Faster (and Better) Entity Linking with Cascades

Adrian Benton, Jay DeYoung, Adam Teichert, Stephen Mayhew
Mark Dredze, Benjamin Van Durme, Max Thomas
Human Language Technology Center of Excellence
Baltimore, MD, USA
{adrian,jay.deyoung,ateichert,mdredze,vandurme}@jhu.edu,
mayhew2@illinois.edu, max.thomas@jhuapl.edu

Abstract

Entity linking requires ranking thousands of candidates for each query, a time consuming process and a challenge for large scale linking. Many systems rely on prediction cascades to efficiently rank candidates. However, the design of these cascades often requires manual decisions about pruning and feature use, limiting the effectiveness of cascades. We present Slinky, a modular, flexible, fast and accurate entity linker based on prediction cascades. We adapt the web-ranking prediction cascade learning algorithm, Cronus, in order to learn cascades that are both accurate and fast. We show that by balancing between accurate and fast linking, this algorithm can produce Slinky configurations that are significantly faster and more accurate than a baseline configuration and an alternate cascade learning method with a fixed introduction of features.

1 Introduction

A key step in mining structured knowledge from unstructured text is to uniquely identify the entities associated with extracted relations, events and facts by linking them to a structured knowledge base (KB) [2, 10, 3, 17, 9, 4]. For KBs with many thousands or millions of KB entries, considering them all is prohibitively expensive. Instead, systems rely on a two stage approach [3, 12, 5], in which a fast method (often constant time) identifies candidates from the KB for a given mention string, and a second slower stage relies on extensive features to rank candidates. Some entity linking systems have extended the two stage approach using prediction cascades [15], such as CALE [11, 12]. There remain several open challenges for how to develop prediction cascades for this task as pruning thresholds, features to be extracted, and number of stages are all parameters that are manually tuned.

We address these challenges by introducing Slinky, an entity linking system developed on a massively parallel distributed processing architecture, allowing cascades with an arbitrary number of stages, processing numerous candidates and queries in parallel. We adapt Cronus [13], a recent cascade learning algorithm designed for web ranking, to entity linking. This algorithm can *learn the structure of prediction cascades*, deciding which feature templates (groups of features that share processing in common) to include in which stage and how much each stage should prune candidates. The learning algorithm makes these decisions by optimizing a global objective function that balances overall accuracy with speed, enabling tuning to balance these two goals by a single scalar hyperparameter. The algorithm can expand or contract how many candidates are processed in each stage, much like a slinky. We evaluate both speed and accuracy and find that the learned cascade is *both fast and accurate*, improving in both over a baseline ranking SVM objective, as well as a fixed introduction of features per stage [14].

Entity Linking An entity linking instance is comprised of a mention string q, a document d and, for training, a reference candidate c from the KB. If the mention's referent is not in the KB, then the candidate c is represented as `NIL`. We treat `NIL` like an ordinary candidate in our system [3, 12].

For each query, the system produces a ranking over a subset of the candidates contained in the KB, returning the highest ranked candidate as the answer to the query.

2 Slinky

Slinky implements prediction cascades for entity linking on top of a highly parallel message passing infrastructure to enable the easy addition of new stages and replication of each stage to scale up to available compute resources.

Slinky's architecture relies on a pipeline to process each query. Learned stages are *italicized*. 1) The **Query Preprocessor** extracts features for each query (mention, document pair) based solely on the query, which saves repeat processing across candidates. 2) The **Triager** efficiently generates candidates for the query from the KB. We use the triage strategy from CALE [12]. We find the triager is highly effective with a recall of 96.3% for our training queries with a median of 191 candidates per query. 3) ***Pruners*** independently score a candidate and filter candidates assigned a negative score. The key difference between a pruner and the triager, is that the pruner can use a large, more expensive set of features while the triager only looks at the mention string and candidate name. Pruners are implemented as binary linear classifiers and we use multiple pruners in a pipeline. 4) The ***Ranker*** orders remaining candidates, including the `NIL` candidate. While the ranker is also a linear classifier, it jointly considers all remaining candidates, allowing for more expressive features.

Slinky implements this cascade using Akka[1], a Scala framework for distributed, concurrent message passing based on the actor model. Each of the above stages is an Actor, which each run on a separate thread. For slower stages (triager and pruners), we instantiate multiple copies of each Actor, which allows us to take advantage of a multi-core machine, and process candidates in parallel within each stage.

We select features commonly used by other entity linking systems. These features are grouped into templates, such that features within a template share processing in common. The templates used in our experiments were grouped into *Bleu*, *Dice*, *Name Match*, *Multi-Name Match*, (features computing name similarity between mention string and candidate name), *Popularity* (popularity of entities in the knowledge base [12]), *Context* (computing similarity between the query and candidate documents), *Nil* (features that indicate query is not in the knowledge base, such as the number of tokens or characters in the query name), and *Bias* (simple features noting candidate and query named entity type and an always-on bias feature). We find that these features with our baseline method gives competitive results with the best methods. See [12] for a full description of the features.

3 Learning Cascade Structure

We adapt Cronus [13] to learn the parameters for each pruner/ranker, the filtering threshold for each pruner, and which features each pruner should learn. The Cronus adaptation objective explicitly trades off between accuracy, speed, and regularization terms:

$$\mathcal{L}(\theta) = (1-\lambda)\mathcal{L}_A(\theta) + \lambda\mathcal{L}_S(\theta) + C\sqrt{\sum_i \theta_i^2} \tag{1}$$

$\mathcal{L}_A$ is the loss due to inaccurate predictions (i.e. 1 - accuracy). $\mathcal{L}_S$ is the loss due to prediction speed, where we desire faster predictions (smaller loss.) We also include an L_2 penalty on the parameters, θ. $\lambda \in [0, 1]$ trades off between accuracy (smaller λ) and speed (higher λ) and C is the regularization parameter – a linear interpolation of L_2-regularized accuracy and runtime. θ includes the parameters for all stages and, in this paper, all stages use linear classifiers. This is where our implementation diverges most from Cronus [13], since they assume each classifier to be a linear combination of decision trees. In this sense, our model is more constrained than Cronus. Our goal is to minimize the total loss $\mathcal{L}$, and we adopt a similar relaxation in order to minimize this function through gradient descent.

Accuracy The relaxation of accuracy (for a single query) is dependent on two quantities:

[1] http://akka.io/

$$p_{c,s} = \frac{1}{1 + \exp\{-f_s(c) \cdot \theta_s\}} \quad (2) \qquad k_{c,s} = \prod_{s' < s} p_{c,s'} \quad (3)$$

where c ranges over all candidates for generated by triage for the query, s ranges over all pruners and ranker stage, θ_s are the sets of weights for the linear classifier at stage s, and f is the feature template extractor for a given stage where if $s' < s$ then f_s extracts a superset of features of $f_{s'}$. p is interpreted as the probability that c passes through stage s without pruning using the logistic function as a relaxation of deterministic pruning with a threshold of 0, and k is the probability that a candidate remains unpruned by stage s (the probability of "keeping" this candidate till stage s). With k, we can define $P(C_r = \emptyset) = \prod_c (1 - k_{c,r})$, the probability that all candidates are independently pruned by the ranking stage – the final set of candidates to be ranked is empty. In this case, Slinky defaults to predicting the `NIL` candidate. If we relax the final ranker as making softmax predictions over all candidates reaching that stage, we can define the loss to accuracy for a single query, q, as:

$$\mathcal{L}_A^q = P(C_r = \emptyset) \cdot \mathbb{1}(y \neq \texttt{NIL}) + (1 - P(C_r = \emptyset))(1 - \frac{k_{y,r} exp\{\psi_y\}}{Z}) \quad (4)$$

where r is the ranker stage, y is the identity of the correct candidate to link to, ψ_y is the score assigned to candidate y by the ranker, and $Z = \sum_c k_{c,r} exp\{\psi_c\}$ is the normalizer. $exp\{\psi_y\}$ can be interpreted as the unnormalized probability that the final ranker ranks the correct candidate as the top prediction. Under this interpretation, $\frac{k_{y,r} exp\{\psi_y\}}{Z}$ is the probability that the correct candidate passed through the cascade unpruned *and* was then ranked as the top prediction (these events are independent.) For a set of queries, Q, $\mathcal{L}_A = \frac{\sum_{q \in Q} \mathcal{L}_A^q}{|Q|}$.

Speed We define the proportion loss in runtime compared to the worst case runtime of extracting all feature templates for all candidates as

$$\mathcal{L}_S = \delta_\theta / \delta_b \quad (5)$$

where δ_θ is the cost of feature extraction using our current set of weights, and δ_b is the cost of feature extraction if all parameters of θ were set to 1 (every stage requires all candidates to be extracted). We assume that if all weights for a feature template in a stage are zeroed, then Slinky will not extract these features. The trick is to define δ_θ such that traversing the gradient will yield sparsity in feature templates used across earlier stages, so that by the end of the cascade, only a small number of unpruned candidates will have many feature templates extracted for them; most candidates are pruned early by stages only requiring inexpensive features to be extracted. Since selecting which feature templates to extract in each stage is effectively L_0 regularization, and would have to be solved with combinatorial optimization [6], we use a variant of the hierarchical group lasso penalty on the weights within a feature template and across prefixes of stages to achieve the proper sparsity pattern with gradient descent optimization:

$$\omega_{c,s} = k_{c,s}(1 - p_{c,s}) \quad (6)$$

$$d_s = \sum_{f \in F^s} \tau_f \sqrt{\frac{1}{|n_f| s} \sum_{i \in f} \sum_{s' < s} (\theta_{s',i})^2} \quad (7)$$

$$\delta_\theta = \sum_c \left(k_{c,r} d_r + \sum_{s < r} \omega_{c,s} d_s \right) \quad (8)$$

where $\omega_{c,s}$ is the probability that candidate c reaches stage s and is then pruned (follows directly from the above definitions of k and p) and d_s is the hierarchical group lasso relaxation of feature template acquisition cost for a candidate that passes through the pipeline to stage s. In this formulation, each group is considered to be all the features in a feature template for all classifiers up till stage s. The group lasso penalty is the L1 penalty of the L2 norms for each group. In Eq. 7, τ_f represents the acquisition cost for feature template f, $|n_f|$ is the number of features in f, and $\theta_{s',i}$ ranges over weights associated with this feature template over all previous stages. Note that defining the group lasso with normalization by $|n_f|s$ is consistent with the formulation in [16]. We employ this normalization so as not to harshly penalize feature templates with many features. This formulation of the group lasso is equivalent to that presented in [13], aside from the fact that their groups also spanned decision trees within each stage.

Parameter Estimation We minimize Eq. 1 using the approximations for accuracy (Eq. 4) and speed (Eq. 5) as well as $\mathcal{L}_R$. Since multiple stages makes the loss non-convex, making parameter

initialization critical, we initialized each pruner's parameters using piece-wise training with a rank SVM objective [8], with the slack parameter set to $\frac{1}{C}$, the inverse of the regularization constant in Eq. 1. This initialization is similar to Cronus since they use a convex initialization method by training each successive stage, holding previous stage weights fixed. We train with L-BFGS and obtain gradients using automatic differentiation from Theano [1].

4 Experiments

We evaluated Slinky on entity linking shared task datasets to determine improvements obtained by using a cascade. We compare with the cascade learning method presented in [14], which uses a similar relaxation to the learning by supposing each pruner probabilistically rejects candidates based on their score passed through a sigmoid function. However, they assume a fixed introduction of feature templates per stage, where earlier stages may only make predictions on less expensive features, and more costly features are successively added to stages. We simulate this **Fixed** feature introduction method in our framework by fixing weights and the gradients of those weights to zero for those stages that do not have access to the feature templates yet. Performance is reported in terms of macro-averaged accuracy.

Our training, development, and test data came from previous Text Analysis Conference Knowledge Base Population (TAC-KBP) entity linking tasks using the TAC-KBP knowledge base containing 818,741 entries. For training, we used the 1615 query training data available for TAC 2009, including supplemental data released by [3], for development the 3904 query evaluation data from TAC 2009, and for test the 2250 query evaluation data from TAC 2010.

Actual runtimes reported are median number of milliseconds per query over 10 runs of a Slinky configuration with 4 actors allotted per stage, operating in parallel. Each experiment used a machine with 64 2.1 GHz AMD Opteron 6272 CPUs and a total of 256G RAM. We start the wall-clock timer once all queries have been read by the system and sent to the preprocessor stage. We stop the timer once all queries have been resolved and all predictions are written to disk. We also include the time to write to disk since we intend the reported runtime to represent the time necessary to resolve queries in a real setting. We tuned λ, C, and the number of stages on development data. We selected two configurations: *most accurate* and *fastest* to evaluate on test data. A *baseline* configuration consisted of only a single ranker and favored accuracy only ($\lambda = 0$). The τ values were estimated by the number of milliseconds to extract each feature template over all training candidates on average under Slinky's framework.

5 Results

Table 1 shows performance (TAC 2009 test data for development, TAC 2010 evaluation data for test) of three selected configurations based on their performance over the development set: fastest, accurate, and baseline ($n = 0$, $\lambda = 0$). n corresponds to the number of pruners in the cascade configuration. The reported runtime is the average number of milliseconds to resolve each query. This evaluation produced a very nice result: the fastest Cronus-learned system over the development set is in fact also the most accurate over test data, improving actual runtime by around 50% and giving large gains in accuracy (over 6% absolute). This is surprising since it did not achieve as high accuracy over the development set as the *Cronus Accurate* configuration (0.742 as opposed to 0.778.) We believe this is due to variation between the development and test sets, but also that the sparsity introduced by λ, the penalty on runtime, may allow this configuration to generalize better than a denser cascade.

Although the method presented in [14] is even faster, this speed comes with a high cost to accuracy. Compared to the systems submitted to the TAC KBP 2010 shared task, our *baseline* system would rank just behind the 10th best system (Figure 3 in [7].) However, a critical difference, aside from feature changes described above, is that systems in TAC KBP 2010 were able to train over all 3904 queries in the TAC KBP 2009 evaluation set. Since we used that set for development, we trained on only 1615 queries, which were not necessarily representative of the true distribution.

It is clear that learning obtains significant reductions in speed over a baseline, but Slinky's architecture in general is already a significant improvement over previous methods. While runtime has

Table 1: Test performance of selected configurations using the Cronus-learned and Fixed Slinky configurations along with a single-stage baseline that performed fastest/most accurately on the development set.

n	λ	C	**Test Accuracy**	$\mathcal{L}_S$	**Test Runtime**	**Configuration**
0	0.00	0.500	0.689	1.0	313	Baseline
1	0.15	0.005	**0.752**	0.26	161	Cronus Fastest
3	0.25	0.050	0.726	0.86	290	Cronus Accurate
3	NA	5.000	0.547	0.03	**121**	Fixed Fastest
3	NA	0.250	0.689	0.37	185	Fixed Accurate

not been a focus of most entity linking systems, [3], one of the first entity linking systems, provide some information on runtime. They report a "fast candidate selection system" that resolved queries in 1,980 milliseconds on average. Our baseline is roughly 6.3 times faster than their system, and our learned "fastest" configuration is about 12.3 times faster.

In summary, Slinky is a highly parallel architecture for entity linking. We learn the configuration of the cascade optimized for both fast and accurate entity linking. The learned cascade produces a reduction in runtime of 50% over an already fast baseline, as well as substantial improvements in accuracy.

References

[1] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Python for Scientific Computing Conference (SciPy)*, 2010.

[2] Silviu Cucerzan. Large-scale named entity disambiguation based on wikipedia data. In *EMNLP*, 2007.

[3] Mark Dredze, Paul McNamee, Delip Rao, Adam Gerber, and Tim Finin. Entity disambiguation for knowledge base population. In *COLING*, 2010.

[4] Stephen Guo, Ming-Wei Chang, and Emre Kiciman. To link or not to link? a study on end-to-end tweet entity linking. In *NAACL*, 2013.

[5] Yuhang Guo, Bing Qin, Yuqin Li, Ting Liu, and Sheng Li. Improving candidate generation for entity linking. In *Natural Language Processing and Information Systems*, pages 225–236. Springer, 2013.

[6] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *JMLR*, 3, 2003.

[7] Heng Ji, Ralph Grishman, Hoa Trang Dang, Kira Griffitt, and Joe Ellis. Overview of the TAC 2010 knowledge base population track. In *TAC*, 2010.

[8] Thorsten Joachims. Training linear svms in linear time. In *KDD*, 2006.

[9] Xiaohua Liu, Yitong Li, Haocheng Wu, Ming Zhou, Furu Wei, and Yi Lu. Entity linking for tweets. In *ACL*, 2013.

[10] Paul McNamee and Hoa Trang Dang. Overview of the TAC 2009 knowledge base population track. In *TAC*, 2009.

[11] Paul McNamee, James Mayfield, Douglas W. Oard, Tan Xu, Ke Wu, Veselin Stoyanov, and David Doerman. Cross-language entity linking in maryland during a hurricane. In *TAC*, 2011.

[12] Paul McNamee, Veselin Stoyanov, James Mayfield, Tim Finin, Tim Oates, Tan Xu, Douglas Oard, and Dawn Lawrie. HLTCOE participation at TAC 2012: Entity linking and cold start knowledge base construction. In *TAC*, 2012.

[13] Minmin Minmin Chen, Kilian Q. Weinberger, Olivier Chapelle, Dor Kedem, and Zhixiang Xu. Classifier cascade for minimizing feature evaluation cost. In Neil D. Lawrence and Mark A. Girolami, editors, *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (AISTATS-12)*, volume 22, pages 218–226, 2012.

[14] Vikas C. Raykar, Balaji Krishnapuram, and Shipeng Yu. Designing efficient cascaded classifiers: Tradeoff between accuracy and cost. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 853–860, New York, NY, USA, 2010. ACM.

[15] David Weiss and Ben Taskar. Structured prediction cascades. In *AIStats*, 2010.

[16] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[17] Zhicheng Zheng, Xiance Si, Fangtao Li, Edward Y. Chang, and Xiaoyan Zhu. Entity disambiguation with freebase. In *Web Intelligence and Intelligent Agent Technology*, 2012.