

Adaptive Filtering Software: User Guide

Updated October 13, 2004

Contents

1. [Preparing the corpus](#)
2. [Interpreting judgment files](#)
3. [Parameter file](#)
 - [Parameters file syntax](#)
 - [Classification model selection](#)
 - [Location and interpretation of input files](#)
 - [Output files and output format](#)
 - [Random number generation](#)
 - [Performance tuning](#)
 - [Classifier parameters common to all vector-sum based models, i.e. both Rocchio and Centroid](#)
 - [Effectiveness measure \(used in thresholding and final eval\)](#)
 - [Classifier parameters \(specific to Rocchio model\)](#)
 - [Classifier parameters \(specific to Centroid model\)](#)
 - [Classifier parameters \(specific to kNN classifier\)](#)
 - [Classifier parameters \(specific to Inverted Model Classification\)](#)
4. [Running the program](#)
 - [Changing the run directory](#)
5. [Output files](#)
 - [Individual judgments](#)
 - [Contingency tables](#)
6. [Miscellaneous utilities](#)
 - [Lemur-to-SVM convertor](#)
 - [Results processing script](#)
 - [Topic dictionary tool \(MkDic\)](#)
 - [Prior knowledge extraction tool \(MkPrior\)](#)
 - [Threshold retuning tool \(Retune\)](#)

Preparing the corpus

The Adaptive Filtering Software uses Lemur to pre-process input data. A number of small sample data sets are found in subdirectories under the directory `data`. To see how you can create your own data set, let us take a look at one very

small sample data set, the one in directory `data/mini1`. (There is also another very small data set, in `data/mini2`, and a larger one in `data/orig-cacm-full`.) Please refer to [Appendix 1, section "Formatting Your Corpus for Lemur"](#), for the description of the files found in that directory.

The user supplies the training set and the test set in the form of SGML documents ("TREC SGML format"). Then two Lemur's utilities are used to convert the data to the format used by our software: first, `ParseToFile` to convert from the TREC SGML format to Lemur's own SGML format. Then, `BuildBasicIndex` is used to build two sets of files in the Lemur's internal index format. One of these two file sets represents all documents from the initial training set; the other represents all documents from both training and test sets.

We provide a sample script `data/prepare.sh` that can be used to do some of the data preparation. This is what it does:

- Copies the data files, from a specified *data source subdirectory* to the *run directory* of your choice;
- Copies Lemur and Adaptive Filtering parameter files not provided in the data source subdirectory from the `filterparam` subdirectory to the run directory;
- Converts the data files copied to the run directory to the Lemur input format and builds Lemur indexes.

Run this script in the directory where it is located. For example, to take the raw input files from `data/mini1`, and to build Lemur indexes for that corpus in the *run directory* `~/myrun`, you can do this:

```
cd data
./prepare.sh mini1 ~/myrun
```

This will also copy all configuration files from `data/filter_param`.

Note: the run directory must be different from the data source directory.

If you see any error messages such as

```
Exception [by BasicDocStream]: can't open BasicDocStream source file
```

or

```
Segmentation fault
```

it likely means that the Lemur helper applications have failed to find some of the input files they are working with. The output of `prepare.sh` will allow you to figure which Lemur program was first to encounter the problem. Look into the script to see the name of the parameter file given to that program, and then into that parameter file to see the names of the files that the program would expect to open. That should help you to figure what is missing and why.

Running the script as shown above will create Lemur indexes in the run directory (`~/myrun`). Each of the two Lemur indexes consists of several files. In this case, the main file for the training set index will be `~/myrun/train-bindex.bsc`, and for the joint training+test set index will be `~/myrun/all-bindex.bsc`. Each `*.bsc` contains the list of all other files that constitute parts of the Lemur index.

You may want to create your own data preparation script, as appropriate for your corpus, based on `prepare.sh`.

Interpreting judgment files

Before you create a judgment file (or *QREL file*, as they are often called) for your corpus, it is important to understand how Adaptive Filtering Software interprets judgment files. There are two possible modes: *fully judged corpus* and *not fully judged corpus* (default). The choice of mode, which is independent for the two judgment files, is controlled by the flags `train.fullyJudged` and `test.fullyJudged`.

In the **fully judged corpus** mode (flag value is 1), the classifier software assumes that you fully know the category membership of all documents in the document set. If the judgment file lacks an entry for a particular (topic, document) pair, the classifier assumes that the judgment you had in mind is 0 (non-relevant). Therefore, in the fully judged mode you only need to store entries with 1's ("relevant") in them in the judgment file. The two big data sets with which we have experimented a lot, RCV1-v2 and WIPO-alpha (See [Appendix 2](#)) are fully judged data sets.

If `train.fullyJudged=0` (default), the classifier only looks at the lines actually present in the judgment file; if the judgment file contains no line for a particular (topic, document) pair, it is assumed that you have no information as to whether the document is relevant to the topic or not. The corpora distributed to the participants of the [Filtering Track](#) of the TREC competition are not fully-judged; that is, the training set QREL contains only a limited number of positive and negative examples for each topic, and no a priori information is available about any other (topic, document) pair. Pseudo-feedback is often used for training classifiers when the input data are not fully judged.

Parameter file

The operation of the Adaptive Filtering Software is controlled by a large parameter file. Please refer to the sample parameter file [filter_param.txt](#) for the explanation of all parameters. This is a local copy of the file

`data/filterparam/filter_param`.

A number of sample parameter files are supplied in the directory `data/filterparam`:

- `filter_param.rb6` : Rocchio, batch filtering. The best batch Rocchio performance on RCV1-v2 corpus
- `filter_param.ra3` : Rocchio, adaptive filtering. Rocchio params same as in `rb6`. `updatedFreq=10`, `maxUpdateCnt = 1000`.
- `filter_param.paul6` : Centroid, batch filtering. Most parameters similar to `rb6`.
- `filter_param.pa3`: Centroid, adaptive filtering. Parameters same as in `paul6`.
- `filter_param.tr192`: kNN, traditional classifier, `k=192`.
- `filter_param.ya192`: kNN, Yang94 classifier, `k=192`.
- `filter_param.ne192`: kNN, New02a classifier, `k=192`.

Parameters file syntax

For reading convenience, the parameter file consists of a number of section, each one dealing either with some general issue (input files, output files, tracing), or with a particular method (kNN, centroid, Rocchio). You usually need to work only with a number of sections, ignoring those that specifically pertain to methods other than the one you currently use.

The parameter file uses the same syntax as Lemur parameter files. (We use Lemur's parameter-file-parsing code). In each section, there is one parameter definition per line. Each parameter is defined as follows:

```
param_name = param_value ;
```

Here *param_name* is the name of the parameter, which may include alphanumeric characters and dots. *param_value* is the parameter value. As you can see from the sample parameter file, some parameters have numerical values, while others have string values. String-values parameters should normally be surrounded with double quotes:

```
some_param = "some_value";
```

but the quotes can be omitted if the value contains no spaces or special characters.

For better readability, the parameter files may contain blank lines and C-style comments (`/* ... */`).

The remainder of this section explains configuration parameters relevant for the classification methods described in this documentation.

Classification model selection

```
filterModel = 0; /* 0 = Rocchio; 1=Centroid; 2=kNN. No other options yet */
```

```

updateFreq = 1; /* After how many Oracle judgments will the classifier
                be updated. Default is 1 (update after each judgment) */

/* Max number of classifier updates allowed per topic. (-1 = unlimited).
   Use a positive value, e.g. 10000, to cut cost of adaptive filtering. */
maxUpdateCnt = -1;

```

Location and interpretation of input files

```

trainIndex = "train-bindex.bsc"; /* The main file for the training set index,
                                   prepared with LEMUR indexer*/
index = "all-bindex.bsc"; /* The main file for the full database (train+test) index */
textQuery = "query"; /* query text stream */

trainQrelFile= "train-qrel"; /* Qrel data file for TRAIN set */
testQrelFile = "test-qrel"; /* Qrel data file for TEST set */
docOrderFile = "doc-order"; /* TEST set document order file */

/* How should the QREL files be interpreted? 0=all judgments are explicit
   (missing=U); 1=fully judged corpus (missing=N) */
train.fullyJudged=0;
test.fullyJudged=0;

```

Output files and output format

```

/** This tag appears at the end of each output line, if TREC format is used */
runTag="DIMACSTest";

resultFile = "res.filter"; /* result file */
/* How many results to print per batch. This parameter
only control how often the output buffer is flushed;
it does not the actual computation. */
resultCount = 100;
/* 0 = simple-format (3 columns); 1 = TREC-format (6 col);
2=Frادkin's 4-column fmt for fusion experiments */
resultFormat = 1;

/* ResultLogModeType {LOG_ALL=0, LOG_SELECTED=1, LOG_ORACLE=2}; */
resultLogMode = 1;

/*-----
Contingency tables will be printed to these files. Use an empty string (")
as the value to avoid producing the table and save computation time */
/* Training set, scored by classifier after initial training */
trainTable = "train-table";
/* Training set, scored by classifier before pseudofeedback. This file is
only produced in methods with pseudo-feedback */
train1Table = "train1-table";
/* Test set, scored during the actual filtering process */
testTable = "test-table";

```

Random number generation

```

/* The seed for random number generator (as used for pseudofeedback, etc).
   If it is ≥ 0, it is passed to srnad().
   If it is < 0, no srand() call will be done. */
randomSeed=1;

```

Performance tuning

```

/* How many queries are run "in parallel". A higher number reduces
the costs of reading document F data from disk, but increases memory needs */
queryBatch=100;

/* How many documents' TF info can be cached at once. Use 0 to disable
caching. For efficient caching, the number should as large as the

```

```
total number of docs (in TRAIN + TEST) that are labeled (or
pseudo-labeled) during the processing of one query; but of course you
should keep RAM size in mind, and not set it so large as to cause
excessive swapping */
```

```
cache.maxDocCacheSize=10000;
cache.cacheAllDocs=0;
cache.clearForEachQuery=1;
cache.termCache=0;
```

Classifier parameters common to all vector-sum based models, i.e. both Rocchio and Centroid

(The prefix `paul1` is used for the Centroid-classifier parameters, because this is the internal name of that method).

```
/* TFIDF within-document weighting parameters (TF scaling),
   for queries and documents.
The log scaling is defined by David Lewis, and is different
from what Lemur retrieval application uses. */

doc.tfMethod = 1; /* 0 = RawTF; 1 = log-TF; */
query.tfMethod = 1; /* 0 = RawTF; 1 = log-TF; */

/* Pseudo-feedback parameters. Definitions are per David Lewis:
   3a. PosProportion: The topranking PosProportion of
examples will be sampled from to choose pseudopositive
examples.
   3b. PosDensity: Proportion of the topranking examples
to use as pseudopositives.
   3c. TotalPosWeight: Sum of weights of all
pseudopositive examples.
   3d. NegProportion: The bottomranking NegProportion of
examples will be sampled from to choose pseudonegative
examples.
   3e. NegDensity: Proportion of the bottomranking
examples to use as pseudonegatives.
   3f. TotalNegWeight: Sum of weights of all
pseudonegative examples
*/

fb.posProportion=0.02; /* what section of TRAIN is used for pseudo pos */
fb.posDensity=1.0;
fb.totalPosWeight= 5;
fb.negProportion = 0.98;
fb.negDensity = 0.05;
fb.totalNegWeight = 5;
```

Effectiveness measure (used in thresholding and final eval)

These parameters control the formula used by the program to evaluate the effectiveness of the classifier (the utility measure). It is used in two ways. First, at the training stage, the classifier threshold for each topic is set as to maximize the classifier's utility for this topic on the training set. Second, once the test set has been classified, the classifier effectiveness for each topic on the test set is reported, as measured by the specified utility measure.

Two utility measures are available: Linear Utility and F-Measure. The formulas for them are explained in the [Contingency Tables](#) subsection of this manual.

The linear utility for a particular topic is defined as

$$E = a*sp_{tp} - b*sp_{tn}.$$

Here sp_{tp} and sp_{tn} are the true positives and false positives for this topic, as defined in the [Contingency Tables](#) subsection of this manual. One can see that the ratio b/a controls the classifier's acceptance of false positives as the price to pay for being able to get more true positives. For example, with the default ratio $b/a=0.5$, the classifier would rather forego discovering one true member of a category if the lowering of the threshold necessary for that would result in the mistaken "discovery" of 2 non-members as members. But if, for example, you don't mind getting even 10 extra false positives if it allows you to get one additional true positive, you can set b/a to 0.1.

Similarly, if the F-measure is use, a higher value of β means a higher emphasis on recall (and a correspondingly lower emphasis on precision). Thus raising β will result in the classifier assigning more documents into each category, i.e. missing fewer documents at the cost of having more false positives.

```
/* 1=default=automatic thresholding (David Lewis' LOO procedure).
   0=use fixed ("analytic") threshold */
thres.learnThreshold=1;
/* Fixed "analytic" threshold; only used if thres.learnThreshold=0,
   i.e. if automatic thresholding is turned off. For a method like
   Rocchio with normalized vector and no feedback, using 1.0 means selecting
   (almost) none; using 0.0 means selecting all (useful for ranking
   experiments). */
thres.threshold=0.0;

/* Choice of the effectiveness measure: 0 linear utility, 1 F-measure */
thres.measure=0;

/* Weight of a discovered unjudged example, relative to that of
   a properly labeled negative example */
thres.unjWt=0.0;

/* for linear utility, a*(P+) - b*(N-) */
thres.linear.a= 1.0;
thres.linear.b= 0.5;

/* The minimum value of T11NU, used to bound it when computing T11SU */
thres.linear.minNU= -0.5;

/* for F measure */
thres.f.const= 0.0;
thres.f.beta=0.5;
```

Classifier parameters (specific to Rocchio model)

```
/* Coefficients of the Q_up vector:
   alpha*query + beta*mean_pos - gamma*sum_neg */
qup.alpha = 1.0;
qup.beta = 0.5;
qup.gamma = 0.25;
qup.fs = 1; /* Feature selection method: 0 none, 1 positive, 2 "top fsTop" */
qup.fsTop = 30; /* How many top components to keep, if qup.fs=2 */

/* The dot product is computed as (qup,x) = (qup * idf^pwr * y).
   Our default for Rocchio is 1; you can use 2 to be more like kNN.
   */
```

Classifier parameters (specific to Centroid model)

```

/* QueryVec fading coefficient */
paul1.a = 0.99;
/* If set to one, P+P' and N+N' sums are divided by the sum of their
respective weights, as it is done in Rocchio model. */
paul1.useCentroid=0;
paul1.fs = 0; /* Feature selection method: 0 none, 2 "top fsTop" */
paul1.fsTop = 30; /* How many top components to keep, if paul1.fs=2 */

```

Classifier parameters (specific to kNN classifier)

```

/* k-NN Lookup method:
-1=random (for testing),
0=direct (not supported),
1=inverted (main method; default),
2=truncated inverted,
3=dot product in RP,
4=restricted inverted,
5=big cube Hamming,
6=small cubes Hamming... */
knn.lookup=1;
/* k-NN classiffier method: 0=traditional, 1=Yang94, 2=New02a, 3="Density" */
knn.classifier=0;
/* Number of nearest neighbors that are used in the classifier.
(With BL85 truncation, this is also the HoodSize) */
knn.k=5;
/* 0=fixed threshold, 1=classifier will learn threshold.
The value must be 1 for Yang92 and New02a. */
knn.learnThreshold=1;

/* 1=Compute and use a local threshold for each test doc. This is not
a practical approach; it is only used for some experiments.
(If you do that, set MODT.k=-1 to avoid excessive tracing) */
knn.localThreshold=0;

/* The fixed threshold value. This param is ignored unless
knn.learnThreshold=0 */
knn.threshold=1.0;

/* NN_Rocchio_Beta for New02a. Ignored in other methods. */
knn.beta=1.0;
/* If 1, similarity is computed as dot product of *normalized* vectors */
knn.similarity.normalize=1;
/* If 1 (default), we don't count a document as its own neighbor
when finding the k next neighbors. This is meant to make thresholding
more realistic */
knn.leaveOneOut=1;
/* How we interpret (topic, document) pairs absent in the qrel file.
0 means implicit NONREL; -1 means implicit UNJUDGED/UNLABELED */
knn.implicit=-1;

/* 1 = we abbreviate the training set, including into the document pool
only documents that have a non-trivial judgment for at least one topic.
This is suitable for computers with not a lot of RAM. The default =0
(include the entire training set into the doc pool) */
knn.compactTrain=0;

/** If simCutoff>0, only docs with sim(d,x)>simCutoff are included into
the neighborhood. This is for an experiment for Martin Strauss (June 2003) */
knn.simCutoff=0.0;

/** Neighborhood completion method: 0 none, 1 pseudo-random */
knn.completion=1;

```

Special notes on kNN

When using kNN, Set queryBatch as high as possible; preferably, to be as large as the total number of topics you have. In kNN, many expensive things -- such as threshold setting -- cost the same per batch no matter how large the batch is; so doing it with 5 batches of 10 topics each will cost you almost 5 times as much as with a single 50-topic batch.

Besides the k-NN specific parameters, you should keep doc.tfMethod=1, to ensure the usual LOG(TF) transformation of document vectors.

Feature selection method in kNN, ILH 1:

```
knn.fs.train=0; /* Truncate pool vectors */
knn.fs.test=0; /* Truncate vectors whose neighbors are sought */
/*
  NONE = 0, // none: default
  TOPX = 1, // select X top components
  TOPXFRAC = 2, // select top fraction (X*100%)
  SUMXFRAC = 3//, // sum {top impact}/sum {impact} >= X
  RANDOM = -1 // random selection (X*100%)
*/
knn.fs.mode = 0;
knn.fs.top = 30; /* X */
knn.fs.normalize1st=0; /* If true, normalize doc vec before truncating */
```

Feature selection method in kNN, ILH 2

```
/* ILH2: ordering lists.
  NONE = 0, // none: default
  LEN=1, // list length
  TF=3, // TF in the test doc (equive to ILH1 applied to TEST)
  MAX=7, // max impact in the list
  AVG=9, // avg impact in the list
*/
knn.ilh2.order=0;
/* ILH2: termination criterion (mode)
  NONE = 0, // none: default
  XLISTS = 1, // select X lists
  XPCLISTS = 2, // select (X*100%) of all lists
  XPCSUMLEN = 3 // select lists with the total length of (X*100%) of all lists' length
*/
knn.ilh2.term=0;
/** ILH2: termination criterion (stoppping point) */
knn.ilh2.top=30;
knn.ilh2.normalize1st=0;
```

Feature selection method in kNN, ILH 3

```
/* ILH3 mode:
  NONE = 0, // none: default
  TOPX = 1, // select X top components
  TOPXFRAC = 2, // select top fraction (X*100%)
  IMPX = 3, // impact >= X
  IMPXMAX = 4 // impact >= X * max(list)
*/
knn.ilh3.mode=0;
knn.ilh3.top=0.10;
```

Classifier parameters (specific to Inverted Model Classification)

Preparations for classification with Inverted Models are described [here](#). The result of preparations should be a file of linear models, stored in a sparse vector format, with one model per line.

The classification is done with AFS software, using parameter settings described below.

The classification is implemented similarly to Inverted Index kNN method, with a few differences:

- i. At training time the training documents are ignored, and models are read from a model file and stored as an inverted index
- ii. At the classification time:
 - It is assumed that the number of queries is the same as that of models, and that they are both stored in the same order.
 - Then, for each test document the score returned for each of the queries is precisely the inner product with a corresponding model. All the scores are returned at once, so the batch size

```

        (the number of queries processed together) must be larger than
        the total number of queries.
        Note: for logistic models the scores are log-odds of an example belonging to the
        positive class.

/* filterModel has to be specified as kNN for this*/
filterModel = 2; /* 0 = Rocchio; 1=Pauli; 2=kNN. No other options yet */
/* no updating - only in batch mode */
updateFreq = -1; /* After how many Oracle judgments will the classifier
        be updated. Default is 1 (update after each judgment).
        Batch mode: -1 - no updates */

/* All models have to be handled at once: so batch has to be large enough*/
queryBatch=20000;

/*
- if models trained on sqidf representation, have to specify fw.sqidf=1 (otherwise can be ignored)
  (this parameter only affects kNN models)
- if models were trained on normalized docs, specify knn.similarity.normalize=1
*/
fw.sqidf = 0;
knn.similarity.normalize=1;

/* Specify model file. Format:
topic-id fid:value fid:value ... -1:intercept
- assumes the same order in modelFile and in query file (and of
  course the identical query ids and model ids)
*/
modelFile = /home/dfradkin/convergence-exper/afs-runs-idf/from-models/all-models.dat;

/* special codes for Inverted models for knn.lookup and knn.classifier parameters
   to do inverted model classification
*/
knn.lookup=7;
knn.classifier=4;

knn.learnThreshold=0; /* don't learn threshold: it is specified in model file, as intercept */
knn.threshold=0; /* from point of view of kNN method, the threshold is 0 */

```

Running the program

Once the Lemur indexes have been built, you can run the Adaptive Filtering Program. As in the example above, suppose that the run directory -- the directory where you want to run AdaptFilter -- is \$RUN=~/.myrun. Let \$AF=~/.filter be the directory where the Adaptive Filtering Software has been compiled (the directory that was created when the downloaded archive file was unzipped). To run the program, choose the configuration file, for example data/filterparam/filter_param, and run the executable:

```

set AF=~/.filter
set RUN=~/.myrun
cp $AF/data/filterparam/filter_param $RUN
cd $RUN
$AF/src/AdaptFilter filter_param > filter.log

```

This will run the Adaptive Filtering program. That includes training the classifier and applying it to all test documents listed in the document order file, classifying them all with respect to all topics listed in the query file. The run creates a log file (filter.log) in the run directory, as well as the output files described in the next section.

Changing the run directory

The examples above assumed a fairly simple situation: source data files were copied from the source data directory to the run directory; Lemur's text parsing and index building utilities were run in the run directory, producing the Lemur index files in that directory; and then we ran the Adaptive Filtering application in the same directory.

Of course, Adaptive Filtering software does not have to read all input files from the same directory, nor does it have to write all output files from the same directory. You can control the location of all input and output files, either absolutely or relatively to the run directory by editing the appropriate sections ([input](#) and [output](#)) of the parameter file.

Some caution is needed here, however. What if you want to produce index once, and then use it for several classifier runs in different directories, perhaps so that you can try different classification methods or to classify the corpus according to different sets of topics/categories? You can do it; however, you need to ensure that Lemur knows how to find all the index files.

The file names listed in the *.bsc files are controlled by the parameter outputPrefix in the parameter files given to Lemur's BuildBasicIndex (see how it is used in prepare.sh). The parameter file used in this example uses the value outputPrefix that defines just a file name, without complete path. This means that that if you want, you can move the entire set of files produced by the Lemur's index builder to some other directory, and then you can run Adaptive Filtering program there. But if you want to run Adaptive Filtering program from a different directory, specifying the actual location of the *.bsc files via its parameter file, it still won't find the index files.

There are three solutions to this. One is to edit the two *.bsc files so that they contain the complete path names for all component files of the Lemur index (e.g. /home/my-user-name/my-data/train-bindex.terms , etc.) Another is to provide complete and correct path name as the outputPrefix in the parameter files that you give to BuildBasicIndex when you build Lemur indexes. The third solution is to provide symbolic links from your run directory to each of the Lemur index component files (using the UNIX ln -s command).

Output files

Two types of output files are produced at each run.

Individual judgments

First, each run produces an output file (often named "res.filter") with the information about individual judgments made by the program. Each line corresponds to a single judgment with respect to a particular (topic, document) pair. By setting the configuration parameter "resultLogMode" the user may choose to log either only the

positive judgments (docs sent to the Oracle; resultLogMode=1), or only judgments for all (topic, doc_id) pairs known to the Oracle (resultLogMode=2), or all judgments (resultLogMode=0).

One of the three formats can be chosen for this output file:

1. 3-column format. Each line may look like this:

```
2 1391 1
(= topic_id, doc_id, system_score)
```

2. TREC 6-column format. Each line may look like this:

```
A101 Q0 NJ4 0 2 DIMACSTest
(= topic_id "Q0" doc_id 0 system_score run_tag).
```

3. Fusion format, used to produce output for fusion experiments. Each line may look like this:

```
0.77 2 12334 A101
(= system_score oracle_judgment doc_id topic_id)
```

Note that none of these formats contains column with a 1/0 value explicitly telling whether the document has been judged positively or negatively by the system. However, it is not an issue in most runs, since normally the default setting resultLogMode=1 is used, and only positively-judged (topic, doc) pairs are logged.

Contingency tables

Second, "contingency tables" are produced. Up to three such tables are produced on each run: one for the training set using the classifier as it exists before the pseudofeedback, one for the training set using the classifier as it exists after the pseudofeedback, and one for the test set using the classifier as it actually exists during filtering. The file names for these tables are specified in the "Output files" section in the [parameter file](#).

Each of these files contains one line per topic. Each line looks as follows:

```
A101 1 0 0 2 0 6
(=topic_id, sp_tp, sp_tu, sp_tn, sn_tp, sn_tu, sn_tn)
```

Here

- sp_tp is the total number of documents that, for this topic, the system judged as positive and the Oracle judged as positive ("true positives");

- `sp_tu` is the total number of documents that the system judged as positive and the Oracle had no judgment for;
- `sp_tn` is the total number of documents that the system judged as positive and the Oracle judged as negative.
- `sn *` are the similar numbers for the docs that the system judged as negative.

These numbers can be used to compute all kinds of effectiveness measures, at least in a run with a fully judged corpus where `sp_tu=sn_tu=0` (no unjudged documents in the corpus).
For example,

```
Recall = sp_tp / (sp_tp + sn_tp)
Precision = sp_tp / (sp_tp + sp_tn)
T11U = 2 * sp_tp - sp_tn
T11NU = T11U / (2 * sp_tp) = (sp_tp - 0.5*sp_tn)/sp_tp
T11SU = (max(T11NU, MinNU) - MinNU) / (1-MinNU), with MinNU=-0.5
```

The F-measure as defined as the weighted geometric measure of the precision and recall:

$$F_measure = (\beta + 1/\beta) / (1/\beta * 1/Precision + \beta/Recall) = (1+\beta^2) / ((1+\beta^2) sp_tp + sp_tn + \beta^2 * sn_tp)$$

The Perl script `util/ny-eff.pl` can be used to compute per-topic efficiency based on the contingency tables.

There is a more detailed discussion of the contingency tables in this document: [contingency.txt](#)

Miscellaneous utilities

The Adaptive Filtering Software distribution includes a number of auxiliary utilities. They were designed for various special purposes. They are not fully documented, as they are not required for running data classification. Some of them are briefly described in this section.

Lemur-to-SVM convertor

The program `Index2SVM` can be used to convert data stored in Lemur index to the format that can be fed to another DIMACS application, Alex Genkin's Bayesian Logistic Regression Software (BBR). As of June 2004, the current version of BBR could be obtained from

<http://www.stat.rutgers.edu/~madigan/BBR/>. (Formerly, at <http://mms-01.rutgers.edu/~aq/Deliver-Feb2004/>).

Build this program in the `src` directory of the Adaptive Filtering Software distribution, using the same Makefile as for the main application. The command is

```
make Index2SVM
```

The executable `Index2SVM` will be built in the same directory.

Inputs: . This program expects the full complement of input files, in the same format as `AdaptFilter` does. If you already have prepared your data for `AdaptFilter`, and now want to export them into SVM format as well, so that you can compare the two classifiers, you are all set. Otherwise, follow the instructions in the [Preparing the corpus](#) section of this User Guide.

Running:

The program `Index2SVM` can be run with a parameter file of the same format as `AdaptFilter` or Alex Genkin's `BBRlemur`, as follows:

```
Index2SVM parameter_file_name
```

The names and meanings of parameters are similar to those used by these two programs; of course, parameters describing classifier details don't need to be supplied to `Index2SVM`, and, if supplied, they will be ignored. Since `AdaptFilter` and `BBRlemur` sometimes use different names for parameters with roughly the same meaning, please refer to the sample file below for details.

Note also, in the example below, that `trainDocOrderFile` and `docOrderFile` can be optionally used to specify files, in the standard "document order" format (one number per line), containing lists of documents to be described in the training and test output files.

The program can be used to describe two data sets (identified as the "training set" and "test set"), or just one data set. This is controlled with the parameters `svm.trainOut` and `svm.testOut`.

```
/** SAMPLE PARAMETER FILE FOR Index2SVM */

/** The optional list of document ids from the training set
    that are to be processed. If not supplied, the entire training set
    will be processed. The IDs must be given relative to the main ("test")
    index. Alternative name is trainDocOrderFile. */
trainDocsFile = train-doc-order;
/** The optional list of document ids from the test set
    that are to be processed. If not supplied, the entire test set
    will be processed. Alternative name is docOrderFile. */
testDocsFile = test-doc-order;

/** This parameter exists for backward compatibility. Prior to Nov 4,
    2004, the order of documents in the output file was always the
    same as the order in which the documents are internally stored in
    the Lemur index. This was the case even when trainDocsFile or
    testDocsFile was given. As of Nov 4, 2004, when such list files
    are given, the output order normally corresponds to the order in
    the index. But if this parameter is set to true, the output will
    still be in the Lemur index's internal order, not the order given
    in the list file.
*/
ignoreDocsFileOrder=0;

/** The optional list of document ids (relative to the main index)
    that are to be used for computing IDF (if the use of IDF is requested.
    Typically, this should have the same value as train-doc-order. If
    omitted, the entire training index will be used to compute IDF. */
idfDocsFile = train-doc-order;

trainIndex=/home/share/cia/cia.bsc;
index=/home/share/cia/cia.bsc;
trainQrelFile=/home/share/cia/cia.qrel;
testQrelFile=/home/share/cia/cia.qrel;
```

```

textQuery=AFGH.query;

train.fullyJudged=1;
test.fullyJudged=1;

/* Processing term weights: {RAWTF=0, LOGTF=1, BINARY=2} */
doc.tfMethod = 2;
/* Feature weighting methods: {FW_NONE = 0, FW_IDF = 1, FW_SRIDF=2} */
doc.idfMethod = 0;
/* Do not include terms that occur in strictly more than dfUT documents or in strictly
fewer than dfLT documents (default values would result in all terms being included)*/
dfUT=20000;
dfLT=2;

/* If set to 1, each document vector will be normalized */
cosineNormalize = 0;

/* normalize vectors before applying IDF weighting (by default IDF is applied before normalization) */
postNormIDF=1;

/* The name of the topic whose labels will go to the output SVM files */
svm.topic = "AFGH";

/* The name of the output SVM file describing the training set.
You can use "-" for standard output, or "/dev/null" to produce no output
at all. */
svm.trainOut = "svm-train.dat";
/* The name of the output SVM file describing the test set.
You can use "-" for standard output, or "/dev/null" to produce no output
at all. */
svm.testOut = "svm-test.dat";

```

As can be seen above, you need to specify a single topic -- just one of those that are listed in your query file -- for this program. This is so because SVM input files are created for a single topic, not a set of topics.

This program can be used to convert a data set "on the fly" and feed it into BBRtrain or BBRclassify. You can see examples of it on mms-02 in /home/bbrlemur/bbr4 (files recalibrate.pl and TOPIC.*.param). For example, the following command (found in recalibrate.pl) is used to feed the Training Population of LYRL2004 corpus to BBRclassify:

```
Index2SVM AFGH.train2svm.param | BBRclassify - AFGH.model AFGH.tp.res
```

The parameter file AFGH.train2svm.param (based on TOPIC.train2svm.param) has

```

trainDocsFile = /home/share/rcv1/lyrl2004/train-pop.docs;
svm.trainOut = "-";
svm.testOut = "/dev/null";

```

which means that only the "training set" (the documents listed in trainDocsFile) will be processed; the training set will be written to its standard output. BBRclassify is given "-" as the name of the source file, which means that it will read the data set from its standard input.

Warning: document order

This is a warning about a feature that may be of concern for some uses of the Index2SVM tool. Namely, the documents in the output file of Index2SVM are always arranged in the "natural order" in which they are stored in Lemur index. This is the case even when Index2SVM is used with trainDocsFile or testDocsFile. That is, although this document list files control the set of documents whose vectors will be printed, they do not control the order.

This is not in a concern in typical applications when you simply feed the document data to BBRtrain and BBRclassify, and have these public BBR tool report classifier effectiveness etc. Nor is it a concern if your BBR tools are built to produce 6-column output (complete with the document label). However, it may be a concern if one wants to do some additional post-processing with the result file produced by BBRclassify, and needs to collate the result file with the QREL file to figure which documents are true positives and which ones are true negative. To handle this problem when it arises, I prepared special "document order files" that reflect the order in which the documents are stored in the Lemur index of the corpus I am using. If/when you need to do the same for your corpus, let me.

A future version of Index2SVM should have this problem fixed.
(Some time after Oct 19, 2004).

If you need it,
this is how you can generate a doc list that reflects the order in Lemur's index (and in the existing Index2SVM's output):

- take the existing param file you used to produce "SVM light" file;
- in the param file, replace the topic name with "", i.e.
svm.topic="";
This, when given to Index2SVM, will produce a SVM regular file,
with the doc id in every line.
- You can extract the first token from every line; that will be the doc id:

Index2SVM svm.param | perl -pe 's/(\d+)\b.*/\$1/' > list.docs

Results processing script

The script util/ny-eff.pl can be used to convert the data in the contingency tables to a table that contains classifier effectiveness on each topic, as well as the average and median effectiveness.

The script util/parseAiOut.pl can be used to extract the data from the output of another DIMACS application, aiSVM, and convert them to contingency tables in the same format that AdaptFilter produces. Furthermore, the script invokes util/ny-eff.pl to produce the effectiveness table as well.

Topic dictionary tool (MkDic)

The MkDic utility prepares a "topic dictionary", that is the list of terms that occur in the specified document set. We used it to prepare a "topic dictionary", i.e. the list of terms that occur in the topic's training set (the *.terms files), as part of the reports for the BBR experiments.

Read the source code for details. Generally, this application can be run with the same parameter file as used for Alex Genkin's BBRlemur. In its result file, it will produce the dictionary file for the training set specified in the parameter file.

Prior knowledge extraction tool (MkPrior)

The MkPriors program prepares an "individual priors" file for Alex Genkin's BBRlemur application (see

[Bayesian Logistic Regression Software](#)). Such an "individual priors" file contains mode values and relative variances for "prior knowledge" (PK) terms extracted from the prior knowledge corpus.

Usage:

```
MkPrior parameter_file
```

Sample parameter file for MkPrior

```
/*=====*
*           File location paramters           *
*=====*/

/* The index for the Prior Knowledge corpus */
priorIndex = "/home/share/cia/cia.bsc";

/* The "target index" is the index on the corpus which will be
classified by the BBR classifier for which we are prearing
the individual priors file.

(Why is that needed? The individual priors file produced by MkPrior
will be given to a BBR classifier. The BBR classifier will be
thinking in term of term IDs, rather than term spellings (actual
words). So we need to be able to use proper term IDs in our
output file, and we need to get them from somewhere.
*/
targetIndex = "/home/share/rcv1/lyrl2004/all.bsc";

/* This QREL file tells which document(s) in the PK corpus provide(s) the
prior knowledge for which topic */
priorQrel = "/home/share/cia/cia.qrel";

/* List of topics for whcih me want to generate prior knowledge */
textQuery = "/home/share/rcv1/lyrl2004/region-name.query";

/** Name of the "prior knwoledge" file to be generated */
priorOut = "priors.iinf.sk1";

/*=====*
*           Term selection
* Which terms occuring in the PK corpus will be designated as "prior
* knowledge" terms?
*=====*/

/* If maxDF < 1, only terms whose document frequency in the PK
corpus is no greater than maxDF will be included. E.g., for
maxDF=0.05, all terms that occur in more than 5% of PK documents
will be ignored.
*/
maxDF = 0.05;

/*=====*
*           Term significance
* If useTF=useIDF=0, term significance is equal to 1 for all included
* terms. Otherwise, terms TF and/or IDF is taken into account.
*
* Set useTF = 0; useIDF = 0; for the prior1 heuristic.
* Set useTF = 1; useIDF = 1; for the prior3 heuristic.
*=====*/

/* If useTF=1, term freq in the PK document will affect its significance */
useTF = 0;
/* If useIDF=1, document frequencies (prevalence of a term among the
documents in the PK corpus) will affect the term's significance */
```

```

useIDF = 0;

/*=====
 *      Converting significance of PK terms to distribution parameters  *
 *=====*/

/* (The mode value for term_i) = modeFactor * significance(term_i) */
modeFactor = 0;
/* (The relative variance for term_i) = varFactor * significance(term_i) */
varFactor = 1;

/*=====
 * You can also control distribution parameters (mode and relative
 * variance) for the intercept. Infinity (intercept.var = inf) is a
 * legal value for intercept.var; it means "unconstrained distribution".
 *=====*/

intercept.mode = 0;
intercept.var = inf;
/*
  You can explicitly specify "intercept.skew=0;" in order to allow
  the intercept value to be negative. It is necessary to explicitly
  say this only if BBRlemur is run with the options that make terms'
  distribution skewed (e.g., exponential distribution, where each
  beta_i is distributed above its mode value). This way you could
  allow BBRlemur to choose a negative intercept value even as all
  beta_i have exponential distribution (a skewed, a.k.a. "non-negative",
  distribution)
*/
intercept.skew=0;

```

Threshold retuning tool (Retune)

This utility was used together with Alex Genkin's BBR applications as part of David Madigan's intercept recalibration experiments (September 2004). It operates as follows:

1. It reads a score file produced BBRclassify that has used a Bayesian model with a certain beta vector and a certain intercept value.
2. It internally adjusts the scores to the values that would have been obtained if the intercept had been different.
3. It applies a threshold tuning algorithm to the resulted adjusted score list. (As of September 2004, only David Lewis' Thresholding Option B.2 is supported)
4. It prints out the threshold so obtained. (Some other information may be printed as well, depending on the logging level, but in any event the tuned threshold is printed all by itself on the last line of the output).

Source code and comiling

Unlike other utilities, Retune lives in filter/retune. All necessary source code (coming mostly from Alex Genkin's BBR package) is in that directory; at this time, there are no dependencies on AdaptFilter source code. To compile, simply type

```
make
```

Usage

The full command line looks as follows:

```
Retune topic file.res id_list.docs file.qrel intercept_change TuneEst TuneThreshold
```

Here

topic

Topic name. This is used to look up the labels in the QREL file, if necessary.

file.res

The results file to be read. This can be the result file produced by BBRclassify, containing estimated class membership probabilities for each scored documents. Other formats are supported; see source code comments in regard to formats.

id_list.docs

The file containing the list of document IDs. This file should contain the same number of lines (other than the headers, if any) as the result file, in the same order. Each line of this file should contain the ID of the document whose score etc. are given on the corresponding line of the score file.

file.qrel

[Currently ignored] The QREL file. In future versions, when e.g. A.2 tuning option is supported, could be used to look up training set labels for documents.

intercept_change

A floating-point number used in score-recalculation. If the scores given by the result file have been computed by a model with the intercept value Y_0 , then the Retune tool will recalculate the scores to the values that would have been obtained if the intercept value had been $Y_1 = Y_0 + \text{intercept_change}$.

TuneEst

An integer, describes the threshold tuning option in use. Corresponds to TuneEst in BBRlemur's parameter file. Values:

- 0:
The threshold is tuned by optimizing some classifier quality measure on the (small) labeled training set.
- 1:
The threshold is tuned by probabilistically optimizing some classifier quality measure on the (large) mostly unlabeled training population. The results file should cover the entire training population.

TuneThreshold

Integer paramter, controls the function to optimize when tuning threshold. Two criteria are supported, with the same parameter values as the parameter TuneThreshold in in BBRlemur's parameter file. (2=optimize T11NU; 3=optimize F1).

More information is available in the source code comments (filter/retune/Retune.cpp). For usage example, see /home/vmenkov/bbrlemur/bbr5/scripts/rescore.pl.