Kaggle LSHTC4 Winning Solution

Antti Puurula[1], Jesse Read[2], and Albert Bifet[3]

[1] Department of Computer Science, The University of Waikato, Private Bag 3105, Hamilton 3240, New Zealand
[2] Department of Information and Computer Science, Aalto University, FI-00076 Aalto, Espoo, Finland
[3] Huawei Noah's Ark Lab, Hong Kong Science Park, Shatin, Hong Kong, China

1 Overview

Our winning submission to the 2014 Kaggle competition for Large Scale Hierarchical Text Classification (LSHTC) consists mostly of an ensemble of sparse generative models extending Multinomial Naive Bayes. The base-classifiers consist of hierarchically smoothed models combining document, label, and hierarchy level Multinomials, with feature pre-processing using variants of TF-IDF and BM25. Additional diversification is introduced by different types of folds and random search optimization for different measures. The ensemble algorithm optimizes macroFscore by predicting the documents for each label, instead of the usual prediction of labels per document. Scores for documents are predicted by weighted voting of base-classifier outputs with a variant of Feature-Weighted Linear Stacking. The number of documents per label is chosen using label priors and thresholding of vote scores.

This document describes the models and software used to build our solution. Reproducing the results for our solution can be done by running the scripts included in the Kaggle package[4]. A package omitting precomputed result files is also distributed[5]. All code is open source, released under GNU GPL 2.0, and GPL 3.0 for Weka and Meka dependencies.

2 Data Segmentation

Source files: MAKE_FILES, nfold_sample_corpus.py, fast_sample_corpus.py, shuffle_data.py, count_labelsets2.py

Training data segmentation is done by the script MAKE_FILES, included in the code package. This segments the original training dataset train.txt by random sampling into portions for base-classifier training and for ensemble training.

[4] https://kaggle2.blob.core.windows.net/competitions/kaggle/3634/media/LSHTC4_winner_solution.zip
[5] https://kaggle2.blob.core.windows.net/competitions/kaggle/3634/media/LSHTC4_winner_solution_omit_resultsfiles.zip

2,341,782 documents are segmented for the former portion and 23,654 documents for the latter. The base-classifier training dataset dry_train.txt is further sampled into 10 different folds, each with a 1000 document held-out portion dry_dev.txt for parameter optimization. Folds 0-2 have exclusive and different sampled sets for dry_dev.txt. Folds 3-5 sample dry_train.txt randomly into 3 exlusive training subsets, with a shared optimization portion. Folds 6-9 segment dry_train.txt in the original data order into 4 exclusive training subsets, with a shared optimization portion. For all folds, the training datasets are further shuffled to improve the online pruning of parameters in training.

3 Base-classifiers

Source files: SGM-45l/, SGM-45l_je/, Metaopt2.py, Make_templates.py, results/, RUN_DEVS, RUN_EVALS, meka.jar

The base-classifiers consist mostly of sparse generative model extensions of Multinomial Naive Bayes (MNB). These extend MNB by introducing constrained finite mixtures at the document and hierarchy level nodes, and performing inference from the Multinomial node-conditional models using hierarchical smoothing, and kernel densities in case of document-conditional nodes. A special case is models using BM25 for kernel densities and no hierarchical smoothing. The models are stored in a sparse precomputed format, and inference using inverted indices is used to reduce the inference complexity according to the sparsity of the model. The constrained mixture modeling and sparse inference makes the models as scalable for text modeling as Naive Bayes and KNN, but with higher modelling accuracy. A detailed description of basic models of this type are given in [1, 2]. Since the LSHTC models can contain up to 100 million parameters for word counts, the models are provided as configuration files in the package. Estimating the models from training data takes negligible time more compared to reading saved model files.

A development version of the SGMWeka toolkit[6] was customized to implement the models. The customized version is included as the Java source directory SGM-45l, and the program SGM_Tests.java used for training and testing the models can be compiled without external dependencies. The documentation for SGMWeka version 1.4.4[7] is accurate, but the development version contains additional functionalities. A modified version is in the directory SGM-45l_je. This includes the Meka toolkit[8] for doing multi-label decomposition used by one of the base-classifiers.

[6] http://sourceforge.net/projects/sgmweka/
[7] http://sourceforge.net/p/sgmweka/wiki/SGMWeka%20Documentation%20v.1.4.4/
[8] http://meka.sourceforge.net/

The script Metaopt2.py optimizes a base-classifier on a development set according to a chosen performance measure, by iteratively estimating the classifier and classifying the development data portion. The script RUN_DEVS runs the development and compresses the log files. The configuration files for Metaopt2.py describes all the parameters provided to a SGM_Tests call, as well as the optimization measure to extract from the last line of the SGM_Tests log file. Metaopt2.py performs a Gaussian Random Search [3] for the chosen parameters, constrained and transformed according to the configuration file. The directories results_* contain the first and last parameter configuration file for each base-classifier type, after a 40x8 iteration random search. Some classifiers were constructed by copying the parameters for similar folds (3,4,5), and some used manually chosen parameter configurations. These classifiers have the final iteration parameter file wikip_large_X_params.txt_39_0, but not the initial file wikip_large_X_params.txt. The script Make_templates.py makes the parameter template files as specified in the global variable "configs".

The template files describe the model by suffixing the file name with modifications. For example, "mnb_mafs2_s8_lp_u_jm2_bm18ti_pct0_ps5_thr16.template" modifies a Multinomial Naive Bayes by optimizing the parameters for a modified version of macro-Fscore (_mafs2), uses data fold 8 (_s8), the Label Powerset method for multi-label classification (_lp), smoothing by a uniform background distribution (_u), a BM25 variant for feature weighting (_bm18ti), uses a safe pruning of pre-computed parameters (_pct0), constrains the scaling of label prior (_ps5) and uses 16 threads for parallel classification.

Some of the modifications have little influence on the results, such as _thr16 that instructs SGM_Tests to use 16 threads. More detailed explanations of the important modifications are given in the following sections. A total of 54 base-classifiers are used in the ensemble, selected down to 42 base-classifiers by model selection. Table 1 shows the base-classifiers sorted according to comb_dev.txt macro-averaged Fscore. It should be noted that the parameter ranges for some of the modifications were adjusted during the competition, and the parameter ranges in the individual template files can differ from those in Make_templates.py.

The word count vectors for LSHTC were preprocessed by the organizers to remove common words, stopwords and short words, as can be seen from looking at the distributions of words in the vectors. This causes problems for some models such as Multinomial models of text, that assume word vectors to distribute normally. Feature transforms and weighting can be used to correct this. Feature weighting is done by each base-classifier separately, using variants of TF-IDF and BM25. All models use 1-3 parameters to optimize the feature weighting on the dry_dev.txt portion of the fold. A variant of BM25 that proved most successful has the suffix "_bm18ti". As seen in TFIDF.java, this combines the term count normalization of BM25 with the parameterized length normalization and

id	excluded	parameter configuration	maFscore
7		mafs3_s1_uc1_jm3_bm18ti_pci7_pct0_psX_fb_iw2	0.4155
9		mafs3_s1_uc1_jm3_bm18ti_pci7_pct0_psX_iw2	0.4082
11	X	mafs3_s2_uc1_jm2_bm18tid_pci7_pct0_ps8_iw1	0.3993
13		mafs3_s3_kd_u_jm3_kdp5_bm18ti_pct0_ps7_iw2	0.3982
17		mafs3_s4_kd_u_jm3_kdp5_bm18ti_pct0_ps7_iw2	0.3982
8		mafs3_s1_uc1_jm3_bm18ti_pci7_pct0_psX_iw1	0.3866
10	X	mafs3_s2_u_lp_jm2_bm18tib_pct0_ps7_iw0	0.3795
20		mafs3_s5_kd_u_jm3_kdp5_bm18ti_pct0_ps7_iw0	0.3771
12		mafs3_s3_kd_u_jm3_kdp5_bm18ti_pct0_ps7_iw0	0.3763
6		mafs3_s1_u_jm3_bm18ti_pct0_ps7_iw0	0.3689
16		mafs3_s4_kd_u_jm3_kdp5_bm18ti_pct0_ps7_iw0	0.3615
14		mafs3_s3_kd_uc1_jm2_kdp5_bm18tid_pct0_ps8_iw1	0.3466
5		mafs3_s0_kd_nobo_bm25c2_mi2_ps2_iw0	0.3380
19		mafs3_s4_u_jm2_bm18tib_pci6_pct0_ps7_cs0_iw2	0.3346
18	X	mafs3_s4_u_jm2_bm18tib_mc0_pci6_pct0_ps7_cs0_iw0	0.3114
21		mafs3_s5_u_jm2_bm18tib_mc0_pci6_pct0_ps7_cs0_iw0	0.3091
15		mafs3_s3_u_jm2_bm18tib_mc0_pci6_pct0_ps7_cs0_iw0	0.3082
33		mafs_s2_lp_u_jm5_pd2_bm16ti_mc0_pct0_ps0	0.2860
50		mjac_s2_kd_nobo_bm25c2_mc0_mlc0_ps2_lt5_mr0_tk1	0.2856
0		mafs2_s2_lp_u_jm2_bm18tib_mc0_pct0_ps5	0.2815
28	X	mafs_s1_kd_nobo_bm25c2_mc0_mlc0_ps2_lt5_mr0_tk2	0.2805
32		mafs_s2_lp_u_jm4_bm20ti_mc0_pct0_ps2	0.2723
27		mafs_s0_lp_u_jm2_bm18tic_fb3_mc0_pct0_ps6	0.2686
44	X	mjac_s0_lp_u_jm2_pd2_tXiX3_fb2_mc0_pci1_pct0_ps0	0.2678
52		ndcg5b_s4_kd_u_jm2_kdp5_bm18tib_mc0_pci0_pct0_mlc0_ps6_tk0	0.2659
51		ndcg5b_s3_kd_u_jm2_kdp5_bm18tib_mc0_pci0_pct0_mlc0_ps6_tk0	0.2650
53		ndcg5b_s5_kd_u_jm2_kdp5_bm18tib_mc0_pci0_pct0_mlc0_ps6_tk0	0.2643
30		mafs_s1_lp_u_jm6_tiX5_mc0_pct0_ps0	0.2618
29	X	mafs_s1_lp_u_jm4_pd2_tXiX2_fb2_mc0_pct0_ps0	0.2612
45	X	mjac_s0_lp_u_jm2_tiX3_mc0_pct0_ps0	0.2592
31	X	mafs_s2_lp_u_jm4_bm18ti_mc0_pct0_ps2	0.2567
23		mafs3_s7_kd_uc1_jm2_kdp5_bm18tid_mc0_pci1_pct0_ps8_iw1_ch80	0.2550
42		mifs_s2_lp_u_jm2_bm18tib_fb3_mc0_pct0_ps5	0.2530
46	X	mjac_s0_lp_u_jm4_bm15ti_mc0_pct0_ps0	0.2489
22		mafs3_s6_kd_uc1_jm2_kdp5_bm18tid_mc0_pci1_pct0_ps8_iw1_ch80	0.2444
35		mafs_s4_kd_u_jm3_kdp5_tXiX2_mc0_pci0_pct0_mlc0_ps5_lt5_mr0_tk2	0.2441
34		mafs_s3_kd_u_jm3_kdp5_tXiX2_mc0_pci0_pct0_mlc0_ps5_lt5_mr0_tk2	0.2422
36		mafs_s5_kd_u_jm3_kdp5_tXiX2_mc0_pci0_pct0_mlc0_ps5_lt5_mr0_tk2	0.2421
49		mjac_s1_u_jm3_tiX1_mc0_pci1_pct0_mlc0_ps1_lt1_mr0	0.2410
48	X	mjac_s1_u_jm2_tiX1_mc0_pct0_mlc0_ps2_lt2_mr0_tk0	0.2395
24		mafs3_s8_kd_uc1_jm2_kdp5_bm18tid_mc0_pci1_pct0_ps8_iw1_ch80	0.2335
26	X	mafs_s0_lp_u_jm2_bm18tib_mc0_pct0_ps5	0.2245
41		mifs_s1_lp_u_jm2_bm18tib_mc0_pct0_ps5	0.2232
25		mafs3_s9_kd_uc1_jm2_kdp5_bm18tid_mc0_pci1_pct0_ps8_iw1_ch80	0.2108
47		mjac_s0_u_jm3_bm18ti_pct0_ps5_je	0.2040
43		mjac_s0_lp_bm25c1_mc0_mlc0_ps3	0.1924
38		mafs_s7_kd_u_jm3_kdp1_bm18ti_mc0_pci1_pct0_ps5_lt5_mr1_tk2_ch80	0.1787
39		mafs_s8_kd_u_jm3_kdp1_bm18ti_mc0_pci1_pct0_ps5_lt5_mr1_tk2_ch80	0.1632
2		mafs2_s7_lp_u_jm2_bm18ti_pct0_ps5	0.1554
1	X	mafs2_s6_lp_u_jm2_bm18ti_pct0_ps5	0.1529
3		mafs2_s8_lp_u_jm2_bm18ti_pct0_ps5	0.1513
37		mafs_s6_kd_u_jm3_kdp1_bm18ti_mc0_pci1_pct0_ps5_lt5_mr1_tk2_ch80	0.1469
40		mafs_s9_kd_u_jm3_kdp1_bm18ti_mc0_pci1_pct0_ps5_lt5_mr1_tk2_ch80	0.1452
4		mafs2_s9_lp_u_jm2_bm18ti_pct0_ps5	0.1357

Table 1. Base-classifiers sorted in the order of comb_dev.txt macro-averaged Fscore, computed over the labels occurring in the set. Excluded models were removed by model selection from the ensemble.

idf weighting from TF-IDF that has been used earlier [3].

The Multinomials use hierarchical smoothing with a uniform background distribution [2]. The variant "_uc1" uses a uniform distribution interpolated with a collection model, improving the accuracy by a small amount. All models use Jelinek-Mercer "_jmX" for smoothing label and hierarchy level Multinomials, and Dirichlet Prior smoothing "_kdpX" for smoothing kernel density document models. The feature selection done by the organizers cause very unusual smoothing parameter configurations to be optimal. With Jelinek-Mercer values less than a heavy amount such as 0.98 become rapidly worse, with some models using a smoothing coefficient of 0.999.

Parameter pruning is chosen by the modifiers "_mcX", "_pciX", "_pctX", "_mlcX". "_mcX" prunes word features based on their frequency. "_pciX" selects on-line pruning of conditional parameters, "_pctX" performs mostly safe pruning of precomputed conditional parameters, "_mlcX" prunes labels based on their frequency.

One special classifier is the variant using the modifer "_je". This requires a development version of the Meka toolkit and the other files in the directory /SGM-45l_je. This model does classification with label powersets decomposed into meta-labels, and transforms the meta-labels back into labelsets after classification. The labelset decompositions are stored in a precomputed file loaded by the modified version of SGM_Tests.

Kernel densities are selected with the modifier "_kd", passing -kernel_densities to SGM_Tests. This constructs document-conditional models, and computes label-conditional probabilities using the document-conditionals as kernel densities [2]. The modifiers "_csX" load the LSHTC4 label hierarchy, and use random parent nodes to smooth the label-conditional Multinomials. The Label Powerset method for mapping a multi-label problem into a multi-class problem is done by the modifier "_lp", passing -label_powerset to SGM_Tests.

The modifier "_nobo" combined with "_kd" produces models for document instances with no back-off smoothing by label-conditional models. The modifiers "_bm25X" use BM25 instead of Multinomial distances. Combined with "_kd" and "_nobo", this produces a model that uses BM25 for kernel densities of each label.

The modifiers "_ndcg5", "_mjac", "_mifs" and "_mafsX" choose the optimization measure for MetaOpt2.py. These correspond to NDCG@5, Mean of Jaccard scores per instance, micro-averaged Fscore, macro-averaged Fscore and surrogate measures for maFscore. It was noticed early in the competition that computing and optimizing maFscore is problematic, since not all labels are present in the training set, and any subset chosen for optimization will contain only a tiny fraction of the 325k+ labels, with the rest being missing labels. Since most labels are missing labels, and any number of false positives for a missing label will equal an fscore of 0, optimizing maFscore becomes problematic. The "_mafsX" surrogates used two attempts to penalize for false positives of missing labels, but these was abandoned for a method that allows optimizing macroFscore better without producing too many instances per label.

The modifiers "_iwX" select a method developed in this competition. This causes the base-classifier to predict instances per label, instead of labels per instance. A sorted list of the best scores for each label is stored, and for each classified instance the lists for labels are updated. A full distribution of labels is computed for each instance, and the label→instance scores are computed from the rank of the label for each instance. After classification of the dataset, the sparse label→instances scores are transposed and outputted and evaluated in the instance→labels format. The arguments -instantiate_weight X and -instantiate_threshold X passed to SGM_Tests control the number of top scoring instances stored for each label. The ensemble combination uses transposed prediction of the same kind to do the classification.

4 Ensemble Model

Source files: RUN_METACOMB, MetaComb2.java, TransposeFile.py, SelectClassifiers.py, SelectDevLabels.py, comb_dev_results/, eval_results/, weka.jar

The ensemble model is built on our earlier LSHTC3 ensemble [3], but performs classification by predicting instances per label. The classifier vote weight prediction is a case of Feature-Weighted Linear Stacking [4], but the regression models are trained separately for each base-classifier, using reference weights that approximate optimal weights per label in a development set.

The base-classifier result files are tranposed from a document→labels per line format to a label→documents per line format. After prediction the ensemble result file is transposed back to the document→labels per line .csv format used by the competition. The script RUN_METACOMB performs all the required steps, using the result files stored in /comb_dev_results for training the ensemble and /eval_results to do the classifier combination.

Metacomb2.java perfoms the ensemble classification. The ensemble uses linear regression models to predict the weight of each base-classifier, using metafeatures computed from label information and classifier outputs to predict the optimal classifier weight for each label. The most useful metafeatures in the LSHTC3 submission used labelset correlation features between the base-classifiers for each document instance [3]. This ensemble uses instance-set correlation features for each label analogously.

metafeature	description
labelProb	indicator feature for low-frequency labels (<10)
labelProb2	indicator feature for high-frequency labels (>50)
uniqInstancesets	# different instance sets in the classifier outputs
maxVotes	# votes given to most voted instance set
minInstFreq_i	the frequency of least frequent instance in the output of classifier i
maxInstFreq_i	the frequency of most frequent instance in the output of classifier i
minInstCount_i	the count of the lowest count instance in the output of classifier i
instCount_i	# instances in the output of classifier i
emptySet_i	indicator if the classifier output i has no instances for the label
setCount_i	# of classifiers with the same output as classifier i
modePrec_i	precision of classifier i using the mode of outputs as reference
modeRec_i	recall of classifier i using the mode of outputs as reference
modeJaccard_i	Jaccard similarity of classifier i and the mode of outputs
maxPrec_i_j	intersection of classifier i and j outputs, divided by maximum length

Table 2. Metafeatures used for voting classifier weights. Metafeatures are computed for each label, given the instance set outputs from each base-classifier. Regression models for each baseclassifier weight uses the features that match the classifier id i, and not the metafeatures for other classifiers. Metafeature maxPrec_i_j is computed for all the other base-classifiers j, resulting in 42-1 additional metafeatures. Metafeatures are normalized and log-transformed based on development set performance.

Table 2 shows the metafeatures used by MetaComb2.java. For efficiency and memory use, MetaComb2 adds the correlation metafeatures to each base-classifier before predicting the vote weights, and doesn't keep all possible metafeatures in memory at any time. This keeps the memory complexity of the ensemble combination linear in the number of base-classifiers. Functions constuctData(), pruneGlobalFeatures() and addLocalFeatures() in MetaComb2.java show how the features are constructed as Weka [5] Instances.

The regression models use Weka LinearRegression for implementing the variant of Feature-Weighted Linear Stacking. For each label in comb_dev.txt, optimal reference weights are approximated by distributing a weight of 1 uniformly to the base-classifiers that score highest on the performance measure. Initially fscore was used as the measure, as averaging the fscores across the labels gives maFscore. This however doesn't use rank information in the instance sets. A small improvement in maFscore was gained by using a similar measure that takes rank information into account. approximateOracleWeights() and updateEvaluationResults() in MetaComb2.java show how the reference vote weights are constructed.

Following vote weight prediction, the label$\rightarrow$instances scores are summed for each instance from the weighted votes in the function voteFold(). A combination of label prior information and thresholding similar to one used in the base-classifiers is used to choose the number of instances per label. The label prior information selects a number of instances for the label proportional to the

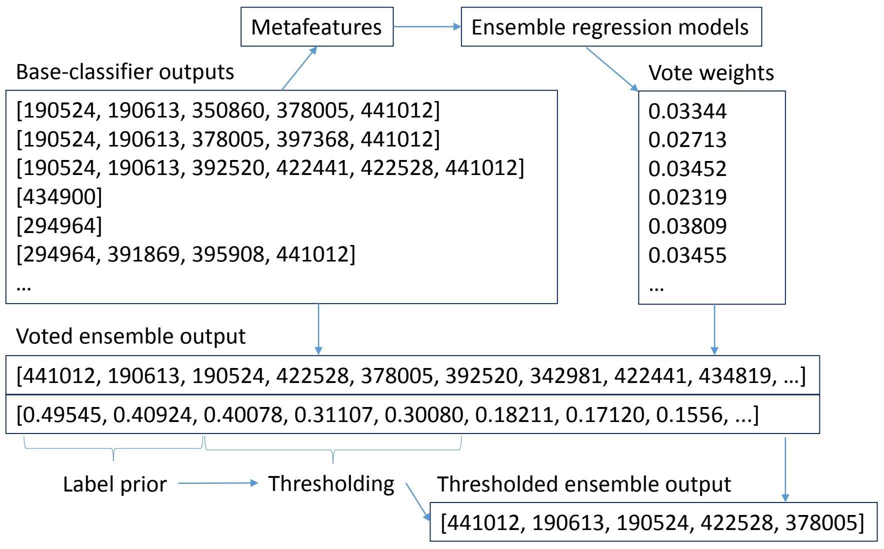

Fig. 1. Ensemble voting and selection of instances from the base-classifier outputs.

label frequency in training data, multiplied by the parameter 0.95 passed to set_instantiate(). The thresholding then includes to the set all instances with score more than 0.5 of the mean of the initial instance set scores. Figure 1 illustrates the ensemble combination and selection of instances.

Development of the ensemble by n-fold cross-validation can be done by changing the global variable "developmentRun" in MetaComb2.java to 1. Selection of base-classifiers can be done by giving the classifiers to remove as integer arguments to MetaComb2. The list of removed classifiers used in the final evaluation run in RUN_METACOMB was developed by running the classifier selection script SelectClassifiers.py with the n-fold crossvalidation. SelectClassifiers.py performs hill-climbing searches, maximizing the output of MetaComb2 by removing and adding classifiers to the ensemble.

5 How to Generate the Solution

The programs and scripts described above can be run to produce the winning submission file. Some of the programs can take considerable computing resources to produce. Both optimizing the base-classifier parameters and classifying the 452k document test set can take several days or more, depending on the model. We used a handful of quadcore i7-2600 CPU processors with 16GB RAM over the competition period to develop and optimize the models. At least 16GB RAM is required to store the word counts reaching 100M parameters. Ensemble combination takes less than 8GB memory, and can be computed from the provided .results files. The base-classifier result files are included in the distribution, as computing these takes considerable time.

For optimizing base-classifiers, compile SGM_Tests.java with javac, configure Make_templates.py or copy an existing template, and run RUN_DEVS. For classifying the comb_dev.txt and test.txt results with a base-classifier, configure and run RUN_EVALS. For combining the base-classifier results with the ensemble, run RUN_METACOMB. The global variables in each script can be modified to change configurations.

6 What Gave us the First Place?

The competition posed a number of complications different from usual Kaggle competitions. Most of our tools were developed over the last LSHTC challenges, and this gave us a big advantage. The biggest complication in the competition was scalability of both the base-classifiers and ensemble. Our solution uses sparse storage and inverted indices for inference, a modeling idea that enabled us to use an ensemble of tens of base-classifiers. With the SGMWeka toolkit we could combine parameterized feature weighting [3], hierarchical smoothing [2], kernel densities [2], model-based feedback [6], etc. Other participants used KNN with inverted indices, but our solution provides a diversity of structured probabilistic base-classifiers with much better modeling accuracies.

Another complication was the preprocessed pruned feature vectors. This made usual Multinomial or Language Model solutions usable only with very untypical and heavy use of linear interpolation smoothing. The commonly used TF-IDF feature transforms also corrected the problem only somewhat. Our solutions for smoothing and feature weighting with a customized BM25 variant took extensive experimentation to discover, but improved the accuracy considerably. It is likely that the other teams had less sophisticated text similarity measures available, and the ones having good measures scored better in the contest.

The most difficult complication in the contest was the choice of maFscore for evaluation measure, in contrast to earlier LSHTC competitions. What surprised the contestants was that optimization of maFscore with high numbers of labels is problematic, since most labels will be missing. With maFscore a label occurring once is just as important as one occurring 1000 times, and a label never predicted and one predicted by a 1000 false positives have the same effect on the score. Combined with most labels missing, normal optimization of classifiers proved difficult. It took us some time to figure out the right way to solve this problem, but the solution made it possible for us to compete for the win. Before

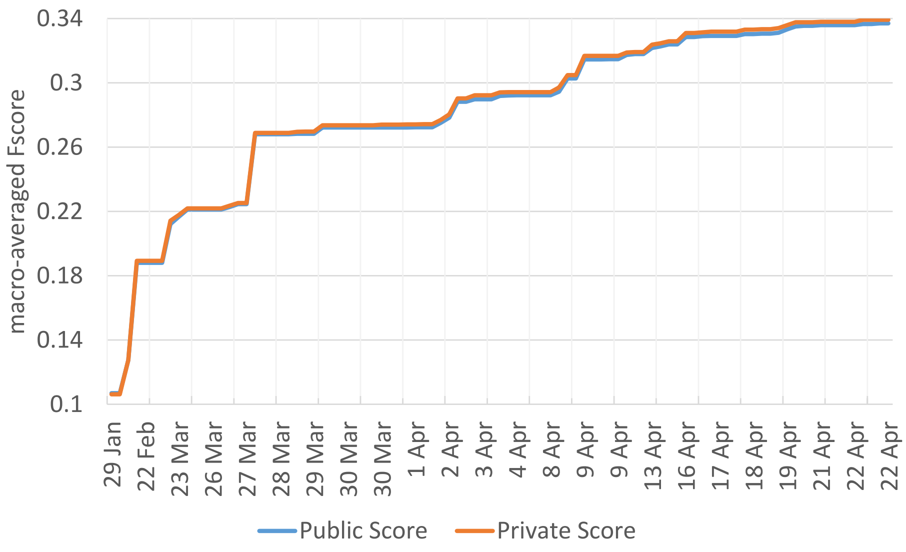

Fig. 2. Progress on the test.txt macro-averaged Fscore during the competition. Growing the ensemble brought steady improvement, implementing the transposed prediction caused jumps in the maFscore.

developing the transposed prediction used in both the base-classifiers and the ensemble, our leaderboard score was around 22%. A couple of simple corrections for maximizing maFscore correctly brought the ensemble combination close to 27%, and using the transposed prediction with a larger and more diverse ensemble gave us the final score close to 34%. Other participants noticed this problem of optimizing maFscore, but likely most of them did not find a good solution.

The use of metafeature regression in the ensemble instead of majority voting proved a moderate improvement of about 0.5%, and this much was needed for the win. It is likely that the metafeatures optimized on the 23k comb_dev.txt documents looked different from the metafeatures computed for the 452k test.txt documents, even though the metafeatures were chosen or normalized to be stable to change in the number of documents. The optimal amount of regularization for the Weka LinearRegression was untypically high at 1000. More complicated Weka regression models for the vote weight prediction failed to improve the test set score, likely due to overfitting the somewhat unreliable features. Another reason could be the small size of the comb_dev.txt for ensemble combination. The ensemble fits the parameters for 55 metafeatures to predict the vote weight of each of the 42 base-classifiers, using only 23k points of data shared by the 42 regression models. The improvement from Feature Weighted Linear Stacking could have been considerably larger, if a larger training set had been segmented for the ensemble.

7 Acknowledgements

We'd like to thank Kaggle and the LSHTC organizers for their work in making the competition a success, and the machine learning group at the University of Waikato for the computers we used for our solution.

References

[1] Puurula, A.: Scalable text classification with sparse generative modeling. In: Proceedings of the 12th Pacific Rim International Conference on Trends in Artificial Intelligence. PRICAI'12, Berlin, Heidelberg, Springer-Verlag (2012) 458–469

[2] Puurula, A., Myaeng, S.H.: Integrated instance- and class-based generative modeling for text classification. In: Proceedings of the 18th Australasian Document Computing Symposium. ADCS '13, New York, NY, USA, ACM (2013) 66–73

[3] Puurula, A.: Combining modifications to multinomial naive bayes for text classification. In Hou, Y., Nie, J.Y., Sun, L., Wang, B., Zhang, P., eds.: Information Retrieval Technology. Volume 7675 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 114–125

[4] Sill, J., Takcs, G., Mackey, L., Lin, D.: Feature-weighted linear stacking. CoRR **abs/0911.0460** (2009)

[5] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. **11**(1) (November 2009) 10–18

[6] Puurula, A.: Cumulative progress in language models for information retrieval. In: Proceedings of the Australasian Language Technology Association Workshop 2013 (ALTA 2013), Brisbane, Australia (December 2013) 96–100