Beyond Myopic Inference in Big Data Pipelines

Karthik Raman, Adith Swaminathan, Johannes Gehrke, Thorsten Joachims
Dept. of Computer Science
Cornell University
{karthik,adith,johannes,tj}@cs.cornell.edu

ABSTRACT

Big Data Pipelines decompose complex analyses of large data sets into a series of simpler tasks, with independently tuned components for each task. This modular setup allows re-use of components across several different pipelines. However, the interaction of independently tuned pipeline components yields poor end-to-end performance as errors introduced by one component cascade through the whole pipeline, affecting overall accuracy. We propose a novel model for reasoning across components of Big Data Pipelines in a probabilistically well-founded manner. Our key idea is to view the interaction of components as dependencies on an underlying graphical model. Different message passing schemes on this graphical model provide various inference algorithms to trade-off end-to-end performance and computational cost. We instantiate our framework with an efficient beam search algorithm, and demonstrate its efficiency on two Big Data Pipelines: parsing and relation extraction.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Experimentation, Theory

Keywords

Big Data Pipelines, Modular Design, Probabilistic Inference

1. INTRODUCTION

Benevolent pipelines propagate knowledge. — Adapted from a quotation in the Sama Veda.

Unlocking value from Big Data is a hard problem. The "three V's" of the data (volume, velocity, and variety) require setting up sophisticated *Big Data Pipelines* — workflows of tasks such as data extraction and transformation, feature creation, model construction, testing, and visualization. In order not to have to develop components for each task from scratch, data scientists build Big Data Pipelines using existing software components, for example, a natural language parser or a text classifier.

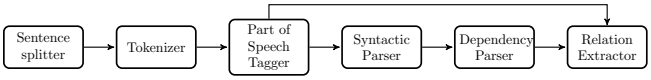


Figure 1: Big Data Pipeline for relation extraction.

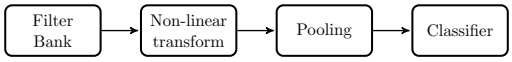


Figure 2: Object Detection & Recognition pipeline.

An example of a Big Data Pipeline for relation extraction from text is given in Figure 1. After sentence splitting and tokenization, a part-of-speech (POS) tagger feeds into a syntactic parser, which is the basis for a dependency parser. Finally, the POS tags and parses provide features for the relation extractor. The components of this pipeline are also used in pipelines for other natural language analysis tasks like joint entity and relation extraction [30], sentiment analysis [23] or opinion extraction [5]. Another example of a Big Data Pipeline from computer vision [14] is shown in Figure 2, where low-level filters (e.g., edge detectors) eventually feed into an object recognizer (e.g., an SVM) after several transformation and pooling operations. Similarly, such pipelines also exist in other domains, ranging from robotics [1] to computational biology [31].

The speed of plug-and-play development of Big Data Pipelines comes with a big drawback: the composition and resulting execution of the pipeline is rather ad-hoc. In particular, we lack a model for global reasoning about the uncertainty that is associated with the predictions of each task in the pipeline, even though components maintain and could provide a wealth of relevant information. In the default approach, where each stage merely passes its canonical output to the next stage, any error in the early stages can lead to cascading and unrecoverable errors in later stages. Each pipeline component makes a locally optimal choice without consideration that it is only one of the steps in a long chain.

In this paper, we present a formal model of inference over Big Data Pipelines in a probabilistically well-founded manner. Our model accounts for and utilizes the uncertainty in the predictions at all stages of the pipeline. We propose inference algorithms that allow us to trade off computational efficiency versus overall quality of the pipeline by optimizing how much

KDD'13, August 11–14, 2013, Chicago, Illinois, USA.

uncertainty is propagated. Our inference algorithms have an elegant formulation in relational algebra which may be of independent importance due to its connections to probabilistic databases and data provenance [2, 29].

In particular, the paper makes the following contributions. First, we formalize pipeline inference as a graphical model in which nodes are processing components and edges denote data flow between components. With this model, we have a formal foundation for reasoning across a whole Big Data Pipeline (Section 2). Second, we propose inference algorithms that resolve uncertainty across the pipeline, including a beam search algorithm that adaptively increases the amount of information that is propagated. We show that these inference algorithms can be formulated in relational algebra well-suited for probabilistic database implementation (Section 3). Third, in a comprehensive experimental evaluation, we demonstrate how our approach gracefully trades off the quality of predictions and computational efficiency of inference on two real-world tasks: a Syntactic Parsing pipeline and a Relation Extraction pipeline (Section 4). We conclude with a discussion of our approach in Section 5 and its connection to related work in Section 6.

2. PROBABILISTIC PIPELINE MODEL

Given an input x, how should we compute the output y of a pipeline with multiple components? The answer to this question depends on the API that each component exposes. In the most basic API, we may only assume that a component takes input x (e.g., sequence of words) and outputs an answer $y^{(1)}$ (e.g., sequence of part-of-speech tags). We call this $y^{(1)}$ the *canonical output*. With this simplistic API for each component, it is unclear how cascading errors can be avoided.

Fortunately, many pipeline components permit a more refined API in two respects. First, a component θ_i can provide a confidence score, interpreted as the conditional probability $\Pr(y \mid x, \theta_i)$, for the returned output $y^{(1)}$. Second, a component can not only return the canonical output $y^{(1)}$, which most naturally is the maximizer of the conditional probability *i.e.* $y^{(1)} = \operatorname{argmax}_y \Pr(y \mid x, \theta_i)$, but also all of the top-k highest scoring predictions

$$[y^{(1)}, \ldots, y^{(k)}] = \text{k-}\operatorname*{argmax}_y \Pr(y \mid x, \theta_i)$$

with their respective conditional probabilities:

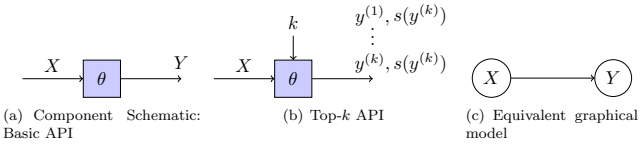
$$[\Pr(y^{(1)} \mid x, \theta_i), \Pr(y^{(2)} \mid x, \theta_i), \ldots, \Pr(y^{(k)} \mid x, \theta_i)]$$

In contrast to the basic API that only returns the canonical output, we call this API the *Top-k* API.

Many standard components (e.g., POS taggers, parsers) already provide this functionality. Even non-probabilistic models (e.g., SVMs) can typically be transformed to produce reasonable conditional probability estimates [25].

The Top-k API allows us to interact with components as if they were truly black boxes. We do not assume we have access to θ_i, nor the form of the internal probabilistic model. This is important in the Big Data application setting where re-training pipeline components for a new domain imposes a significant scalability bottleneck, and plug-and-play development of pipelines is desirable.

We now explore how the information provided by this extended interface can be used to not only make pipeline predictions more reliable and robust, but also to provide probabilistic outputs for the pipeline as a whole.



(a) Component Schematic: Basic API (b) Top-k API (c) Equivalent graphical model

Figure 3: API of a black box component.

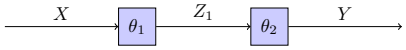


Figure 4: Composition of two components.

2.1 Pipelines as Graphical Models

A note about notation: We denote specific values using lower-case letters and the set of all possible values by upper-case letters. This is consistent with the usual notation for random variables. For instance, we denote any general output of a component by Y, and a specific output by y.

To derive a general model of pipeline inference, let us for now assume that each component, when provided an input x, exposes the full distribution $\Pr(Y \mid x, \theta_i)$ through the API. Equivalently, one may view this as providing the top-k, but for a large enough value of k that enumerates all possible y. Under this API for each component, what should the final pipeline output be and what confidence should the pipeline assign to this output?

Assume that each component θ_i in the pipeline is a trained or hand-coded model. We shall use $s_{x,i}(y)$ as shorthand for $\Pr(y \mid x, \theta_i)$. When it is clear from the context which component and input is being talked about, we shall drop the subscript to yield $s(y)$.

The schematic of a black box component is illustrated in Figure 3. It is very natural to view input X and output Y as random variables, and the component θ specifies the conditional dependencies between X and Y. Note that this is a general definition of a component, encompassing deterministic rule-based transformations (the conditional distribution is such that all the probability mass is concentrated on the mapping of the input x) as well as generative and discriminative probabilistic models.

To provide intuition for the inference problem, first consider a pipeline of only two components, θ_1 and θ_2, where the output of θ_1 is piped to the second component. This is illustrated in Figure 4. Let the input to the pipeline be X, the outputs of θ_1 and θ_2 be Z_1 and Y respectively. From Figure 3, we know that this is equivalent to a graphical model with three nodes X, Z_1, Y. The law of joint probability gives us

$$\begin{aligned}\Pr(Y \mid X) &= \sum_{Z_1} \Pr(Y, Z_1 \mid X) \\ &= \sum_{Z_1} \Pr(Y \mid Z_1, \theta_2) \Pr(Z_1 \mid X, \theta_1), \qquad (1)\end{aligned}$$

where (1) follows from the conditional independence induced by the graphical model. Thus, for a particular input x, the ideal output according to Equation 1 would be:

$$y^{\text{Bayes}} = \operatorname*{argmax}_y \sum_{z_1} \Pr(y \mid z_1, \theta_2) \Pr(z_1 \mid x, \theta_1) \qquad (2)$$

We call y^{Bayes} the *Bayes Optimal* output since it considers all possible intermediate outputs in determining the prediction.

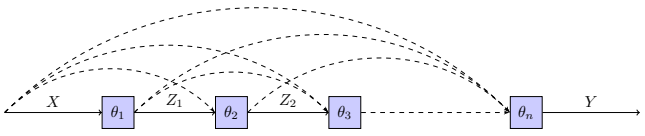


Figure 5: Schematic of a general pipeline.

Going beyond the composition of two components, any general pipeline can be viewed as a Directed Acyclic Graph, where the nodes are the black box components and the edges denote which outputs are piped as inputs to which components. The basic idea illustrated above for a linear two-step pipeline can be extended to general pipelines such as the one illustrated in Figure 5. Any general pipeline structured as a DAG admits a topological ordering. Denoting the input to the pipeline as X, eventual output Y, and the output of each intermediate stage i in this topological ordering as Z_i, then any acyclic pipeline is a subgraph of the graph shown in Figure 5. From our modeling assumption, this DAG is an equivalent representation of a directed graphical model: each of $X, Z_1, Z_2 \ldots Y$ are nodes denoting random variables, and the components enforce conditional independences. The graphical model corresponding to Figure 5 makes the fewest independence assumptions across stages.

Analogous to the two stage pipeline from above, the Bayes Optimal prediction y^{Bayes} for general pipelines is

$$y^{\text{Bayes}} = \operatorname*{argmax}_y \sum_{z_1} \Big[\Pr(z_1 \mid x, \theta_1) \sum_{z_2} \big[\Pr(z_2 \mid x, z_1, \theta_2) \sum_{z_3} \Big(\Pr(z_3 \,|\, x, z_1, z_2, \theta_3) \ldots \Pr(y \,|\, x, z_1 \ldots z_{n-1}, \theta_n) \Big) \ldots \big] \Big] \quad (3)$$

Computing the Bayes Optimal y^{Bayes} is infeasible for real-world pipelines because it requires summing up over all possible intermediate output combinations $z_1, \ldots z_{n-1}$. We will therefore explore in the following section how this intractable sum can be approximated efficiently using the information that the Top-k API provides.

2.2 Canonical Inference

Even the naive model of passing only the top output to the next stage can be viewed as a crude approximation of the Bayes Optimal output. We call this procedure the *Canonical Inference* algorithm. For simplicity, consider again the two component pipeline, such that the naive model results in

$$z_1{}^{(1)} = \operatorname*{argmax}_{z_1} \Pr(z_1 \mid x, \theta_1)$$
$$y^{\text{Can}} = \operatorname*{argmax}_y \Pr(y \mid z_1{}^{(1)}, \theta_2) \quad (4)$$

The optimization over y in (4) is unaffected if it is multiplied by the term $\Pr(z_1{}^{(1)} \mid x, \theta_1)$. Once the summation in the marginal in (1) is restricted to $z_1{}^{(1)}$, the sum reduces to

$$\Pr(y \mid z_1{}^{(1)}, \theta_2) \Pr(z_1{}^{(1)} \mid x, \theta_1) \leq \Pr(y \mid x).$$

Hence, this objective provides a lower bound on the true $\Pr(y \mid x)$. It is easy to see that this holds true for general pipeline architectures too, where the canonical inference procedure proceeds stage-by-stage through the topological ordering. For $i = 1, \ldots, n$, define:

$$z_i{}^{(1)} = \operatorname*{argmax}_{z_i} \Pr(z_i \mid x, z_1{}^{(1)}, \ldots, z_{i-1}{}^{(1)}, \theta_i)$$
$$y^{\text{Can}} = \operatorname*{argmax}_y \Pr(y \mid x, z_1{}^{(1)}, \ldots, z_{n-1}{}^{(1)}, \theta_n) \quad (5)$$

then $\Pr(y \,|\, z_1{}^{(1)}, \ldots, z_{n-1}{}^{(1)}, \theta_n) \ldots \Pr(z_1{}^{(1)} \,|\, x, \theta_1) \leq \Pr(y \,|\, x)$.

In this sense, canonical inference is optimizing a crude lower bound to the Bayes Optimal objective (again, the terms $\Pr(z_1{}^{(1)} \mid x, \theta_1)$ etc. do not affect the optimization over y). This suggests how inference can be improved, namely by computing improved lower bounds on $\Pr(y|x)$. This is the direction we take in Section 3, where we exploit the Top-k API and explore several ways for making the computation of this bound more efficient.

3. EFFICIENT INFERENCE IN PIPELINES

Canonical inference and full Bayesian inference are two extremes of the efficiency vs. accuracy tradeoff. We now explore how to strike a balance between the two when pipeline components expose the Top-k API. In particular, given input x suppose each component θ is capable of producing a list of k outputs $\left[y^{(1)}, y^{(2)}, \ldots y^{(k)}\right]$ along with their conditional probabilities $\left[\Pr(y^{(1)} \mid x, \theta), \ldots \Pr(y^{(k)} \mid x, \theta)\right]$ (in decreasing order). We now outline three algorithms, characterize their computational efficiency, and describe their implementation in terms of relational algebra. This interpretation enables direct implementation of these algorithms on top of traditional/probabilistic database systems.

3.1 Top-k Inference

The natural approach to approximating the full Bayesian marginal (Eq. 3) when computation is limited to a few samples is to choose the samples as the top-k realizations of the random variable, weighted by their conditional probabilities. We call this Top-k Inference, which proceeds in an analogous fashion to canonical inference, stage-by-stage through the topological ordering. However, inference now becomes more involved. Each component produces a scored list, and each item in the lists generated by predecessors will have a corresponding scored list as output.

Denote the list of outputs returned by the component θ_i (each of which are possible values for the random variable Z_i in the equivalent graphical model) as $\{z_i\}_k$. Note that this set depends on the inputs to θ_i, but we omit this dependence from the notation to reduce clutter. This gives rise to the following optimization problem for identifying the eventual output of the pipeline.

$$y^{\text{TopK}} = \operatorname*{argmax}_y \sum_{\{z_1\}_k} \Big[\Pr(z_1 \mid x, \theta_1) \sum_{\{z_2\}_k} \big[\Pr(z_2 \mid x, z_1, \theta_2) \sum_{\{z_3\}_k} \Big(\Pr(z_3 \,|\, x, z_1, z_2, \theta_3) \ldots \Pr(y \,|\, x, z_1, \ldots z_{n-1}, \theta_n) \Big) \ldots \big] \Big] \quad (6)$$

A simple extension of top-k inference is to have a parameter k_i for each component θ_i that allows variable length lists of outputs $\{z_i\}_{k_i}$ for different components in the pipeline. We refer to these parameters collectively as $\vec{\boldsymbol{k}}$.

Consider again the simple two stage pipeline in Figure 4. Each component now operates on relational tables and generates relational tables. We initialize the table T_0 with the input to the pipeline x:

IX_0	$VALUE_0$	$SCORE_0$
1	x	1

The subscripts in the attributes refers to the component ID, *IX* denotes the index into the scored list of outputs produced by the component, *VALUE* is the actual output and *SCORE* is its conditional score. Now, given a parameter vector $\vec{k} = (k_1, k_2)$, the first component will generate the table T_1.

IX_0	IX_1	$VALUE_1$	$SCORE_1$
1	1	${z_1}^{(1)}$	$s_{x,1}({z_1}^{(1)})$
1	2	${z_1}^{(2)}$	$s_{x,1}({z_1}^{(2)})$
...	...	...	...
1	k_1	${z_1}^{(k_1)}$	$s_{x,1}({z_1}^{(k_1)})$

Note that IX_0 carries no additional information, if component 1 is the only successor to component 0 (as is the case in this example). However, in a general pipeline, we need columns like IX_0 to uniquely identify inputs to a component.

The second component now takes the table T_1 and produces the final table T_2.

IX_0	IX_1	IX_2	$VALUE_2$	$SCORE_2$
1	1	1	${z_2}^{(1,1)}$	$s_{{z_1}^{(1)},2}({z_2}^{(1,1)})s_{x,1}({z_1}^{(1)})$
1	1	2	${z_2}^{(1,2)}$	$s({z_2}^{(1,2)})s({z_1}^{(1)})$
...	...	...	...	...
1	1	k_2	${z_2}^{(1,k_2)}$	$s({z_2}^{(1,k_2)})s({z_1}^{(1)})$
1	2	1	${z_2}^{(2,1)}$	$s({z_2}^{(2,1)})s({z_1}^{(2)})$
...	...	...	...	...
1	k_1	k_2	${z_2}^{(k_1,k_2)}$	$s({z_2}^{(k_1,k_2)})s({z_1}^{(k_1)})$

The final output for top-k inference y^{TopK} is generated by grouping the values $T_2.VALUE_2$, aggregating by $SCORE_2$ and selecting the element with largest sum. For linear pipelines (Figure 5 without the curved edge dependencies), this procedure is summarized in Algorithm 1. The linear pipeline has the nice property that each component has exactly one successor, so we need not store additional information to trace the provenance of each intermediate output. We assume each component provides the interface *GenTopK*(*ID*, *k*, *INPUT*) which returns a table with attributes $VALUE_{ID}$, $SCORE_{ID}$ that has k tuples. For the n-stage linear pipeline, performing top-k inference with parameters $\vec{k}$ will yield a final table T_n with at most $k_1 k_2 \ldots k_n$ tuples. There may be fewer tuples, since any output generated by different combinations of intermediate outputs results in only one tuple.

For top-k inference in general pipelines, the algorithm needs to maintain additional information to uniquely identify each input that a component processes. Specifically, for a component θ_i, let the indices of its parents in the DAG be denoted by J. We need to construct a table that θ_i can process, which has all compatible combinations of outputs in tables $\{T_j\}_{j \in J}$. This procedure is outlined in Algorithm 2. Let the lowest common ancestor of two parents of θ_i (say, θ_p and θ_q) be denoted by θ_m. If θ_p and θ_q do not share an ancestor, m defaults to 0. The only compatible output combinations between θ_p and θ_q are those that were generated from the same output from θ_m. We store columns IX_* in our tables to identify precisely this fact: T_i traces all ancestors of the component θ_i. Hence, the natural join of tables T_p and T_q will match on the *IX* attributes corresponding to

Algorithm 1 Top-k Inference for Linear Pipelines.

Require: $GenTopK(id, k, x)$, $\vec{k}$
  $T_0 \leftarrow$ CreateTable($VALUE_0$:X ; $SCORE_0$:1)
  for $i = 1$ **to** n **do**
    $T_i \leftarrow \phi$
    for t **in** T_{i-1} **do**
      $x \leftarrow t.VALUE_{i-1}$
      $s \leftarrow t.SCORE_{i-1}$
      $OP \leftarrow GenTopK(i, \vec{k_i}, x)$
      $OP.SCORE_i \leftarrow s * OP.SCORE_i$
      $T_i \leftarrow T_i \cup OP$
    end for
    if $|\{Successors(i)\}| \leq 1$ **then**
      $T_i \leftarrow Sum(T_i.SCORE_i, GroupBy(VALUE_i))$
    end if
    $T_i \leftarrow Transform(T_i)$ {Section 3.2, 3.3}
  end for
  return $Max(T_n.SCORE_n)$

Algorithm 2 Creating Compatible Inputs.

Require: J
1: $p \leftarrow \max J$
2: $T \leftarrow T_p$
3: $A \leftarrow J \setminus p$
4: **repeat**
5: $q \leftarrow \max A$
6: $r \leftarrow LowestCommonAncestor(p, q)$
7: $T \leftarrow T \bowtie T_q$
8: $T.SCORE_r \leftarrow T.SCORE_p * \frac{T_q.SCORE_q}{T_r.SCORE_r}$
9: $A \leftarrow A \cup \{r\} \setminus \{p, q\}$
10: $p \leftarrow r$
11: **until** $|A| \leq 1$
12: **return** T

their common ancestors up to m, as shown in lines 6-7 in Algorithm 2. Furthermore, we need to update the probability mass for each compatible output combination. From (6), we see that the product of the scores in T_p and T_q will have the contribution from their common ancestors double-counted. Hence, in line 8 of the algorithm, we divide out this score.

With this algorithm, and the understanding that *GenTopK*(*ID*, *k*, *INPUT*) augments its returned table with IX_{ID} information, inference proceeds exactly as the algorithm for the linear pipeline (Algorithm 1) with T_{i-1} replaced by the output of Algorithm 2 given *Parents*(i) as input. It should now be clear why we always check the *Successors*(i) before aggregating tuples in a table: Such an aggregation step loses provenance information (in the form of *IX* attributes) which is necessary for correct recombination later in the pipeline.

3.2 Fixed Beam Inference

The top-k inference procedure described above follows Equation (6) exactly. However, this can result in a multiplicative increase in the sizes of the intermediate tables, with the eventual output requiring aggregation over potentially $k_1 k_2 \ldots k_n$ outputs. This may become infeasible for even moderate pipeline sizes and modest choices for $\vec{k}$.

One way to control this blowup is to interpret the parameters $\vec{k}$ as limits on the sizes of the intermediate tables. In particular, at each stage i of the pipeline, we sort table T_i

Algorithm 3 Adaptive Deepening.

Require: $\delta, [s_1, \ldots, s_k]$
  $d \leftarrow 0$
  repeat
    $d \leftarrow d + 1$
    $r \leftarrow s_{d+1}/s_d$
  until $d \geq k - 1$ **or** $r < \delta$
  return $[s_1, \ldots, s_d]$

according to the $SCORE_i$ and only retain the top k_i tuples. This pruning can be implemented as the *Transform* function of Algorithm 1. We refer to this inference method as Beam-M Inference. This strategy provides an optimization objective that is a scalable intermediate between canonical inference and top-k inference.

3.3 Adaptive Inference

Since it may be difficult to set the parameters $\vec{k}$ limiting the beam width a priori, we now explore an adaptive inference method. Intuitively, the beam width k_i of component θ_i should depend on the uncertainty in its prediction. If that component had perfect confidence, doing canonical inference for this step suffices to yield the Bayes optimal solution since all of the probability mass is concentrated on a single prediction. If a component is less confident, the beam width should be larger. A larger beam is computationally more intensive but promises to be a more robust estimate of the Bayes optimal objective.

This leads to the operational goal of adapting the beam width to ensure that each result list includes all predictions that have high probability, but ideally not many more so that efficiency does not suffer. This can be operationalized by looking for sharp drops in the conditional probability as the algorithm goes down the scored list of predictions generated by a component. To control this process, we not only have the maximum beam width $\vec{k}$ like in top-k inference, but also a vector of parameters $\vec{\delta} = [\delta_1, \delta_2, \ldots, \delta_n]$ that specify the drop threshold for each component. As outlined in Algorithm 3, we only deepen the beam (up to the maxima specified by $\vec{k}$), if the next prediction in the top-k list has a sufficiently large (as specified by $\vec{\delta}$) conditional probability relative to the previous element of the list. We typically set the value of $\vec{\delta}$ to be the same for all components. Adaptive Inference also proceeds through the topological ordering as in Algorithm 1, with $AdaptiveDeepening(\delta_i, k_i)$ (Algorithm 3) being used as the *Transform* function.

4. EXPERIMENTS

The following experiments evaluate the proposed inference methods on three pipelines.

Parse Pipeline: The *syntactic parsing* pipeline is an important NLP task and a common sub-pipeline in many application problems. The pipeline consists of two stages, namely POS tagging followed by parsing. We use the benchmark Wall Street Journal Treebank dataset [20] with the standard setup: POS tagger and parser with models trained on Sections 2-21, the remaining sections are used for evaluation. This leads to a test set of 9370 sentences. We used two popular off-the-shelf components for this pipeline: the HMM-based LingPipe system [4] to predict POS tags and the phrase grammar-based Enju parser [26]. There was no modification required to use these systems, as they both already expose interfaces to get the top k outputs. In addition to the standard Bracketed F1-measure [24], we also report 0/1-loss performance (*i.e.,* the number of sentences with the correct parse tree predicted) denoted as $\#Cor$.

Relation Extraction (RelEx) Pipeline: We built a three-stage *Relation Extraction* (RE) pipeline similar to the one from Figure 1, which adds the RE stage at the end of the POS/parsing pipeline. We use the benchmark ACE 2004 dataset, which contains sentences labeled with entity-pairs along with the corresponding relation type. As in previous studies [22], the labeled relation types are collapsed into seven pre-defined categories. To extract relations we used the RECK relation extraction system [22], which employs an SVM with a convolution kernel. Given a sentence, its POS tag sequence, and its parse tree, and a pair of entities, the SVM predicts the relation between them. RECK predicts the class with the highest score if it exceeds a fixed pre-defined threshold τ. Since RECK uses the Stanford parser [15], so do we. Lingpipe was used for POS tagging.

A large fraction of this dataset can be trivially classified using just the sentence tokens. Thus in order to better study the impact of pipeline inference, we use a subset of the data which consists of the "difficult" examples: all examples where the predicted relation differs for at least one of the candidate *(POS sequence, parse tree)* values. The candidate set consists of the top 9 POS tag sequences predicted by LingPipe and the top 5 parses predicted by the Stanford Parser for each tag sequence (our results are stable to a broad choice of these numbers). This leads to a set of 11016 examples.

Mean average precision (MAP) over the difficult examples is used as the performance measure, computed by the ranking scores of all examples for each of the seven relations. To compare with results from the literature, we also report the precision of the classification (using the same pre-defined threshold for prediction τ as in [22]).

Synthetic Data (Synth) Pipeline: We also report results for a synthetic pipeline, since this allowed us to investigate a wide range of scenarios, such as the amount of noise at each stage of the pipeline. To generate this data, we used the following model.

Each black box in the pipeline is represented by two distributions: the true distribution $\Pr(\cdot \mid \cdot)$ (instantiated as a Dirichlet distribution $Dir(\alpha)$) mapping inputs to outputs and the learned distribution $\hat{\Pr}(\cdot \mid \cdot)$. The learned distribution is a mixture of the true distribution and a $Dir(\alpha)$ noise distribution. This captures the notion of noisy training without having to materialize a training dataset or algorithm. The default mixture weight for the noise was set to 10%. The true label for an example is obtained as the Bayes Optimal value using the true distributions.

By default, we use a three-stage pipeline. The first black box takes in one of 500 input values and outputs one of 50 outputs. Using the above as input, the second black box outputs one of 20 outputs, which is used in the final stage to generate one of 10 possible outputs. We report 0/1 accuracies for these final outputs, averaged over 500 synthetic pipelines drawn from $Dir(\alpha)$.

4.1 Results

We now show the results of our experimental study. In the remainder, we distinguish between *performance* of the inference, which is measured by problem-specific quality

Data	Top-1 (k=1)	Top-k		
		k = 2	k = 3	k = 5
Parse	83.0	83.9	84.4	85.3
RelEx	35.0	35.9	36.4	37.0
Synth	25.4	34.4	44.2	59.2

Table 1: Performance of Top-k inference.

Data	Top-1 (M=1)	Beam-M				
		M=2	M=3	M=5	M=7	M=15
Parse	83.0	83.9	84.4	85.3	85.7	85.9
RelEx	35.1	35.5	35.7	36.3	36.7	37.0
Synth	25.4	30.2	34.5	42.2	49.5	74.2

Table 4: Performance of Beam-M inference.

measures and *computational cost*, which we measure by the number of calls to the black box components in the pipeline.

4.1.1 Performance of Top-k

The first experiment verifies whether Top-k inference indeed provides better prediction performance than the canonical Top-1 inference. The results for the three pipelines are shown in Table 1. $\vec{\boldsymbol{k}}$ is set so that each component returns the same number of outputs. All measures are on a 0-100 scale – the larger the better.

We can see that going from Top-1 to even just Top-2 has a substantial impact on all the pipelines; for instance, we see a $\sim 3\%$ improvement in the MAP score for Relation Extraction. With increasing k, the performance continues to improve monotonically.

Closer inspection reveals that this difference is largely due to the errors made by the Top-1 inference in the early stages of the pipeline. On the *Parse* pipeline, for example, using the *True* POS tag sequence (available from the labeled test set), instead of the Lingpipe output, we obtain a significantly higher F-1 score of 89.1% even when using top-1 in the second stage. Similarly, in the *Synth* pipeline, the Bayes-Optimal solution using the noisy learned distributions $\hat{\Pr}(\cdot \mid \cdot)$ leads to an accuracy of 90.1%. Using Top-k inference narrows the gap between the naive Top-1 baseline and these skylines by overcoming early-stage errors in the pipeline.

4.1.2 The Choice of k Throughout The Pipeline

How should we set k in the pipeline — should we use the same k for all components or should k vary by component? To answer this question, we experimented with the two real-world pipelines using up to nine POS tag sequences and five parse trees. The results are shown in Tables 2 and 3. We can see that using a larger k helps more for tagging than parsing. For instance, using the top-5 POS tag sequences and the top parse tree leads to larger improvements than using the top-5 parse trees and the top POS tag sequence across both pipelines. This indicates that the gains observed are mainly due to "fixing" errors in early stages of the pipeline. Sending more outputs from the early stages provides a mechanism for the pipeline to recover the true eventual output even if the top intermediate output is incorrect. Thus including a sufficiently good POS sequence in the top-k improves the confidence of the correct parse. This effect is particularly pronounced in the two-stage parsing pipeline (compared to the three-stage RelEx pipeline) as here *fixing errors* in the POS tag sequence has a larger impact on the eventual output.

Data	Top 1	Top-k			Beam-M		
		2	3	5	2	3	5
Parse	83.4	84.7	85.3	85.9	84.7	85.3	85.9
RelEx	34.3	35.3	35.5	35.4	35.5	35.4	35.8
Synth	24.7	33.0	40.0	52.5	28.7	31.9	37.7

Table 5: The effect of noise in the learned models.

These performance improvements exist for nearly every combination of k values for both datasets. In particular, using only the top-1 parse tree but all the top-9 tag sequences leads to more than 250 additional sentences parsed correctly and a gain of $\sim 10\%$ in precision for the RelEx pipeline. Thus, if we have limited resources, we should use different values of k throughout the pipeline.

4.1.3 Performance of Beam-M

Next we investigate if the computationally cheaper *Beam-M* inference also improves over Top-1 inference. We use a fixed beam width M across all stages of the pipeline. Table 4 shows the results of varying M for the three pipelines. As with Top-k inference, we find that deepening the beam by even one ($M = 2$) leads to improvements on all pipelines. Further increasing M provides additional gains. Again, it seems helpful to pass more information at earlier stages of the pipeline. Comparing $k=3$ and $M=7$ for the RelEx Pipeline shows that passing on more (3 vs 7) POS tag sequences helps despite passing fewer (9 vs 7) parse trees.

Note that the amount of information that is passed between components is constant in Beam-M, while it can grow exponentially with pipeline length for Top-k. Scalability can be further improved by reducing the beam for components that are computationally intensive.

4.1.4 Adaptive Inference

Our results indicate that the inference algorithms can trade off computational cost and performance by varying k or M. Algorithm 3 automatically selects the beam size individually for each pipeline component on a per-example basis. We ran Algorithm 3 for different values of $\delta \in [0.001, 0.999]$. For each run, we set identical values of δ for all components in the pipeline. We measured the computational cost in terms of the sum of black box calls.

Figure 6 shows the tradeoff achieved by this deepening strategy. In all pipelines, we find a diminishing return in performance gain when we increase computation. This means that we can achieve significant improvement in performance with a small computational overhead as compared to the canonical Top-1 approach. As the computational cost is monotone in δ, we can estimate the value of δ that fits within the available computational budget (and maximizes performance) by profiling over a small sample of examples. Other beam deepening strategies lead to similar results, but are omitted for brevity. Our cost model assumed a uniform cost across pipeline components. However, more sophisticated cost models could plausibly be incorporated and further extensions to adaptive deepening are discussed in Section 5.

4.1.5 The Effect of Noise in the Learned Models

How robust are Top-k and Beam-M inference algorithms to errors in the predictions of the components? To answer this question, we experimented using different sources of noise and error in the learned models. A common source of error is that models may have been trained on out-of-domain

Number of Parse Trees	Number of POS Sequences													
	1		2		3		4		5		7		9	
	F1	#Cor	F1	#Cor	F1	#Cor	F1	#Cor	F1	#Cor	F1	#Cor	F1	#Cor
1	83.0	2629	83.9	2749	84.4	2820	84.9	2841	85.2	2864	85.7	2886	85.9	2893
2	83.0	2629	83.9	2750	84.4	2823	84.9	2843	85.2	2868	85.7	2893	85.9	2899
3	83.0	2630	83.9	2748	84.4	2823	84.9	2841	85.3	2865	85.7	2892	85.9	2900
4	83.0	2630	83.9	2750	84.4	2824	84.9	2843	85.2	2865	85.7	2891	85.9	2899
5	83.0	2634	83.9	2753	84.4	2828	84.9	2847	85.3	2868	85.7	2894	85.9	2902

Table 2: Results of Top-k inference for Parse pipeline on varying k for the tagging and parsing components.

Number of Parse Trees	Number of POS Sequences													
	1		2		3		4		5		7		9	
	MAP	Prec	MAP	Prec	MAP	Prec	MAP	Prec	MAP	Prec	MAP	Prec	MAP	Prec
1	35.1	68.5	35.4	70.1	35.8	72.6	36.0	72.3	36.4	74.4	36.6	77.5	36.8	75.9
2	35.5	70.0	35.9	69.8	36.1	72.0	36.4	72.0	36.6	74.1	36.6	75.9	36.8	75.3
3	35.7	70.3	36.0	70.9	36.4	72.6	36.5	72.3	36.8	74.7	36.7	76.2	36.9	75.6
4	35.8	69.2	36.2	70.9	36.4	72.6	36.6	72.9	37.0	73.8	37.0	76.5	37.1	75.9
5	35.9	68.9	36.3	70.9	36.5	72.6	36.7	72.6	37.0	73.5	37.1	76.2	37.3	75.6

Table 3: Results of Top-k inference for RelEx pipeline on varying k for the tagging and parsing components.

Length	Top 1	Top-k			Beam-M		
		2	3	5	2	3	5
2	48.7	57.4	65.1	76.1	57.4	65.1	76.1
3	24.4	35.0	45.1	62.3	30.6	36.4	46.1
4	17.2	29.4	44.2	64.9	21.6	26.6	34.5
5	16.0	33.3	46.0	71.6	20.1	22.5	31.8

Table 6: Effect of pipeline length for Synth.

α	Top 1	Top-k			Beam-M		
		2	3	5	2	3	5
0.1	58.6	75.0	85.1	94.8	70.9	78.6	87.5
0.2	42.1	56.8	68.0	83.2	52.1	59.4	70.2
0.5	24.4	35.0	45.1	62.3	30.6	36.4	46.1
1	17.1	24.0	33.7	50.7	20.6	25.1	33.4
5	12.2	16.3	22.1	37.5	13.6	15.5	21.5

Table 7: Effect of black box complexity for Synth.

data. Thus on the *Parse* pipeline (which is tested on financial newswire text), we increased the error rate of the Lingpipe POS tagger by training it on the literary *Brown* corpus [11]. Another source of error is the use of less powerful learning techniques. As an example of this case, on the *RelEx* pipeline we degraded the final RE stage by using the CK_1 kernel instead of the CK_1+SSK kernel from [22], a weaker kernel, in the SVM classifier. In the *Synth* pipeline, we increased the noise component in the learned distribution mixture of all pipeline components to 30%.

Table 5 shows the results comparing the different inference methods in the presence of increased error and noise. While the performance of different inference methods drops compared to the low error/noise settings, we find that the Top-k and Beam-M inference methods still significantly outperform the canonical Top-1 inference method. Furthermore, the performance for both proposed inference techniques improves on increasing the value of k in Top-k or M in Beam-M as already observed in the low error/noise settings. In general, the trends from our earlier experiments continue to hold, showing the robustness of our inference methods.

4.1.6 Pipeline Length and Complexity

In our last set of experiments, we explore the effects of increasing pipeline length and complexity.

Task	Tools
POS Tagging	LingPipe, NLTK, Stanford Tagger, GENiA, OpenNLP
Parsing	Stanford Parser, Enju, NLTK

Table 8: Popular NLP modules for tagging and parsing that provide the Top-k API

First, we varied the length of the synthetic pipeline by adding more components at the end of the pipeline. The results are shown in Table 6. As expected, the performance for all methods drops as we increase the length. However, we still find that the two proposed inference methods significantly outperform the canonical method, and they maintain their trends across varying pipeline lengths.

Second, another source of complexity in pipelines is the difficulty of correct inference in the individual components. To study this effect, we changed the Dirichlet distribution parameter α that was used to generate the distributions for each of the black boxes. The results are shown in Table 7. We find that while the task gets harder as we increase α, the Top-k and Beam-M inference methods maintain their advantage compared to the Top-1 canonical inference.

5. DISCUSSION

Generality of our methods: Our DAG formalism is general enough to subsume most pipelines used in practice. For the general case of graphs with cycles (corresponding to pipelines with feedback), there still exists an underlying *undirected* graphical model, and message passing schemes (like Loopy Belief Propagation) may correspond to natural procedures for pipeline inference. In cases where feedback is limited to a maximum number of iterations, the underlying graph can be "unrolled" to a DAG, and our approach applies.

Since the Top-k API makes very mild assumptions about the components, it is directly applicable in pipelines across several domains, such as vision and robotics. We experimented with NLP pipelines in particular as they offer an attractive experimental test-bed to demonstrate our approach since the community has clearly structured linguistic tasks into well defined sub-tasks, and has agreed upon performance measures for each sub-task. Additionally there exist a variety of NLP software that provide the Top-k API. For example,

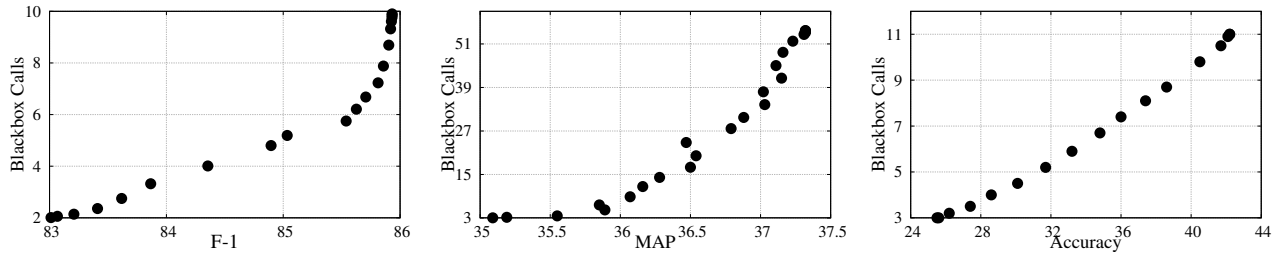


Figure 6: Tradeoff between performance and computational cost (in terms of number of black box calls) for the Parse (*left*), RelEx (*middle*) and Synth (*right*) pipelines.

Table 8 lists commonly used software for tagging and parsing which provide this functionality. Relation extraction is most commonly solved as a supervised multi-class prediction problem, which trivially yields top-k output along with conditional probabilities.

Applicable even without the Top-k API: In our exposition, we assumed that black boxes provide the Top-k API. However our methods can still be used when this assumption is violated, albeit with small modifications. For example, some software provide joint probabilities $\Pr(Y, X \mid \theta)$ instead of conditional models. These can still be incorporated into our framework via normalization, which can be done efficiently for many learning models. When components do not model a probabilistic distribution and only provide the canonical output, we can still obtain a proxy for the Top-k in our method, by using *redundant* techniques/software that solve the same problem. This is not uncommon as seen in Table 8. Techniques such as Boosting or Mixture-of-Experts provide reliable proxies for the confidence scores assigned by redundant components for each output. Another alternative, that is applicable when model parameters can be re-learned, is to use variants of the same model obtained via training from different data samples or other such parameter perturbations.

Alternatives to Adaptive Inference: The *Transform* operator in Algorithm 1 is just one way to limit computational complexity during inference. An alternative transformation of the intermediate output table T_i is to pass along tuples so as to capture a fixed mass of the conditional probability. This heuristic allows us to bound the gap between the inference objective and the Bayes Optimal objective.

Inference in a pipeline should be a goal-oriented process. When deciding whether to deepen the beam at a particular component, we would ideally want to know the sensitivity of the eventual output of the pipeline to additional intermediate outputs. We can create adaptive inference strategies that operate in phases. In the first phase, one may propagate limited amounts of information throughout the pipeline to reliably estimate the sensitivity of different components. This can then help determine how to deepen the beams. A version of this adaptive beam deepening heuristic that performs sensitivity analysis of just the successor component (rather than the end of the pipeline) showed promising empirical results.

Limitations of Top-k Inference: The Top-k and Fixed Beam approaches (as well as the canonical approach) are not impervious to badly designed pipelines. For example, in a two stage pipeline, if the second component ignores the output of the first component when making its prediction, generating additional outputs from the first component cannot improve overall performance. Sensitivity analysis and adaptive beam deepening promise to address this limitation.

Applying our ideas in monolithic software pipelines can be challenging. These software, which typically deploy fine-tuned instantiations of pipeline components, have been designed to combat the problem of cascading errors in pipelines. For instance, some relation extraction software bundle together a hand-tuned parser and POS tagger rather than accepting inputs from off-the-shelf components. This makes these systems brittle to changes in the evaluation domain and prevents potential improvements from incorporating better models developed for early stage components. Thus in our experimental set-up, we made a conscious effort to have a modular pipeline throughout, with different off-the-shelf components plugged in for different tasks.

Our approach can also fail if the Bayes Optimal prediction is not the ideal goal. For example, if an early component encounters a catastrophic failure on an input, and the entire conditional distribution is "wrong", the Bayes optimal prediction need not be the correct eventual output. In light of our work, this provides interesting insight into model development efforts. During training, it may be more prudent to spend modeling effort on limiting catastrophic failures across the input space, than tweaking parameters to make the correct output appear at the top of the scored prediction lists rather than a few places below.

Scalability of Inference: Pipeline inference is embarrassingly parallel: each instance in the test set is usually processed independently. Hence, the overall cost of inference is linear in the size of the test set. Thus approaches that bring down the linear cost of testing are complementary to our efforts at improving the quality of predictions.

Bridging DBs and Big Data Pipelines: The relational algebra formulation of our inference procedures draws connections with probabilistic databases and data provenance, and could be used as computationally-efficient inference methods for these problems as well.

6. RELATED WORK

In different domains, pipelines take different shapes. Li et al. [16] have developed a model of shallow and broad pipelines for scene understanding in computer vision, while NLP pipelines are typically deep e.g., Machine Translation, Semantic Role Labeling, Relation Extraction and Opinion Extraction [27, 10, 5]. In Natural Language Processing, UIMA [12], GATE [8] and more recently CURATOR [6] provide several components that can be plugged into a configurable pipeline. Our approach to improve inference in pipelines is orthogonal to their implementations, and can be incorporated into each of these frameworks seamlessly.

Beam search has been used commonly in combination with algorithms for efficient inference [28]. To propagate the outputs in the beam efficiently, a number of efficient structures have been devised to compactly encode certain

families of distributions [21, 17]. Efficient encodings for top-k lists can improve the scalability of our approach as well.

Adaptively sub-sampling data while *training* models with bounded loss guarantees during inference is a way to trade-off training efficiency with inference fidelity [9]. Researchers have attempted to migrate code to FPGAs and GPUs [7] and devised parallel implementations of expensive analysis tasks [18, 19]. Our approach is complementary to these efforts, and promises to trade-off efficiency and performance during inference in a graceful manner.

An alternative to propagating the top-k outputs would be to sample from the posterior distribution of probabilistic models [10]. However we recognize that most practical components do not support such an operation. Other work alleviates the cascading error problem by routing the eventual outputs back as inputs to the pipeline [16]. These iterative methods do not have clear guarantees on convergence or on the quality of inference. Finally, other groups have recognized that canonical inference ignores the scores $s(y)$ that components provide. These scores can be incorporated as features for downstream tasks [3] or used to iteratively constrain outputs of upstream tasks in subsequent passes [13]. Another approach to the cascading error problem is performing joint inference across multiple stages of the pipeline, as done for joint entity and relation extraction [30]. However, this is limited to certain kinds of probabilistic models and to small pipelines, due to the complexity of joint inference.

7. CONCLUSIONS

We propose a probabilistic graphical model that allows principled reasoning about Big Data Pipelines. The model provides a well-founded interface between pipeline components that accounts for uncertainty, while maintaining modularity that enables reuse of pipeline components. Abstracting pipeline components behind an API that is already widely supported by off-the-shelf components, the paper demonstrated how Top-k inference, Beam search, and Adaptive beam selection can provide improved prediction performance and robustness of the overall pipeline. More generally, we anticipate that this probabilistic model of Big Data Pipelines will be useful in many other ways (e.g., the global tuning of pipelines based on partial training data). This work was supported in part by NSF Awards IIS-1012593 and IIS-0911036.

8. REFERENCES

[1] A. Anand, H. S. Koppula, T. Joachims, and A. Saxena. Contextually guided semantic labeling and search for three-dimensional point clouds. *IJRR*, 32(1):19–34, 2013.

[2] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, pages 983–992, 2008.

[3] R. C. Bunescu. Learning with probabilistic features for improved pipeline models. In *EMNLP*, pages 670–679, 2008.

[4] B. Carpenter. Scaling high-order character language models to Gigabytes. In *Software*, pages 86–99, 2005.

[5] Y. Choi, C. Cardie, E. Riloff, and S. Patwardhan. Identifying sources of opinions with Conditional Random Fields and extraction patterns. In *EMNLP*, 2005.

[6] J. Clarke, V. Srikumar, M. Sammons, and D. Roth. An NLP curator (or: How I learned to stop worrying and love NLP pipelines). In *LREC*, 2012.

[7] A. Coates, P. Baumstarck, Q. Le, and A. Y. Ng. Scalable learning for object detection with GPU hardware. In *IROS*, pages 4287–4293, 2009.

[8] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, and W. Peters. *Text Processing with GATE (Version 6)*. 2011.

[9] P. Domingos and G. Hulten. A general method for scaling up machine learning algorithms and its application to clustering. In *ICML*, pages 106–113, 2001.

[10] J. R. Finkel, C. D. Manning, and A. Y. Ng. Solving the problem of cascading errors: Approximate bayesian inference for linguistic annotation pipelines. In *EMNLP*, 2006.

[11] W. N. Francis. A standard corpus of edited present-day American English. *College English*, 26(4):pp. 267–273, 1965.

[12] T. Götz and O. Suhre. Design and implementation of the UIMA common analysis system. *IBM Syst. J.*, 2004.

[13] K. Hollingshead and B. Roark. Pipeline iteration. In *ACL*, pages 952–959, 2007.

[14] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *ICCV*, pages 2146–2153, 2009.

[15] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *ACL*, pages 423–430, 2003.

[16] C. Li, A. Kowdle, A. Saxena, and T. Chen. Towards holistic scene understanding: Feedback enabled cascaded classification models. In *NIPS*, pages 1351–1359, 2010.

[17] R. Llobet, J.-R. Cerdan-Navarro, J. C. Pérez-Cortes, and J. Arlandis. OCR post-processing using weighted finite-state transducers. In *ICPR*, pages 2021–2024, 2010.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[20] M. P. Marcus, B. Santorini, M. A. Marcinkiewicz, and A. Taylor. Treebank-3. Technical report, Linguistic Data Consortium, Philadelphia, 1999.

[21] M. Mohri. Compact representations by finite-state transducers. In *ACL*, pages 204–209, 1994.

[22] T.-V. T. Nguyen, A. Moschitti, and G. Riccardi. Convolution kernels on constituent, dependency and sequential structures for relation extraction. In *EMNLP*, pages 1378–1387, 2009.

[23] B. Pang and L. Lee. Opinion mining and sentiment analysis. *Found. Trends Inf. Retr.*, 2(1-2):1–135, 2008.

[24] P. Paroubek. Evaluating Part-of-Speech Tagging and Parsing. In *Evaluation of Text and Speech Systems*, pages 99–124. 2007.

[25] J. C. Platt. Probabilistic outputs for Support Vector Machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74, 1999.

[26] K. Sagae, Y. Miyao, and J. Tsujii. HPSG parsing with shallow dependency constraints. In *ACL*, 2007.

[27] B. Taskar, D. Klein, M. Collins, D. Koller, and C. Manning. Max-margin parsing. In *EMNLP*, 2004.

[28] C. Tillmann and H. Ney. Word reordering and a dynamic programming beam search algorithm for statistical machine translation. *Computational Linguistics*, 29:97–133, 2003.

[29] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. BayesStore: Managing large, uncertain data repositories with probabilistic graphical models. *Proc. VLDB Endow.*, 1(1):340–351, Aug. 2008.

[30] L. Yao, S. Riedel, and A. McCallum. Collective cross-document relation extraction without labelled data. In *EMNLP*, pages 1013–1023, 2010.

[31] Z. Yao, J. Barrick, Z. Weinberg, S. Neph, R. Breaker, M. Tompa, and W. L. Ruzzo. A computational pipeline for high-throughput discovery of cis-regulatory noncoding RNA in prokaryotes. *PLoS Comput Biol*, 2007.