

Adaptive Filtering Software: Overview

Contents

1. [Classification methods overview](#)
2. [Classic Rocchio](#)
3. [Classic Centroid](#)
4. [kNN With Inverted File Heuristic](#)
5. [TREC competition](#)

Classification methods overview

The purpose of the Adaptive Filtering Software described in this manual is classifying documents from a large collection (corpus). The user supplies two collections of documents: the [initial] *training set* and the *test set*. Each document may belong to one or several categories, based on its content. The user also provides the system with the list of *topics*, or categories. For each topic the user needs to tell the system something about what documents relevant to this topic are like. This information may include an optional text descriptions of a topic (the *query text*); more importantly, the user should tell the system it how all (or at least some) documents from the training set are classified, i.e. to which topics/categories they belong. Based on this information, the software builds a *classifier* using one of the available classification methods. The classifier then proceeds to classify the documents from the test with respect to all the topics, i.e. to report to the user to which categories each of them belongs.

Batch filtering vs. adaptive filtering. Each of the classification methods can be used for batch filtering or adaptive filtering. In *batch filtering* (classifying), the classifier is built once, based on the initial training set, and is used to classify all documents from the test set.

Adaptive filtering is designed to make use of any feedback (experts' judgments of a document) that are made available to the classifier after the classifier has classified a document. initial classifier is constructed based In adaptive filtering, an initial classifier is constructed based on the initial training set. Documents from the test set are classified in the order indicated by the user; as more documents have been classified, experts' judgments, if any, for these documents are used to update the classifier.

The Adaptive Filtering Software described in this document allows the user to control the frequency of the classifier updates during adaptive filtering. Batch filtering is implementing simply as a special case of adaptive filtering: by setting the classifier update frequency to **never**, you make the program operate in batch filtering mode.

Implementation limitation: no real-time processing.

Theoretically, adaptive filtering algorithms can be used with real-time feedback: that is, after a document has been judged by the classifier as potentially promising for some topics, the human expert(s) would have a chance to look at them and provide the system with the feedback that it can use to improve the classifier. Moreover, the test set could be augmented during the run, as new documents keep coming in from the outside world. In this implementation, however, the entire test set and all the "feedback" have to be supplied to the classifier beforehand.

Classifier = model + threshold. Each classifier consists of a *classification model* and *thresholds*. The classification model is used to compute, for each (document, topic) pair, the numeric score that reflects the model's estimate of the document's relevance for the topic. The classifier has a threshold for each topic; each scores computed by the model are compared to this threshold to decide whether the document is relevant to the topic.

Naturally, both the classifier and the threshold are constant in batch filtering, while both may be adjusted from document to document during adaptive filtering.

Main Classification Models. This software implements three document classifiers, each with its own classification model:

1. Classic Rocchio;
2. Classic Centroid;
3. kNN with inverted file heuristic.

Classic Rocchio

Based on the write-up provided by David Lewis, July 17, 2002

I. This section describes the design and operation of our Classic Rocchio classifier.

II.A. Data Preparation

This phase is the same for all classifiers. One index is built for the training set, and one for the entire corpus (training set + test set).

II.B. Training Initial Classifiers

Initial training for each topic uses the training set (on which relevance status with respect to the topic is known only for three of the documents), and the topic description.

FOR EACH Topic

B2. Read initial positive training examples for this topic. Give each of these examples a weight of 1.0.

B3. Call scoring model learning algorithm (Rocchio in our baseline) to produce linear model based on the topic description and the initial positive examples.

B4. Call pseudolabeling algorithm (Section V) to run the linear model trained in Step B3 against all training documents. It will return some portion of the training documents, and will have associated with them positive or negative pseudolabels and fractional weights.

B5. Call classifier learning algorithm (in our baseline,

this is Rocchio, followed by thresholding) to create a linear classifier based on topic description and all labeled examples (which includes both initial positive examples, and pseudoexamples from Step B4.).

II.C. Adaptive Filtering

In adaptive filtering, we run through the test documents in a specified order, applying classifiers, getting judgments only for documents judged relevant, and updating the classifiers.

FOR EACH Test Document

FOR EACH Topic

C1. Apply current classifier for topic to test document, computing score and determining if score is above threshold.

IF score is $\geq$ threshold

C2.1. Pass document ID, topic ID, score, and label ("relevant") to routine that writes output for evaluation

C2.2. Pass document ID and topic ID to judging routine, which will return label (Relevant vs. Nonrelevant vs. Unjudged).

ELSE

C3.1 Label = Unknown

C4. Pass current classifier, document ID, Label, and a weight of 1.0 to Learner (which for the baseline will be an object that in turn calls Rocchio and TROT).

III.The Rocchio Algorithm for Linear Model Training

The Rocchio algorithm produces a linear model (not a linear classifier, i.e. there's no threshold). The basic inputs to the algorithm are:

Inputs:

1. An initial "query" vector
2. A set of document vectors. Each vector is accompanied by a weight and a label.
3. The Rocchio weighting parameters (alpha, beta, gamma)
4. Feature weighting parameters
5. Feature selection parameters

III.A. Batch Implementation of Rocchio

In a batch implementation of Rocchio, the algorithm computes the weighted, elementwise vector sum:

$$Q_{up} = \alpha * Query + \beta * mean_{pos} - \gamma * mean_{neg}$$

We then do any desired feature weighting (Section VI.) and feature selection (Section III.D.). The Query is optional (or, equivalently, it could be allowed to be all 0's). If no Query and no positive documents are presented, then a linear model with all coefficients equal to 0 is produced.

In the above $mean_{pos}$ is the weighted average of the positive document vectors, i.e.:

$$(\sum_i Weight[i] * Vec[i]) / (\sum_i Weight[i])$$

where i runs over document vectors with label "Relevant". $mean_{neg}$ is defined similarly. Rocchio ignores documents with label "Unknown" or "Unjudged" if they are given. Note that the document vectors used at this stage will have been transformed using a within document weighting scheme (Section VII.).

III.B. Adaptive Rocchio

If adaptive mode is on, the Q_{up} vector described above is updated after each judgment, or every time after a specified number of judgments.

III.C. Feature Weighting For Rocchio

The effectiveness of models produced by the Rocchio algorithm is greatly increased by the use of corpus based weights. These should be computed outside the Rocchio algorithm but made available to the Rocchio algorithm.

The idea is that after the basic Rocchio procedure (Section III.A.) is run, we multiply the resulting weight for each term by a corpus-based weight for that term. IDF and IDF**2 weights are available.

III.D. Feature Selection for Rocchio

When good feature weighting and within document weighting are used, the Rocchio algorithm produces a reasonably effective linear model even without feature selection. However, there are two common feature selection approaches we implement:

1. Replacing all negative coefficients with 0. This usually improves effectiveness and always improves efficiency. It should be the default, though there should be a parameter to allow this to be turned off.
2. Zeroing out all but k highest coefficients typically improves effectiveness if k is set to a sensible value. k should be a parameter passed to Rocchio.

IV. Threshold Training

Some learning algorithms produce a model that assigns scores to documents, but does not make Relevant/Nonrelevant classification decisions. Rocchio, for instance, produces a linear model that assign scores.

The most common way to convert a scoring model into a classifier is to associate a threshold with the model. The threshold indicates that when the score is greater than or equal to the threshold, we output the class label Relevant, and otherwise output the class label Nonrelevant.

IV.A. Training Set Optimization of Threshold (TROT) Algorithm

Here we describe a simple approach, TROT, to find a good threshold for a scoring model.

The inputs to the algorithm are:

Inputs:

- A scoring model
- A set of document vectors. Each vector is accompanied by a weight and a label.
- An effectiveness measure the threshold is to optimize. (This probably should take the form of a function (or object) that can be applied to a contingency to produce an effectiveness value.)

The algorithm is:

1. Run the scoring model over a set of weighted, labeled examples. Only examples with labels Relevant and Nonrelevant (not Unknown or Unjudged) should be included.

2. Sort the examples by score.

3. Initialization

- 3a. Initialize the four cells of a standard 2x2 contingency table to:

a (sys R, true R): 0
b (sys R, true N): 0
c (sys N, true R): sum of weights of relevant examples
d (sys N, true N): sum of weights of nonrelevant examples.

- 3b. Compute value of the effectiveness measure of interest from this contingency table, and record this as the best effectiveness so far. Record infinity as the best threshold so far. (See the TREC filtering official document for the effectiveness measures of interest.)

4. FOR each score taken on by some example, in decreasing order

- 4a. Update the contingency table to correspond to the

system saying Yes to all examples with score at or above this value. This only requires looking at the weights and labels for examples with scores equal to the threshold.

4b. Compute effectiveness for the updated contingency table. If better than best effectiveness seen so far, update best effectiveness and best threshold.

Note that we assume (as is usual) that the scoring model was trained so that it would tend to associate higher scores with relevant documents and lower scores with nonrelevant documents.

IV.B. Threshold training in adaptive Rocchio

In adaptive Rocchio, threshold is re-trained every time after we modify the Q_{up} vector in the classification model.

V. Pseudolabeling (pseudo-feedback)

Pseudolabeling uses a scoring model to automatically assign labels to some or all of a set of documents for which human-assigned labels are not available. The goal is to produce additional labeled data which can then be used by algorithms such as Rocchio and TROT.

Pseudolabeling has unpredictable effectiveness and so considerable tuning is likely to be necessary. We give an algorithm below that is simple but allows considerable flexibility.

V.A. Simple Pseudolabeling Algorithm

The inputs are:

1. A scoring model
2. A set of document vectors. Let NumDocs be the number of document vectors.
3. Parameters controlling how many examples will be pseudolabeled and with what weights:
 - 3a. MaxPosRank: Maximum rank of an example that will allow it to be pseudolabeled as relevant.
 - 3b. PosFraction: Proportion of high scoring examples to be pseudolabeled as relevant.
 - 3c. PosWeight: Weight to be given each example that is pseudolabeled as relevant.
 - 3d. MinNegRank: Minimum rank of an example that will allow it to be pseudolabeled as nonrelevant.
 - 3e. NegFraction: Proportion of low scoring examples to be pseudolabeled as nonrelevant.
 - 3f. NegWeight: Weight to be given each example that is pseudolabeled as nonrelevant.

The algorithm is as follows:

1. Apply the scoring model to all examples.
2. Sort examples by score.
3. Randomly select $\text{PosFraction} * \text{MaxPosRank}$ of the top MaxPosRank scoring examples. Any examples which are already labeled Relevant or Nonrelevant should not be changed. The other selected examples should be labeled Relevant and given weight PosWeight .
4. Randomly select $\text{NegFraction} * (\text{NumDocs} - \text{MinNegRank} + 1)$ of the bottom $\text{NumDocs} - \text{MinNegRank} + 1$ examples. Any examples which are already labeled Relevant or Nonrelevant should not be changed. The other selected examples should be labeled Nonrelevant and given weight NegWeight .

VI. Corpus-Based Feature Weights

Corpus-based feature weights are weights that capture the general usefulness of a term for discriminating documents. The most common, and the one we'll use in the baseline, is the IDF (inverse document frequency) weight. The IDF weight of a term is:

$$\ln(N) - \ln(n)$$

where N is the number of documents in some collection, and n is the number of documents in that collection that contain a particular term.

We use the simplest approach, namely computing IDF weights from the training set only. That is a large enough set of documents for the weights to be fairly stable, and avoids the need to be updating the IDF weights as we progress through the test documents. (We're of course not allowed to look ahead into the whole test set to compute IDF weights.)

Using the training set raises one question: what to do about words that occur in test documents, particularly judged test documents, but were not in the training set. A simple approach is to replace the IDF formula with:

$$\ln(N+1) - \ln(n+1)$$

VII. Within Document Weighting

Within document weighting refers to computing what value each term should have for each particular document. There are two main factors to within document weighting:

1. TF scaling
2. Document length normalization

TF scaling is easy. A robust and effective approach is "log TF weighting". This replaces the raw TF values (what's stored in Lemur's index) with:

0 , if $TF = 0$ [so omitted term ids stay omitted]
 $1 + \ln(TF)$, if $TF > 0$

TF scaling is enabled by setting

```
doc.tfMethod = 1;  
query.tfMethod = 1;
```

in the parameter file.

Document length normalization is a less settled issue, but on the other hand will probably not have a big effect for our data sets. We don't currently use it.

Classic Centroid

This method uses a classification model proposed by Paul Kantor in August 2002. The general algorithm is similar to Rocchio; but instead of the Rocchio score, the score is computed as follows:

Each time a new document is scored, two vectors, P and N are updated.

```
P = Normalized[a^{k}*QueryVector + Sum {positive judged vectors}]  
N = Normalized [Sum {negative judged vectors }]
```

Therefore, these are two unit vectors.

a is adjustable -- default is .99

k is the number of documents that have been truly judged

The scoring parameter is computed as follows,. For the new document d , form the Normalized vector: call it d .

Compute score = $\|d-N\| / \|d-P\| = \sqrt{(1-(d,n)) / (1-(d,p))}$.

kNN With Inverted File Heuristic

(1) Basic Batch kNN

Detailed plan for our design for basic batch kNN can be found in this document: [Plan for Fall 2002 Software Development for Experiments on Approximate Nearest Neighbor Text Classification](#). While the actual code design may be somewhat different from that described in that document, all methods described there are supported.

(2) Inverted List Heuristics

[Outline of Inverted List Heuristics \(ILH\) for kNN](#)

(3) Adaptive kNN

Basic kNN (without ILH) is available in adaptive form as well. We use the concept of a *document pool*: the set of documents from which the k nearest neighbors are selected for the documents being classified. In batch kNN, the document pool is constant, and typically coincides with the initial training set. In adaptive kNN, new documents are added to the document pool as the system keeps classifying the test set.

In adaptive kNN, A document are added to the do When a new document is added to

TREC competition

FIXME: explain relevance.

http://trec.nist.gov/act_part/tracks.html and

http://trec.nist.gov/act_part/guidelines/filter2002_guide.html