

# splitSVM: Fast, Space-Efficient, non-Heuristic, Polynomial Kernel Computation for NLP Applications

Yoav Goldberg and Michael Elhadad

Ben Gurion University of the Negev

Department of Computer Science

POB 653 Be'er Sheva, 84105, Israel

{yoavg, elhadad}@cs.bgu.ac.il

## Abstract

We present a fast, space efficient and non-heuristic method for calculating the decision function of polynomial kernel classifiers for NLP applications. We apply the method to the MaltParser system, resulting in a Java parser that parses over 50 sentences per second on modest hardware without loss of accuracy (a 30 time speedup over existing methods). The method implementation is available as the open-source splitSVM Java library.

## 1 Introduction

Over the last decade, many natural language processing tasks are being cast as classification problems. These are then solved by off-the-shelf machine-learning algorithms, resulting in state-of-the-art results. Support Vector Machines (SVMs) have gained popularity as they constantly outperform other learning algorithms for many NLP tasks.

Unfortunately, once a model is trained, the decision function for kernel-based classifiers such as SVM is expensive to compute, and can grow linearly with the size of the training data. In contrast, the computational complexity for the decision functions of most non-kernel based classifiers does not depend on the size of the training data, making them orders of magnitude faster to compute. For this reason, research effort was directed at speeding up the classification process of polynomial-kernel SVMs (Isozaki and Kazawa, 2002; Kudo and Matsumoto, 2003; Wu et al., 2007). Existing accelerated SVM solutions, however, either require large amounts of

memory, or resort to heuristics – computing only an approximation to the real decision function.

This work aims at speeding up the decision function computation for low-degree polynomial kernel classifiers while using only a modest amount of memory and still computing the exact function. This is achieved by taking into account the Zipfian nature of natural language data, and structuring the computation accordingly. On a sample application (replacing the libsvm classifier used by MaltParser (Nivre et al., 2006) with our own), we observe a speedup factor of 30 in parsing time.

## 2 Background and Previous Work

In classification based NLP algorithms, a word and its context is considered a learning sample, and encoded as *Feature Vectors*. Usually, context data includes the word being classified ( $w_0$ ), its part-of-speech (PoS) tag ( $p_0$ ), word forms and PoS tags of neighbouring words ( $w_{-2}, \dots, w_{+2}, p_{-2}, \dots, p_{+2}$ , etc.). Computed features such as the length of a word or its suffix may also be added. A feature vector ( $F$ ) is encoded as an indexed list of all the features present in the training corpus. A feature  $f_i$  of the form  $w_{+1} = \text{dog}$  means that the word following the one being classified is ‘dog’. Every learning sample is represented by an  $n = |F|$  dimensional binary vector  $x$ .  $x_i = 1$  iff the feature  $f_i$  is active in the given sample, 0 otherwise.  $n$  is the number of different features being considered. This encoding leads to vectors with extremely high dimensions, mainly because of lexical features  $w_i$ .

SVM is a supervised binary classifier. The result of the learning process is the set  $SV$  of Sup-

port Vectors, associated weights  $\alpha_i$ , and a constant  $b$ . The Support Vectors are a subset of the training feature vectors, and together with the weights and  $b$  they define a hyperplane that optimally separates the training samples. The basic SVM formulation is of a linear classifier, but by introducing a kernel function  $K$  that non-linearly transforms the data from  $R^n$  into a space of higher dimension, SVM can be used to perform non-linear classification. SVM’s decision function is:

$$y(x) = \text{sgn} \left( \sum_{j \in SV} y_j \alpha_j K(x_j, x) + b \right)$$

where  $x$  is an  $n$  dimensional feature vector to be classified. The kernel function we consider in this paper is a polynomial kernel of degree  $d$ :  $K(x_i, x_j) = (\gamma x_i \cdot x_j + c)^d$ . When using binary valued features (with  $\gamma = 1$  and  $c = 1$ ), this kernel function essentially implies that the classifier considers not only the explicitly specified features, but also all available sets of size  $d$  of features. For  $d = 2$ , this means considering all feature pairs, while for  $d = 3$  all feature triplets. In practice, a polynomial kernel with  $d = 2$  usually yields the best results in NLP tasks, while higher degree kernels tend to overfit the data.

## 2.1 Decision Function Computation

Note that the decision function involves a summation over all support vectors  $x_j$  in  $SV$ . In natural language applications, the size  $|SV|$  tends to be very large (Isozaki and Kazawa, 2002), often above 10,000. In particular, the size of the support vectors set can grow linearly with the number of training examples, of which there are usually at least tens of thousands. As a consequence, the computation of the decision function is computationally expensive. Several approaches have been designed to speed up the decision function computation.

**Classifier Splitting** is a common, application specific heuristic, which is used to speed up the training as well as the testing stages (Nivre et al., 2006). The training data is split into several datasets according to an application specific heuristic. A separate classifier is then trained for each dataset. For example, it might be known in advance that nouns usually behave differently than verbs. In such a case, one can train one classifier on noun instances, and a different classifier on verb instances. When

testing, only one of the classifiers will be applied, depending on the PoS of the word. This technique reduces the number of support vectors in each classifier (because each classifier was trained on only a portion of the data). However, it relies on human intuition on the way the data should be split, and usually results in a degradation in performance relative to a single classifier trained on all the data points.

**PKI – Inverted Indexing** (Kudo and Matsumoto, 2003), stores for each feature the support vectors in which it appears. When classifying a new sample, only the set of vectors relevant to features actually appearing in the sample are considered. This approach is non-heuristic and intuitively appealing, but in practice brings only modest improvements.

**Kernel Expansion** (Isozaki and Kazawa, 2002) is used to transform the  $d$ -degree polynomial kernel based classifier into a linear one, with a modified decision function  $y(x) = \text{sgn}(w \cdot x^d + b)$ .  $w$  is a very high dimensional weight vector, which is calculated beforehand from the set of support vectors and their corresponding  $\alpha_i$  values. (the calculation details appear in (Isozaki and Kazawa, 2002; Kudo and Matsumoto, 2003)). This speeds up the decision computation time considerably, as only  $|x|^d$  weights need to be considered,  $|x|$  being the number of active features in the sample to be classified, which is usually a very small number. However, even the sparse-representation version of  $w$  tends to be very large: (Isozaki and Kazawa, 2002) report that some of their second degree expanded NER models were more than 80 times slower to load than the original models (and 224 times faster to classify).<sup>1</sup> This approach obviously does not scale well, both to tasks with more features and to larger degree kernels.

**PKE – Heuristic Kernel Expansion**, was introduced by (Kudo and Matsumoto, 2003). This heuristic method addresses the deficiency of the Kernel Expansion method by using a basket-mining algorithm in order to greatly reduce the number of non-zero elements in the calculated  $w$ . A parameter is used to control the number of non-zero elements in  $w$ . The smaller the number, the smaller the memory requirement, but setting this number too low hurts classification performance, as only an approxima-

<sup>1</sup>Using a combination of 33 classifiers, the overall loading time is about 31 times slower, and classification time is about 21 times faster, than the non-expanded classifiers.

tion of the real decision function is calculated.

“**Semi Polynomial Kernel**” was introduced by (Wu et al., 2007). The intuition behind this optimization is to “extend the linear kernel SVM toward polynomial”. It does not train a polynomial kernel classifier, but a regular linear SVM. A basket-mining based feature selection algorithm is used to select “useful” pairs and triplets of features prior to the training stage, and a linear classifier is then trained using these features. Training (and testing) are faster than in the polynomial kernel case, but the result suffer quite a big loss in accuracy as well.<sup>2</sup>

### 3 Fast, Non-Heuristic Computation

We now turn to present our fast, space efficient and non-heuristic approach for computing the Polynomial Kernel decision function.<sup>3</sup> Our approach is a combination of the PKI and the Kernel Expansion methods. While previous works considered kernels of the form  $K(x, y) = (x \cdot y + 1)^d$ , we consider the more general form of the polynomial kernel:  $K(x, y) = (\gamma x \cdot y + c)^d$ .

Our key observation is that in NLP classification tasks, few of the features (e.g., `POS is X`, or `prev word is the`) are very frequent, while most others are extremely rare (e.g., `next word is polynomial`). The common features are active in many of the support-vectors, while the rare features are active only in few support vectors. This is true for most language related tasks: the Zipfian nature of language phenomena is reflected in the distribution of features in the support vectors.

It is because of common features that the PKI reverse indexing method does not yield great improvements: if at least one of the features of the current instance is active in a support vector, this vector is taken into account in the sum calculation, and the common features are active in many support vectors.

On the other hand, the long tail of rare features is the reason the Kernel Expansion methods requires

<sup>2</sup>This loss of accuracy in comparison to the PKE approach is to be expected, as (Goldberg and Elhadad, 2007) showed that the effect of removing features prior to the learning stage is much more severe than removing them after the learning stage.

<sup>3</sup>Our presentation is for the case where  $d = 2$ , as this is by far the most useful kernel. However, the method can be easily adapted to higher degree kernels as well. For completeness, our toolkit provides code for  $d = 3$  as well as 2.

so much space: every rare feature adds many possible feature pairs.

We propose a combined method. We first split common from rare features. We then use Kernel Expansion on the few common features, and PKI for the remaining rare features. This ensures small memory footprint for the expanded kernel vector, while at the same time keeping a low number of vectors from the reverse index.

#### 3.1 Formal Details

The polynomial kernel of degree 2 is:  $K(x, y) = (\gamma x \cdot y + c)^2$ , where  $x$  and  $y$  are binary feature vectors.  $x \cdot y$  is the dot product between the vectors, and in the case of binary feature vectors it corresponds to the count of shared features among the vectors.  $F$  is the set of all possible features.

We define  $F_R$  and  $F_C$  to be the sets of rare and common features.  $F_R \cap F_C = \emptyset$ ,  $F_R \cup F_C = F$ . The mapping function  $\phi_R(x)$  zeros out all the elements of  $x$  not belonging to  $F_R$ , while  $\phi_C(x)$  zeroes out all the elements of  $x$  not in  $F_C$ . Thus, for every  $x$ :  $\phi_R(x) + \phi_C(x) = x$ ,  $\phi_R(x) \cdot \phi_C(x) = 0$ . For brevity, denote  $\phi_C(x) = x_C$ ,  $\phi_R(x) = x_R$ .

We now rewrite the kernel function:

$$\begin{aligned} K(x, y) &= K(x_R + x_C, y_R + y_C) = \\ &= (\gamma(x_R + x_C) \cdot (y_R + y_C) + c)^2 \\ &= (\gamma x_R \cdot y_R + \gamma x_C \cdot y_C + c)^2 \\ &= (\gamma x_R \cdot y_R)^2 \\ &\quad + 2\gamma^2(x_R \cdot y_R)(x_C \cdot y_C) \\ &\quad + 2c\gamma(x_R \cdot y_R) \\ &\quad + (\gamma(x_C \cdot y_C) + c)^2 \end{aligned}$$

The first 3 terms are non-zero only when at least one rare feature exists. We denote their sum  $K_R(x, y)$ . The last term involves only common features. We denote it  $K_C(x, y)$ . Note that  $K_C(x, y)$  is the polynomial kernel of degree 2 over feature vectors of only common features.

We can now write the SVM decision function as:

$$\sum_{j \in SV} y_j \alpha_j K_R(x_j, x_R) + \sum_{j \in SV} y_j \alpha_j K_C(x_j, x_C) + b$$

We calculate the first sum via PKI, taking into account only support-vectors which share at least one feature with  $x_R$ . The second sum is calculated via kernel expansion while taking into account only the

common features. Thus, only pairs of common features appear in the resulting weight vector using the same expansion as in (Kudo and Matsumoto, 2003; Isozaki and Kazawa, 2002). In our case, however, the expansion is memory efficient, because we consider only features in  $F_C$ , which is small.

Our approach is similar to the PKE approach (Kudo and Matsumoto, 2003), which used a basket mining approach to prune many features from the expansion. In contrast, we use a simpler approach to choose which features to include in the expansion, and we also compensate for the feature we did not include by the PKI method. Thus, our method generates smaller expansions while computing the exact decision function and not an approximation of it.

We take every feature occurring in less than  $s$  support vectors to be rare, and the other features to be common. By changing  $s$  we get a trade-off between space and time complexity: smaller  $s$  indicate more common features (bigger memory requirement) but also less rare features (less support vectors to include in the summation), and vice-versa. In contrast to other methods, changing  $s$  is guaranteed not to change the classification accuracy, as it does not change the computed decision function.

## 4 Toolkit and Evaluation

Using this method, one can accelerate SVM-based NLP application by just changing the classification function, keeping the rest of the logic intact. We implemented an open-source software toolkit, freely available at <http://www.cs.bgu.ac.il/~nlpproj/>. Our toolkit reads models created by popular SVM packages (libsvm, SVMLight, TinySVM and Yamcha) and transforms them into our format. The transformed models can then be used by our efficient Java implementation of the method described in this paper. We supply wrappers for the interfaces of libsvm and the Java bindings of SVMLight. Changing existing Java code to accommodate our fast SVM classifier is done by loading a different model, and changing a single function call.

### 4.1 Evaluation: Speeding up MaltParser

We evaluate our method by using it as the classification engine for the Java version of MaltParser, an SVM-based state of the art dependency parser (Nivre et al., 2006). MaltParser uses the libsvm

classification engine. We used the pre-trained English models (based on sections 0-22 of the Penn WSJ) supplied with MaltParser. MaltParser already uses an effective *Classifiers Splitting* heuristic when training these models, setting a high baseline for our method. The pre-trained parser consists of hundreds of different classifiers, some very small. We report here on actual memory requirement and parsing time for sections 23-24, considering the classifier combination. We took rare features to be those appearing in less than 0.5% of the support vectors, which leaves us with less than 300 common features in each of the “big” classifiers. The results are summarized in Table 1. As can be seen, our method parses

Method	Mem.	Parsing Time	Sents/Sec
Libsvm	240MB	2166 (sec)	1.73
ThisPaper	750MB	70 (sec)	53

Table 1: Parsing Time for WSJ Sections 23-24 (3762 sentences), on Pentium M, 1.73GHz

about 30 times faster, while using only 3 times as much memory. MaltParser coupled with our fast classifier parses above 3200 sentences per minute.

## 5 Conclusions

We presented a method for fast, accurate and memory efficient calculation for polynomial kernels decisions functions in NLP application. While the method is applied to SVMs, it generalizes to other polynomial kernel based classifiers. We demonstrated the method on the MaltParser dependency parser with a 30-time speedup factor on overall parsing time, with low memory overhead.

## References

- Y. Goldberg and M. Elhadad. 2007. SVM model tampering and anchored learning: A case study in hebrew. np chunking. In *Proc. of ACL2007*.
- H. Isozaki and H. Kazawa. 2002. Efficient support vector classifiers for named entity recognition. In *Proc. of COLING2002*.
- T. Kudo and Y. Matsumoto. 2003. Fast methods for kernel-based text analysis. In *ACL-2003*.
- J. Nivre, J. Hall, and J. Nillson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. In *Proc. of LREC2006*.
- Y. Wu, J. Yang, and Y. Lee. 2007. An approximate approach for training polynomial kernel svms in linear time. In *Proc. of ACL2007 (short-paper)*.