MULTIBOOST (v1.2.02) documentation

Djalel Benbouzid[1,2] DJALEL.BENBOUZID@GMAIL.COM

Róbert Busa-Fekete[1,3] BUSAROBI@GMAIL.COM

Norman Casagrande[4] NORMAN@WAVII.COM

François-David Collin[1] FRADAV@GMAIL.COM

Balázs Kégl[1,2] BALAZS.KEGL@GMAIL.COM

[1]*Linear Accelerator Laboratory (LAL), University of Paris-Sud, CNRS Orsay 91898, France*
[2]*Computer Science Laboratory (LRI), University of Paris-Sud, CNRS Orsay 91405, France*
[3]*Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged, Aradi vértanúk tere 1., H-6720 Szeged, Hungary*
[4]*wavii.com*

Contents

1 Introduction

This manual provides a brief description of the main features of MULTIBOOST package. We start by the user manual (Sections 2-8). Section 9 contains some information on how to add simple modules to MULTIBOOST. The algorithms implemented in our package are described formally in Appendices A and B. For further information, contact

multiboost@googlegroups.com

2 Getting started

Download The source code of MULTIBOOST package is available at

http://www.multiboost.org.

One can also obtain the package from the github repository project using the following command:

```
> git clone https://github.com/AppStat/MultiBoost.git
```

We also provide pre-compiled executables for linux, MacOS and Windows. Windows users needs to install Visual C++ Redistributable for Visual Studio 2012 Update 1 in order to use the windows x64 executable.

Compilation The easiest way of getting the binary is to compile the package using `cmake`. The `cmake` program is an open source cross-platform build system which allows the user to configure easily the compilation process. It is available at `http://www.cmake.org`. We provide a `cmake` configuration file along with the source code of MULTIBOOST.

Assuming that `cmake` is installed on your system, the MULTIBOOST package can be compiled and run using the following commands:

```
> unzip multiboost.zip
> mkdir MultiBoost-Build
> cd Multiboost-Build
> cmake ../MultiBoost-Release
> make all
> ./multiboost
```

It is highly recommended to do out-of-source builds with `cmake`, that is, to choose a build directory which is different from the sources directory. If you are using the GUI version of `cmake`, just fill the path where you unzipped the sources (where the source code is) and your build path (where you will build the binaries). Apply "configure" and then "generate". After this you can use the build files created in the build directory to compile the multiboost project. `cmake` builds work on all platforms supported by `cmake` (including Xcode and Visual Studio). Be aware that when using `cmake` builds, if you add a `.cpp` file to or remove a file from your project, you will have to touch or remove the `CMakeCache.txt` file. This will force `cmake` to renew the file list and update the project to take into account your added/removed files. The source code of MULTIBOOST can also be imported into integrated development environments (IDE) such as XCode and Visual Studio. Users should be advised that in the future version 2.0, only `cmake` builds will be supported (on all platforms already supported by `cmake`), and drag'n'drop imports of source trees in IDE will not be supported.

Linking In addition of the executable, you can link your own modules and executables with `MultiboostLib` library, in that case, you will have to include `Registrations.h` to trigger the factory initialization of the learner.

3 Basics on setting the configurations

Every option can be set from command line using the following format:

```
--<option name> <option value 1> ... <option value K>
```

Optional values are put between square brackets (eg. `[<value>]`). Some flag options have no value. In this case the option can be switched on or off by

```
--<option name>
```

For convenience, one can use an input configuration file containing one option per line (without the double dash `--`) in the form of

```
<option name> <option value 1> ... <option value K>
<option name> <option value 1> ... <option value K>
...
```

If a configuration file is provided, the inline options will not be taken into account. The name of configuration file can be given by the following command line option

```
--configfile <name of the config file>
```

For a comprehensive list of options, use the help option

```
--help
```

4 Setting the strong learners

The strong learner[1] can be chosen using the

```
--stronglearner <name of strong learner>
```

option. The following four strong learners are implemented in the current version:

1. `AdaBoostMH` for AdaBoost.MH [1], Section 4.1, Section A.2
2. `ArcGV` for Arc-gv [2, 3], Section 4.2, Section A.3
3. `FilterBoost` for FilterBoost [4], Section 4.3, Section A.4
4. `VJcascade` for Viola-Jones cascade [5], Section 4.4, Section A.5.1
5. `SoftCascade` for soft cascade [6], Section 4.5, Section A.5.2

The algorithms are described in Appendix A. The following basic options are available for every strong learner:

1. `--train <training file> <number of iterations>`: trains on a given dataset.
2. `--traintest <training file> <test file> <number of iterations>`: trains on a given dataset and provides performance metrics in every iteration on a given test set.
3. `--test <test file> <model file> <number of iterations> [<outfile>]`: tests on a given dataset using a model file up to a given number of iterations.
4. `--posteriors <test file> <model file> <outfile> <number of iterations> [<period>]`: outputs the posteriors (the classification score) for a given dataset using a model file up to a given number of iterations.

[1] The name originally came from the boosting (PAC) literature. Here we use it in a broader sense to indicate the "outer" loop of the boosting iteration.

5. `--cmatrix <test file> <model file> [<outfile>]`: outputs the confusion matrix to the standard output or, if provided, to a file.

The following optional parameters can be provided to all learners:

1. `--shypname <modelfile>`: the output model file name. If not given, the default is `shyp.xml`.
2. `--outputinfo <outfile> <name of metrics>`: the different metrics or performance measure one would like to output. Please note that the description of the output is written in `<outfile>.header`. See Section 7.1 for the description of the metrics.

4.1 AdaBoost.MH

AdaBoost.MH (Section A.2) is set as default learner thus omitting the `stronglearner` option will set the strong learner to `AdaBoostMH` (with a warning message). AdaBoost.MH can be launched using

```
./multiboost --stronglearner AdaBoostMH
```

In the following subsections we describe the options related to AdaBoost.MH.

4.1.1 Initial weighting policies

Initial weighting is an important concept in multi-class AdaBoost.MH. The usual single-label setup is to set the weight of the positive label of each point equally to the sum of the weights of the negative labels, and weight points uniformly. This is the default behavior if no weights are specified in the training set and no explicit weight policy is given. Formally,[2]

$$w_{i,\ell}^{(1)} = \begin{cases} \frac{1}{2n} & \text{if } \ell \text{ is the correct class of } \mathbf{x}_i \text{ (if } y_{i,\ell} = 1\text{)}, \\ \frac{1}{2n(K-1)} & \text{otherwise (if } y_{i,\ell} = -1\text{)}. \end{cases} \tag{1}$$

The user can provide unnormalized initial weights $w_{i,\ell}$ in the data file (see Section 6.1). If some labels are not specified (the sparse label case), we assume that the corresponding weight is $w_{i,\ell} = 0$. If a label is specified but its weight is not, we set $w_{i,\ell} = 1$. The default policy (which reproduces (1) in the multi-class, single-label case if the user does not specify the initial weights) is that

1. we share the weights equally among data points,
2. inside a data point we share the weights equally between positive and negative labels, and
3. among the labels of the same sign we share the weights proportionally to the user specified weights $w_{i,\ell}$.

Formally,

$$w_{i,\ell}^{(1)} = \frac{1}{2n} \frac{w_{i,\ell}}{\sum_{\ell'=1}^{K} \mathbb{I}\left\{y_{i,\ell'} = y_{i,\ell}\right\} w_{i,\ell'}}.$$

This policy can be set explicitly by

```
--weightpolicy sharepoints
```

[2] The notations are defined in Appendix A.

We can have two other policies depending on the choice we make in the first two points. If 1. is held but 2. is not, we have

$$w_{i,\ell}^{(1)} = \frac{1}{n} \frac{w_{i,\ell}}{\sum_{\ell'=1}^{K} w_{i,\ell'}}.$$

This policy can be set by

```
--weightpolicy sharelabels
```

If neither 1. nor 2. is held, we have

$$w_{i,\ell}^{(1)} = \frac{w_{i,\ell}}{\sum_{i'=1}^{n} \sum_{\ell'=1}^{K} w_{i',\ell'}}. \tag{2}$$

This policy can be set by

```
--weightpolicy proportional
```

A fourth policy assures that each class (column vector of the label matrix) has the same total weight, inside the column vector the weights are shared equally between positive and negative labels, and inside labels of the same sign weights are proportional to the user specified weights $w_{i,\ell}$. Formally,

$$w_{i,\ell}^{(1)} = \frac{1}{2K} \frac{w_{i,\ell}}{\sum_{i'=1}^{n} \mathbb{I}\left\{y_{i',\ell} = y_{i,\ell}\right\} w_{i',\ell}}.$$

This policy can be set by

```
--weightpolicy balanced
```

Note that with default weights $w_{i,\ell}^{(1)} = 1$ and *balanced* class distribution, this policy also boils down to (1).

4.1.2 Time limit

If the time limit is given using

```
--timelimit <time limit in minutes>
```

training will stop either if the trainin time hits the limit or if the number of iterations hits the limit, whichever occurs first.

4.1.3 Early stopping

In batch experiments on unknown data sets, it may be difficult to set either the time limit or the number of iterations "blindly". Even though overfitting is a rare issue (and it can be dealt with "offline", after the run has stopped), it may be computationally inefficient to run the algorithm for a long time once the test performance is no longer improving. At the same time, setting the time limit or the maximum number of iterations to a too small value may result in underfitting. Starting with version v1.2, we provide and early stopping option that can be activated by

```
--earlystopping <minIterations> <smoothingWindowRate> <maxLookaheadRate>
--earlystoppingoutputinfo <outputColumn>
```

Since the stopping criteria is based on the test results, it only works in `--traintest` mode. The semantics is the following. We compute the smoothed test error at every iteration T as the average test error of the last `<smoothingWindowRate>` $\times T$ iterations. Let the current minimum of the smoothed test error occur at $T_{\min}$. We stop the run if $T >$ `<minIterations>` and the smoothed test error has not improved during the last `<maxLookaheadRate>` $\times T_{\min}$ iterations. The training will also stop either if the training time hits the limit (if `--timelimit` is set) or if the number of iterations hits the limit (`<number of iterations>`), whichever occurs first. One could use another processed value than the test error with the optional `--earlystoppingoutputinfo` argument, currently only `e01, w01, r01, ham, wha, auc` are supported.

If the run is resumed (Section 7.4), the counter and the minimization is reset to the iteration where the resume occured. This behavior will remain the same with the standard (fast) resume setting since in this case we do not recompute the test errors preceeding the iteration where the resume occured. When the `--slowresumeprocess` switch is set, we use the pre-resume test errors to reproduce the same early stopping behavior that would happen without resuming (this has been implemented since 1.2.02).

There are no default values for the parameters since it is important that the user understand the semantics. In large-scale cross validation experiments we have been using `--earlystopping 50 0.2 2`.

4.2 ARC-GV

ARC-GV (Section A.3) can be launched using

```
--stronglearner ArcGV
```

ARC-GV has all the options of ADABOOST.MH plus the parameter which controls the threshold $\theta_{\min}$ on the minimum margin

```
--minmarginthreshold <threshold>
```

The reason of having this parameter is that the coefficient assignment (22) gives $\alpha = \infty$ as long as the normalized minimum margin is -1, in other words, as long as there exist any training point misclassified by all the base classifiers. So we set $\theta = \theta_{\min}$ as long as there exists any point and label with normalized margin $\tilde{\rho}_{i,\ell} < \theta_{\min}$. The default value of $\theta_{\min}$ is 0 which means that we run ADABOOST.MH as long as the training error is larger than zero and ARC-GV afterwards. To start ARC-GV earlier, $\theta_{\min}$ has to be set to $\theta_{\min} = -1 + \epsilon$ with an appropriately selected small ϵ. Setting $\theta_{\min} = 1$ is equivalent to running ADABOOST.MH.

4.3 FILTERBOOST

FILTERBOOST (Section A.4) can be launched using

```
--stronglearner FilterBoost
```

FILTERBOOST has all the options of ADABOOST.MH plus the parameter which controls the size of the subsamples which can be set using

```
--C <size of subsamples>
```

4.4 The Viola-Jones cascade

The Viola-Jones cascade learner (Section A.5.1) can be launched using

```
--stronglearner VJCascade
```

VJCASCADE always starts with `sharepoints` initial weighting policy. The `<number of iterations>` parameter refers to the number of stages here. In each stage, the number of iterations is controled implicitely by the minimum acceptable true positive rate and maximum acceptable false negative rate. These latter options can be set by

```
--minacctpr <rate>
--maxaccfpr <rate>
```

One can give a validation set for tuning the threshold in a stage by

```
--trainvalidtest <training file> <validation file> <test file>\\
 <number of iterations>
```

In cascade detectors the positive and negative labels are handled asymmetrically. The positive label refers to the class to be detected, and it can be set using

```
--positivelabel <label name>
```

4.5 The SoftCascade Learner

The soft cascade learner (Section A.5.2) can be launched using

```
--stronglearner SoftCascade
```

SOFTCASCADE also has additional parameters. It learns over a previously trained strong classifier (it reads a shyp file, cf. 7.3) which has to be specified using

```
--calibrate <file name> [<limit>]
```

`<limit>` limits the number of base classifiers read. If `--calibrate` is not provided, ADABOOST.MH will be run beforehand. The accuracy/speed trade-off parameter has to be given using

```
--expalpha <value>
```

Negative values favor speed over the accuracy while the positives ones tend to favor the accuracy. This parameter is set through the option. The target true positive rate is set using

```
--detectionrate <value>
```

The learner adds new false positives to the training set after each iteration, so the learner needs an additional data file, containing only negative instances along with the percentage of examples drawn from this set in each iteration. These parameters are set using

```
--bootstrap <file_name> <rate>
```

As for VJCASCADE, the positive label is specified using

```
--positivelabel <label name>
```

5 Setting the base learners

The base learner is set using

```
--learnertype <base learner name>
```

We describe here some of the most common base learners. Others usually follow the same syntax and are described more thoroughly in Appendix B. For a comprehensive list of the implemented base learners, use the `--help` command.

5.1 Decision stumps

A decision stump (Section B.2) is a one-decision one-decision two-leaf tree learned on real-valued features. It is indexed by the feature it cuts and the threshold where it cuts the feature. The command line parameter for this base learner is `SingleStumpLearner`.

5.2 Haar filters

`HaarSingleStumpLearner` is a decision stump over Haar features [5]. It is assumed that the training set contains integral images. The image dimensions can be set using

```
--iisize <image dimensions>
```

If not set, rectangular images are assumed. The number of Haar filters evaluated in each iteration can be set using

```
--csample num <number of filters>
```

Alternatively, a time limit per iteration for each type of filter can be set using

```
--csample time <time in seconds>
```

If not set, all possible Haar filters are evaluated. This should be avoided unless the images are small.

5.3 Meta base learners

MULTIBOOST implements two meta base learners that construct more complex structures using simple base learners. A decision tree (Section B.4) of a given number of leaves can be specified using

```
--learnertype TreeLearner --baselearnertype <base learner>\\
<number of leaves>
```

A decision product (Section B.5, [7]) of a given number of terms can be specified using

```
--learnertype ProductLearner --baselearnertype <base learner>\\
 <number of terms>
```

The product learner has an optional parameter

```
--noloop
```

which will make it stop after (at most) $m =$`<number of terms>` iterations. The default is false, so we will loop over all terms until the edge stops increasing (as specified in the pseudocode in Figure 12). Setting `--noloop` will make it faster but on certain data sets the performance will slightly decrease. The default setting is especially expensive for ternary classifiers, so we suggest that `--noloop` be set in this case.

6 Input file formats

The format of the input data sets can be specified using

```
--fileformat <format>
```

MULTIBOOST supports three formats:

1. `arff` for the Attribute-Relation File Format (ARFF),

2. `svmlight` for the SVM light format,

3. `simple` for a simple comma separated text file.

MULTIBOOST can read in the standard version of these formats. The following sections describe the extensions of these formats that allow inputting more complex information (such as the initial weighting) to MULTIBOOST.

6.1 Attribute-Relation File Format (ARFF)

The ARFF standard is described at

http://www.cs.waikato.ac.nz/~ml/weka/arff.html.

We support two attribute types: *nominal* and *numerical*. In the case of *nominal* or *categorical* feature, the possible feature values have to be enumerated. For example:

```
@relation nominal
@attribute t0 {-1,0,1,2}
@attribute t1 {-1,0,1,2}
@attribute t2 {-1,1}
```

Numerical features can be defined by `numerical`, `real`, or `integer`.

```
@relation numerical
@attribute t0 numerical
@attribute t1 integer
@attribute t2 real
```

Class names can be defined in two ways. In the *simple* format there is one nominal class attribute:

```
@attribute class {alma,korte,barack}
```

This format is detected by having a single attribute named `class`. This format supports both single-label data

```
@data
-1.0,0,3.4,alma
1.3,1,-5.5,korte
```

and multi-label data

```
@data
-1.0,0,3.4,alma,barack
1.3,1,-5.5,alma,korte
```

The number of input variables must be equal to the number of input attributes (that is, no way to have sparse data representation in this case). The semantics of both cases is that $y_{i,\ell} = 1$ if the ℓth label is in the label list of the ith row and $y_{i,\ell} = -1$ otherwise. All initial weights are set to $w_{i,\ell} = 1$.

In the *sparse/dense* format each class name is listed as a numerical attribute.

```
@attribute classalma NUMERICAL
@attribute classkorte NUMERICAL
@attribute classbarack NUMERICAL
```

Each class name has to be *prefixed* by `class`. Given a header in the sparse/dense format, the data section can be in two different formats. In the *dense format*, each data point has the exact same number of (comma-separated) entries as the number of attributes (including classes) in the header:

```
@data
-1.0,0,3.4,1,1,-1
1.3,1,-5.5,-1,0,1
```

The values of the input (non-class) attributes represent the entries of $\mathbf{x}_i$, and the values of the class attributes represent $w_{i,\ell}y_{i,\ell}$ (that is, the sign of the value is $y_{i,\ell}$ and its absolute value is $w_{i,\ell}$). The *sparse format* allows for ignoring zeros. Each entry in the data section now is an (index, value) pair separated by a space, and each row is now surrounded by curly brackets:

```
@data
{0 -1.0,2 3.4,3 1,4 1,5 -1}
{0 1.3,1 1,2 -5.5,3 -1,5 1}
```

The indices start at zero, and class indices simply continue after the input indices (they do not re-start at zero). The semantics is exactly the same as in the dense case. Non-enumerated values are considered zero, that is, for input attributes $x_i^{(j)} = 0$, and for class attributes $w_{i,\ell} = 0$ (and $y_{i,\ell}$ is undefined).

Note that Section 4.1.1 descibes the different policies of setting the *initial* weights $w_{i,\ell}^{(1)}$ in ADABOOST.MH starting from the *input* weights $w_{i,\ell}$ specified in the input file.

6.2 SVM light file format

The description of SVM light file format is available at `http://svmlight.joachims.org/`. Features can have string-valued names (so it is not necessary to index them from 1 to N as in the original svmlight format). If a feature is called "qid", it will not be used in the training. The labels have to be indexed from 0 to $K-1$, but if one uses additional header file, it can be given arbitrary label names.

The label names, feature names, and initial label weighting can be specified in a separate header file that can be given using

```
--headerfile <header file>
```

The header file consists of two or three lines: 1) the class labels separated by spaces, 2) the feature names separated by spaces, and 3) (optionally) the class weighting. The real numbers $w_1, \ldots, w_K$ in this last line set initial weights accoring to

$$w_{i,\ell} = w_\ell y_{i,\ell}$$

which are then converted to the normalized weigths $w_{i,\ell}^{(1)}$ according to the specified initial weighting policy (Section 4.1.1).

6.3 CSV file format

In this format the labels and floating point features can be given in a matrix form. Fields are separated by a special character that can be given using the `--d` option. If it is not specified, the default is `\t` and `\r`. There is no possibility to set the weights. If the `--classend` option is used, the last column of the dataset will be considered as class label, otherwise, by default, the first column is considered as the label.

7 Miscellaneous options

7.1 The output information

Some strong learners, such as ADABOOST.MH, allow the user the specify the performance metrics to be written in the output file in each iteration t using

`--outputinfo <file> <output list>`

where the `<output list>` corresponds to the concatenation of three letter codes. The following metrics are implemented:

e01 refers to the *one-error*

$$\widehat{R}_{\mathrm{I}}\left(\mathbf{f}^{(t)},\mathbf{X},\mathbf{Y}\right)=\frac{1}{n}\sum_{i=1}^{n}\mathrm{I}\left\{y_{i,\ell_{\mathbf{f}^{(t)}(\mathbf{x}_i)}}<0\right\}=\frac{1}{n}\sum_{i=1}^{n}\mathrm{I}\left\{\max_{\ell:y_{i,\ell}<0}f_{\ell}^{(t)}(\mathbf{x}_i)\geq\max_{\ell:y_{i,\ell}>0}f_{\ell}^{(t)}(\mathbf{x}_i)\right\}. \quad (3)$$

where

$$\ell_{\mathbf{f}^{(t)}}(\mathbf{x}_i)=\arg\max_{\ell'}f_{\ell'}^{(t)}(\mathbf{x}_i) \quad (4)$$

is the single label predicted by $\mathbf{f}^{(t)}$.[3] If only one label is positive (single-label multi-class), the error condition is equivalent to $\ell_{\mathbf{f}^{(t)}}(\mathbf{x}_i)\neq\ell_i$ where ℓ_i is the correct single label index of $\mathbf{x}_i$. If more than one labels are positive (multi-label), it is sufficient for a good prediction if the predicted label (4) is *one* of the positive labels.

w01 refers to the *weighted one-error*

$$\widehat{R}_{\mathrm{WI}}\left(\mathbf{f}^{(t)},\mathbf{X},\mathbf{Y}\right)=\sum_{i=1}^{n}\mathrm{I}\left\{y_{i,\ell_{\mathbf{f}^{(t)}(\mathbf{x}_i)}}<0\right\}\sum_{\ell=1}^{K}w_{i,\ell}^{(1)}.$$

r01 refers to the *restrictive one-error*

$$\widehat{R}_{\mathrm{RI}}\left(\mathbf{f},\mathbf{X},\mathbf{Y}\right)=\frac{1}{n}\sum_{i=1}^{n}\mathrm{I}\left\{\max_{\ell:y_{i,\ell}<0}f_{\ell}^{(t)}(\mathbf{x}_i)\geq\min_{\ell:y_{i,\ell}>0}f_{\ell}^{(t)}(\mathbf{x}_i)\right\}.$$

If only one label is positive (single-label multi-class), the error condition is equivalent to $\ell_{\mathbf{f}^{(t)}}(\mathbf{x}_i)\neq\ell_i$ where ℓ_i is the correct single label of $\mathbf{x}_i$, and so in this case the restrictive one-error is equivalent to the one-error. If more than one labels are positive (multi-label), the score of *all* positive labels must be greater than or equal to the score of *any* negative label, which means that this error is always greater than or equal to the one-error.

ham refers to the *Hamming error*

$$\widehat{R}_{\mathrm{H}}\left(\mathbf{f}^{(t)},\mathbf{X},\mathbf{Y}\right)=\sum_{i=1}^{n}\sum_{\ell=1}^{K}\mathrm{I}\left\{\mathrm{sign}\left(f_{\ell}^{(t)}(\mathbf{x}_i)\right)\neq y_{i,\ell}\right\}.$$

wha refers to the *weighted Hamming error*

$$\widehat{R}_{\mathrm{WH}}\left(\mathbf{f}^{(t)},\mathbf{X},\mathbf{Y}\right)=\sum_{i=1}^{n}\sum_{\ell=1}^{K}w_{i,\ell}^{(1)}\mathrm{I}\left\{\mathrm{sign}\left(f_{\ell}^{(t)}(\mathbf{x}_i)\right)\neq y_{i,\ell}\right\}.$$

auc refers the class-wise area under the ROC curve. Formally, let the *true positive rate* and *false negative rate* for class ℓ at a given threshold a be

$$\widehat{G}_{\mathrm{TP}}(\mathbf{f}^{(t)},\mathbf{X},\mathbf{Y},\ell,a)=\frac{\sum_{i=1}^{n}\mathrm{I}\left\{y_{i,\ell}=+1\wedge a\leq f_{\ell}^{(t)}(\mathbf{x}_i)\right\}}{\sum_{i=1}^{n}\mathrm{I}\left\{y_{i,\ell}=+1\right\}}$$

and

$$\widehat{R}_{\mathrm{FP}}(\mathbf{f}^{(t)},\mathbf{X},\mathbf{Y},\ell,a)=\frac{\sum_{i=1}^{n}\mathrm{I}\left\{y_{i,\ell}=-1\wedge a\leq f_{\ell}^{(t)}(\mathbf{x}_i)\right\}}{\sum_{i=1}^{n}\mathrm{I}\left\{y_{i,\ell}=-1\right\}},$$

[3] In case of ties at the maximum, which usually only happens in the first boosting iterations, the error is computed according to the second equality of (3), which means that the negative label "wins".

respectively. Let $j_{\ell,i}$ be the rank of $(\mathbf{x}_i, \mathbf{y}_i)$ when ordered according to $f_\ell^{(t)}(\mathbf{x}_i)$, that is, $f_\ell^{(t)}(\mathbf{x}_{j_{\ell,i}}) \leq f_\ell^{(t)}(\mathbf{x}_{j_{\ell,i'}})$ if $j_{\ell,i} < j_{\ell,i'}$. Denoting $\widehat{G}_{\text{TP}}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}, \ell, f_\ell^{(t)}(\mathbf{x}_{j_{\ell,i}}))$ by $g_{\ell,j_{\ell,i}}$ and $\widehat{R}_{\text{FP}}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}, \ell, f_\ell^{(t)}(\mathbf{x}_{j_{\ell,i}}))$ by $r_{\ell,j_{\ell,i}}$ and defining $(g_{\ell,0}, r_{\ell,0}) = (0,0)$, the monotonically increasing polygonal *receiver operating characteristic* (ROC) curve is defined by the points $(g_{\ell,j_{\ell,i}}, r_{\ell,j_{\ell,i}})$ and the *area under the ROC curve* is defined as

$$\begin{aligned}
\text{AUC}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}, \ell) &= \frac{1}{2}\sum_{i=1}^{n}(r_{\ell,j_{\ell,i}} - r_{\ell,j_{\ell,i-1}})(g_{\ell,j_{\ell,i}} + g_{\ell,j_{\ell,i-1}}) \\
&= \frac{1}{\sum_{i=1}^{n}\mathbb{I}\left\{y_{i,\ell} = -1\right\}}\sum_{i=1}^{n}\mathbb{I}\left\{r_{\ell,j_{\ell,i}} \neq r_{\ell,j_{\ell,i-1}}\right\}g_{\ell,j_{\ell,i}} \\
&= \frac{\sum_{i=1}^{n}\mathbb{I}\left\{y_{i,\ell} = -1\right\}g_{\ell,j_{\ell,i}}}{\sum_{i=1}^{n}\mathbb{I}\left\{y_{i,\ell} = -1\right\}}.
\end{aligned}$$

tfr refers to the class-wise true positive and false positive rates and their averages. The true positive rate for class ℓ is

$$\widehat{G}_{\text{TP}}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}, \ell) = \frac{\sum_{i=1}^{n}\mathbb{I}\left\{y_{i,\ell} = +1 \wedge \ell = \ell_{\mathbf{f}^{(t)}}(\mathbf{x}_i)\right\}}{\sum_{i=1}^{n}\mathbb{I}\left\{y_{i,\ell} = +1\right\}}.$$

Then the average true positive rate is

$$\widehat{G}_{\text{TP}}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}) = \frac{1}{K}\sum_{\ell=1}^{K}\widehat{G}_{\text{TP}}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}, \ell).$$

Similarly, the false positive rate for class ℓ is

$$\widehat{R}_{\text{FP}}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}, \ell) = \frac{\sum_{i=1}^{n}\mathbb{I}\left\{y_{i,\ell} = -1 \wedge \ell = \ell_{\mathbf{f}^{(t)}}(\mathbf{x}_i)\right\}}{\sum_{i=1}^{n}\mathbb{I}\left\{y_{i,\ell} = -1\right\}},$$

the average false positive rate is

$$\widehat{R}_{\text{FP}}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}) = \frac{1}{K}\sum_{\ell=1}^{K}\widehat{R}_{\text{FP}}(\mathbf{f}^{(t)}, \mathbf{X}, \mathbf{Y}, \ell).$$

In the case of class-wise metrics, the name of the class is indicated in the header of the columns.

edg refers to the *edge*

$$\gamma(\mathbf{h}^{(t)}, \mathbf{X}, \mathbf{Y}, \mathbf{W}^{(t)}) = \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell}^{(t)} h_\ell^{(t)}(x_i) y_{i,\ell}$$

of the base classifier $\mathbf{h}^{(t)}$.

The iteration counter and the ellapsed time are always written into the output file. If no `--outputinfo` is specified, in addition, the one-error is also written to the file called `results.dta` by default. When the `--traintest` option is used, the evaluation metrics are computed for both the training and test datasets. For example, the command

```
--outputinfo results.dta hamauc
```

outputs the iteration counter, and the ellapsed time, the iteration-wise Hamming error and AUC.

7.2 Logging

The `--verbose <level>` option can be used to control the amount of information written to the standard output. The logging level is between 0 and 5 (0 corresponds to the silent mode).

7.3 Saving the strong classifier

The strong classifier is serialized in XML format in each iteration. The name of the model file can be specifed by

```
--shypname <file name>
```

The default name is `shyp.xml`.

7.4 Resuming a run

Some strong learners, such as `AdaBoostMH` and `FilterBoost`, are anytime algorithms: they output the strong classifier and the logging information in each iteration and they can be resumed if they are interrupted. The model file can be specified using

```
--resume <file name>
```

By default, ADABOOST.MH and FILTERBOOST output the performance metrics starting from the resumed iteration. If there exists a file with the same name, this file will be appended by the errors starting from the resumed iteration. If it does not, it will be created and the errors will be output starting from the resumed iteration. In case the file is lost or broken, ADABOOST.MH can also be resumed using the `--slowresumeprocess` switch which will force the algorithm to reconstruct the full output file from the first iteration. In this case, the existing file will be deleted.

7.5 The constant learner

Turning the `--constant` switch on will force the algorithm to explicitly try the constant classifier $\varphi(\mathbf{x}) \equiv 1$ in each iteration, and use it if its edge is higher than the edge of any of the other base classifiers tested in the given iteration. The votes $\mathbf{v}$ of the constant classifier are computed according to Section B.1.

7.6 The seed

The random seed used to initialize the pseudorandom number generator can be set using

```
--seed <number>
```

8 Examples

8.1 Data sets used in examples

All the data sets used in this section are available at `http://www.multiboost.org`. Most of them are coming from the UCI Machine Learning Repository.

8.2 Basic examples

This simple example shows how ADABOOST.MH can be trained using various base learners, such as (real-valued) decision stumps (Section B.2), (categorical) subset indicators (Section B.3), decision trees (Section B.4), and decision products (Section B.5), on the UCI pendigits dataset. The command

```
./multiboost --fileformat arff --traintest pendigitsTrain.arff
pendigitsTest.arff 100 --verbose 3 --learnertype SingleStumpLearner
--outputinfo resultsSingleStump.dta --shypname shypSingleStump.xml
```

runs ADABOOST.MH over 100 iterations using decision stumps as base learners. The model will be saved into the `shypSingleStump.xml` file. The default metrics will be calculated for each iteration and stored in the `resultsSingleStump.dta` file. The command

```
./multiboost --fileformat arff --traintest trainNominal.arff
testNominal.arff 100 --verbose 3 --learnertype IndicatorLearner
--outputinfo resultsIndicator.dta --shypname shypIndicator.xml
```

runs ADABOOST.MH with subset indicators as base learners. The command

```
./multiboost --fileformat arff --traintest pendigitsTrain.arff
pendigitsTest.arff 100 --verbose 3 --learnertype TreeLearner
--baselearnertype SingleStumpLearner 8 --outputinfo
resultsPendigitTree.dta --shypname shypPendigitTree.xml
```

runs ADABOOST.MH with decision trees of (at most) 8 leaves, whereas the command

```
./multiboost --fileformat arff --traintest pendigitsTrain.arff
pendigitsTest.arff 100 --verbose 3 --learnertype ProductLearner
--baselearnertype SingleStumpLearner 3 --outputinfo
resultsPendigitProduct.dta --shypname shypPendigitProduct.xml
```

runs ADABOOST.MH with decision products of 3 terms.

8.3 FilterBoost and LazyBoost

The strong learner can be set by using the `--stronglearner` argument. The command

```
./multiboost --fileformat arff --traintest pendigitsTrain.arff
pendigitsTest.arff 100 --verbose 3 --learnertype SingleStumpLearner
--outputinfo resultsFilterSingleStump.dta --shypname
shypFilterSingleStump.xml --stronglearner FilterBoost
```

runs FILTERBOOST (Section A.4).

LAZYBOOST [8] can be run by choosing `AdaBoostMH` as the strong learner and setting the `--rsample` parameter to the number of features to be sampled in each iteration. This command

```
./multiboost --fileformat arff --traintest pendigitsTrain.arff
pendigitsTest.arff 100 --verbose 3 --learnertype SingleStumpLearner
--outputinfo resultsLazySingleStump.dta --shypname shypLazySingleStump.xml
--rsample 100
```

runs LAZYBOOST with 100 features sampled in each iteration.

8.4 Haar filters

The command

```
./multiboost --fileformat arff --traintest uspsHaarTrain.arff
uspsHaarTest.arff 100 --verbose 3 --learnertype TreeLearner
--baselearnertype HaarSingleStumpLearner 16 --csample num 100 --iisize 15x15
--seed 71 --outputinfo resultHaarFilters.dta --shypname shypHaarFilters.xml
```

runs ADABOOST.MH with stumps over Haar filters as base learners. The `uspsHaarTrain.arff` and `uspsHaarTest.arff` files contain integral images. The `--csample` argument sets the number of filters sampled in each iteration. The `--iisize` option specifies the size of the image. The `--seed` argument intializes the random number generator. For more information about Haar Filters, see [5].

8.5 Using bandits to accelerate AdaBoost

Bandit boosting (Section B.6; [9, 10]) is an alternative to LAZYBOOST that makes feature sampling more efficient. The command

```
./multiboost --fileformat arff --traintest pendigitsTrain.arff
pendigitsTest.arff 100 --verbose 3 --learnertype
BanditSingleStumpLearner --outputinfo resultsBanditsSingleStump.dta
--shypname shypBanditsSingleStump.xml --banditalgo UCBK
--updaterule logedge
```

boost a UCB-based decision stump base learner.

8.6 Sparse data

The command

```
./multiboost --fileformat svmlight --traintest train.txt test.txt 1000
--verbose 5 --learnertype SingleSparseStumpLearner --outputinfo
resultsSparseStump.dta --constant --shypname shypSparseStump.xml
 --weightpolicy proportional
```

runs ADABOOST.MH using decision stumps implemented for sparse features. The base learner is usually coupled with the `svmlight` input format which allows storing sparse data. The command

```
./multiboost --fileformat svmlight --traintest train.txt test.txt 1000
--verbose 5 --learnertype BanditSingleSparseStump --outputinfo --constant
resultsBanditsSparseStump.dta  --shypname shypBanditsSparseStump.xml
--weightpolicy proportional --banditalgo EXP3P --updaterule logedge
--rsample 2
```

runs the same learner but uses a bandit-based acceleration.

9 Developer notes

This section contains some high level notes for potential developers of MULTIBOOST. For further information we suggest that you contact the authors.

Figure 1 shows the modular architecture of MULTIBOOST. In general, each modul can be modified or complemented without modifying the other moduls.

9.1 Notational conventions

The code of MULTIBOOST is written in standard C++ and follows some notation guidelines:

- All the internal data and function members are documented using Doxygen[4] syntax, rare exceptions are tolerated when the semantics are clear from the signature of the function or the name of the member.
- All MULTIBOOST code belongs to the `MultiBoost` namespace.
- Private and protected data members are prefixed with an underscore.
- Pointer variables are prefixed with the letter `p`.

[4] http://www.doxygen.org

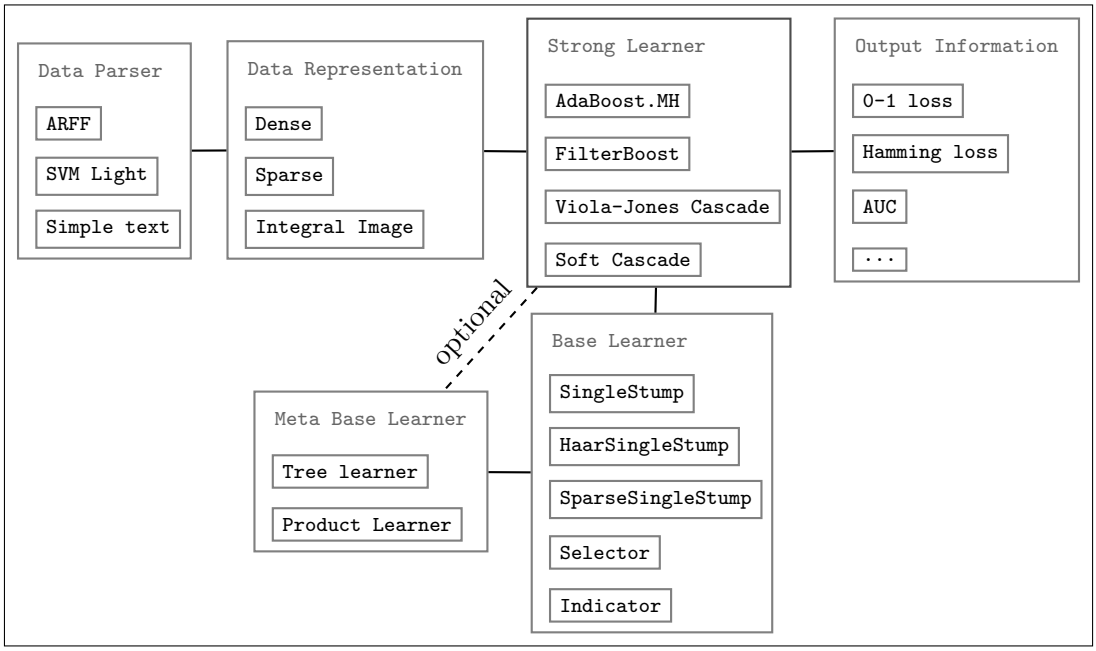

Figure 1: The architecture of MULTIBOOST

- The naming convention suggests to use letter-case seperated word identifiers (for the first letter, use a capital letter for class names and a lowercase letter for instances). Global constant names are in capital letters seperated by underscores. Enumeration names a prefixed with the letter `e`, their values are in capital letters and prefixed with an abbreviation of the enumeration name. For instance,

```
enum eTokenType
{
    TT_EOF,
    TT_COMMENT,
    TT_RELATION,
    TT_ATTRIBUTE,
    TT_DATA,
    TT_UNKNOWN
};
```

- Use `AlphaReal` and `FeatureReal` instead of `float` or `double` whenever possible. Those two `typedef`s are used respectively to describe strong learner output type ($\mathbf{f}(\mathbf{x})$, α, etc.) and the real-valued features $\mathbf{x}$ in the data sets.

9.2 Defining a new base learner

Base learners are implemented through the *factory design pattern* which makes it easy to add new base learners without modifying the existing code. The first steps are obviously to create new C++ files (.h and .cpp) inside the `WeakLearners` directory and declare the class of the new base learner, usually postfixed by the word `Learner` (such as `SingleStumpLearner`). To let MULTIBOOST know about the new base learner, the macro `REGISTER_LEARNER` should be added in the `Registrations.h` file:

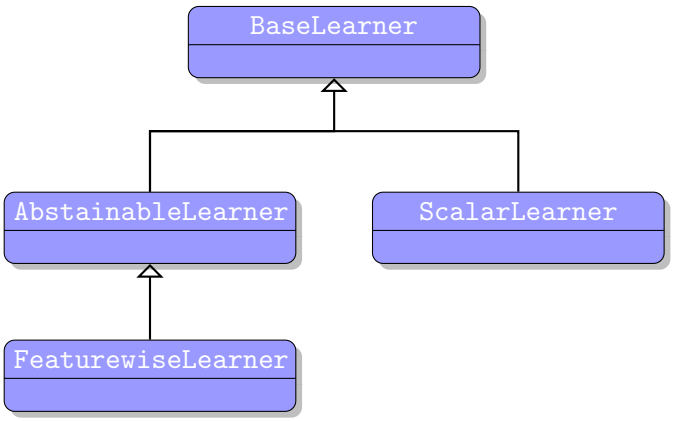

Figure 2: UML diagram of the base learners abstract class hierarchy.

```
...
namespace MultiBoost {
    REGISTER_LEARNER( <base learner name> )
...
```

The new class should inherit from one or more of the abstract base classes and implement the corresponding virtual methods. Figure 9.2 shows the most important subtree of the base learner inheritance structure of MULTIBOOST.

`BaseLearner` All base learners are derived from the `BaseLearner` abstract class therefore they must implement the following pure virtual methods (for more technical details, see the doxygen documentation):

- `virtual AlphaReal run()` is called to train the base learner.
- `virtual BaseLearner* subCreate()` returns an instance of the derived class itself.
- `virtual void subCopyState(BaseLearner *pBaseLearner)` clones the parameters of the trained base learner to `pBaseLearner`. Note that it is mandatory to call the `subCopyState()` function of the super-class explicitly.
- `virtual AlphaReal classify(InputData* pData, int idx, int classIdx)` performs the base classification.

The following non-pure virtual methods could (and probably should) also be overriden.

- `virtual void declareArguments(nor_utils::Args& args)` is used to handle user specified parameters by declaring them through the `Args` class.
- `virtual void initLearningOptions(const nor_utils::Args& args)` is called before the `run()` method for initializing the parameters of the base learner using parameters specified by the user.
- `virtual void save(ofstream& outputStream, int numTabs = 0)` saves the base classifier model to a shyp file (see 7.3).
- `virtual void load(nor_utils::StreamTokenizer& st)` loads a base classifier model from a shyp file (see 7.3).

The `save()` and `load()` methods use the `Serialization` class that facilitates XML writing/reading. We refer the reader to the doxygen documentation for more details about the `Args` and `Serialization` classes. Note that it is very important to call the super-class versions of the last four methods explicitly if they are overridden.

`AbstainableLearner` This abstract class represents the set of base learners that can be written in the form of $\mathbf{h}(\mathbf{x}) = \alpha \mathbf{v} \varphi(\mathbf{x})$ (see Section B) and have the possibility to "abstain" from classifying an instance into a particular class by setting the corresponding element of $\mathbf{v}$ to zero. The pure virtual method

```
virtual AlphaReal phi(InputData* pData, int idx, int classIdx) const
```

representing the binary discriminant function must be implemented within its direct or indirect subclasses.

`FeaturewiseLearner` This class represents base learners that learn classifiers that use only one feature for computing the $\varphi(.)$ function (e.g., decision stumps B.2). Accordingly, the pure virtual method

```
virtual AlphaReal phi(FeatureReal val, int classIdx) const
```

representing the binary discriminant function must be implemented within its direct or indirect subclasses.

`ScalarLearner` If a new base learner is to be used as the partitioning function in the `TreeLearner` meta base learner, the new base learner must inherit from this abstract base class and implement the pure virtual method

```
virtual AlphaReal cut( InputData* pData, int idx ) const
```

In `AbstainableLearners`, `cut()` usually coincides with the binary discriminant function `phi()`.

Note 1: Most of the implemented base learners inherit from both `FeaturewiseLearner` and `ScalarLearner`. If the new base class follows this pattern, *virtual* inheritance should be used (at the risk of falling into the diamond problem).

Note 2: If the base learner needs a specific data representation, a new class can be derived from `InputData` and override the member function

```
virtual void load(const string& fileName,
                  eInputType inputType = IT_TRAIN,
                  int verboseLevel = 1)
```

The base learner must then override the

```
virtual InputData* createInputData()
```

member function to return an instance of the new data class. `SortedData` for `SingleStumpLearner` and `HaarData` for `HaarLearner` are examples of specialized `InputData`.

9.3 Defining a new strong learner

To implement a new strong learner, one has to create some C++ files (`.h` and `.cpp`) in the `StrongLearners` directory, and then follow these steps:

- Derive a new class from `GenericStrongLearner`. Usually, the learner names are postfixed by the word `Learner`[5] (such as `AdaBoostMHLearner`)
- Implement the pure virtual function members of `GenericStrongLearner` and other specific functions if needed.
- If the learner needs specific command line options, implement a static method

  `declareBaseArguments()`

  and call this method from the main.
- Finally, to make the new strong learner accessible, add a conditional statement to the method

  `BaseLearner::createGenericStrongLearner`

9.4 Defining a new output information

To implement a new output routine, one must derive a new class from `BaseOutputInfoType` and implement its two pure virtual methods. The naming conventions encourage to postfix the name of the new class by `Output`, as for `ErrorOutput`, `AUCOutput`, etc. Optionally, one can add other variables and methods if needed. They will be accessed through the method

`OutputInfo::getOutputInfoObject()`

Finally, the factory has to know about the new implemented class by adding a new conditional statement to the method

`BaseOutputInfoType::createOutput()`

[5] There is no ambiguity between the base/strong learners name since the semantics is clear from the name of the class.

A Multi-class ADABOOST

The original ADABOOST paper of Freund and Schapire [11] also described two multi-class extensions, ADABOOST.M1 and ADABOOST.M2. Both required a quite strong performance from the base learners, partly defeating the purpose of boosting. The breakthrough came with Schapire and Singer's seminal paper [1], which described, among other interesting algorithms, ADABOOST.MH. In this section we first introduce the general multi-class learning setup (Section A.1), then we give the details and ADABOOST.MH and show its algorithmic convergence (Section A.2).

A.1 The multi-class setup: single-label, multi-label, and multi-task

The goal of supervised learning is to infer a function $g : \mathcal{X} \to \mathcal{L}$ from a data set

$$\mathcal{D} = \big\{(\mathbf{x}_1, \ell_1), \ldots, (\mathbf{x}_n, \ell_n)\big\} \in (\mathcal{X} \times \mathcal{L})^n$$

comprised of pairs of *observations* and *labels*. The elements $x^{(j)}$ of d-dimensional *feature* or *observation* vector $\mathbf{x}$ are either real numbers or they come from an unordered set of cardinality M_j. In this latter case, without loss of generality, we will assume that $x^{(j)} \in \mathcal{I}^{(j)} = \{1, \ldots, M^{(j)}\}$. When the label space $\mathcal{L}$ is real-valued, the problem is known as *regression*, whereas when the labels come from a finite set of classes, we are talking about *classification*.

In *multi-class* classification, the label ℓ of the observation $\mathbf{x}$ comes from a finite set. Without loss of generality, we will suppose that $\ell \in \mathcal{L} = \{1, \ldots, K\}$. We will refer to the label of the observation $\mathbf{x}$ as $\ell(\mathbf{x})$, or by ℓ_i, when talking about the ith data point $\mathbf{x}_i$. For technical reasons that will become clear later, we will encode the label using a K-dimensional binary vector $\mathbf{y} \in \mathcal{Y} = \{\pm 1\}^K$. In the classical multi-class setup, $\mathbf{y}$ is known as the *one-hot* representation: the $\ell(\mathbf{x})$th element of $\mathbf{y}$ will be 1 and all the other elements will be -1, that is,

$$y_\ell = \begin{cases} +1 & \text{if } \ell = \ell(\mathbf{x}), \\ -1 & \text{otherwise.} \end{cases}$$

Beside reflecting well the architecture of multi-class neural network or multi-class ADABOOST, this representation has the advantage to be generalizable to *multi-label* or *multi-task* learning when and observation $\mathbf{x}$ can belong to several classes. To avoid confusion, from now on we will call $\mathbf{y}$ and ℓ the label and the label *index* of $\mathbf{x}$, respectively. For emphasizing the distinction between multi-class and multi-label classification, we will use the term *single-label* for the classical multi-class setup, and reserve multi-class to situations we talk about the three setups in general.

The general multi-class training data is thus

$$\mathcal{D} = \big\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\big\} \in (\mathcal{X} \times \mathcal{Y})^n.$$

and the goal of learning is to infer a vector-valued multi-class classifier $\mathbf{g} : \mathcal{X} \to \mathcal{Y}$ from $\mathcal{D}$. Sometimes we will use the notion of an $n \times d$ *observation matrix* of $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and an $n \times K$ *label matrix* $\mathbf{Y} = (\mathbf{y}_1, \ldots, \mathbf{y}_n)$ instead of the set of pairs $\mathcal{D}$.

As in binary classification, learning algorithms usually output a multi-class *discriminant* function $\mathbf{f} : \mathcal{X} \to \mathbb{R}^K$. The semantics of $\mathbf{f}$ is that the higher the ℓth element $f_\ell(\mathbf{x})$ of $\mathbf{f}(\mathbf{x})$, the more likely is that the real label index of $\mathbf{x}$ is ℓ. The vector-valued classifier $\mathbf{g}$ is formally obtained by simply thresholding the elements of $\mathbf{f}(\mathbf{x})$ at 0, that is,

$$g_\ell(\mathbf{x}) = \operatorname{sign}\big(f_\ell(\mathbf{x})\big), \quad \ell = 1, \ldots, K.$$

The single-label output of the algorithm is then the class index ℓ for which $f_\ell(\mathbf{x})$ is maximal, that is,

$$\ell_{\mathbf{f}}(\mathbf{x}) = \arg\max_\ell f_\ell(\mathbf{x}).$$

The classical measure of the performance of the multi-class discriminant function $\mathbf{f}$ is the *single-label one-loss*

$$L_{\mathrm{I}}\big(\mathbf{f},(\mathbf{x},\ell)\big) = \mathbb{I}\left\{\ell \neq \ell_{\mathbf{f}}(\mathbf{x}_i)\right\}$$

that defines the single-label training error

$$\widehat{R}_{\mathrm{I}}(\mathbf{f}) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}\left\{\ell_i \neq \ell_{\mathbf{f}}(\mathbf{x}_i)\right\}. \tag{5}$$

Another, perhaps more comprehensive, way to measure of the performance of $\mathbf{f}$ is by computing the *weighted Hamming loss*

$$L_{\mathrm{H}}\big(\mathbf{f},(\mathbf{x},\mathbf{y}),\mathbf{w}\big) = \sum_{\ell=1}^{K} w_\ell \mathbb{I}\left\{\mathrm{sign}\big(f_\ell(\mathbf{x})\big) \neq y_\ell\right\}$$

where $\mathbf{w} = [w_\ell]_{\ell=1\ldots,K}$ is an $\mathbb{R}^k$-valued "user-defined" weight vector over labels. The corresponding empirical risk (training error) is

$$\widehat{R}_{\mathrm{H}}(\mathbf{f},\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell}\mathbb{I}\left\{\mathrm{sign}\big(f_\ell(\mathbf{x}_i)\big) \neq y_{i,\ell}\right\}, \tag{6}$$

where $\mathbf{W} = \left[w_{i,\ell}\right]_{i=1\ldots,n}^{\ell=1\ldots,K}$ is an $n \times k$ weight matrix over data points and labels.

In the multi-label/multi-task setup, when, for example, it is equally important to predict that a song is "folk" as predicting that it is sung by a woman, the Hamming loss with uniform weights

$$w_\ell = \frac{1}{K}, \quad \ell = 1,\ldots,K \tag{7}$$

is a natural measure of performance: it represents the uniform error rate of missing any class sign y_ℓ of a given observation $\mathbf{x}$. In single-label classification, $\mathbf{w}$ is usually set asymmetrically to

$$w_\ell = \begin{cases} \frac{1}{2} & \text{if } \ell = \ell(\mathbf{x}) \text{ (i.e., if } y_\ell = 1), \\ \frac{1}{2(K-1)} & \text{otherwise (i.e., if } y_\ell = -1). \end{cases} \tag{8}$$

The idea behind this scheme is that it will create K well-balanced *one-against-all* binary classification problems: if each of the K classes have n/K examples in $\mathcal{D}$, then for each class ℓ, the sum of the weights of the positive examples in the column $\mathbf{w}_{\cdot,\ell}$ of the weight matrix $\mathbf{W}$ will be equal to the sum of the weights of the negative examples. Indeed, for positive examples in class ℓ we have

$$\frac{1}{n}\sum_{i=1}^{n} w_{i,\ell}\mathbb{I}\left\{y_{i,\ell} = 1\right\} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\mathbb{I}\left\{y_{i,\ell} = 1\right\} = \frac{1}{n}\times\frac{1}{2}\times\frac{n}{K} = \frac{1}{2K},$$

and for negative examples of the same column we have

$$\frac{1}{n}\sum_{i=1}^{n} w_{i,\ell}\mathbb{I}\left\{y_{i,\ell} = -1\right\} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2(K-1)}\mathbb{I}\left\{y_{i,\ell} = -1\right\} = \frac{1}{n}\times\frac{1}{2(K-1)}\times\frac{n(K-1)}{K} = \frac{1}{2K}.$$

Note that both schemes (7) and (8) boil down to the classical uniform weighting in binary classification.

It is easy to see that if the Hamming error $\widehat{R}_{\mathrm{H}}(\mathbf{f},\mathbf{W})$ is 0 then the one-error $\widehat{R}_{\mathrm{I}}(\mathbf{f})$ is also 0, but not the other way around: it is possible that a point is correctly classified, that is $f_{\ell(\mathbf{x})}(\mathbf{x}) > f_\ell(\mathbf{x})$ for all $\ell \neq \ell(\mathbf{x})$, yet several of the negative classes $\ell : y_\ell = -1$ have positive outputs $f_\ell(\mathbf{x})$. When both $\widehat{R}_{\mathrm{H}}(\mathbf{f},\mathbf{W})$ and $\widehat{R}_{\mathrm{I}}(\mathbf{f})$ are nonzero, both $\widehat{R}_{\mathrm{H}}(\mathbf{f},\mathbf{W}) > \widehat{R}_{\mathrm{I}}(\mathbf{f})$ and $\widehat{R}_{\mathrm{H}}(\mathbf{f},\mathbf{W}) < \widehat{R}_{\mathrm{I}}(\mathbf{f})$ can happen. What can be said is that for every misclassified point according to $\widehat{R}_{\mathrm{I}}(\mathbf{f})$, $\mathbf{f}(\mathbf{x})$ misses the sign of at least one of the labels y_ℓ, so $\widehat{R}_{\mathrm{I}}(\mathbf{f}) < K\widehat{R}_{\mathrm{H}}(\mathbf{f},\mathbf{W})$ for (7) and $\widehat{R}_{\mathrm{I}}(\mathbf{f}) < 2K\widehat{R}_{\mathrm{H}}(\mathbf{f},\mathbf{W})$ for (8).

A.2 AdaBoost.MH

The goal of the AdaBoost.MH algorithm ([1], Figure 3) is to return a vector-valued discriminant function $\mathbf{f}^{(T)} : \mathcal{X} \to \mathbb{R}^K$ with a small Hamming loss $\widehat{R}_{\mathrm{H}}(\mathbf{f}, \mathbf{W})$ (6) by minimizing the *weighted multi-class exponential margin-based error*

$$\widehat{R}_{\mathrm{EXP}}\left(\mathbf{f}^{(T)}, \mathbf{W}\right) = \frac{1}{n} \sum_{i=1}^{n} L_{\mathrm{EXP}}\left(\mathbf{f}^{(T)}, (\mathbf{x}, \mathbf{y}), \mathbf{w}_i\right) \tag{9}$$

with the convex loss L_{EXP} defined as

$$L_{\mathrm{EXP}}\left(\mathbf{f}, (\mathbf{x}, \mathbf{y}), \mathbf{w}\right) = \sum_{\ell=1}^{K} w_\ell \exp\left(-f_\ell(\mathbf{x}) y_\ell\right).$$

Since $\exp(-\rho) \geq \mathbb{I}\{\rho < 0\}$, the exponential loss $L_{\mathrm{EXP}}\left(\mathbf{f}, (\mathbf{x}, \mathbf{y}), \mathbf{w}\right)$ upper bounds the Hamming loss $L_{\mathrm{H}}\left(\mathbf{f}, (\mathbf{x}, \mathbf{y}), \mathbf{w}\right)$, and so

$$\widehat{R}_{\mathrm{EXP}}\left(\mathbf{f}^{(T)}, \mathbf{W}\right) \geq \widehat{R}_{\mathrm{H}}\left(\mathbf{f}^{(T)}, \mathbf{W}\right).$$

AdaBoost.MH builds the final discriminant function

$$\mathbf{f}^{(T)}(\mathbf{x}) = \sum_{t=1}^{T} \mathbf{h}^{(t)}(\mathbf{x}) \tag{10}$$

as a sum of T *base classifiers* $\mathbf{h}^{(t)} : \mathcal{X} \to \mathbb{R}^K$ returned by a *base learner* algorithm $\textsc{Base}\left(\mathbf{X}, \mathbf{Y}, \mathbf{W}^{(t)}\right)$ in each iteration t.

AdaBoost.MH$(\mathbf{X}, \mathbf{Y}, \mathbf{W}, \textsc{Base}(\cdot,\cdot,\cdot), T)$

1 $\quad \mathbf{W}^{(1)} \leftarrow \frac{1}{n}\mathbf{W}$

2 $\quad$ **for** $t \leftarrow 1$ **to** T

3 $\quad\quad \left(\alpha^{(t)}, \mathbf{v}^{(t)}, \varphi^{(t)}(\cdot)\right) \leftarrow \textsc{Base}\left(\mathbf{X}, \mathbf{Y}, \mathbf{W}^{(t)}\right)$

4 $\quad\quad \mathbf{h}^{(t)}(\cdot) \leftarrow \alpha^{(t)} \mathbf{v}^{(t)} \varphi^{(t)}(\cdot)$

5 $\quad\quad$ **for** $i \leftarrow 1$ **to** n **for** $\ell \leftarrow 1$ **to** K

6 $\quad\quad\quad w_{i,\ell}^{(t+1)} \leftarrow w_{i,\ell}^{(t)} \underbrace{\dfrac{\exp\left(-h_\ell^{(t)}(\mathbf{x}_i) y_{i,\ell}\right)}{\sum_{i'=1}^{n} \sum_{\ell'=1}^{K} w_{i',\ell'}^{(t)} \exp\left(-h_{\ell'}^{(t)}(\mathbf{x}_{i'}) y_{i',\ell'}\right)}}_{Z\left(\mathbf{h}^{(t)}, \mathbf{W}^{(t)}\right)}$

7 $\quad$ **return** $\mathbf{f}^{(T)}(\cdot) = \sum_{t=1}^{T} \mathbf{h}^{(t)}(\cdot)$

Figure 3: The pseudocode of the AdaBoost.MH algorithm. $\mathbf{X}$ is the $n \times d$ observation matrix, $\mathbf{Y}$ is the $n \times K$ label matrix, $\mathbf{W}$ is the user-defined weight matrix used in the definition of the weighted Hamming error (6) and the weighted exponential margin-based error (9), $\textsc{Base}(\cdot,\cdot,\cdot)$ is the base learner algorithm, and T is the number of iterations. $\alpha^{(t)}$ is the base coefficient, $\mathbf{v}^{(t)}$ is the vote vector, $\varphi^{(t)}(\cdot)$ is the scalar base (weak) classifier, $\mathbf{h}^{(t)}(\cdot)$ is the vector-valued base classifier, and $\mathbf{f}^{(T)}(\cdot)$ is the final (strong) discriminant function.

A.2.1 Algorithmic convergence

The following derivation shows that

$$\widehat{R}_{\mathrm{EXP}}\left(\mathbf{f}^{(T)}, \mathbf{W}\right) = \prod_{t=1}^{T} Z\left(\mathbf{h}^{(t)}, \mathbf{W}^{(t)}\right),$$

where

$$Z\big(\mathbf{h}, \mathbf{W}^{(t)}\big) = \sum_{i=1}^{n} \sum_{\ell=1}^{K} w_{i,\ell}^{(t)} \exp\left(-h_{\ell_i} y_{i,\ell}\right) \tag{11}$$

is the *base objective*. This means that the goal of the base learner in iteration t is to minimize $Z\big(\mathbf{h}, \mathbf{W}^{(t)}\big)$ in $\mathbf{h}$. Minimizing the base objective (11) in each iteration is therefore equivalent to minimizing the weighted multi-class exponential margin-based error (9) in an iterative greedy fashion. Indeed,

$$\begin{aligned}
\widehat{R}_{\text{EXP}}\big(\mathbf{f}^{(T)}, \mathbf{W}\big) &= \frac{1}{n} \sum_{i=1}^{n} \sum_{\ell=1}^{K} w_{i,\ell} \exp\left(-f_{\ell_i}^{(T)} y_{i,\ell}\right) && (12)\\
&= \sum_{i=1}^{n} \sum_{\ell=1}^{K} w_{i,\ell}^{(1)} \exp\left(-f_{\ell_i}^{(T)} y_{i,\ell}\right) && (13)\\
&= Z\big(\mathbf{h}^{(1)}, \mathbf{W}^{(1)}\big) \sum_{i=1}^{n} \sum_{\ell=1}^{K} w_{i,\ell}^{(2)} \frac{\exp\left(-f_{\ell_i}^{(T)} y_{i,\ell}\right)}{\exp\left(-h_{\ell}^{(1)}(\mathbf{x}_i) y_{i,\ell}\right)} && (14)\\
&\vdots \\
&= \prod_{t=1}^{T} Z\big(\mathbf{h}^{(t)}, \mathbf{W}^{(t)}\big) \sum_{i=1}^{n} \sum_{\ell=1}^{K} w_{i,\ell}^{(T+1)} \frac{\exp\left(-f_{\ell_i}^{(T)} y_{i,\ell}\right)}{\prod_{t=1}^{T} \exp\left(-h_{\ell}^{(t)}(\mathbf{x}_i) y_{i,\ell}\right)} && (15)\\
&= \prod_{t=1}^{T} Z\big(\mathbf{h}^{(t)}, \mathbf{W}^{(t)}\big) \sum_{i=1}^{n} \sum_{\ell=1}^{K} w_{i,\ell}^{(T+1)} && (16)\\
&= \prod_{t=1}^{T} Z\big(\mathbf{h}^{(t)}, \mathbf{W}^{(t)}\big). && (17)
\end{aligned}$$

In (12) we use the definition of the weighted exponential margin-based error (9). (13) follows from the weight initialization of line 1. In (14)-(15) we repeatedly apply the weight update formula of line 6. In (16) we use the definition (10). Finally, (17) follows from the fact that, by lines 1 and 6, the weight matrix $\mathbf{W}^{(t)}$ remains normalized

$$\sum_{i=1}^{n} \sum_{\ell=1}^{K} w_{i,\ell}^{(t)} = 1$$

in each iteration t.

In binary ADABOOST (Figure 4) with binary base classifiers $h(\mathbf{x}) \in \{\pm 1\}$ and coefficients α, (11) simplifies to

$$\begin{aligned}
Z(h, \alpha) &= \sum_{i=1}^{n} w_i \exp\left(-\alpha h(\mathbf{x}_i) y_i\right) \\
&= \sum_{i=1}^{n} w_i \mathbb{I}\left\{h(\mathbf{x}_i) = y_i\right\} \exp(-\alpha) + \sum_{i=1}^{n} w_i \mathbb{I}\left\{h(\mathbf{x}_i) \neq y_i\right\} \exp(\alpha) \\
&= (1 - \epsilon) \exp(-\alpha) + \epsilon \exp(\alpha), && (18)
\end{aligned}$$

where

$$\epsilon = \epsilon(h, \mathbf{w}) = \sum_{i=1}^{n} w_i \mathbb{I}\left\{h(\mathbf{x}_i) \neq y_i\right\}$$

is the weighted error of the base classifier h. It is easy to see that for a given h, $Z\big(h, \alpha\big)$ is minimized by

$$\alpha = \frac{1}{2} \log \frac{1 - \epsilon}{\epsilon} = \frac{1}{2} \log \frac{1 + \gamma}{1 - \gamma}. \tag{19}$$

where

$$\gamma = \gamma(h, \mathbf{w}) = 1 - 2\epsilon = \sum_{i=1}^{n} w_i h(\mathbf{x}_i) y_i$$

is the so called *edge* of the base classifier $h(\mathbf{x}_i)$. Plugging (19) back into (18), we have

$$Z(h, \alpha) = (1-\epsilon)\sqrt{\frac{\epsilon}{1-\epsilon}} + \epsilon\sqrt{\frac{1-\epsilon}{\epsilon}} = 2\sqrt{\epsilon(1-\epsilon)} = \sqrt{1-\gamma^2}.$$

Minimizing $Z\big(h, \alpha\big)$ is therefore equivalent to maximizing the edge γ or minimizing the weighted error ϵ. The chain of minimizing the exponential error $\widehat{R}_{\text{EXP}}\big(f^{(T)}\big) \rightarrow$ minimizing the base objective $Z(h, \alpha) \rightarrow$ minimizing the weighted error $\epsilon(h, \mathbf{w}) \rightarrow$ setting α to (19) explains the formulas of binary AdaBoost that might have been arbitrary-looking at the first sight. It is also easy to see that if

$$\gamma^{(t)} \triangleq \gamma\big(h^{(t)}, \mathbf{w}^{(t)}\big) \geq \delta > 0$$

(or, equivalently, if $\epsilon^{(t)} < \frac{1}{2} - \frac{\delta}{2}$), then the binary margin-based exponential error can be bounded from above by

$$\widehat{R}_{\text{EXP}}\big(f^{(T)}\big) \leq \sqrt{1-\delta^2}^{\,T} \leq \exp\left(-\frac{\delta^2 T}{2}\right), \tag{20}$$

where the second inequality follows from $1 - x \leq \exp(-x)$. This means that $\widehat{R}_{\text{EXP}}\big(f^{(T)}\big)$ becomes smaller than $\frac{1}{n}$ after at most

$$T^* = \left\lceil \frac{2\log n}{\delta^2} \right\rceil + 1$$

iterations. Since $\widehat{R}_{\text{EXP}}\big(f^{(T)}\big)$ is an upper bound of the training error $\widehat{R}_{\text{I}}\big(f^{(T)}\big)$, and since the smallest non-zero value of $\widehat{R}_{\text{I}}\big(f^{(T)}\big)$ is $\frac{1}{n}$, after T^* iterations $f^{(T)}$ cannot commit any error on the data set $\mathcal{D}$. To summarize, if the base classifiers $h^{(t)}$ are slightly better than a random guess, the final classifier $f^{(T)}$ has zero training error in a number of steps that is logarithmic in the size n of the data set $\mathcal{D}$.

In the general multi-class setup, base learning is more complicated (although still elementary) so we devote a separate section for the technical details (Section B). But even without understanding how $\mathbf{h}$ is found, it can be seen, similarly to the binary case, that if $Z\big(\mathbf{h}, \mathbf{W}^{(t)}\big) \leq \sqrt{1-\delta^2}$, then the exponential error $\widehat{R}_{\text{EXP}}\big(\mathbf{f}^{(T)}, \mathbf{W}\big)$ (9) is also bounded above by (20). The smallest nonzero multi-class Hamming loss $\widehat{R}_{\text{H}}(\mathbf{f}, \mathbf{W})$ (6) is $w_{\min}/n$ where

$$w_{\min} = \min_{i,\ell : w_{i,\ell} \neq 0} w_{i,\ell}$$

is the smallest weight in the initial weight matrix $\mathbf{W}$, so $\widehat{R}_{\text{H}}(\mathbf{f}, \mathbf{W})$ will be zero after at most

$$T^* = \left\lceil \frac{2\log(n/w_{\min})}{\delta^2} \right\rceil + 1$$

iterations. If $\mathbf{W}$ is set to (7), then $w_{\min} = \frac{1}{K}$, so

$$T^* = \left\lceil \frac{2\log(nK)}{\delta^2} \right\rceil + 1, \tag{21}$$

whereas if $\mathbf{W}$ is set to (8), then $w_{\min} = \frac{1}{2(K-1)}$, so

$$T^* = \left\lceil \frac{2\log\big(2n(K-1)\big)}{\delta^2} \right\rceil + 1.$$

ADABOOST$\big(\mathcal{D}_n, \text{BASE}(\cdot,\cdot), T\big)$

$\mathbf{w}^{(1)} \leftarrow (1/n, \ldots, 1/n)$ ▷ *initial weights*

for $t \leftarrow 1$ **to** T

  $h^{(t)} \leftarrow \text{BASE}\big(\mathcal{D}_n, \mathbf{w}^{(t)}\big)$ ▷ *base classifier*

  $\epsilon^{(t)} \leftarrow \sum_{i=1}^{n} w_i^{(t)} \mathbb{I}\left\{h^{(t)}(\mathbf{x}_i) \neq y_i\right\}$ ▷ *weighted error of the base classifier*

  $\alpha^{(t)} \leftarrow \frac{1}{2} \ln\left(\frac{1-\epsilon^{(t)}}{\epsilon^{(t)}}\right)$ ▷ *coefficient of the base classifier*

  for $i \leftarrow 1$ **to** n ▷ *re-weighting the training points*

    if $h^{(t)}(\mathbf{x}_i) \neq y_i$ **then** ▷ *error*

      $w_i^{(t+1)} \leftarrow \frac{w_i^{(t)}}{2\epsilon^{(t)}}$ ▷ *weight increases*

    else ▷ *correct classification*

      $w_i^{(t+1)} \leftarrow \frac{w_i^{(t)}}{2\left(1-\epsilon^{(t)}\right)}$ ▷ *weight decreases*

return $f^{(T)}(\cdot) = \sum_{t=1}^{T} \alpha^{(t)} h^{(t)}(\cdot)$ ▷ *weighted "vote" of base classifiers*

Figure 4: The pseudocode of binary ADABOOST. $\mathcal{D} = \big\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\big\}$ is the training set, $\text{BASE}(\cdot,\cdot)$ is the base learner, and T is the number of iterations.

We showed at the end of Section A.1 that the one-error $\widehat{R}_{\mathbb{I}}(\mathbf{f})$ (5) can be bounded from above by $K\widehat{R}_{\text{H}}(\mathbf{f}, \mathbf{W})$. This bound implies the same limit (21) as minimum weight argument. On the other hand, it can be shown [1] that with the single-label initialization (8),

$$\widehat{R}_{\mathbb{I}}\big(\mathbf{f}^{(T)}\big) \leq \sqrt{K-1} \prod_{t=1}^{T} Z\big(\mathbf{h}^{(t)}, \mathbf{W}^{(t)}\big)$$

which implies the limit

$$T^* = \left\lceil \frac{2\log\left(n\sqrt{K-1}\right)}{\delta^2} \right\rceil + 1.$$

This result is another motivation for setting the weights to (8). Note, however, that these bounds have very few practical implications: on the one hand, the training error usually becomes 0 much before T^*, and on the other hand, ADABOOST is usually trained much longer after the training error becomes 0.

A.3 ARC-GV

Breiman's ARC-GV [2, 3] can be interpreted as ADABOOST.MH with an L_1 weight decay on the base classifier coefficients with an adaptive weight decay coefficient. Formally, let

$$\tilde{\rho}_{i,\ell}^{(t)} = \tilde{f}_{\ell_i}^{(t)} y_{i,\ell} \triangleq \sum_{\tau=1}^{t} \tilde{\alpha}^{(\tau)} h_{\ell}^{(\tau)}(x_i) y_{i,\ell}$$

be the *normalized margin* of the training point $\mathbf{x}_i$ on label $y_{i,\ell}$ at iteration t, where

$$\tilde{\alpha}^{(t)} = \frac{\alpha^{(t)}}{\sum_{\tau=1}^{t} \alpha^{(\tau)}}$$

are the *normalized* base coefficients, and let

$$\tilde{\rho}_{\min}^{(t)} = \min_{i,\ell} \tilde{\rho}_{i,\ell}^{(t)}$$

be the *minimum (normalized) margin* across training points and labels. Then we set the base coefficient to

$$\alpha^{(t)} = \frac{1}{2}\log\frac{1+\gamma^{(t)}}{1-\gamma^{(t)}} - \frac{1}{2}\log\frac{1+\tilde{\rho}_{\min}^{(t)}}{1-\tilde{\rho}_{\min}^{(t)}}, \tag{22}$$

and we change nothing else in AdaBoost.MH. Notice that

$$\begin{aligned}
\tilde{\rho}_{\min}^{(t)} \triangleq \min_{i,\ell} \tilde{\rho}_{i,\ell}^{(t)} \triangleq \min_{i,\ell} \sum_{\tau=1}^{t} \tilde{\alpha}^{(\tau)} h_\ell^{(\tau)}(x_i) y_{i,\ell} \leq \\
\max_{\alpha':\|\boldsymbol{\alpha}'\|_1=1} \min_{i,\ell} \sum_{\tau=1}^{t} \alpha'^{(\tau)} h_\ell^{(\tau)}(x_i) y_{i,\ell} \leq \min_{w':\|\mathbf{w}'\|_1=1} \max_{\mathbf{h}} \sum_{i=1}^{n}\sum_{\ell=1}^{K} w'_{i,\ell} h_\ell(x_i) y_{i,\ell} \qquad (23) \\
\leq \max_{\mathbf{h}} \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell}^{(t)} h_\ell(x_i) y_{i,\ell} = \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell}^{(t)} h_\ell^{(t)}(x_i) y_{i,\ell} \triangleq \gamma^{(t)},
\end{aligned}$$

where the crucial inequality (23) follows from the Min-Max-Theorem of von Neumann [12, 13], so $\alpha^{(t)}$ is positive. Furthermore, $\tilde{\rho}_{\min}^{(t)} = -1$ as long as there exists an $(\mathbf{x}_i, y_{i,\ell})$ pair for which all base classifiers $h^{(\tau)}$, $\tau = 1, \ldots, t$ we have $h^{(\tau)}(\mathbf{x}_i) y_{i,\ell} = -1$. To avoid this singularity, we set the base coefficient to

$$\alpha^{(t)} = \frac{1}{2}\log\frac{1+\gamma^{(t)}}{1-\gamma^{(t)}} - \frac{1}{2}\log\frac{1+\max\left(\theta_{\min}, \tilde{\rho}_{\min}^{(t)}\right)}{1-\max\left(\theta_{\min}, \tilde{\rho}_{\min}^{(t)}\right)},$$

where $\theta_{\min}$ is a user-provided input parameter. The default in MultiBoost is $\theta_{\min} = 0$ which means that we run AdaBoost.MH as long as the training error is larger then zero and Arc-gv afterwards.

The significance of Arc-gv is that it maximizes the minimum margin $\tilde{\rho}_{\min}^{(t)}$ asymptotically. In practice, AdaBoost.MH often does it also, but it is easy to construct counter-examples when it does not happen [14]. From the point of view of the generalization error, Arc-gv is often worse than AdaBoost.MH. The original explanation of this phenomenon was that maximizing the minimum margin aggressively is not necessarily advantageous. More recently, [15] proposed a more refined argument: Arc-gv seems to pick more complex base classifiers to reach the same minimum margin than AdaBoost.MH.

A.4 FilterBoost

FilterBoost [4] is an accelerated version of AdaBoost.MH. The main idea of FilterBoost is to filter the training examples in each iteration and give the base learner only this smaller, filtered subset of the original training dataset, leading to fast base learning. The edge of the base classifier is also estimated on a subset of the training set resulting in further improvement in speed.

The algorithm in Figure 5 is implemented according to the practical suggestions of the authors of FilterBoost found in the experimental section of [4]. This means that 1) the number of randomly selected instances is $C\log(t+1)$ in the tth iteration, and 2) in the Filter method the instances are first selected randomly then re-weighted based on their margins.

A.5 Cascade learners

In the last decade the *cascade learning* algorithms have gained great importance in many practical applications (e.g., object detection or trigger design in experimental physics). The cascade architecture is typically used for solving classification tasks where there are many so-called *background*

FILTERBOOST$(\mathbf{X}, \mathbf{Y}, \text{BASE}(\cdot,\cdot,\cdot), T, C)$

$\mathbf{f}^{(0)}(\mathbf{x}) \leftarrow 0$
for $t \leftarrow 1$ **to** T
  $C_t \leftarrow C\log(t+1)$
  $(\mathbf{X}', \mathbf{Y}', \mathbf{W}') \leftarrow \text{FILTER}(\mathbf{X}, \mathbf{Y}, \mathbf{f}^{(t-1)}(\cdot), C_t)$ ▷ *draw a random subset*
  $(\mathbf{v}^{(t)}, \varphi^{(t)}(\cdot)) \leftarrow \text{BASE}(\mathbf{X}', \mathbf{Y}', \mathbf{W}')$ ▷ *train the base learner*
  $(\mathbf{X}', \mathbf{Y}', \mathbf{W}') \leftarrow \text{FILTER}(\mathbf{X}, \mathbf{Y}, \mathbf{f}^{(t-1)}(\cdot), C_t)$ ▷ *draw a second random subset*
  $n' \leftarrow |\mathbf{X}'|$
  $\gamma \leftarrow \sum_{i}^{n'} \sum_{\ell}^{K} w'_{i,\ell} v_{\ell}^{(t)} \varphi^{(t)}(\mathbf{x}'_i)$ ▷ *estimate the edge on a filtered data*
  $\alpha^{(t)} \leftarrow \frac{1}{2}\log\frac{1+\gamma}{1-\gamma}$
  $\mathbf{h}^{(t)}(\cdot) \leftarrow \alpha^{(t)}\mathbf{v}^{(t)}\varphi^{(t)}(\cdot)$
return $\mathbf{f}^{(T)}(\cdot) = \sum_{t=1}^{T} \mathbf{h}^{(t)}(\cdot)$

FILTER$(\mathbf{X}, \mathbf{Y}, \mathbf{f}(\cdot), C)$

$\mathbf{X}' \leftarrow \mathbf{Y}' \leftarrow \mathbf{W} \leftarrow ()$ ▷ *empty matrices*
for $t \leftarrow 1$ **to** C
  $i \leftarrow \text{RANDOMINT}(1, n)$ ▷ *draw a random index between* 1 *and* n
  for $\ell \leftarrow 1$ **to** K
    $w_\ell \leftarrow \frac{1}{1+\exp\left(f(\mathbf{x}_i)y_{i,\ell}\right)}$
  $r \leftarrow \text{RANDOMREAL}(0, 1)$ ▷ *draw a random real number between* 0 *and* 1
    if $r < \frac{1}{K}\sum_{\ell=1}^{K} w_\ell$ **then**
      $\mathbf{X}' \leftarrow \text{APPEND}(\mathbf{X}', \mathbf{x}_i)$
      $\mathbf{Y}' \leftarrow \text{APPEND}(\mathbf{Y}', \mathbf{y}_i)$
      $\mathbf{W} \leftarrow \text{APPEND}(\mathbf{W}, \mathbf{w})$
for $i \leftarrow 1$ **to** C **for** $\ell \leftarrow 1$ **to** K
  $w_{i,\ell} \leftarrow \frac{w_{i,\ell}}{\sum_{i'=1}^{C}\sum_{\ell'=1}^{K} w_{i',\ell'}}$ ▷ *normalize the weights*
return $(\mathbf{X}', \mathbf{Y}', \mathbf{W})$

Figure 5: The pseudocode of the FILTERBOOST algorithm.We used the same notations as in Figure 4.

or *negative* events, whereas there are only a relatively small number of *signal* or *positive* events. A cascade classifier consists of *stages*. In each stage a binary classifier attempts to eliminate background instances by classifying them negatively. Positive classification in inner stages sends the instance to the next stage, so detection can only be made in the last stage. By using simple and fast classifiers in the first stages, "easy" background instances can be rejected fast, shortening the expected testing time. The cascade structure also allows the use of different training sets at

different stages, having more difficult background samples at later stages.

A.5.1 The Viola-Jones cascade

One of the pioneering work in cascade learning is the Viola-Jones cascade learner (VJCASCADE) [5]. This cascade learning algorithm is based on the ADABOOST algorithm and, in its original form, it can be used only for binary classification problems. Our implementation fully corresponds to the original version.

A.5.2 The soft cascade

Another popular algorithm is the soft cascade learner (SOFTCASCADE) [6]. It consists of a single monolithic classifier where the weak classifiers are ordered and augmented with rejection thresholds. The rejection thresholds are used to discard negative examples after each weak classifier evaluation.

B Multi-class base learning

The goal of multi-class base learning is to minimize $Z(\mathbf{h}, \mathbf{W})$ (11). In general, any vector-valued multi-class learning algorithm can be used here. Although this goal is clearly defined in [1], efficient base learning algorithms have never been described in detail. The algorithms are elementary: they are mostly based on exhaustive or greedy search, but some of the implementation details are somewhat tricky.

In this section we will describe base learning algorithms that look for $\mathbf{h}(\mathbf{x})$ in the form of

$$\mathbf{h}(\mathbf{x}) = \alpha \mathbf{v} \varphi(\mathbf{x}), \tag{24}$$

where $\alpha \in \mathbb{R}^+$ is a positive real valued *base coefficient*, $\mathbf{v}$ is a binary $\mathbf{v} \in \{\pm 1\}^K$ *vote* vector, and $\varphi(\mathbf{x}) : \mathcal{X} \to \{\pm 1\}$ is a binary classifier. This setup is known as *discrete* ADABOOST.MH. The setup can be extended to real-valued classifiers $\varphi(\mathbf{x}) : \mathcal{X} \to \mathbb{R}$, also known as *confidence-rated* classifiers or, simply, discriminant functions, and it is also easy to make the vote vector $\mathbf{v}$ real-valued (in which case, without the loss of generality, α would be set to 1). Both variants are known under the name of *real* ADABOOST.MH. Although there might be slight differences in the practical performance of real and discrete ADABOOST.MH, here we decided to stick to the discrete case for the sake of simplicity.

To start, in Section B.1 we show how to set α and $\mathbf{v}$ in general if the scalar base classifier φ is given. Some of the elements defined here will be reused throughout the section. In Section B.2 and Section B.3 we describe two simple base classifiers used for numerical and nominal features, respectively. In practice, boosting these simple learners has often been found to be sub-optimal, mainly due to the lack of capacity of the resulted strong classifier (large approximation error). The most common solution for overcoming this problem is to use trees as base learners that call the simple base learner in a recursive fashion to partition the input space. We describe a basic tree learner adapted to the multi-class setup of ADABOOST.MH in Section B.4. In Section B.5 we introduce another meta-base learner that combines simple base learners in a multiplicative fashion.

B.1 Casting the votes

The intuitive semantics of (24) is the following. The binary classifier $\varphi(\mathbf{x})$ cuts the input space into a positive and a negative region. In binary classification this is the end of the story: we need $\varphi(\mathbf{x})$ to be well-correlated with the binary class labels y. In multi-class classification it is possible that $\varphi(\mathbf{x})$ correlates with some of the class labels y_ℓ and anti-correlates with others. This free choice is expressed by the binary votes $v_\ell \in \{\pm 1\}$. We say that $\varphi(\mathbf{x})$ votes *for* class ℓ if $v_\ell = +1$ and it votes *against* class ℓ if $v_\ell = -1$. As in binary classification, α expresses the overall quality

of the classifier $\mathbf{v}\varphi(\mathbf{x})$: α is monotonically decreasing with respect to the weighted error of $\mathbf{v}\varphi(\mathbf{x})$ (or, equivalently, monotonically increasing with respect to the edge of $\mathbf{v}\varphi(\mathbf{x})$).

The advantage of the setup is that, given the binary classifier $\varphi(\mathbf{x})$, the optimal vote vector $\mathbf{v}$ and the coefficient α can be set in an efficient way. To see this, first let us define the *weighted per-class error rate*

$$\mu_{\ell-} = \sum_{i=1}^{n} w_{i,\ell} \mathbb{I}\left\{\varphi(\mathbf{x}_i) \neq y_{i,\ell}\right\}, \tag{25}$$

and the *weighted per-class correct classification rate*

$$\mu_{\ell+} = \sum_{i=1}^{n} w_{i,\ell} \mathbb{I}\left\{\varphi(\mathbf{x}_i) = y_{i,\ell}\right\} \tag{26}$$

for each class $\ell = 1, \ldots, K$. With this notation, $Z(\mathbf{h}, \mathbf{W})$ simplifies to

$$\begin{aligned}
Z(\mathbf{h}, \mathbf{W}) &= \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell} \exp\left(-h_{\ell_i} y_{i,\ell}\right) = \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell} \exp\left(-\alpha v_\ell \varphi(\mathbf{x}_i) y_{i,\ell}\right) && (27)\\
&= \sum_{i=1}^{n}\sum_{\ell=1}^{K} \left(w_{i,\ell}\mathbb{I}\left\{v_\ell \varphi(\mathbf{x}_i) y_{i,\ell} = 1\right\} e^{-\alpha} + w_{i,\ell}\mathbb{I}\left\{v_\ell \varphi(\mathbf{x}_i) y_{i,\ell} = -1\right\} e^{\alpha}\right)\\
&= \sum_{\ell=1}^{K} \left(\mu_{\ell+}\mathbb{I}\left\{v_\ell = +1\right\} + \mu_{\ell-}\mathbb{I}\left\{v_\ell = -1\right\}\right) e^{-\alpha}\\
&\quad + \sum_{\ell=1}^{K} \left(\mu_{\ell-}\mathbb{I}\left\{v_\ell = +1\right\} + \mu_{\ell+}\mathbb{I}\left\{v_\ell = -1\right\}\right) e^{\alpha} && (28)\\
&= \sum_{\ell=1}^{K} \left(\mathbb{I}\left\{v_\ell = +1\right\}\left(e^{-\alpha}\mu_{\ell+} + e^{\alpha}\mu_{\ell-}\right) + \mathbb{I}\left\{v_\ell = -1\right\}\left(e^{-\alpha}\mu_{\ell-} + e^{\alpha}\mu_{\ell+}\right)\right)\\
&= \sum_{\ell=1}^{K} \left(\frac{1+v_\ell}{2}\left(e^{-\alpha}\mu_{\ell+} + e^{\alpha}\mu_{\ell-}\right) + \frac{1-v_\ell}{2}\left(e^{-\alpha}\mu_{\ell-} + e^{\alpha}\mu_{\ell+}\right)\right)\\
&= \frac{1}{2}\sum_{\ell=1}^{K} \left(\left(e^{\alpha} + e^{-\alpha}\right)\left(\mu_{\ell+} + \mu_{\ell-}\right) - v_\ell\left(e^{\alpha} - e^{-\alpha}\right)\left(\mu_{\ell+} - \mu_{\ell-}\right)\right)\\
&= \frac{e^{\alpha} + e^{-\alpha}}{2} - \frac{e^{\alpha} - e^{-\alpha}}{2}\sum_{\ell=1}^{K} v_\ell\left(\mu_{\ell+} - \mu_{\ell-}\right). && (29)
\end{aligned}$$

(27) comes from the definition (24) of $\mathbf{h}$ and (28) follows from the definitions (25) and (26) of $\mu_{\ell-}$ and $\mu_{\ell+}$. In the final step (29) we used the fact that

$$\sum_{\ell=1}^{K}\left(\mu_{\ell+} + \mu_{\ell-}\right) = \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell} = 1.$$

The quantity

$$\gamma_\ell = v_\ell\left(\mu_{\ell+} - \mu_{\ell-}\right) = \sum_{i=1}^{n} w_{i,\ell} v_\ell \varphi(\mathbf{x}_i) y_{i,\ell} \tag{30}$$

is called the *classwise edge* of $\mathbf{h}(\mathbf{x})$ with $\boldsymbol{\gamma}$ denoting the vector $(\gamma_1, \ldots, \gamma_K)$. The full multi-class edge of the classifier is then

$$\gamma = \gamma(\mathbf{v}, \varphi, \mathbf{W}) = \|\boldsymbol{\gamma}\|_1 = \sum_{\ell=1}^{K}\gamma_\ell = \sum_{\ell=1}^{K} v_\ell\left(\mu_{\ell+} - \mu_{\ell-}\right) = \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell} v_\ell \varphi(\mathbf{x}_i) y_{i,\ell}. \tag{31}$$

With this notation, the optimal binary coefficient α (19) is recovered: it is easy to see that (29) is minimized when

$$\alpha = \frac{1}{2}\log\frac{1+\gamma}{1-\gamma}. \tag{32}$$

With this optimal coefficient, (29) becomes

$$Z(\mathbf{h}, \mathbf{W}) = \sqrt{1-\gamma^2},$$

so $Z(\mathbf{h}, \mathbf{W})$ is minimized when γ is maximized. From (31) it then follows that $Z(\mathbf{h}, \mathbf{W})$ is minimized if v_ℓ agrees with the sign of $(\mu_{\ell+} - \mu_{\ell-})$, that is,

$$v_\ell = \begin{cases} 1 & \text{if } \mu_{\ell+} > \mu_{\ell-} \\ -1 & \text{otherwise} \end{cases} \tag{33}$$

for all classes $\ell = 1, \ldots, K$.

The setup of vector-valued based classification has another important consequence: the preservation of the weak-learning condition. Indeed, if $\varphi(\mathbf{x})$ is slightly better than a coin toss (in maximizing the edge), γ will be positive. Another way to look at it is to say that if a $(\varphi, \mathbf{v})$ combo has a negative edge $\gamma < 0$, the edge of its *complement* (either $(-\varphi, \mathbf{v})$ or $(\varphi, -\mathbf{v})$) will be $-\gamma > 0$. To understand the significance of this, consider a classical single-label base classifier $h : \mathcal{X} \to \mathcal{L} = \{1, \ldots, K\}$, required by ADABOOST.M1. Now if $h(\mathbf{x})$ is slightly better than a coin toss, all one can hope for is an error rate slightly lower than $\frac{K-1}{K}$. To achieve the error of $\frac{1}{2}$, required for continuing the boosting iterations, one has to come up with a base learner which is significantly better than a coin toss.

Finally, note that computing $\mathbf{v}$ and α is a $\Theta(nK)$ operation which may be prohibitive if we have to do it for every base classifier φ in an exhaustive search. Fortunately, it turns out that the two simple base classifiers STUMPBASE (Section B.2) and INDICATORBASE (Section B.3) can make cheap updates to the edge (31) while doing the exhaustive search, keeping the time complexity of *full* base learning at $\Theta(nK)$.

B.2 Decision stumps for numerical features

The simplest scalar base learner used in practice on numerical features is the *decision stump*, a one-decision two-leaf decision tree of the form

$$\varphi_{j,b}(\mathbf{x}) = \begin{cases} 1 & \text{if } x^{(j)} \geq b, \\ -1 & \text{otherwise,} \end{cases}$$

where j is the index of the selected feature and b is the decision threshold. If the feature values $(x_1^{(j)}, \ldots, x_n^{(j)})$ are pre-ordered before the first boosting iteration, a decision stump maximizing the edge (31) (or minimizing the energy (27)[6]) can be found very efficiently in $\Theta(ndK)$ time.

The pseudocode of the algorithm is given in Figure 6. STUMPBASE first calculates the edge vector $\boldsymbol{\gamma}^{(0)}$ of the constant classifier $\mathbf{h}^{(0)}(\mathbf{x}) \equiv \mathbf{1}$ which will serve as the initial edge vector for each featurewise edge-maximizer. Then it loops over the features, calls BESTSTUMP to return the best featurewise stump, and then selects the best of the best by minimizing the energy (27). BESTSTUMP loops over all (sorted) feature values $s_1, \ldots, s_{n-1}$. It considers all thresholds b halfway between two non-identical feature values $s_i \neq s_{i+1}$. The main trick (and, at the same time, the bottleneck of the algorithm) is the update of the classwise edges in lines 4-5: when the threshold moves from $b = \frac{s_{i-1}+s_i}{2}$ to $b = \frac{s_i+s_{i+1}}{2}$, the classwise edge γ_ℓ of $\mathbf{1}\varphi(\mathbf{x})$ (that is, $\mathbf{v}\varphi(\mathbf{x})$ with $\mathbf{v} = \mathbf{1}$) can only change by $\pm w_{i,\ell}$, depending on the sign $y_{i,\ell}$ (Figure 8). The total edge of $\mathbf{v}\varphi(\mathbf{x})$ with optimal votes (33) is then the sum of the absolute values of the classwise edges of $\mathbf{1}\varphi(\mathbf{x})$ (line 7).

[6]Note the distinction: for full binary $\mathbf{v}$ the two are equivalent, but for ternary or real valued $\mathbf{v}$ and/or real valued $\phi(\mathbf{x})$ they are not. In Figure 6 we are maximizing the edge within each feature (line 7 in BESTSTUMP) but across features we are minimizing the energy (line 7 in STUMPBASE). Updating the energy inside the inner loop (line 4) could not be done in $\Theta(K)$ time.

StumpBase$(\mathbf{X}, \mathbf{Y}, \mathbf{W})$

1 **for** $\ell \leftarrow 1$ **to** K ▷ *for all classes*

2 $\quad \gamma_\ell^{(0)} \leftarrow \sum_{i=1}^{n} w_{i,\ell} y_{i,\ell}$ ▷ *classwise edges (30) of constant classifier* $\mathbf{h}^{(0)}(\mathbf{x}) \equiv \mathbf{1}$

3 **for** $j \leftarrow 1$ **to** d ▷ *all (numerical) features*

4 $\quad \mathbf{s} \leftarrow \text{Sort}(x_1^{(j)}, \ldots, x_n^{(j)})$ ▷ *sort the jth column of* $\mathbf{X}$

5 $\quad (\mathbf{v}_j, b_j, \gamma_j) \leftarrow \text{BestStump}(\mathbf{s}, \mathbf{Y}, \mathbf{W}, \boldsymbol{\gamma}^{(0)})$ ▷ *best stump per feature*

6 $\quad \alpha_j \leftarrow \frac{1}{2} \log \frac{1+\gamma_j}{1-\gamma_j}$ ▷ *base coefficient (32)*

7 $j^* \leftarrow \arg\min_j Z(\alpha_j \mathbf{v}_j \varphi_{j,b_j}, \mathbf{W})$ ▷ *best stump across features*

8 **return** $(\alpha_{j^*}, \mathbf{v}_{j^*}, \varphi_{j^*,b_{j^*}}(\cdot))$

BestStump$(\mathbf{s}, \mathbf{Y}, \mathbf{W}, \boldsymbol{\gamma}^{(0)})$

1 $\boldsymbol{\gamma}^* \leftarrow \boldsymbol{\gamma}^{(0)}$ ▷ *best edge vector*

2 $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma}^{(0)}$ ▷ *initial edge vector*

3 **for** $i \leftarrow 1$ **to** $n-1$ ▷ *for all points in order* $s_1 \leq \ldots \leq s_{n-1}$

4 $\quad$ **for** $\ell \leftarrow 1$ **to** K ▷ *for all classes*

5 $\quad\quad \gamma_\ell \leftarrow \gamma_\ell - 2 w_{i,\ell} y_{i,\ell}$ ▷ *update classwise edges of stump with* $\mathbf{v} = \mathbf{1}$

6 $\quad$ **if** $s_i \neq s_{i+1}$ **then** ▷ *no threshold if identical coordinates* $s_i = s_{i+1}$

7 $\quad\quad$ **if** $\sum_{\ell=1}^{K} |\gamma_\ell| > \sum_{\ell=1}^{K} |\gamma_\ell^*|$ **then** ▷ *found better stump*

8 $\quad\quad\quad \boldsymbol{\gamma}^* \leftarrow \boldsymbol{\gamma}$ ▷ *update best edge vector*

9 $\quad\quad\quad b^* \leftarrow \frac{s_i + s_{i+1}}{2}$ ▷ *update best threshold*

10 **for** $\ell \leftarrow 1$ **to** K ▷ *for all classes*

11 $\quad v_\ell^* \leftarrow \text{sign}(\gamma_\ell^*)$ ▷ *set vote vector according to (33)*

12 **if** $\boldsymbol{\gamma}^* = \boldsymbol{\gamma}^{(0)}$ ▷ *did not beat the constant classifier*

13 $\quad$ **return** $(\mathbf{v}^*, -\infty, \|\boldsymbol{\gamma}^*\|_1)$ ▷ *constant classifier with optimal votes*

14 **else**

15 $\quad$ **return** $(\mathbf{v}^*, b^*, \|\boldsymbol{\gamma}^*\|_1)$ ▷ *best stump*

Figure 6: Exhaustive search for the best decision stump. BestStump receives a sorted column (feature) $\mathbf{s}$ of the observation matrix $\mathbf{X}$. The sorting in line 4 can be done once for all features outside of the boosting loop. BestStump examines all thresholds b halfway between two non-identical coordinates $s_i \neq s_{i+1}$ and returns the threshold b^* and vote vector $\mathbf{v}^*$ that maximizes the edge $\gamma(\mathbf{v}, \varphi_{j,b}, \mathbf{W})$. StumpBase then sets the coefficient α_j according to (32) and chooses the stump across features that minimizes the energy (11).

Starting with version v1.2 we are using the slightly more complicated but faster BESTSTUMP-FAST routine (Figure 7). The main idea behind the algorithm is that if we know the most frequent feature value x^*, we can omit looping over instances with value $s_i = x^*$. We start by an increasing loop as BESTSTUMP but stop if the feature value $s_i \geq x^*$ (lines 4 to 11). Then we run a symmetric search from s_n down to x^* (lines 14 to 21). The resulting algorithm is sub-linear in n.

B.3 Subset indicators for nominal features

Nominal features $x_i^{(j)} \in \mathcal{I}^{(j)} = \{1, \ldots, M^{(j)}\}$ are as abundant in practice as numerical features, and they need to be treated differently at the base learning. Their main feature is that they cannot be ordered naturally; they share this feature with the class variable ℓ. The textbook example is color, e.g., $\mathcal{I}^{(j)} = \{$, , , , $\}$, but applications range from text processing (word- or tag-valued features) to collaborative filtering (movie or user indices).

The classical way to handle nominal features with numerical learners is to encode a feature with $M^{(j)}$ possible values into $M^{(j)}$ binary (for example, $\{0,1\}$-valued) features. Learning a decision stump on such encoding means that each base classifiers selects one value $\iota \in \mathcal{I}^{(j)}$, and separates observations with $x^{(j)} = \iota$ from $x^{(j)} \neq \iota$. We call this the SELECTORBASE learner, and define it formally as

$$\varphi_{j,\iota}(\mathbf{x}) = \begin{cases} 1 & \text{if } x^{(j)} = \iota, \\ -1 & \text{otherwise.} \end{cases} \tag{34}$$

Using our example, the base learner that selects the blue value can be represented by $\varphi_{j,}$ = $\{-, +, -, -, -\}$.

Optimizing this SELECTORBASE takes $\Theta(ndK + K\Sigma)$ time, where $\Sigma = \sum_{j=1}^{d} M^{(j)}$ is the number of all the different values of all features $j = 1, \ldots, d$. Hence, to obtain a strong learner that potentially uses (tests) $\Omega(M^{(j)})$ values for each feature, we will need $\Omega\big((ndK + K\Sigma)\Sigma\big)$ operations, which can be prohibitive if Σ is large. On the other hand, it turns out that we can optimize a base learner that makes a decision on *all* values of $\mathcal{I}^{(j)}$ in each iteration, using essentially the same number $\Theta(ndK + K\Sigma)$ of operations per iteration as the selector base learner. Formally, we consider base learners of the form

$$\varphi_{j,\mathbf{u}}(\mathbf{x}) = \mathbf{u}_{x^{(j)}}, \tag{35}$$

where $\mathbf{u}$ is a $\{\pm 1\}$-valued vector over the index set $\mathcal{I}^{(j)}$. For example, the base classifier

$$\varphi_{j,\{+1,-1,-1,+1,+1\}} = \{+, -, -, +, +\}$$

outputs $+1$ for observations $\mathbf{x}$ whose jth feature is , , or , and -1 for feature values or .

Learning a subset indicator is essentially a rank-1 matrix factorization problem. For showing this, we first construct the $M^{(j)} \times K$-dimensional *edge matrix* $\boldsymbol{\Gamma}^{(j)} = \big[\gamma_{\iota,\ell}^{(j)}\big]$ with elements

$$\gamma_{\iota,\ell}^{(j)} = \sum_{i=1}^{n} \mathbb{I}\left\{x_i^{(j)} = \iota\right\} w_{i,\ell} y_{i,\ell} \tag{36}$$

for each (nominal) feature $j = 1, \ldots, d$, label index $\ell = 1, \ldots, K$ and feature value $\iota = \{1, \ldots, M^{(j)}\}$.

BESTSTUMPFAST$(\mathbf{s}, \mathbf{Y}, \mathbf{W}, \boldsymbol{\gamma}^{(0)}, x^*)$

1 $\quad\boldsymbol{\gamma}^* \leftarrow \boldsymbol{\gamma}^{(0)}$ ▷ *best edge vector*
2 $\quad\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma}^{(0)}$ ▷ *initial edge vector*
3 $\quad i \leftarrow 1$
4 $\quad$**while** $i \leq n-1$ **and** $s_i < x^*$ ▷ *for all points in order* $s_1 \leq \ldots \leq s_\cdot < x^*$
5 $\quad\quad$**for** $\ell \leftarrow 1$ **to** K ▷ *for all classes*
6 $\quad\quad\quad\gamma_\ell \leftarrow \gamma_\ell - 2w_{i,\ell}y_{i,\ell}$ ▷ *update classwise edges of stump with* $\mathbf{v} = \mathbf{1}$
7 $\quad\quad$**if** $s_i \neq s_{i+1}$ **then** ▷ *no threshold if identical coordinates* $s_i = s_{i+1}$
8 $\quad\quad\quad$**if** $\sum_{\ell=1}^K |\gamma_\ell| > \sum_{\ell=1}^K |\gamma_\ell^*|$ **then** ▷ *found better stump*
9 $\quad\quad\quad\quad\boldsymbol{\gamma}^* \leftarrow \boldsymbol{\gamma}$ ▷ *update best edge vector*
10 $\quad\quad\quad\quad b^* \leftarrow \frac{s_i + s_{i+1}}{2}$ ▷ *update best threshold*
11 $\quad\quad i \leftarrow i + 1$
12 $\quad\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma}^{(0)}$ ▷ *initial edge vector*
13 $\quad i \leftarrow n$
14 $\quad$**while** $i \geq 2$ **and** $s_i > x^*$ ▷ *for all points in order* $s_n \geq \ldots \geq s_\cdot > x^*$
15 $\quad\quad$**for** $\ell \leftarrow 1$ **to** K ▷ *for all classes*
16 $\quad\quad\quad\gamma_\ell \leftarrow \gamma_\ell - 2w_{i,\ell}y_{i,\ell}$ ▷ *update classwise edges of stump with* $\mathbf{v} = \mathbf{1}$
17 $\quad\quad$**if** $s_i \neq s_{i-1}$ **then** ▷ *no threshold if identical coordinates* $s_i = s_{i-1}$
18 $\quad\quad\quad$**if** $\sum_{\ell=1}^K |\gamma_\ell| > \sum_{\ell=1}^K |\gamma_\ell^*|$ **then** ▷ *found better stump*
19 $\quad\quad\quad\quad\boldsymbol{\gamma}^* \leftarrow \boldsymbol{\gamma}$ ▷ *update best edge vector*
20 $\quad\quad\quad\quad b^* \leftarrow \frac{s_i + s_{i-1}}{2}$ ▷ *update best threshold*
21 $\quad\quad i \leftarrow i - 1$
22 $\quad$**for** $\ell \leftarrow 1$ **to** K ▷ *for all classes*
23 $\quad\quad v_\ell^* \leftarrow \text{sign}(\gamma_\ell)$ ▷ *set vote vector according to (33)*
24 $\quad$**if** $\boldsymbol{\gamma}^* = \boldsymbol{\gamma}^{(0)}$ ▷ *did not beat the constant classifier*
25 $\quad\quad$**return** $(\mathbf{v}^*, -\infty, \|\boldsymbol{\gamma}^*\|_1)$ ▷ *constant classifier with optimal votes*
26 $\quad$**else**
27 $\quad\quad$**return** $(\mathbf{v}^*, b^*, \|\boldsymbol{\gamma}^*\|_1)$ ▷ *best stump*

Figure 7: Exhaustive fast search for the best decision stump. BESTSTUMPFAST receives a sorted column (feature) $\mathbf{s}$ of the observation matrix $\mathbf{X}$ and a special feature value x^*. It proceeds as BESTSTUMP but stops if the feature value $s_i \geq x^*$ (lines 4 to 11). Then it runs a symmetric search from s_n down to x^* (lines 14 to 21). Compared to BESTSTUMP, it saves time by not looping over points with value $s_i = x^*$.

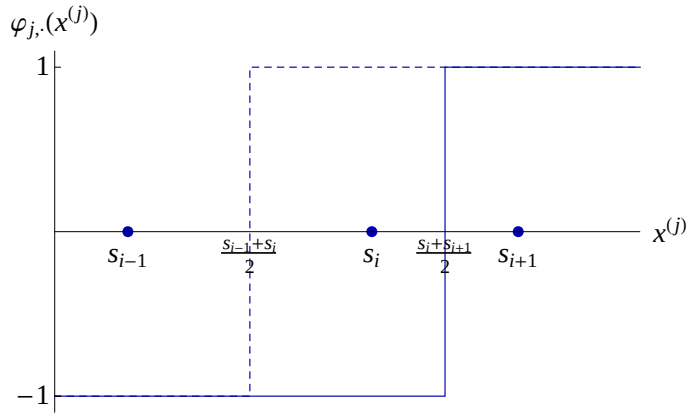

Figure 8: Updating the edge γ_ℓ in line 5 of BESTSTUMP. If $y_{i,\ell} = 1$, then γ_ℓ decreases by $2w_{i,\ell}$, and if $y_i = -1$, then γ_ℓ increases by $2w_{i,\ell}$.

The edge of $\varphi_{j,\mathbf{u}}(\mathbf{x})$ with vote vector $\mathbf{v}$ is

$$\begin{aligned}
\gamma(\mathbf{v}, \varphi_{j,\mathbf{u}}, \mathbf{W}) &= \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell} v_\ell \varphi_{j,\mathbf{u}}(\mathbf{x}_i) y_{i,\ell} && (37)\\
&= \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell} v_\ell u_{x_i^{(j)}} y_{i,\ell} && (38)\\
&= \sum_{\ell=1}^{K} v_\ell \sum_{i=1}^{n} w_{i,\ell} u_{x_i^{(j)}} y_{i,\ell} && (39)\\
&= \sum_{\ell=1}^{K} v_\ell \sum_{i=1}^{n}\sum_{\iota=1}^{M} w_{i,\ell} \mathbb{I}\left\{x_i^{(j)} = \iota\right\} u_\iota y_{i,\ell} && (40)\\
&= \sum_{\ell=1}^{K} v_\ell \sum_{\iota=1}^{M} u_\iota \sum_{i=1}^{n} \mathbb{I}\left\{x_i^{(j)} = \iota\right\} w_{i,\ell} y_{i,\ell} && (41)\\
&= \sum_{\ell=1}^{K}\sum_{\iota=1}^{M} v_\ell u_\iota \gamma_{\iota,\ell}^{(j)}. && (42)
\end{aligned}$$

(37) comes from the definition (31) of the multi-class edge $\gamma(\mathbf{v}, \varphi_{j,\mathbf{u}}, \mathbf{W})$. In (38) we used the transcription (35) of $\varphi_{j,\mathbf{u}}(\mathbf{x})$. In (39) and (41) we simply re-ordered the summations. In (40) we used the fact that

$$u_{x_i^{(j)}} = \sum_{\iota=1}^{M} \mathbb{I}\left\{x_i^{(j)} = \iota\right\} u_\iota.$$

Finally, (42) follows from the definition (36) of the edge matrix $\mathbf{\Gamma}^{(j)}$. Introducing the elementwise matrix multiplication operator $\otimes$, the edge of $\varphi_{j,\mathbf{u}}(\mathbf{x})$ with vote vector $\mathbf{v}$ is

$$\gamma(\mathbf{v}, \varphi_{j,\mathbf{u}}, \mathbf{W}) = \left\|\mathbf{\Gamma}^{(j)} \otimes \mathbf{v}\mathbf{u}^\intercal\right\|_1,$$

where $\mathbf{v}\mathbf{u}^\intercal$ is the outer product of $\mathbf{v}$ and $\mathbf{u}$ and $\|\cdot\|_1$ is the matrix 1-norm (sum of elements). Since

$$v_\ell u_\iota \gamma_{\iota,\ell}^{(j)} = \left|\gamma_{\iota,\ell}^{(j)}\right| \left(1 - 2\mathbb{I}\left\{v_\ell u_\iota \neq \operatorname{sign}\left(\gamma_{\iota,\ell}^{(j)}\right)\right\}\right),$$

maximizing $\gamma(\mathbf{v}, \varphi_{j,\mathbf{u}}, \mathbf{W})$ is equivalent to finding $\mathbf{u}$ and $\mathbf{v}$ that minimize

$$\frac{\sum_{\ell=1}^{K}\sum_{\iota=1}^{M}\left|\gamma_{\iota,\ell}^{(j)}\right| - \gamma(\mathbf{v}, \varphi_{j,\mathbf{u}}, \mathbf{W})}{2} = \sum_{\ell=1}^{K}\sum_{\iota=1}^{M}\left|\gamma_{\iota,\ell}^{(j)}\right| \mathbb{I}\left\{v_\ell u_\iota \neq \operatorname{sign}\left(\gamma_{\iota,\ell}^{(j)}\right)\right\},$$

which is a weighted (binary) one-error. Therefore, in a nutshell, maximizing the edge in $\mathbf{u}$ and $\mathbf{v}$ is equivalent to a rank-1 matrix factorization with an elementwise weighted one-loss.

Whatever beautiful name we give it, maximizing the edge $\gamma(\mathbf{v}, \varphi_{j,\mathbf{u}}, \mathbf{W})$ is a difficult combinatorial optimization problem. Fortunately, ADABOOST.MH does not require the maximization of the edge: we only need and edge $\gamma^{(t)} > 0$ in each boosting iteration t to be able to continue. The BESTINDICATOR[7] algorithm (Figure 9) carries out a local search for a good $(\mathbf{u}, \mathbf{v})$ pair. It is easy to see that the optimal $\mathbf{v}$ for a fixed $\mathbf{u}$ is

$$v_\ell = \text{sign}\left(\sum_{\iota=1}^{M^{(j)}} \gamma^{(j)}_{\iota,\ell} u_\iota\right), \tag{43}$$

and, symmetrically, the optimal $\mathbf{u}$ for a fixed $\mathbf{v}$ is

$$u_\iota = \text{sign}\left(\sum_{\ell=1}^{K} \gamma^{(j)}_{\iota,\ell} v_\ell\right). \tag{44}$$

After computing $\mathbf{\Gamma}^{(j)}$ (lines 2-4) and initializing $\mathbf{u}$ to a random $\{\pm 1\}$-valued vector (line 6), BESTINDICATOR iterates between (43) and (44) until no change in the edge. Convergence is guaranteed since $\gamma(\mathbf{v}, \varphi_{j,\mathbf{u}}, \mathbf{W})$ is bounded from above by 1, it must increase in each iteration, and the increase is bounded away from zero because the weights $w_{i,\ell}$ are bounded away from zero. In practice BESTINDICATOR always stops after a few iterations. Once it returns the pair $(\mathbf{u}_j, \mathbf{v}_j)$ for each feature j, INDICATORBASE, as STUMPBASE, takes care of setting the coefficient α_j and finding the best feature j^*.

Another advantage of INDICATORBASE over SELECTORBASE (besides being faster) is that the vote vector $\mathbf{v}$ is shared among the selectors while in the standard learner we learn a vote vector for each selector. This means that the capacity of the base classifier is considerably reduced (compared to the sum of Σ selectors), suggesting that the algorithm is less susceptible to overfitting. Sharing the vote vector may also help to "pick up" dependencies between feature values in the case where there is a structure in the feature space (e.g., using discrete ADABOOST.MH, each indicator $\varphi_{j,\mathbf{u}}$ clusters the feature values $\mathcal{I}^{(j)}$ into two groups).

In a broader sense, the proposed learner is related to Cohen's [16] model which allows set-valued features as well as the common real-valued and nominal features. The main difference is that in [16] it is the *features of the observations* that can take a subset of a large set of possible values but the decision functions are still *selectors* of the form (34), whereas what we propose here are subset-indicator *decision functions* (35). In another related algorithm (SLIPPER), Cohen and Singer [17] build rules that are *conjunctions of selectors* (34) using a growing/pruning procedure. The selectors in the same base rule can use different *features*, whereas INDICATORBASE acts on different *values* of *one* feature. Although superficially similar to our method, both approaches use different base classifiers and different algorithms to learn the classifiers. On the other hand, they can be incorporated into our approach with just a few modifications to INDICATORBASE.

B.4 Hamming trees

Classification trees [18] have been widely used for multivariate classification since the 80s. They are especially efficient when used as base learners in ADABOOST [19]. Their main disadvantage is their variance with respect to the training data, but when averaged over T different runs, this problem disappears. The reason of using trees is to increase the expressiveness (complexity) of base classifiers. For example, learning a linear combination of decision stumps on pixels of an image ignores completely the correlation between pixels. A small tree of even two nodes, on the other hand, can easily pick up this correlation.

[7] The name comes from the fact that $\mathbf{u}$ is a binary vector that formally indicates a subset of $\mathcal{I}$.

INDICATORBASE$(\mathbf{X}, \mathbf{Y}, \mathbf{W})$

1 **for** $j \leftarrow 1$ **to** d ▷ *all (nominal) features*

2 $(\mathbf{v}_j, \mathbf{u}_j, \gamma_j) \leftarrow$ BESTINDICATOR$(\mathbf{x}^{(j)}, \mathbf{Y}, \mathbf{W}, \mathcal{I}^{(j)})$ ▷ $\mathbf{x}^{(j)} \triangleq (x_1^{(j)}, \ldots, x_n^{(j)})$

3 $\alpha_j \leftarrow \frac{1}{2} \log \frac{1+\gamma_j}{1-\gamma_j}$ ▷ *base coefficient (32)*

4 $j^* \leftarrow \arg\min_j Z(\alpha_j \mathbf{v}_j \varphi_{j,\mathbf{u}_j}, \mathbf{W})$

5 **return** $(\alpha_{j^*}, \mathbf{v}_{j^*}, \varphi_{j^*,\mathbf{u}_{j^*}}(\cdot))$

BESTINDICATOR$(\mathbf{x}, \mathbf{Y}, \mathbf{W}, \mathcal{I})$

1 $M \leftarrow |\mathcal{I}|$ ▷ *number of feature values*

2 **for** $\ell \leftarrow 1$ **to** K ▷ *compute* $\boldsymbol{\Gamma}$ *(36)*

3 **for** $\iota \leftarrow 1$ **to** M $\gamma_{\iota,\ell} \leftarrow 0$

4 **for** $i \leftarrow 1$ **to** n $\gamma_{x_i,\ell} \leftarrow \gamma_{x_i,\ell} + w_{i,\ell} y_{i,\ell}$

5 $\gamma^* \leftarrow 0$

6 **for** $\iota \leftarrow 1$ **to** M $u_\iota \leftarrow$ RANDOM(± 1)

7 **while** TRUE

8 **for** $\ell \leftarrow 1$ **to** K ▷ *compute optimal* $\mathbf{v}$ *given* $\mathbf{u}$ *(43)*

9 $\gamma_\ell \leftarrow \sum_{\iota=1}^{M} \gamma_{\iota,\ell} u_\iota$

10 $v_\ell \leftarrow \operatorname{sign}(\gamma_\ell)$

11 **if** $\gamma^* \geq \sum_{\ell=1}^{K} |\gamma_\ell|$ **then return** $(\mathbf{u}, \mathbf{v}, \gamma^*)$

12 $\gamma^* \leftarrow \sum_{\ell=1}^{K} |\gamma_\ell|$

13 **for** $\iota \leftarrow 1$ **to** M ▷ *compute optimal* $\mathbf{u}$ *given* $\mathbf{v}$ *(44)*

14 $\gamma_\iota \leftarrow \sum_{\ell=1}^{K} \gamma_{\iota,\ell} v_\ell$

15 $u_\iota \leftarrow \operatorname{sign}(\gamma_\iota)$

16 **if** $\gamma^* \geq \sum_{\iota=1}^{M} |\gamma_\iota|$ **then return** $(\mathbf{u}, \mathbf{v}, \gamma^*)$

17 $\gamma^* \leftarrow \sum_{\iota=1}^{M} |\gamma_\iota|$

Figure 9: The pseudocode of the subset indicator base learner. After computing $\boldsymbol{\Gamma}^{(j)}$ (lines 2-4) and initializing $\mathbf{u}$ to a random $\{\pm 1\}$-valued vector (line 6), BESTINDICATOR iterates between (43) and (44) until no change in the edge. STUMPBASE, sets the coefficients α_j and finds the best feature j^*.

The most commonly used tree learner is C4.5 [20]. Whereas this tree implementation is a perfect choice for binary ADABOOST, it is suboptimal for ADABOOST.MH since it outputs a single-

label classifier with no guarantee of a positive multi-class edge (31). Although this problem can be solved in practice by building large trees, there is no reason why we should not optimize the multi-class edge using the tree learning framework.

The advantage of our formalization is that we can use any multi-class base classifier of the form (24) for the tree cuts. Formally, a binary classification tree with N inner nodes ($N+1$ leaves) consists of a list of N base classifiers $\mathfrak{H} = (\mathbf{h}_1, \ldots, \mathbf{h}_N)$ of the form

$$\mathbf{h}_j(\mathbf{x}) = \alpha_j \mathbf{v}_j \varphi_j(\mathbf{x}),$$

and two index lists $\mathfrak{l} = (\mathfrak{l}_1, \ldots, \mathfrak{l}_N)$ and $\mathfrak{r} = (\mathfrak{r}_1, \ldots, \mathfrak{r}_N)$ with $\mathfrak{l}, \mathfrak{r} \in (\mathbb{N} \cup \{\text{NULL}\})^N$. $\mathfrak{l}_j$ and $\mathfrak{r}_j$ represent the indices of the left and right children of the jth node of the tree, respectively. The *node* classifier in the jth node is defined recursively as

$$\mathfrak{h}_j(\mathbf{x}) = \begin{cases} -\mathbf{v}_j & \text{if } \varphi_j(\mathbf{x}) = -1 \wedge \mathfrak{l}_j = \text{NULL} \quad \text{(left leaf)}, \\ \mathbf{v}_j & \text{if } \varphi_j(\mathbf{x}) = +1 \wedge \mathfrak{r}_j = \text{NULL} \quad \text{(right leaf)}, \\ \mathfrak{h}_{\mathfrak{l}_j}(\mathbf{x}) & \text{if } \varphi_j(\mathbf{x}) = -1 \wedge \mathfrak{l}_j \neq \text{NULL}, \quad \text{(left inner node)}, \\ \mathfrak{h}_{\mathfrak{r}_j}(\mathbf{x}) & \text{if } \varphi_j(\mathbf{x}) = +1 \wedge \mathfrak{r}_j \neq \text{NULL}, \quad \text{(right inner node)}. \end{cases} \tag{45}$$

The final tree classifier is then

$$\mathbf{h}_{\mathfrak{H},\mathfrak{l},\mathfrak{r}}(\mathbf{x}) = \alpha \mathfrak{h}_1(\mathbf{x}).$$

Note that the tree classifier $\mathbf{h}_{\mathfrak{H},\mathfrak{l},\mathfrak{r}}(\mathbf{x})$ itself is not a factorized classifier (24). In particular, $\mathbf{h}_{\mathfrak{H},\mathfrak{l},\mathfrak{r}}(\mathbf{x})$ uses the local vote vectors $\mathbf{v}_j$ determined by each leaf instead of a global vote vector. On the other hand, the coefficient α is unique, and it is determined in the standard way

$$\alpha = \frac{1}{2} \log \frac{1 + \gamma(\mathfrak{h}_1, \mathbf{W})}{1 - \gamma(\mathfrak{h}_1, \mathbf{W})}$$

based on the edge of the root classifier $\mathfrak{h}_1$. The local coefficients α_j returned by the base learners are discarded (along with the vote vectors in the inner nodes).

Finding the optimal N-leaf tree is a difficult combinatorial problem. Most tree-building algorithms are therefore sub-optimal by construction. For AdaBoost, again, this is not a problem: we can continue boosting as long as the edge is positive. Classification trees are usually built in a greedy manner: at each stage we try to cut all the current leaves j by calling the base learner of the data points reaching the jth leaf, then select the best node to cut, convert the old leaf into an inner node, and add two new leaves. The difference between the different algorithms is in the way the best node is selected. Usually, we select the node that *improves* a gain function the most. In AdaBoost.MH the natural gain is the edge (31) of the base classifier. Since the data set $(\mathbf{X}, \mathbf{Y})$ is different at each node, we include it explicitly in the argument of the full multi-class edge

$$\gamma(\mathbf{v}, \varphi, \mathbf{X}, \mathbf{Y}, \mathbf{W}) = \sum_{i=1}^{n} \sum_{\ell=1}^{K} \mathbb{I}\{x_i \in \mathbf{X}\} w_{i,\ell} v_\ell \varphi(\mathbf{x}_i) y_{i,\ell}.$$

Note that in this definition we do not require that the weights of the selected points add up to 1. Also note that this gain function is additive on subsets of the original data set, so the local edges in the leaves add up to the edge of the full tree. This means that any improvement in the local edge directly translates to an improvement of the tree edge.

The basic operation when adding a tree node with a scalar binary classifier (cut) φ is to separate the data matrices $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{W}$ according to the sign of the classification $\varphi(\mathbf{x}_i)$ for all $\mathbf{x}_i \in \mathbf{X}$. Figure 10 contains the pseudocode of this simple operation.

Building a tree is usually described in a recursive way but we find the iterative procedure easier to explain, so our pseudocode in Figure 11 contains this version. The main idea is to maintain a *priority queue*, a data structure that allows *inserting* objects with numerical *keys* into a set, and extracting the object with the *maximum* key [21]. The key will represent the improvement

```
CutDataSet(X, Y, W, φ(·))
 1   X_- ← Y_- ← W_- ← X_+ ← Y_+ ← W_+ ← ()        ▷ empty vectors
 2   for i ← 1 to n
 3       if x_i ∈ X then
 4           if φ(x_i) = −1 then
 5               X_- ← Append(X_-, x_i)
 6               Y_- ← Append(Y_-, y_i)
 7               W_- ← Append(W_-, w_i)
 8           else
 9               X_+ ← Append(X_+, x_i)
10               Y_+ ← Append(Y_+, y_i)
11               W_+ ← Append(W_+, w_i)
12   return (X_-, Y_-, W_-, X_+, Y_+, W_+)
```

Figure 10: The basic operation when adding a tree node is to separate the data matrices $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{W}$ according to the sign of classification $\varphi(\mathbf{x}_i)$ for all $\mathbf{x}_i \in \mathbf{X}$.

of the edge when cutting a leaf. We first call the base learner on the full data set (line 1) and insert it into the priority queue with its edge $\gamma(\mathbf{v}, \varphi, \mathbf{X}, \mathbf{Y}, \mathbf{W})$ (line 3) as the key. Then in each iteration, we extract the leaf that would provide the best edge improvement among all the leaves in the priority queue (line 7), we partition the data set (line 11), call the base learners on the two new leaves (line 12), and insert them into the priority queue using the difference between the old edge *on the partitioned data sets* and the new edges of the base classifiers in the two new leaves (line 13). When inserting a leaf into the queue, we also save the sign of the cut (left or right child) and the index of the parent, so the index vectors $\mathfrak{l}$ and $\mathfrak{r}$ can be set properly in line 8.

When the priority queue is implemented as a heap, both the insertion and the extraction of the maximum takes $O(\log N)$ time [21], so the total running time of the procedure is $O\big(N(T_{\text{BASE}} + n + \log N)\big)$, where T_{BASE} is the running time of the base learner. Since N cannot be more than n, the running time is $O\big(N(T_{\text{BASE}} + n)\big)$. If the base learners cutting the leaves are decision stumps, the total running time is $O(nKdN)$. In the procedure we have no explicit control over the shape of the tree, but if it happens to be balanced, the running time can further be improved to $O(nKd \log N)$.

B.5 Decision products

In this section we describe another meta-base learner that combines simple base classifiers in a generic way [7]. Similarly to trees, we call the base learner as a subroutine but in an iterative rather than recursive fashion. In the optimization loop we fix all but one of the base classifier terms, temporarily re-label the points, and call the base learner using the "virtual" labels.

Formally, we learn base classifiers of the form

$$\mathbf{h}(\cdot) = \alpha \bigotimes_{j=1}^{m} \mathbf{v}_j \varphi_j(\cdot),$$

where the vote vectors $\mathbf{v}_j$ are multiplied elementwise, and the number of terms m is a complexity parameter that should be validated (similarly to the number of leaves N of decision trees). To maximize the edge (31), we follow a simple iterative approach (Figure 12). In each iteration we fix each base learner except for one φ_j, and maximize the edge with respect to φ_j. Because of

TREEBASE$(\mathbf{X}, \mathbf{Y}, \mathbf{W}, \text{BASE}(\cdot,\cdot,\cdot), N)$

1 $\quad (\alpha, \mathbf{v}, \varphi(\cdot)) \leftarrow \text{BASE}(\mathbf{X}, \mathbf{Y}, \mathbf{W})$

2 $\quad S \leftarrow$ PRIORITYQUEUE $\quad \triangleright\ O(\log N)$ *insertion and extraction of maximum key*

3 $\quad \text{INSERT}\big(S, (\mathbf{v}, \varphi(\cdot), \mathbf{X}, \mathbf{Y}, \text{NULL}, 0), \gamma(\mathbf{v}, \varphi, \mathbf{X}, \mathbf{Y}, \mathbf{W})\big) \quad \triangleright$ *key = edge* γ

4 $\quad \mathfrak{H} \leftarrow ()$ $\quad \triangleright$ *initialize classifier list*

5 $\quad$ **for** $j \leftarrow 1$ **to** N

6 $\quad\quad \mathfrak{l}_j \leftarrow \mathfrak{r}_j \leftarrow \text{NULL}$ $\quad \triangleright$ *initialize child indices*

7 $\quad\quad (\mathbf{v}_j, \varphi_j(\cdot), \mathbf{X}_j, \mathbf{Y}_j, \bullet, j_\text{P}) \leftarrow \text{EXTRACTMAX}(S)$ $\quad \triangleright$ *best node in the priority queue*

8 $\quad\quad$ **if** $\bullet = -$ **then** $\mathfrak{l}_{j_\text{P}} \leftarrow j$ **else if** $\bullet = +$ **then** $\mathfrak{r}_{j_\text{P}} \leftarrow j$ $\quad \triangleright$ *child index of parent*

9 $\quad\quad \mathfrak{H} \leftarrow \text{APPEND}(\mathfrak{H}, \mathbf{v}_j\varphi_j(\cdot))$ $\quad \triangleright$ *adding* $\mathbf{h}_j(\cdot) = \mathbf{v}_j\varphi_j(\cdot)$ *to* $\mathfrak{H}$

10 $\quad\quad (\mathbf{X}_-, \mathbf{Y}_-, \mathbf{W}_-, \mathbf{X}_+, \mathbf{Y}_+, \mathbf{W}_+) \leftarrow \text{CUTDATASET}\big(\mathbf{X}_j, \mathbf{Y}_j, \mathbf{W}, \varphi_j(\cdot)\big)$

11 $\quad\quad$ **for** $\bullet \in \{-, +\}$ $\quad \triangleright$ *insert children into priority queue*

12 $\quad\quad\quad (\alpha_\bullet, \mathbf{v}_\bullet, \varphi_\bullet(\cdot)) \leftarrow \text{BASE}(\mathbf{X}_\bullet, \mathbf{Y}_\bullet, \mathbf{W}_\bullet)$

13 $\quad\quad\quad \text{INSERT}\big(S, (\mathbf{v}_\bullet, \varphi_\bullet(\cdot), \mathbf{X}_\bullet, \mathbf{Y}_\bullet, \bullet, j), \gamma(\mathbf{v}_\bullet, \varphi_\bullet, \mathbf{X}_\bullet, \mathbf{Y}_\bullet, \mathbf{W}_\bullet) - \gamma(\mathbf{v}_j, \varphi_j, \mathbf{X}_\bullet, \mathbf{Y}_\bullet, \mathbf{W}_\bullet)\big)$
$\quad\quad\quad\quad \triangleright$ *key = edge improvement over parent edge*

14 $\quad \alpha = \dfrac{1}{2}\log\dfrac{1+\gamma(\mathfrak{h}_1, \mathbf{W})}{1-\gamma(\mathfrak{h}_1, \mathbf{W})}$ $\quad \triangleright$ *standard coefficient of the full tree classifier* $\mathfrak{h}_1$ *(45)*

15 $\quad$ **return** $(\alpha, \mathfrak{H}, \mathfrak{l}, \mathfrak{r})$

Figure 11: The pseudocode of the tree base learner. N is the number of inner nodes. The algorithm returns a list of base classifiers $\mathfrak{H}$, two index lists $\mathfrak{l}$ and $\mathfrak{r}$, and the base coefficient α. The tree classifier is then defined by (45).

the product form of the edge (31), we can carry out this maximization by simply calling the base learner of φ_j with "virtual" labels defined as the product of the real labels and the outputs of the remaining base learners (line 9 in Figure 12). Formally, when optimizing the jth term $\mathbf{v}_j\varphi_j(\cdot)$, the edge can be written as

$$\gamma = \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell}\prod_{j'=1}^{m} v_{j',\ell}\varphi_{j'}(\mathbf{x}_i)y_{i,\ell} = \sum_{i=1}^{n}\sum_{\ell=1}^{K} w_{i,\ell}v_{j,\ell}\varphi_j(\mathbf{x}_i)\left(\prod_{\substack{j'=1\\ j'\neq j}}^{m} v_{j',\ell}\varphi_{j'}(\mathbf{x}_i)y_{i,\ell}\right),$$

so maximizing the edge can be simply done by setting the "virtual" labels to

$$y'_{i,\ell} = y_{i,\ell}\prod_{\substack{j'=1\\ j'\neq j}}^{m} v_{j',\ell}\varphi_{j'}(\mathbf{x}_i) = y_{i,\ell}\frac{\prod_{j'=1}^{m} v_{j',\ell}\varphi_{j'}(\mathbf{x}_i)}{v_{j,\ell}\varphi_j(\mathbf{x}_i)}$$

and calling the base learner with these labels.

The procedure is guaranteed to converge since in each batch of m iterations at least one "virtual" sign $y'_{i,\ell}$ must change (otherwise the base learner in line 10 returns the same set of m classifiers and we have equality in line 11) and the number of different sign vectors is finite. As with INDICATORBASE in Section B.3, we found that in practice PRODUCTBASE always returns after a few iterations.

The algorithm can be applied directly to multi-valued preferences by adding the optimization of the vote vectors $\mathbf{v}^{(t)}$. In this case, the preference prediction matrix becomes a hyper-matrix with

PRODUCTBASE($\mathbf{X}, \mathbf{Y}, \mathbf{W}$, BASE$(\cdot,\cdot,\cdot), m$)

1 **for** $j \leftarrow 1$ **to** m

2 $\quad \varphi_j(\cdot) \leftarrow 1, \mathbf{v}_j \leftarrow \mathbf{1}$ ▷ *terms are initialized to the constant* 1 *function*

3 $\alpha \leftarrow 1$

4 **while** TRUE

5 $\quad$ **for** $j \leftarrow 1$ **to** m ▷ *loop over all terms while improvement*

6 $\quad\quad \varphi^*(\cdot) \leftarrow \prod_{j'=1}^{m} \varphi_{j'}(\cdot), \mathbf{v}^* \leftarrow \bigotimes_{j'=1}^{m} \mathbf{v}_{j'}, \alpha^* \leftarrow \alpha$ ▷ *save current optimal classifier*

7 $\quad\quad$ **for** $i \leftarrow 1$ **to** n

8 $\quad\quad\quad$ **for** $\ell \leftarrow 1$ **to** K

9 $\quad\quad\quad\quad y'_{i,\ell} \leftarrow y_{i,\ell} \dfrac{v^*_\ell \varphi^*(\mathbf{x}_i)}{v_{j,\ell} \varphi_j(\mathbf{x}_i)}$ ▷ *"virtual" labels*

10 $\quad\quad (\alpha, \mathbf{v}_j, \varphi_j(\cdot)) \leftarrow$ BASE$(\mathbf{X}, \mathbf{Y}', \mathbf{W})$

11 $\quad\quad$ **if** $Z\left(\alpha \bigotimes_{j'=1}^{m} \mathbf{v}_{j'} \varphi_{j'}, \mathbf{W}\right) \geq Z\left(\alpha^* \mathbf{v}^* \varphi^*, \mathbf{W}\right)$ **then** ▷ *no improvement*

12 $\quad\quad\quad$ **return** $(\alpha^*, \mathbf{v}^*, \varphi^*(\cdot))$

Figure 12: The pseudocode of the product base learner. m is the number of base classifier terms. The vote vectors are multiplied elementwise in lines 6 and 11.

vector-valued elements. The formulation also permits one to boost products of more than two indices (or to use two or more times a base learner on the same index) which goes beyond the matrix-decomposition-based formulation of the collaborative filtering problem. It is also quite straightforward to combine collaborative features with "traditional" numerical or nominal covariates by allowing mixed products.

B.6 Bandit based acceleration of base learning

The bandit based acceleration technique helps the base learner to find faster a reasonably good base classifier in every boosting iteration. The (adversarial) bandit setup can be thought as a two-player game where one of the players is the decision maker and the other is the adversary. The adversary assigns real rewards (typically from $[0, 1]$) to each arm. Then the decision maker pulls an arm and incurs the reward of the pulled arm. They play for T iterations. The goal of the decision maker is to maximize the sum of the rewards incurred during the T iterations. In the stochastic bandit setup the rewards are drawn from stationary distributions for each arm.

For using adversary bandit algorithms to accelerate base learning, first, it is assumed that a partitioning is defined on the space of base classifiers in advance. These partitions correspond to the arms in the bandit setup. The bandit algorithm selects a subset in each boosting (or bandit) iteration, and the base learner runs only on the selected subset. The reward is monotonically increasing with respect to the weighted accuracy of the base classifier found by the base learner, so the adversary player is fully defined by the current weighting $\mathbf{W}^{(t)}$. This means that the rewards are changing from iteration to iteration and so, in principle, the stochastic bandit setup does not fit to this acceleration framework since the rewards are not stationary over the boosting iterations.

In the case of decision stumps (Section B.2) the most natural partitioning is to assign a subset to each feature: $\mathcal{H}_j = \{\varphi_{j,b}(\mathbf{x}) : b \in \mathbb{R}\}$. The MULTIBOOST package implements stumps with either the adversarial bandit algorithm EXP3.P [22] or the stochasic bandit algorithms UCB [23] EXP3G [24]. Although the theoretical setup favors adversarial bandits, the stochastic versions

also work well in practice. For further information we refer the reader to [9, 10].

References

[1] R. E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[2] L. Breiman. Prediction games and arcing classifiers. *Neural Computation*, 11:1493–1518, 1999.

[3] G. Rätsch and M. K. Warmuth. Maximizing the margin with boosting. In *Proceedings of the 15th Conference on Computational Learning Theory*, 2002.

[4] J.K. Bradley and R.E. Schapire. FilterBoost: Regression and classification on large datasets. In *Advances in Neural Information Processing Systems*, volume 20. The MIT Press, 2008.

[5] P. Viola and M. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57:137–154, 2004.

[6] L. Bourdev and J. Brandt. Robust object detection via soft cascade. In *Conference on Computer Vision and Pattern Recognition*, volume 2, pages 236–243. IEEE Computer Society, 2005.

[7] B. Kégl and R. Busa-Fekete. Boosting products of base classifiers. In *International Conference on Machine Learning*, volume 26, pages 497–504, Montreal, Canada, 2009.

[8] G. Escudero, L. Màrquez, and G. Rigau. Boosting applied to word sense disambiguation. In *Proceedings of the 11th European Conference on Machine Learning*, pages 129–141, 2000.

[9] R. Busa-Fekete and B. Kégl. Accelerating AdaBoost using UCB. In *KDDCup 2009 (JMLR W&CP)*, volume 7, pages 111–122, Paris, France, 2009.

[10] R. Busa-Fekete and B. Kégl. Fast boosting using adversarial bandits. In *International Conference on Machine Learning*, volume 27, pages 143–150, 2010.

[11] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55:119–139, 1997.

[12] J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Math. Ann.*, 100:295–320, 1928.

[13] G. Rätsch and M. K. Warmuth. Efficient margin maximizing with boosting. *Journal of Machine Learning Research (submitted)*, 2003.

[14] C. Rudin, I. Daubechies, and R.E. Schapire. The dynamics of AdaBoost: Cyclic behavior and convergence of margins. *Journal of Machine Learning Research*, 5:1557–1595, 2004.

[15] L. Reyzin and R. E. Schapire. How boosting the margin can also boost classifier complexity. In *International Conference on Machine Learning*, pages 753–760, 2006.

[16] W. Cohen. Learning trees and rules with set-valued features. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 709–716, 1996.

[17] W. Cohen and Y. Singer. A simple, fast, and effective rule learner. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 335–342, 1999.

[18] J. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[19] J. Quinlan. Bagging, boosting and C4.5. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 725–730, 1996.

[20] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[21] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 2009.

[22] P. Auer, N. Cesa-Bianchi, Y. Freund, and R.E. Schapire. The non-stochastic multi-armed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[23] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002.

[24] L. Kocsis and Cs. Szepesvári. Reduced-variance payoff estimation in adversarial bandit problems. In *Proceedings of the ECML-2005 Workshop on Reinforcement Learning in Non-Stationary Environments*, 2005.