Feature Selection in Kernel Space: A Case Study on Dependency Parsing

Xian Qian and **Yang Liu**
The University of Texas at Dallas
800 W. Campbell Rd., Richardson, TX, USA
{qx,yangl}@hlt.utdallas.edu

Abstract

Given a set of basic binary features, we propose a new L_1 norm SVM based feature selection method that explicitly selects the features in their polynomial or tree kernel spaces. The efficiency comes from the anti-monotone property of the subgradients: the subgradient with respect to a combined feature can be bounded by the subgradient with respect to each of its component features, and a feature can be pruned safely without further consideration if its corresponding subgradient is not steep enough. We conduct experiments on the English dependency parsing task with a third order graph-based parser. Benefiting from the rich features selected in the tree kernel space, our model achieved the best reported unlabeled attachment score of 93.72 without using any additional resource.

1 Introduction

In Natural Language Processing (NLP) domain, existing linear models typically adopt exhaustive search to generate tons of features such that the important features are included. However, the brute-force approach will guickly run out of memory when the feature space is extremely large. Unlike linear models, kernel methods provide a powerful and unified framework for learning a large or even infinite number of features implicitly using limited memory. However, many kernel methods scale quadratically in the number of training samples, and can hardly reap the benefits of learning a large dataset. For example, the popular Penn Tree Bank (PTB) corpus for training an English part of speech (POS) tagger has approximately $1M$ words, thus it takes $1M^2$ time to compute the kernel matrix, which is unacceptable using current hardwares.

In this paper, we propose a new feature selection method that can efficiently select representative features in the kernel space to improve the quality of linear models. Specifically, given a limited number of basic features such as the commonly used unigrams and bigrams, our method performs feature selection in the space of their combinations, e.g, the concatenation of these n-grams. A sparse discriminative model is produced by training L_1 norm SVMs using subgradient methods. Different from traditional training procedures, we divide the feature vector into a number of segments, and sort them in a coarse-to-fine order: the first segment includes the basic features, the second segment includes the combined features composed of two basic features, and so on. In each iteration, we calculate the subgradient segment by segment. A combined feature and all its further combinations in the following segments can be safely pruned if the absolute value of its corresponding subgradient is not sufficiently large. The algorithm stops until all features are pruned. Besides, two simple yet effective pruning strategies are proposed to filter the combinations.

We conduct experiments on English dependency parsing task. Millions of deep, high order features derived by concatenating contextual words, POS tags, directions and distances of dependencies are selected in the polynomial kernel and tree kernel spaces. The result is promising: these features significantly improved a state-of-the-art third order dependency parser, yielding the best reported unlabeled attachment score of 93.72 without using any additional resource.

Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing, pages 1180–1190,
Beijing, China, July 26-31, 2015. ©2015 Association for Computational Linguistics

2 Related Works

There are two solutions for learning in ultra high dimensional feature space: kernel method and feature selection.

Fast kernel methods have been intensively studied in the past few years. Recently, randomized methods have attracted more attention due to its theoretical and empirical success, such as the Nyström method (Williams and Seeger, 2001) and random projection (Lu et al., 2014). In NLP domain, previous studies mainly focused on polynomial kernels, such as the splitSVM and approximate polynomial kernel (Wu et al., 2007).

In feature selection domain, there has been plenty of work focusing on fast computation, while feature selection in extremely high dimensional feature space is relatively less studied. Zhang et al. (2006) proposed a progressive feature selection framework that splits the feature space into tractable disjoint sub-spaces such that a feature selection algorithm can be performed on each one of them, and then merges the selected features from different sub-spaces. The search space they studied contained more than 20 million features. Tan et al. (2012) proposed adaptive feature scaling (AFS) scheme for ultra-high dimensional feature selection. The dimensionality of the features in their experiments is up to 30 millions.

Previous studies on feature selection in kernel space typically used mining based approaches to prune feature candidates. The key idea for efficient pruning is to estimate the upper bound of statistics of features without explicit calculation. The simplest example is frequent mining where for any n-gram feature, its frequency is bounded by any of its substrings.

Suzuki et al. (Suzuki et al., 2004) proposed to select features in convolution kernel space based on their chi-squared values. They derived a concise form to estimate the upper bound of chi-square values, and used PrefixScan algorithm to enumerates all the significant sub-sequences of features efficiently.

Okanohara and Tsujii (Okanohara and Tsujii, 2009) further combined the pruning technique with L_1 regularization. They showed the connection between L_1 regularization and frequent mining: the L_1 regularizer provides a minimum support threshold to prune the gradients of parameters. They selected the combination features in a coarse-to-fine order, the gradient value for a combination feature can be bounded by each of its component feature, hence may be pruned without explicit calculation. They also sorted the features to tighten the bound. Our idea is similar with theirs, the difference is that our search space is much larger: we did not restrict the number of component features. We recursively pruned the feature set and in each recursion we selected feature in a batch manner. We further adopted an efficient data structure, spectral bloom filter, to estimate the gradients for the candidate features without generating them.

3 The Proposed Method

3.1 Basic Idea

Given n training samples $x_1 \dots x_n$ with labels $y_1 \dots y_n \in \mathcal{Y}$, we extend the kernel over the input space to the joint input and output space by simply defining $\mathbf{f}^T(x_i, y)\mathbf{f}(x_i, y') = K(x_i, x_j)I(y == y')$, which is the same as Taskar's (see (Taskar, 2004), Page 68), where $\mathbf{f}$ is the explicit feature map for the kernel, and $I(\cdot, \cdot)$ is the indicator function.

Our task is to select a subset of representative elements in the feature vector $\mathbf{f}$. Unlike previously studied feature selection problems, the dimension of $\mathbf{f}$ could be extremely high. It is impossible to store the feature vector in the memory or even on the disk.

For easy illustration, we describe our method for the polynomial kernel, and it can be easily extended to the tree kernel space.

The R degree polynomial kernel space is established by a set of basic features $\mathcal{B} = \{b_0 = 1, b_1, \dots, b_{|\mathcal{B}|}\}$ and their combinations. In other words, each feature is the product of at most R basic features $f_j = b_{j_1} * b_{j_2} * \dots * b_{j_r}$, $r \le R$. As we assume that all features are binary [1], f_j can be rewritten as the minimum of these basic features: $f_j = \min\{b_{j_1}, b_{j_2}, \dots, b_{j_r}\}$. We use $\mathcal{B}_j = \{b_{j_1}, b_{j_2}, \dots, b_{j_r}\}$ to denote the set of component basic features for f_j. r is called the order of feature f_j. For two features f_j, f_k, we say f_k is an extension of f_j if $\mathcal{B}_j \subset \mathcal{B}_k$.

Take the document classification task as an example, the basic features could be word n-grams, and the quadratic kernel (degree=2) space includes the combinated features composed of two

[1] Binary features are often used in NLP.

n-grams, a second order feature is true if both n-grams appear in the document, it is an extension of any of its component n-grams (first order features).

We use L_1 norm SVMs for feature selection. Traditionally, the L_1 norm SVMs can be trained using subgradient descent and generate a sparse weight vector $\mathbf{w}$ for feature $\mathbf{f}$. Due to the high dimensionality in our case, we divide $\mathbf{f}$ into a number of segments according to the order of the feature, the k-th segment includes the k-order features. In each iteration, we update the weights of features segment by segment. When updating the weight of feature f_j in the k-th segment, we estimate the subgradients with respective to f_j's extensions in the rest $k+1, k+2, \ldots$ segments and keep their weights at zero if the subgradients are not sufficiently steep. In this way, we could ignore these features without explicit calculation.

3.2 L_1 Norm SVMs

Specifically, the objective function for learning L_1 norm SVMs is:

$$\min_{\mathbf{w}} \mathcal{O}(\mathbf{w}) = C\|\mathbf{w}\|_1 + \sum_i loss(i)$$

where

$$loss(i) = \max_{y \in \mathcal{Y}} \{\mathbf{w}^T \Delta\mathbf{f}(x_i, y) + \delta(y_i, y)\}$$

is the hinge loss function for the i-th sample. $\Delta\mathbf{f}(x_i, y) = \mathbf{f}(x_i, y_i) - \mathbf{f}(x_i, y)$ is the residual feature vector, $\delta(a, b) = 0$ if $a = b$, otherwise $\delta(a, b) = 1$. Regularization parameter C controls the sparsity of $\mathbf{w}$. With higher C, more zero elements are generated. We call a feature is fired if its value is 1.

The objective function is a sum of piecewise linear functions, hence is convex. Subgradient descent algorithm is one poplar approach for minimizing non-differentiable convex functions, it updates $\mathbf{w}$ using

$$\mathbf{w}^{new} = \mathbf{w} - \mathbf{g}\alpha^t$$

where $\mathbf{g}$ is the subgradient of $\mathbf{w}$, α^t is the step size in the t-th iteration. Subgradient algorithm converges if the step size sequence is properly selected (Boyd and Mutapcic, 2006).

We are interested in the non-differentiable point $w_j = 0$. Let $y_i^* = \max_y \{\mathbf{w}^T \Delta\mathbf{f}(x_i, y) + \delta(y_i, y)\}$, the prediction of the current model. According to the definition of subgradient, we have, for each sample x_i, $\Delta\mathbf{f}(x_i, y_i^*)$ is a subgradient of $loss(i)$, thus, $\sum_i \Delta\mathbf{f}(x_i, y_i^*)$ is a subgradient of $\sum_i loss(i)$.

Adding the penalty term $C\|\mathbf{w}\|_1$, we get the subset of subgradients at $w_j = 0$ for the objective function

$$\sum_i \Delta f_j(x_i, y_i^*) - C \leq g_j \leq \sum_i \Delta f_j(x_i, y_i^*) + C$$

We can pick any g_j to update w_j. Remind that our purpose is to keep the model sparse, and we would like to pick $g_j = 0$ if possible. That is, we can keep $w_j = 0$ if $|\sum_i \Delta f_j(x_i, y_i^*)| \leq C$.

Obviously, for any j, we have $|\sum_i \Delta f_j(x_i, y_i^*)| \leq \sum_i \sum_y f_j(x_i, y) = \#f_j$, i.e., the frequency of feature f_j. Thus, we have

Proposition 1 *Let C be the threshold of the frequency, the model generated by the subgradient method is sparser than frequent mining.*

3.3 Feature Selection Using Gradient Mining

Now the problem is how to estimate $|\sum_i \Delta f_j(x_i, y_i^*)|$ without explicit calculation for each f_j.

In the following, we mix the terminology gradient and subgradient without loss of clarity. We define the positive gradient and negative gradient for w_j

$$\#f_j^+ = \sum_{i, y_i \neq y_i^*} f_j(x_i, y_i)$$

$$\#f_j^- = \sum_{i, y_i \neq y_i^*} f_j(x_i, y_i^*)$$

We have

$$\begin{aligned} \sum_i \Delta f_j(x_i, y_i^*) &= \sum_{i, y_i^* \neq y_i} \Delta f_j(x_i, y_i^*) \\ &= \#f_j^+ - \#f_j^- \end{aligned}$$

The estimation problem turns out to be a counting problem: we collect all the incorrectly predicted samples, and count $\#f_j^+$, the frequency of f_j fired by the gold labels, and $\#f_j^-$ the frequency of f_j fired by the predictions.

As mentioned above, each feature in polynomial kernel space is defined as $f_j = \min\{b \in \mathcal{B}_j\} = \min\{b_{j_1}, \ldots, b_{j_r}\}$. Equivalently, we can define f_j in a recursive way, which is more frequently used in the rest of the paper. That is, $f_j =$

$\min\{\min\{b_{j_2},\dots,b_{j_r}\},\min\{b_{j_1},b_{j_3},\dots,b_{j_r}\},\dots\}$, which is the mimum of r features of order $r-1$. Formally, denote $\mathcal{B}_j^{-i}$ as the subset of $\mathcal{B}_j$ by removing its i-th element, then the r-order feature, we have $f_j = \min\{h_1,\dots,h_r\}$, where $h_k = \min\{b \in \mathcal{B}_j^{-k}\}, 1 \le k \le r$.

We have the following anti-monotone property, which is the basis of our method

$$\#f_j^+ \le \#h_k^+ \quad \forall k$$
$$\#f_j^- \le \#h_k^- \quad \forall k$$

If there exists a k, such that $\#h_k^+ \le C$ and $\#h_k^- \le C$, we have

$$\begin{aligned} & |\sum_i \Delta f_j(x_i, y_i^*)| \\ = & \ |\#f_j^+ - \#f_j^-| \\ \le & \ \max\{\#f_j^+, \#f_j^-\} \\ \le & \ \max\{\min_k\{\#h_k^+\}, \min_k\{\#h_k^-\}\} \\ \le & \ \min_k\{\max\{\#h_k^+, \#h_k^-\}\} \\ \le & \ C \end{aligned}$$

The third inequality comes from the well known min-max inequality: $\max_i \min_j\{a_{ij}\} \le \min_j \max_i\{a_{ij}\}$. Thus, we could prune f_j without calculating its corresponding gradient.

This is a chain rule, which means that any feature that has f_j as its component can also be pruned safely. To see this, suppose $\phi = \min\{\dots, f_j, \dots\}$ is such a combined feature, we have

$$\begin{aligned} |\#\phi^+ - \#\phi^-| & \le \max\{\#\phi^+, \#\phi^-\} \\ & \le \max\{\#f_j^+, \#f_j^-\} \\ & \le C \end{aligned}$$

Based on this, we present the gradient mining based feature selection framework in Algorithm 1.

Algorithm 1 Feature Selection Using Gradient Mining

Require: Samples $X = \{x_1,\dots,x_n\}$ with labels $\{y_1,\dots,y_n\}$, basic features $\mathcal{B} = \{b_1,\dots,b_{|\mathcal{B}|}\}$, threshold $C > 0$, max iteration number M, degree of polynomial kernel R, sequence of learning step $\{\alpha^t\}$.
Ensure: Set of selected features $\mathcal{S} = \{f_j\}$, where $f_j = \min\{b \in \mathcal{B}_j\}, \mathcal{B}_j \subseteq \mathcal{B}, |\mathcal{B}_j| \le R$.

1: $\mathcal{S}_r = \emptyset, r = 1,\dots,R$ {$\mathcal{S}_r$ denotes the selected r-order features}
2: **for** $t = 1 \to M$ **do**
3: Set $\mathcal{S} = \bigcup_{r=1}^R \mathcal{S}_r$, $\mathbf{f}$ = the vector of features in $\mathcal{S}$.
4: Calculate $y_i^* = \max_y\{\mathbf{w}^T\mathbf{f}(x_i,y) + \delta(y_i,y)\}, \forall i$.
5: Initialize candidate set $\mathcal{A} = \mathcal{B}$
6: **for** $r = 1 \to R$ **do**
7: **for all** $f_j \in \mathcal{A}$ **do**
8: Calculate $\#f_j^+ = \sum_{i,y_i \ne y_i^*} f_j(x_i,y_i)$ and $\#f_j^- = \sum_{i,y_i \ne y_i^*} f_j(x_i,y_i^*)$
9: **if** $\#f_j^+, \#f_j^- \le C$ **and** $w_j = 0$ **then**
10: Remove f_j from $\mathcal{A}$
11: **else**
12: $w_j = w_j + (\#f_j^+ - \#f_j^- + C\text{sign}(w_j))\alpha^t$
13: **end if**
14: **end for**
15: $\mathcal{S}_r = \mathcal{A}$
16: **if** $r < R$ **then**
17: Generate order-$r+1$ candidates: $\mathcal{A} = \mathcal{S}_{r+1} \bigcup \{h | h = \min\{f_1,\dots f_r \in \mathcal{S}_r\}$, order of h is $r+1\}$
18: **end if**
19: **end for**
20: **end for**

4 Prune the Candidate Set

In practice, Algorithm 1 is far from efficient because Line 17 may generate large amounts of candidate features that quickly consume the memory. In this section, we introduce two pruning strategies that could greatly reduce the size of candidates.

4.1 Pre-Training

Usually, the weights of features are initialized with 0 in the training procedure. However, this will select too many features in the first iteration, because all samples are mis-classified in Line 4, the gradients $\#f_j^+$ and $\#f_j^-$ equal to the frequencies of the features, and many of them could be larger than C. Luckily, due to the convexity of piecewise linear function, the optimality of subgradient method is irrelevant with the initial point. So we can start with a well trained model using a small subset of features such as the set of lower order features so that the prediction is more accurate and the gradients $\#f^+$ and $\#f^-$ are much lower.

4.2 Bloom Filter

The second strategy is to use bloom filter to reduce candidates before putting them into the candidate set $\mathcal{A}$.

A bloom filter (Bloom, 1970) is a space efficient probabilistic data structure designed to rapidly check whether an element is present in a set. In this paper, we use one of its extension, spectral

bloom filter (Cohen and Matias, 2003), which can efficiently calculate the upper bound of the frequencies of elements.

The base data structure of a spectral bloom filter is a vector of L counters, where all counters are initialized with 0. The spectral bloom filter uses m hash functions, $h_1, \ldots, h_m$, that map the elements to the range $\{1, \ldots L\}$. When adding an element f to the bloom filter, we hash it using the m hash functions, and get the hash codes $h_1(f), \ldots, h_m(f)$, then we check the counters at positions $h_1(f), \ldots, h_m(f)$, and get the counts $\{c_1, \ldots, c_m\}$. Let c^* be the minimal count among these counts: $c^* = \min\{c_1, \ldots, c_m\}$, we increase only the counters whose counts are c^*, while keeping other counters unchanged.

To check the frequency of an element, we hash the element and check the counters in the same way. The minimum count c^* provides the upper bound of the frequency. In other words, when pruning elements with frequencies no greater than a predefined threshold θ, we could safely prune the element if $c^* \leq \theta$.

In our case, we use the spectral bloom filter to eliminate the low-frequency candidates.

To estimate the gradients of newly generated $r+1$-order candidates, we run Line 17 twice. In the first round, we estimate the upper bound of $\#h^+$ for each candidate and add the candidate to $\mathcal{A}$ if its upper bound is greater than a predefined threshold θ. The second round is similar, we add the candidates using the upper bound of h^-. We did not estimate $\#h^+$ and $\#h^-$ simultaneously, because this needs two bloom filters for positive and negative gradients respectively, which consumes too much memory.

Specifically, in the first round, we initialize the spectral bloom filter so that all counters are set to zero. Then for each incorrectly predicted sample x_i, we generate $r+1$-order candidates by combining r-order candidates that are fired by the gold label i.e., $f(x_i, y_i) = 1$. Once a new candidate is generated, we hash it and check its corresponding m counters in the spectral bloom filter. If the minimal count $c^* = \theta$, we know that its positive gradient $\#f^+$ may be greater than θ. So we keep all counts unchanged, and add the candidate to $\mathcal{A}$. Otherwise, we increase the counts by 1 using the method described above. The second round is similar.

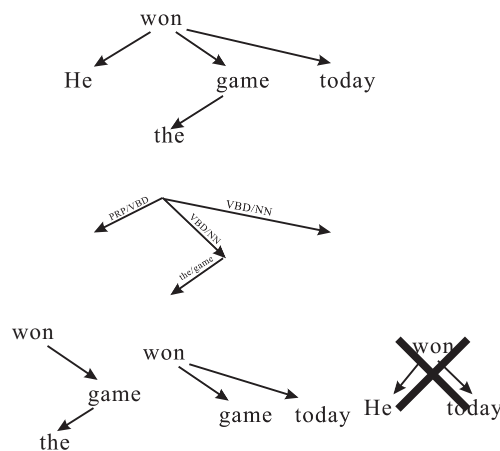

Figure 1: A dependency parse tree (top), one of its feature trees (middle) and some of its subtrees (bottom). *He* ← *won* → *today* is not a subtree because *He* and *today* are not adjacent siblings.

5 Efficient Candidate Generation

5.1 Polynomial Kernel

As mentioned above, we generate the $r+1$-order candidates by combining the candidates of order r. An efficient feature generation algorithm should be carefully designed to avoid duplicates, otherwise $\#f^+$ and $\#f^-$ may be over counted.

The candidate generation algorithm is kernel dependent. For polynomial kernel, we just combine any two r-order candidates and remove the combined feature if its order is not $r+1$. This method requires square running time for each example.

5.2 Dependency Tree Kernel

5.2.1 Definition

Collins and Duffy (2002) proposed tree kernels for constituent parsing which includes the all-subtree features. Similarly, we define dependency tree kernel for dependency parsing. For compatibility with the previously studied subtree features for dependency parsing, we propose a new dependency tree kernel that is different from Culotta and Sorensen's (Culotta and Sorensen, 2004). Given a dependency parse tree T composed of L words, $L-1$ arcs, each arc has several basic features, such as the concatenation of the head word and the modifier word, the concatenation of the word left to the head and the lower case of the word right to the modifier, the distance of the arc, the direction of the arc, the

concatenation of the POS tags of the head and the modifier, etc.

A feature tree of T is a tree that has the same structure as T, while each arc is replaced by any of its basic features. For a parse tree that has $L-1$ arcs, and each arc has d basic features, the number of the feature trees is d^{L-1}. For example, the dependency parse tree for sentence *He won the game today* is shown in Figure 1. Suppose each arc has two basic features: word pair and POS tag pair. Then there are 2^4 feature trees, because each arc can be replaced by either word pair or POS tag pair.

A subtree of a tree is a connected fragment in the tree. In this paper, to reduce computational cost, we restrict that adjacent siblings in the subtrees must be adjacent in the original tree. For example *He* ← *won* → *game* is a subtree, but *He* ← *won* → *today* is not a subtree. The motivation of the restriction is to reduce the number of subtrees, for a node having k children, there are $k^(k-1)/2$ subtrees, but without the restriction the number of subtrees is exponential: 2^k.

A sub feature tree of a dependency tree T is a feature tree of any of its subtrees. For example, the dependency tree in Figure 1 has 12 subtrees including four arcs, four arc pairs, the three arc triples and the full feature tree, and each subtree having s arcs has 2^s sub feature trees. Thus the dependency tree has $2*4+4*2^2+3*2^3+2^4=64$ sub feature trees.

Given two dependency trees T_1 and T_2, the dependency tree kernel is defined as the number of common sub feature trees of T_1 and T_2. Formally, the kernel function is defined as

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

where $\Delta(n_1, n_2)$ denotes the number of common sub feature trees rooted in n_1 and n_2 nodes.

Like tree kernel, we can calculate $\Delta(n_1, n_2)$ recursively. Let c_i and c'_j denote the i-th child of n_1 and j-th child of n_2 respectively, let $ST_{p,l}(n_1)$ denote the set of the sub feature trees rooted in node n_1 and the children of the root are $c_p, c_{p+1}, \ldots, c_{p+l-1}$, we denote $ST_{q,l}(n_2)$ similarly. Then we define

$$\Delta_{p,q,l}(n_1, n_2) = \sum_{p,q} |ST_{p,l}(n_1) \bigcap ST_{q,l}(n_2)|$$

the number of common sub feature trees in $ST_{p,l}(n_1)$ and $ST_{q,l}(n_2)$.

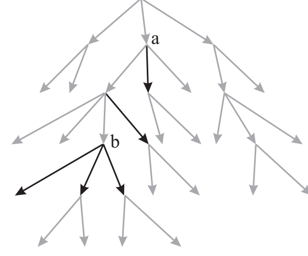

Figure 2: For any subtree rooted in a with the rightmost leaf b, we could extend the subtree by any arc below or right to the path from a to b (shown in black)

To calculate $\Delta_{p,q,l}(n_1, n_2)$, we first consider the sub feature trees with only two levels, i.e., sub feature trees that are composed of n_1, n_2 and some of their children. We initialize $\Delta_{p,q,1}(n_1, n_2)$ with number of the common features of arcs $n_1 \to c_p$ and $n_2 \to c'_q$. Then we calculate $\Delta_{p,q,l}(n_1, n_2)$ recursively using

$$\begin{aligned}&\Delta_{p,q,l}(n_1, n_2)\\ =&\Delta_{p,q,l-1}(n_1, n_2) * \Delta_{p+l,q+l,1}(n_1, n_2)\end{aligned}$$

And $\Delta(n_1, n_2) = \sum_{p,q,l} \Delta_{p,q,l}(n_1, n_2)$

Next we consider all the sub feature trees, we have

$$\begin{aligned}&\Delta_{p,q,l}(n_1, n_2)\\ =&\Delta_{p,q,l-1}(n_1, n_2) * \left(1 + \Delta(c_{p+l-1}, c'_{q+l-1})\right)\end{aligned}$$

Computing the dependency tree kernel for two parse trees requires $|T_1|^2 * |T_2|^2 * \min\{|T_1|, |T_2|\}$ running time in the worst case, as we need to enumerate p, q, l and n_1, n_2.

One way to incorporate the dependency tree kernel for parsing is to rerank the K best candidate parse trees generated by a simple linear model. Suppose there are n training samples, the size of the kernel matrix is $(K * n)^2$, which is unacceptable for large datasets.

5.2.2 Candidate Generation

For constituent parsing, Kudo et al. showed such an all-subtrees representation is extremely redundant and a comparable accuracy can be achieved using just a small set of subtrees (Kudo et al., 2005). Suzuki et al. even showed that the over-fitting problem often arises when convolution kernels are used in NLP tasks (Suzuki et al., 2004). Now we attempt to select representative sub

feature trees in the kernel space using Algorithm 1. The r-order features in dependency tree kernel space are the sub feature trees with r arcs. The candidate feature generation in Line 17 has two steps: first we generate the subtrees with r arcs, then we generate the sub feature trees for each subtree.

The simplest way for subtree generation is to enumerate the combinations of $r + 2$ words in the sentence, and check if these words form a subtree.

We can speed up the generation by using the results of the subtrees with $r + 1$ words (r arcs). For each subtree S_r with r arcs, we can add an extra word to S_r and generate S_{r+1} if the words form a subtree.

This method has three issues: first, the time complexity is exponential in the length of the sentence, as there are 2^L combinations of words, L is the sentence length; second, it may generate duplicated subtrees, and over counts the gradients. For example, there are two ways to generate the subtree *He won the game* in Figure 1: we can either add word *He* to the subtree *won the game*, or add word *the* to the subtree *He won game*; third, checking a fragment requires $O(L)$ time.

These issues can be solved using the well known rightmost-extension method (Zaki, 2002; Asai et al., 2002; Kudo et al., 2005) which enumerates all subtrees from a given tree efficiently. This method starts with a set of trees consisting of single nodes, and then expands each subtree attaching a new node.

Specifically, it first indexes the words in the pre-order of the parse tree. When generating S_{r+1}, only the words whose indices are larger than the greatest index of the words in S_r are considered. In this way, each subtree is generated only once. Thus, we only need to consider two types of words: (i) the children of the rightmost leaf of S_r, (ii) the adjacent right sibling of the any node in S_r, as shown in Figure 2.

The total number of subtrees is no greater than L^3, because the level of a subtree is less than L, and for the children of each node, there are at most L^2 subsequences of siblings. Therefore the time complexity for subtree extraction is $O(L^3)$.

6 Experiments

6.1 Experimental Results on English Dataset

6.1.1 Settings

First we used the English Penn Tree Bank (PTB) with standard train/develop/test for evaluation. Sections 2-21 (around 40K sentences) were used as training data, section 22 was used as the development set and section 23 was used as the final test set.

We extracted dependencies using Joakim Nivre's Penn2Malt tool with Yamada and Matsumoto's rules (Yamada and Matsumoto, 2003). Unlabeled attachment score (UAS) ignoring punctuation is used to evaluate parsing quality.

We apply our technique to rerank the parse trees generated by a third order parser (Koo and Collins, 2010) trained using 10 best MIRA algorithm with 10 iterations. We generate the top 10 best candidate parse trees using 10 fold cross validation for each sentence in the training data. The gold parse tree is added if it is not in the candidate list. Then we learn a reranking model using these candidate trees. During testing, the score for a parse tree T is a linear combination of the two models:

$$score(T) = \beta score^{O3}(T) + score^{rerank}(T)$$

where the meta-parameter $\beta = 5$ is tuned by grid search using the development dataset. $score^{O3}(T)$ and $score^{rerank}(T)$ are the outputs of the third order parser and the reranking classifier respectively.

For comparison, we implement the following reranking models:

- Perceptron with Polynomial kernels $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b} + 1)^d$, $d = 2, 4, 8$
- Perceptron with Dependency tree kernel.
- Perceptron with features generated by templates, including all siblings and fourth order features.
- Perceptron with the features selected in polynomial and tree kernel spaces, where threshold $C = 3$.

The basic features to establish the kernel spaces include the combinations of contextual words or POS tags of head and modifier, the length and

$w_h w_m, p_h p_m, w_h p_m, p_h w_m$
$p_{h-1} p_m, p_{h-1} w_m, p_h p_{m-1}, w_h p_{m-1}$
$p_{h+1} p_m, p_{h+1} w_m, p_h p_{m+1}, w_h p_{m+1}$
$p_{h-1} p_h p_m, p_h p_{h+1} p_m, p_h p_{m-1} p_m, p_h p_m p_{m+1}$
Concatenate features above with length and direction
$p_h p_b p_m$

Table 1: Basic features in polynomial and dependency tree kernel spaces, w_h: the word of head node, w_m denotes the word of modifier node, p_h: the POS of head node, p_m denotes the POS of modifier node, p_{h+1}: POS to the right of head node, p_{h-1}: POS to the left of modifier node, p_{m+1}: POS to the right of head node, p_{m-1}: POS to the left of modifier node, p_b: POS of a word in between head and modifier nodes.

direction of the arcs, and the POS tags of the words lying between the head and modifier, as shown in Table 1. The POS tags are automatically generated by 10 fold cross validation during training, and a POS tagger trained using the full training data during testing which has an accuracy of 96.9% on the development data and 97.3% on the test data.

As kernel methods are not scalable for large datasets, we applied the strategy proposed by Collins and Duffy (2002), to break the training set into 10 chunks of roughly equal size, and trained 10 separate kernel perceptrons on these data sets. The outputs from the 10 runs on test examples were combined through the voting procedure.

For feature selection, we set the maximum iteration number $M = 100$. We use the first order and second order features for pre-training. We choose the constant step size $\alpha^t = 1$ because we find this could quickly reduce the prediction error in very few iterations.

We use the SHA-1 hash function to generate the hash codes for the spectral bloom filter. The SHA-1 hash function produces a 160-bit hash code for each candidate feature. The hash code is then segmented into 5 segments, in this way we get five hash codes $h_1, \ldots, h_5$. Each code has 32 bits. Then we create 2^{32}(4G) counters. The threshold θ is set to 3, thus each counter requires 2 bits to store the counts. The spectral bloom filter costs $1G$ memory in total.

Furthermore, to reduce memory cost, we save the local data structure such as the selected features in Step 15 of Algorithm 1 whenever possible, and load them into the memory when needed.

After feature selection, we did not use the L_1 SVM for testing, instead, we trained an averaged perceptron with the selected features. Because we find that the averaged perceptron significantly outperforms L_1 SVM.

System	UAS	Training Time
Third Order Parser	93.07	20 hrs
Quadratic Kernel(QK)	93.41	6 hrs
Biquadratic Kernel(BK)	93.45	6 hrs
8-th Degree Polynomial Kernel(8K)	93.27	6 hrs
Dependency Tree Kernel (DTK)	93.65	10 days
LM with Template Features	93.39	4 mins
LM with Features in QK	93.39	9 mins
LM with Features in BK	93.44	0.5 hrs
LM with Features in 8K	93.30	6 hrs
LM with Features in DTK	**93.72**	36 hrs
(Zhang and McDonald, 2014)	93.57	N/A
(Zhang et al., 2013)	93.50	N/A
(Ma and Zhao, 2012)	93.40	N/A
(Bohnet and Kuhn, 2012)	93.39	N/A
(Rush and Petrov, 2012)	93.30	N/A
(Qian and Liu, 2013)	93.17	N/A
(Hayashi et al., 2013)	93.12	1 hr
(Martins et al., 2013)	93.07	N/A
(Zhang and McDonald, 2012)	93.06	N/A
(Koo and Collins, 2010)	93.04	N/A
(Zhang and Nivre, 2011)	92.90	N/A

Table 2: Comparison between our system and the state-of-art systems on English dataset. LM is short for Linear Model, hrs, mins are short for hours and minutes respectively

6.1.2 Results

Experimental results are listed in Table 2, all systems run on a 64 bit Fedora operation system with a single Intel core i7 3.40GHz and 32G memory. We also include results of representative state-of-the art systems.

It is clear that the use of kernels or the deep features in kernel spaces significantly improves the baseline third order parser and outperforms the reranking model with shallow, template-generated features. Besides, our feature selection outperforms kernel methods in both efficiency and accuracy.

It is unsurprising that the dependency tree kernel outperforms polynomial kernels, because it captures the structured information. For example, polynomial kernels can not distinguish the grand-child feature or sibling feature from the combination of two separated arc features.

When no additional resource is available, our parser achieved the best reported performance 93.72% UAS on English PTB dataset. It is

C	#Feat	#Template	Hours	Mem(G)	UAS
1	$0.34G$	N/A	stalled	OOM	N/A
2	$0.34G$	N/A	stalled	OOM	N/A
3	$33.1M$	$11.4K$	36	4.0	93.72
5	$6.32M$	$2.1K$	20	2.2	93.55
10	$2.10M$	$1.6K$	5	1.4	93.40

Table 3: Feature selection in dependency kernel space with different threshold C.

worth pointing that our method is orthogonal to other reported systems that benefit from advanced inference algorthms, such as cube pruning (Zhang and McDonald, 2014), AD^3 (Martins et al., 2013), etc. We believe that combining our techniques with others' will achieve further improvement.

Reranking the candidate parse trees of 2416 testing sentences takes 67 seconds, about 36 sentences per second.

To further understand the complexity of our algorithm, we perform feature selection in dependency tree kernel space with different thresholds C and record the number of selected features and feature templates, the speed and memory cost. Table 3 shows the results. We can see that our algorithm works efficiently when $C \geq 3$, but for $C < 3$, the number of selected features grows drastically, and the program runs out of memory (OOM).

6.2 Experimental Results on CoNLL 2009 Dataset

Now we looked at the impact of our system on non-English treebanks. We evaluate our system on six other languages from the CoNLL 2009 shared-task. We used the best setting in the previous experiment: reranking model is trained using the features selected in the dependency tree kernel space. For POS tag features we used the predicted tags.

As the third order parser can not handle non-projective parse trees, we used the graph transformation techniques to produce non-projective structures (Nivre and Nilsson, 2005). First, the training data for the parser is projectivized by applying a number of lifting operations (Kahane et al., 1998) and encoding information about these lifts in arc labels. We used the path encoding scheme where the label of each arc is concatenated with two binary tags, one indicates if the arc is lifted, the other indicates if the arc is along the lifting path from the syntactic to the linear head. Then we train a projective parser on the transformed data without arc label information and a classifier to predict the arc labels based on the projectivized gold parse tree structure. During testing, we run the parser and the classifier in a pipeline to generate a labeled parse tree. Labeled syntactic accuracy is reported for comparison.

Language	Ours	Official Best
Chinese	76.77	**79.17**
Japanese	**92.68**	92.57
German	87.40	**87.48**
Spanish	**87.82**	87.64
Czech	**80.51**	80.38
Catalan	86.98	**87.86**

Table 4: Experimental Results on CoNLL 2009 non-English datasets.

Comparison results are listed in Table 4. We achieved the best reported results on three languages, Japanese, Spanish and Czech. Note that CoNLL 2009 also provide the semantic labeling annotation which we did not used in our system. While some official systems benefit from jointly learning parsing and semantic role labeling models.

7 Conclusion

In this paper we proposed a new feature selection algorithm that selects features in kernel spaces in a coarse to fine order. Like frequent mining, the efficiency of our approach comes from the anti-monotone property of the subgradients. Experimental results on the English dependency parsing task show that our approach outperforms standard kernel methods. In the future, we would like to extend our technique to other real valued kernels such as the string kernels and tagging kernels.

Acknowledgments

We thank three anonymous reviewers for their valuable comments. This work is partly supported by NSF award IIS-0845484 and DARPA under Contract No. FA8750-13-2-0041. Any opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Sakamoto, and Setsuo Arikawa. 2002. Efficient substructure discovery from large semi-structured data. In *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*, pages 158–174.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July.

Bernd Bohnet and Jonas Kuhn. 2012. The best of bothworlds – a graph-based completion model for transition-based parsers. In *Proc. of EACL*.

S. Boyd and A. Mutapcic. 2006. Subgradient methods. *notes for EE364*.

Saar Cohen and Yossi Matias. 2003. Spectral bloom filters. In *Proc. of SIGMOD*, SIGMOD '03.

Michael Collins and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proc. of ACL*, ACL '02.

Aron Culotta and Jeffrey Sorensen. 2004. Dependency tree kernels for relation extraction. In *Proc. of ACL*, ACL '04.

Katsuhiko Hayashi, Shuhei Kondo, and Yuji Matsumoto. 2013. Efficient stacked dependency parsing by forest reranking. *TACL*, 1.

Sylvain Kahane, Alexis Nasr, and Owen Rambow. 1998. Pseudo-projectivity, a polynomially parsable non-projective dependency grammar. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*, pages 646–652, Montreal, Quebec, Canada, August. Association for Computational Linguistics.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proc. of ACL*.

Taku Kudo, Jun Suzuki, and Hideki Isozaki. 2005. Boosting-based parse reranking with subtree features. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 189–196, Ann Arbor, Michigan, June. Association for Computational Linguistics.

Zhiyun Lu, Avner May, Kuan Liu, Alireza Bagheri Garakani, Dong Guo, Aurélien Bellet, Linxi Fan, Michael Collins, Brian Kingsbury, Michael Picheny, and Fei Sha. 2014. How to scale up kernel methods to be as good as deep neural nets. *CoRR*, abs/1411.4000.

Xuezhe Ma and Hai Zhao. 2012. Fourth-order dependency parsing. In *Proceedings of COLING 2012: Posters*, pages 785–796, Mumbai, India, December. The COLING 2012 Organizing Committee.

Andre Martins, Miguel Almeida, and Noah A. Smith. 2013. Turning on the turbo: Fast third-order non-projective turbo parsers. In *Proc. of ACL*.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 99–106, Stroudsburg, PA, USA. Association for Computational Linguistics.

Daisuke Okanohara and Jun'ichi Tsujii. 2009. Learning combination features with l1 regularization. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Short Papers*, pages 97–100, Boulder, Colorado, June. Association for Computational Linguistics.

Xian Qian and Yang Liu. 2013. Branch and bound algorithm for dependency parsing with non-local features. *TACL*, 1.

Alexander Rush and Slav Petrov. 2012. Vine pruning for efficient multi-pass dependency parsing. In *Proc. of NAACL*. Association for Computational Linguistics.

Jun Suzuki, Hideki Isozaki, and Eisaku Maeda. 2004. Convolution kernels with feature selection for natural language processing tasks. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 119–126, Barcelona, Spain, July.

Mingkui Tan, Ivor W. Tsang, and Li Wang. 2012. Towards large-scale and ultrahigh dimensional feature selection via feature generation. *CoRR*, abs/1209.5260.

Ben Taskar. 2004. *Learning Structured Prediction Models: A Large Margin Approach*. Ph.D. thesis, Stanford University.

Christopher K. I. Williams and Matthias Seeger. 2001. Using the nyström method to speed up kernel machines. In *NIPS*.

Yu-Chieh Wu, Jie-Chi Yang, and Yue-Shi Lee. 2007. An approximate approach for training polynomial kernel svms in linear time. In *Proc. of ACL*, ACL '07.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proc. of IWPT*.

Mohammed J. Zaki. 2002. Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 71–80, New York, NY, USA. ACM.

Hao Zhang and Ryan McDonald. 2012. Generalized higher-order dependency parsing with cube pruning. In *Proc. of EMNLP*.

Hao Zhang and Ryan McDonald. 2014. Enforcing structural diversity in cube-pruned dependency parsing. In *Proc. of ACL*.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proc. of ACL-HLT*.

Qi Zhang, Fuliang Weng, and Zhe Feng. 2006. A progressive feature selection algorithm for ultra large feature spaces. In *Proc. of ACL*.

Hao Zhang, Liang Huang, Kai Zhao, and Ryan McDonald. 2013. Online learning for inexact hypergraph search. In *Proc. of EMNLP*, pages 908–913. Association for Computational Linguistics.