"Text Mining and Natural Language Processing", Anne Kao and Steve Poteet (eds.), pp. 29-44, Springer, 2007.

Extracting Relations from Text
From Word Sequences to Dependency Paths

Razvan C. Bunescu and Raymond J. Mooney

Department of Computer Sciences
University of Texas at Austin
1 University Station C0500
Austin, TX 78712-0233
razvan,mooney@cs.utexas.edu

1 Introduction

Extracting semantic relationships between entities mentioned in text documents is an important task in natural language processing. The various types of relationships that are discovered between mentions of entities can provide useful structured information to a text mining system [1]. Traditionally, the task specifies a predefined set of entity types and relation types that are deemed to be relevant to a potential user and that are likely to occur in a particular text collection. For example, information extraction from newspaper articles is usually concerned with identifying mentions of people, organizations, locations, and extracting useful relations between them. Relevant relation types range from social relationships, to roles that people hold inside an organization, to relations between organizations, to physical locations of people and organizations. Scientific publications in the biomedical domain offer a type of narrative that is very different from the newspaper discourse. A significant effort is currently spent on automatically extracting relevant pieces of information from Medline, an online collection of biomedical abstracts. Proteins, genes and cells are examples of relevant entities in this task, whereas subcellular localizations and protein-protein interactions are two of the relation types that have received significant attention recently. The inherent difficulty of the relation extraction task is further compounded in the biomedical domain by the relative scarcity of tools able to analyze the corresponding type of narrative. Most existing natural language processing tools, such as tokenizers, sentence segmenters, part-of-speech (POS) taggers, shallow or full parsers are trained on newspaper corpora, and consequently they inccur a loss in accuracy when applied to biomedical literature. Therefore, information extraction systems developed for biological corpora need to be robust to POS or parsing errors, or to give reasonable performance using shallower but more reliable information, such as chunking instead of full parsing.

In this chapter, we present two recent approaches to relation extraction that differ in terms of the kind of linguistic information they use:

1. In the first method (Section 2), each potential relation is represented implicitly as a vector of features, where each feature corresponds to a *word sequence* anchored at the two entities forming the relationship. A relation extraction system is trained based on the subsequence kernel from [2]. This kernel is further generalized so that words can be replaced with word classes, thus enabling the use of information coming from POS tagging, named entity recognition, chunking or Wordnet [3].
2. In the second approach (Section 3), the representation is centered on the shortest *dependency path* between the two entities in the dependency graph of the sentence. Because syntactic analysis is essential in this method, its applicability is limited to domains where syntactic parsing gives reasonable accuracy.

Entity recognition, a prerequisite for relation extraction, is usually cast as a sequence tagging problem, in which words are tagged as being either outside any entity, or inside a particular type of entity. Most approaches to entity tagging are therefore based on probabilistic models for labeling sequences, such as Hidden Markov Models [4], Maximum Entropy Markov Models [5], or Conditional Random Fields [6], and obtain a reasonably high accuracy. In the two information extraction methods presented in this chapter, we assume that the entity recognition task was done and focus only on the relation extraction part.

2 Subsequence Kernels for Relation Extraction

One of the first approaches to extracting interactions between proteins from biomedical abstracts is that of Blaschke *et al.*, described in [7, 8]. Their system is based on a set of manually developed rules, where each rule (or frame) is a sequence of words (or POS tags) and two protein-name tokens. Between every two adjacent words is a number indicating the maximum number of intervening words allowed when matching the rule to a sentence. An example rule is "*interaction of (3) <P> (3) with (3) <P>*", where '<P>' is used to denote a protein name. A sentence matches the rule if and only if it satisfies the word constraints in the given order and respects the respective word gaps.

In [9] the authors described a new method ELCS (Extraction using Longest Common Subsequences) that automatically learns such rules. ELCS' rule representation is similar to that in [7, 8], except that it currently does not use POS tags, but allows disjunctions of words. An example rule learned by this system is "- *(7) interaction (0) [between | of] (5) <P> (9) <P> (17)* .". Words in square brackets separated by '|' indicate disjunctive lexical constraints, i.e. one of the given words must match the sentence at that position. The numbers in parentheses between adjacent constraints indicate the maximum number of unconstrained words allowed between the two.

2.1 Capturing Relation Patterns with a String Kernel

Both Blaschke and ELCS do relation extraction based on a limited set of matching rules, where a rule is simply a sparse (gappy) subsequence of words or POS tags anchored on the two protein-name tokens. Therefore, the two methods share a common limitation: either through manual selection (Blaschke), or as a result of a greedy learning procedure (ELCS), they end up using only a subset of all possible anchored sparse subsequences. Ideally, all such anchored sparse subsequences would be used as features, with weights reflecting their relative accuracy. However explicitly creating for each sentence a vector with a position for each such feature is infeasible, due to the high dimensionality of the feature space. Here, we exploit dual learning algorithms that process examples only via computing their dot-products, such as in Support Vector Machines (SVMs) [10, 11]. An SVM learner tries to find a hyperplane that separates positive from negative examples and at the same time maximizes the separation (margin) between them. This type of max-margin separator has been shown both theoretically and empirically to resist overfitting and to provide good generalization performance on unseen examples.

Computing the dot-product (i.e. the kernel) between the features vectors associated with two relation examples amounts to calculating the number of common anchored subsequences between the two sentences. This is done efficiently by modifying the dynamic programming algorithm used in the string kernel from [2] to account only for common sparse subsequences constrained to contain the two protein-name tokens. The feature space is further prunned down by utilizing the following property of natural language statements: when a sentence asserts a relationship between two entity mentions, it generally does this using one of the following four patterns:

- [**FB**] **F**ore–**B**etween: words before and between the two entity mentions are simultaneously used to express the relationship. Examples: 'interaction of $\langle P_1 \rangle$ with $\langle P_2 \rangle$', 'activation of $\langle P_1 \rangle$ by $\langle P_2 \rangle$'.
- [**B**] **B**etween: only words between the two entities are essential for asserting the relationship. Examples: '$\langle P_1 \rangle$ interacts with $\langle P_2 \rangle$', '$\langle P_1 \rangle$ is activated by $\langle P_2 \rangle$'.
- [**BA**] **B**etween–**A**fter: words between and after the two entity mentions are simultaneously used to express the relationship. Examples: '$\langle P_1 \rangle$ – $\langle P_2 \rangle$ complex', '$\langle P_1 \rangle$ and $\langle P_2 \rangle$ interact'.
- [**M**] **M**odifier: the two entity mentions have no words between them. Examples: *U.S. troops* (a ROLE:STAFF relation), *Serbian general* (ROLE:CITIZEN).

While the first three patterns are sufficient to capture most cases of interactions between proteins, the last pattern is needed to account for various relationships expressed through noun-noun or adjective-noun compounds in the newspaper corpora.

Another observation is that all these patterns use at most 4 words to express the relationship (not counting the two entity names). Consequently,

when computing the relation kernel, we restrict the counting of common anchored subsequences only to those having one of the four types described above, with a maximum word-length of 4. This type of feature selection leads not only to a faster kernel computation, but also to less overfitting, which results in increased accuracy.

The patterns enumerated above are completely lexicalized and consequently their performance is limited by data sparsity. This can be alleviated by categorizing words into classes with varying degrees of generality, and then allowing patterns to use both words and their classes. Examples of word classes are POS tags and generalizations over POS tags such as Noun, Active Verb or Passive Verb. The entity type can also be used, if the word is part of a known named entity. Also, if the sentence is segmented into syntactic chunks such as noun phrases (NP) or verb phrases (VP), the system may choose to consider only the head word from each chunk, together with the type of the chunk as another word class. Content words such as nouns and verbs can also be related to their synsets via WordNet. Patterns then will consist of sparse subsequences of words, POS tags, generalized POS tags, entity and chunk types, or WordNet synsets. For example, 'Noun of $\langle P_1 \rangle$ by $\langle P_2 \rangle$' is an FB pattern based on words and general POS tags.

2.2 A Generalized Subsequence Kernel

Let $\Sigma_1, \Sigma_2, ..., \Sigma_k$ be some disjoint feature spaces. Following the example in Section 2.1, Σ_1 could be the set of words, Σ_2 the set of POS tags, etc. Let $\Sigma_\times = \Sigma_1 \times \Sigma_2 \times ... \times \Sigma_k$ be the set of all possible feature vectors, where a feature vector would be associated with each position in a sentence. Given two feature vectors $x, y \in \Sigma_\times$, let $c(x, y)$ denote the number of common features between x and y. The next notation follows that introduced in [2]. Thus, let s, t be two sequences over the finite set $\Sigma_\times$, and let $|s|$ denote the length of $s = s_1...s_{|s|}$. The sequence $s[i{:}j]$ is the contiguous subsequence $s_i...s_j$ of s. Let $\mathbf{i} = (i_1, ..., i_{|\mathbf{i}|})$ be a sequence of $|\mathbf{i}|$ indices in s, in ascending order. We define the length $l(\mathbf{i})$ *of the index sequence* $\mathbf{i}$ *in* s as $i_{|\mathbf{i}|} - i_1 + 1$. Similarly, $\mathbf{j}$ is a sequence of $|\mathbf{j}|$ indices in t.

Let $\Sigma_\cup = \Sigma_1 \cup \Sigma_2 \cup ... \cup \Sigma_k$ be the set of all possible features. We say that the sequence $u \in \Sigma_\cup^*$ is a (sparse) subsequence of s if there is a sequence of $|u|$ indices $\mathbf{i}$ such that $u_k \in s_{i_k}$, for all $k = 1, ..., |u|$. Equivalently, we write $u \prec s[\mathbf{i}]$ as a shorthand for the component-wise '$\in$' relationship between u and $s[\mathbf{i}]$.

Finally, let $K_n(s, t, \lambda)$ (Equation 1) be the number of weighted sparse subsequences u of length n common to s and t (i.e. $u \prec s[\mathbf{i}]$, $u \prec t[\mathbf{j}]$), where the weight of u is $\lambda^{l(\mathbf{i})+l(\mathbf{j})}$, for some $\lambda \leq 1$.

$$K_n(s, t, \lambda) = \sum_{u \in \Sigma_\cup^n} \sum_{\mathbf{i}: u \prec s[\mathbf{i}]} \sum_{\mathbf{j}: u \prec t[\mathbf{j}]} \lambda^{l(\mathbf{i})+l(\mathbf{j})} \quad (1)$$

Let $\mathbf{i}$ and $\mathbf{j}$ be two index sequences of length n. By definition, for every k between 1 and n, $c(s_{i_k}, t_{j_k})$ returns the number of common features between s and t at positions i_k and j_k. If $c(s_{i_k}, t_{j_k}) = 0$ for some k, there are no common feature sequences of length n between $s[\mathbf{i}]$ and $t[\mathbf{j}]$. On the other hand, if $c(s_{i_k}, t_{j_k}) = 0$ is greater than 1, this means that there is more than one common feature that can be used at position k to obtain a common feature sequence of length n. Consequently, the number of common feature sequences of length n between $s[\mathbf{i}]$ and $t[\mathbf{j}]$, i.e. the size of the set $\{u \in \Sigma_{\cup}^{n} | u \prec s[\mathbf{i}], u \prec t[\mathbf{j}]\}$, is given by $\prod_{k=1}^{n} c(s_{i_k}, t_{j_k})$. Therefore, $K_n(s, t, \lambda)$ can be rewritten as in Equation 2:

$$K_n(s,t,\lambda) = \sum_{\mathbf{i}:|\mathbf{i}|=n} \sum_{\mathbf{j}:|\mathbf{j}|=n} \prod_{k=1}^{n} c(s_{i_k}, t_{j_k}) \lambda^{l(\mathbf{i})+l(\mathbf{j})} \tag{2}$$

We use λ as a decaying factor that penalizes longer subsequences. For sparse subsequences, this means that wider gaps will be penalized more, which is exactly the desired behavior for our patterns. Through them, we try to capture head-modifier dependencies that are important for relation extraction; for lack of reliable dependency information, the larger the word gap is between two words, the less confident we are in the existence of a head-modifier relationship between them.

To enable an efficient computation of K_n, we use the auxiliary function $K_n^{'}$ with a similar definition as K_n, the only difference being that it counts the length from the beginning of the particular subsequence u to the end of the strings s and t, as illustrated in Equation 3:

$$K_n^{'}(s,t,\lambda) = \sum_{u \in \Sigma_{\cup}^{n}} \sum_{\mathbf{i}:u \prec s[\mathbf{i}]} \sum_{\mathbf{j}:u \prec t[\mathbf{j}]} \lambda^{|s|+|t|-i_1-j_1+2} \tag{3}$$

An equivalent formula for $K_n^{'}(s, t, \lambda)$ is obtained by changing the exponent of λ from Equation 2 to $|s| + |t| - i_1 - j_1 + 2$.

Based on all definitions above, K_n is computed in $O(kn|s||t|)$ time, by modifying the recursive computation from [2] with the new factor $c(x, y)$, as shown in Figure 1. In this figure, the sequence sx is the result of appending x to s (with ty defined in a similar way). To avoid clutter, the parameter λ is not shown in the argument list of K and K', unless it is instantiated to a specific constant.

2.3 Computing the Relation Kernel

As described at the beginning of Section 2, the input consists of a set of sentences, where each sentence contains exactly two entities (protein names in the case of interaction extraction). In Figure 2 we show the segments that will be used for computing the relation kernel between two example sentences s and t. In sentence s for instance, x_1 and x_2 are the two entities, s_f is the

$$
\begin{aligned}
K_0^{'}(s,t) &= 1, \ for\ all\ s,t \\
K_i^{''}(sx,ty) &= \lambda K_i^{''}(sx,t) + \lambda^2 K_{i-1}^{'}(s,t)\cdot c(x,y) \\
K_i^{'}(sx,t) &= \lambda K_i^{'}(s,t) + K_i^{''}(sx,t) \\
K_n(s,t) &= 0, \ if\ \ \min(|s|,|t|) < n \\
K_n(sx,t) &= K_n(s,t) + \sum_j \lambda^2 K_{n-1}^{'}(s,t[1:j-1])\cdot c(x,t[j])
\end{aligned}
$$

Fig. 1. Computation of subsequence kernel.

sentence segment before x_1, s_b is the segment between x_1 and x_2, and s_a is the sentence segment after x_2. For convenience, we also include the auxiliary segment $s_b^{'} = x_1 s_b x_2$, whose span is computed as $l(s_b^{'}) = l(s_b) + 2$ (in all length computations, we consider x_1 and x_2 as contributing one unit only).

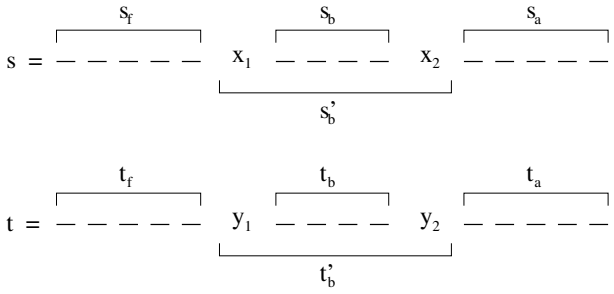

Fig. 2. Sentence segments.

The relation kernel computes the number of common patterns between two sentences s and t, where the set of patterns is restricted to the four types introduced in Section 2.1. Therefore, the kernel $rK(s,t)$ is expressed as the sum of four sub-kernels: $fbK(s,t)$ counting the number of common fore–between patterns, $bK(s,t)$ for between patterns, $baK(s,t)$ for between–after patterns, and $mK(s,t)$ for modifier patterns, as in Figure 3. The symbol $\mathbb{1}$ is used there as a shorthand for the indicator function, which is 1 if the argument is true, and 0 otherwise.

The first three sub-kernels include in their computation the counting of common subsequences between $s_b^{'}$ and $t_b^{'}$. In order to speed up the computation, all these common counts are calculated separately in bK_i, which is defined as the number of common subsequences of length i between $s_b^{'}$ and $t_b^{'}$, anchored at x_1/x_2 and y_1/y_2 respectively (i.e. constrained to start at x_1 in $s_b^{'}$

$$
\begin{aligned}
rK(s,t) &= fbK(s,t) + bK(s,t) + baK(s,t) + mK(s,t) \\
bK_i(s,t) &= K_i(s_b, t_b, 1) \cdot c(x_1, y_1) \cdot c(x_2, y_2) \cdot \lambda^{l(s_b^{'}) + l(t_b^{'})} \\
fbK(s,t) &= \sum_{i,j} bK_i(s,t) \cdot K_j^{'}(s_f, t_f), \quad 1 \leq \mathrm{i},\ 1 \leq \mathrm{j},\ \mathrm{i} + \mathrm{j} < \mathrm{fb}_{\max} \\
bK(s,t) &= \sum_{i} bK_i(s,t), \quad 1 \leq \mathrm{i} \leq \mathrm{b}_{\max} \\
baK(s,t) &= \sum_{i,j} bK_i(s,t) \cdot K_j^{'}(s_a^{-}, t_a^{-}), \quad 1 \leq \mathrm{i},\ 1 \leq \mathrm{j},\ \mathrm{i} + \mathrm{j} < \mathrm{ba}_{\max} \\
mK(s,t) &= \mathbb{1}(s_b = \emptyset) \cdot \mathbb{1}(t_b = \emptyset) \cdot c(x_1, y_1) \cdot c(x_2, y_2) \cdot \lambda^{2+2},
\end{aligned}
$$

Fig. 3. Computation of relation kernel.

and y_1 in $t_b^{'}$, and to end at x_2 in $s_b^{'}$ and y_2 in $t_b^{'}$). Then fbK simply counts the number of subsequences that match j positions before the first entity and i positions between the entities, constrained to have length less than a constant fb_{max}. To obtain a similar formula for baK we simply use the reversed (mirror) version of segments s_a and t_a (e.g. s_a^- and t_a^-). In Section 2.1 we observed that all three subsequence patterns use at most 4 words to express a relation, therefore the constants fb_{max}, b_{max} and ba_{max} are set to 4. Kernels K and $K^{'}$ are computed using the procedure described in Section 2.2.

3 A Dependency-Path Kernel for Relation Extraction

The pattern examples from Section 2.1 show the two entity mentions, together with the set of words that are relevant for their relationship. A closer analysis of these examples reveals that all relevant words form a shortest path between the two entities in a graph structure where edges correspond to relations between a word (head) and its dependents. For example, Figure 4 shows the full dependency graphs for two sentences from the ACE (Automated Content Extraction) newspaper corpus [12], in which words are represented as nodes and word-word dependencies are represented as directed edges. A subset of these word-word dependencies capture the predicate-argument relations present in the sentence. Arguments are connected to their target predicates either directly through an arc pointing to the predicate ('troops $\rightarrow$ raided'), or indirectly through a preposition or infinitive particle ('warning $\leftarrow$ to $\leftarrow$ stop'). Other types of word-word dependencies account for modifier-head relationships present in adjective-noun compounds ('several $\rightarrow$ stations'), noun-noun compounds ('pumping $\rightarrow$ stations'), or adverb-verb constructions ('recently $\rightarrow$ raided').

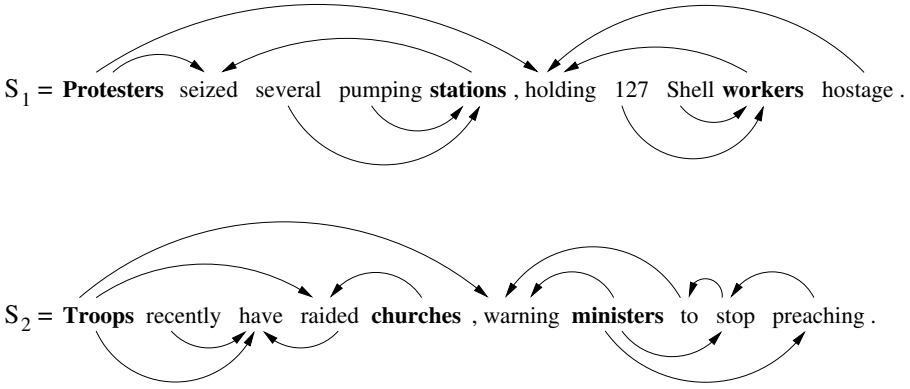

Fig. 4. Sentences as dependency graphs.

Table 1. Shortest Path representation of relations.

Relation Instance	Shortest Path in Undirected Dependency Graph
S_1:protesters AT stations	**protesters** → seized ← **stations**
S_1:workers AT stations	**workers** → holding ← protesters → seized ← **stations**
S_2:troops AT churches	**troops** → raided ← **churches**
S_2:ministers AT churches	**ministers** → warning ← troops → raided ← **churches**

Word-word dependencies are typically categorized in two classes as follows:

- [**Local Dependencies**] These correspond to local predicate-argument (or head-modifier) constructions such as 'troops → raided', or 'pumping → stations' in Figure 4.
- [**Non-local Dependencies**] Long-distance dependencies arise due to various linguistic constructions such as coordination, extraction, raising and control. In Figure 4, among non-local dependencies are 'troops → warning', or 'ministers → preaching'.

A Context Free Grammar (CFG) parser can be used to extract local dependencies, which for each sentence form a dependency tree. Mildly context sensitive formalisms such as Combinatory Categorial Grammar (CCG) [13] model word-word dependencies more directly and can be used to extract both local and long-distance dependencies, giving rise to a directed acyclic graph, as illustrated in Figure 4.

3.1 The Shortest Path Hypothesis

If e_1 and e_2 are two entities mentioned in the same sentence such that they are observed to be in a relationship R, then the contribution of the sentence

dependency graph to establishing the relationship $R(e_1, e_2)$ is almost exclusively concentrated in the shortest path between e_1 and e_2 in the undirected version of the dependency graph.

If entities e_1 and e_2 are arguments of the same predicate, then the shortest path between them will pass through the predicate, which may be connected directly to the two entities, or indirectly through prepositions. If e_1 and e_2 belong to different predicate-argument structures that share a common argument, then the shortest path will pass through this argument. This is the case with the shortest path between 'stations' and 'workers' in Figure 4, passing through 'protesters', which is an argument common to both predicates 'holding' and 'seized'. In Table 1, we show the paths corresponding to the four relation instances encoded in the ACE corpus for the two sentences from Figure 4. All these paths support the LOCATED relationship. For the first path, it is reasonable to infer that if a PERSON entity (e.g. 'protesters') is doing some action (e.g. 'seized') to a FACILITY entity (e.g. 'station'), then the PERSON entity is LOCATED at that FACILITY entity. The second path captures the fact that the same PERSON entity (e.g. 'protesters') is doing two actions (e.g. 'holding' and 'seized') , one action to a PERSON entity (e.g. 'workers'), and the other action to a FACILITY entity (e.g. 'station'). A reasonable inference in this case is that the 'workers' are LOCATED at the 'station'.

In Figure 5, we show three more examples of the LOCATED (AT) relationship as dependency paths created from one or two predicate-argument structures. The second example is an interesting case, as it illustrates how annotation decisions are accommodated in our approach. Using a reasoning similar with that from the previous paragraph, it is reasonable to infer that 'troops' are LOCATED in 'vans', and that 'vans' are LOCATED in 'city'. However, because 'vans' is not an ACE markable, it cannot participate in an annotated relationship. Therefore, 'troops' is annotated as being LOCATED in 'city', which makes sense due to the transitivity of the relation LOCATED. In our approach, this leads to shortest paths that pass through two or more predicate-argument structures.

The last relation example is a case where there exist multiple shortest paths in the dependency graph between the same two entities – there are actually two different paths, with each path replicated into three similar paths due to coordination. Our current approach considers only one of the shortest paths, nevertheless it seems reasonable to investigate using all of them as multiple sources of evidence for relation extraction.

There may be cases where e_1 and e_2 belong to predicate-argument structures that have no argument in common. However, because the dependency graph is always connected, we are guaranteed to find a shortest path between the two entities. In general, we shall find a shortest sequence of predicate-argument structures with target predicates $P_1, P_2, ..., P_n$ such that e_1 is an argument of P_1, e_2 is an argument of P_n, and any two consecutive predicates P_i and P_{i+1} share a common argument (where by "argument" we mean both arguments and complements).

(1) He had no regrets for **his** actions in **Brcko**.

his → actions ← in ← **Brcko**

(2) U.S. **troops** today acted for the first time to capture an alleged Bosnian war criminal, rushing from unmarked vans parked in the northern Serb-dominated **city** of Bijeljina.

troops → rushing ← from ← vans → parked ← in ← **city**

(3) Jelisic created an atmosphere of terror at the **camp** by killing, abusing and threatening the **detainees**.

detainees → killing ← Jelisic → created ← at ← **camp**
detainees → abusing ← Jelisic → created ← at ← **camp**
detainees → threatning ← Jelisic → created ← at ← **camp**
detainees → killing → by → created ← at ← **camp**
detainees → abusing → by → created ← at ← **camp**
detainees → threatening → by → created ← at ← **camp**

Fig. 5. Relation examples.

3.2 Learning with Dependency Paths

The shortest path between two entities in a dependency graph offers a very condensed representation of the information needed to assess their relationship. A dependency path is represented as a sequence of words interspersed with arrows that indicate the orientation of each dependency, as illustrated in Table 1. These paths however are completely lexicalized and consequently their performance will be limited by data sparsity. The solution is to allow paths to use both words and their word classes, similar with the approach taken for the subsequence patterns in Section 2.1.

The set of features can then be defined as a Cartesian product over words and word classes, as illustrated in Figure 6 for the dependency path between 'protesters' and 'station' in sentence S_1. In this representation, sparse or contiguous subsequences of nodes along the lexicalized dependency path (i.e. path fragments) are included as features simply by replacing the rest of the nodes with their corresponding generalizations.

Examples of features generated by Figure 6 are "protesters → seized ← stations", "Noun → Verb ← Noun", "PERSON → seized ← FACILITY", or "PERSON → Verb ← FACILITY". The total number of features generated by this dependency path is $4 \times 1 \times 3 \times 1 \times 4$.

For verbs and nouns (and their respective word classes) occurring along a dependency path we also use an additional suffix '(-)' to indicate a negative polarity item. In the case of verbs, this suffix is used when the verb (or an attached auxiliary) is modified by a negative polarity adverb such as 'not' or

$$\begin{bmatrix} \text{protesters} \\ \text{NNS} \\ \text{Noun} \\ \text{PERSON} \end{bmatrix} \times [\rightarrow] \times \begin{bmatrix} \text{seized} \\ \text{VBD} \\ \text{Verb} \end{bmatrix} \times [\leftarrow] \times \begin{bmatrix} \text{stations} \\ \text{NNS} \\ \text{Noun} \\ \text{FACILITY} \end{bmatrix}$$

Fig. 6. Feature generation from dependency path.

'never'. Nouns get the negative suffix whenever they are modified by negative determiners such as 'no', 'neither' or 'nor'. For example, the phrase "He never went to Paris" is associated with the dependency path "He $\rightarrow$ went(-) $\leftarrow$ to $\leftarrow$ Paris".

As in Section 2, we use kernel SVMs in order to avoid working explicitely with high-dimensional dependency path feature vectors. Computing the dot-product (i.e. kernel) between two relation examples amounts to calculating the number of common features (i.e. paths) between the two examples. If $\boldsymbol{x} = x_1 x_2 ... x_m$ and $\boldsymbol{y} = y_1 y_2 ... y_n$ are two relation examples, where x_i denotes the set of word classes corresponding to position i (as in Figure 6), then the number of common features between $\boldsymbol{x}$ and $\boldsymbol{y}$ is computed as in Equation 4.

$$K(\mathbf{x}, \mathbf{y}) = \mathbb{1}(m = n) \cdot \prod_{i=1}^{n} c(x_i, y_i) \tag{4}$$

where $c(x_i, y_i) = |x_i \cap y_i|$ is the number of common word classes between x_i and y_i.

This is a simple kernel, whose computation takes $O(n)$ time. If the two paths have different lengths, they correspond to different ways of expressing a relationship – for instance, they may pass through a different number of predicate argument structures. Consequently, the kernel is defined to be 0 in this case. Otherwise, it is the product of the number of common word classes at each position in the two paths. As an example, let us consider two instances of the LOCATED relationship, and their corresponding dependency paths:

1. '**his** actions in **Brcko**' (**his** $\rightarrow$ actions $\leftarrow$ in $\leftarrow$ **Brcko**).
2. '**his** arrival in **Beijing**' (**his** $\rightarrow$ arrival $\leftarrow$ in $\leftarrow$ **Beijing**).

Their representation as a sequence of sets of word classes is given by:

1. $\boldsymbol{x} = [x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7]$, where $x_1 = \{\text{his, PRP, PERSON}\}$, $x_2 = \{\rightarrow\}$, $x_3 = \{\text{actions, NNS, Noun}\}$, $x_4 = \{\leftarrow\}$, $x_5 = \{\text{in, IN}\}$, $x_6 = \{\leftarrow\}$, $x_7 = \{\text{Brcko, NNP, Noun, LOCATION}\}$
2. $\boldsymbol{y} = [y_1\ y_2\ y_3\ y_4\ y_5\ y_6\ y_7]$, where $y_1 = \{\text{his, PRP, PERSON}\}$, $y_2 = \{\rightarrow\}$, $y_3 = \{\text{arrival, NN, Noun}\}$, $y_4 = \{\leftarrow\}$, $y_5 = \{\text{in, IN}\}$, $y_6 = \{\leftarrow\}$, $y_7 = \{\text{Beijing, NNP, Noun, LOCATION}\}$

Based on the formula from Equation 4, the kernel is computed as $K(\boldsymbol{x}, \boldsymbol{y}) = 3 \times 1 \times 1 \times 1 \times 2 \times 1 \times 3 = 18$.

4 Experimental Evaluation

The two relation kernels described above are evaluated on the task of extracting relations from two corpora with different types of narrative, which are described in more detail in the following sections. In both cases, we assume that the entities and their labels are known. All preprocessing steps – sentence segmentation, tokenization, POS tagging and chunking – were performed using the OpenNLP[1] package. If a sentence contains n entities ($n \geq 2$), it is replicated into $\binom{n}{2}$ sentences, each containing only two entities. If the two entities are known to be in a relationship, then the replicated sentence is added to the set of corresponding positive sentences, otherwise it is added to the set of negative sentences. During testing, a sentence having n entities ($n \geq 2$) is again replicated into $\binom{n}{2}$ sentences in a similar way.

The dependency graph that is input to the shortest path dependecy kernel is obtained from two different parsers:

- The CCG parser introduced in [14][2] outputs a list of functor-argument dependencies, from which head-modifier dependencies are obtained using a straightforward procedure (for more details, see [15]).
- Head-modifier dependencies can be easily extracted from the full parse output of Collins' CFG parser [16], in which every non-terminal node is annotated with head information.

The relation kernels are used in conjunction with SVM learning in order to find a decision hyperplane that best separates the positive examples from negative examples. We modified the LibSVM[3] package by plugging in the kernels described above. The factor λ in the subsequence kernel is set to 0.75. The performance is measured using *precision* (percentage of correctly extracted relations out of the total number of relations extracted), *recall* (percentage of correctly extracted relations out of the total number of relations annotated in the corpus), and *F-measure* (the harmonic mean of *precision* and *recall*).

4.1 Interaction Extraction from AIMed

We did comparative experiments on the AIMed corpus, which has been previously used for training the protein interaction extraction systems in [9]. It consists of 225 Medline abstracts, of which 200 are known to describe interactions between human proteins, while the other 25 do not refer to any interaction. There are 4084 protein references and around 1000 tagged interactions in this dataset.

The following systems are evaluated on the task of retrieving protein interactions from AIMed (assuming gold standard proteins):

[1] URL: http://opennlp.sourceforge.net
[2] URL:http://www.ircs.upenn.edu/~juliahr/Parser/
[3] URL:http://www.csie.ntu.edu.tw/~cjlin/libsvm/

• [**Manual**]: We report the performance of the rule-based system of [7, 8].

• [**ELCS**]: We report the 10-fold cross-validated results from [9] as a Precision-Recall (PR) graph.

• [**SSK**]: The subseqeuence kernel is trained and tested on the same splits as ELCS. In order to have a fair comparison with the other two systems, which use only lexical information, we do not use any word classes here.

• [**SPK**]: This is the shortest path dependency kernel, using the head-modifier dependencies extracted by Collins' syntactic parser. The kernel is trained and tested on the same 10 splits as ELCS and SSK.

The Precision-Recall curves that show the trade-off between these metrics are obtained by varying a threshold on the minimum acceptable extraction confidence, based on the probability estimates from LibSVM. The results, summarized in Figure 7(a), show that the subsequence kernel outperforms the other three systems, with a substantial gain. The syntactic parser, which is originally trained on a newspaper corpus, builds less accurate dependency structures for the biomedical text. This is reflected in a significantly reduced accuracy for the dependency kernel.

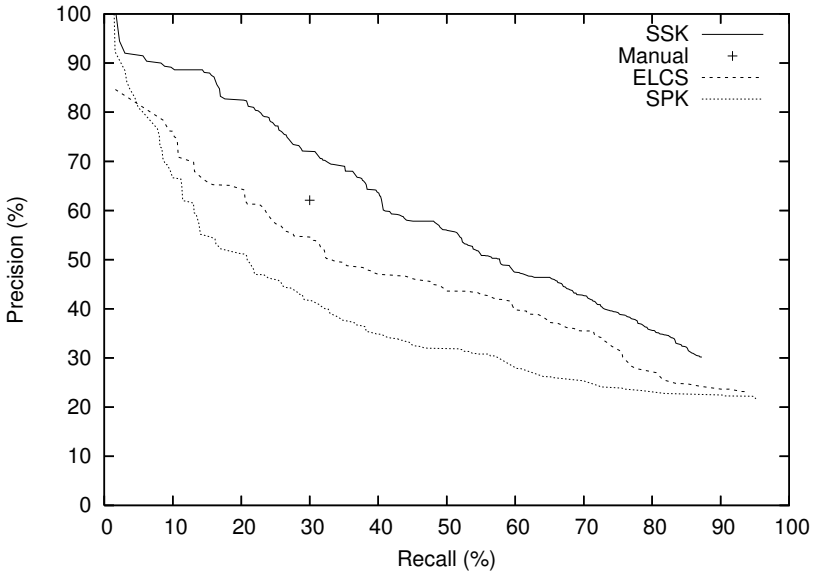

Fig. 7. Precision-Recall curves for protein interaction extractors.

4.2 Relation Extraction from ACE

The two kernels are also evaluated on the task of extracting top-level relations from the ACE corpus [12], the version used for the September 2002 evaluation. The training part of this dataset consists of 422 documents, with a separate set of 97 documents reserved for testing. This version of the ACE corpus contains three types of annotations: coreference, named entities and relations. There are five types of entities – PERSON, ORGANIZATION, FACILITY, LOCATION, and GEO-POLITICAL ENTITY – which can participate in five general, top-level relations: ROLE, PART, LOCATED, NEAR, and SOCIAL. In total, there are 7,646 intra-sentential relations, of which 6,156 are in the training data and 1,490 in the test data.

A recent approach to extracting relations is described in [17]. The authors use a generalized version of the tree kernel from [18] to compute a kernel over relation examples, where a relation example consists of the smallest dependency tree containing the two entities of the relation. Precision and recall values are reported for the task of extracting the 5 top-level relations in the ACE corpus under two different scenarios:

– [**S1**] This is the classic setting: one multi-class SVM is learned to discriminate among the 5 top-level classes, plus one more class for the no-relation cases.

– [**S2**] One binary SVM is trained for *relation detection*, meaning that all positive relation instances are combined into one class. The thresholded output of this binary classifier is used as training data for a second multi-class SVM, trained for *relation classification*.

The subsequence kernel (SSK) is trained under the first scenario, to recognize the same 5 top-level relation types. While for protein interaction extraction only the lexicalized version of the kernel was used, here we utilize more features, corresponding to the following feature spaces: Σ_1 is the word vocabulary, Σ_2 is the set of POS tags, Σ_3 is the set of generic POS tags, and Σ_4 contains the 5 entity types. Chunking information is used as follows: all (sparse) subsequences are created exclusively from the chunk heads, where a head is defined as the last word in a chunk. The same criterion is used for computing the length of a subsequence – all words other than head words are ignored. This is based on the observation that in general words other than the chunk head do not contribute to establishing a relationship between two entities outside of that chunk. One exception is when both entities in the example sentence are contained in the same chunk. This happens very often due to noun-noun ('U.S. troops') or adjective-noun ('Serbian general') compounds. In these cases, the chunk is allowed to contribute both entity heads.

The shortest-path dependency kernel (SPK) is trained under both scenarios. The dependencies are extracted using either Hockenmaier's CCG parser (SPK-CCG) [14], or Collins' CFG parser (SPK-CFG) [16].

Table 2 summarizes the performance of the two relation kernels on the ACE corpus. For comparison, we also show the results presented in [17] for

their best performing kernel K4 (a sum between a bag-of-words kernel and a tree dependency kernel) under both scenarios.

Table 2. Extraction Performance on ACE.

(Scenario) Method	Precision	Recall	F-measure
(S1) K4	70.3	26.3	38.0
(S1) SSK	73.9	35.2	47.7
(S1) SPK-CCG	67.5	37.2	48.0
(S1) SPK-CFG	**71.1**	**39.2**	**50.5**
(S2) K4	67.1	35.0	45.8
(S2) SPK-CCG	63.7	41.4	50.2
(S2) SPK-CFG	**65.5**	**43.8**	**52.5**

The shortest-path dependency kernels outperform the dependency kernel from [17] in both scenarios, with a more substantial gain for SP-CFG. An error analysis revealed that Collins' parser was better at capturing local dependencies, hence the increased accuracy of SP-CFG. Another advantage of shortest-path dependency kernels is that their training and testing are very fast – this is due to representing the sentence as a chain of dependencies on which a fast kernel can be computed. All the four SP kernels from Table 2 take between 2 and 3 hours to train and test on a 2.6GHz Pentium IV machine.

As expected, the newspaper articles from ACE are less prone to parsing errors than the biomedical articles from AIMed. Consequently, the extracted dependency structures are more accurate, leading to an improved accuracy for the dependency kernel.

To avoid numerical problems, the dependency paths are constrained to pass through at most 10 words (as observed in the training data) by setting the kernel to 0 for longer paths. The alternative solution of normalizing the kernel leads to a slight decrease in accuracy. The fact that longer paths have larger kernel scores in the unnormalized version does not pose a problem because, by definition, paths of different lengths correspond to disjoint sets of features. Consequently, the SVM algorithm will induce lower weights for features occurring in longer paths, resulting in a linear separator that works irrespective of the size of the dependency paths.

5 Future Work

There are cases when words that do not belong to the shortest dependency path do influence the extraction decision. In Section 3.2, we showed how negative polarity items are integrated in the model through annotations of words along the dependency paths. Modality is another phenomenon that is in-

fluencing relation extraction, and we plan to incorporate it using the same annotation approach.

The two relation extraction methods are very similar: the subsequence patterns in one kernel correspond to dependency paths in the second kernel. More exactly, pairs of words from a subsequence pattern correspond to pairs of consecutive words (i.e. edges) on the dependency path. The lack of dependency information in the subsequence kernel leads to allowing gaps between words, with the corresponding exponential penalty factor λ. Given the observed similarity between the two methods, it seems reasonable to use them both in an integrated model. This model would use high-confidence head-modifier dependencies, falling back on pairs of words with gaps, when the dependency information is unreliable.

6 Conclusion

Mining knowledge from text documents can benefit from using the structured information that comes from entity recognition and relation extraction. However, accurately extracting relationships between relevant entities is dependent on the granularity and reliability of the required linguistic analysis. In this chapter, we presented two relation extraction kernels that differ in terms of the amount of linguistic information they use. Experimental evaluations on two corpora with different types of discourse show that they compare favorably to previous extraction approaches.

7 Acknowledgement

This work was supported by grants IIS-0117308 and IIS-0325116 from the NSF. We would like to thank Arun Ramani and Edward Marcotte for their help in preparing the AIMed corpus.

References

1. R. J. Mooney, R. C. Bunescu, Mining knowledge from text using information extraction, SIGKDD Explorations (special issue on Text Mining and Natural Language Processing) 7 (1) (2005) 3–10.
2. H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, C. Watkins, Text classification using string kernels, Journal of Machine Learning Research 2 (2002) 419–444.
3. C. D. Fellbaum, WordNet: An Electronic Lexical Database, MIT Press, Cambridge, MA, 1998.
4. L. R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, Proceedings of the IEEE 77 (2) (1989) 257–286.

5. A. McCallum, D. Freitag, F. Pereira, Maximum entropy Markov models for information extraction and segmentation, in: Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000), Stanford, CA, 2000.
6. J. Lafferty, A. McCallum, F. Pereira, Conditional random fields: Probabilistic models for segmenting and labeling sequence data, in: Proceedings of 18th International Conference on Machine Learning (ICML-2001), Williamstown, MA, 2001, pp. 282–289.
7. C. Blaschke, A. Valencia, Can bibliographic pointers for known biological data be found automatically? protein interactions as a case study, Comparative and Functional Genomics 2 (2001) 196–206.
8. C. Blaschke, A. Valencia, The frame-based module of the Suiseki information extraction system, IEEE Intelligent Systems 17 (2002) 14–20.
9. R. Bunescu, R. Ge, R. J. Kate, E. M. Marcotte, R. J. Mooney, A. K. Ramani, Y. W. Wong, Comparative experiments on learning information extractors for proteins and their interactions, Artificial Intelligence in Medicine (special issue on Summarization and Information Extraction from Medical Documents) 33 (2) (2005) 139–155.
10. V. N. Vapnik, Statistical Learning Theory, John Wiley & Sons, 1998.
11. N. Cristianini, J. Shawe-Taylor, An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, Cambridge University Press, 2000.
12. National Institute of Standards and Technology, ACE – Automatic Content Extraction, http://www.nist.gov/speech/tests/ace (2000).
13. M. Steedman, The Syntactic Process, MIT Press, Cambridge, MA, 2000.
14. J. Hockenmaier, M. Steedman, Generative models for statistical parsing with combinatory categorial grammar, in: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL-2002), Philadelphia, PA, 2002, pp. 335–342.
15. R. C. Bunescu, R. J. Mooney, A shortest path dependency kernel for relation extraction, in: Proceedings of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP-05), Vancouver, BC, 2005, pp. 724–731.
16. M. J. Collins, Three generative, lexicalised models for statistical parsing, in: Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL-97), 1997, pp. 16–23.
17. A. Culotta, J. Sorensen, Dependency tree kernels for relation extraction, in: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04), Barcelona, Spain, 2004, pp. 423–429.
18. D. Zelenko, C. Aone, A. Richardella, Kernel methods for relation extraction, Journal of Machine Learning Research 3 (2003) 1083–1106.