Towards Using Reranking in Hierarchical Classification

Qi Ju, Richard Johansson, and Alessandro Moschitti

DISI, University of Trento, Italy
{qi,johansson,moschitti}@disi.unitn.it

Abstract. We consider the use of *reranking* as a way to relax typical independence assumptions often made in hierarchical multilabel classification. Our reranker is based on (i) an algorithm that generates promising k-best classification hypotheses from the output of local binary classifiers that classify nodes of a target tree-shaped hierarchy; and (ii) a tree kernel-based reranker applied to the classification tree associated with the hypotheses above. We carried out a number of experiments with this model on the Reuters corpus: we firstly show the potential of our algorithm by computing the oracle classification accuracy. This demonstrates that there is a significant room for potential improvement of the hierarchical classifier. Then, we measured the accuracy achieved by the reranker, which shows a significant performance improvement over the baseline.

Keywords: hierarchical classification, kernel methods, reranking

1 Introduction

Hierarchical multilabel classifiers often impose a number of simplifying restrictions on their models. In particular, category assignments are normally assumed to be conditionally independent: The probability of a document D belonging to a subcategory C_i of a category C is assumed to depend only on D and C, but not on other subcategories of C, or any other categories in the hierarchy. This independence assumptions clearly does not hold, since categories may well be subject to relationships that are not simply explained by the hierarchy. However, the introduction of these long-range dependencies will lead to computational intractability, since simple maximization algorithms based on divide-and-conquer strategies are no longer applicable.

In this paper, we propose to use *reranking* as a way to handle the computational issues. We first use a conventional hierarchical classifier to generate a hypothesis set of limited size, and then apply a more complex model – which can be more liberal in its use of statistical dependencies – to pick the final output.

Such a model is a reranker based on a classifier taking pairs of hypotheses as its input. These are represented by means of trees, whose nodes are the categories and whose edges connect fathers with children of the hierarchy. The model is learned using Support Vector Machines and tree kernels. To prepare the ground for the use of reranking, we also present an algorithm to generate the top k category assignments from a large-scale hierarchical classifier; it is clear that this can be useful also for other purposes than reranking.

We carried out experiments on the well-known Reuters Volume 1 collection. First, we evaluated the oracle performance, which shows high potential for improvement (i.e. 8 points in Microaverage and 15 in Macroaverage). Then, we tested the impact of reranker, which shows significant improvement. This is higher for rare categories, which are typically associated with lower basic accuracy.

Although, we focus on a small hierarchy the approach is easily extendable to larger structures, also considering that we do not need to encode the entire hierarchy in a tree since the very long-distant nodes intuitively can be assumed independent.

In the reminder, Section 2 introduces preliminaries for the hypothesis generation algorithm, which is then presented in Section 3. Section 4 illustrates our reranking approach based on tree kernels, Section 5 reports our experiments, and finally Section 6 derives the conclusions.

2 Preliminaries

We address the problem of *hierarchical classification*, which we define as the task of assigning an object – henceforth referred to as a *document* – to one or more hierarchically organized *categories*: If it belongs to a category C, then it also implicitly belongs to all supercategories of C, including the top category T consisting of all documents. In this work we consider tree-shaped hierarchies; we leave the extension to general DAG-shaped category systems to future work.

We base our model on the computation of two types of probabilities. First, for a given document D, and a category C with subcategories $C_1, \ldots, C_n$, we define the *stop probability* as the probability of "stopping" at C, i.e. that D does not belong to any of the subcategories of C:

$$p_0(C) = P(D \notin C_1 \wedge \ldots \wedge D \notin C_n | D \in C)$$

Secondly, in the case where we know that at least one subcategory has been selected, we can compute the probabilities of selecting a particular subcategory:

$$p_{C_i}(C) = P(D \in C_i | D \in C \wedge (D \in C_1 \vee \ldots \vee D \in C_n)), i \in \{1, \ldots, n\}$$

At this stage, we assume conditional independence between the subcategories, so the probability will depend only on the document and the supercategory.

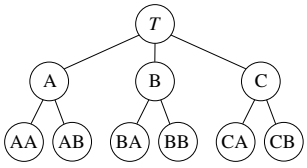

Fig. 1. Example of a hierarchy.

These probabilities can be used to compute the probability of a complete assignment of categories to a document. To exemplify, consider the hierarchy in Figure 1. To compute the probability of a document D belonging to the categories AB and C (and then also implicitly to T and A) but not to AA, B, CA, or CB, we decompose the probability using the above-mentioned conditional probabilities:

$$(1 - p_0(T)) \cdot p_A(T) \cdot (1 - p_B(T)) \cdot p_C(T) \cdot (1 - p_0(A)) \cdot (1 - p_{AA}(A)) \cdot p_{AB}(A) \cdot p_0(C)$$

3 An Algorithm to Generate the Top k Hypotheses

The number of category assignments is exponential in the number of categories, so for any nontrivial hierarchy a brute-force search to find the best hypothesis

Algorithm 1 Generation of the top hypothesis.

```
function TOP1(C)
  // Returns the top hypothesis
  // and its probability
  if p_0(C) > 0.5
    return ⟨{C}, p_0(C)⟩
  ⟨S, P⟩ ← MAXSUBCATS(C)
  if S = ∅
    ⟨S, P⟩ ← MAXONESUBCAT(C, P)
  if p_0(C) > P
    return ⟨{C}, p_0(C)⟩
  else
    return ⟨{C} ∪ S, P⟩

function MAXONESUBCAT(C, P)
  q_min ← ∞
  for each subcategory C_i ⊂ C
    ⟨S_i, P_i⟩ ← TOP1(C_i)
    q_i ← (1 − p_{C_i}(C))/(P_i · p_{C_i}(C))
    if q_i < q_min
      q_min ← q_i, S_min ← S_i
  return ⟨S_min, P/q_min⟩

function MAXSUBCATS(C)
  S ← ∅, P ← 1 − p_0(C)
  for each subcategory C_i ⊂ C
    if p_{C_i}(C) > 0.5
      ⟨S_i, P_i⟩ ← TOP1(C_i)
      if p_{C_i}(C) · P_i > (1 − p_{C_i}(C))
        P ← P · p_{C_i}(C) · P_i
        S ← S ∪ S_i
      else
        P ← P · (1 − p_{C_i}(C))
    else
      P ← P · (1 − p_{C_i}(C))
  return ⟨S, P⟩
```

is not applicable. However, the independence assumptions ensure that the search space is decomposable so that the best assignment – and the k best assignments – can be found quickly. Similar to the fastest k-best algorithm for natural language parsing presented in [12], our algorithm proceeds in two steps: first we find the best assignment, and then we construct the k-best list by incremental modifications.

The algorithm exploits the fact that the scores are probabilities in order to prune the search space slightly: If we see that the stop probability p_0 is greater than 0.5, we do not need to compute the probability of entering any subcategory since $(1-p_0)\cdot p_{C_i}$ is then guaranteed to be less than 0.5. If we rewrite the algorithm without this trick, it can easily be generalized to the situation where the scores are not probabilistic.

3.1 Generation of the Top Hypothesis

We first describe the function TOP1 that finds the category assignment having the highest probability; note that this is not necessarily what we would get by a greedy algorithm selecting the highest probability assignment at each choice point. Algorithm 1 shows the pseudocode. The algorithm is fairly straightforward; the only complication is that we need to ensure that at least one subcategory $C_i \subset C$ is enabled if we do not stop at a category C. In practice, the implementation will cache the probabilities and maximal assignments to avoid redundant recomputations. For brevity, we omit these details from the pseudocode.

3.2 Expansion of Hypotheses

The algorithm TOPK to generate the k top hypotheses (Algorithm 2) relies on the fact that the conditional independence assumptions we have made ensure that the search space is monotonic. The hypothesis at position i in the list of hypotheses is then a one-step modification of one of the first $i-1$ hypotheses. To generate k

Algorithm 2 Generation of the top k hypotheses.

```
function TopK(C, k)
  // Returns the top k hypotheses
  // and their probabilities
  H ← ∅
  q ← empty priority queue
  Enqueue(q, Top1(C))
  while |H| < k and q is nonempty
    repeat
      ⟨S, P⟩ ← Dequeue(q)
    until ⟨S, P⟩ ∉ H
    H ← H ∪ {⟨S, P⟩}
    if |H| < k
      for each h ∈ Succs(C, P, S)
        Enqueue(q, h)
  return H

function Succs(C, P, S)
  // Returns the set of modifications
  // of the hypothesis S
  if C has no subcategory
    return ∅
  H ← ∅
  if S ≠ {C}
    Stop(C, P, S, H)
    EnableEachSubcat(C, P, S, H)
    DisableEachSubcat(C, P, S, H)
    SubcatSuccs(C, P, S, H)
  else
    Unstop(C, P, S, H)
  return H
```

hypotheses, we thus start with the most probable one and put it into a priority queue ordered by probability. Until we have found k hypotheses, we pop the front item and put it into the output list. We then apply the function Succs to find all one-step modifications of the item, and we add them all back to the queue.

The Succs function applies the following one-step modification operations: Stop, which changes an assignment with subcategories to a stop; EnableEachSubcat, which enables every disabled subcategory; DisableEachSubcat, which disables every enabled subcategory if there are more than one; Unstop, which enables at least one subcategory of an assignment without subcategories; and finally SubcatSuccs, which recursively computes a one-step modification of every enabled subcategory. Note that we only need to carry out the modifications that reduce the probability. The pseudocode for the modification operations is shown in Algorithm 3. The pseudocode uses two auxiliary functions: Subtree(C), which returns the set of categories that are subcategories of C, and ProbSubcats, which returns the (previously computed) probability of an assignment of a set of subcategories.

3.3 Efficiency of the Hypothesis Set Generation Algorithm

The complexity of the algorithm is $O(ks \log(ks))$ where s is the maximal number of modified items generated by the Succs function, since the complexity of the Enqueue operation is logarithmic in a standard priority queue. A non-tight upper bound on s is $2N$, where N is the number of nodes in the hierarchy, but this is of limited interest: In practice, the number of modified items will be much smaller, and depend on parameters such as the shape of the hierarchy and the number of enabled subcategories in an assignment. However, it is clear that the algorithm is able to handle very large hierarchies even in the worst case.

The bottleneck in practice will typically be the call to the probability estimation procedure, and we note that the worst case – for 1-best as well as k-best generation – occurs when we have to estimate all probabilities in the hierarchy. The number of estimations in a hierarchy of N nodes is at most $N - 1$ stop probabilities and $N - 1$ subcategory probabilities; note that these two worst-case numbers do not occur at the same time. However, since we generate the probabilities only when we need them, the number of estimations will typically be much smaller in practice.

Algorithm 3 Functions that generate one-step modifications of a hypothesis.

function STOP(C, P, S, H)
  $P' \leftarrow P \cdot p_0(C)/(1 - p_0(C))$
  for each subcategory $C_i \subset C$
    if $C_i \in S$
      $P' \leftarrow P'/p_{C_i}(C)$
      $P' \leftarrow P'/$PROBSUBCATS(S, C_i)
    else
      $P' \leftarrow P'/(1 - p_{C_i}(C))$
  if $P' < P$
    $H \leftarrow H \cup \{\langle\{C\}, P'\rangle\}$

function UNSTOP(C, P, S, H)
  $\langle S_s, P_s\rangle \leftarrow$ MAXSUBCATS(C)
  if $S_s = \emptyset$
    $\langle S_s, P_s\rangle \leftarrow$ MAXONESUBCAT(C, P)
  $P' \leftarrow P \cdot (1 - p_0(C)) \cdot P_s/p_0(C)$
  if $P' < P$
    $H \leftarrow H \cup \{\langle S \cup S_s, P'\rangle\}$

function ENABLEEACHSUBCAT(C, P, S, H)
  for each subcategory $C_i \subset C$
    if $C_i \notin S$
      $\langle S_i, P_i\rangle \leftarrow$ TOP1(C_i)
      $P' \leftarrow P \cdot p_{C_i}(C) \cdot P_i/(1 - p_{C_i}(C))$
      if $P' < P$
        $H \leftarrow H \cup \{\langle S \cup S_i, P'\rangle\}$

function DISABLEEACHSUBCAT(C, P, S, H)
  for each subcategory $C_i \subset C$
    if $C_i \in S$
      $P' \leftarrow P \cdot (1 - p_{C_i}(C))$
      $P' \leftarrow P'/p_{C_i}(C)/$PROBSUBCATS$(S, C_i)$
      $S' \leftarrow S \setminus$ SUBTREE(C_i)
      if $P' < P$ and $S' \neq \{C\}$
        $H \leftarrow H \cup \{\langle S', P'\rangle\}$

function SUBCATSUCCS(C, P, S, H)
  for each subcategory $C_i \subset C$
    if $C_i \in S$
      $P_i \leftarrow$ PROBSUBCATS(S, C_i)
      $S_i \leftarrow S \cap$ SUBTREE(C_i)
      for each $\langle S_s, P_s\rangle \in$ SUCCS(C_i, P_i, S_i)
        $H \leftarrow H \cup \{\langle (S \setminus S_i) \cup S_s, P/P_i \cdot P_s\rangle\}$

How much of the hierarchy we actually need to explore will of course depend on the particular probabilities.

3.4 Encoding Hypotheses in a Tree Structure

Once hypotheses are generated, we need a representation from which dependencies between the different nodes of the hierarchy can be learned. Since we do not know in advance which can be the important dependencies and not even the scope of the interaction between the different structure subparts, we rely on automatic feature engineering via structural kernels. For this paper, we consider tree-shaped hierarchies so that tree kernels, e.g. [17], can be applied.

More in detail, in this paper, we focus on the subhierarchy of Reuters in Figure 2 regarding Markets (MCAT) and its subcategories: Equity Markets (M11), Bond Markets (M12), Money Markets (M13) and Commodity Markets (M14). These also have subcategories: Interbank Markets (M131), Forex Markets (M132), Soft Commodities (M141), Metals Trading (M142) and Energy Markets (M143).

As the input of our reranker, we can simply use a tree representing the hierarchy above, marking the assignments of the current hypothesis in the node labels, e.g. -M143 means that the document was not classified as Energy Markets. For example, Figure 3 shows the representation of a classification hypothesis, whose only assigned category is M132.

Note that such tree substructures can capture dependencies between the different categories.

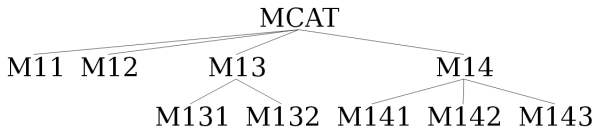

Fig. 2. A subhierarchy of Reuters.

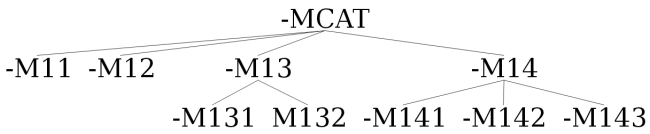

Fig. 3. A tree representing a category assignment hypothesis.

4 A Kernel-based Reranker for Hierarchical Classification

The vast majority of tasks in natural language processing involve the processing of *structured objects*. Building classifiers for these objects is traditionally carried out by implementing rule-based extractors of features. However, the complexity of the structure prevents an exhaustive approach to feature generation since the use of all possible substructures produces an exponential number of features, and consequently the development of such systems is typically guided by heuristics rather than a systematic approach. For instance, [5] commented on the development of features for a parse tree reranker: "It is worth noting that developing feature schemata is much more an art than a science."

As a way to avoid the feature selection problem, learning methods that work directly with objects instead of feature vectors have been proposed. The generalization from linear classifiers (that apply to vectors) to *kernel-based classifiers* (that apply to objects) is straightforward. To derive the kernel-based decision function, we start from the decision function of a linear classifier:

$$f(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x_i} \cdot \boldsymbol{x} + b \tag{1}$$

where $\boldsymbol{x}$ is a classifying example and $\boldsymbol{w}$ and b are the separating hyperplane's *gradient* and its *bias*, respectively. The gradient is a linear combination of the training points $\boldsymbol{x_i}$, their labels y_i and their weights α_i. Applying the so-called *kernel trick* it is possible to replace the scalar product with a *kernel function* defined over pairs of *objects*:

$$f(o) = \sum_{i=1}^{n} \alpha_i y_i \mathrm{k}(o_i, o) + b$$

with the advantage that we do not need to provide an explicit mapping $\phi(\cdot)$ of our examples in a vector space; instead, the scalar product can be computed implicitly, which may be much more efficient. It is also easy to show that for kernels k_1 and k_2, we may form new kernels $k_1 + k_2$ and $k_1 \cdot k_2$, allowing for a modular decomposition. Kernel functions have proven very effective for natural language applications as suggested by the large body of related work, e.g. [6, 15, 8, 4, 7, 9, 26, 16, 25, 18, 10].

4.1 Tree Kernels

In the case where the objects we want to classify are trees, there exist efficient algorithms based on dynamic programming that compute kernel functions based on

counting the shared substructures of the trees: *tree kernels*. These computations are efficient since they do not have to enumerate the whole fragment space explicitly.

Let $\mathcal{F} = \{f_1, f_2, \ldots, f_{|\mathcal{F}|}\}$ be the set of tree fragments and $\chi_i(n)$ an indicator function equal to 1 if the target f_i is rooted at node n and equal to 0 otherwise. A tree kernel function over T_1 and T_2 is defined as

$$TK(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2),$$

where N_{T_1} and N_{T_2} are the sets of nodes in T_1 and T_2, respectively, and $\Delta(n_1, n_2) = \sum_{i=1}^{|\mathcal{F}|} \chi_i(n_1)\chi_i(n_2)$.

The Δ function is equal to the number of common fragments rooted in nodes n_1 and n_2 and thus depends on the fragment type. Below, we report the algorithm to compute Δ for the partial tree fragments (PTFs) [17].

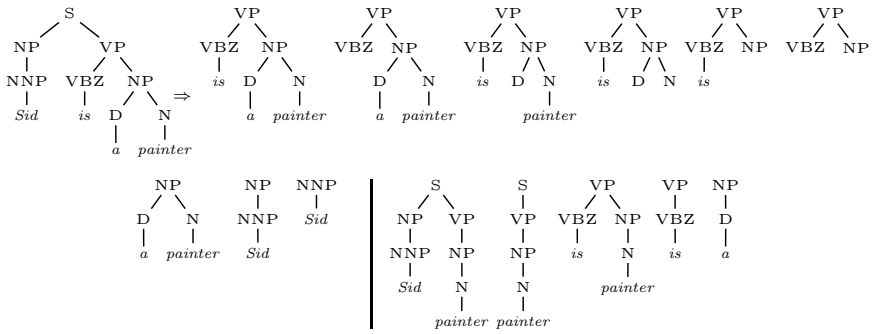

Fig. 4. A tree for the sentence "Sid is a painter" with some of its syntactic tree fragments and specific partial tree fragments (PTFs), before and after the vertical line, respectively.

Partial Tree Kernel (PTK) The Δ function for PTK is the following. Given two nodes n_1 and n_2, a tree kernel [6] is applied to all possible child subsequences of the two nodes, i.e. a String Kernel is applied to enumerate their substrings and the tree kernel is applied on each of such child substrings. More formally:

1. if the node labels of n_1 and n_2 are different then $\Delta(n_1, n_2) = 0$;
2. else $\Delta(n_1, n_2) =$

$$= 1 + \sum_{\boldsymbol{I}_1, \boldsymbol{I}_2, l(\boldsymbol{I}_1) = l(\boldsymbol{I}_2)} \prod_{j=1}^{l(\boldsymbol{I}_1)} \Delta(c_{n_1}(\boldsymbol{I}_{1j}), c_{n_2}(\boldsymbol{I}_{2j}))$$

where $\boldsymbol{I}_1 = \langle h_1, h_2, h_3, .. \rangle$ and $\boldsymbol{I}_2 = \langle k_1, k_2, k_3, .. \rangle$ are index sequences associated with the ordered child sequences c_{n_1} of n_1 and c_{n_2} of n_2, respectively, $\boldsymbol{I}_{1j}$ and $\boldsymbol{I}_{2j}$ point to the j-th child in the corresponding sequence, and again, $l(\cdot)$ returns the sequence length, i.e. the number of children. Furthermore, we add two decay factors: μ for the depth of the tree and λ for the length of the child subsequences with respect to the original sequence, i.e. we account for gaps. It follows that $\Delta(n_1, n_2) =$

$$= \mu\Big(\lambda^2 + \sum_{\boldsymbol{I}_1, \boldsymbol{I}_2, l(\boldsymbol{I}_1) = l(\boldsymbol{I}_2)} \lambda^{d(\boldsymbol{I}_1) + d(\boldsymbol{I}_2)} \prod_{j=1}^{l(\boldsymbol{I}_1)} \Delta(c_{n_1}(\boldsymbol{I}_{1j}), c_{n_2}(\boldsymbol{I}_{2j}))\Big),$$

where $d(\boldsymbol{I}_1) = \boldsymbol{I}_{1l(\boldsymbol{I}_1)} - \boldsymbol{I}_{11}$ and $d(\boldsymbol{I}_2) = \boldsymbol{I}_{2l(\boldsymbol{I}_2)} - \boldsymbol{I}_{21}$.

This way, we penalize both larger trees and child subsequences with gaps. An efficient algorithm for the computation of PTK is given in [17].

Category Name	Train (Train1 ∪ Train2)	Train1	Train2	TEST
MCAT	24	8	16	23
M11	346	191	155	327
M12	202	97	105	184
M13	10	5	5	16
M131	303	133	170	220
M132	187	96	91	175
M14	53	26	27	34
M141	378	183	195	410
M142	85	47	38	78
M143	172	101	71	148
Total	1760	887	873	1595

Table 1. Instance distributions on Reuters subhierarchy $\mathcal{S}$.

4.2 Tree Kernels-based Reranker

The reranking machine learning problem consists of learning to select the best candidate from a given candidate set. In order to be able to apply machine learning methods for binary classifiers such as support vector learning, we applied the reduction known as the Preference Kernel method [24]. The development of reduction methods from ranking tasks to binary classification is an active research area; see for instance [2] and [1].

In the Preference Kernel approach, the reranking problem – learning to pick the correct candidate h_1 from a candidate set $\{h_1, \ldots, h_k\}$ – is reduced to a binary classification problem by creating *pairs*: positive training instances $\langle h_1, h_2\rangle, \ldots, \langle h_1, h_k\rangle$ and negative instances $\langle h_2, h_1\rangle, \ldots, \langle h_k, h_1\rangle$. This training set can then be used to train a binary classifier. At classification time, pairs are not formed (since the correct candidate is not known); instead, the standard one-versus-all binarization method is still applied.

The kernels are then engineered to implicitly represent the *differences* between the objects in the pairs. If we have a valid kernel K over the candidate space T, we can construct a function D_K over the space of pairs $T \times T$ as follows:

$$\begin{aligned} D_K(x, y) &= D_K(\langle x_1, x_2\rangle, \langle y_1, y_2\rangle) \\ &= K(x_1, y_1) + K(x_2, y_2) \\ &\quad - K(x_1, y_2) - K(x_2, y_1). \end{aligned}$$

It is easy to show [24] that D_K is also a valid Mercer kernel. This makes it possible to use kernel methods to train the reranker.

We trained the rerankers using SVM-light-TK[1], a tree-kernel-enabled version of SVM-light [13].

5 Evaluations

The aim of our evaluation is to demonstrate that our reranking approach can introduce dependencies in the hierarchical classification model, which improve accuracy. For this purpose, we first show that the hypotheses generated by our algorithms allow for improving the classification task, i.e. we compute the so-called oracle accuracy. Then we carried out experiments on reranking the best hypothesis by measuring the impact of the automatic reordering on the classification performance.

[1] `http://disi.unitn.it/moschitti/Tree-Kernel.htm`

	Reranked				Oracle			
K	Prec.	Rec.	Micro-F1	Macro-F1	Prec.	Rec.	Micro-F1	Macro-F1
1	0.9460	0.8712	0.9070	0.7597	0.9459	0.8712	0.9070	0.7597
2	0.9436	0.8955	0.9189	0.7870	0.9462	0.9521	0.9491	0.8372
4	0.9350	0.9036	0.9190	0.7891	0.9579	0.9764	0.9670	0.9029
8	0.9368	0.9042	0.9202	0.7910	0.9642	0.9882	0.9760	0.9429
16	0.9414	0.9004	0.9205	0.7914	0.9733	0.9963	0.9846	0.9778

Table 2. Global performance of the reranker (on the left) together with the best achievable accuracy on k hypotheses or oracle performance (on the right).

5.1 Setup

We used the subhierarchy $\mathcal{S}$ introduced in Section 3.4, which is part of the overall corpus of Reuters Volume1 (http://trec.nist.gov/data/reuters/reuters.html). We divided the documents of $\mathcal{S}$ in three chunks of data: Train1, Train2 and test set (Test). The multiclass classifiers (MCC) are trained on Train1 and tested on Train2 (and vice versa) to generate the hypotheses and thus the training data for the reranker. The test set is used to measure the accuracy of the final model. The distribution of the data instances through the different categories can be observed in Table 1.

The hypotheses are represented with trees like the one in Figure 3 and are processed by SVMs using PTK (see Section 4.1). To the latter a simple linear kernel applied to the hypothesis probability (i.e. a vector with only one feature) is added. This allows the reranker to exploit the contribution of the bag-of-words used for the basic classifiers.

5.2 Experiments on Reranking

To derive the oracle performance of reranking, i.e. the accuracy of MCC by always selecting the best hypothesis (according to the gold standard classification) out of k, MCC is trained on Train1 $\cup$ Train2 and tested on the test set. The single binary classification models are used to generate the hypotheses as explained in Section 3. Table 2 shows the Precision, Recall, Microaverage F1 and Macroaverage F1 according to different numbers of hypotheses. The latter are reranked by our tree kernel model. We note that:

(i) the accuracy of the baseline MCC is 0.9070 (i.e. for k=1) whereas the best result, 0.9205 is achieved for $k = 16$, for an absolute improvement of 1.35 percent points in Microaverage.
(ii) The reranker can better improve small categories for which the small availability of training data causes lower accuracy of the basic MCC. This explains the larger improvement in Macroaverage, i.e. 3.3 = 87.73 - 84.43.
(iii) There is still large possibility to improve the above outcome as the oracle performance shows that about 8 and 13 points can be potentially gained in Micro/Macro F1, respectively.

To support point (ii), we report the results of the individual binary classifiers in Table 3. We can see that small categories like for example MCAT, are associated with a very large F1 improvement when using the oracle information. This also results in a large improvement of the reranker.

		k=1		k=2		k=4		k=8		k=16	
		Reranked	Oracle	RR	Oracle	RR	Oacle	RR	Oracle	RR	Oracle
MCAT	Precision	0.8750	0.8750	0.7333	0.7778	0.7333	0.8400	07333	0.8519	0.7333	0.8519
	Recall	0.3044	0.3044	0.4783	0.6087	0.4783	0.9130	0.4783	1.0000	0.4783	1.0000
	F1	0.4516	0.4516	0.5790	0.6829	0.5790	0.8750	0.5790	0.9200	0.5789	0.9200
M11	Precision	0.9676	0.9676	0.9569	0.9582	0.9514	0.9643	0.9543	0.9672	0.9569	0.9759
	Recall	0.9200	0.9200	0.9569	0.9877	0.9631	0.9969	0.9631	0.9969	0.9569	0.9969
	F1	0.9432	0.9432	0.9569	0.9727	0.9572	0.9803	0.9587	0.9818	0.9569	0.9863
M12	Precision	0.9338	0.9338	0.9264	0.9337	0.9273	0.9412	0.9268	0.9572	0.9273	0.9731
	Recall	0.7747	0.7747	0.8297	0.9286	0.8407	0.9670	0.8352	0.9835	0.8407	0.9945
	F1	0.8469	0.8469	0.8754	0.9311	0.8818	0.9540	0.8786	0.9702	0.8818	0.9837
M13	Precision	0.0000	0.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000
	Recall	0.0000	0.0000	0.0000	0.0625	0.0000	0.2500	0.0000	0.5000	0.0000	0.9375
	F1	0.0000	0.0000	0.0000	0.1177	0.0000	0.4000	0.0000	0.6667	0.0000	0.9677
M131	Precision	0.8462	0.8462	0.8630	0.8745	0.8341	0.9103	0.8377	0.9195	0.8597	0.9481
	Recall	0.8500	0.8500	0.8591	0.9500	0.8682	0.9682	0.8682	0.9864	0.8636	0.9955
	F1	0.8481	0.8481	0.8611	0.9107	0.8508	0.9383	0.8527	0.9518	0.8617	0.9712
M132	Precision	0.9250	0.9250	0.9355	0.9392	0.9079	0.9503	0.9080	0.9663	0.9136	0.9667
	Recall	0.8506	0.8506	0.8333	0.9770	0.8506	0.9885	0.8506	0.9885	0.8506	1.0000
	F1	0.8862	0.8862	0.8815	0.9578	0.8783	0.9690	0.8783	0.9773	0.8810	0.9831
M14	Precision	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
	Recall	0.7059	0.7059	0.8235	0.8529	0.8235	0.9412	0.8235	0.9751	0.3530	0.9760
	F1	0.8276	0.8276	0.9032	0.9206	0.9032	0.9697	0.9032	0.9851	0.9032	0.9851
M141	Precision	0.9925	0.9925	0.9828	0.9830	0.9828	0.9831	0.9828	0.9831	0.9828	0.9855
	Recall	0.9681	0.9681	0.9804	0.9902	0.9828	0.9951	0.9828	0.9976	0.9804	0.9976
	F1	0.9802	0.9802	0.9816	0.9866	0.9828	0.9890	0.9828	0.9903	0.9816	0.9915
M142	Precision	0.9839	0.9839	0.9841	0.9855	0.9846	0.9867	0.9851	1.0000	0.9844	1.0000
	Recall	0.7821	0.7821	0.7949	0.8718	0.8205	0.9487	0.8062	1.0000	0.8077	1.0000
	F1	0.8714	0.8714	0.8794	0.9252	0.8951	0.9673	0.9103	1.0000	0.8873	1.0000
M143	Precision	0.9452	0.9452	0.9467	0.9477	0.9533	0.9735	0.9597	0.9735	0.9533	0.9800
	Recall	0.9388	0.9388	0.9660	0.9864	0.9728	1.0000	0.9728	1.0000	0.9728	1.0000
	F1	0.9420	0.9420	0.9562	0.9667	0.9630	0.9866	0.9662	0.9866	0.9630	0.9899
Global	Micro-P.	0.9460	0.9460	0.9436	0.9462	0.9350	0.9579	0.9368	0.9642	0.9414	0.9733
	Micro-R.	0.8712	0.8712	0.8955	0.9521	0.9036	0.9764	0.9042	0.9882	0.9004	0.9963
	Micro-F1	**0.9070**	0.9070	**0.9189**	0.9491	**0.9190**	0.9670	**0.9202**	0.9760	**0.9205**	0.9846
	Macro-F1	**0.7597**	0.7597	**0.7870**	0.8372	**0.7891**	0.9029	**0.7910**	0.9429	**0.7914**	0.9778

Table 3. F1 for the individual categories together with the best achievable F1 on k hypotheses (oracle performance).

5.3 Experiments using the Full Reuters Hierarchy

We finally carried out an experiment in classification using the full Reuters hierarchy, although on a relatively small subset of the available data. This includes 5,598 documents in training set and 4,195 documents in the test set. The result is shown in Table 4 and demonstrates that the approach also scales up to larger hierarchies. The improvement for the 16-best reranker over the baseline is 4.6 points in micro F-measure and 7.1 points in macro F-measure.

	Reranked				Oracle			
K	Prec.	Rec.	Micro-F1	Macro-F1	Prec.	Rec.	Micro-F1	Macro-F1
1	0.8034	0.4639	0.5882	0.4227	0.8034	0.4639	0.5882	0.4227
2	0.7025	0.5657	0.6267	0.4842	0.7606	0.6765	0.7161	0.5734
4	0.7057	0.5716	0.6316	0.4883	0.7968	0.7542	0.7749	0.6315
8	0.7033	0.5769	0.6338	0.4905	0.8201	0.8117	0.8159	0.6796
16	0.6969	0.5821	0.6343	0.4933	0.8350	0.8479	0.8414	0.7096

Table 4. Oracle and reranker performance on the full Reuters hierarchy.

6 Conclusions

We have described a framework for reranking the output of an MCC. This is based on SVMs using structural kernels, which can learn to reorder a set of ranked hypothesis based on complex statistical dependencies. It should be noted that this algorithm is based on a simple binary classifier that selects the best hypothesis. We have seen a consistent improvement that is especially notable for categories with few training documents; it will be important to study whether our method addresses any of the well-known problems with large hierarchies with sparse training data [3].

One problem of the proposed approach may arise when very large categorization schemes are used since the use of tree kernel-based models may become impractical. However, our approach can be also applied by dividing the large hierarchy in different subparts and then applying tree kernels on such smaller subtrees. This is intuitively both efficient and accurate since it would be less likely to see strong statistical dependencies between nodes if these are very far away in the hierarchy. Additionally, two recent results support the viability of our approach: (i) fast algorithms for structural kernels have shown that large scale learning is practical [22, 23] and (ii) models based on structural kernels can be efficiently and effectively converted in linear models [19–21].

Finally, while we have presented a simple inference strategy based on reranking, there are also other approximate inference strategies that can be constructed with the k-best as a starting point. For instance, the k-best search algorithm for natural-language parsing presented in [12] was later used as the main building block in the *forest reranking* method for approximate inference in complex discriminative parsing models [11]. This approximate search method has also been used in joint syntactic and role-semantic analysis [14]. The forest reranking method is one way to address the common criticism of reranking systems, that is: they may be too constrained by the limited internal variation of the k-best hypothesis set.

Acknowledgements

The research described in this paper has been partially supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant 231126: LivingKnowledge – Facts, Opinions and Bias in Time, and under grant 247758: EternalS – Trustworthy Eternal Systems via Evolving Software, Data and Knowledge. Many thanks to the reviewers of LSHC2.

References

1. Ailon, N., Mohri, M.: Preference-based learning to rank. Machine Learning (2010)
2. Balcan, M.F., Bansal, N., Beygelzimer, A., Coppersmith, D., Langford, J., Sorkin, G.B.: Robust reductions from ranking to classification. Machine Learning 72(1-2), 139–153 (2008)
3. Bennett, P.N., Nguyen, N.: Refined experts: improving classification in large taxonomies. In: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval. pp. 11–18. ACM (2009)
4. Cancedda, N., Gaussier, E., Goutte, C., Renders, J.M.: Word sequence kernels. Journal of Machine Learning Research 3, 1059–1082 (2003)
5. Charniak, E., Johnson, M.: Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In: Proceedings of ACL. pp. 173–180. Ann Arbor, United States (2005)
6. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: Proceedings of ACL'02. pp. 263–270 (2002)

7. Culotta, A., Sorensen, J.: Dependency Tree Kernels for Relation Extraction. In: Proceedings of ACL'04 (2004)
8. Cumby, C., Roth, D.: On kernel methods for relational learning. In: Proceedings of ICML 2003 (2003)
9. Daumé III, H., Marcu, D.: NP bracketing by maximum entropy tagging and SVM reranking. In: Proceedings of EMNLP'04 (2004)
10. Diab, M., Moschitti, A., Pighin, D.: Semantic role labeling systems for Arabic using kernel methods. In: Proceedings of ACL-08: HLT. pp. 798–806 (2008)
11. Huang, L.: Forest reranking: Discriminative parsing with non-local features. In: Proceedings of ACL-08: HLT. pp. 586–594. Columbus, United States (2008)
12. Huang, L., Chiang, D.: Better k-best parsing. In: Proceedings of the 9th International Workshop on Parsing Technologies (IWPT 2005). pp. 53–64. Vancouver, Canada (2005)
13. Joachims, T.: Making large-scale SVM learning practical. Advances in Kernel Methods – Support Vector Learning 13 (1999)
14. Johansson, R.: Statistical bistratal dependency parsing. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing. pp. 561–569. Singapore (2009)
15. Kudo, T., Matsumoto, Y.: Fast methods for kernel-based text analysis. In: Proceedings of ACL'03 (2003)
16. Kudo, T., Suzuki, J., Isozaki, H.: Boosting-based parse reranking with subtree features. In: Proceedings of ACL'05 (2005)
17. Moschitti, A.: Efficient convolution kernels for dependency and constituent syntactic trees. In: Proceedings of ECML'06 (2006)
18. Moschitti, A., Pighin, D., Basili, R.: Tree kernels for semantic role labeling. Computational Linguistics 34(2), 193–224 (2008)
19. Pighin, D., Moschitti, A.: Efficient linearization of tree kernel functions. In: Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL-2009). pp. 30–38. Association for Computational Linguistics, Boulder, Colorado (June 2009)
20. Pighin, D., Moschitti, A.: Reverse engineering of tree kernel feature spaces. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing. pp. 111–120. Association for Computational Linguistics, Singapore (August 2009)
21. Pighin, D., Moschitti, A.: On reverse feature engineering of syntactic tree kernels. In: Proceedings of the Fourteenth Conference on Computational Natural Language Learning. pp. 223–233. Association for Computational Linguistics, Uppsala, Sweden (July 2010)
22. Severyn, A., Moschitti, A.: Large-scale support vector learning with structural kernels. In: ECML. pp. 229–244 (2010)
23. Severyn, A., Moschitti, A.: Fast support vector machines for structural kernels. In: ECML (2011)
24. Shen, L., Joshi, A.: An SVM-based voting algorithm with application to parse reranking. In: Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003. pp. 9–16 (2003)
25. Titov, I., Henderson, J.: Porting statistical parsers with data-defined kernels. In: Proceedings of CoNLL-X (2006)
26. Toutanova, K., Markova, P., Manning, C.e.: The leaf path projection view of parse trees: Exploring string kernels for HPSG parse selection. In: Proceedings of EMNLP 2004 (2004)