
using namespace LibEDM;
```

5.2. Members

Construction / Destruction / Load	
CDataSet	Construction.
~CDataSet	Destruction.
Load	Read/Load description and data into this data object
LoadInfo	Get only description of data from a file.
Extraction	
GetData	Get a reference to the instances (real data) of this data object.
GetInfo	Get a reference to the description for the data.
GreatestValBelow	Get max value for an attribute, which is less than a specified value.
AllContinuous	Return true if all attributes are continous.
Manipulation	
RemoveNullAttribute	Remove attributes if all their corresponding values are unknown.
RemoveUnknownInstance	Remove instances if some of whose values are unknown.
SwapInstance	Swich positions of two instances
Insert	Insert a data at the end
Remove	Remove an instance at given position
ClearData	Remove all instances
Transformation	
ExpandDiscrete	Return a new data set object, in which each mutil-value discrete attribute is transformed into several continuous (boolean) attributes.
BootStrap	Bootstrap re-sampling
SplitData	Randomly seperating original data set into multiple data sets (No duplicated instances for each set).
SubSet	Get a random subset of the original data set.

DevideBySetNum	Partition original data set into several new data sets from begging to end, by specifying the set number. Each set has the same size.
DevideByDataNum	Partition the data set into several new data sets from begging to end, by specifying size for all the new sets.
Dump	
DumpInfo	Dumping description of data set into a file.
DumpData	Dumping pure data out.

5.2.1. CDataSet::CDataSet

Construct a data object.

```
//create a empty data object (no instance and no description)
CDataset();
//copy construction
CDataset(const CDataset &DataSet);
//construct data object by giving data description and instances
CDataset(const CASE_INFO &Info, const MATRIX &Data)
```

Parameters

DataSet

Original data set from which the discrete attribute will be expanded.

Info

User input data description.

Data

User input instances.

Return Value

Remarks

To create a data object from files or other media, user should create a empty object of a corresponding subclass of CDataSet according to your file format, then call a proper Load function to do the real reading.

5.2.2. CDataSet::Load

Load real data and/or description in to the data object.

```
//load a data set which is stored in two files (e.g. C4.5 format, a names file and a data file).
void Load(const string &InfoFileName,const string &DataFileName);
//load data set from two file (data file is opened)
void Load(const string &InfoFileName,ifstream &DataFile,int Number=0);

//load data set from single file (e.g. ARFF format).
void Load(const string &DataFile);
//load several data from the single file
void Load(const CASE_INFO &uCaseInfo,ifstream &DataFile,int Number=0);

//load data set from an data vector
void Load(const CASE_INFO &uCaseInfo,const vector<StringArray> &Instances);
```

Parameters

InfoFileName

name for data description file

DataFileName

Name for data file

DataFile

Stream of open file, the data will be read from it.

Number

Number of instances will be read from the opened file stream.

uCaseInfo

User input data description.

Instances

Array of strings, each item of the array represents an instance.

Return Value

Remarks

Dataset can be fullly loaded from disk files, either from two files (the description file and the data file) or from one file (the description and the data are in the same file).

Data can also be loaded partialy through reading from an opened file stream, several instances each time, which is useful when only part of the data need/can to be processed at one time. Notice that the description file (if exists) is still needed.

CDataset can also create data object from a 2D string array pre-loaded in memory. This is useful when the data format is different from what LibEDM supports (for exmaple it is from a database). Each row of the string array represents an instance, and fields of the instances must in the order which is described by the data description.

5.2.3. CDataSet::LoadInfo

Load only the description into the data object

```
//Load Information of data from the single file
void LoadInfo(ifstream &DataFile);
```

Parameters

DataFile

Stream of opened file.

Return Value

Remarks

If the data is stored in a single file, after calling this function, the read pointer of this opened file will move to the position where the first instance starts.

5.2.4. CDataSet::~CDataSet

Destroy a data object.

```
~CDataset();
```

Parameters

Return Value

Remarks

5.2.5. CDataSet::GetData

Get a constant reference to the internal data of the object.

```
const MATRIX &GetData() const;
```

Parameters

None

Return Value

A constant reference to the internal data.

Remarks

5.2.6. CDataSet::GetInfo

Get a constant reference to the description of the data object.

```
const CASE_INFO &GetInfo() const;
```

Parameters

Return Value

A constant reference to the description of the data object.

Remarks

5.2.7. CDataSet::GreatestValBelow

Get the maximum value of a attribute which is no greater than a specified value.

```
double GreatestValBelow(const int Att,const double &t) const;
```

Parameters

Att

The # of the attribute.

t

The upper boundary of the return.

Return Value

A value that is no greater than t

Remarks

The retuen value is meaningless when the attribute is non-continuous (but you can still use it).

5.2.8. CDataSet::AllContinuous

Return a boolean value to indicate whether all attributes for this data object are continuous.

```
bool AllContinuous() const;
```

Parameters

Return Value

A boolean value. True if all attributes for this data object are continuous, False otherwise.

Remarks

5.2.9. CDataSet::ExpandDiscrete

Create a new dataset, by expanding each multi-value discrete attribute into multiple boolean attributes.

```
//create a new dataset, by expanding every multi-valued discrete attribute into multi boolean
attributes(needed by BPNN and/or SVM)
CDataset *ExpandDiscrete() const;
```

Parameters

Return Value

A pointer to the new created data set.

Remarks

Some of the learner (SVM or Neural networks) can not work (or work well) on discrete attributes, so by this function we can expanding each mutil-value (more than two values) discrete attribute into several (equal to the number of values for this attribute) boolean attribute..

5.2.10. CDataSet::RemoveNullAttribute

Remove the attributes of which all values are null(unknow).

```
void RemoveNullAttribute();
```

Parameters

Return Value

Remarks

Remove null attributes to improve the efficiency when this data object is to be used as training set. If you use this function to the training set, you must also call the function to each of the prediction set to avoid inconsistency between them.

5.2.11. CDataSet::RemoveUnknownInstance

Remove instances whose label are unknown.

Parameters

Return Value

Remarks

LibEDM can not process instances with unknown values at present, so these instances have been removed during construction of the data object. But instances with unknown labels are allowed to exist when its data object is only used as a prediction data set.

If a data object will be used to create classifiers (a training set), *RemoveUnknownInstance* can be used before training start to improve the efficiency.

5.2.12. CDataSet::SwapInstance

Switch position of two instances in the data object.

```
void SwapInstance(const int a,const int b);
```

Parameters

a, b

Original position of the two instances in the data array.

Return Value

False if any of the input position is invalid; True otherwise.

Remarks

If any of the input position is invalid, nothing will happen.

5.2.13. CDataSet::Insert

Insert the input instace at the end of the data set.

```
//insert instances
void Insert(const InstanceStr &Instance);
//remove
void Remove(int Pos);
void ClearData();
```

Parameters

Instance

Input instance described as array of values.

Return Value

False if any of the input position is invalid; True otherwise.

Remarks

5.2.14. CDataSet::Remove

Remove the instance at specified position.

```
void Remove(int Pos);
```

Parameters

Pos

Position of the instance to be removed.

Return Value

Remarks

5.2.15. CDataSet::ClearData

Remove all instance of this data set.

```
void ClearData();
```

Parameters

Return Value

Remarks

The description is kept except the number of instances is set to zero.

5.2.16. CDataSet::BootStrap

Wighted/non-weighted bootstrap resampling to create new data object. For weighted bootstramp, the size of weights array must match that of the data array.

```
bool BootStrap(const int DataNum,CDataset &TrainSet) const;
bool BootStrap(const vector<double> &Weights,const int DataNum,vector<int> &OrginalPos,
     CDataset &TrainSet) const;
```

Parameters

TrainSet

New data object to place data in, must be empty. If the input data object is not empty, all old data will be erased before new data putted in.

DataNum

Size of new data object to be created.

Weights

Weights for weighted bootstrap.

OriginalPos

For each instance in the new data object, its original position in the original data array.

Return Value

True if successful;

For weighted bootstramp, if the size of weights array doesn't match that of the data array, false is returned.

Remarks

Bootstrap resampling take instances from this data object randomly and copy them to the new data object, instances are allowed to be duplicated. For weighted bootstrap, instances with higher weights have more possibility to be taken. If don't want the instances to be duplicated in new data object, use the function SubSet or SplitData.

5.2.17. CDataSet::SubSet

Randomly take instances of this data object and copy to the new object, each instance once at most.

```
bool SubSet(const int DataNum,CDataset &SubSet) const;
```

Parameters

TrainSet

New data object to place data in, must be empty. If the input data object is not empty, all old data will be erased before new data putted in.

DataNum

Size of new data object to be created.

Return Value

True if successful; False otherwise.

Remarks

There will be no duplicated instance of original data object in the new data object.

5.2.18. CDataSet::SplitData

```
bool SplitData(const int DataNum,CDataset &TrainSet,CDataset &TestSet) const;
```

Randomly divide the data object into two parts, each of which is used to create a new data object. Each instance must only belong to one data object. Instances of original data object is put into *TrainSet* at first, and the rest is put into *TestSet*.

```
bool SplitData(const int DataNum,const int SetNum,vector<CDataset> &TrainSets,CDataset &TestSet) const;
```

Randomly select instances from this data object and put them into *SetNum* data objects, each with size *DataNum*. Each instance can only be selected once and all the new data objects will be saved in TrainSets. The rest instances will put into the data object *TestSet*.

Parameters

TrainSet, TestSet

New data object to put instances in, must be empty. If it isn't empty, all instance in it will be removed.

TrainSets

Array of data objects to be created, must be empty. If it isn't empty, all object in it will be destroyed.

DataNum

Size of the data object *TrainSet*.

Size of each data object in *TrainSets*.

SetNum

Number of data objects will be created in *TrainSets*.

Return Value

False if this data object hasn't enough instances to create all the data object with specified size; True otherwise.

Remarks

Each instance from the original data object will only appear once in all the newly created data objects.

5.2.19. CDataSet::DevideBySetNum

From begging to end, devide the original data set into several new data sets.

```
//data set is divided into some several parts from begging to end.
bool DevideBySetNum(int SetNum,vector<CDataset> &TrainSets) const;
```

Parameters

TrainSets

Array of data objects to be created, must be empty. If it isn't empty, all object in it will be cleared.

SetNum

Number of data objects will be created in *TrainSets*.

Return Value

Remarks

If there are not enough instances for at least one instance in each data object, exception will be thrown out. There is possibility that instances can not be evenly put into each new data objects.

5.2.20. CDataSet::DevideByDataNum

From begging to end, devide the original data set into several new data sets.

```
//data set is divided into some several parts from begging to end.
bool DevideByDataNum(int DataNum,vector<CDataset> &TrainSets) const;
```

Parameters

TrainSets

Array of data objects to be created, must be empty. If it isn't empty, all object in it will be cleared.

DataNum

Max size for each data object in *TrainSets*.

Return Value

Remarks

5.2.21. CDataSet::DumpInfo

Output detailed description of the data object to disk file for examnation.

```
void DumpInfo(const string &FileName) const;
```

Parameters

FileName

File name of the output file.

Return Value

Remarks

5.2.22. CDataSet::DumpData

Output instances of the data object to disk file for examnation.

```
void DumpData(const string &FileName,bool Append=false) const;
```

Parameters

FileName

File name of the output file.

Append

Use append mode file output, this is, start writing from end of the file.

Return Value

Remarks

5.3. Data structures

Description of instances	
CASE_INFO	Description of instances.
AttrStr	Description of an attribute.
DiscValueStr	Description of a value for a discrete attribute.
ClassStr	Description of a class label.
Data of instances	
MATRIX	Array of instances
InstanceStr	A instance.
ValueData	A value for a field

5.3.1. ClassStr

Description of a class label. A class label is often a string, but inside LibEDM it is marked as a number according to the appearance order of it in the data description file.

```
typedef struct ClassStr
{
    string   Name;
}ClassStr;
```

Fields

Name

Name of a class lable.

Remarks

5.3.2. DiscValueStr

Description of a value for a discrete attribute, this value is often a string, but inside LibEDM it is marked as a number according to the appearance order of it in the data description file.

```
typedef struct DiscValueStr
{
    string          Name;
}DiscValueStr;
```

Fields

Name

Name of a value for a discrete attribute.

Remarks

5.3.3. AttrStr

An attribute may be one of the three possible type: continuous, discrete and ignore. For a discrete attribute, all the possible values must be listed in the field *Disc*. For a continuous attribute, some information need to be gathered during construction of the data object.

```
typedef struct AttrStr
{
    int       AttType;
    string    Name;
    double    Max;//maximum value
    double    Min;//minimum value
    bool      MMSet;//Have max and min value been set?
    //corresponding position of a valid attribute:
    //if it a attribute in ValidAttrs, it’s its original position in ReadAttrs
    //if it a attribute in ReadAttrs, it’s its new position in ValidAttrs
    int       OtherPos;

    //discrete attribute: list of all values
    vector<DiscValueStr> Disc;
    .......
```

Fields

AttType

Type of the attribute: IGNORED - 0,DISCRETE – 1, CONTINUOUS -2, CLASSLABEL – 3.

Name

Name of the attribute.

Max

For continuous attribute, maximum value of this attribute.

Min

For continuous attribute, minimum value of this attribute.

MMSet

Are the Max and Min fields valid?

OtherPos

Indicates position of an attribute: if this attribute is an attribute in ValidAttrs,OtherPos is its original position in ReadAttrs; if it is a attribute in ReadAttrs, OtherPos is its new position in ValidAttrs

Disc

List all possible values for a discrete attribute.

Remarks

5.3.4. CASE_INFO

Basiclly the description of a data object is the set of descriptions for all its attributes and class labels.

```
typedef struct CASE_INFO
{
    int ReadWidth;//number of attribute in each row(including label)
    int ValidWidth;//number of real attribute (ignored attributes are excluded)
    int ClassNum;
    int Height;//number of instances
    vector<DiscValueStr> Classes;
    vector<AttrStr> ReadAttrs;//all attributes in a row (including label)
    vector<AttrStr> ValidAttrs;//all attributes in a row (ignored attributes are excluded)
    ......
```

Fields

ReadWidth

Number of all attributes in data description, the class label and ignored attributes are also included.

ValidWidth

Number of all valid attributes, the class label is included but ignored attributes are excluded.

ClassNum

Number of all possible class labels.

Height

Number of instances.

Classes

Descriptions for all class labels.

ReadAttrs

Descriptions for all attributes (including labels), this field is used to read data from files.

ValidAttrs

Descriptions for all attributes (including labels but excluding all ignored attributes), this field is used to access data which has been read into memory.

Remarks

5.3.5. ValueData

Used to store a field value for a discrete attribute(integer) or a continuous attribute (float number).

```
typedef union ValueData
{
    int      Discr;
    float    Cont;
}ValueData;
```

Fields

Discr

Value for a discrete field

Cont

Value for a continuous field

Remarks

5.3.6. InstanceStr

All values for an instance, notice that the class label is stored at the last position of this structure as a discrete attribute.

```
typedef vector<ValueData> InstanceStr;
```

Fields

Remarks

A intances is a array of all its field values (also include its class label).

5.3.7. MATRIX

All data of a data object.

```
typedef vector<InstanceStr> MATRIX;
```

Fields

Remarks

The data of a data object is a array of all its instances.

6. Data Formats

In the C4.5 format data is stored in two files, one is the data description file and the other is data file. The functions reading from C4.5 format files are encapsulated in CUCIData (because it is usually used by the UCI repositary). In the C4.5 format data is stored in two files, one is the data description file and the other is data file. In the ARFF format data is stored in a single file, the data and its description is in the same file.

A description file has two parts, i.e. the description of class labels and the descriptions of attributes. All class labels must be listed at the first valid (non-commented) line of the file. Then descriptions of all attributes are followed, each attribute a line, where the discrete attributes are described by listing all its values and the continuous attributes are simply described as "*continuous*".

Attribute Type	**Description**	**Note**
CONTINUOUS	Attribute has a continous value	
DISCRETE	Valus of attribute will be enumrated in the same line	
IGNORE	Attribute should be ignored	

Table 6-1 Supported attributes

In the data file each instance is putted in a line, where values of all the attributes are listed as the order that they are appeared in the description file. And labels of the training instances should be put at the last field of each line. Any unknown/missed value (or label) should be represented by an interrogation mark (?), and leaving it with a blank is not allowed.

6.1. CUCIData

```
class CUCIData: public CDataset
```

Support functions for reading from files of C4.5 format:

C4.5 format. http://www.cs.washington.edu/dm/vfml/appendixes/c45.htm

6.1.1. Requirements

```
#include <string>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <istream>
#include <sstream>
#include <ctime>
using namespace std;
#include "Obj.h"
#include "zString.h"
#include "DateTime.h"
#include "RandSequence.h"
#include "DataSet.h"
#include "UCIData.h"
using namespace LibEDM;
```

6.1.2. Members

6.1.2.1. CUCIData:: CUCIData

Create an empty C4.5 format data set object.

```
CUCIData(){};
```

Parameters

Return Value

Remarks

Call this function to create a empty data set object then call *Load()* of its super-class to do the real reading.

6.2. CArffData

```
class CArffData: public CDataset
```

Support functions for reading from files of ARFF format:

Attribute-Relation File Format (ARFF). http://www.cs.waikato.ac.nz/ml/weka/arff.html

ARFF Type	corresponding Type in LibEDM	Note
Numeric	CONTINUOUS	
Real	CONTINUOUS	
Date	CONTINUOUS	Only support time after 12:00 am, 1970
Nominal	DISCRETE	
String	IGNORE	Ignored in LibEDM

Table 6-2 Attribute Types for ARFF format

6.2.1. Requirements

```
#include <string>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <istream>
#include <sstream>
using namespace std;
#include "Obj.h"
#include "zString.h"
#include "DateTime.h"
#include "RandSequence.h"
#include "DataSet.h"
#include "UCIData.h"
using namespace LibEDM;
```

6.2.2. Members

6.2.2.1. CArffData::CArffData

Create an empty ARFF format data set object.

```
CArffData(){};
```

Parameters

Return Value

Remarks

Call this function to create a empty data set object then call *Load()* of its super-class to do the real reading.

6.3. Example for Data Set (Example1)

This example shows how to create a data object (CUCIData) from data files, how to extract data and information from it and some operations to manipulate the data.

6.3.1. How to compile?

- Windows: Open examples/dataset.vcproj in MSVC 2005 or higher version
- Linux: type “make dataset” in the examples directory of LibEDM

6.3.2. Description

Let’s take a sample data set (which is modified from *sick* of UCI repository) to describe class *CDataSet*. The data description file looks like below(sample.names):

```
sick, negative.                |   classes


age:                        continuous.

sex:                        M, F.

on thyroxine:                    f, t.

query on thyroxine:              f, t.

on antithyroid medication:       f, t.

sick:                       f, t.
```

```
pregnant:                   f, t.
thyroid surgery:            f, t.
I131 treatment:                 f, t.
query hypothyroid:              f, t.
query hyperthyroid:             f, t.
lithium:                    f, t.
goitre:                     f, t.
tumor:                      f, t.
hypopituitary:                  f, t.
psych:                      f, t.
TSH measured:                   ignore.
TSH:                        continuous.
T3 measured:                    ignore.
T3:                         continuous.
TT4 measured:                   ignore.
TT4:                        continuous.
T4U measured:                   ignore.
T4U:                        continuous.
FTI measured:                   ignore.
FTI:                        continuous.
TBG measured:                   ignore.
TBG:                        ignore.
referral source:            ignore.
```

There are 29 discrete (described by listing all possible values) or continuous (just described as "continuous") attributes and two class labels in this data set.

Below is the data file (sample.data):

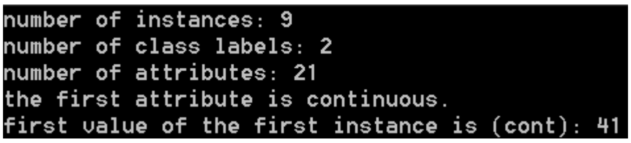
```
41,F,f,f,f,f,f,f,f,f,f,f,f,f,f,f,t,1.3,t,2.5,t,125,t,1.14,t,109,f,?,SVHC,negative.|3733
23,F,f,f,f,f,f,f,f,f,f,f,f,f,f,f,t,4.1,t,2,t,102,f,?,f,?,f,?,other,negative.|1442
46,M,f,f,f,f,f,f,f,f,f,f,f,f,f,f,t,0.98,f,?,t,109,t,0.91,t,120,f,?,other,negative.|2965
70,F,t,f,f,f,f,f,f,f,f,f,f,f,f,f,t,0.16,t,1.9,t,175,f,?,f,?,f,?,other,negative.|806
70,F,f,f,f,f,f,f,f,f,f,f,f,f,f,f,t,0.72,t,1.2,t,61,t,0.87,t,70,f,?,SVI,?.|2807
18,F,t,f,f,f,f,f,f,f,f,f,f,f,f,f,t,0.03,f,?,t,183,t,1.3,t,141,f,?,other,negative.|3434
59,F,f,f,f,f,f,f,f,f,f,f,f,f,f,f,f,?,f,?,t,72,t,0.92,t,78,f,?,other,negative.|1595
80,F,f,f,f,f,f,f,f,f,f,f,f,f,f,f,t,2.2,t,0.6,t,80,t,0.7,t,115,f,?,SVI,sick.|1367
```

```
66,F,f,f,f,f,f,f,f,f,f,f,f,t,f,f,t,0.6,t,2.2,t,123,t,0.93,t,132,f,?,SVI,negative.|1787
68,M,f,f,f,f,f,f,f,f,f,f,f,f,f,f,t,2.4,t,1.6,t,83,t,0.89,t,93,f,?,SVI,negative.|2534
84,F,f,f,f,f,f,f,f,f,f,f,f,t,f,f,t,1.1,t,2.2,t,115,t,0.95,t,121,f,?,SVI,negative.|1485
67,F,t,f,f,f,f,f,f,f,f,f,f,f,f,f,t,0.03,f,?,t,152,t,0.99,t,153,f,?,other,negative.|3448
71,F,f,f,f,t,f,f,f,f,t,f,f,f,f,f,t,0.03,t,3.8,t,171,t,1.13,t,151,f,?,other,negative.|1027
59,F,f,f,f,f,f,f,f,f,f,f,f,f,f,f,t,2.8,t,1.7,t,97,t,0.91,t,107,f,?,SVI,negative.|3331
28,M,f,f,f,f,f,f,f,f,f,f,f,f,f,f,t,3.3,t,1.8,t,109,t,0.91,t,119,f,?,SVHC,negative.|2043
```

There are 15 instances (each a line) in this data file, also notice that each unknown value is marked as a “?”. Next is the code to read and manipulate the sample data set:

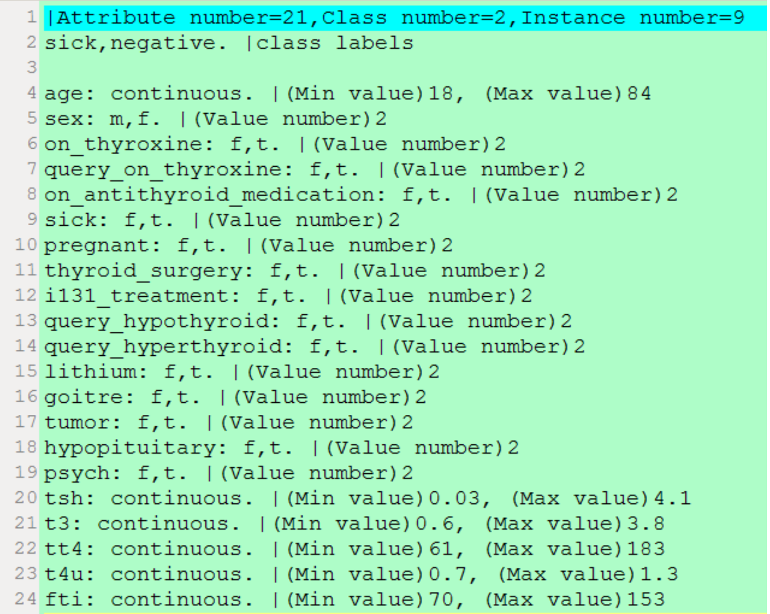
```
//creating dataset object by reading from a description file and a data file
CUCIData ds;
ds.Load("example.names","example.data");
ds.DumpInfo("Info.txt");
ds.DumpData("Data.txt");
//get a reference to the instances in the data set
MATRIX Matrix=ds.GetData();
// get a reference to the data description of the data set
CASE_INFO CaseInfo=ds.GetInfo();
cout<<"number of instances: "<<CaseInfo.Height<<endl;
cout<<"number of class labels: "<<CaseInfo.ClassNum<<endl;
cout<<"number of attributes: "<<CaseInfo.Width<<endl;
//extract type of the 1st attribute
int AttType=CaseInfo.Attrs[0].AttType;
//and first value of first instance
if(AttType==ATT_DISCRETE)
{
        cout<<"the first attribute is discrete."<<endl;
        cout<<"first value of first instance is (Discrete): "<<Matrix[0][0].Discr<<endl;
}
else if(AttType==ATT_CONTINUOUS)
{
        cout<<"the first attribute is continuous."<<endl;
        cout<<"first value of the first instance is (cont): "<<Matrix[0][0].Cont<<endl;
}

//remove all single-valued attributes
ds.RemoveNullAttribute();
ds.DumpInfo("Info1.txt");
ds.DumpData("Data1.txt");
//remove the instances whose label are unknown
ds.RemoveUnknownInstance();
ds.DumpInfo("Info2.txt");
ds.DumpData("Data2.txt");

//create a new data set by extending all discrete attributes of the original data set
CDataset *Extds=ds.ExpandDiscrete();
Extds->DumpInfo("Info3.txt");
Extds->DumpData("Data3.txt");
delete Extds;
```

The output of this program looks like:



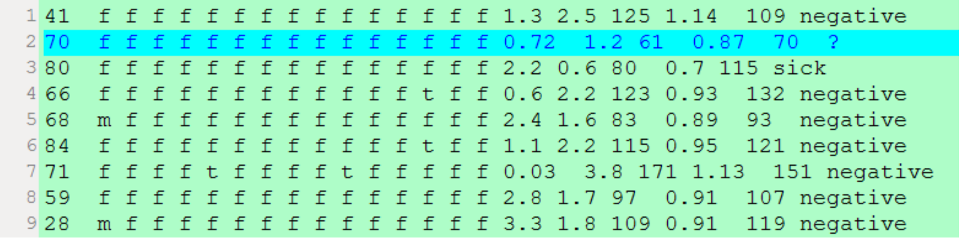
```
number of instances: 9
number of class labels: 2
number of attributes: 21
the first attribute is continuous.
first value of the first instance is (cont): 41
```

After loading from files the dumped description file (info.txt) looks like:



```
|Attribute number=21,Class number=2,Instance number=9
sick,negative. |class labels

age: continuous. |(Min value)18, (Max value)84
sex: m,f. |(Value number)2
on_thyroxine: f,t. |(Value number)2
query_on_thyroxine: f,t. |(Value number)2
on_antithyroid_medication: f,t. |(Value number)2
sick: f,t. |(Value number)2
pregnant: f,t. |(Value number)2
thyroid_surgery: f,t. |(Value number)2
i131_treatment: f,t. |(Value number)2
query_hypothyroid: f,t. |(Value number)2
query_hyperthyroid: f,t. |(Value number)2
lithium: f,t. |(Value number)2
goitre: f,t. |(Value number)2
tumor: f,t. |(Value number)2
hypopituitary: f,t. |(Value number)2
psych: f,t. |(Value number)2
tsh: continuous. |(Min value)0.03, (Max value)4.1
t3: continuous. |(Min value)0.6, (Max value)3.8
tt4: continuous. |(Min value)61, (Max value)183
t4u: continuous. |(Min value)0.7, (Max value)1.3
fti: continuous. |(Min value)70, (Max value)153
```

And the dumped data file (data.txt) looks like:

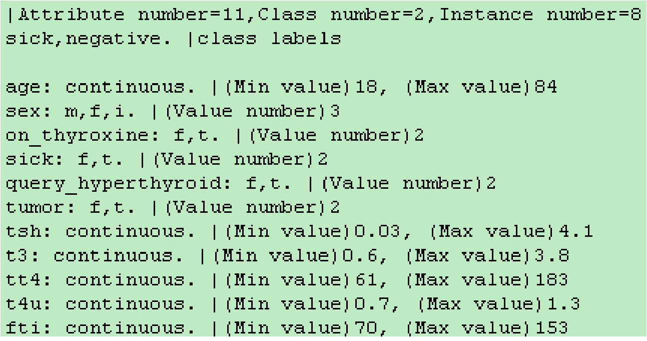


```
41  f f f f f f f f f f f f f f f 1.3 2.5 125 1.14  109 negative
70  f f f f f f f f f f f f f f f 0.72  1.2 61  0.87  70  ?
80  f f f f f f f f f f f f f f f 2.2 0.6 80  0.7 115 sick
66  f f f f f f f f f f f f t f f 0.6 2.2 123 0.93  132 negative
68  m f f f f f f f f f f f f f f 2.4 1.6 83  0.89  93  negative
84  f f f f f f f f f f f f t f f 1.1 2.2 115 0.95  121 negative
71  f f f f t f f f f t f f f f f 0.03  3.8 171 1.13  151 negative
59  f f f f f f f f f f f f f f f 2.8 1.7 97  0.91  107 negative
28  m f f f f f f f f f f f f f f 3.3 1.8 109 0.91  119 negative
```

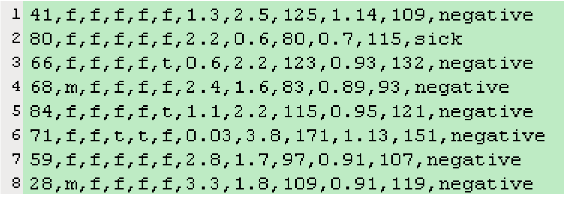
We can find that all the attributes described as “ignored” have been removed. Also in the dumping data file (data.txt) all values of the ignored attributes have been removed. And in the next step, all instances with unknown values have been removed, too. After all those changes there are only 21 attributes and 9 instances left in the data set, which is reflected in the dumped data description file (info.txt).

After being constructed, the data set can be refined by removing attributes that have only one value in all instances (call RemoveNullAttribute(), this function will change the structure of the data set, so make sure ALL the instances have been loaded). And because instances with unknown labels will be ignored in a training process, they'd better been removed from the training set to improve the efficiency(call RemoveUnknownInstance()).

In this example, the 3th,4th,5th,7th,8th,9th,10th,12th,13th,15th and 16th attributes have only one value and they are removed. And the 2nd instance has an unknown label, so they are removed from the original data set. All these changes are shown in the dump files below (info1.txt, data1.txt):



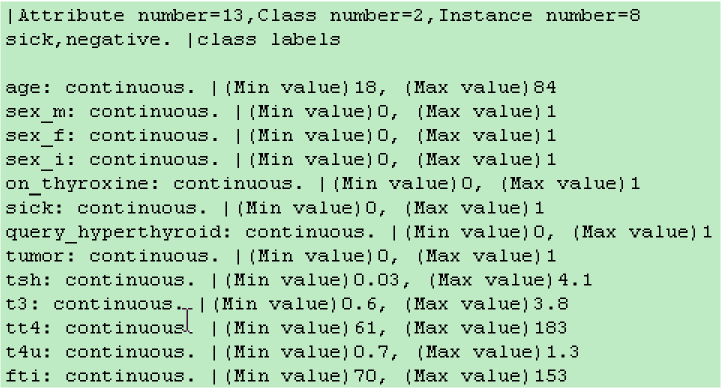
```
|Attribute number=11,Class number=2,Instance number=8
sick,negative. |class labels

age: continuous. |(Min value)18, (Max value)84
sex: m,f,i. |(Value number)3
on_thyroxine: f,t. |(Value number)2
sick: f,t. |(Value number)2
query_hyperthyroid: f,t. |(Value number)2
tumor: f,t. |(Value number)2
tsh: continuous. |(Min value)0.03, (Max value)4.1
t3: continuous. |(Min value)0.6, (Max value)3.8
tt4: continuous. |(Min value)61, (Max value)183
t4u: continuous. |(Min value)0.7, (Max value)1.3
fti: continuous. |(Min value)70, (Max value)153
```



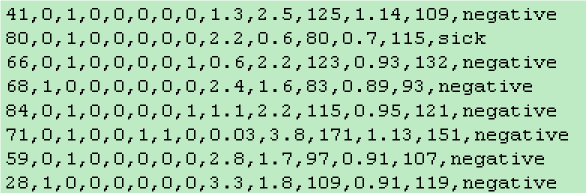
```
41,f,f,f,f,f,1.3,2.5,125,1.14,109,negative
80,f,f,f,f,f,2.2,0.6,80,0.7,115,sick
66,f,f,f,f,t,0.6,2.2,123,0.93,132,negative
68,m,f,f,f,f,2.4,1.6,83,0.89,93,negative
84,f,f,f,f,t,1.1,2.2,115,0.95,121,negative
71,f,f,t,t,f,0.03,3.8,171,1.13,151,negative
59,f,f,f,f,f,2.8,1.7,97,0.91,107,negative
28,m,f,f,f,f,3.3,1.8,109,0.91,119,negative
```

There are totally 11 attributes and 8 instances left.

LibEDM also provides a constructor to manipulate discrete attributes which can not be directly used by trainer of neural network and SVM. It transforms a Boolean attribute into a continuous attribute with value range of [0, 1] and also transforms a multiple-value discrete attribute into several continuous attributes with value range of [0, 1]. This function is shown in the dump files below (info2.txt, data2.txt):



```
|Attribute number=13,Class number=2,Instance number=8
sick,negative. |class labels

age: continuous. |(Min value)18, (Max value)84
sex_m: continuous. |(Min value)0, (Max value)1
sex_f: continuous. |(Min value)0, (Max value)1
sex_i: continuous. |(Min value)0, (Max value)1
on_thyroxine: continuous. |(Min value)0, (Max value)1
sick: continuous. |(Min value)0, (Max value)1
query_hyperthyroid: continuous. |(Min value)0, (Max value)1
tumor: continuous. |(Min value)0, (Max value)1
tsh: continuous. |(Min value)0.03, (Max value)4.1
t3: continuous. |(Min value)0.6, (Max value)3.8
tt4: continuous. |(Min value)61, (Max value)183
t4u: continuous. |(Min value)0.7, (Max value)1.3
fti: continuous. |(Min value)70, (Max value)153
```



```
41,0,1,0,0,0,0,0,1.3,2.5,125,1.14,109,negative
80,0,1,0,0,0,0,0,2.2,0.6,80,0.7,115,sick
66,0,1,0,0,0,0,1,0.6,2.2,123,0.93,132,negative
68,1,0,0,0,0,0,0,2.4,1.6,83,0.89,93,negative
84,0,1,0,0,0,0,1,1.1,2.2,115,0.95,121,negative
71,0,1,0,0,1,1,0,0.03,3.8,171,1.13,151,negative
59,0,1,0,0,0,0,0,2.8,1.7,97,0.91,107,negative
28,1,0,0,0,0,0,0,3.3,1.8,109,0.91,119,negative
```

We can find that the second attribute (sex) of the original data set has been extended to three new Boolean attribute, they are sex_m, sex_f and sex_i, and the values for these new attributes have also been created according to the values of the original attribute.

7. Prediction result: CPrediction

```
class CPrediction: public CObj
```

Result of prediciting made by a classifier to a data set.

7.1. Requirements

```
#include <string>
#include <fstream>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Prediction.h"
using namespace LibEDM;
```

7.2. Members

Construction / destruction	
CPrediction	Construct a prediction object.
~CPrediction	Destruct the prediction.
Information retrieve	
GetProbs	Get the probabilities that each instance belongs to each class label.
GetPredictedLabelIndices	Get predicted label indices for all instances.
GetCorrectness	Get correctness of all prediciton.

GetAccuracy	Get accuracies of predictions.
GetClassNum	Get number of possible class labels for instances.
GetCaseNum	Get number of instances be predicted.
Data members	
ClassNum	Number of possible class labels for instances.
CaseNum	Number of instances be predicted
Probs	Array of probabilities that each instance belongs to each class label.
Accuracy	Accuracies of predictions.
PredClass	Predicted labels for all instances.
IsCorrect	Correctness of all prediciton

7.2.1. CPrediction::CPrediction

Construction.

```
CPrediction(const CDataset &Dataset,const DoubleArray2d &Probabilities,clock_t PredictTime);
```

Parameters

Dataset

The data object being predicted.

Probabilities

The probabilities of each instance belonging to each class label, which are calculated by the classifier.

PredictTime

Time used by the prediciton.

Return Value

Remarks

7.2.2. CPrediction::GetProbs

Get the probabilities that each instance belongs to each class label.

```
const DoubleArray2d& GetProbs() const;
```

Parameters

Return Value

A 2D array of double values.

Remarks

7.2.3. CDataSet::GetPredictedLabelIndices

Get predicted label indices for all instances.

```
const IntArray &GetPredictedLabelIndices() const;
```

Parameters

Return Value

A array of integer values indicate the predicited labels of all instances.

Remarks

The index of first class label is zero, that of the second class label is one, etc. The order of all labels is as their order appeared in the data file.

7.2.4. CDataSet::GetCorrectness

Get correctness of each prediction.

```
const BoolArray& GetCorrectness() const;
```

Parameters

Return Value

A array of boolean values indicate whether each prediction is correct.

Remarks

The actual labels for all instances must be provided in the data object, if they are unknown or absent the return value is meaningless.

7.2.5. CDataSet::GetAccuracy

Get total prediction accuracy for the input data object.

```
double GetAccuracy() const;
```

Parameters

Return Value

A double values indicate total prediction accuracy.

Remarks

The actual labels for all instances must be provided in the data object, if they are unknown or absent the return value is meaningless.

7.2.6. CDataSet::GetClassNum

Get total number of class labels.

```
int  GetClassNum();
```

Parameters

Return Value

A integer value indicate the number of all possible values for instances.

Remarks

7.2.7. CDataSet::GetCaseNum

Get total number of instances being predicted.

```
int  GetCaseNum()
```

Parameters

Return Value

A integer value indicate the number of instances involved in this prediction.

Remarks

8. Base class for all classifiers: CClassifier

```
    class CClassifier : virtual public CObj
```

CClassifier is the base class for all classifier classes, including base classifiers, ensembles and pruned ensembles. CClassifier is a abstract class, can not be instanced directly. If you want to use it, derive a new class from it.

8.1. Requirements

```
#include "Obj.h"

#include "DataSet.h"

#include "Prediction.h"

#include "Classifier.h"

using namespace LibEDM;
```

8.2. Members

Dump	
Save	Save a classifier to disk file.
Dump	Dump all inside data of classifier to disk file for inspecting.
Predict	
Classify	Use this classifier to predicte a data set.
Clone	
Clone	Clone a new classifier from this classifier

8.2.1. CClassifier::Save, CClassifier::Dump

Overrided super class functions, saving or dumping inside data of this BPNN to disk file.

```
virtual int Save(const string &Path,const string &FileName) const =0;

virtual bool Dump(const string &FileName) const =0;
```

Parameters

Path

Location of the output file.

FileName

Name of the output file.

Return Value

True/none zero upon succeed; 0/false othersize.

Remarks

A saved file only contains data useful for restoring the object and may have non-printable characters. In a dumped file all charactars are readable and may contain extra words helping understanding.

8.2.2. CClassifier::Classify

Do pridicting.

```
virtual CPrediction *Classify(const CDataset &DataSet) const =0;
```

Parameters

DataSet

A data object to be predicted.

Return Value

A prediction result.

Remarks

Calling this function you get a CPrediction object (pridiction result), you must destroy it when you no longer need it.

8.2.3. CClassifier::Clone

Clone a new classifier.

```
virtual CClassifier* Clone() const =0;
```

Parameters

Return Value

A pointer to the new classifier copied from this classifier.

Remarks

9. Base class for incremental classifiers: CIncrementalClassifier

```
class CIncrementalClassifier : public CClassifier
```

CIncrementalClassifier is the base class for all classifiers that can be trained incrementally, including incremental base classifiers, incremental ensembles. An incremental base classifier can change its internal structure to reflect the concept change of the input training instances, while an incremental ensemble usually change itself through changing, removing or creating new base classifers, so as to adapt to the change of training instances.

CIncrementalClassifier is a abstract classes, can not be instanced directly. If user want to use it, derive new classes from it.

9.1. Requirements

```
#include "Obj.h"

#include "Classifier.h"

#include "IncrementalClassifier.h"

using namespace LibEDM;
```

9.2. Members

Training	
Train	Use new instances to train this classifier.
Reset	Reset this classifier to un-trained state.

9.2.1. CIncrementalClassifier::Train

Use new instances to train this classifier, so it can adapt to the change of data.

```
virtual void Train(const CDataset &Dataset)=0;
```

Parameters

Dataset

New instances.

Return Value

Remarks

A incremental classifier can be trained continuously as the coming of new instances.

9.2.2. CIncrementalClassifier::Reset

Reset the classifier to the initialize state before its first training.

```
virtual void Reset()=0;
```

Parameters

Return Value

Remarks

Some incremental classifier need this function when performance of this classifier becomes so low that it lose the value of being reused(retrained).

10.Base classifiers

At present LibEDM offers five base classifier trainers, i.e. BPNN (two training algorithms: training with the momentum algorithm and training with the RPROP algorithm, class *CBpnn*), C4.5 decision tree (class *CC45*), SVM (support vector machine, class *CSVM*), Naive Bayes based on continuous attribute discretization (CNaiveBayes, class) and Naive Bayes based on assumption of Gauss distribution (CGaussNaiveBayes, class). Use of CSVM requires the support of *Libsvm*, while other trainers can be used independently. The adjustable parameters for each trainer are listed in the table below.

Trainers	Parameters	Descriptions	Default values
BPNN	HideNode	Nodes of hiden layer	0 (Same as input layer)
	MaxEpoch	Maximum training epochs before training stop.	5000 (Momentum) 3000 (Rprop)
	MinMSE	Minimum MSE before training stop.	0.015
	Alpha	Learing rate (for momentum training)	0.9
	Beta	Momentum variable (for momentum training)	0.5
C4.5	MINOBJS	Minimum instances for each Node after a cut.	2
	Epsilon	Minimum entropy gains.	1e-3
	CF	Upper limit of the confidence level in branch pruning	0.25
SVM	struct svm_parameter		Refer to the documents of Libsvm.
Naïve Bayes	SPLITNUM	For discretizing continuous attributes: Initial number of regions that the value range of a continuous attribute is devided into.	10

Table 10-1 Parameters for base classifier trainers

10.1. Back Propagation Neural Network (BPNN)

```
class CBpnn : public CClassifier
```

Here is the implementation of back propagation neural network. Two training algorithms have been presented. One is the momentum algorithm, the other is a fast algorithm called resilient propagation (Rprop). Both of the two training methods are encapsulated in a C++ class, and put in one source file- Bpnn.cpp

Hecht-Nielsen, R., "Theory of the backpropagation neural network," Neural Networks, 1989. IJCNN., International Joint Conference on , vol., no., pp.593,605 vol.1, 0-0 1989

RPROP: A Fast Adaptive Learning Algorithm. International Symposium on Computer and Information Science VII. pp. 279 - 286, Antalya, Turkey, 1992

10.1.1. Requirements

```
#include <cmath>

#include <iostream>

#include <fstream>

using namespace std;

#include "CObj.h"

#include "Classifier.h"

#include "DataSet.h"

#include "Prediction.h"

#include "bpnn.h"

#include "DateTime.h"

using namespace LibEDM;
```

10.1.2. Members

Construction / Destruction	
CBpnn	Construction.
~ CBpnn	Destroy
Initialize and create	
Create	Static function, return a BPNN created by momemtum method using the default parameters.
RpropCreate	Static function, return a BPNN created by Rprop method using the default parameters.
FileCreate	Create a BPNN using previously saved file data.
Overriding	
Save	Save all inside data to disk file.
Dump	Dump all inside data to disk file for inspecting.
Classify	Use this classifier to predicte a data set.
Information	
GetStaticName	Get internal name for this type of classifier.

10.1.2.1. CBpnn::CBpnn

LibEDM provides two methods to create BPNN from training data, Rprop is faster while momentum method is mostly used by researchers.LibEDM also provide method to save and restore BPNN to/from disk files.

```
//constructing from data set by momentum method
 CBpnn(const CDataset &TrainData,double Alpha=0.9,double Beta=0.5,double MinMSE=0.015,int MaxEpoch=5000,int HideNode=0);
//constructing from data set by RPROP method
 CBpnn(const CDataset &TrainData,double MinMSE=0.015,int MaxEpoch=3000,int HideNode=0);
//restoring from file
CBpnn(const string &Path,const string &FileName);
```

Parameters

TrainData

A data object used to train the BPNN.

alpha

Learning rate.

Beta

Momentum varible.

MinMSE

Minimum MSE(Mean Square Error) before training stop.

MaxEpoch

Max training times before training stop.

HideNode

Node number for hidden layer, if it is zero number of hidden nodes it set as that of input layer.

Path

Location for the previously saved file.

FileName

Name for the previously saved file.

Return Value

Remarks

10.1.2.2. CBpnn::Create, CBpnn::RpropCreate

Use given parameters to create a BPNN by momentum method.

```
static CClassifier *Create(const CDataset &TrainData,const void* Params)
```

Use given parameters to create a BPNN by Rprop method.

```
static CClassifier *RpropCreate(const CDataset &TrainData,const void* RpropParams)
```

Parameters

TrainData

A data object for training.

Params

A point to a CBpnn::Params structure.

RpropParams

A point to a CBpnn::RpropParams structure.

Return Value

A BPNN used as a CClassifier object.

Remarks

When creating an ensemble automaticaly, these functions are used to create BPNNs (if has set for the ensemble) by given parameters.

10.1.2.3. CBpnn::FileCreate

Restore a BPNN from pre-saved file.

```
static CClassifier *FileCreate(const string &Path,const string &FileName)
```

Parameters

See the description of CBpnn::CBpnn().

Return Value

A BPNN used as a CClassifier object.

Remarks

This function is called during the restoring of a file-saved ensemble. A construction will call this function for each base classfier of an ensemble to restore the ensemble from archive files.

10.1.2.4. CBpnn::Save, CBpnn::Dump

Overrided super class functions, saving or dumping inside data of this BPNN to disk file.

```
virtual bool Dump(const string &FileName) const;
virtual int Save(const string &Path,const string &FileName) const;
```

Parameters

Path

Location of the output file.

FileName

Name of the output file.

Return Value

True/none zero upon succeed; 0/false othersize.

Remarks

A saved file only contains data useful for restoring the object and may have non-printable characters. In a dumped file all charactars are readable and may contain extra words helping understanding.

10.1.2.5. CBpnn::Classify

Overrided super class functions, do pridicting.

```
virtual CPrediction *Classify(const CDataset &DataSet) const;
```

Parameters

DataSet

A data object to be predicted.

Return Value

A prediction result.

Remarks

10.1.2.6. CBpnn::GetStaticName

Get internal name for this type of classifier.

```
static string GetStaticName();
```

Parameters

Return Value

The internal name for this type of classifier.

Remarks

This is a static function, can get internal name for all classifiers of this type without creating such a object.

10.1.3. Data Structures

Parameters for creating	
ParamStr	Parameters for creating bpnn by momentum method.
RpropParamStr	Parameters for creating bpnn by Rprop method.

10.1.3.1. CBpnn::ParamStr

```
typedef struct ParamStr
{
    double Alpha;
    double Beta;
    double MinMSE;
    int MaxEpoch;
    int HideNode;
}ParamStr;
```

Fields

See the description of the construct function.

10.1.3.2. CBpnn::RpropParamStr

```
typedef struct ParamStr
{
    double MinMSE;
    int MaxEpoch;
    int HideNode;
}ParamStr;
```

Fields

See the description of construct function.

10.2. C4.5 decision tree (C4.5)

```
class CC45 : public CClassifier
```

Here is the implementation of the C4.5 decision tree. LibEDM's implementation for C4.5 is almost the same as the original one, except that it doesn't support processing of instances with missed field values, that is, all instances with missed values need to be removed from the training data set or it will just been ignored by the training algorithm.

C4.5: programs for machine learning, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1993

10.2.1. Requirements

```
#include <cmath>
#include <cstring>
#include <string>
#include <algorithm>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Prediction.h"
#include "C45.h"
using namespace LibEDM;
```

10.2.2. Members

Construction / Destruction	
CC45	Construction.
~ CC45	Destroy
Initialize and create	

Create	Static function, return a BPNN created by momemtum method using the default parameters.
FileCreate	Create a BPNN using previously saved file data.
Overriding	
Save	Save all inside data to disk file.
Dump	Dump all inside data to disk file for inspecting.
Classify	Use this classifier to predicte a data set.
Information	
GetStaticName	Get internal name for this type of classifier.

10.2.2.1. CC45::CC45

Create a C4.5 decision tree.

```
//constructing from data set
CC45(const CDataset &TrainSet,int MINOBJS=2,double Epsilon=1e-3,double CF=0.25);
//restoring from file
CC45(const string &Path,const string &FileName);
```

Parameters

TrainData

A data object used to train the BPNN.

MINOBJS

Minimum instances in a node after a cut, that is, if instances in a node will be too few after a cut, the cut should not be executed.

Epsilon

Minimum entropy gains for a possible cut.

CF

Upper limit of the confidence level in branch pruning.

Path

Location for the previously saved file.

FileName

Name for the previously saved file.

Return Value

Remarks

10.2.2.2. CC45::Create

Use default parameters to create a C4.5 decision tree.

```
static CClassifier *Create(const CDataset &TrainData,const void *Params)
```

Parameters

TrainData

A data object for training.

Params

A point to a CC45::Params structure.

Return Value

A CC45 object used as a CClassifier object.

Remarks

When creating an ensemble automaticaly from a data object, these functions are used to create C4.5 trees (if has set for the ensemble) by given parameters.

10.2.2.3. CC45::FileCreate

Restore a C4.5 tree from pre-saved file.

```
static CClassifier *FileCreate(const string &Path,const string &FileName)
```

Parameters

See the description of CC45::CC45().

Return Value

A CC45 object used as a CClassifier object.

Remarks

This function is called during the restoring of a file-saved ensemble. A construction will call this function for each base classfier of an ensemble to restore the ensemble from archive files.

10.2.2.4. CC45::GetStaticName

Get internal name for this type of classifier.

```
static string GetStaticName();
```

Parameters

Return Value

The internal name for this type of classifier.

Remarks

This is a static function, can get internal name for all classifiers of this type without creating such a object.

10.2.3. Data Structures

Parameters for creating	
ParamStr	Parameters for creating bpnn by momentum method.

10.2.3.1. CC45::ParamStr

```
typedef struct ParamStr
{
    int       MINOBJS;//minimum instances in a node
    double    Epsilon;//minimum entropy gain
    double    CF;//upper limit of confidence level
}ParamStr;
```

Fields

See descriptions of the construct function.

10.3. Supported vector machine (SVM)

```
class CSVM : public CClassifier
```

LibEDM's implementation of SVM is only the C++ encapsulation of LibSvm. Detailed description of SVM need to reference Libsvm's manual.

Cortes, Corinna; and Vapnik, Vladimir N.; "Support-Vector Networks", Machine Learning, 20, 1995.

10.3.1. Requirements

```
#include <cmath>
#include <cstring>
#include <iostream>
#include <fstream>
using namespace std;
#include "Obj.h"
#include "Classifier.h"
#include "DataSet.h"
```

```
#include "Prediction.h"
#include "svm.h"
#include "b-svm.h"
using namespace LibEDM;
```

10.3.2. Members

Construction / Destruction	
CSVM	Construction.
~ CSVM	Destroy
Initialize and create	
Create	Static function, return a BPNN created by momemtum method using the default parameters.
FileCreate	Create a BPNN using previously saved file data.
Overriding	
Save	Save all inside data to disk file.
Dump	Dump all inside data to disk file for inspecting.
Classify	Use this classifier to predicte a data set.
Information	
GetStaticName	Get internal name for this type of classifier.

10.3.2.1. CSVM::CSVM

Create a SVM.

```
//constructing from data set
CSVM(const CDataset &TrainData,
     int svm_type=C_SVC,
     int kernel_type=RBF,
     int degree=3,
     double coef0=0,
     double cache_size=100,
     double eps=0.001,
     double C=1,
     int nr_weight=0,
     int *weight_label=NULL,
     double *weight=NULL,
     double nu=0.5,
     double p=0.1,
     int shrinking=1,
     int probability=0
     );
 //restoring from file
 CSVM(const string &Path,const string &FileName);
```

Parameters

TrainData

A data object used to train the BPNN.

Path

Location for the previously saved file.

FileName

Name for the previously saved file.

svm_type

C_SVC:	C-SVM classification
NU_SVC:	nu-SVM classification
ONE_CLASS:	one-class-SVM
EPSILON_SVR:	epsilon-SVM regression
NU_SVR:	nu-SVM regression

kernel_type: kernel function type

LINEAR:	u’*v
POLY:	(gamma*u’*v + coef0)^degree
RBF:	exp(-gamma*|u-v|^2)
SIGMOID:	tanh(gamma*u’*v + coef0)
PRECOMPUTED:	kernel values in training_set_file.

degree

degree in kernel function (for poly).

gamma

gamma in kernel function (for poly/rbf/sigmoid).

coef0

coef0 in kernel function (for poly/sigmoid).

cachesize

cache memory size in MB.

eps

tolerance of termination criterion.

C

the parameter C of C-SVC, epsilon-SVR, and nu-SVR (the cost of constraints violation).

nr_weight, weight, weight_label

nr_weight, weight_label, and weight are used to change the penalty for some classes (If the weight for a class is not changed, it is set to 1). This is useful for training classifier using unbalanced input data or with asymmetric misclassification cost.

nr_weight is the number of elements in the array weight_label and weight. Each weight[i] corresponds to weight_label[i], meaning thats the penalty of class weight_label[i] is scaled by a factor of weight[i].

nu

the parameter nu of nu-SVC, one-class SVM, and nu-SVR.

p

the parameter p of EPSILON_SVR.

shrinking

whether to use the shrinking heuristics, 0 or 1.

probability

whether to train a SVC or SVR model for probability estimates, 0 or 1.

Return Value

Remarks

For detailde description of these parameters, see Libsvm's documents.

10.3.2.2. CSVM::Create

Use default parameters to create a SVM.

```
static CClassifier *Create(const CDataset &TrainData,const void *Params)
```

Parameters

TrainData

A data object for training.

Params

A point to a CSVM::Params structure.

Return Value

A CSVM object used as a CClassifier object.

Remarks

When creating an ensemble automaticaly from a data object, these functions are used to create SVMs (if has set for the ensemble) by given parameters.

10.3.2.3. CSVM::FileCreate

Restore a SVM from pre-saved file.

```
static CClassifier *FileCreate(const string &Path,const string &FileName)
```

Parameters

See the description of CSVM::CSVM().

Return Value

A CSVM object used as a CClassifier object.

Remarks

This function is called during the restoring of a file-saved ensemble. A construction will call this function for each base classfier of an ensemble to restore the ensemble from archive files.

10.3.2.4. CSVM::GetStaticName

Get internal name for this type of classifier.

```
static string GetStaticName();
```

Parameters

Return Value

The internal name for this type of classifier.

Remarks

This is a static function, can get internal name for all classifiers of this type without creating such a object.

10.3.3. Data Structures

Parameters for creating	
ParamStr	Parameters for creating bpnn by momentum method.

10.3.3.1. CSVM::SVMParamStr

```
typedef struct svm_parameter SVMParamStr;
```

Fields

See descriptions of the construct function.

Remarks

SVMParamStr in only the alias of the struct svm_parameter in LibSVM.

10.4. Naive Bayes

```
class CNaiveBayes : public CIncrementalClassifier
class CGaussNaiveBayes : public CIncrementalClassifier
```

A naive Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem with strong (naive) independence assumptions.

Two training algorithms based on two assumptions on distributions of continuous attributes (event models) are presented. The first one (CNaiveBayes) use binning to discretize values for a continuous attribute. The second method (CGaussNaiveBayes) assume that values associated with each class are distributed according to a Gaussian (normal) distribution.

These two classes are almost identical except their implementation detail. Class CGaussNaiveBayes has no adjustable paramater.

Both these two classifiers support incremental training.

hang, Harry. "The Optimality of Naive Bayes". FLAIRS2004 conference.

10.4.1. Requirements

```
#include <set>
#include <map>
#include <cmath>
#include <fstream>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Prediction.h"
#include "NaiveBayes.h" (or "GaussNaiveBayes.h")
#include "Statistic.h"
using namespace LibEDM;
```

10.4.2. Members

Construction / Destruction	
CNaiveBayes	Construction.
~ CNaiveBayes	Destroy
Initialize and create	
Create	Static function, return a naive Bayes training by the default parameters.
FileCreate	Create by using previously saved file data.
Overriding from CClassifier	
Save	Save all inside data to disk file.
Dump	Dump all inside data to disk file for inspecting.
Classify	Use this classifier to predicte a data set.
Overriding from CIncrementalClassifier	
Train	Incremental learning.
Reset	Reset the internal data to the initial (un-trained) state.
Information	
GetStaticName	Get internal name for this type of classifier.

10.4.2.1. CNaiveBayes::CNaiveBayes

Create a naive Bayes classifier.

```
//Constructing from data set
//Construction by continuous discretization:
CNaiveBayes::CNaiveBayes(const CDataset &TrainData, int SplitNum=10);
//Construction by Gauss distribution
CGaussNaiveBayes::CGaussNaiveBayes(const CDataset &TrainData);

//restoring from file
CNaiveBayes(const string &Path, const string &FileName);
```

Parameters

TrainData

A data object used to train the BPNN.

Path

Location for the previously saved file.

FileName

Name for the previously saved file.

SplitNum

The number of binning for discretizing a continuous attribute.

Return Value

Remarks

For detailde description of these parameters, see Libsvm’s documents.

10.4.2.2. CNaiveBayes::Create

Use default parameters to create a naive Bayes.

```
static CClassifier *Create(const CDataset &TrainData,const void *Params)
```

Parameters

TrainData

A data object for training.

Params

A point to a CNaiveBayes::Params structure.

Return Value

A CNaiveBayes/CGaussNaiveBayes object used as a CClassifier object.

Remarks

When creating an ensemble automaticaly from a data object, these functions are used to create naive Bayes (if has set for the ensemble) by given parameters.

10.4.2.3. CNaiveBayes::FileCreate

Restore a naive Bayes from pre-saved file.

```
static CClassifier *FileCreate(const string &Path,const string &FileName)
```

Parameters

See the description of CNaiveBayes::CNaiveBayes ().

Return Value

A CNaiveBayes object used as a CClassifier object.

Remarks

This function is called during the restoring of a file-saved ensemble. A construction of the ensemble will call this function for each base classfier of an ensemble to restore the ensemble from archive files.

10.4.2.4. CNaiveBayes::GetStaticName

Get internal name for this type of classifier.

```
static string GetStaticName();
```

Parameters

Return Value

The internal name for this type of classifier.

Remarks

This is a static function, can get internal name for all classifiers of this type without creating such a object.

10.4.3. Data Structures

Parameters for creating	
ParamStr	Parameters for creating Naive Bayes by continuous attribute discretization.

10.4.3.1. CNaiveBayes::ParamStr

```
typedef struct ParamStr
{
    int SplitNum;
}ParamStr;
```

Fields

See descriptions of the construct function.

10.5. Examples for Base Classifiers and Prediction Result (example2, example2.1)

This example shows how to create a base classifier from a training data set, how to use the base classifier to predict and how to process the prediction result information.

10.5.1. Source Files

- Example2.cpp: Create a classifier from a data set.
- Example2.1.cpp: Restore a classifier from achive files (need to run Example2 at first to create the archives).

10.5.2. How to compile?

- Windows: Open examples/Classifier.vcproj or examples/Classifier1.vcproj in MSVC 2005 or higher version
- Linux: type “make classifier” or “make classifier1” in the examples directory of LibEDM

10.5.3. Description

There are two methods to Create a base classifier.

The first is to create(train) from a data set:

```
CDataset TrainSet("zoo.names","zoo.data");
//remove the instances whose label are unknown
TrainSet.RemoveUnknownInstance();


CBpnn BP1(TrainSet,0.01,3000,0);
//dump to check
BP1.Dump("BP1.dmp");
//save this BP neural network into a file under the current directory
BP1.Save("","BP1.sav");
```

First create a training data object (CDataSet), remember to remove the instances with unknown labels, then create base classifieris using any of the base learners (BPNN, C4.5, SVM, Naive Bayes) with appropriate parameters. At last one optionally saves the classifiers in achive files for future use.

The second method to create a base classifier object is to restore it from achive files:

```
//Achive files are in the current directory
CBpnn BP2("","BP1.sav");
cout<<"Type of the classifier: "<<BP2.GetName()<<endl;
cout<<"Time creating the classifier: "<<BP2.GetCreateTime()<<endl;
//dump to check
BP2.Dump("BP2.dmp");
```

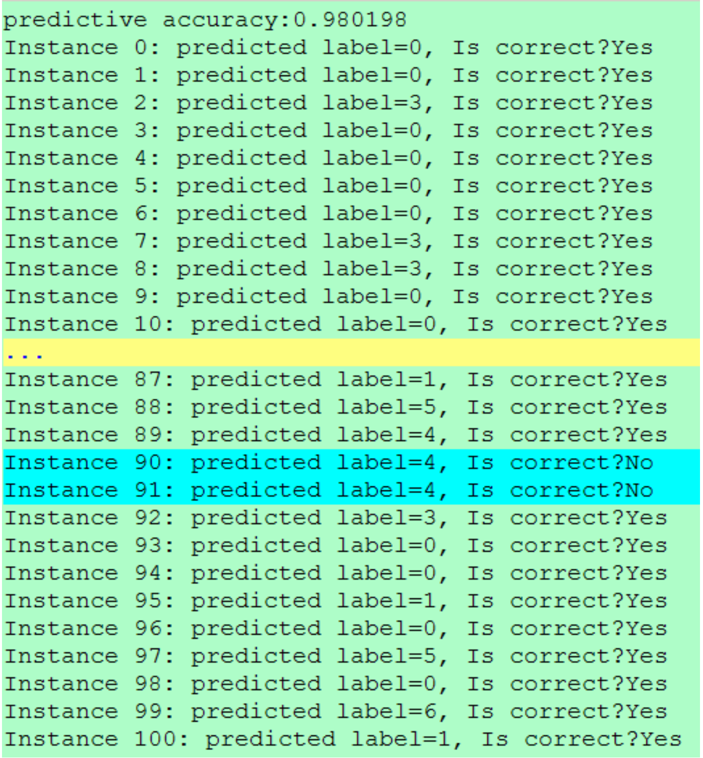
After being created, a base classifier can then be used to predict a data set (a CDataSet object):

```
//use the BPNN to predict the original dataset
const CPrediction *Result=BP1.Classify(TrainSet);
//predicting accuracy
```

```
cout<<"predictive accuracy:"<<Result->GetAccuracy()<<endl;
//extract information of the prediction
const vector<int> &Predicted=Result->GetPredictedLabelIndices();
const vector<bool> &Correct=Result->GetCorrectness();
for(int i=0;i<(int)Predicted.size();i++)
{
    cout<<"Instance "<<i<<": predicted label="<<Predicted[i]<<
        ", Is correct?"<<(Correct[i]?"Yes":"No")<<endl;
}
delete Result;
```

To do a prediction, call the base classifier object's Classify() method. And a CPrediction object will be created inside the classify function, it should be destroied after use. To see the information of a prediciton, calling GetAccuracy to get the total accuracy of the prediction if the lables of instances are listed in the data set, calling GetPredictedLabelIndices and GetCorrectness to know each instance's predicted label and if this prediction for this instance is correct correspondingly.

The output of this example looks like the following:



```
predictive accuracy:0.980198
Instance 0: predicted label=0, Is correct?Yes
Instance 1: predicted label=0, Is correct?Yes
Instance 2: predicted label=3, Is correct?Yes
Instance 3: predicted label=0, Is correct?Yes
Instance 4: predicted label=0, Is correct?Yes
Instance 5: predicted label=0, Is correct?Yes
Instance 6: predicted label=0, Is correct?Yes
Instance 7: predicted label=3, Is correct?Yes
Instance 8: predicted label=3, Is correct?Yes
Instance 9: predicted label=0, Is correct?Yes
Instance 10: predicted label=0, Is correct?Yes
...
Instance 87: predicted label=1, Is correct?Yes
Instance 88: predicted label=5, Is correct?Yes
Instance 89: predicted label=4, Is correct?Yes
Instance 90: predicted label=4, Is correct?No
Instance 91: predicted label=4, Is correct?No
Instance 92: predicted label=3, Is correct?Yes
Instance 93: predicted label=0, Is correct?Yes
Instance 94: predicted label=0, Is correct?Yes
Instance 95: predicted label=1, Is correct?Yes
Instance 96: predicted label=0, Is correct?Yes
Instance 97: predicted label=5, Is correct?Yes
Instance 98: predicted label=0, Is correct?Yes
Instance 99: predicted label=6, Is correct?Yes
Instance 100: predicted label=1, Is correct?Yes
```

The total prediction accuracy is 0.98 and only two (No 90 and 91) of 100 instances are wrongly predicted by the BPNN we created.

11. Base class of ensembles: CEnsemble

```
class CEnsemble : public CClassifier
```

An emsemble is a collection of classifiers which may include all kind of classifiers such as base classifiers or even other ensembles. All these classifiers work together as if they are a single classifier. The common way that many classifiers work together is voting or weighted voting.

CEnsemble is the base class for ensemble methods in LibEDM, any type of ensemble whatever its internel struction must inherit from CEnsemble.

All base classifiers are managed by this ensemble, i.e. when the ensemble is desstructed all its base classifiers will be destructed, too.

Opitz, D.; Maclin, R. (1999). "Popular ensemble methods: An empirical study". Journal of Artificial Intelligence Research 11: 169–198.

11.1. Requirements

```
#include <cstring>
#include <vector>
#include <fstream>
using namespace std;
#include "Obj.h"
#include "Classifier.h"
#include "zString.h"
#include "DataSet.h"
#include "Prediction.h"
#include "Ensemble.h"
using namespace LibEDM;
```

11.2. Members

Building	
Register	Registering the entries of create function for all types of base classifiers that compose the ensemble.

Getting Information	
GetSize	Size of this ensemble, including all base classifiers managed by this ensemble.
GetRealSize	Size of this ensemble, only including base classifiers whose weights are greater than zero.
GetAllClassifiers	Retrive all classifiers of an ensemble.
GetWeights	Weights for all classifiers of an ensemble, useful when each classifier is created with different weight (weighted voting).
Predict	
Classify	Use this ensemble to predicte a data set.
AllClassify	Each classifier of this ensemble predicts a data set, return the collection of all the prediction results.
Manipulation	
Flush	Remove all classifiers of the ensemble.
Overriding	
Save	Save all inside data to disk file.
Dump	Dump all inside data to disk file for inspecting.
Classify	Use this classifier to predicte a data set.
Clone	Create a new ensemble by copying this ensemble

11.2.1. CEnsemble::Register

Registering entrys of file-creating functions for all types of base classifiers, which is used in an ensemble, which will be called when each base classifier is to be restored from files. For all ensembles, each base classifier only need to be registered once.

```
    template <class T> static int Register()
```

Parameters

T

A class of base classifier that support restoring from files.

Return Value

Remarks

Before an ensemble is restored from disk files, the creating-from-file function must be registered for each type of base classifiers used in the ensemble.

11.2.2. CEnsemble::GetSize

Get number of all base classifiers managed by an ensemble.

```
int GetSize()const
```

Parameters

Return Value

Size of the ensemble.

Remarks

11.2.3. CEnsemble::GetRealSize

Get number of base classifiers in an ensemble, excluing classifiers weight zero.

```
int GetRealSize()const
```

Parameters

Return Value

Remarks

11.2.4. CEnsemble::GetAllClassifiers

Get all the classifiere belonging to this ensemble.

```
const vector<const CClassifier*> &GetAllClassifiers()const
```

Parameters

Return Value

Pointers to all the classifiere of this ensemble in an array.

Remarks

Using this function if want to inspect base classifiers one by one. Don't try to destroy a classifier if it is member of an ensemble, the ensemble is in charge of deconstruction of all its base classifiers.

11.2.5. CEnsemble::GetWeights

Get weights for all classifiers in an ensemble.

```
const vector<double> &GetWeights()const
```

Parameters

Return Value

Weights for all classifiers in this ensemble in an array.

Remarks

Some ensemble creating method creates base classifiers with different weights, user can inspect these weights by calling this function.

11.2.6. CEnsemble::AllClassify

Each classifier in an ensemble predicts the input data set and each return a individual prediction.

```
vector<CPrediction*> *AllClassify(const CDataset &DataSet) const;
```

Parameters

DataSet

The data object to be predicted.

Return Value

All the predictions in an array.

Remarks

This function is often used by ensemble pruners, they inspect each classifier's prediction (not all classifiers as a whole) to evaluate the performance and perform selecting.

11.2.7. CEnsemble::Classify

Classify a data set under different condition.

```
//Predict a data set
virtual CPrediction *Classify(const CDataset &DataSet) const;
//vote predicting, if we have prediction of each classifiers
CPrediction *Classify(const CDataset &DataSet,const vector<CPrediction*> &Predictions) const;
//vote predicting, using user-defined weights vector
CPrediction *Classify(const CDataset &DataSet,const vector<double> &UserWeights) const;
//vote predicting, using user-defined weights vector
CPrediction *Classify(const CDataset &DataSet,const vector<CPrediction*> &Predictions,const
    vector<double> &UserWeights) const;
```

Parameters

DataSet

The data set to be predict by the ensemble.

Predictions

Pre-obtained predictions of all classifiers of the ensemble to the input data set.

UserWeights

User defined weights for all classifiers in the ensemble. Weighted voting by ignoring the (if exists) internal weights of the ensemble.

Return Value

Predicition of the ensmeble to the input data set.

Remarks

For a input data set to be predicted, each classifier of the ensemble has to predict the data set individually at first, then all ther prediction will be combined by a method (commonly voting), that may be a long process. This is just the first form of classify() does.

But in some stuation before ensemble predicting, individual prediction of each classifier may already available (for exmaple, the process of ensemble pruning need the prediciton of individual classifier, then for the pruned ensemble all the prediciton of base classifiers are available), the second and the fourth forms of classify() can be used to save time.

Also in some situation user want to set weights to each classifier of the ensemble by himself, while ignoring the original weights (for a bagging ensemble, all the weights are identical; for booosting they may be different), then the third and the fourth form of classify() can be used.

11.2.8. CEnsemble::Flush

Remove all classifiers in the Ensemble.

```
bool Flush();
```

Parameters

Return Value

Remarks

All classifiers in the ensemble are destroyed and all related data structures are cleared.

11.3. Data Structures

Creating Parameters	
CreatorRegisterStr	Information of the creating function for a type of classifiers .
FileCreatorRegisterStr	Information of the creating-by-files function for a type of classifiers .
Weights and Classifiers	
Weights	Weights for all base classifiers in this ensemble.
Classifiers	Pointers to all base classifiers.

11.3.1. CreatorRegisterStr & FileCreatorRegisterStr

Before creating ensembles, the creating function for a base classifiers must register in the ensemble, as well as its creating parameters and ratio of this type of classifiers to the size of ensembble.

All these information is saved in the data strutures list below.

```
typedef struct CreatorRegisterStr
{
    double     Ratio;// percent of this classifier in ensemble
    CreateFunc *Creator;//entry of construction function building classifier from training data
    const void *Params;//Create parameters for the construction
}CreatorRegisterStr;

typedef struct FileCreatorRegisterStr
{
    string            ClassifierType;//type name of the classifier
    FileCreateFunc    *Creator;//entry of the function restoring classifier from files
}FileCreatorRegisterStr;
```

This data struture is used to save information got from the rigister funtions:

```
    template <class T> static int Register()
```

Fields

Ratio

The percentage of this type of classifier (or same type of classifier with different creating parameters) in the ensemble.

Creator

Creating function of the classifer being registered.

Params

Create parameters for the creating function.

ClassifierType

Name for a type of classifier. Used to match creating function for file saved classifiers.

FileCreator

Creating function for restoring a type of classifers from archive files.

Remarks

11.3.2. Weights & Classifiers

```
    vector<CClassifier*> Classifiers;
    vector<double> Weights;
```

Fields

Remarks

These two members can be accessed by all sub-classes of CEnsemble, so all the sub-classes need only create all its base classifiers and put the pointers of them into *Classifiers* (the same to Weights if applicable). And the ensemble will manage the use and destroy of all base classifiers.

12. Ensembles

LibEDM presents two ensemble creating methods: Bagging and Boosting (AdaBoost). For Bagging Bootstrap re-sampling is preformed to build many new data sets from one original data set then many classifiers are created from these new data sets. For boost each training instance is weighted according to its difficulty in predicting, and weighted re-sampling is performed to create new data sets as well as new classifiers. For both of these methods all created classifiers are used to build the ensemble, but in Bagging all classifiers are with same importance, while in Boosting the classifiers may have different weights.

12.1. Bagging

```
class CBagging : public CEnsemble
```

Use bootstrap re-sampling to creating training sets for classifiers.

L. Breiman. Bagging Predictors. Machine Learning, 24(2):123–140, 1996.

12.1.1. Requirements

```
#include <vector>
#include <ctime>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "Bagging.h"
using namespace LibEDM;
```

12.1.2. Members

Construction / Destruction

CBagging	Construction.

12.1.2.1. CBagging::CBagging

Create an ensemble by Bagging algorithm.

```
CBagging(const CDataset &TrainSet,
    double DataPerc,int TotalModel,const vector<CreatorRegisterStr> &Creators);
```

Parameters

TrainData

A data object used to train the Ensemble.

DataPerc

Ratio of size of the new data set used to size of the original data set, should be within (0,1].

TotalModel

Size of the ensemble to be created, an intenger number > 0.

Creators

Entrys and parameters for creating base classifiers, and ratios for each type of base classifiers in the ensemble as well.

Return Value

An Bagging ensemble.

Remarks

Same type of base classifiers (such as BP neural network, SVM and C45, etc.) with different creating parameters can be regarded as diferent types.

12.2. AdaBoost

```
class CAdaBoost : public CEnsemble
```

Use weighted re-sampling to creating training sets for each classifier of the ensemble, each instance is weighted according to its difficulty when being predicted, and each classifier is weighted according to their prediction accuracy.

D. D. Margineantu and T. G. Dietterich. Pruning adaptive boosting. In ICML-97, pages 211–218, 1997.

12.2.1. Requirements

```
#include <vector>
#include <ctime>
#include <cmath>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "Prediction.h"
#include "AdaBoost.h"
using namespace LibEDM;
```

12.2.2. Members

Construction / Destruction	
CAdaBoost	Construction.

12.2.2.1. CAdaBoost::CAdaBoost

Create an ensemble by AdaBoost algorithm.

```
CAdaBoost(const CDataset &TrainSet,
     double DataPerc,int TotalModel,CreateFunc *Creator,const void *Params)
```

Parameters

TrainData

A data object used to train the Ensemble.

DataPerc

Ratio of size of the new data set used to size of the original data set, should be within (0,1].

TotalModel

Size of the ensemble to be created, an intenger number > 0.

Creator, Params

Entry and parameters for creating basec classifiers.

Return Value

An AdaBoost ensemble.

Remarks

12.3. User customizable ensemble

```
class CCustomEnsemble : public CEnsemble
```

Classifiers are not created by the ensemble automatically. User can create base classifier outside by himself and add the classifiers into the ensemble or remove classifier from the ensemble.

12.3.1. Requirements

```
#include <vector>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "CustomEnsemble.h"
using namespace LibEDM;
```

12.3.2. Members

Construction / Destruction	
CCustomEnsemble	Construction a empty ensemble.
Base Classifiers Manipulating	
Add	Add a classifiers into the ensemble
Remove	Remove a classifiers from the ensemble

Notice: Once a classifer is added into an ensemble it is fully managed by the ensemble, user should not destroy the classifier from outside the ensemble. The classifier will be destroyed automatically as the deconstruction of the ensemble or through calling reomve function by user. One classifier can not be added into en emsemble more than once and can not be added into more than one ensemble.

12.3.2.1. CCustomEnsemble::CCustomEnsemble

Create an empty ensemble.

```
CCustomEnsemble();
```

Parameters

Return Value

Remarks

12.3.3. CCustomEnsemble::Add

Add a user created base classifier into the ensemble.

```
bool Add(const CClassifier *Classifier,double Weight=1.0);
```

Parameters

Classifier

The classifier to be added.

Weight

The corresponding weight of the classfier.

Return Value

True if succeed, false otherwise.

Remarks

12.3.4. CCustomEnsemble::Remove

Remove (and destroy) a classifier at the given position from the ensemble.

```
bool Remove(int ID);
```

Parameters

ID

Position of the classifier to be removed.

Return Value

True if succeed, false otherwise.

Remarks

12.4. Example for Ensemble (Example3)

This example shows how to register base classifier learners to an ensemble builder and how to create and use ensemble to predict.

12.4.1. How to compile?

- Windows: Open examples/ensemble.vcproj in MSVC 2005 or higher version
- Linux: type “make ensemble” in the examples directory of LibEDM

12.4.2. Description

To create an semble a training set must be created before, inside which each instance has a known lable with it (taking *zoo* of UCI repository as example):

```
CDataset Original("zoo.names","zoo.data");
//remove the instances whose label are unknown
Original.RemoveUnknownInstance();
//Get information of the training set
const CASE_INFO &Info=Original.GetInfo();

//instances are group into training set and test set
CDataset TrainSet,TestSet;
//90% as training set
int Sample_num=(int)(Info.Height*0.9);
//the rest as test set
Original.SplitData(Sample_num,TrainSet,TestSet);
```

In this exmaple, the original data set is randomly divided into two parts at first, 90% of the instances are used as the training set of the ensemble and the rest instances are as the test set to examine the performance of the ensemble.

Next we will prepare base-classifier learners with their corresponding paramters for the ensemble builder, as well as percentage of each type of base classifiers in the ensemble. After doing that, the ensemble builder can create all base classifiers automatically, so we needn’t create the base classifier by ourselves directly.

```
//register parameters for base classifier trainers
vector<CEnsemble::CreatorRegisterStr> Creators;
//using two different training parameters for BPNN
{
    //parameters for first BPNN trainer
    CBpnn::RpropParamStr Param;
    Param.HideNode=Info.Width*2;
    Param.MaxEpoch=3000;
    Param.MinMSE=0.01;
    //first trainer
    CEnsemble::CreatorRegisterStr CreatorRegister;
    CreatorRegister.Creator=CBpnn::RpropCreate;
    CreatorRegister.Params=(void*)&Param;
    CreatorRegister.Ratio=0.4;
    Creators.push_back(CreatorRegister);
    //second trainer
    //BPNN using default parameters
    CreatorRegister.Params=(void*)NULL;
    Creators.push_back(CreatorRegister);
    //third trainer: C4.5 decision tree with default parameters
    CreatorRegister.Creator=CC45::Create;
    CreatorRegister.Params=(void*)NULL;
    CreatorRegister.Ratio=0.2;
    Creators.push_back(CreatorRegister);
}
```

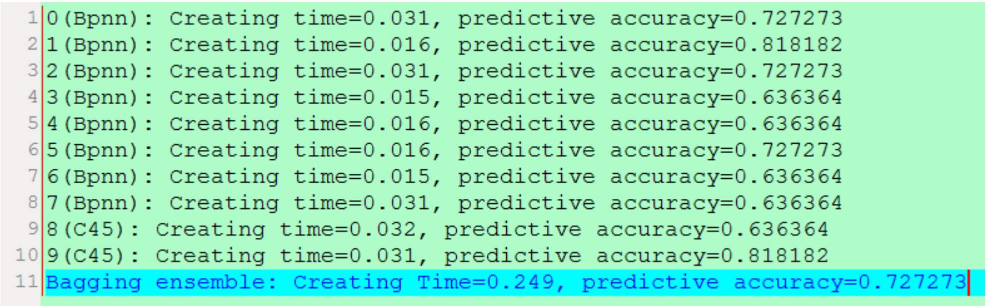
Here we will create an ensemble, in which 40% of base classifiers are BPNN (trained by RPROP algorithm) training with user customed parameters, 40% of BPNN training with default paramters and 20% of C4.5 descision trees training with default parameters.

```
//bagging ensemble
CBagging BaggingEnsemble(TrainSet,0.5,10,Creator);
//use the ensemble to predict the test dataset
const CPrediction *Result=BaggingEnsemble.Classify(TestSet);
//predicting accuracy
cout<<BaggingEnsemble.GetName()<<" ensemble: Creating Time="<<BaggingEnsemble.GetCreateTime()
    <<", predictive accuracy="<<Result->GetAccuracy()<<endl;
delete Result;
```

Our example takes Bagging as the ensemble training algorithm. During training two paramters need to be set: the size of the ensemble and the percentage of the training set used to train a single base classifier. If we set 0.5 for the percentage of training set, a local training set of 50% size of the original training set will be created by bagging re-sampling for each base classifier.

After creating the ensemble can be used as a normal classifier to do predicting, but don’t forget to remove the prediciton result after use. LibEDM also provides methods to extract all the base classifier from the ensemble and to save an ensemble into archive files, see details in front of this chapter.

The output of this example looks like:



```
0(Bpnn): Creating time=0.031, predictive accuracy=0.727273
1(Bpnn): Creating time=0.016, predictive accuracy=0.818182
2(Bpnn): Creating time=0.031, predictive accuracy=0.727273
3(Bpnn): Creating time=0.015, predictive accuracy=0.636364
4(Bpnn): Creating time=0.016, predictive accuracy=0.636364
5(Bpnn): Creating time=0.016, predictive accuracy=0.727273
6(Bpnn): Creating time=0.015, predictive accuracy=0.636364
7(Bpnn): Creating time=0.031, predictive accuracy=0.636364
8(C45): Creating time=0.032, predictive accuracy=0.636364
9(C45): Creating time=0.031, predictive accuracy=0.818182
Bagging ensemble: Creating Time=0.249, predictive accuracy=0.727273
```

We can find there are eight BPNN and 2 C4.5 decision tree in this ensemble and the creating time for this ensemble is 0.249 second and its total prediction accuracy for the test set is anout 0.73. This example also show infrmation for each individual base classifier, such as creating time and prediction accuracy for the test set.

13. Base class for incremental Ensembles: CIncrementalClassifier

```
class CIncrementalEnsemble : public CEnsemble, public CIncrementalClassifier
```

CIncrementalClassifier is the base class for all ensemble classifiers, which can be trained incrementally. An incremental ensemble usually adapt to the change of training instances through changing, removing or creating new base classifers.

CIncrementalEnsemble is a abstract class, can not be instanced directly. If user want to use it, derive new classes from it.

13.1. Requirements

```
#include "Obj.h"
#include "Classifier.h"
#include "IncrementalClassifier.h"
#include "Ensemble.h"
#include "IncrementalEnsemble.h"
using namespace LibEDM;
```

13.2. Members

Training	
Train	Use new instances to train this classifier.
Reset	Reset this classifier to un-trained state.
Classify	
Classfy	Use this classifier to predict.
Dump	
Save	Save to disk files, which this classifier can be restored from.
Dump	Save to readable files for debuging.

13.2.1. CIncrementalEnsemble::Train

Use new instances to train this classifier, so it can adapt to concept changes in the data.

```
    virtual void Train(const CDataset &Dataset)=0;
```

Parameters

Dataset

New instances.

Return Value

Remarks

13.2.2. CIncrementalEnsemble::Reset, Classify, Save, Dump

All these member only call corresponding function in CEnsemble.

```
    virtual void Reset()=0;
```

Parameters

Return Value

Remarks

An incremental ensemble works exactly like an ensemble except that it can be train incrementally.

14. Base class for trunk-based incremental Ensembles: CIncrementalTrunkClassifier

```
    class CIncrementalTrunkEnsemble : public CIncrementalEnsemble
```

Most trunk based incremental ensemble works like this: when a data trunk comes, it creates a base classifiers based on the data trunk, it then judge the value of the new base classifier and assign it a weight or just bandon it.

CIncrementalTrunkEnsemble is a abstract class, can not be instanced directly. If user want to use it, derive new classes from it.

14.1. Requirements

```
#include "Obj.h"

#include "Classifier.h"

#include "IncrementalClassifier.h"

#include "Ensemble.h"

#include "IncrementalEnsemble.h"

#include "IncrementalTrunkEnsemble.h"

using namespace LibEDM;
```

14.2. Members

Training	
Train	Use new instances to train this classifier (classifier is built outside the Train function).

14.2.1. CIncrementalTrunkEnsemble::Train

Use new instances to train this classifier, but the base classifier is created outside, if this ensemble want it to be added, a new classifier should be cloned from it.

```
virtual void Train(const CDataset &Dataset, const CClassifier *Classifier)=0;
```

Parameters

Dataset

New instances.

Classifier

New classifier which is built outside.

Return Value

Remarks

This function is useful when comparing many data-trunk-based incremental ensemble, each ensemble can use the same base classifier, so the random factors of creating base classifiers can be reduced. For incremental ensemble that creates more than one base classifiers on each trunk data, or cretes base classifiers on internal data memory, they should avoid be inherited from CIncrementalTrunkEnsemble.

14.2.2. CIncrementalEnsemble::Reset, Classify, Save, Dump

All these member only call corresponding function in CEnsemble.

```
    virtual void Reset()=0;
```

Parameters

Return Value

Remarks

An incremental ensemble works exactly like an ensemble except that it can be train incrementally.

15. Incremental ensembles

Usually an incremental ensemble will create new base classifiers when new data is come. But the size of the ensemble can not be unlimited, so the ensemble needs to find old classifiers to be replaced. So most part of the incremental ensemble algorithms are deal with the way to create and replace base classifiers.

At present LibEDM implements four incremental ensemble methods:

Pruning methods	Description
CSEA	SEA: Remove the base classifier with worst prediction accuracy to the training data, creating and insert it to ensemble.
CAWE	AWE: Each base classifier is assigned a weight which is related to

	MSE of recent prediction, and the classifier with lowest weight is replaced.
CACE	ACE: It is stream learning algorithm, i.e. it doesn't need to pile some amount of instances before start training, it trains on each instance. It has a dectect mechanism which can find drifts of concepts in the instances. When it find drifts, new classifiers will then be created.

Table 15-1 Incremental ensemble algorithms

The (weighted) classifying, saving and dumping functions are implemented in CIncrementalEnsemble, so most incremental ensembles can directly use them and don not need to implemented it again.

15.1. CSEA

```
class CSEA : public CIncrementalTrunkEnsemble
```

It works on blocks of instances. Each time it creates a new classifier and replace the worst-accurate classifier with it.

W. N. Street and Y. Kim, "A streaming ensemble algorithm (SEA) for large-scale classification," in Proc. 7th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, 2001, pp. 377–382.

15.1.1. Requirements

```
#include <vector>
#include <ctime>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Prediction.h"
#include "Classifier.h"
#include "IncrementalClassifier.h"
#include "Ensemble.h"
#include "IncrementalEnsemble.h"
#include "SEA.h"
```

```
using namespace LibEDM;
```

15.1.2. Members

Construction / Destruction	
CSEA	Construction, input parameters for training.
Overriding of CIncrementalClassifier	
Train	Incremental learning

15.1.2.1. CSEA::CSEA

Input all training parameters.

```
CSEA(int MaxSize,CreateFunc *Creator, const void *Params);
```

Parameters

MaxSize

Max size of the ensemble.

Creator

The entry of creating function for base classifier.

Params

Parameters for base-classifier creating.

Return Value

Remarks

15.2. CAWE

```
class CAWE : public CIncrementalTrunkEnsemble
```

It is also a block-based incremental ensemble. And it also assigns a weight to each base classifier, but the weight is derived from MSE for each base classifier’s prediction. Each time a data comes, the classifier with least weight is replaced.

H. Wang, W. Fan, P. S. Yu, and J. Han, "Mining concept-drifting data streams using ensemble classifiers," in Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, 2003, pp. 226–235.

15.2.1. Requirements

```
#include <vector>
#include <ctime>
#include <iostream>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Prediction.h"
#include "Classifier.h"
#include "IncrementalClassifier.h"
#include "Ensemble.h"
#include "IncrementalEnsemble.h"
#include "CrossValidate.h"
#include "AWE.h"
using namespace LibEDM;
```

15.2.2. Members

Construction / Destruction	
CAWE	Construction, input parameters for training.
Overriding of CIncrementalClassifier	
Train	Incremental learning

15.2.2.1. CAWE::CAWE

Input all training parameters.

```
CAWE(int MaxSize,CreateFunc *Creator, const void *Params);
```

Parameters

MaxSize

Max size of the ensemble.

Creator

The entry of creating function for base classifier.

Params

Parameters for base-classifier creating.

Return Value

Remarks

15.3. CACE

```
class CACE : public CIncrementalEnsemble
```

It is a stream-based incremental ensemble, i.e. each time it only accept and process one instance. So if is feed a data trunk, ACE may build several new base classifiers for each incremental training.

It has two different kinds of base classifiers, one base classifier is trained by a *on-line learner* which can be incrementally trained, all the other base classifiers are trained by a *batch learner* which can be non-incremental classifiers. ACE uses a long term memory to store recent instance, base on which all classifiers except the on-line one are trained.

When each instance comes, ACE uses the possibility of concept drifting to decide whether to create and replace base classifiers, which is derived from the error of prediction to this instance. Although the original ACE algorithm doesn't have a limitation to size of the ensemble, which makes it becoming too heavy for long periods of training, LibEDM adds a parameter for max size of the ensemble.

K. Nishida, K. Yamauchi, and T. Omori, "ACE: Adaptive classifiers-ensemble system for concept-drifting environments," in Proc. 6th Int.Workshop Multiple Classifier Syst., 2005, pp. 176–185.

15.3.1. Requirements

```
#include <vector>
#include <ctime>
#include <set>
#include <cmath>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Prediction.h"
#include "Classifier.h"
#include "IncrementalClassifier.h"
#include "Ensemble.h"
#include "IncrementalEnsemble.h"
#include "GaussNaiveBayes.h"
#include "C45.h"
#include "Statistic.h"
#include "ACE.h"
using namespace LibEDM;
```

15.3.2. Members

Construction / Destruction	
CACE	Construction, input parameters for training.
Overriding of CIncrementalClassifier	
Train	Incremental learning

15.3.2.1. CACE::CACE

Input all training parameters.

```
CACE(int MaxSize,double Alpha=0.01,int Sa=30,int Sc=200,double u=3.0,
    IncermentalCreateFunc *Online=CGaussNaiveBayes::Create, const void *OnlineParams=NULL,
    CreateFunc *Batch=CC45::Create,const void *BtachParams=NULL);
```

Parameters

MaxSize

Max size of the ensemble.

Alpha

Significant level for detecting conecpt drifting.

Sa

Size of short term memory (control the mimimum time before creating a new base classifier).

Sc

Size of long term memory (control the maximum time before creating a new base classifier).

Online

The entry for on-line training.

OnlineParams

The papameters for on-line trainer.

Batch

The entry for batch training (non-incremental training).

OnlineParams

The papameters for batch trainer.

Return Value

Remarks

15.3.3. Example for Incremental Ensemble (Example7)

15.3.3.1. Source Files

- Example7.cpp: Using SEA ensemble to show incremental training.

15.3.3.2. How to compile?

- Windows: Open examples/IncEnsemble.vcpro in MSVC 2005 or higher version
- Linux: type “make CrossValidate” in the examples directory of LibEDM

15.3.3.3. Description

This example show predicting and incremental training of SEA ensemble. In each incremental step, several instances are read from the data file to build a data set, on which the ensemble is tested and then the ensemble is incrementally trained.

Also this example shows how to read block by block from an ARFF format data file, in which description and data are in the same single file. First the header (data description) should be read from the file:

```
//open data file
ifstream DataFile;
DataFile.open("zoo.arff");
if(DataFile.fail())
{
    throw(CError("open file failed!",100,0));
}
//reading header of the ARFF format file; then the file pointer moves to the beginning of data
//here we only get the description of this data set
//we will read the data block by block
CASE_INFO Info;
{
    CArffData DataInfo;
    DataInfo.LoadInfo(DataFile);
    Info=DataInfo.GetInfo();
}
```

From the description, the format of data can be known, by knowing which we can load several instances (data block) each time to construct the data set for each incremental step:

```
//ten instance as a data block
CArffData DataSet;
DataSet.Load(Info,DataFile,BlockSize);
if(DataSet.GetInfo().Height<=0)
    break;
```

To create a SEA ensemble, we just need instance an object from class CSEA by giving necessary parameters (here we assume the ensemble is made up of BP neural networks trained by default parameters):

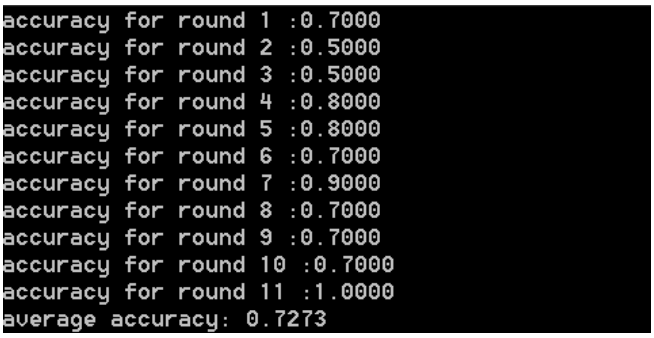
```
//create and initializing a SEA incremental ensemble
CSEA Sea(25,CBpnn::RpropCreate,NULL);
double SeaAvg=0;
```

Then we can use the ensemble to predict and the data set for each incremental step is used to re-train it:

```
//try predicting new data using old classifier
CPrediction *Prediction=Sea.Classify(DataSet);
double Acc=Prediction->GetAccuracy();
```

```
delete Prediction;
cout<<"accuracy for round "<<i+1<<" :"<<setprecision(4)<<setiosflags(ios::fixed)<<Acc<<endl;
SeaAvg+=Acc;
i++;

//incremental training
Sea.Train(DataSet);
```

The output of this example looks like:



```
accuracy for round 1  :0.7000
accuracy for round 2  :0.5000
accuracy for round 3  :0.5000
accuracy for round 4  :0.8000
accuracy for round 5  :0.8000
accuracy for round 6  :0.7000
accuracy for round 7  :0.9000
accuracy for round 8  :0.7000
accuracy for round 9  :0.7000
accuracy for round 10  :0.7000
accuracy for round 11  :1.0000
average accuracy: 0.7273
```

16. Base class for ensemble pruning: CEnsemblePruner

```
class CEnsemblePruner : public CClassifier
```

A pruned ensemble is a subset of the original ensemble. It improves the perfomance and efficiency of the original ensemble by selecting only good classifiers (or good combination of some classifiers). A pruned ensemble must work with the original ensemble, because all its members are actually members of the original ensemble. Many methods have be presented for selecting base classifiers.

CEnsemblePruner is the base class for all the ensemble pruning method. CEnsemblePruner prunes an ensemble by giving to some of its base classifiers zero weights (then these base classifiers will not attend the voting), the original weights of the ensemble (if existing) are overrided.

Z. H. Zhou, J. Wu and W. Tang. Ensembling neural networks: many could be better than all. Artificial Intelligence, 137(1-2):239-263, 2002.

16.1. Requirements

```
#include <vector>
#include <ctime>
#include <fstream>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "EnsemblePruner.h"
using namespace LibEDM;
```

16.2. Members

Construction	
CEnsemblePruner	Create a pruned emsemble.
Getting Information	
GetWeights	Retrun all weights of the ensemble after pruned.
GetSize	Size of the pruned ensemble (number of classifiers in original ensmble with weights >0)
GetStaticName	The name is for restoring an pruned ensemble from files.
Predict	
Classify	Use this pruned ensemble to predicte a data set.
Overriding	
Save	Save all inside data to disk file.
Dump	Dump all inside data to disk file for inspecting.
Classify	Use this classifier to predicte a data set.
Clone	Clone a pruner.
Data Members	
Ensemble	The original ensemble been pruned.
Weights	The new weights to all classifiers of the original ensemble after pruning.

16.2.1. CEnsemblePruner::CEnsemblePruner

Restoring a pruned ensemble from disk files. Or pruned an ensemble by user-defined weights (some classifiers are zero weighted).

```
CEnsemblePruner(const string &Path,const string &FileName,const CEnsemble &Ensemble);
CEnsemblePruner(const CEnsemble &Ensemble,const vector<double> &UserWeights);
```

Parameters

Path

    The directory of the files where the pruned ensemble will be restored from.

FileName

    The name of restoring files.

Ensemble

    The ensemble have been pruned.

UserWeights

    User-defined weights. Classifers with zero weights are pruned.

Return Value

The pruned ensemble.

Remarks

16.2.2. CEnsemblePruner::GetWeights

Return the new weights for all classifiers in the ensemble after pruning.

```
const vector<double> &GetWeights()
```

Parameters

Return Value

A read-only array of weights.

Remarks

16.2.3. CEnsemblePruner::GetSize

Get size of the pruned ensemble.

```
int GetSize()
```

Parameters

Return Value

Size of the pruned ensemble.

Remarks

16.2.4. CEnsemblePruner::GetStaticName

Internal name for all pruned ensemble.

```
static string GetStaticName()
```

Parameters

Return Value

Name for all pruned ensemble.

Remarks

16.2.5. CEnsemblePruner::Classify

```
virtual CPrediction *Classify(const CDataset &DataSet) const
CPrediction *Classify(const CDataset &DataSet,const vector<CPrediction*> &Predictions) const
```

Parameters

DataSet

The data set to be predict by the ensemble.

Predictions

Pre-obtained predictions of all classifiers of the ensemble to the input data set.

Return Value

Predicition of the pruned ensmeble to the input data set.

Remarks

For an input data set to be predicted, each classifier of the ensemble has to predict the data set individually at first, then all ther prediction will be combined by a method (commonly voting), that may be a long process. This is just the first form of classify() does.
In some stuation before ensemble predicting, individual prediction of each classifier may already available (for exmaple, the process of ensemble pruning need the prediciton of individual classifier, then for the pruned ensemble all the prediciton of base classifiers are available), the second forms of classify() can be used to save time.

16.2.6. CEnsemblePruner::Clone

Copy a pruner.

```
virtual CClassifier *Clone() const;
```

Parameters

Return Value

A new pruner to the original ensemble.

Remarks

16.2.7. CEnsemblePruner::Ensemble

The ensemble been pruned.

```
const CEnsemble &Ensemble;
```

Fields

Remarks

16.2.8. CEnsemblePruner::Weights

The new weights for all classifiers in the original ensemble after pruning.

```
vector<double> Weights;
```

Fields

Remarks

17. Ensemble pruning algorithms

At present LibEDM implements eight ensemble pruning methods described below:

Pruning methods	Description
CSelectAll	Select all classifiers.
CSelectBest	Select best classifier according to their performances on the

	validation set.
CForwardSelect	Add the best classifier to the ensemble, one classifier each time.
CGasen	Select classifiers according to their weights, which is evolved by a genetic algorithm.
CCluster	Selecting by deviding classifiers into groups, then perform pruning in each group
CPMEP	Selecting by pattern mining.
CMDSQ	Order classifiers by the distances between their corresponding vectors to the target vector then cloestest classifiers are selected.
COrientOrder	Order classifiers by the angels between their corresponding vector to then target vector then cloestest classifiers are selected.

Table 17-1 Pruning Methods

17.1. CSelectAll

```
class CSelectAll : public CEnsemblePruner
```

Select all classifiers of the ensemble (no pruning is performed). This method is often used as the basis when comparing performance of different pruning method.

17.1.1. Requirements

```
#include <vector>
#include <ctime>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "EnsemblePruner.h"
#include "Prediction.h"
#include "SelectAll.h"
using namespace LibEDM;
```

17.1.2. Members

Construction / Destruction	
CSelectAll	Construction.
Create	Static function used to prune an ensemble by calling this pruning method

17.1.2.1. CSelectAll::CSelectAll

Select all classifiers of the ensemble.

```
CSelectAll(const CEnsemble &Ensemble,const CDataset &ValidatingSet);
CSelectAll(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const vector<CPrediction*>
    &Predictions);
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used here).

Predictions

Predictions of all classifiers to the validation data set.

Return Value

A pruned ensemble.

Remarks

If the prediction of each base classifier to the validation set is already obtained, use the second form of the construction functions to save pruning time.

17.1.2.2. CSelectAll::Create

Pruning an ensemble by calling this pruning method.

```
static CEnsemblePruner *Create(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const
        vector<CPrediction*> &Predictions,  const void *Params)
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used in this method).

Predictions

Predictions of all classifiers to the validation data set.

Params

Not used.

Return Value

A pruned ensemble.

Remarks

This function is useful when comparing different pruning methods (or same method with different pruning parameters). First registering the create function (and pruning parameters) then all the functions can be called automatically.

17.2. CSelectBest

```
class CSelectBest : public CEnsemblePruner
```

Select the best base classifier of the ensemble (based the prediction accuracy of each classifier on the validation set), all other classifiers are pruned.

17.2.1. Requirements

```
#include <vector>
```

```
#include <ctime>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "EnsemblePruner.h"
#include "Prediction.h"
#include "SelectBest.h"
using namespace LibEDM;
```

17.2.2. Members

Construction / Destruction	
CSelectBest	Construction.
Create	Static function used to prune an ensemble by calling this pruning method

17.2.2.1. CSelectBest::CSelectBest

Select all classifiers of the ensemble.

```
CSelectAll(const CEnsemble &Ensemble,const CDataset &ValidatingSet);
CSelectAll(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const vector<CPrediction*>
    &Predictions);
```

Parameters

Ensemble

    The ensemble to be pruned.

ValidatingSet

    The validation data set used to evaluate the classifiers (not used here).

Predictions

    Predictions of all classifiers to the validation data set.

Return Value

A pruned ensemble.

Remarks

If the prediction of each base classifier to the validation set is already obtained, use the second form of the construction functions to save pruning time.

17.2.2.2. CSelectBest::Create

Pruning an ensemble by calling this pruning method.

```
static CEnsemblePruner *Create(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const
        vector<CPrediction*> &Predictions, const void *Params)
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used in this method).

Predictions

Predictions of all classifiers to the validation data set.

Params

Not used in this method.

Return Value

A pruned ensemble.

Remarks

This function is useful when comparing different pruning methods (or same method with different pruning parameters). First registering the create function (and pruning parameters) then all the functions can be called automatically.

17.3. CForwardSelect

```
class CSelectBest : public CEnsemblePruner
```

Select the best base classifier of the ensemble (based the prediction accuracy of each classifier on the validation set), all other classifiers are pruned.

R. Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes. Ensemble selection from libraries of models. In ICML ‘04, Banff, Alberta, Canada, 2004.

17.3.1. Requirements

```
#include <vector>
#include <ctime>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "EnsemblePruner.h"
#include "Prediction.h"
#include "SelectBest.h"
using namespace LibEDM;
```

17.3.2. Members

Construction / Destruction	
CForwardSelect	Construction.
Create	Static function used to prune an ensemble by calling this pruning method

17.3.2.1. CForwardSelect::CForwardSelect

Inspect classifier one by one in each selecting round, the best or none classifier is added to the ensemble.

```
CForwardSelect(const CEnsemble &Ensemble,const CDataset &ValidatingSet);
CForwardSelect(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const
    vector<CPrediction*> &Predictions);
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used here).

Predictions

Predictions of all classifiers to the validation data set.

Return Value

A pruned ensemble.

Remarks

If the prediction of each base classifier to the validation set is already obtained, use the second form of the construction functions to save pruning time.

17.3.2.2. CForwardSelect::Create

Pruning an ensemble by calling this pruning method.

```
static CEnsemblePruner *Create(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const
        vector<CPrediction*> &Predictions, const void *Params)
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used in this method).

Predictions

Predictions of all classifiers to the validation data set.

Params

Not used in this method.

Return Value

A pruned ensemble.

Remarks

This function is useful when comparing different pruning methods (or same method with different pruning parameters). First registering the create function (and pruning parameters) then all the functions can be called automatically.

17.4. CGasen

```
class CGasen : public CEnsemblePruner
```

Evolve a float array of weights corresponding to all the base classifiers of the ensemble then only classifiers with high weights are added to the target ensemble.

Z. H. Zhou, J. Wu and W. Tang. Ensembling neural networks: many could be better than all. Artificial Intelligence, 137(1-2):239-263, 2002.

17.4.1. Requirements

```
#include <vector>
#include <list>
using namespace std;
```

```
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "EnsemblePruner.h"
#include "Prediction.h"
#include "GA.h"
#include "Gasen.h"
using namespace LibEDM;
```

17.4.2. Members

Construction / Destruction	
CGasen	Construction.
Create	Static function used to prune an ensemble by calling this pruning method

17.4.2.1. CGasen::CGasen

Evolve weights corresponding to all base classifiers by genetic algorithm, and select base classifiers whoes weights are higher than the threshold to target ensemble.

```
CGasen(const CEnsemble &UEnsemble, const CDataset &UValidatingSet, int TribeNum=30, double Pc=0.5,
    double Pm=0.05, int MaxGen=100, int MaxNoInc=100, double Lamda=0);
        //predict a dataset
CGasen(const CEnsemble &UEnsemble, const CDataset &UValidatingSet, const vector<CPrediction*>
    &UPredictions, int TribeNum=30, double Pc=0.5,
    double Pm=0.05, int MaxGen=100, int MaxNoInc=100, double Lamda=0);
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used here).

Predictions

Predictions of all classifiers to the validation data set.

TribeNum

Size of tribe (Number of individuals for each generation).

Pc

probability of crossover

Pm

probability of mutation

MaxGen

Stop criteria: max number of evolution.

MaxNoInc

Stop criteria: Max number of evolution if no better individual can be found.

Lamda

Threshold of weights for selecting base classifiers.

Return Value

A pruned ensemble.

Remarks

If the prediction of each base classifier to the validation set is already obtained, use the second form of the construction functions to save pruning time.

17.4.2.2. CGasen::Create

Pruning an ensemble by calling this pruning method.

```
static CEnsemblePruner *Create(const CEnsemble &Ensemble, const CDataset &ValidatingSet, const
        vector<CPrediction*> &Predictions,  const void *Params)
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used in this method).

Predictions

Predictions of all classifiers to the validation data set.

Params

Parameters for the pruning method.

Return Value

A pruned ensemble.

Remarks

This function is useful when comparing different pruning methods (or same method with different pruning parameters). First registering the create function (and pruning parameters) then all the functions can be called automatically.

17.4.3. Data Structures

ParamStr	Parameters for the genetic algorithm.

17.4.3.1. CGasen::ParamStr

Paramesters for controling the behavior of Genetic Algorithm and selecting of base classifiers.

```
typedef struct ParamStr
{
	//parameters used for GA
	static int TribeNum;//size of tribe
	static double Pc;//probability of crossover
	static double Pm;//probability of mutation
	static int MaxGen;//max number of evolution
```

```
    static int MaxNoInc;//Max number of evolution if no better individual is found
    static double Lamda;//threshold for weights
}ParamStr;
```

Parameters

TribeNum

Size of tribe (Number of individuals for each generation).

Pc

probability of crossover

Pm

probability of mutation

MaxGen

Stop criteria: max number of evolution.

MaxNoInc

Stop criteria: Max number of evolution if no better individual can be found.

Lamda

Threshold of weights for selecting base classifiers.

Remarks

17.5. CCluster

```
class CCluster : public CEnsemblePruner
```

Group the base classifiers by *k-mean* and centroid for each group is selected (or calculated). Then in each group, each classifier will calculate its disagreement with the centroid. After that, base classifiers with low disagreement (that is, they are similar to the centroid and are more likely to be redundant) are pruned.

A. Lazarevic and Z. Obradovic. Effiective pruning of neural network classifier ensembles. In IJCNN 2001, pages 796–801, 2001.

17.5.1. Requirements

```
#include <vector>

#include <cmath>

#include <algorithm>

using namespace std;

#include "Obj.h"

#include "DataSet.h"

#include "Classifier.h"

#include "Ensemble.h"

#include "Prediction.h"

#include "EnsemblePruner.h"

#include "Cluster.h"

using namespace LibEDM;
```

17.5.2. Members

Construction / Destruction	
CCluster	Construction.
Create	Static function used to prune an ensemble by calling this pruning method

17.5.2.1. CCluster::CCluster

Grouping and selecting.

```
CCluster(const CEnsemble &Ensemble,const CDataset &ValidatingSet,double Lamda=1.0);

CCluster(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const vector<CPrediction*>
      &Predictions,double Lamda=1.0);
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used here).

Predictions

Predictions of all classifiers to the validation data set.

Lamda

In each group if a node (base classifier) is disagree with the centroid (disagreement value is less than Lamda), it is pruned.

Return Value

A pruned ensemble.

Remarks

If the prediction of each base classifier to the validation set is already obtained, use the second form of the construction functions to save pruning time.

17.5.2.2. CCluster::Create

Pruning an ensemble by calling this pruning method.

```
static CEnsemblePruner *Create(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const
        vector<CPrediction*> &Predictions, const void *Params)
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used in this method).

Predictions

Predictions of all classifiers to the validation data set.

Params

Parameters for this pruning method.

Return Value

A pruned ensemble.

Remarks

This function is useful when comparing different pruning methods (or same method with different pruning parameters). First registering the create function (and pruning parameters) then all the functions can be called automatically.

17.5.3. Data Structures

ParamStr	Parameters for the genetic algorithm.

17.5.3.1. CCluster::ParamStr

Paramesters for selecting.

```
typedef struct ParamStr
{
    static double Lamda;//threshold of disagreement
}ParamStr;
```

Parameters

Lamda

<=1.0 and >0, Threshold of disagreement for selecting base classifiers, base classifiers with disagreement value less than Lamda will be remove.

Remarks

A classifier with lower disagreement value to the centroid means it is similar to the centroid, so it is removed from ensemble because it can be represented by the centroid.

17.6. CPMEP

```
class CPMEP : public CEnsemblePruner
```

All classifiers' predictions to each instance of the validation set is regarded as an item set, based on all item sets a FP-tree is built up. Then patterns of items are mined from the FP-tree according to each possible size of target ensembles, among which the best is output.

Q. L. Zhao, Y. H. Jiang and M. Xu. A fast ensemble pruning algorithm based on pattern mining. Data Mining and Knowledge Discovery, 19(2): 277-292, 2009.

17.6.1. Requirements

```
#include <cmath>
#include <string>
#include <set>
#include <algorithm>
#include <fstream>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "Prediction.h"
#include "EnsemblePruner.h"
#include "PMEP.h"
using namespace LibEDM;
```

17.6.2. Members

Construction / Destruction	
CPMEP	Construction.
Create	Static function used to prune an ensemble by calling this pruning

	method
Dump	Dump the

17.6.2.1. CPMEP::CPMEP

Ensemble pruning by pattern mining.

```
CPMEP(const CEnsemble &Ensemble,const CDataset &ValidatingSet);
CPMEP(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const vector<CPrediction*>
      &Predictions);
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used here).

Predictions

Predictions of all classifiers to the validation data set.

Return Value

A pruned ensemble.

Remarks

If the prediction of each base classifier to the validation set is already obtained, use the second form of the construction functions to save pruning time.

17.6.2.2. CPMEP::Create

Pruning an ensemble by calling this pruning method.

```
static CEnsemblePruner *Create(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const
        vector<CPrediction*> &Predictions, const void *Params)
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used in this method).

Predictions

Predictions of all classifiers to the validation data set.

Params

Not used.

Return Value

A pruned ensemble.

Remarks

This function is useful when comparing different pruning methods (or same method with different pruning parameters). First registering the create function (and pruning parameters) then all the functions can be called automatically.

17.6.2.3. CPMEP::Dump

Dump internal informations of this algorithm to files, for debugging.

```
void Dump(const string &FileName,const vector<CaseClassArrayStr> &CaseClassTab);
void Dump(const string &FileName,const vector<TreePathStr> &TreePathTable);
void Dump(const string &FileName,const vector<SelClassifierStr> &SelClassifiers);
```

Parameters

FileName

Name of the output file.

CaseClassTab

TreePathTable

SelClassifiers

    Internal data of this pruning algorithm

Return Value

Remarks

See the description of data structures.

17.6.3. Data Structures

CaseClassRecStr	A classifier's prediction to an individual instance.
TreeNodeStr	A node of the FP-tree.
TreePathStr	A path in the FP-tree (from root to a node) and number of instances in the node (Number of correctly predicted instances by classifiers on the path).
SelClassifierStr	A set of selected classifiers and the correctness of its prediciton.

17.6.3.1. CPMEP::CaseClassRecStr

Paramesters for selecting.

```
typedef struct CaseClassRecStr
{
    int Correct;//0- wrong, 1- correct, >1- number of correctness(only last row)
    int Classifier;//id of classifier
}CaseClassRecStr;
```

Parameters

Classifier

    Id of a classifier.

Correct

For a instance:

0: If this classifier predict it wrongly;

1: If this classifier predict it correctly;

For all instancesin of the validation set:

>1: Number of correct precitions made by this classifier.

Remarks

Using this data structure, all classifiers' prediction to a single instance is transform to a vector, furthermore, an item (classifiers) set if we only keep the classifiers which predict correctly.

17.6.3.2. CPMEP::TreeNodeStr

Node of a FP-tree.

```
typedef struct TreeNodeStr
{
    int                    Classifier;//id of classifier of this node
    int                    Count;//number of instances
    vector<TreeNodeStr>    SubNodes;//sub nodes
}TreeNodeStr;
```

Parameters

Classifier

Id of a classifier.

Count

Number of instances can be correctly predicted by each classifier from root to this node.

SubNodes

Sub nodes of this node.

Remarks

17.6.3.3. CPMEP::TreePathStr

A root-start path.

```
typedef struct TreePathStr
{
    vector<int>   Classifiers;// a path in FP-tree
    int           Count;//instances that on this path
}TreePathStr;
```

Parameters

Classifiers

    Ids of classifiers in this path.

Count

    Number of instances can be correctly predicted by each classifier on this path.

Remarks

17.6.3.4. CPMEP::SelClassifierStr

Result of a sub selecting.

```
typedef struct SelClassifierStr
{
    set<int> Set;
    int Count;
}SelClassifierStr;
```

Parameters

Set

    Ids of all selected classifier.

Count

Number of instances can be correctly predicted by combining all the prediction of the selected classifiers.

Remarks

For each possible size of target ensemble, PMEP will do a sub selecting. Among all the results of sub selecting, only the best one is output.

17.7. CMDSQ

```
class CMDSQ : public CEnsemblePruner
```

For each classifier, its prediction to all the instances is formulated as a vector (1 if it can predict an instance correctly, -1 otherwise). This algorithm includes several iterations of selection. In each selection the classifiers are ordered by their corresponding vector's Euclidean distance to a reference vector which is placed somewhere in the first quadrant. And the closest classifier is selected into the final ensemble.

G. Martinez-Munoz, D. Hernandez-Lobato and A. Suarez. An analysis of ensemble pruning techniques based on ordered aggregation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 31(2):245–259, 2009.

17.7.1. Requirements

```
#include <vector>
#include <cmath>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "Prediction.h"
#include "EnsemblePruner.h"
#include "MDSQ.h"
using namespace LibEDM;
```

17.7.2. Members

Construction / Destruction	
CMDSQ	Construction.
Create	Static function used to prune an ensemble by calling this pruning method

17.7.2.1. CMDSQ::CMDSQ

Inspect classifier one by one in each selecting round, the best or none classifier is added to the ensemble.

```
CMDSQ(const CEnsemble &Ensemble,const CDataset &ValidatingSet);
CMDSQ(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const vector<CPrediction*>
    &Predictions);
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used here).

Predictions

Predictions of all classifiers to the validation data set.

Return Value

A pruned ensemble.

Remarks

If the prediction of each base classifier to the validation set is already obtained, use the second form of the construction functions to save pruning time.

17.7.2.2. CMDSQ::Create

Pruning an ensemble by calling this pruning method.

```
static CEnsemblePruner *Create(const CEnsemble &Ensemble, const CDataset &ValidatingSet, const
        vector<CPrediction*> &Predictions,  const void *Params)
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used in this method).

Predictions

Predictions of all classifiers to the validation data set.

Params

Not used in this method.

Return Value

A pruned ensemble.

Remarks

This function is useful when comparing different pruning methods (or same method with different pruning parameters). First registering the create function (and pruning parameters) then all the functions can be called automatically.

17.8. COrientOrder

```
class COrientOrder : public CEnsemblePruner
```

For each classifier, its prediction to all the instances is formulated as a vector (1 if it can predict an instance correctly, -1 otherwise). Then the classifiers are order by their corresponding vector's angels to the reference vector, among which classifiers with angels no less than average are pruned.

G. Martinez-Munoz, D. Hernandez-Lobato and A. Suarez. An analysis of ensemble pruning techniques based on ordered aggregation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 31(2):245–259, 2009.

17.8.1. Requirements

```
#include <vector>
#include <cmath>
using namespace std;
#include "Obj.h"
#include "DataSet.h"
#include "Classifier.h"
#include "Ensemble.h"
#include "Prediction.h"
#include "EnsemblePruner.h"
#include "MDSQ.h"
using namespace LibEDM;
```

17.8.2. Members

Construction / Destruction	
COrientOrder	Construction.
Create	Static function used to prune an ensemble by calling this pruning method

17.8.2.1. COrientOrder::COrientOrder

Inspect classifier one by one in each selecting round, the best or none classifier is added to the ensemble.

```
COrientOrder(const CEnsemble &Ensemble,const CDataset &ValidatingSet);
COrientOrder(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const
    vector<CPrediction*> &Predictions);
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used here).

Predictions

Predictions of all classifiers to the validation data set.

Return Value

A pruned ensemble.

Remarks

If the prediction of each base classifier to the validation set is already obtained, use the second form of the construction functions to save pruning time.

17.8.2.2. COrientOrder::Create

Pruning an ensemble by calling this pruning method.

```
static CEnsemblePruner *Create(const CEnsemble &Ensemble,const CDataset &ValidatingSet,const
        vector<CPrediction*> &Predictions, const void *Params)
```

Parameters

Ensemble

The ensemble to be pruned.

ValidatingSet

The validation data set used to evaluate the classifiers (not used in this method).

Predictions

Predictions of all classifiers to the validation data set.

Params

Not used in this method.

Return Value

A pruned ensemble.

Remarks

This function is useful when comparing different pruning methods (or same method with different pruning parameters). First registering the create function (and pruning parameters) then all the functions can be called automatically.

17.9. Example for Ensemble Pruning (Example4, Example4.1)

In this example we will create an ensemble of 50 BPNN at first, then we will try to prune it by two algorithm: Select Best and MDSQ.

17.9.1. How to compile?

- Windows: Open examples/pruner.vcproj or examples/pruner1.vcproj in MSVC 2005 or higher version
- Linux: type “make pruner” in the examples directory of LibEDM

17.9.2. Description

First we create an ensemble of 50 base classifiers from the data set *zoo* of UCI. For the purpose of simplicity, all the base classifiers are BPNNs:

```
CDataset Original("zoo.names","zoo.data");
//remove the instances whose label are unknown
Original.RemoveUnknownInstance();
//Get information of the training set
const CASE_INFO &Info=Original.GetInfo();

//instances are group into training set and test set
CDataset TrainSet,TestSet,ValSet;
```

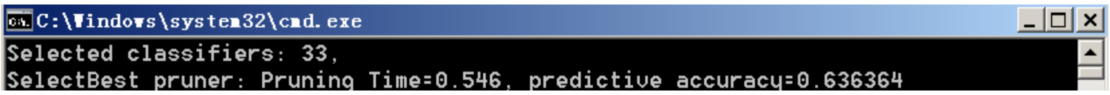
```
//90% as train set
int Sample_num=(int)(Info.Height*0.9);
//the rest as testing
Original.SplitData(Sample_num,TrainSet,TestSet);
//the validation set for ensemble pruning
TrainSet.BootStrap((int)(Sample_num*0.5),ValSet);

//register parameters for base classifier trainers
vector<CEnsemble::CreatorRegisterStr> Creators;
{
    //BPNN with default parameters
    CEnsemble::CreatorRegisterStr CreatorRegister;
    CreatorRegister.Creator=CBpnn::RpropCreate;
    CreatorRegister.Ratio=1.0;
    CreatorRegister.Params=(void*)NULL;
    Creators.push_back(CreatorRegister);
}
 //bagging ensemble， size 50
CBagging BaggingEnsemble(TrainSet,0.5,50,Creators);
```

Usually a validation data set is required for ensemble pruning. Using the validation set as a basis, an ensemble pruner can then judge which base classifiers are better and which are worst so can be pruned, hence we know that the validation set is very important. There are many way to create the validation set although in this example we just use bootstrap re-sampling to create it from the training set of the ensemble.

Next prune this ensemble by creating a pruner object:

```
//pruning the ensemble by selecting best
CSelectBest SelectBest(BaggingEnsemble,ValSet);
//show selected classifiers
cout<<"Selected classifiers: ";
const vector<double> &Weights=SelectBest->GetWeights();
for(int i=0;i<(int)Weights.size();i++)
    if(Weights[i]>0)
        cout<<i<<",";
cout<<endl;

//predicting
CPrediction *Result=SelectBest.Classify(TestSet);
//prediction accuracy
cout<<SelectBest.GetName()<<" pruner: Pruning Time="<<SelectBest.GetCreateTime()<<
    ", predictive accuracy="<<Result->GetAccuracy()<<endl;
delete Result;
```

Once created, the pruner can be used as a normal classifier to do prediction. This example also shows the way to know which classifiers have been pruned by the pruner, that is, in a pruned ensemble a classifier with zero weight means it is pruned from the ensemble. The output looks like:

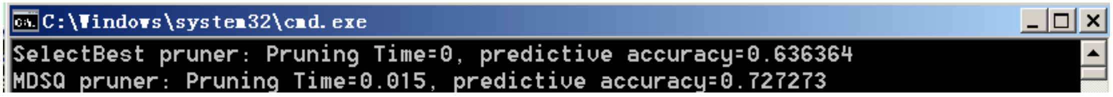


```
Selected classifiers: 33,
SelectBest pruner: Pruning Time=0.546, predictive accuracy=0.636364
```

In the example above (example4.cpp) only an ensemble pruning method is performed, but in many situations we need to compare several pruning methods, so we can use the second form of pruner creating function to save the time consumed (Example4.1.cpp):

```
//because all ensemble pruner need the prediction of each base classifier to the validation set
//we can do the predictions only once and all the pruner (if exist) can share the results
//each base classifier predict on the validation set
vector<CPrediction*> *Predictions=BaggingEnsemble.AllClassify(ValSet);

//All the pruner ensemble created by different methods
vector<CEnsemblePruner*> PrunedEnsembles;
CEnsemblePruner* Pruned=CSelectBest::Create(BaggingEnsemble,ValSet,(*Predictions),NULL);
PrunedEnsembles.push_back(Pruned);

Pruned=CMDSQ::Create(BaggingEnsemble,ValSet,(*Predictions),NULL);
PrunedEnsembles.push_back(Pruned);

//number of ensemble pruning methods
for(int j=0;j<(int)PrunedEnsembles.size();j++)
{
    //pruned ensemble predict the test set
    CPrediction *Result=PrunedEnsembles[j]->Classify(TestSet);
    //predicting accuracy
    cout<<PrunedEnsembles[j]->GetName()<<
        " pruner: Pruning Time="<<PrunedEnsembles[j]->GetCreateTime()<<
        ", predictive accuracy="<<Result->GetAccuracy()<<endl;
    delete Result;
    //remove the pruned ensemble
    delete PrunedEnsembles[j];
}

//destroy the predictions of all base classifiers
for(int j=0;j<(int)(*Predictions).size();j++)
    delete ((*Predictions)[j]);
delete Predictions;
```

Because almost all ensemble pruner need the predictions of all base classifiers to the validation set and all these prediction may be very time-consuming, we can do all these predictions only once and let all the ensemble pruner share these predictions. We can do this by using the *create* functions (which actually call the second form of the construction function) for all the pruners and use iteration to process all the pruners in the same way. The output for this example looks like:



```
SelectBest pruner: Pruning Time=0, predictive accuracy=0.636364
MDSQ pruner: Pruning Time=0.015, predictive accuracy=0.727273
```

18. Utility Classes

LibEDM also provides functions other than ensemble and pruning that may be useful to users who do experiments about ensemble or machine learning.

18.1. Weighted re-sampling

```
class CRoulette
```

For weighted sampling there is an item set at first, each item in the set is connected to a weight which is the probability of the item to be selected. Each time an item is randomly selected according to its weight, and one item is allowed to be selected more than once.
Algorithm is implemented in the idea of roulette.

18.1.1. Requirements

```
#include <vector>
using namespace std;
#include "Obj.h"
using namespace LibEDM;
```

18.1.2. Members

Construction / Destruction	
CRoulette	Construction.

Methods	
Poll	Randomly select an item.

18.1.2.1. CRoulette::CRoulette

Create a roulette for weighted sampling.

```
CRoulette(const vector<double> &Weights)
```

Parameters

Weights

Weights associated with the items are be selected.

Return Value

Remarks

Weights will be normalized during construction, so sum of all weights doesn’t need to be 1.0, i.e. the input weights are just relative probabilities.

18.1.2.2. CRoulette::Poll

Select an item randomly according to the weights.

```
int Poll()
```

Parameters

Return Value

A integer value indicates the number of the selected item.

Remarks

18.2. Random Sequence

```
class CRandSequence
```

Given an item set, randomly return an item each time, no duplication allowed.

18.2.1. Requirements

```
#include <vector>
using namespace std;
#include "Obj.h"
#include "RandSequence.h"
using namespace LibEDM;
```

18.2.2. Members

Construction / Destruction	
CRandSequence	Construction.
Methods	
Poll	Randomly select an item.
Reset	Reset each item’s status to unselected.

18.2.2.1. CRandSequence::CRandSequence

Create a roulette for weighted sampling.

```
CRandSequence(int UMax);
```

Parameters

UMax

Number of items in the item set.

Return Value

Remarks

18.2.2.2. CRoulette::Poll

Return an item randomly which has not be selected before.

```
int Poll()
```

Parameters

Return Value

A integer value indicates the number of the selected item.

Remarks

18.2.2.3. CRoulette::Reset

Reset each item’s status to unselected.

```
void Reset();
```

Parameters

Return Value

Remarks

18.3. Genetic Algorithm

```
class CGA
```

Encapsulation of the Genetic Algorithm.

18.3.1. Requirements

```
#include <cmath>
using namespace std;
#include "Obj.h"
#include "GA.h"
using namespace LibEDM;
```

18.3.2. Members

Construction / Destruction	
CGA	Construct a CGA object, setting parameters and preparing for evolving.
Methods	
Evolve	Run a evolving according to given fitness function
Data Structure	
FitFunc	An user-defined function used to caculate the fitness of an individual.

18.3.2.1. CGA::CGA

Set all paramters for evolving.

```
CGA(int TribeNum, double Pc, double Pm, int MaxGen, int MaxNoInc);
```

Parameters

TribeNum

Number of individuals in each generation.

Pc

Probability of crossover.

Pm

Probability of mutation.

MaxGen

Max number of evolution (gerneration).

MaxNoInc

Max number of evolution if no better individual can be found.

Return Value

Remarks

18.3.2.2. CGA::Eovle

Run an evolution and return the best individual ever found.

```
double Evolve(void *Params,FitFunc Fit,vector<double> &Best);
```

Parameters

Fit

User defined function to calculate the fitnesses of all individuals in a generation.

Params

Parameters may be used by the user-defined fitness function.

Best

Best individual evolved.

Return Value

The fitness of the best individual.

Remarks

18.3.2.3. CGA::FitFunc

User-defined fitness function.

```
typedef void (*FitFunc)(void *Params,const DoubleArray2d &Tribe,vector<double> &Fitnesses);
```

Parameters

Params

Parameters may be used by the user-defined fitness function.

Tribe

All individuals in a generation.

Fitnesses

Fitnesses for all input individuals.

Remarks

This function is designed to get all individuals' fitness in a generation, so speedup mechanism such as parallel, cache can be applied inside it.

18.4. Cross Validation

```
//test a trainer
template <class StatisticStr> bool CrossValidate(int FoldNum, const CDataset &DataSet,
    CreateFunc *Creator, const void *Param, vector<StatisticStr> &Statistics)
//test a classifier
template <class StatisticStr> bool CrossValidate(int FoldNum, const CDataset &DataSet,
    const CClassifier *Classifier, vector<StatisticStr> &Statistics)
```

Cross validation is a process that tests a trainer or a classifier use only one data set. First the data set is separated into several parts evenly. Then the rest of this process is composed of several similar rounds (also call *folders*). In each round, one part of the data set is left out as the test set and the rest parts are used as training set; then some statistics can be obtained. When all the rounds are finished, all the statistics will be synthesized to get the final statistic result for the testing object.

For testing a trainer, in each round a new classifier is created on the training set and test on the test set. While for testing a classifier, only test sets are used to test it and the training set is not used.

Kohavi, Ron (1995). "A study of cross-validation and bootstrap for accuracy estimation and model selection". Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence 2 (12): 1137–1143.(Morgan Kaufmann, San Mateo, CA)

18.4.1. Requirements

```
#include "CrossValidate.h"
using namespace LibEDM;
```

18.4.2. Members

Template Function	
CrossValidate	Remove all leading spaces (including control characters).
Class for Return	
StatisticStr	User-provided class for return, which must can be constructed from prediction result of each round..

18.4.2.1. CrossValidate

```
//test a trainer
template <class StatisticStr> bool CrossValidate(int FoldNum, const CDataset &DataSet,
     CreateFunc *Creator, const void *Param, vector<StatisticStr> &Statistics)
//test a classifier
template <class StatisticStr> bool CrossValidate(int FoldNum, const CDataset &DataSet,
     const CClassifier *Classifier, vector<StatisticStr> &Statistics)
```

Parameters

StatisticStr

User-provided class for return, which must can be constructed from prediction result of each round. It should contain statistics that user want to obtain in each round of testing, such as accuracy and time for predicting, etc.

FoldNum

Number of folders for cross validating .

DataSet

Data set use to test.

Creator

Entry of trainer for training new classifiers.

Params

Parameters used by trainer.

Classifier

Classifier for testing.

Statistics

Set of the statistics obtained from each of the round.

Return Value

True upon success; false otherwise.

Remarks

After calling the function, only statistics for each round is obtained, so user need to do the rest (syntheses).

18.4.2.2. class StatisticStr

```
class StatisticStr
{
    StatisticStr:: StatisticStr(const CDataset &Data,const CPrediction *Prediction);
    ...
}
```

Parameters

Data

The test set of each round. In each round of cross validation, there is test set and the classifier will be tested on it.

Prediction

The prediction result of this classifier on the test set.

Return Value

Remarks

The cross validating function will call this construction to create an object of this class, which will be return to user.

18.4.3. Example for Cross Validating (Example6)

18.4.3.1. Source Files

- Example6.cpp: Test RPROP Back Propogation Neural Network trainer on data set *zoo* by three-folder cross validating.

18.4.3.2. How to compile?

- Windows: Open examples/CrossValidate.vcpro in MSVC 2005 or higher version
- Linux: type "make CrossValidate" in the examples directory of LibEDM

18.4.3.3. Description

This example does a three-round cross validation on data set *zoo* for RPROP BPNN trainer with default parameters.

To use cross validating of LibEDM, user has to define a return class at first, which must have a construction that constructing from the prediction result of the classifier (or the trainer) being tested. In this example, we first define a class that records predictive accuracy of each round:

```
class StAccuracy
{
```

```
public:
    StAccuracy(const CDataset &Data,const CPrediction *Prediction)
    {
        Accuracy=Prediction->GetAccuracy();
    };

    double GetResult(){return Accuracy;};

private:
    double Accuracy;
};
```

This class just save and return the predictive accuracy on the test set of each new classifier, which is created by the trainer in each round.

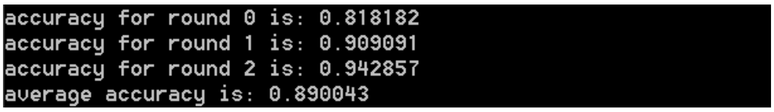
Then we can call the CrossValidate template class by using StAccuracy as parameter:

```
//Evaluate BPNN trainer with default parameters through cross validation
//We want the accuracy of the trainer
vector<StAccuracy> Accuracys;
double AverageAccuracy=0;
//three folders
int Cross=3;
CrossValidate<StAccuracy>(Cross,TrainSet,CBpnn::RpropCreate,NULL,Accuracys);
```

After the cross validating process finished, statistics (only accuracy in this example) of each round is put in the array Accuracys, so we can get the final result (the average accuracy of all rounds):

```
//get average accuracy of each folder
for(int i=0;i<Cross;i++)
{
    double Acc=Accuracys[i].GetResult();
    cout<<"accuracy for round "<<i<<" is: "<<Acc<<endl;
    AverageAccuracy+=Acc;
}
AverageAccuracy/=Cross;
cout<<"average accuracy is: "<<AverageAccuracy<<endl;
```

The output of this example looks like:



```
accuracy for round 0 is: 0.818182
accuracy for round 1 is: 0.909091
accuracy for round 2 is: 0.942857
average accuracy is: 0.890043
```

18.5. Date and Time Manipulation

```
class CDateTime
```

Date and time manipulation is commonly use function, but there is not a cross-platform c++ class for both MS-Windows and Linux (UNIX). That's why the CDateTime class is implemented in LibEDM.

Simply CDateTime is just the encapsulation of some selected date and time functions from the standard c programming language. It can only represent a date after 1970-01-01 00:00:00. For more information reference the manual of c language.

18.5.1. Requirements

```
#include <ctime>
#include <cstring>
#include <string>
using namespace std;
#include "zString.h"
#include "DateTime.h"
using namespace LibEDM;
```

18.5.2. Members

Construction / Destruction	
CDateTime	Construct a CDatetime object.
Now	Creates a CDateTime object represents the current date and time.
Today	Creates a CDateTime object represents the current date.
Operators	
+, -	Add and subtract CDateTime values.
+=, -=	Add and subtract CDateTime values from this CDateTime object.
==, !=, >, >=, <, <=	Compare two CDateTime objects.
Operations	
FormatDateTime	Generates a formatted string representation of a this object.
FormatDate	Generates a formatted string representation date part of a this object.
FormatTime	Generates a formatted string representation time part of a this

	object.
Attributes	
GetYear	Returns the year this object represents.
GetMonth	Returns the month this object represents (1-12).
GetDay	Returns the day this object represents (1-31).
GetHour	Returns the hour this object represents (0-23).
GetMinute	Returns the minute this object represents (0-59).
GetSecond	Returns the second this object represents (0-59).

18.5.2.1. CDateTime::CDateTime

Concstructs a CDateTime object.

```
CDateTime();
CDateTime(int Year, int Month, int Day, int Hour, int Minute, int Second);
CDateTime(int Year, int Month, int Day);
//Construct by providing the seconds elapsed since 1970-1-1 00:00:00.
CDateTime(time_t Time);
//Construct by providing a string representing the time
CDateTime(const string &TimeStr);
```

Parameters

Year, Month, Day, Hour, Minute, Second

Date and time values for the new CDateTime object to be created, must after or equal to the epoch time (1970-1-1 00:00:00).

Time

Seconds elapsed since the epoch time, must >=0.

TimeStr

A string representing the time, must after or equal to the epoch time.

Return Value

Remarks

CDateTime can only manipulate date after 1970-01-01 00:00:00.

18.5.2.2. CDateTime::operator +, -

Add and substract CDateTime values.

```
CDateTime operator +(const CDateTime &Time);
CDateTime operator -(const CDateTime &Time);
```

Parameters

Time

CDateTime value will be add/subtract.

Return Value

A new CDateTime object whose value is calculate by add/subtract the input value to/from this object.

Remarks

The result data time must be after the epoch time, or an exception will be thrown out.

18.5.2.3. CDateTime::operator +=, -=

Add and substract CDateTime values to/from this object.

```
CDateTime &operator +=(const CDateTime &Time);
CDateTime &operator -=(const CDateTime &Time);
```

Parameters

Time

CDateTime value will be add/subtract from this object.

Return Value

This CDateTime object.

Remarks

The result data time must be after the epoch time, or an exception will be thrown out.

18.5.2.4. CDateTime::operator ==, !=, >, >=, <, <=

Compares values of two CDateTime object.

```
bool operator ==(const CDateTime &Time) const;
bool operator !=(const CDateTime &Time) const;
bool operator >(const CDateTime &Time) const;
bool operator >=(const CDateTime &Time) const;
bool operator <(const CDateTime &Time) const;
bool operator <=(const CDateTime &Time) const;
```

Parameters

Time

CDateTime obeject to be compared with this object.

Return Value

A boolean value indicates the camparison result.

Remarks

18.5.2.5. CDateTime::FormatDateTime

Create a string respresents this CDatetime object.

```
string FormatDateTime() const;
```

Parameters

Return Value

A string respresents this CDatetime object.

Remarks

The return string is in the format like “1970-01-01 00:00:00”.

18.5.2.6. CDateTime::FormatDate

Create a string respresents the date value of this CDatetime object.

```
string FormatDate() const;
```

Parameters

Return Value

A string respresents the Date value of this CDatetime object.

Remarks

The return string is in the format like “1970-01-01”.

18.5.2.7. CDateTime::FormatTime

Create a string respresents the time value of this CDatetime object.

```
string FormatTime() const;
```

Parameters

Return Value

A string respresents the time value of this CDatetime object.

Remarks

The return string is in the format like “23:59:59”.

18.5.2.8. CDateTime::GetYear

Create the year represented by this CDatetime object.

```
int GetYear();
```

Parameters

Return Value

A integer >= 1970.

Remarks

18.5.2.9. CDateTime::GetMonth

Create the month represented by this CDatetime object.

```
int GetMonth();
```

Parameters

Return Value

A integer between 1 and 12.

Remarks

18.5.2.10.CDateTime::GetDay

Create the day represented by this CDatetime object.

```
int GetDay();
```

Parameters

Return Value

A integer between 1 and 31.

Remarks

18.5.2.11.CDateTime::GetHour

Create the hour represented by this CDatetime object.

```
int GetHour();
```

Parameters

Return Value

A integer between 0 and 23.

Remarks

18.5.2.12.CDateTime::GetMinute

Create the minute represented by this CDatetime object.

```
int GetMinute();
```

Parameters

Return Value

A integer between 0 and 59.

Remarks

18.5.2.13.CDateTime::GetSecond

Create the second represented by this CDatetime object.

```
int GetSecond();
```

Parameters

Return Value

A integer between 0 and 59.

Remarks

18.6. String Manipulation

```
namespace CzString
```

In CzString, some functions are implemented for the string class of STL.

18.6.1. Requirements

```
#include <cstdlib>
#include <cstdio>
#include <string>
using namespace std;
#include "zString.h"
using namespace LibEDM;
```

18.6.2. Members

Trim	
TrimLeft	Remove all leading spaces (including control characters).
TrimRight	Remove all trailing spaces.
Trim	Remove all leading and trailing spaces.
Convention	
ToInt	Return a integer represented by the string.
ToDouble	Return a double number represented by the string.
DoubleToStr	Return a string representing a double number.
IntToStr	Return a string representing a decimal integer number.
IntToBinStr	Return a string representing a binary number.
Split	Return the first word of a string.

18.6.2.1. CzString::Trim, TrimLeft, TrimRight

Remove all leading and/or trailng spaces (including control characters).

```
void Trim(string &Str);
void TrimLeft(string &Str);
void TrimRight(string &Str);
```

Parameters

Year, Month, Day, Hour, Minute, Second

Date and time values for the new CDateTime object to be created, must after or equal to the epoch time (1970-1-1 00:00:00).

Time

Seconds elapsed since the epoch time, must >=0.

TimeStr

A string representing the time, must after or equal to the epoch time.

Return Value

Remarks

CDateTime can only manipulate date after 1970-01-01 00:00:00.

18.6.2.2. CzString::ToInt

Return a integer represented by the string.

```
int ToInt(const string &Str);
```

Parameters

Str

The string represents a integer number.

Return Value

An integer.

Remarks

This function is encapsulation of atoi() function in c programming language, see c manual for details.

18.6.2.3. CzString::ToDouble

Return a double number represented by the string.

```
int ToDouble(const string &Str);
```

Parameters

Str

The string represents a double number.

Return Value

An double.

Remarks

This function is encapsulation of atof() function in c programming language, see c manual for details.

18.6.2.4. CzString::IntToStr

Return a string represents the integer number in decimal.

```
string IntToStr(int a);
```

Parameters

Str

The string represents a integer number.

Return Value

A string represent the integer number.

Remarks

Maximum 1023 characters allowed in the output string.

18.6.2.5. CzString::IntToBinStr

Return a string represents the integer number in binary.

```
string IntToBinStr(int a,int Digital=0);
```

Parameters

Str

The string represents a integer number.

Digit

Minimum length of the output string.If number of characters in the output string is less than the specified length, zeros are added to the left until minimum length is reached.

Return Value

A string represent the integer number in binary.

Remarks

Maximum 1023 characters allowed in the output string.

18.6.2.6. CzString::DoubleToStr

Return a string represents the double number.

```
string IntToStr(double a);
```

Parameters

Str

    The string represents a double number.

Return Value

A string represent the double number.

Remarks

Maximum 1023 characters allowed in the output string, precision is fixed to six.

18.6.2.7. CzString::Split

Return the first word in a string with the given delimiter.

```
const string Split(const string &Src,char Del,string &Word);
```

Parameters

Src

    The string which to find a word from.

Del

The delimiter character to distinguish each word.

Word

The first word if found.

Return Value

The rest of the input string after removing the first word and the succeeding delimiter. If no character (except delimiters) left in the string, a empty string is returned.

Remarks

18.7. Statistical Comparison for Multiple Machine Learning Methods

```
class CStat
```

In research of machine learning, there are always requirements to compare more multiple learning methods from the facets of classifiers creating speed, prediction accuracy and prediction speed etc. And people also want to know how reliable the comparison results are, so some statistic methods are presented. In LibEDM two powerful statistical methods are presented, they are Friedman test and Bergmann and Hommel test.

For both these two tests, multiple (*N*) data sets are suggested to be used to improve the accuracy of the comparisons. And the ranks (*R*) of all methods (*k*) on the criteria (often is prediction accuracy) not the criteria itself is used to calculate the statistics.

Friedman test is to determine whether there is significant difference among all compared methods.

To use Friedman test, first set the null hypothesis by assuming that all compared methods are equivalent, then compute the Friedman statistics by calling CStat::Ff():

$$\chi_F^2 = \frac{12N}{k(k+1)}\left[\sum_j R_j^2 - \frac{k(k+1)^2}{4}\right]$$

$$F_F = \frac{(N-1)\chi_F^2}{N(k+1)-\chi_F^2}$$

Because this statistic is distributed according to F distribution, so by inspecting the F-table we can know whether the null hypothesis can be accepted.

If the null hypothesis is rejected in Friedman test, Bergmann and Hommel test can be performed in the next step to compare all the methods one-to-one (All Pairwise Comparisons). To compare any two methods (*i-th* and *j-th* method e.g.) in BH test, first set the null hypothesis by assuming that the *i-th* and *j-th* methods are equivalent, next calculate this statistic:

$$z = \frac{R_i - R_j}{\sqrt{\frac{k(k+1)}{6N}}}$$

z is distributed according to Gauss (normal) distribution so the corresponding probability *p* can be got, *p* is the just probability that the null hypothesis can be accepted. In BH test *p* will be adjusted in the next steps to maintain the consistency between all comparisons, so we get *adjusted-p*.

18.7.1. Requirements

```
#include <set>
#include <vector>
#include <map>
#include <algorithm>
#include <cmath>
#include <cassert>
using namespace std;
#include "Statistic.h"
using namespace LibEDM;
```

18.7.2. Members

Statistical Tests	
BH	Bergmann & Hommel test.
Ff	Calculate statitic for the Friedman test.
Basic Statistical Functions	
Multip	Return the factorial for a number.
Gauss	Return the Gauss probability.

rGauss	Given a probability, return the
Data Structures	
RankStr	Statistical result for a comparison.

18.7.2.1. CStat::Ff

Perform a Friedman test.

```
static double Ff(vector<double> &Ranks, int N, double &Skf);
```

Parameters

Ranks

Input. The average ranks for all methods on all comparing data sets.

N

Input. Number of data sets used to compare all the methods.

Skf

OutPut. Value of χ_F^2.

Return Value

Value of F_F, it is distributed according to F-distribution. Check F-table to see if all comparing methods are equivalent (under a preset significance level).

Remarks

18.7.2.2. CStat::BH

Perform a Bergmann & Hommel test.

```
static void BH(int N, vector<double> &Ranks, vector<RankStr> &table);
```

Parameters

N

Input. Number of data sets on which all the methods are compared.

Ranks

Input. The average rank of each comapring method for all data sets.

table

OutPut. One-to-one comparison results for all comparing methods. See details on data structures description.

Return Value

Remarks

This is a recursion implementation so it may be slow and memory-consuming. At present NO more than nine methods are supported in comparison.

18.7.2.3. CStat::Mutilp

Calculate factorial of the given number.

```
static double Multip(int n);
```

Parameters

n

Input. A non-negative integer number.

Return Value

Remarks

18.7.2.4. CStat::Gauss

Caculate the probability for a Gauss(Normal) distributed random varible.

```
static double Gauss(double x);
```

Parameters

n

Input. Value of a random variable.

Return Value

The probability if this random varible is distributed according to Gauss distribution.

Remarks

18.7.2.5. CStat::rGauss

Given a probability, get the non-negative value for a Gauss(Normal) distributed random varible.

```
static double rGauss(double p);
```

Parameters

p

Input. Probability for this random variable.

Return Value

A non-negative value for the random varible.

Remarks

18.7.3. Data Structures

RankStr	Statistical result for a comparison.

18.7.3.1. RankStr

Statistical result for a one to one comparison. If we assume that these two methods are equvalent (the null hypothesis), the probability that the hypothesis can be accepted.

```
typedef struct RankStr
{
    int  vs;//the two methods being compared
    double z;//z statistic
    double p;//probability
    double APV;//adjusted probability
    double alpha;//adjusted alpha
}RankStr;
```

Fields

vs

Combined id for the two methods being compared. Each the decimal digit in this integer number represent an id of a method being compared, i.e. 1 for the first method, 2 for the 2nd method,...,9 for the nineth. So maximally nine methods can be compared at the same time.

z

Value for z statistic.

p

Probability that the two methods are equvalent.

APV

Adjusted probability of BH test. Probability of all the null hypotheses may need be adjusted to avoid the inconsistency among them.

alpha

Unused.

Remarks

Why adjust the probabilities? Let’s assume there are three methods being compared: C1, C2 and C3. So there will be three null hypotheses: H1=”C1 and C2 are equivalent”, H2=”C2 and C3 are equivalent” and H3=”C3 and C1 are equivalent”. Under some circumstance there may be such situation: H1 and H2 are accepted but C3 is rejected, so there is inconsistency among the null hypotheses. APV can avoid this situation.

18.7.4. Example for Statistical Test (Example5)

18.7.4.1. Source Files

- Example5.cpp: Friedman test and BH test for an example test result.

18.7.4.2. How to compile?

- Windows: Open examples/Statistic.vcpro in MSVC 2005 or higher version
- Linux: type “make statistic” in the examples directory of LibEDM

18.7.4.3. Description

In a typical experiment if we want to compare many ensemble pruning methods, we will build ensembles on many (more than 20) data sets at first and on each of the ensemble all the pruning methods we want to compare will be performed. Then the performance (i.e. the prediction accuracy to the test set) of all the pruning methods will be recorded for each data set (ensemble) and each pruning method is ranked according to its performance. Usually all the phases mentioned above have to be repeated several (10 to 100) times to eliminate the randomness.

Now assume we have to compare performance (accuracy) of six pruning methods on 30 different data sets. After obtaining the accuracy data for all the methods, we also want to know the liability of each conclusion if we compare two methods base on their corresponding prediction accuracy. This is the situation where Friedman test and Bergmann-Hommel test can be used.

To perform these tests, first we need rank all the pruning methods according to their prediction accuracy on every experimental data set, next the average ranks of all methods on all the data set will be the input of the statistical tests. During the ranking process if two or more methods rank the same, all these methods should equally share the sum of the original rank number. For example if two methods all rank 2, the real rank number given to both of them should be (2+3)/2=2.5. This adjustment ensures all the data sets have same influence on the final result.

File *rank.txt* contain an example rank table for 6 methods (M1 to M6) on 30 data sets (D1 to D38):

	M1	**M2**	**M3**	**M4**	**M5**	**M6**			**M1**	**M2**	**M3**	**M4**	**M5**	**M6**
D1	5	4	3	1	6	2		**D20**	2	6	3	4	5	1
D2	6	3	5	2	1	4		**D21**	2	2	6	5	4	2
D3	6	3	3	3	1	5		**D22**	6	4	2	1	5	3
D4	6	5	3	1	4	2		**D23**	5	4	1.5	3	1.5	6
D5	6	4	2	1	5	3		**D24**	5	3	3	1	6	3
D6	6	3.5	5	2	3.5	1		**D25**	5	3	2	1	6	4
D7	5.5	5.5	3	1.5	1.5	4		**D26**	6	3	5	2	1	4
D8	6	4	3	2	1	5		**D27**	4.5	6	1	2	3	4.5
D9	6	3	5	2	1	4		**D28**	2	1	4	4	4	6
D10	2	3	6	5	4	1		**D29**	4	5	2	1	6	3
D11	5	5	1	2	3	5		**D30**	2.5	4	1	6	5	2.5
D12	6	2	5	1	3	4		**D31**	4.5	4.5	3	2	6	1
D13	6	5	2	1	3	4		**D32**	6	5	3	2	1	4
D14	5	2	1	3.5	6	3.5		**D33**	2.5	2.5	5.5	5.5	2.5	2.5
D15	3.5	3.5	3.5	3.5	3.5	3.5		**D34**	5	3.5	3.5	1.5	6	1.5
D16	5	3	5	1.5	1.5	5		**D35**	5	4	2	1	6	3
D17	6	4.5	2	3	1	4.5		**D36**	6	3	5	2	1	4
D18	6	3	5	2	1	4		**D37**	5.5	4	2	1	5.5	3
D19	4	4	1	4	6	2		**D38**	4	5	1.5	1.5	6	3

In this example (example5.cpp) we first read this table from file:

```
//open rank file
ifstream DataFile;
DataFile.open("Ranks.txt");
if(DataFile.fail())
{
    cout<<"Rank.txt not found!"<<endl;
    return 1;
}
//buffer to analyze a line of input file
char buf[8192];
```

```
//Ranks of each methods on all data sets
vector<DoubleArray> AllRanks;
while(!DataFile.eof())
{
    //read a line
    DataFile.getline(buf,sizeof(buf));
    if(DataFile.fail())
    {
        if(DataFile.eof())
            continue;
        cout<<"error reading!"<<endl;
        return 1;
    }

    //read data from the line
    basic_istringstream<char> DataLine(buf);
    //rank of all methods on this data set
    vector<double> Ranks;
    //number of ranks has read
    while(!DataLine.eof())
    {
        //read a value
        string Word;
        DataLine>>Word;
        //read failed
        if(DataLine.fail())
            break;

        //read a rank number
        double Rank=CzString::ToDouble(Word);
        Ranks.push_back(Rank);
    }
    AllRanks.push_back(Ranks);
}
DataFile.close();
```

Then we get the average rank for each method on all 30 data sets.

```
//number of pruning methods
int MethodNum=(int)(AllRanks[0]).size();
//number of data set
int DatasetNum=(int)AllRanks.size();
//each algorithm: average ranks for all datasets
vector<double> AvgRanks(MethodNum,0);
for(k=0;k<DatasetNum;k++)
    for(i=0;i<MethodNum;i++)
```

```
        AvgRanks[i]+=AllRanks[k][i];
for(i=0;i<MethodNum;i++)
    AvgRanks[i]/=DatasetNum;
```

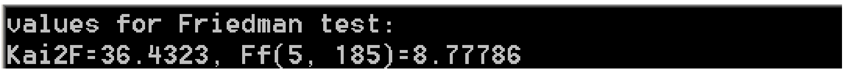
Next we perform the Friedman test:

```
//Friedman test
cout<<"values for Friedman test:"<<endl;
double Skf;
double Ff=CStat::Ff(AvgRanks,DatasetNum,Skf);
cout<<"Kai2F="<<Skf<<", Ff("<<AvgRanks.size()-1<<",
    "<<(AvgRanks.size()-1)*(DatasetNum-1)<<")="<<Ff<<endl;
```

Return of function CStat::Ff() is the value for Friedman test (FF) statistic, which is distributed according to the F-distribution. Through inspecting the F-table, we can know the probability for the null hypothesis:

H=”All the comparing methods are equivalent”.

By a given confidence level we can know if to reject the hypothesis.The output of Friedman test looks like:

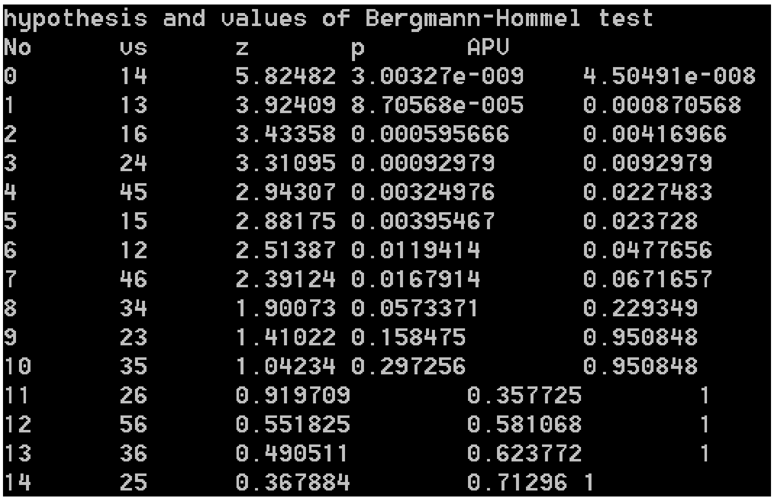


```
values for Friedman test:
Kai2F=36.4323, Ff(5, 185)=8.77786
```

Next is the BH test (Actually BH is unnecessary if the above hypothesis is not rejected):

```
//BH test
vector<RankStr> Tab;
CStat::BH(DatasetNum,AvgRanks,Tab);
//Bergmann-Hommel test(all pairwise tests)
cout<<endl<<"hypothesis and values of Bergmann-Hommel test";
cout<<endl<<"No\tvs\tz\tp\tAPV"<<endl;
for(i=0;i<(int)Tab.size();i++)
    cout<<i<<"\t"<<Tab[i].vs<<"\t"<<Tab[i].z<<"\t"<<Tab[i].p<<"\t"<<Tab[i].APV<<endl;
```

BH test actually contains a serial of tests although they are performed at the same time. The return of BH test is an array of the structure *RankStr*. It contains null hypotheses for all possible pair-wise comparisons and their corresponding possibility. The output of BH test looks like:



```
hypothesis and values of Bergmann-Hommel test
No      vs      z       p       APV
0       14      5.82482 3.00327e-009    4.50491e-008
1       13      3.92409 8.70568e-005    0.000870568
2       16      3.43358 0.000595666     0.00416966
3       24      3.31095 0.00092979      0.0092979
4       45      2.94307 0.00324976      0.0227483
5       15      2.88175 0.00395467      0.023728
6       12      2.51387 0.0119414       0.0477656
7       46      2.39124 0.0167914       0.0671657
8       34      1.90073 0.0573371       0.229349
9       23      1.41022 0.158475        0.950848
10      35      1.04234 0.297256        0.950848
11      26      0.919709        0.357725        1
12      56      0.551825        0.581068        1
13      36      0.490511        0.623772        1
14      25      0.367884        0.71296 1
```

It is actually a table shows null hypotheses and probabilities (for six comparing methods, there are 15 pair-wise comparisons maximally):

No	**vs** **(Null hypothesis)**	**z** **(Statistic)**	**p** **(Probability)**	**APV** **(Adjusted probability)**
0	1 vs. 4	5.82482	3.00327e-009	4.50491e-008
1	1 vs. 3	3.92409	8.70568e-005	0.000870568
2	1 vs. 6	3.43358	0.000595666	0.00416966
3	2 vs. 4	3.31095	0.00092979	0.0092979
4	4 vs. 5	2.94307	0.00324976	0.0227483
5	1 vs. 5	2.88175	0.00395467	0.023728
6	1 vs. 2	2.51387	0.0119414	0.0477656
7	4 vs. 6	2.39124	0.0167914	0.0671657
8	3 vs. 4	1.90073	0.0573371	0.229349
9	2 vs. 3	1.41022	0.158475	0.950848
10	3 vs. 5	1.04234	0.297256	0.950848
11	2 vs. 6	0.919709	0.357725	1
12	5 vs. 6	0.551825	0.581068	1
13	3 vs. 6	0.490511	0.623772	1
14	2 vs. 5	0.367884	0.71296	1

The second column is the null hypotheses: “1 vs. 4” means M1 and M4 are equivalent, “4 vs. 5” means M4 and M5 are equivalent, etc. The third and fourth columns are values for the z-statistic and probabilities for the null hypotheses correspondingly. The last column is the adjusted probabilities (of BH test) for the null hypotheses, by which all these hypotheses are ascendingly ordered. Given a significant level α=0.05 as example, we can accept the following null hypotheses:

2 vs. 5, 2 vs. 6, 3 vs. 6 and 5 vs. 6, that means method 2, 5, 6 are equivalent, method 3 and 6 are equivalent, all the others pairs are significant different.

19.LibEDM developers' guides

In this section, we will describe how to support new data file formats, and how to develop base classifiers, ensemble and pruning methods.

19.1. Supporting a New Data-File Format

To support a new file format you must understand how LibEDM reading from data files. First LibEDM will need the data description (i.e. format of data file, attributes and class labels, etc.); then LibEDM can read the real data as the format description and also during this phase LibEDM checks validation of each input value according to the description of attributes and class labels.

All these phases have been assembled in CDataSet, developer only need to override two virtual member functions (*ReadInfo* and *ReadDate*). When overriding these functions, developer can use some member function of class CDataSet to reduce redundant work.

19.1.1. Necessary members

See C4.5 format (CUCIData.h, CUCIData.cpp) as an example.

Overriding Members (CDataSet)	
virtual void ReadInfo(ifstream &InfoFile);	Read data description from an open file stream.
virtual void ReadMatrix(ifstream &DataFile, int Number=0);	Read real data from an open file stream, assuming that the data description has been obtained.
Useful function when implementing (from CDataSet)	
int FormatLine(string &Line) const;	Format one line of a text-based data file. return: *SKIP_LINE* if this line is NULL (or commented) *LINE_OK* if this line is valid and can pass to succeeding analyses.
bool Which(ValueData &Item, const string &Name) const;	Search for the corresponding value for a string of class label. If can not find a match, exception is thrown.

bool Which(ValueData &Item, int AttrNum, const string &Name) const;	Search for the corresponding value for a string of a discrete attribute. If can not find a match, exception is thrown.

19.1.2. Remarks

For ARFF format (CArffData) and C4.5 format (CUCIData), the data description for a data set are different, so both of them need to override member function ReadInfo, but their real data are stored in the same format, so they just use ReadMatrix in CDataSet and don’t have to override it.

19.2. Developing a New Base classifier

To implement a new base classifier, first derive a new class from *CClassifier*, which is the base class for all classifier of LibEDM. *CClassifier* is an abstract class and it has four virtual member functions all of which needed to be overridden. There are also several static member function need to be implemented to support embedding this type of base classifier into ensembles.

19.2.1. Necessary members

Here Back-Propagation Neural Network (Bpnn.h, Bpnn.cpp) is taken as an example.

Overriding Members	
virtual CPrediction *Classify(const CDataset &DataSet) const;	Use this classifier to predict a data set.
virtual inline ~CClassifier()=0;	Destructor
virtual int Save(const string &Path, const string &FileName) const;	Save this classifier info disk files.
virtual bool Dump(const string &FileName) const;	Dump inside data for inspecting.
Static Members	
static CClassifier *FileCreate(const string &Path, const string &FileName)	Used by ensembles, load a base classifier of this type from archive files. Will be registered to an ensemble creator.
static CClassifier *Create(const CDataset &TrainData, const void*	Used by ensembles, create a classifier of this class by using given parameters. Will be registered to an ensemble creator.

UParams)	
static string GetStaticName()	Used by users, name for this type of classifiers. When register a file-creator for this type of classifier to the ensemble creator, must provide its internal type name.

19.2.2. Remarks

The overriding members are basic functions of a classifier, which must be implemented. The static members are necessary functions need by users who want to create ensembles automatically. User can register a classifier class's *FileCreate* and *Create* function (with corresponding parameters) to an ensemble class (CBagging or CAdaBoost e.g.), then the ensemble creator can automatically call the base classifier's creator and creating corresponding type of base classifiers (from data files or archived files).

For creating classifier from files, each archive file records an internal type name for the archived classifier. And the ensemble's file-creating creator relies on this type name to find a matching classifier creator to recover the classifier, so be careful that no classifier class in LibEDM shares the same internal type name (same return for function *GetStaticName*).

19.3. Developing a New Incremental Classifier

Implementing a new incremental base-classifier/ensemble is very similar to implementing a new base-classifier/ensemble, except that two extra virtual functions from class *CIncrementalClassifier* should be overridden.

19.3.1. Necessary members

See Gauss-Distribution based Naive Bayes (*GaussNaiveBayes.h, GaussNaiveBayes.cpp)* and SEA incremental ensemble (*CSEA.h, CSEA.cpp*) as examples.

Overriding Members (CIncrementalClassifier)	
virtual void Train(const CDataset &Dataset)=0;	Incremental training.
virtual void Reset()=0;	Reset the incremental classifier to an un-trained state.

19.3.2. Remarks

19.4. Developing a New Ensemble Method

An ensemble is a set of classifiers (called base classifiers), but all these base classifiers can work together, so from users' point an ensemble is just like a normal classifier.

Different ensemble methods vary in the way creating base classifiers and combining their predictions. To add a new ensemble method to LibEDM, a new class has to be derived from CEnsemble, which is the base class for all ensembles and also derived from CClassifier.

The new ensemble should put all its base classifiers into *Classifiers*, which is a member variable inherited from *CEnsemble*. And it should put the corresponding weights for all base classifiers into *Weights* (which is also a member of *CEnsemble*) if the new ensemble use a (weighted) voting-based predicting, which is already implemented in LibEDM. The creating process for the new ensemble should to be implemented in the constructor of the new class.

CEnsemble presents two most common ensemble-based predicting algorithms, i.e. voting and weighted voting, which are used by Bagging, AdaBoost and some other ensemble methods. If the new ensemble method uses a different predicting method, all forms of the member function *classify*() from CEnsemble need to be overridden, otherwise it can use the predicting methods provided by CEnsemble.

CEnsemble overrides class CClassifier's member function *save*() and *Dump*(), which save base classifiers and their corresponding weights in disk files. If the new ensemble doesn't use (weighted) voting, it should also override these member functions.

19.4.1. Necessary members

Here Bagging ensemble (Bagging.h, Bagging.cpp) is taken as an example:

Overriding Members (Override only if new ensemble not using voting or weighted voting)	
`vector<CClassifier*> Classifiers`	Base classifiers for this ensemble.
`vector<double> Weights`	Weights for all base classifiers (Valid if the ensemble uses voting or weighted voting).

virtual CPrediction *Classify(const CDataset &DataSet) const;	Use this ensemble to predict a data set.
virtual CPrediction *Classify(const CDataset &DataSet, const vector<CPrediction*> &Predictions) const	Ensemble predicting if we already have each base classifier's prediction of the input data set.
virtual CPrediction *Classify(const CDataset &DataSet, const vector<double> &UserWeights) const	Ensemble predicting using user defined weights vector. A zero weight in the vector means the corresponding base classifier doesn't participate in the predicting (as if it is removed).
virtual CPrediction *Classify(const CDataset &DataSet, const vector<CPrediction*> &Predictions, const vector<double> &UserWeights) const	Ensemble predicting using user defined weights vector if we already have each base classifier's prediction of the input data set.
virtual inline ~CClassifier()=0;	Destructor. All base classifiers should be destroyed here.
virtual int Save(const string &Path, const string &FileName) const;	Save this classifier info disk files.
virtual bool Dump(const string &FileName) const;	Dump inside data for inspecting.
Members	
CBagging()	Create a set of base classifiers for this ensemble.

19.4.2. Remarks

If new ensemble uses (weighted) voting, only the constructor need to be implemented, which is used to create all base classifiers for the ensemble. If new ensemble uses a different predicting method, override member function *classify*(). Overriding save() or Dump() if the ensemble also has a internal organization different from just vector of weights.

An ensemble is in charge of all of its base classifiers. A base classifier can not be destroyed outside the ensemble. Base classifiers are destroyed as the destruction of the ensemble, so a base classifier can not belong to more than one ensemble.

19.5. Developing a New Ensemble Pruner

A pruner is to refine the contents of an ensemble, which removes the base classifiers that are redundant or have negative effect on the performance of the ensemble, thus improves performance and efficiency of the ensemble at the same time.

In LibEDM a pruner doesn't really remove base classifiers from an ensemble. Base classifiers are managed by the ensemble it belongs to. A pruner just mark a base classifier it wants to remove as "Removed", so this base classifier can not participate in the ensemble predicting.

To add a new pruner to LibEDM, a new class should be derived from *CEnsemblePruner*. The pruning process should be implemented in the constructor of the new class. After pruning, *CEnsemblePruner::Weights* should be initialized with an array of double values, whose size should equal to the size of the ensemble. A zero value in the weight vector means the corresponding base classifier is removed from the ensemble, which should not participate in the predicting; any other value mean base classifier will participate in the ensemble predicting.

CEnsemblePruner is also an abstract class and you should avoid instancing from it directly. A new pruner class should instance a *CEnsemblePruner* object in its constructor's initialization table with the original ensemble object as the input parameter, because the new pruner needs to access the member of CEnsemblePruner, i.e. Weights and Ensemble.

19.5.1. Necessary members

See FS.h (Forward Selection) for example:

Overriding Members	
virtual CPrediction *Classify(const CDataset &DataSet) const;	In most situations, these functions need not be overridden.
virtual inline ~CClassifier()=0;	
virtual int Save(const string &Path, const string &FileName) const;	
virtual bool Dump(const string &FileName) const;	
Members	
CForwardSelect(const CEnsemble &Ensemble, const CDataset &ValidatingSet)	Pruning the ensemble base on a validation data set.
CForwardSelect(const CEnsemble &Ensemble, const CDataset &ValidatingSet, const vector<CPrediction*> &Predictions)	Pruning the ensemble base on a validation data set, if we already have each base classifier's prediction of the data set. This is useful if you have an ensemble wanted to be pruned by several different pruner.

19.5.2. Remarks

The new pruner class has to instance a CEnsemblePruner object in its initialization table, because the new class needs to access inherited member of its parent class, but CEnsemblePruner is an abstract class thus can not be instanced directly. See examples from Forward Selection (FS.cpp):

```
CForwardSelect::CForwardSelect(const CEnsemble &UEnsemble,const CDataset &ValidatingSet)
:CEnsemblePruner(UEnsemble)
{
...
Here access protected member of CEnsemblePruner.
...
}
...
CForwardSelect::CForwardSelect(const CEnsemble &UEnsemble,const CDataset &ValidatingSet,const
vector<CPrediction*> &Predictions)
:CEnsemblePruner(UEnsemble)
{
......
}
```