A Unified Kernel Approach for Learning Typed Sentence Rewritings

Martin Gleize
LIMSI-CNRS, Orsay, France
Université Paris-Sud, Orsay, France
gleize@limsi.fr

Brigitte Grau
LIMSI-CNRS, Orsay, France
ENSIIE, Evry, France
bg@limsi.fr

Abstract

Many high level natural language processing problems can be framed as determining if two given sentences are a rewriting of each other. In this paper, we propose a class of kernel functions, referred to as type-enriched string rewriting kernels, which, used in kernel-based machine learning algorithms, allow to learn sentence rewritings. Unlike previous work, this method can be fed external lexical semantic relations to capture a wider class of rewriting rules. It also does not assume preliminary syntactic parsing but is still able to provide a unified framework to capture syntactic structure and alignments between the two sentences. We experiment on three different natural sentence rewriting tasks and obtain state-of-the-art results for all of them.

1 Introduction

Detecting implications of sense between statements stands as one of the most sought-after goals in computational linguistics. Several high level tasks look for either one-way rewriting between single sentences, like recognizing textual entailment (RTE) (Dagan et al., 2006), or two-way rewritings like paraphrase identification (Dolan et al., 2004) and semantic textual similarity (Agirre et al., 2012). In a similar fashion, selecting sentences containing the answer to a question can be seen as finding the best rewritings of the question among answer candidates. These problems are naturally framed as classification tasks, and as such most current solutions make use of supervised machine learning. They have to tackle several challenges: picking an adequate language representation, aligning semantically equivalent elements and extracting relevant features to learn the final decision. Bag-of-words and by extension bag-of-ngrams are traditionally the most direct approach and features rely mostly on lexical matching (Wan et al., 2006; Lintean and Rus, 2011; Jimenez et al., 2013). Moreover, a good solving method has to account for typically scarce labeled training data, by enriching its model with lexical semantic resources like WordNet (Miller, 1995) to bridge gaps between surface forms (Mihalcea et al., 2006; Islam and Inkpen, 2009; Yih et al., 2013). Models based on syntactic trees remain the typical choice to account for the structure of the sentences (Heilman and Smith, 2010; Wang and Manning, 2010; Socher et al., 2011; Calvo et al., 2014). Usually the best systems manage to combine effectively different methods, like Madnani et al.'s meta-classifier with machine translation metrics (Madnani et al., 2012).

A few methods (Zanzotto et al., 2007; Zanzotto et al., 2010; Bu et al., 2012) use kernel functions to learn what makes two sentence pairs similar. Building on this work, we present a type-enriched string rewriting kernel giving the opportunity to specify in a fine-grained way how words match each other. Unlike previous work, rewriting rules learned using our framework account for syntactic structure, term alignments and lexico-semantic typed variations in a unified approach. We detail how to efficiently compute our kernel and lastly experiment on three different high-level NLP tasks, demonstrating the vast applicability of our method. Our system based on type-enriched string rewriting kernels obtains state-of-the-art results on paraphrase identification and answer sentence selection and outperforms comparable methods on RTE.

2 Type-Enriched String Rewriting Kernel

Kernel functions measure the similarity between two elements. Used in machine learning methods like SVM, they allow complex decision functions to be learned in classification tasks (Vapnik, 2000). The goal of a well-designed kernel function is to have a high value when computed on two instances of same label, and a low value for two instances of different label.

2.1 String rewriting kernel

String rewriting kernels (Bu et al., 2012) count the number of common rewritings between two pairs of sentences seen as sequences of words. The rewriting rule (A) in Figure 1 can be viewed as a kind of phrasal paraphrase with linked variables (Madnani and Dorr, 2010). Rule (A) rewrites (B)'s first sentence into its second but it does not however rewrite the sentences in (C), which is what we try to fix in this paper.

Following the terminology of string kernels, we use the term *string* and *character* instead of *sentence* and *word*. We denote $(s, t) \in (\Sigma^* \times \Sigma^*)$ an instance of string rewriting, with a source string s and a target string t, both finite sequences of elements in Σ the finite set of characters. Suppose that we are given training data of such instances labeled in $\{+1, -1\}$, for paraphrase/non-paraphrase or entailment/non-entailment in applications. We can use a kernel method to train on this data and learn to automatically classify unlabeled instances. A kernel on string rewriting instances is a map:

$$K : (\Sigma^* \times \Sigma^*) \times (\Sigma^* \times \Sigma^*) \to \mathbb{R}$$

such that for all $(s_1, t_1), (s_2, t_2) \in \Sigma^* \times \Sigma^*$,

$$K((s_1, t_1), (s_2, t_2)) = \langle \Phi(s_1, t_1), \Phi(s_2, t_2) \rangle \quad (1)$$

where Φ maps each instance into a high dimension feature space. Kernels allow us to avoid the potentially expensive explicit representation of Φ through the inner product space they define. The purpose of the string rewriting kernels is to measure the similarity between two pairs of strings in term of the number of rewriting rules of a set R that they share. Φ is thus naturally defined by $\Phi(s, t) = (\phi_r(s, t))_{r \in R}$ with $\phi_r(s, t) = n$ the number of contiguous substring pairs of (s, t) that rewriting rule r matches.

2.2 Typed rewriting rules

Let the wildcard domain $D \subseteq \Sigma^*$ be the set of strings which can be replaced by wildcards. We now present the formal framework of the type-enriched string rewriting kernels.

Let Γ_p be the set of *pattern types* and Γ_v the set of *variable types*.

To a type $\gamma_p \in \Gamma_p$, we associate the *typing relation* $\overset{\gamma_p}{\approx} \subseteq \Sigma \times \Sigma$.

To a type $\gamma_v \in \Gamma_v$,we associate the *typing relation* $\overset{\gamma_v}{\rightsquigarrow} \subseteq D \times D$.

Together with the typing relations, we call the association of Γ_p and Γ_v the *typing scheme* of the kernel. Let Σ_p be defined as

$$\Sigma_p = \bigcup_{\gamma \in \Gamma} \{[a|b] \mid \exists a, b \in \Sigma, a \overset{\gamma}{\approx} b\} \quad (2)$$

We finally define typed rewriting rules. A *typed rewriting rule* is a triple $r = (\beta_s, \beta_t, \tau)$, where $\beta_s, \beta_t \in (\Sigma_p \cup \{*\})^*$ denote source and target string typed patterns and $\tau \subseteq ind_*(\beta_s) \times ind_*(\beta_t)$ denotes the alignments between the wildcards in the two string patterns. Here $ind_*(\beta)$ denotes the set of indices of wildcards in β.

We say that a rewriting rule (β_s, β_t, τ) *matches* a pair of strings (s, t) if and only if the following conditions are true:

- string patterns β_s, resp. β_t, can be turned into s, resp. t, by:
  - substituting each element $[a|b]$ of Σ_p in the string pattern with an a or b ($\in \Sigma$)
  - substituting each wildcard in the string pattern with an element of the wildcard domain D
- $\forall (i, j) \in \tau$, s, resp. t, substitutes the wildcards at index i, resp. j, by $s_* \in D$, resp. t_*, such that there exists a variable type $\gamma \in \Gamma_v$ with $s_* \overset{\gamma}{\rightsquigarrow} t_*$.

A *type-enriched string rewriting kernel* (TESRK) is simply a string rewriting kernel as defined in Equation 1 but with R a set of typed rewriting rules. This class of kernels depends on wildcard domain D and the typed rewriting rules R which can be tuned to allow for more flexibility in the matching of pairs of characters in a rewriting rule. Within this framework, the k-gram bijective string rewriting kernel (kb-SRK) is defined by the wildcard domain $D = \Sigma$ and the ruleset

$$R = \{(\beta_s, \beta_t, \tau) \mid \beta_s, \beta_t \in (\Sigma_p \cup \{*\})^k, \tau \text{ bijective}\}$$

under $\Gamma_p = \Gamma_v = \{id\}$ with $a \overset{id}{\approx} b$, resp. $a \overset{id}{\rightsquigarrow} b$, if and only if $a = b$.

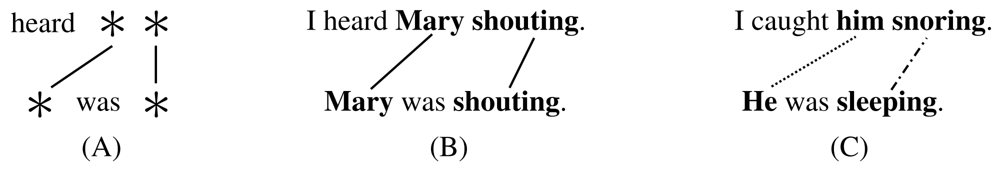

Figure 1: Rewriting rule (A) matches pair of strings (B) but does not match (C).

We now present an example of how kb-SRK is applied to real pairs of sentences, what its limitations are and how we can deal with them by reworking its typing scheme. Let us consider again Figure 1, (A) is a rewriting rule with $\beta_s = ($*heard*, $*, *)$, $\beta_t = (*,$ *was*, $*)$, $\tau = \{(2,1);(3,3)\}$. Each string pattern has the same length, and pairs of wildcards in the two patterns are aligned bijectively. This is a valid rule for kb-SRK. It matches the pair of strings (B): each aligned pair of wildcards is substituted in source and target sentences by the same word and string patterns of (A) can indeed be turned into pairs of substrings of the sentences. However, it cannot match the pair of sentences (C) in the original kb-SRK. We change Γ_p to $\{$hypernym, id$\}$ where $a \overset{hypernym}{\approx} b$ if and only if a and b have a common hypernym in WordNet. And we change Γ_v to $\Gamma_v = \{$same_pronoun, entailment, id$\}$ where $a \overset{same\_pronoun}{\rightsquigarrow} b$ if and only if a and b are a pronoun of the same person and same number, and $a \overset{entailment}{\rightsquigarrow} b$ if and only if verb a has a relation of entailment with b in WordNet.
By redefining the typing scheme, rule (A) can now match (C).

3 Computing TESRK

3.1 Formulation

The k-gram bijective string rewriting kernel can be computed efficiently (Bu et al., 2012). We show that we can compute its type-enriched equivalent at the price of a seemingly insurmountable loosening of theoretical complexity boundaries. Experiments however show that its computing time is of the same order as the original kernel.
A type-enriched kb-SRK is parameterized by k the length of k-grams, and its typing scheme the sets Γ_p and Γ_v and their associated relations. The annotations of Γ_p and Γ_v to K_k and $\bar{K}_k$ will be omitted for clarity and because they typically will not change while we test different values for k.
We rewrite the inner product in Equation 1 to better fit the k-gram framework:

$$K_k((s_1,t_1),(s_2,t_2)) = \sum_{\substack{\alpha_{s_1}\in\text{k-grams}(s_1)\\ \alpha_{t_1}\in\text{k-grams}(t_1)}} \sum_{\substack{\alpha_{s_2}\in\text{k-grams}(s_2)\\ \alpha_{t_2}\in\text{k-grams}(t_2)}} \bar{K}_k((\alpha_{s_1},\alpha_{t_1}),(\alpha_{s_2},\alpha_{t_2})) \tag{3}$$

where $\bar{K}_k$ is the number of different rewriting rules which match two pairs of k-grams (the same rule cannot trigger twice in k-gram substrings):

$$\bar{K}_k((\alpha_{s_1},\alpha_{t_1}),(\alpha_{s_2},\alpha_{t_2})) = \sum_{r\in R} \mathbb{1}_r(\alpha_{s_1},\alpha_{t_1})\mathbb{1}_r(\alpha_{s_2},\alpha_{t_2}) \tag{4}$$

with $\mathbb{1}_r$ the indicator function of rule r: 1 if r matches the pair of k-grams, 0 otherwise.
Computing K_k as defined in Equation 3 is obviously intractable. There is $\mathcal{O}((n-k+1)^4)$ terms in the sum, where n is the length of the longest string, and each term involves enumerating every rewriting rule in R.

3.2 Computing $\bar{K}_k$ in type-enriched kb-SRK

Enumerating all rewriting rules in Equation 4 is itself intractable: there are more than $|\Sigma|^{2k}$ rules without wildcards, where $|\Sigma|$ is conceivably the size of a typical lexicon. In fact, we just have to constructively generate the rules which substitute their string patterns correctly to simultaneously produce both pairs of k-grams $(\alpha_{s_1},\alpha_{t_1})$ and $(\alpha_{s_2},\alpha_{t_2})$.

Let the operator $\otimes$ be such that $\alpha_1 \otimes \alpha_2 = ((\alpha_1[1],\alpha_2[1]),...,(\alpha_1[k],\alpha_2[k]))$. This operation is generally known as *zipping* in functional programming. We use the function *CountPerfectMatchings* computed by Algorithm 1 to recursively count the number of rewriting rules matching both $(\alpha_{s_1},\alpha_{t_1})$ and $(\alpha_{s_2},\alpha_{t_2})$. The workings of the algorithm will make clearer why we can compute $\bar{K}_k$ with the following formula:

$$\bar{K}_k((\alpha_{s_1},\alpha_{t_1}),(\alpha_{s_2},\alpha_{t_2})) = \text{CountPerfectMatchings}(\alpha_{s_1}\otimes\alpha_{s_2},\alpha_{t_1}\otimes\alpha_{t_2}) \tag{5}$$

Algorithm 1 takes as input remaining character pairs in $\alpha_{s_1} \otimes \alpha_{s_2}$ and $\alpha_{t_1} \otimes \alpha_{t_2}$, and outputs the number of ways they can substitute aligned wildcards in a matching rule.
First (lines 2 and 3) we have the base case where both remaining sets are empty. There is exactly 1 way the empty set's wildcards can be aligned with each other: nothing is aligned. In lines 4 to 9, there is no source pairs anymore, so the algorithm continues to deplete target pairs as long as they have a common pattern type, i.e. as long as they do not have to substitute a wildcard. If a candidate wildcard is found, as the opposing set is empty, we cannot align it and we return 0. In the general case (lines 11 to 19), consider the first character pair (a_1, a_2) in the reminder of $\alpha_{s_1} \otimes \alpha_{s_2}$ in line 12. What follows in the computation depends on its types. Every character pair in $\alpha_{t_1} \otimes \alpha_{t_2}$ that can be paired through variable types with (a_1, a_2) (lines 15 to 19) is a new potential wildcard alignment, so we try all the possible alignment and recursively continue the computation after removing both aligned pairs. And if (a_1, a_2) does not need to substitute a wildcard because it has common pattern types (lines 13 and 14), we can choose to not create any wildcard pairing with it and ignore it in the recursive call.
This algorithm enumerates all configurations such that each character pair has a common pattern type or is matched 1-for-1 with a character pair with common variable types, which is exactly the definition of a rewriting rule in TESRK.

This problem is actually equivalent to counting the perfect matchings of the bipartite graph of potential wildcards. It has been shown intractable (Valiant, 1979) and Algorithm 1 is a naive recursive algorithm to solve it. In our implementation we represent the graph with its bi-adjacency matrix, and if our typing relations are independent of k, the function has a $\mathcal{O}(k)$ time complexity without including its recursive calls. The number of recursive calls can be greater than $k!^2$ which is the number of perfect matchings in a complete bipartite graph of $2k$ vertices. In our experiments on linguistic data however, we observed a linear number of recursive calls for low values of k, and up to a quadratic number for $k > 10$ –which is way past the point where the kernel becomes ineffective.

As an example, Figure 2 shows the zipped k-grams for source and target as a bipartite graph

Algorithm 1: Counting perfect matchings

```
1  CountPerfectMatchings(remS, remT)
   Data: remS: remaining char. pairs in source
   remT: remaining char. pairs in target
   graph: $\alpha_{s_1} \otimes \alpha_{s_2}$ and $\alpha_{t_1} \otimes \alpha_{t_2}$ as a bipartite
   graph, not added in the arguments to avoid
   cluttering the recursive calls
   ruleSet: $\Gamma_p$ and $\Gamma_v$
   Result: Number of rewriting rules matching
           $(\alpha_{s_1}, \alpha_{t_1})$ and $(\alpha_{s_2}, \alpha_{t_2})$
2  if remS == $\emptyset$ and remT == $\emptyset$ then
3      return 1;
4  else if remS == $\emptyset$ then
5      $(b_1, b_2)$ = remT.first();
6      if $\exists \gamma \in \Gamma_p \mid b_1 \overset{\gamma}{\approx} b_2$ then
7          return CountPerfectMatchings($\emptyset$,
           remT - $\{(b_1, b_2)\}$);
8      else
9          return 0;
10 else
11     result = 0;
12     $(a_1, a_2)$ = remS.first();
13     if $\exists \gamma \in \Gamma_p \mid a_1 \overset{\gamma}{\approx} a_2$ then
14         res += CountPerfectMatchings(remS -
           $\{(a_1, a_2)\}$, remT);
15     for $(b_1, b_2) \in$ remT
16       $\mid \exists \gamma \in \Gamma_v \mid a_1 \overset{\gamma}{\rightsquigarrow} b_1$ and $a_2 \overset{\gamma}{\rightsquigarrow} b_2$ do
17         res += CountPerfectMatchings(
18             remS - $\{(a_1, a_2)\}$,
19             remT - $\{(b_1, b_2)\}$
20         );
```

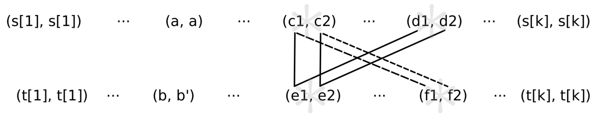

Figure 2: Bipartite graph of character pairs, with edges between potential wildcards

with $2k$ vertices and potential wildcard edges. Assuming that vertices (a, a) and (b, b') have common pattern types, they can be ignored as in lines 7 and 14. (c_1, c_2) to (f_1, f_2) however must substitute wildcards in a matching rewriting rule. If we align (c_1, c_2) with (e_1, e_2) in line 16, the recursive call will return 0 because the other two pairs cannot be aligned. A valid rule is generated if c's are paired with f's and d's with e's. This kind of choices is the main source of computational cost.

This problem did not arise in the original kb-SRK because of the transitivity of its only type (*identity*). In type-enriched kb-SRK, wildcard pairing is less constrained.

3.3 Computing K_k

Even with an efficient method for computing $\bar{K}_k$, implementing K_k directly by applying Equation 3 remains impractical. The main idea is to efficiently compute a reasonably sized set $\mathbb{C}$ of elements $((\alpha_{s_1}, \alpha_{t_1}), (\alpha_{s_2}, \alpha_{t_2}))$ which has the essential property of including all elements such that $\bar{K}_k((\alpha_{s_1}, \alpha_{t_1}), (\alpha_{s_2}, \alpha_{t_2})) \neq 0$.
By definition of $\mathbb{C}$, we can compute efficiently

$$K_k((s_1, t_1), (s_2, t_2)) = \sum_{((\alpha_{s_1}, \alpha_{s_2}), (\alpha_{t_1}, \alpha_{t_2})) \in \mathbb{C}} \bar{K}_k((\alpha_{s_1}, \alpha_{t_1}), (\alpha_{s_2}, \alpha_{t_2})) \tag{6}$$

There are a number of ways to do it, with a trade-off between computation time and number of elements in the reduced domain $\mathbb{C}$. The main idea of our own algorithm is that $\bar{K}_k((\alpha_{s_1}, \alpha_{t_1}), (\alpha_{s_2}, \alpha_{t_2})) = 0$ if the character pairs $(a_1, a_2) \in \alpha_{s_1} \otimes \alpha_{s_2}$ with no common pattern type are not **all** matched with pairs $(b_1, b_2) \in \alpha_{t_1} \otimes \alpha_{t_2}$ such that $a_1 \overset{\gamma}{\rightsquigarrow} b_1$ and $a_2 \overset{\gamma}{\rightsquigarrow} b_2$ for some $\gamma \in \Gamma_v$. This is conversely true for character pairs in $\alpha_{t_1} \otimes \alpha_{t_2}$ with no common pattern type. More simply, character pairs with no common pattern type are mismatched and have to substitute a wildcard in a rewriting rule matching both $(\alpha_{s_1}, \alpha_{t_1})$ and $(\alpha_{s_2}, \alpha_{t_2})$. But introducing a wildcard on one side of the rule means that there is a matching wildcard on the other side, so we can eliminate k-gram quadruples that do not fill this wildcard inclusion. This filtering can be done efficiently and yields a manageable number of quadruples on which to compute $\bar{K}_k$.

Algorithm 2 computes a set $\mathbb{C}$ to be used in Equation 6 for computing the final value of kernel K_k. In our experiments, it efficiently produces a reasonable number of inputs. All maps in the algorithm are maps to multisets, and multisets are used extensively throughout. *Multisets* are an extension of sets where elements can appear multiple times, the number of times being called the *multiplicity*. Typically implemented as hash tables from set elements to integers, they allow for constant-time retrieval of the number of a given element. Union ($\cup$) and intersection ($\cap$) have special definitions on multisets. If $\mathbb{1}_A(x)$ is the multiplicity of x in A, we have $\mathbb{1}_{A \cup B}(x) = max(\mathbb{1}_A(x), \mathbb{1}_B(x))$ and $\mathbb{1}_{A \cap B}(x) = min(\mathbb{1}_A(x), \mathbb{1}_B(x))$.

Algorithm 2: Computing a set including all elements on which $\bar{K}_k \neq 0$

```
Data: $s_1, t_1, s_2, t_2$ strings, and $k$ an integer
Result: Set $\mathbb{C}$ which include all inputs such
        that $\bar{K}_k \neq 0$
1  Initialize maps $e^i_{s \to t}$ and maps $e^i_{t \to s}$, for
   $i \in \{1, 2\}$;
2  for $i \in \{1, 2\}$ do
3      for $a \in s_i, b \in t_i \mid a \overset{\gamma}{\rightsquigarrow} b, \gamma \in \Gamma_v$ do
4          $e^i_{s \to t}[a]$ += $(b, \gamma)$; $e^i_{t \to s}[b]$ += $(a, \gamma)$;
5  $w_{s \to t}, aP_t$ =
   OneWayInclusion($s_1, s_2, t_1, t_2, e^1_{s \to t}, e^2_{s \to t}$);
6  $w_{t \to s}, aP_s$ =
   OneWayInclusion($t_1, t_2, s_1, s_2, e^1_{t \to s}, e^2_{t \to s}$);
7  Initialize multiset res;
8  for $(\alpha_{s_1}, \alpha_{s_2}) \in aP_s$ do
9      for $(\alpha_{t_1}, \alpha_{t_2}) \in aP_t$ do
10         res += $((\alpha_{s_1}, \alpha_{s_2}), (\alpha_{t_1}, \alpha_{t_2}))$;
11 res = res $\cup w_{s \to t} \cup w_{t \to s}.map(swap)$;
12 return res;
13 ________________________________________
14 OneWayInclusion($s_1, s_2, t_1, t_2, e^1, e^2$)
   Initialize map $d$ multisets resWildcards,
   resAllPatterns;
15 for $(\alpha_{s_1}, \alpha_{s_2}) \in kgrams(s_1) \times kgrams(s_2)$ do
16     for $(b_1, b_2) \mid \exists \gamma \in \Gamma_v, (a_1, a_2) \in$
       $\alpha_{s_1} \otimes \alpha_{s_2}, (b_i, \gamma) \in e^i[a_i] \; \forall i \in \{1, 2\}$ do
17         $d[(b_1, b_2)]$ += $(\alpha_{s_1}, \alpha_{s_2})$;
18 for $(\alpha_{t_1}, \alpha_{t_2}) \in kgrams(t_1) \times kgrams(t_2)$ do
19     for $(b_1, b_2) \in \alpha_{t_1} \otimes \alpha_{t_2} \mid b_1 \overset{\gamma}{\not\rightsquigarrow} b_2 \forall \gamma \in \Gamma_p$
       do
20         if compatWkgrms not initialized then
21             Initialize multiset compatWkgrms
               = $d[(b_1, b_2)]$;
22         compatWkgrms = compatWkgrms
           $\cap \; d[(b_1, b_2)]$;
23     if compatWkgrms not initialized then
24         resAllPatterns += $(\alpha_{t_1}, \alpha_{t_2})$;
25     for $(\alpha_{s_1}, \alpha_{s_2}) \in$ compatWkgrms do
26         resWildcards+=$((\alpha_{s_1}, \alpha_{s_2}), (\alpha_{t_1}, \alpha_{t_2}))$;
27 return (resWildcards, resAllPatterns);
```

Let us now comment on how the algorithm unfolds. In lines 1 to 4, we index characters in source strings by characters in target strings which have common variable types, and vice versa. It allows in lines 15 to 19 to quickly map a character pair to the set of opposing k-gram pairs with a matching –in the sense of variable types– character pair, i.e. potential aligned wildcards. In lines 20 to 28 we keep only the k-gram quadruples whose wildcard candidates (character pairs with no common pattern) from one side **all** find matches on the other side. We do not check for the other inclusion, hence the name of the function *OneWayInclusion*. At line 26, we did not find any character pair with no common pattern, so we save the k-gram pair as "all-pattern". All-pattern k-grams will be paired in lines 8 to 10 in the result. Finally, in line 11, we add the union of one-way compatible k-gram quadruples; calling swap on all the pairs of one set is necessary to consistently have sources on the left side and targets on the right side in the result.

4 Experiments

4.1 Systems

We experimented on three tasks: paraphrase identification, recognizing textual entailment and answer sentence selection. The setup we used for all experiments was the same save for the few parameters we explored such as: k, and typing scheme. We implemented 2 kernels, kb-SRK, henceforth simply denoted *SRK*, and the type-enriched kb-SRK, denoted *TESRK*. All sentences were tokenized and POS-tagged using OpenNLP (Morton et al., 2005). Then they were stemmed using the Porter stemmer (Porter, 2001) in the case of SRK. Various other pre-processing steps were applied in the case of TESRK: they are considered as types in the model and are detailed in Table 1. We used LIBSVM (Chang and Lin, 2011) to train a binary SVM classifier on the training data with our two kernels. The default SVM algorithm in LIBSVM uses a parameter C, roughly akin to a regularization parameter. We 10-fold cross-validated this parameter on the training data, optimizing with a grid search for f-score, or MRR for question-answering. All kernels were normalized using $\tilde{K}(x,y) = \frac{K(x,y)}{\sqrt{K(x,x)}\sqrt{K(y,y)}}$. We denote by "+" a sum of kernels, with normalizations applied both before and after summing. Following Bu et al. (Bu et al., 2012) experimental setup, we introduced an auxiliary vector kernel denoted *PR* of features named *unigram precision* and *recall*, defined in (Wan et al., 2006). In our experiments a linear kernel seemed to yield the best results. Our Scala implementation of kb-SRKs has an average throughput of about 1500 original kb-SRK computations per second, versus 500 type-enriched kb-SRK computations per second on a 8-core machine. It typically takes a few hours on a 32-core machine to train, cross-validate and test on a full dataset.

Finally, Table 1 presents an overview of our types with how they are defined and implemented. Every type can be used both as a pattern type or as a variable type, but the two roles are different. Pattern types are useful to unify different surface forms of rewriting rules that are semantically equivalent, i.e. having semantically similar patterns. Variable types are useful for when the semantic relation between 2 entities across the same rewriting is more important than the entities themselves. That is why some types in Table 1 are inherently more fitted to be used for one role rather than the other. For example, it is unlikely that replacing a word in a pattern of a rewriting rule by one of its holonyms will yield a semantically similar rewriting rule, so *holonym* would not be a good pattern type for most applications. On the contrary, it can be very useful in a rewriting rule to type a wildcard link with the relation holonym, as this provides constrained semantic roles to the linked wildcards in the rule, thus *holonym* would be a good variable type.

4.2 Paraphrase identification

Paraphrase identification asks whether two sentences have the same meaning. The dataset we used to evaluate our systems is the MSR Paraphrase Corpus (Dolan and Brockett, 2005), containing 4,076 training pairs of sentences and 1,725 testing pairs. For example, the sentences *"An injured woman co-worker also was hospitalized and was listed in good condition."* and *"A woman was listed in good condition at Memorial's HealthPark campus, he said."* are paraphrases in this corpus. On the other hand, *"'There are a number of locations in our community, which are essentially vulnerable,' Mr Ruddock said."* and *"'There are a range of risks which are being seriously examined by competent authorities,' Mr Ruddock said."* are not paraphrases.

We report in Table 2 our best results, the system *TESRK + PR*, defined by the sum of PR and typed-enriched kb-SRKs with k from 1 to 4, with types $\Gamma_p = \Gamma_v = \{stem, synonym\}$. We observe

Type	Typing relation on words (a, b)	Tool/resources
id	words have same surface form and tag	OpenNLP tagger
idMinusTag	words have same surface form	OpenNLP tokenizer
lemma	words have same lemma	WordNetStemmer
stem	words have same stem	Porter stemmer
synonym, antonym	words are [type]	WordNet
hypernym, hyponym	b is a [type] of a	WordNet
entailment, holonym		
ne	a and b are both tagged with the same Named Entity	BBN Identifinder
lvhsn	words are at edit distance of 1	Levenshtein distance

Table 1: Types

Paraphrase system	Accuracy	F-score
All paraphrase	66.5	79.9
Wan et al. (2006)	75.6	83.0
Bu et al. (2012)	76.3	N/A
Socher et al. (2011)	76.8	83.6
Madnani et al. (2012)	**77.4**	**84.1**
PR	73.5	82.1
SRK + PR	76.2	83.6
TESRK	76.6	83.7
TESRK + PR	**77.2**	**84.0**

Table 2: Evaluation results on MSR Paraphrase

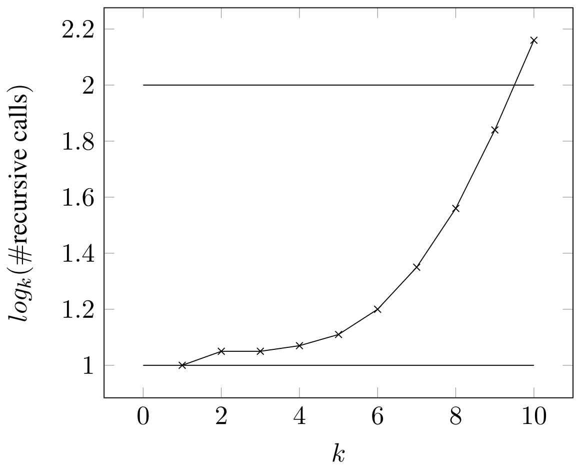

Figure 3: Evolution of the number of recursive calls to CountPerfectMatchings with k

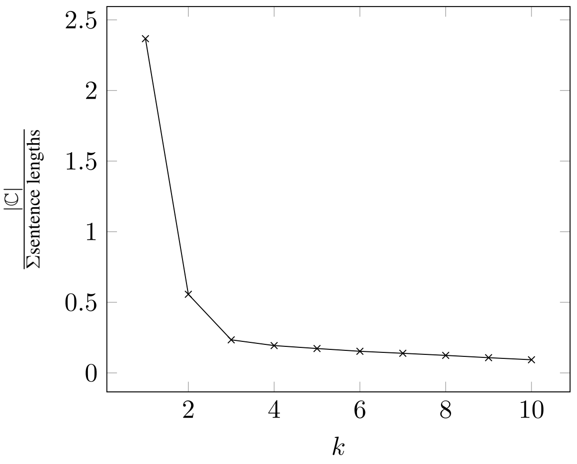

Figure 4: Evolution of the size of $\mathbb{C}$ with k

that our results are state-of-the-art and in particular, they improve on the orignal kb-SRK by a good margin. We tried other combinations of types but it did not yield good results, this is probably due to the nature of the MSR corpus, which did not contain much more advanced variations from WordNet. The only statistically significant improvement we obtained was between *TESRK + PR* and our PR baseline ($p < 0.05$). The performances obtained by all the cited systems and ours are not significantly different in any statistical sense. We made a special effort to try to reproduce as best as we could the original kb-SRK performances (Bu et al., 2012), although our implementation and theirs should theoretically be equivalent.

Figure 3 plots the average number of recursive calls to CountPerfectMatchings (algorithm 1) during a kernel computation, as a function of k. Composing with log_k, we can observe whether the empiric number of recursive calls is closer to $\mathcal{O}(k)$ or $\mathcal{O}(k^2)$. We conclude that this element of complexity is linear for low values of k, but tends to explode past $k = 7$. Thankfully, counting common rewriting rules on pairs of 7-to-10-grams rarely yields non-zero results, so in practice using high values of k is not interesting.

Figure 4 plots the average size of set $\mathbb{C}$ computed by algorithm 2, as a function of k (divided by the sum of lengths of the 4 sentences involved in the kernel computation). We can observe that this

RTE system	Accuracy
All entailments	51.2
Heilman and Smith (2010)	62.8
Bu et al. (2012)	65.1
Zanzotto et al. (2007)	65.8
Hickl et al. (2006)	**80.0**
PR	61.8
TESRK (All)	62.1
SRK + PR	63.8
TESRK (Syn) + PR	64.1
TESRK (All) + PR	66.1

Table 3: Evaluation results on RTE-3

quantity is small, except for a peak at low values of k, which is not an issue because the computation of $\bar{K}_k$ is very fast for those values of k.

4.3 Recognizing textual entailment

Recognizing Textual Entailment asks whether the meaning of a sentence *hypothesis* can be inferred by reading a sentence *text*. The dataset we used to evaluate our systems is RTE-3. Following similar work (Heilman and Smith, 2010; Bu et al., 2012), we took as training data (text, hypothesis) pairs from RTE-1 and RTE-2's whole datasets and from RTE-3's training data, which amounts to 3,767 sentence pairs. We tested on RTE-3 testing data containing 800 sentence pairs. For example, a valid textual entailment in this dataset is the pair of sentences *"In a move widely viewed as surprising, the Bank of England raised UK interest rates from 5% to 5.25%, the highest in five years."* and *"UK interest rates went up from 5% to 5.25%."*: the first entails the second. On the other hand, the pair *"Former French president General Charles de Gaulle died in November. More than 6,000 people attended a requiem mass for him at Notre Dame cathedral in Paris."* and *"Charles de Gaulle died in 1970."* does not constitute a textual entailment.

We report in Table 3 our best results, the system *TESRK (All) + PR*, defined by the sum of PR, 1b-SRK and typed-enriched kb-SRKs with k from 2 to 4, with types $\Gamma_p = \{$stem, synonym$\}$ and $\Gamma_v = \{$stem, synonym, hypernym, hyponym, entailment, holonym$\}$. Our results are to be compared with systems using techniques and resources of similar nature, but as reference the top performance at RTE-3 is still reported. This time we did not manage to fully reproduce Bu et al. 2012's performance, but we observe that type-enriched kb-SRK greatly improves upon our original implementation of kb-SRK and outperforms their system anyway. Combining TESRK and the PR baseline yields significantly better results than either one alone ($p < 0.05$), and performs significantly better than the system of (Heilman and Smith, 2010), the only one which was evaluated on the same three tasks as us ($p < 0.10$). We tried with less types in our system *TESRK (Syn) + PR* by removing all WordNet types but synonyms but got lower performance. This seems to indicate that rich types indeed help capturing more complex sentence rewritings. Note that we needed for $k = 1$ to replace the type-enriched kb-SRK by the original kernel in the sum, otherwise the performance dropped significantly. Our conclusion is that including richer types is only beneficial if they are captured within a context of a couple of words and that including all those variations on unigrams only add noise.

4.4 Answer sentence selection

Answer sentence selection is the problem of selecting among single candidate sentences the ones containing the correct answer to an open-domain factoid question. The dataset we used to evaluate our system on this task was created by (Wang et al., 2007) based on the QA track of past Text REtrieval Conferences (TREC-QA)[1]. The training set contains 4718 question/answer pairs, for 94 questions, originating from TREC 8 to 12. The testing set contains 1517 pairs for 89 questions. As an example, a correct answer to the question *"What do practitioners of Wicca worship?"* is *"An estimated 50,000 Americans practice Wicca, a form of polytheistic nature worship."* On the other hand, the answer candidate *"When people think of Wicca, they think of either Satanism or silly mumbo jumbo."* is incorrect. Sentences with more than 40 words and questions with only positive or only negative answers were filtered out (Yao et al., 2013). The average fraction of correct answers per question is 7.4% for training and 18.7% for testing. Performances are evaluated as for a re-ranking problem, in term of Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR). We report our results in Table 4. We evaluated several combinations of features. *IDF word-count* (IDF) is a baseline of

[1]Available at `http://nlp.stanford.edu/mengqiu/data/qg-emnlp07-data.tgz`

System	MAP	MRR
Random baseline	0.397	0.493
Wang et al. (2007)	0.603	0.685
Heilman and Smith (2010)	0.609	0.692
Wang and Manning (2010)	0.595	0.695
Yao et al. (2013)	0.631	0.748
Yih et al. (2013) LCLR	**0.709**	**0.770**
IDF word-count (IDF)	0.596	0.650
SRK	0.609	0.669
SRK + IDF	0.620	0.677
TESRK (WN)	0.642	0.725
TESRK (WN+NE)	0.656	0.744
TESRK (WN) + IDF	0.678	0.759
TESRK (WN+NE) + IDF	0.672	**0.768**

Table 4: Evaluation results on QA

IDF-weighted common word counting, integrated in a linear kernel. Then we implemented SRK and TESRK (with k from 1 to 5) with two typing schemes: *WN* stands for $\Gamma_p = \{$stem, synonym$\}$ and $\Gamma_v = \{$stem, synonym, hypernym, hyponym, entailment, holonym$\}$, and *WN+NE* adds type ne to both sets of types. We finally summed our kernels with the IDF baseline kernel. We observe that types which make use of WordNet variations seem to increase the most our performance. Our assumption was that named entities would be useful for question answering and that we could learn associations between question type and answer type through variations: NE does seem to help a little when combined with WN alone, but is less useful once TESRK is combined with our baseline of IDF-weighted common words. Overall, typing capabilities allow TESRK to obtain way better performances than SRK in both MAP and MRR, and our best system combining all our features is comparable to state-of-the-art systems in MRR, and significantly outperforms *SRK + IDF*, the system without types ($p < 0.05$).

5 Related work

Lodhi et al. (Lodhi et al., 2002) were among the first in NLP to use kernels: they apply *string kernels* which count common subsequences to text classification. Sentence pair classification however require the capture of 2 types of links: the link between sentences within a pair, and the link between pairs. Zanzotto et al. (Zanzotto et al., 2007) used a kernel method on syntactic tree pairs. They expanded on graph kernels in (Zanzotto et al., 2010). Their method first aligns tree nodes of a pair of sentences to form a single tree with placeholders. They then use *tree kernel* (Moschitti, 2006) to compute the number of common subtrees of those trees. Bu et al. (Bu et al., 2012) introduced a string rewriting kernel which can capture at once lexical equivalents and common syntactic dependencies on pair of sentences. All these kernel methods require an exact match or assume prior partial matches between words, thus limiting the kind of learned rewriting rules. Our contribution addresses this issue with a type-enriched string rewriting kernel which can account for lexico-semantic variations of words. Limitations of our rewriting rules include the impossibility to skip a pattern word and to replace wildcards by multiple words.

Some recent contributions (Chang et al., 2010; Wang and Manning, 2010) also provide a uniform way to learn both intermediary representations and a decision function using potentially rich feature sets. They use heuristics in the joint learning process to reduce the computational cost, while our kernel approach with a simple sequential representation of sentences has the benefit of efficiently computing an exact number of common rewriting rules between rewriting pairs. This in turn allows to precisely fine-tune the shape of desired rewriting rules through the design of the typing scheme.

6 Conclusion

We developed a unified kernel-based framework for solving sentence rewriting tasks. Types allow for an increased flexibility in counting common rewriting rules, and can also add a semantic layer to the rewritings. We show that we can efficiently compute a kernel which takes types into account, called type-enriched k-gram bijective string rewriting kernel. A SVM classifier with this kernel yields state-of-the-art results in paraphrase identification and answer sentence selection and outperforms comparable systems in recognizing textual entailment.

References

Eneko Agirre, Mona Diab, Daniel Cer, and Aitor Gonzalez-Agirre. 2012. Semeval-2012 task 6: A pilot on semantic textual similarity. In *Proceedings of the First Joint Conference on Lexical and Computational Semantics-Volume 1: Proceedings of the main conference and the shared task, and Volume*

2: Proceedings of the Sixth International Workshop on Semantic Evaluation, pages 385–393. Association for Computational Linguistics.

Fan Bu, Hang Li, and Xiaoyan Zhu. 2012. String re-writing kernel. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 449–458. Association for Computational Linguistics.

Hiram Calvo, Andrea Segura-Olivares, and Alejandro García. 2014. Dependency vs. constituent based syntactic n-grams in text similarity measures for paraphrase recognition. *Computación y Sistemas*, 18(3):517–554.

Chih-Chung Chang and Chih-Jen Lin. 2011. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27.

Ming-Wei Chang, Dan Goldwasser, Dan Roth, and Vivek Srikumar. 2010. Discriminative learning over constrained latent representations. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 429–437. Association for Computational Linguistics.

Ido Dagan, Oren Glickman, and Bernardo Magnini. 2006. The pascal recognising textual entailment challenge. In *Machine learning challenges. evaluating predictive uncertainty, visual object classification, and recognising tectual entailment*, pages 177–190. Springer.

William B Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Proc. of IWP*.

Bill Dolan, Chris Quirk, and Chris Brockett. 2004. Unsupervised construction of large paraphrase corpora: Exploiting massively parallel news sources. In *Proceedings of the 20th international conference on Computational Linguistics*, page 350. Association for Computational Linguistics.

Michael Heilman and Noah A Smith. 2010. Tree edit models for recognizing textual entailments, paraphrases, and answers to questions. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 1011–1019. Association for Computational Linguistics.

Aminul Islam and Diana Inkpen. 2009. Semantic similarity of short texts. *Recent Advances in Natural Language Processing V*, 309:227–236.

Sergio Jimenez, Claudia Becerra, Alexander Gelbukh, Av Juan Dios Bátiz, and Av Mendizábal. 2013. Softcardinality: hierarchical text overlap for student response analysis. In *Proceedings of the 2nd joint conference on lexical and computational semantics*, volume 2, pages 280–284.

Mihai C Lintean and Vasile Rus. 2011. Dissimilarity kernels for paraphrase identification. In *FLAIRS Conference*.

Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. 2002. Text classification using string kernels. *The Journal of Machine Learning Research*, 2:419–444.

Nitin Madnani and Bonnie J Dorr. 2010. Generating phrasal and sentential paraphrases: A survey of data-driven methods. *Computational Linguistics*, 36(3):341–387.

Nitin Madnani, Joel Tetreault, and Martin Chodorow. 2012. Re-examining machine translation metrics for paraphrase identification. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 182–190. Association for Computational Linguistics.

Rada Mihalcea, Courtney Corley, and Carlo Strapparava. 2006. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, volume 6, pages 775–780.

George A Miller. 1995. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41.

Thomas Morton, Joern Kottmann, Jason Baldridge, and Gann Bierner. 2005. Opennlp: A java-based nlp toolkit. http://opennlp.sourceforge.net.

Alessandro Moschitti. 2006. Efficient convolution kernels for dependency and constituent syntactic trees. In *Machine Learning: ECML 2006*, pages 318–329. Springer.

Martin F Porter. 2001. Snowball: A language for stemming algorithms.

Richard Socher, Eric H Huang, Jeffrey Pennin, Christopher D Manning, and Andrew Y Ng. 2011. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Advances in Neural Information Processing Systems*, pages 801–809.

Leslie G Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421.

Vladimir Vapnik. 2000. *The nature of statistical learning theory*. Springer Science & Business Media.

Stephen Wan, Mark Dras, Robert Dale, and Cécile Paris. 2006. Using dependency-based features to take the para-farce out of paraphrase. In *Proceedings of the Australasian Language Technology Workshop*, volume 2006.

Mengqiu Wang and Christopher D Manning. 2010. Probabilistic tree-edit models with structured latent variables for textual entailment and question answering. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 1164–1172. Association for Computational Linguistics.

Mengqiu Wang, Noah A Smith, and Teruko Mitamura. 2007. What is the jeopardy model? a quasi-synchronous grammar for qa. In *EMNLP-CoNLL*, volume 7, pages 22–32.

Xuchen Yao, Benjamin Van Durme, Chris Callison-Burch, and Peter Clark. 2013. Answer extraction as sequence tagging with tree edit distance. In *HLT-NAACL*, pages 858–867. Citeseer.

Wen-tau Yih, Ming-Wei Chang, Christopher Meek, and Andrzej Pastusiak. 2013. Question answering using enhanced lexical semantic models. In *Proceedings of the 26rd International Conference on Computational Linguistics*. Association for Computational Linguistics.

Fabio Massimo Zanzotto, Marco Pennacchiotti, and Alessandro Moschitti. 2007. Shallow semantics in fast textual entailment rule learners. In *Proceedings of the ACL-PASCAL workshop on textual entailment and paraphrasing*, pages 72–77. Association for Computational Linguistics.

Fabio Massimo Zanzotto, Lorenzo DellArciprete, and Alessandro Moschitti. 2010. Efficient graph kernels for textual entailment recognition. *Fundamenta Informaticae*.