Jan Hajič
hajicj@gmail.com
17. 2. 2011

Semantic Collocation Recognition

PFL054 2010/11 Term project report

Jan Hajič
17. 2. 2011
hajicj@gmail.com

Jan Hajič
hajicj@gmail.com
17. 2. 2011

1. Task description

The aim of the project is to provide a best-effort solution to a binary classification task of Semantic Collocation Recognition (SCR). The classifier will work with word pairs and should distinguish semantic collocations from non-semantic collocations. The task is restricted to two-word Czech collocations from a supplied data set.

1.1 Semantic collocations

The term *collocation* denotes a meaningful and grammatical word combination that frequently or typically occurs in a natural language. *Semantic collocations* then refer to collocations such that their meaning is not compositional – cannot be derived as the "sum" of meanings of its parts. Semantic collocations, therefore, are themselves lexical units; that means, for instance, that they should be listed separately in a lexicon.

Consider the following expressions: "I think", "brown house", "brown envelope", "ground zero", "hot chocolate". The first two are simple collocations, the other three are semantic collocations, according to our definition.

From now on, we will use the terms *collocation* and *semantic collocation* interchangeably.

1.2 Motivation

Given the lexical independence of semantic collocations, SCR is one of the core tasks of lexicography and corpus linguistics. As semantic collocations are "hidden" items of the lexicon, any NLP task which takes lexical features into consideration benefits from an SCR system – which applies to more or less any NLP task nowadays. An SCR system may significantly improve natural language generation, word sense disambiguation, information retrieval, automatic summarization or machine translation, to name a few.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

2. Data

The Czech collocations data set (henceforth referred to simply as the data) contains 9232 entries. There are 97 columns: two for the lemmas of the collocation candidate, three columns of manual classification, the final classification column (generated from the manual classifications), which is our response variable, and 91 features.

The features are further described in Appendix A and B.

The data set was extracted from the Prague Dependency Treebank, version 2.0.

We have used several different data setups in our experiments (some arrangements had to be made for technical restrictions – allotted CPU time on distant machines, etc.); generally, however, the data split was 6232 training – 3000 test data for our final evaluation of the model (this split was chosen to get a slightly pessimistic reading). We performed cross-validation whenever possible (again, due to CPU time limitations, more extensive feature selection experiments had to do without cross-validation).

3. Methods used

We focused on three machine learning methods:

- k-Nearest Neighbor learning,
- Naïve Bayes classifiers,
- Adaptive Boosting.

All of those are *supervised* learning methods – based on a set of *training instances* (x_1, y_1), (x_2, y_2), … (x_m, y_m), where x_i is a *feature vector* from the set of $F_1 \times F_2 \times ... \times F_t = F$ (called the *feature space*) and y_i is the *classification* of the instance into one of the classes $C_1 \ldots C_k \in CL$ set of all classes, we want to find a classifying function h: $F_1 \times F_2 \times ... \times F_t \rightarrow CL$ (called a *hypothesis*) such that it best approximates the *true classification r*: $F_1 \times F_2 \times ... \times F_t \rightarrow CL$. Note that *best* is a vague term: it may signify accuracy as well as a host of other measures (most notably recall, precision, f-score or a weighted combination of those four).

A special and frequent case of classification is *binary classification*, when $|CL| = 2$. That amounts to classification being simply a yes-no questions, with *positive classification* $C_1 = 1$ and *negative classification* $C_2 = -1$.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

3.1 k-Nearest Neighbor learning

The k-Nearest Neighbor machine learning method, or kNN, is based on a very simple and intuitive notion: assuming we have a relevant set of features, points close to each other in the feature space tend to be classified similarly. Therefore, all we have to do is choose a metric and then decide the classification of a data point *x* by majority voting: assign *x* to the most frequent class among *k* nearest neighbors from the training data set.

There are two relevant parameters of a k-NN model: the choice of the metric and the choice of *k*. Since we are dealing with continuous numerical features, the natural choice of a metric is Euclidean distance. The choice of optimal *k* is more complicated and has to be determined empirically.

Less rudimentary version of the algorithm perform scaling (since features with greater standard deviation naturally have a greater weight in non-scaling kNN) and/or weigh the neighbors' votes in inverse proportion to their distance from the data point being classified.

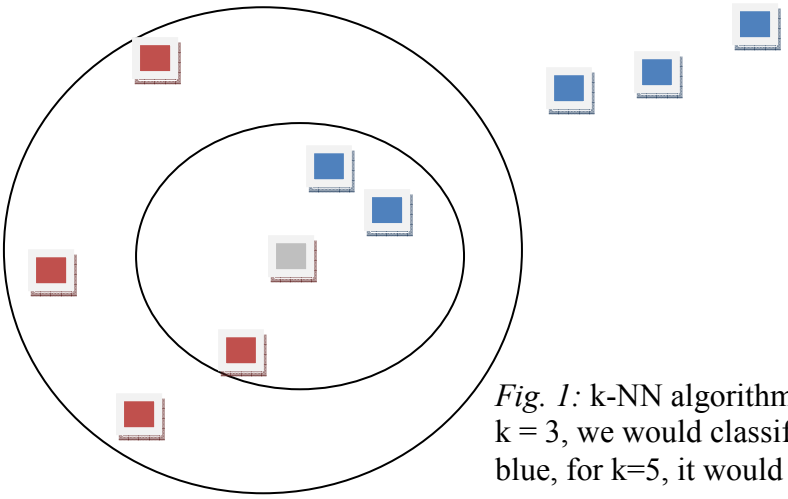


Fig. 1: k-NN algorithm. Choosing k: for k = 3, we would classify the data point as blue, for k=5, it would end up red.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

3.2 Naïve Bayes classifiers

Naïve Bayes classifiers are simple classifiers based on a direct application of Bayes's theorem. Suppose we want to classify an instance into a class from a given set of classes *CL* of classes C_1, C_2 … C_k. For each class *C*, we want to determine the probability $P(C|F_1,F_2 \ldots F_n)$ for F_1 ...F_n features and then choose $argmax_C(P(C|F_1,F_2 \ldots F_n))$. By Bayes's theorem, we have:

$$P(C \mid F_1, F_2 ... F_n) = \frac{P(F_1, F_2 ... F_n \mid C) P(C)}{P(F_1, F_2 ... F_n)}$$

If we had conditionally independent features, we could rewrite this as:

$$P(C \mid F_1, F_2 ... F_n) = \frac{P(F_1 \mid C) P(F_2 \mid C) ... P(F_n \mid C) P(C)}{P(F_1, F_2 ... F_n)}$$

These conditional and class apriori probabilities we can easily estimate using our training data (simply by computing frequency ratios).

We generally do not have independent features. But we when we assume we do have them and compute probabilities according to the second equation, Naïve Bayes classifiers actually perform surprisingly well.

Note that $P(F_1,F_2 \ldots F_n)$ stays the same throughout the data, so we may omit the term from our *argmax* search.

3.2 Decision Trees and AdaBoost.M1

As a third method, we utilized boosting with the Adaboost.M1 algorithm.

Let us pose the following question (as did Kearns[2]): can multiple weak learners be combined into a strong learner?

Boosting is a class of meta-algorithms that attempts to answer 'yes' to that question: they combine weak learners into a strong one. Typically, a boosting algorithm will be iteratively adding weak classifiers with respect to some distribution of weights over the training instances.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

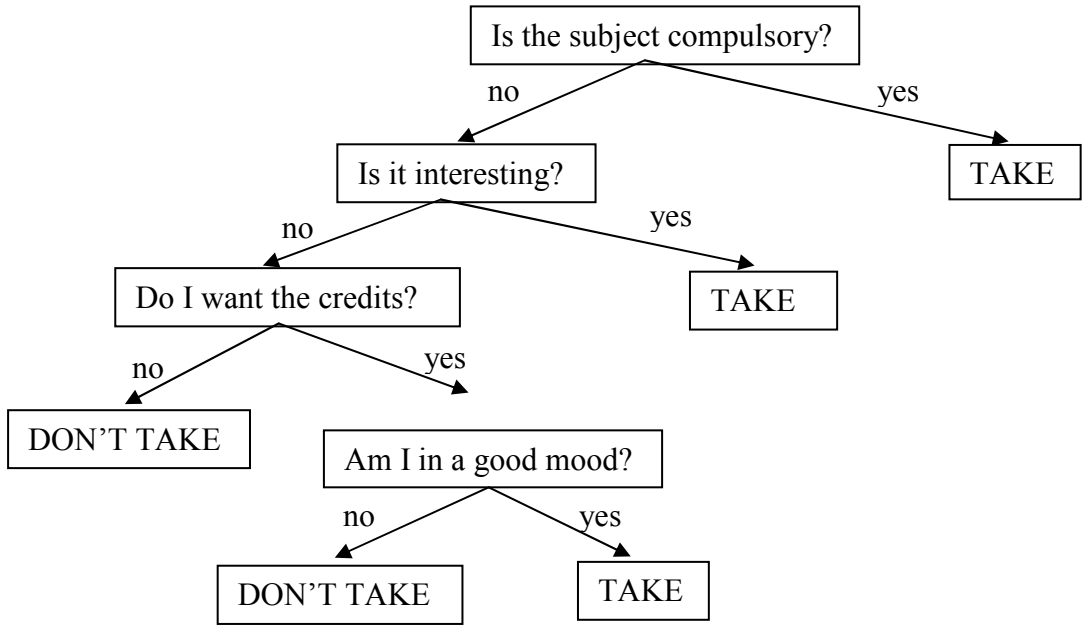
AdaBoost specifically is an adaptive boosting algorithm: it takes into account the performance of previously fitted weak learners when building a new one. (It was also the first boosting algorithm to do so.) In subsequent learners, it changes the distribution over training instances to favor those previously misclassified.

The adaboost.M1 implementation in the R system uses decision trees as the weak learners; we will briefly describe them.

3.2.1 Decision Trees

Intuitively, Decision Trees classify an instance by running it through a set of yes-no questions about its features. Suppose you are trying to decide whether to take a specific subject. In deciding, you ask yourself a set of questions to guide you through the decision process.

Each question is represented by a node of the decision tree, the final decisions are represented by its leaves. To each edge is mapped a yes/no answer to the parent node's question. Our subject-taking decision process can then represented by the following decision tree:



We can state the problem in the following way: each non-leaf node represents a split in the data, each leaf represents a subset of data that will be classified uniformly. We want to minimize classification errors, which means splitting the data into subsets that are classified as homogenously as possible.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

Growing a good decision tree resides with choosing the correct splits that will partition the data into the most homogenous subsets. We therefore need a splitting criteria, a way to evaluate each suggested split, and so compare them and choose the optimal one.

A problem with growing decision trees is that in the process of growing it, we do not know how it will eventually perform – we can only have local splitting criteria, we can only measure the immediate effects, i.e.: with the splits we now have, what is the next best split? Thus, we may opt for a split that (to borrow any drunk's excuse) seemed like a good idea at the time, only to run into dead ends later – and we may have missed a slightly worse split that, however, has relevant follow-up.

However, there is no way to work around this problem other than the ability to backtrack, which would, in turn, severely increase training times (the problem of growing an optimal decision tree is essentially NP-complete[1]) and, if we allowed locally suboptimal feature thresholds, dramatically inflate the search space. In practice, we make do with greedy local splitting criteria (and perhaps permit very limited backtracking).

What are these splitting criteria? We have already determined that the "goodness" of a partitioning depends on the homogeneity – or, purity – of the classes that correspond to the leaf nodes. A good split is one that results in purer classes than not splitting, and the larger the difference, the better the split.

For evaluating splits, there is a well-established set of formulae from which to choose. Of those, we picked for our model Gini index. Gini index defines the impurity of node *t* as:

$$i(t) = \sum_{j \neq i} p(j \mid t) p(i \mid t)$$

This essentially is the expected error rate: suppose the classification rule for a randomly selected instance from the class is "classify instance to class *i* with probability $P(i|t)$." Then, Gini index measures the probability that we were wrong: that the instance actually belongs to class *j*.

Other usual (im)purity measures are information gain (based on entropy, can be scaled for its bias towards features that partition the data into many subsets – which tends to generalize poorly, overfit and have problems with unseen data) and misclassification error.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

3.2.2 AdaBoost

As we've said, AdaBoost is the first adaptive boosting algorithm. It was presented by Freund and Schapire[3] in 1995 and proved a large success. The idea of adaptation is very intuitive: the subsequent weak learners should not repeat the mistakes of the learners we already have. Therefore, in the classifiers to come, we will make them focus on the harder-to-classify instances – those the previous classifiers have made errors on – by increasing their weights. The more problematic an instance is, the more weight it will be gaining throughout the boosting process until at some point, it is sure to draw enough attention of the weak classifiers (unless the algorithm stops – a fixed number of iterations may be selected, or an improvement threshold: when there isn't enough improvement between iterations for some time, the algorithm decides it will not get better and stop).

Recall the notation from the beginning of Section 3. The algorithm (for binary classification, which is what applies to our experimentation) is as follows:

1. Initialize a distribution D_1. $D_1(i) = \frac{1}{m}$ for $i = 1, \dots, m$
2. Let T be the number of iterations we want the algorithm to perform. For $t = 1 \dots T$:
    2.1 Find the classifier h_t: $F \rightarrow \{0,1\}$ that minimizes the error on the training instances over distribution D_i (proportion of misclassified instances, which are weighed individually by the distribution)
    2.2 If the error is greater than 0.5, stop.
    2.3 Choose a coefficient α_t (often: $\alpha_t = \frac{1}{2}\ln\frac{1-error}{error}$).
    (Note that always $error < 1$, so α_t is well-defined.)
    2.4 Update the distribution: $D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}$, where Z_t is a *normalization factor* chosen so that the updated D sums to 1.
3. Output the final classifier $H(x)$:

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

If we examine the equation in 2.4, we see that $-\alpha_t y_i h_t(x_i)$ evaluates as positive if and only if y_i and $h_i(x)$ do not match – i.e. x is misclassified by the weak learner h_i. That exactly corresponds to what we wanted – misclassified instances have their weight boosted, correctly classified ones on the other hand lose significance for upcoming classifiers.

AdaBoost can fit rather well complicated data that a single decision tree couldn't. However, its positive discrimination of notoriously hard instances also lead to great sensitivity to outliers and noisy data in general. Also, it – inevitably – takes asymptotically at least *T*–times as much time to complete as a single decision tree. While not prohibitive, this increase, in practice, can be rather significant.

4. Experimentation

We used several core data setups which we utilized in accordance with the specific experiment requirements. Essentially, this meant different data sizes and cross-validation parameters according to the computational demands of the experiment. The limitations on computation were sometimes rather severe: our personal computer couldn't handle the computations, so they had to be done in the MFF Rotunda lab by remote access with SSH – with a one-hour limit of CPU time imposed upon remotely activated processes.

We therefore utilized three different setups:

- *S1*. *"Light"*: Intended for exhaustive feature selection experiments. Uses only a third of the data (3077 randomly chosen instances), of those 2077 for training, 1000 for testing. No cross-validation was possible. Therefore, experiments done under the *S1* setup could not be relied on too, much *per se*; however, since their results were intended as heuristics for further feature selection, this did not matter too much. Typically, *S1* experiments exhaustively searched and evaluated feature pairs or triplets.
- *S2*. *"Heavy"*: Intended for experiments where we could afford to do more rigorous evaluation. Uses all the data in a 6232/3000 train/test split and performs three-fold cross-validation.
- *S3*. "*Flexible"*: Sometimes, neither *S1* or *S2* satisfied our requirements. Under *S2* fall all other setups (we will state them explicitly whenever needed).

The baseline accuracy (zero-classifier, everything classified as negative, non-collocations) on the complete data set is 0.789. (This translates to an accuracy of 0.895 removing roughly half of the errors we can expect.)

4.1 k-NN

We used k-NN mostly to get a feel for the feature space: how are the data distributed, what kind of results can we expect based on their structure. We experimented with *k* and feature selection. The data for k-NN experiments was scaled.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

The first step in feature selection was evaluating the performance of all feature pairs. This was done under the *S1* setup with $k = 5$, so we interpreted the result as a certain heuristic for further feature selection. (The best pairs scored around 0.85 on accuracy and 0.66 on f-score.) The average results were:

PAIRS	Avg.accuracy	Avg. recall	Avg. precision	Avg. f-score
All pairs	0.835	0.441	0.543	0.482
20 best pairs	0.846	0.635	0.669	0.651
Best pair	0.848	0.694	0.665	0.679

Next, we found the best features – "best" meaning "the average performance of feature sets it participated in was highest". We measured performance simply as the sum of the four elementary evaluation measures (accuracy, recall, precision, f-score). We then took the 20 best features, formed all best feature triplets and evaluated them (again under *S1*). The triplets yielded the following performance:

Pairwise 20-best feature triplets	Avg.accuracy	Avg. recall	Avg. precision	Avg. f-score
All triplets	0.835	0.548	0.624	0.583
20 best triplets	0.852	0.617	0.659	0.637
Best triplet	0.854	0.648	0.652	0.650

We can see that the performance of pairs and heuristically generated triplets is not exactly stellar; in fact, in f-score, the set of 20-best triplets performs worse than 20-best pairs. It would seem that the features truly behave rather unpredictably: while the average performance of all 20-best feature triplets is better – at least in f-score – than that of *all* feature pairs, 20-best triplets generated from the 20-best features from all pairs bring no improvement over the 20-best feature pairs. Also, the best pair of features performs better than the best triplet in f-score (not in accuracy) and its recall in on par with the best 20-random features model.

Note: the best-performing pair was Mutual Dependency (x5) with Cosine Similarity in *tf* vector space (x78) – a combination of an direct association and context measure.

The careful conclusion we draw from this is: iteratively increasing the feature set will probably not be successful, because for different feature set sizes, the combinations that fared well previously will not perform as well when "cross-bred" with each other.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

The logical follow-up question is: does the choice of features matter at all, or is the classifier performance only dependent on the *number* of features? We will seek to answer by choosing 100 random feature sets of different sizes and evaluating intra-set standard deviations, plus training a model on *all* the available features. This time, we can use the *S2* setup. We will also compare randomly chosen feature triplets with our heuristically chosen ones to see whether our heuristic makes sense.

100 feature sets of *N* random features	Avg.Acc.	Avg.Rec	Avg.Pr.	Avg.F.	SD.Acc	SD.Rec	SD.Pr	SD.F
$N = 3$	0.842	0.522	0.658	0.569	0.017	0.137	0.048	0.125
$N = 5$	0.852	0.580	0.673	0.620	0.009	0.066	0.021	0.045
$N = 20$	0.868	0.646	0.703	0.673	0.005	0.021	0.013	0.015
$N = 50$	0.873	0.680	0.710	0.694	0.003	0.012	0.009	0.014
All features	0.875	0.695	0.709	0.701	0.001	0.005	0.004	0.004

We can also see that the 50-random average and all-feature results are essentially the same, but since there is greater deviation for the 50-random averages, odds are there will be a set of 50 features which will perform *better* than the all-feature model. And, indeed, the best-performing set of 50 features does perform better.

We also found the best-performing sets for $N = 3$, 5 and 20 and compared the results:

Best feature set of *N* random features	Accuracy	Recall	Precision	F-score
$N = 3$	0.858	0.664	0.660	0.661
$N = 5$	0.871	0.625	0.723	0.670
$N = 20$	0.879	0.688	0.725	0.706
$N = 50$	0.880	0.703	0.722	0.712

So, random choice beats our heuristic in the best model department, too.

Our best 50-feature model thus effectively eliminates nearly half the errors and can achieve an f-score over 0.71. Note that the top precision effectively does not change between 5 and 50 features, what changes is recall: increasing the dimension helps to "round up" more instances we may have previously missed.

As the last experiment for k-NN, we used (with the *S2* setup) the set of 14 features found by Pavel Pecina, Ph.D. in his dissertation[4] to generally provide the most relevant information about the data set for classification models:

x6 – Log frequency biased MD,
x23 – First Kulczynski,
x39 – Unigram subtuple measure,
x41 – S cost,
x57 – Left context phrasal entropy,
x58 – Right context phrasal entropy,
x59 – Left divergence,
x62 – Reverse cross entropy,
x68 – Reverse confusion probability,
x75 – Phrase work coocurrence,
x77 – Cosine context similarity in bl vector space,
x81 – Dice context similarity in *tf* v.s.,
x82 – Dice context similarity in *tf.idf* v.s.

(We will refer to this feature set as *Pecina's set* further in our work) The k-NN classifier then gave us the following result:

	Accuracy	Recall	Precision	F-score
Pecina feature set	0.868	0.608	0.726	0.662

Conclusions drawn from the experiments:

The heuristic based on pairwise feature performance does not guarantee improvement in iterative dimension-increase feature selection with k-NN, or not directly.

The structure of our (scaled) data is such that up to 50, higher dimensionality means better-defined neighborhoods. However, we haven't conducted experiments to determine whether a small set of features is actually responsible and the reason why a 50-random choice erforms better is simply because members of this core feature set occur more often in sets of 50 features than in 20. We have a certain clue that it may only partially be so: at least in the very first step, using our dimensionality-increase iterative feature selection with the additive feature performance heuristic does *not* work (using an assumed core feature set only for generating the higher-dimensional samples did not present an improvement). Also, this dimensionality result tells us that we don't have to *fear* adding dimensions at least up to 50: chances are we will not introduce too much noise.

With these results in mind, we move towards

Jan Hajič
hajicj@gmail.com
17. 2. 2011

4.2 Naïve Bayes

The most significant difference between the data sets used for k-NN models and Naïve Bayes classifiers is that in Naïve Bayes, we can use the non-numerical features, most importantly morphological tags (t1 and t2). Also, scaling is not relevant for Naïve Bayes. However, we expect it to be much more sensitive to the actual feature set used, rather than the number of features. That is due to the independence assumption for Naïve Bayes: choosing features which are more correlated generally results in significantly poorer performance.

To compare the properties of our feature-pair heuristic under k-NN and Naïve Bayes, we have also obtained classification results for all feature pairs:

PAIRS	Avg.accuracy	Avg. recall	Avg. precision	Avg. f-score
All pairs	0.728	0.448	0.589	0.412
20 best pairs	0.877	0.733	0.673	0.700
Best pair	0.891	0.798	0.679	0.734
Best pair 3x*S2*	0.871	0.798	0.663	0.724

The best pair is consists of – the morphological tag of the first word (t1) and Mutual dependency (x5). That sounds somewhat familiar. Also, the best pair performed so well we decided to test it under *S2* multiple times (to obtain 9-fold cross-validation) as well, to get a more reliable reading. The results are not quite as cheerful, but not *too* much worse for the heuristic to be a total scam, at least on the higher end.

Given the properties arising from the independence assumption (and seeing them confirmed by the vastly different average performances vs. the 20-best performances of feature pairs) of Naïve Bayes, we did not bother doing experiments with randomly chosen feature sets. Instead, we used our manually selected set from December and the principles we used, together with the heuristic on all pairs and Pecina's results[4], to get the following results under *S2*:

	Accuracy	Recall	Precision	F-score
Best pair 3x*S2* (from previous table)	0.871	0.798	0.663	0.724
kNN best pair (x5, x78)	0.849	0.558	0.676	0.610
Best pair + kNN best pair (t1, x5, x78)	0.873	0.794	0.671	0.727
Pecina's set	0.789	0.886	0.501	0.640
Pecina's set + t1, t2 (further called P+POS)	0.821	0.907	0.546	0.681
December set, no POS (x5,x6,x73,x82)	0.854	0.678	0.648	0.662
December set (t1,t2,x5,x6,x73,x82)	0.870	0.788	0.663	0.720
December set without x6	0.877	0.751	0.690	0.719

etc.

We can see that using POS tags without doubt improves performance, so the ability to use these features *does* prove a significant advantage over the *k*-NN method.

Going back to our assumption about the futility of testing random feature sets and a feature choice pool-limiting heuristic, we've attempted to generate random 5-feature sets from the features in P+POS. The results were less than impressive:

P+POS 5-random	Avg.accuracy	Avg. recall	Avg. precision	Avg. f-score
All sets	0.765	0.727	0.529	0.566
20 best sets	0.825	0.773	0.573	0.651
Best set	0.835	0.783	0.583	0.668

It generally seems we could tweak our feature set to suit either recall or precision with no problem (adding some features aids preicision, some aid recall); using P+POS or the December set as a certain core, we can still be reasonably sure we will get solid f-score as well.

Feature selection with Naïve Bayes is rather alchemic in nature. Methods to make it less of an obscure science may include building a correlation matrix between features to favor the uncorrelated.

If an evolutionary approach is to be successful with Naïve Bayes feature selection, it will need to keep a very strong exploration parameter.

It is time to bring out the heavy weapons and use:

4.3 Adaboost

Since the Adaboost algorithm is slow compared to the others we used, we could not afford extensive feature selection or parameter tuning and were confined to making good guesses, possibly based on previous experience with feature sets from using other methods. However, the boosting algorithm is so powerful it made do with our guesswork to outperform *k*-NN and Naïve Bayes in both accuracy and f-score.

Again, as Adaboost uses decision trees, we are able to use POS tags as features. Furthermore, these features may just strike the balance between providing well-defined distinct classes and branching the tree out too much, which would cause overfitting and inability to process unseen data.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

We have obtained the following results (10-fold cross-validation, 6232-3000 train-test data split, confidence interval width in all cases under 0.001, i.e. a difference of as little as 0.002 is statistically significant):

	Accuracy	Recall	Precision	F-score
Naïve Bayes best pair (t1, x5)	0.882	0.655	0.750	0.700
kNN best pair (x5, x78)	0.855	0.573	0.688	0.625
Best pair + kNN best pair (t1, x5, x78)	0.884	0.654	0.761	0.703
Pecina's set	0.874	0.617	0.737	0.671
P+POS	0.891	0.692	0.772	0.729
P+POS+x , x , x , (randomly picked)	0.889	0.705	0.761	0.732
December set (t1,t2,x5,x6,x73,x82)	0.886	0.665	0.765	0.711
December set without x6	0.870	0.615	0.723	0.664
December set plus x77	0.884	0.655	0.763	0.705
All features	0.902	0.723	0.793	0.756

The conclusion is clear: the power of boosting wins and given pruning (even just with the default value of the complexity parameter at cp = 0.01), it can filter out irrelevant features well enough so that using *all* the features gives us the best results in all departments. (We suppose that parameter tuning may significantly improve its performance. See future work in the Conclusion section.)

5. Conclusions

As far as feature selection is concerned, we have concluded that making educated guesses is a fast way of getting performance that may be suboptimal, but still is better than anything we could come up with by a more rigorous method now. That is both a tribute to intuition and, more importantly, a call to broaden our horizons in the machine learning department.

More importantly: choosing a good model matters much more than selecting good features. We can't make far-reaching conclusions about parameter tuning because we, well, haven't tuned any.

Future work definitely includes more investigating in the way model parameters: how they affect classifier performance and what is their relationship with feature selection (i.e.: do certain methods of feature selection prefer certain parameter settings and vice versa?). We have merely scratched the surface of the challenges our task poses. Since our Bachelor thesis is essentially a machine learning experiment, we are greatly interested in improving our skills in the field – fast.

Jan Hajič
hajicj@gmail.com
17. 2. 2011

References:

[1] Hyafil, Laurent; Rivest, RL (1976). "Constructing Optimal Binary Decision Trees is NP-complete". Information Processing Letters 5 (1): 15–17.

[2] Michael Kearns. Thoughts on hypothesis boosting. Unpublished manuscript. 1988

[3] Yoav Freund, Robert E. Schapire. "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting"

[4] Pecina, Pavel: “Lexical Association Measures: Collocation Extraction”, ÚFAL, 2009