Efficient programmable learning to search

Hal Daumé III
University of Maryland
me@hal3.name

John Langford
Microsoft Research
jcl@microsoft.com

Stephane Ross
Google
stephaneross@google.com

Abstract

We improve "learning to search" approaches to structured prediction in two ways. First, we show that the search space can be defined by an arbitrary imperative program, reducing the number of lines of code required to develop new structured prediction tasks by orders of magnitude. Second, we make structured prediction orders of magnitude faster through various algorithmic improvements.

1 Introduction

In structured prediction problems, the goal is creating a good set of joint predictions. As an example, consider recognizing a handwritten word where each character might be recognized in turn to understand the word. Here, it is commonly observed that exposing information from related predictions (i.e. adjacent letters) aids individual predictions. Furthermore, optimizing a joint loss function can improve the gracefulness of error recovery. Despite this, it is empirically common to build independent predictors in settings where structured prediction naturally applies. Why? Because independent predictors are much simpler, easier and faster to train. Our primary goal is to make structured prediction algorithms as easy and fast as possible to both program and compute.

A new programming abstraction, together with several algorithmic pearls, radically reduce the complexity of programming and the running time of our solution.

1. We enable structured prediction as a *library* which has a function PREDICT(...) returning predictions. The PREDICT(...) interface is the minimal complexity approach to producing a structured prediction. Surprisingly, this single library interface is sufficient for both testing *and training*, when augmented to include label "advice" from a training set. This means that a developer need only code desired *test time behavior* and gets training "for free."

2. Although the PREDICT(...) interface is the same as the interface for an online learning algorithm, the structured prediction setting commonly differs in two critical ways. First, the loss may not be simple 0/1 loss over subproblems. For optimization of a joint loss, we add a LOSS(...) function which allows the declaration of an *arbitrary* loss for the joint set of predictions. The second difference is that predictions are commonly used as features for other predictions. This can be handled either implicitly or explicitly, but the algorithm is guaranteed to work either way.

Here PREDICT(...) and LOSS(...) enable a concise specification of structured prediction problems.

Basic sequence labeling as shown in algorithm 1 is the easiest possible structured prediction problem, so it forms a good use case. The algorithm takes as input a sequence of examples (consider features of handwritten digits in words), and predicts the meaning of each element in turn. This is a specific case of sequential decision making, in which the ith prediction may depend on previous predictions. In this example, we make use of the library's support for implicit feature-based dependence on previous predictions.

Algorithm 1 SEQUENTIAL_RUN(*examples*)

1: **for** $i = 1$ to LEN(*examples*) **do**
2:     *prediction* ← PREDICT(*examples*[i], *examples*[i] . *label*)   // make a prediction on the ith example
3:     **if** *output* . *good* **then**
4:         *output* « ' ' « *prediction*   // if we should generate output, append our prediction
5:     **end if**
6: **end for**

The use of this function for decoding is clear, but how can the PREDICT(...) interface be effective? There are two challenges to overcome in creating a viable system.

1. Given the available information, are there well-founded structured prediction algorithms? For Conditional Random Fields [Lafferty et al., 2001] and structured SVMs [Taskar et al., 2003, Tsochantaridis et al., 2004], the answer is "no", because we have not specified the conditional independence structure of the system of predicted variables. Instead, we use a system that implements search-based structured prediction methods such as Searn [Daumé III et al., 2009] or DAgger [Ross et al., 2011]. These have formal correctness guarantees which differ qualitatively from the conditional log loss guarantees of CRFs. For example, given a low regret cost-sensitive classification algorithm, Searn guarantees competition according to LOSS(...) with an oracle policy and local optimality w.r.t. one-step deviations from the learned policy. We discuss how these work below.

2. A sequential program has only one execution stack, which is used by the decoding algorithm above. This conflicts because the learning algorithm would naturally also use the stack. We refactor the learning algorithm into a state machine which runs before the RUN function is called and after the various library calls are made. In essence, RUN is invoked many times with different example sequences and different versions of PREDICT(...) so as to find a version of PREDICT(...) with a small LOSS(...).

Given this high level design, the remaining challenge is computational. How do we efficiently and effectively find a PREDICT(...) which achieves a small LOSS(...)?

2   Learning to Search

A discrete search space is defined by states $s \in S$ and a mapping $m : S \to 2^S$ defining the set of valid next states. One of the states is a unique start state a while some of the others are end states $s \in E$. A loss function $l(s)$ is defined for any end state $s \in E$ on the training set. We are interested in algorithms which learn the transition function $f : X_s \to S$ which uses the features of an input state (X_s) to choose a next state so as to minimize the loss l on a heldout test set. Two canonical algorithms to solve this problem are Searn [Daumé III et al., 2009] and DAgger [Ross et al., 2011] which we review next.

Searn uses some oracle transition function f^* which is defined on the training set, but not on the heldout test set. As searn operates it learns a sequence of transition functions $f_0, f_1, ..., f_n$ where $f_0 = f^*$ and f_n is entirely learned. At each iteration $i \in \{1, ..., n\}$, Searn uses f_{i-1} to generate a set of cost-sensitive examples. A cost-sensitive example is defined using local features, features which express previous predictions, and a set of costs defined for each possible next state. The costs are derived by rollouts: for each $s' \in m(s)$, the transition function is applied until an end state $s \in E$ is observed and a loss $l(s)$ is computed. This vector of losses, one for each $s' \in m(s)$, forms the vector of costs. Together with local features it is fed to the cost sensitive learning algorithm. The cost-sensitive learning algorithm generates a classifier $c_i : X_s \to S$, then a new policy $f_i = (1 - \alpha)f_{i-1} + \alpha c_i$ is defined using stochastic interpolation. In essence, with probability α c_i is used to define the transition while with probability $1 - \alpha$ f_{i-1} is used to define the transition matrix. Since the probability of calling f^* decreases exponentially, a fully learned policy is quickly found.

DAgger differs from Searn in two computationally helpful ways: it mixes datasets rather than policies and uses a loss function l' defined on all states, so rollouts are not required. When a loss is only defined for end states, a DAgger style algorithm can operate with rollouts.

2

Algorithm 2 TDOLR(X)

1: $s \leftarrow a$
2: **while** $s \notin E$ **do**
3:     Compute X_s from X and s
4:     $s \leftarrow \mathrm{O}(X_s)$
5: **end while**
6: **return** $\mathrm{Loss}(s)$

The rollout versions of the previous algorithms require $\mathcal{O}(t^2 knp)$ where t is the average end state depth (i.e. sequence length), $k = |m(s)|$ is the number of next states (i.e. branching factor), n is the number of distinct searches (i.e. sequences), and p is the number of data passes. Three computational tricks: online learning [Collins, 2002, Bottou, 2011], memoization, and rollout collapse allow the computational complexity to be reduced to $\mathcal{O}(tkn)$, similar to independent prediction. For example, we can train a part-of-speech tagger 100 times faster than CRF++ [Kudo, 2005] which is unsurprising since Viterbi decoding in a CRF is $\mathcal{O}(tk^2)$. Surpringly we can do it with 6 lines of "user code," versus almost 1000.

We show that learning to search can be implemented with the library interface in section 3. This provides a radical reduction in the coding complexity of solving new structured prediction problems as discussed. We also radically reduce the computational complexity as discussed next in section 4, then conduct experiments in section 5.

3   System Equivalences

Here we show the equivalence of a class of programs and search spaces. The practical implication of this equivalence is that instead of specifying a search space, we can specify a program, which can radically reduce the programming complexity of structured prediction.

Search spaces are defined in the introduction, so we must first define the set of programs that we consider. Terminal Discrete Oracle Loss Reporting (TDOLR) programs:

1. Always terminate.

2. Takes as input any relevant feature information X.

3. Make zero or more calls to an oracle $O : X' \rightarrow Y$ which provides a discrete outcome.

4. Report a loss L on termination.

To show equivalence, we prove a theorem. This theorem holds for the case where the number of choices is fixed in a search space (and, hence, $m(s)$ is implicitly defined).

Theorem 1. *For every TDOLR program there exist an equivalent search space and for every search space there exists an equivalent TDOLR program.*

The practical implication of this theorem is that instead of speicfying search spaces, we can specify a TDOLR program (such as algorithm 1), and apply any learning to search algorithm such as Searn, DAgger, or variants thereof.

Proof. A search space is defined by (a, E, S, l). We show there is a TDOLR program which can simulate the search space in algorithm 3. This algorithm does a straightforward execution of the search space, followed by reporting of the loss on termination. This completes the second claim.

For the first claim, we need to define, (a, E, S, l) given a TDOLR program such that the search space can simulate the TDOLR program. At any point in the execution of TDOLR, we define an equivalent state $s = (O(X_1), ..., O(X_n))$ where n is the number of calls to the oracle. We define a as the sequence of zero length, and we define E as the set of states after which TDOLR terminates. For each $s \in E$ we define $l(s)$ as the loss reported on termination. This search space manifestly outputs the same loss as the TDOLR program.

$\square$

3

Algorithm 3 LEARN(X)

1: $T \leftarrow 0$
2: $ex \leftarrow []$
3: Define PREDICT(x, y) := { $ex[++T] \leftarrow x$; return $f_i(x, y)$ }
4: Define SNAPSHOT(...) := RECORDSNAPSHOT(...)
5: RUN(X)
6: for $t_0 = 1$ to T do
7:      $losses \leftarrow []$
8:      for $a_0 = 1$ to M($ex[t_0]$) do
9:          Define PREDICT(...) := $\begin{cases} \text{return } a_0 & \text{if } t = t_0 \\ \text{return } f_i(...) & \text{if } t \neq t_0 \end{cases}$
10:         Define SNAPSHOT(...) := $\begin{cases} \text{JUMPTO}(t_0) & \text{if } t < t_0 \\ \text{TRYFASTFORWARD}(...) & \text{if } t > t_0 \\ \text{no op} & \text{if } t = t_0 \end{cases}$
11:         Define LOSS(val) := { $losses[a_0]$ += val }
12:         RUN(X)
13:     end for
14:     Online update with cost-sensitive example ($ex[t_0]$, $losses$)
15: end for

---

4 Imperative Structured Prediction

The full learning algorithm (for a single structured input, X) is depicted in Algorithm 4. In lines 1–5, an "initialization" pass of RUN is executed. RUN can generally be any TDOLR program as discussed in appendix 3, with a specific example being algorithm 1. In this pass, predictions are made according to the current policy, f_i, and every time SNAPSHOT is called, the results are memoized for future use (on the current example). Furthermore, the examples (feature vectors) encountered during prediction are stored in ex, indexed by their position in the sequence (T).

The algorithm then initiates one-step deviations from this initial trajectory. For every time step, (line 6), we generate a *single* cost-sensitive classification example; its features are $ex[t_0]$, and there are M($ex[t_0]$) possible labels (=actions). For each action (line 8), we compute the *cost* of that action. To do so, we execute RUN again (line 12) with a "tweaked" PREDICT that at a particular time-step (t_0) simply returns the perturbed action a_0. Finally, the LOSS function simply accumulates the loss for the query action. Finally, a cost-sensitive classification is generated (line 14) and fed into an online learning algorithm.

When the learning-to-search algorithm is Searn, this implies a straightforward update of the next policy. The situation with online DAgger is more subtle—in essence a dependence on the reference policy must be preserved for many updates to achieve good performance. We do this using policy interpolation (as in Searn) between the reference policy and learned policy.

Without any speed enhancements, each execution of RUN takes $\mathcal{O}(t)$ time, and we execute it $tk + 1$ times, yielding an overall complexity of $\mathcal{O}(kt^2)$ per structured example. For comparison, structured SVMs or CRFs with first order Markov dependencies run in $\mathcal{O}(k^2t)$ time.

To improve this running time, we make two optimizations using the idea of SNAPSHOTs. Together, they reduce the overall runtime to $\mathcal{O}(kt)$, when paths collapse frequently (this is tested empirically in Section 6.1). These optimizations take advantage of the fact that most predictions only depend on a small subset of previous predictions, not all of them. In particular, if the ith prediction only depends on the $i − 1$st prediction, then there are at *most tk* unique predictions ever made.[1] This is what enables dynamic programming for sequences (the Viterbi algorithm). We capitalize on this observation in a more generic way: memoization. A program is allowed to SNAPSHOT its state before making a prediction. Because the SNAPSHOT encapsulates its *entire* state, we can efficiently store that state together with relevant statistics in a hash table.

---

[1]We use *tied randomness* [Ng and Jordan, 2000] to ensure that for any time step, the same policy is called.

4.1 Optimization 1: JumpTo

In Algorithm 4, suppose that when we execute RUN on line 12, we have $t_0 = T - 1$. Naïvely, one must execute $T - 1$ PREDICTs in order to reach the desired state at which we vary a_0. This is inefficient. Instead, assuming that RUN recorded a snapshot at time $T - 1$ during the initialization (line 5), we simply *restore* that stored state the first time SNAPSHOT is called: the $t < t_0$ condition in line 10. Even when we cannot restore the state *precisely* to $T - 1$ (for instance, perhaps the most recent snapshot was at $T - 2$), we can additionally memoize the previous results of PREDICT and regurgitate those predictions. This alone saves $\mathcal{O}(td)$ time, where d is the time to make a prediction.

Correctness. In line 14, the learned policy changes. For policy mixing algorithms (like Searn), this is fine and correctness is guaranteed. However, for data mixing algorithms (like DAgger), this potentially changes f_i, implying the memoized predictions may no longer be up-to-date so the recorded snapshots may no longer be accurate. Thus, for DAgger-like algorithms, this optimization is okay *if* the policy does not change much. We evaluate this empirically in Section 6.1. The next section has the same correctness properties.

4.2 Optimization 2: TryFastForward

The second optimization is fast forwarding to the end of the sequence using TRYFASTFORWARD. For example, suppose $t_0 = 2$. After perturbing the action at time point 2 we have $t > 2$. Every time SNAPSHOT is called, the snapshotted data might *exactly match* a previous snapshot. Suppose at $t = 3$ it does *not*, because the perturbation at $t = 2$ cascaded and changed the prediction at $t = 3$. But perhaps at $t = 4$ there is a perfect match (paths have collapsed). We remember that a match has occurred and then at $t = 5$, we can "fast forward" to $t = T$ because all subsequent predictions are identical.

This intuitive explanation is correct, except for accumulating LOSS(...). If LOSS(...) is only declared at the end of RUN, then we must execute $T - t_0$ time steps making (possibly memoized) predictions. However, for many problems, it is possible to declare loss *early* as with Hamming loss (= number of incorrect predictions). There is no need to wait until the end of the sequence to declare a per-sequence loss: one can declare it after every prediction, and have the total loss accumulate (hence the "+=" on line 11). We generalize this notion slightly to that of a history-independent loss:

Definition 1 (History-independent loss). *A loss function is* history-independent at state s_0 *if, for any final state e reachable from s_0, and for any sequence $s_0 s_1 s_2 \ldots s_i = e$: it holds that* $\text{LOSS}(e) = A(s_0) + B(s_1 s_2 \ldots s_i)$, *where B does not depend on any state before s_1.*

For example, Hamming loss is history-independent: $A(s_0)$ corresponds to Hamming loss up to and including s_0 and $B(s_1 \ldots s_i)$ is the Hamming loss after s_0.[2]

When the loss function being optimized is history-independent, we allow LOSS(...) to be declared *early*, allowing an additional SNAPSHOT optimization. In the previous example, at time $t = 4$ the snapshot matched. Suppose that at this time, a total loss of 2 had been accumulated (this corresponds to $A(\ldots)$). Then at time $t = 5$ we can immediately jump to the end of the sequence $t \leftarrow T$, provided that we've memoized the total loss incurred from $t = 5$ to $t = T$ on this trajectory (this corresponds to $B(\ldots)$), which may be 0. The total cost for this a_0 perturbation is then $2 + 0 = 2$.

4.3 Overall Complexity

Suppose that the cost of calling the policy is d.[3] Then the complexity of the unoptimized learning function is $\mathcal{O}(t^2 kd)$. By adding the memoization optimizations only, and assuming paths collapse after a constant number of steps, this drops to $\mathcal{O}(t^2 k + tkd)$. (The first term is from retrieving memoized predictions, the second from executing the policy a constant number of times for each perturbed sequence.) Adding the SNAPSHOT restoration in addition to the memoization, the complexity drops

---

[2] Any loss function that decomposes over structure, as required by structured SVMs, is guaranteed to also be history-independent; the reverse is not true. Furthermore, when structured SVMs are run with a non-decomposable loss function, their runtime becomes exponential in t. When our approach is used with a loss function that's not history-independent, our runtime increases by a factor of t.

[3] Because the policy is a multiclass classifier, d might hide a factor of k or $\log k$.

POS	NNP NNP , CD NNS JJ , MD VB DT NN IN DT JJ NN
	Pierre Vinken , 61 years old , will join the board as a nonexecutive director ...
NER	LOC ORG PER
	Germany 's rep to the European Union 's committee Werner Zwingmann said ...

Figure 1: Example inputs (below, black) and desired outputs (above, blue) for part of speech tagging task and named entity recognition task.

further to $\mathcal{O}(tkd)$. In comparison, a first order CRF or structured SVM for sequence labeling has a complexity of $\mathcal{O}(tk^2 f)$, where f is the number of features and $d \approx fk$ or $\approx f \log k$ depending on the underlying classifier used.

5 Experimental Results

We conduct two experiments based on variants of the sequence labeling problem (Algorithm 1). The first is a pure sequence labeling problem: Part of Speech tagging based on data from the Wall Street Journal portion of the Penn Treebank. The second is a sequence *chunking* problem: named entity recognition using data from the CoNLL 2003 dataset. See Figure 1 for example inputs and outputs for these two tasks.

We use the following freely available systems/algorithms as points of comparison:

CRF++ The popular CRF++ toolkit [Kudo, 2005] for conditional random fields [Lafferty et al., 2001], which implements both L-BFGS optimization for CRFs [Nash and Nocedal, 1991, Malouf, 2002] as well as "structured MIRA" [Crammer and Singer, 2003, McDonald et al., 2004].

CRF SGD A stochastic gradient descent conditional random field package [Bottou, 2011].

Structured Perceptron An implementation of the structured perceptron [Collins, 2002] due to [Chang et al., 2013].

Structured SVM The cutting-plane implementation [Joachims et al., 2009] of the structured SVMs [Tsochantaridis et al., 2004] for "HMM" problems.

Structured SVM (DEMI-DCD) A multicore algorithm for optimizing structured SVMs called DEcoupled Model-update and Inference with Dual Coordinate Descent.

VW Search Our approach is implemented in the Vowpal Wabbit [Langford et al., 2007] toolkit on top of a cost-sensitive classifier [Beygelzimer et al., 2005] that reduces to regression trained with an online rule incorporating AdaGrad [Duchi et al., 2011], per-feature normalized updates [Ross et al., 2013], and importance invariant updates [Karampatziakis and Langford, 2011].

VW Classification An *unstructured* baseline that predicts each label independently, using one-against-all multiclass classification [Beygelzimer et al., 2005].

These approaches vary both objective function (CRF, MIRA, structured SVM, learning to search) and optimization approach (L-BFGS, cutting plane, stochastic gradient descent, AdaGrad). All implementations are in C/C++, except for the structured perceptron and DEMI-DCD (Java).

5.1 Methodology

Comparing different systems is challenging because one wishes to hold constant as many variables as possible. In particular, we want to control for both **features** and **hyperparameters**. In general, if a methodological decision cannot be made "fairly," we made it in favor of competing approaches.

To control for **features**, we use the built-in *feature template* approach of CRF++ (duplicated in CRF SGD) to generate features. The other approaches (Structured SVM, VW Search and VW Classification) all use the features generated (offline) by CRF++. For each task, we tested six feature templates and picked the one with best development performance using CRF++. The templates included neighboring words and, in the case of NER, neighboring POS tags. *However,* because VW Search is also

	Training					Heldout		Test	
	Sents	Toks	Labels	Features	Unique Fts	Sents	Toks	Sents	Toks
POS	38k	912k	45	13,685k	629k	5.5k	132k	5.5k	130k
NER	15k	205k	7	8,592k	347k	3.5k	52k	3.6k	47k

Table 1: Basic statistics about the data sets used for part of speech (POS) tagging and named entity recognition (NER).

able to generate features from its own templates, we also provide results for **VW Search (own fts)** in which it uses its own, internal, feature template generation, which were tuned to maximize it's heldout performance on the most time-consuming run (4 passes) and include neighboring words (and POS tags, for NER) and word prefixes/suffixes.[4] In all cases we use *first order Markov dependencies,* which lessens the speed advantage of search based structured prediction.

To control for **hyperparameters**, we first separated each system's hyperparameters into two sets: (1) those that affect termination condition and (2) those that otherwise affect model performance. When available, we tune hyperparameters for (a) learning rate and (b) regularization strength[5]. Additionally, we vary the termination conditions to sweep across different amounts of time spent training. For each termination condition, we can compute results using either the **default hyperparameters** or the **tuned hyperparameters** that achieved best performance on heldout data. We report both conditions to give a sense of how sensitive each approach is to the setting of hyperparameters (the amount of hyperparameter tuning directly affects effective training time).

One final confounding issue is that of **parallelization**. Of the baseline approaches, only CRF++ supports parallelization via multiple threads at training time. In our reported results, CRF++'s time is the total CPU time (i.e., effectively using only one thread). Experimentally, we found that wall clock time could be decreased by a factor of 1.8 by using 2 threads, a factor of 3 using 4 threads, and a (plateaued) factor of 4 using 8 threads. This should be kept in mind when interpreting results. DEMI-DCD (for structured SVMs) also *must* use multiple threads. To be as fair as possible, we used 2 threads. Likewise, it can be sped up more using more threads [Chang et al., 2013]. VW (Search and Classification) can also easily be parallelized using AllReduce [Agarwal et al., 2011]. We do not conduct experiments with this option here because none of our training times warranted parallelization (a few minutes to train, max).

5.2 Task Specifics

Part of speech tagging for English is based on the Penn Treebank tagset that includes 45 discrete labels for different parts of speech. The overall accuracy reported is Hamming accuracy (number of tokens tagged correctly). This is a *pure* sequence labeling task. We use $912k$ tokens (words) of training data and approximately $50k$ tokens of heldout data and test data. The CRF++ templates generate $630k$ unique features for the training data; additional statistics are in Table 1.

Named entity recognition for English is based on the CoNLL 2003 dataset that includes four entity types: Person, Organization, Location and Miscellaneous. We report accuracy as macro-averaged F-measure over the correct identification and labeling of these entity spans (the standard evaluation metric). In order to cast this *chunking* task as a sequence labeling task, we use the standard Begin-In-Out (BIO) encoding, though some results suggest other encodings may be preferable [Ratinov and Roth, 2009]. The example sentence from Figure 1 in this encoding is:

	LOC					ORG				PER		
Germany	's	rep	to	the	European	Union	's	committee	Werner	Zwingmann	said	...
B-LOC	O	O	O	O	B-ORG	I-ORG	O	O	B-PER	I-PER	O	

In our system, the only change made to the sequence labeling algorithm (Algorithm 1) is that I-x may only follow B-x or I-x. We still optimize Hamming loss because macro-averaged F measure does not decompose over individual sentences.
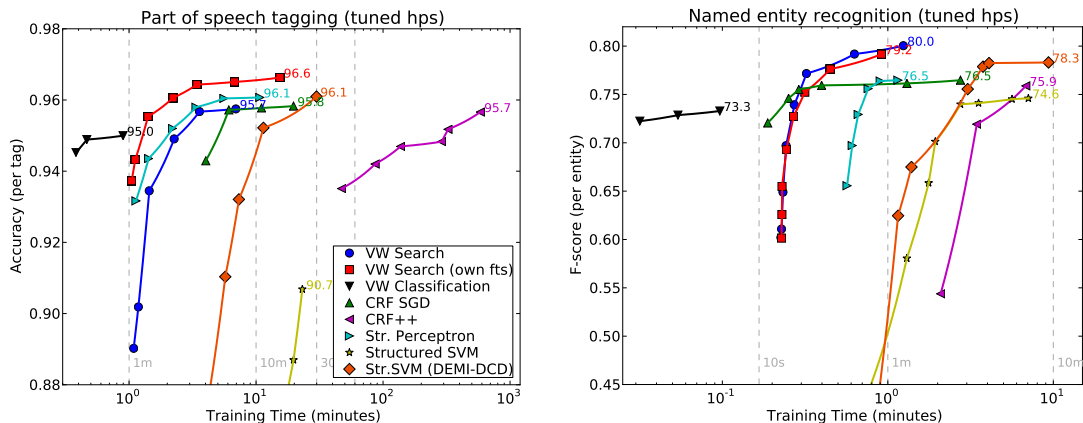
Figure 2: Training time versus evaluation accuracy for part of speech tagging (left) and named entity recognition (right). X-axis is in log scale. Different points correspond to different termination criteria for training. Both figures use hyperparameters that were tuned (for accuracy) on the heldout data. (Note: lines are curved due to log scale x-axis.)

5.3 Efficiency versus Accuracy

In Figure 2, we show trade-offs between training time (x-axis, log scaled) and prediction accuracy (y-axis) for the six systems described previously. The left figure is for part of speech tagging (912k training tokens) and the right figure is for named entity recognition (205k training tokens).

For POS tagging, the independent classifier is by far the fastest (trains in less than one minute) but its performance peaks at 95% accuracy. Three other approaches are in roughly the same time/accuracy tradeoff: VW Search, VW Search (own fts) and Structured Perceptron. All three can achieve very good prediction accuracies in just a few minutes of training. CRF SGD takes about twice as long. DEMI-DCD eventually achieves the same accuracy, but it takes a half hour. CRF++ is not competitive (taking over five hours to even do as well as VW Classification). Structured SVM (cutting plane implementation) looks promising, but runs out of memory before achieving competitive performance (likely due to too many constraints).

For NER the story is a bit different. The independent classifiers are quite fast (a few seconds to train) but are far from being competitive[6]. Here, the two variants of VW Search totally dominate.[7]. In this case, Structured Perceptron, which did quite well on POS tagging, is no longer competitive and is essentially dominated by CRF SGD (in the sense that it is faster for the same accuracy). The only system coming close to VW Search's performance is DEMI-DCD, although it's performance flattens out after a few minutes.[8]

To achieve the results in Figure 2 required fairly extensive hyperparameter tuning (on the order of 50 to 100 different runs for each system). To see the effects of hyperparameter tuning, we also ran each system with the built-in hyperparameter options.[9] The trends in the runs with default hyperparame-

---

[4]The exact templates used are provided in the supplementary materials.

[5]Precise details of hyperparameters tuned and their ranges is in the supplementary materials.

[6]When evaluating F measure for a system that may produce incoherent tag sequences, like "O I-LOC" we replace any malpositioned I-x with B-x; of all heuristics we tried, this worked best.

[7]We verified the prediction performance with great care here—it is the first time we have observed learning to search approaches significantly exceeding the prediction performance of other structured prediction techniques when the feature information available is precisely the same.

[8]We also tried giving CRF SGD the features computed by VW Search (own fts) on both POS and NER. On POS, its accuracy improved to 96.5—on par with VW Search (own fts)—with essentially the same speed. On NER it's performance decreased. For both tasks, clearly features matter. But which features matter *is* a function of the approach being taken.

[9]The only exceptions is Structured SVMs, which do not have a default C value (we used $C = 128$ because that setting won most often across all experiments).
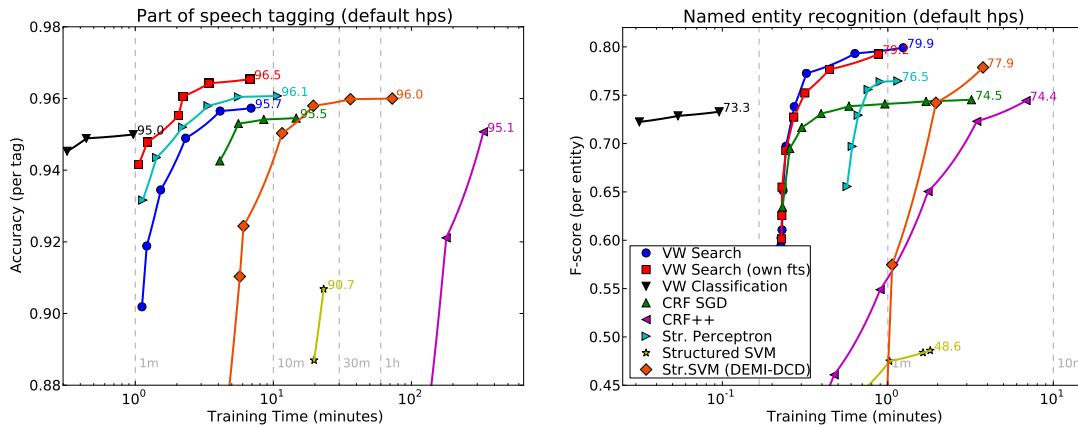
8

Figure 3: Training time versus evaluation accuracy for POS tagging (left) and NER (right). X-axis is in log scale. Different points correspond to different termination criteria for training. Both figures use *default* hyperparameters.
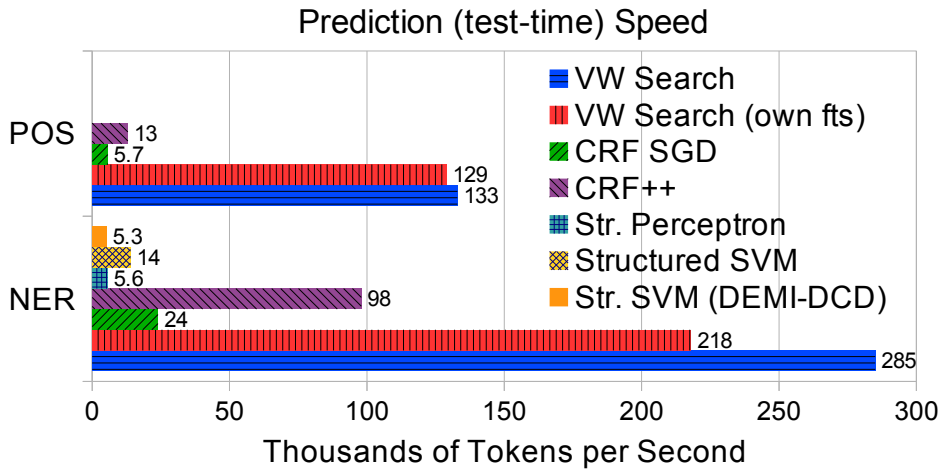

Figure 4: Comparison of test-time efficiency of the different approaches in thousands of tokens per second. For NER, this ranges from 5k tokens/sec (DEMI-DCD) to over a quarter million tokens/sec. These numbers include feature computation time only for the two CRF approaches.

ters (Figure 3) show similar behavior to those with tuned, though some of the competing approaches suffer significantly in prediction performance. Structured Perceptron has no hyperparameters.

6   Test-time Prediction Performance

In addition to training time, one might care about test time behavior. On NER, prediction times varied from $5.3k$ tokens/second (DEMI-DCD and Structured Perceptron to around $20k$ (CRF SGD and Structured SVM) to $100k$ (CRF++) to $220k$ (VW (own fts)) and $285k$ (VW). Although CRF SGD and Structured Perceptron fared well in terms of training time, their test-time behavior is suboptimal. When looking at POS tagging, the effect of the $\mathcal{O}(k^2)$ dependence on the size of the label set further increased the (relative) advantage of VW Search over the alternatives.

Figure 4 shows the speed at which the different systems can make predictions on raw text. Structured SVMs and friends (DEMI-DCD and Structured Perceptron) are by far the slowest (NER: $14k$ tokens per second), followed closely by CRF SGD (NER: $24k$ t/s). This is disappointing because CRF SGD performed very well in terms of training efficiency. CRF++ achieves respectable test-time efficiency

9

(NER: almost $100k$ t/s).[10] VW Search using CRF++'s features is the fastest (NER: $285k$ t/s) but, like Structured SVM, this is a bit misleading because it requires feature computation from CRF++ to be run as a preprocessor. A fairer comparison is VW Search (own fts), which runs on (nearly) raw text and achieves a speed of $218k$ t/s for NER.

One thing that is particularly obvious comparing the prediction speed for VW Search against the other three approaches is the effect of the size of the label space. When the number of labels increases from 9 (NER) to 45 (POS), the speed of VW Search is about halved. For the others, it is cut down by as much as a factor of 8. This is a direct complexity argument. Prediction time for VW Search is $\mathcal{O}(tkf)$ versus $\mathcal{O}(tk^2f)$ for all other approaches.

Overall, we found our approach to achieve comparable or higher accuracy in as little or less time, both with tuned hyperparameters and default hyperparameters. The closest competitor for POS tagging was the Structured Perceptron (but that did poorly on NER); the closest competitor for NER was CRF SGD (but that was several times slower on POS tagging).

6.1 Empirical evaluation of path-collapse

In Section 4, we discussed two approaches for computational improvements. First, memoization: avoid re-predicting on the same input multiple times (which is fully general). Second, snapshot restoration: to jump to an arbitrary desired position in the search space (which requires a history-independent loss function). Both are effectively only when paths collapse frequently.

The effect of different optimizations (none, memoization alone, or memoization combined with snapshot restoration) is shown below. Columns are internal loss, F measure on heldout data, number of predictions made, time to train and one standard deviation of training time over 5 runs.

Optimization	heldout loss	heldout F-score	# training predictions	training time
All	0.426	85.6%	1,722,173	1.24m ± 0.16
Memoization	0.432	85.6%	1,724,957	1.76m ± 0.09
None	0.431	85.4%	18,620,344	2.23m ± 0.04

The above results show the effect of these optimizations on the *best* NER system we trained, which achieved a test F score of 79.9% in $1.24m$. In this table, we can see that memoization alone reduces the number of predictions made by over 90%, with only a very small increase of 0.001 in loss on the heldout data (the loss reported here is the internal average per-sequence Hamming loss, rather than F measure). Recall that this optimization is only provably correct in Searn mode, not DAgger mode as run here. The memoization reduces overall runtime by about 21% because not all time is being spent making predictions. When the second optimization (snapshot jumps) is enabled, the total number of predictions drops imperceptibly, but the runtime improves by another 30%, yielding a total improvement of about 45% over the baseline. Note that the loss here actually goes *down* slightly, perhaps due to a slightly less noisy cost function.

7 Relation to Probabilistic Programming

Probabilistic programming [Gordon et al., 2014] has been an active area of research for the past decade or more. While our approach bears a family resemblace to the idea of probabilistic programming, it differs in two key ways. First, we have not designed a new programming language. Instead we have a three-function library. This is advantageous because it makes adoption easier. Moreover, our library is in C/C++, which makes integration into existing code bases (relatively) easy. Second, the abstract we focus on is that of *prediction*. In contrast, the typical abstraction for probabilistic programming is *distributions*. We believe that prediction is a more natural abstraction for a lay programmer to think about than probability distributions.

The closest work to ours is Factorie [McCallum et al., 2009]. Factorie is a domain specific language embedded in Scala, and is essentially an embedded language for writing factor graphs. It compiles

---

[10]This suggests gains are possible by combining CRF++'s "decoder" and I/O system with CRF SGD's optimizer.

them into Scala, which in turn produces JVM code that can be run reasonably efficiently. Nonetheless, as far as we are aware, Factorie-based implementations of simple tasks like sequence labeling are still less efficient than systems like CRF SGD. Factorie, more than other probabilistic programming languages we are aware of, acts more like a library than a language; though it's abstraction is still distributions (more precisely: factor graphs). Another approach which takes the approach of formulating a library is Infer.NET, from Minka et al. [2010]. Infer.NET is a library for constructing probabilistic graphical models in a .NET programming framework. It supports approximate inference methods for things like variational inference and message passing.

In the same spirit as Factorie of having a concise programmic method of specifying factors in a Markov network are: Markov Logic Networks (MNLs) due to Richardson and Domingos [2006] and Probabilistic Soft Logic (PSL) due to Kimmig et al. [2012]. Although neither of these was derived specifically from the perspective of formulating factors in a conditional random field (or hinge-loss Markov network), that is the net result. Neither of these is an embedded language: one must write declarative code and provide data in an appropriate format, which makes it somewhat difficult to use in complex systems. BLOG [Milch et al., 2007] falls in the same category, though with a very different focus. Similarly, Dyna [Eisner et al., 2005] is a related declarative language for specifying probabilistic dynamic programs which can be compiled into C++ (and then used as library code inside another C++ program). All of these example have picked particular aspects of the probabilistic modeling framework to focus on.

Beyond these examples, there are several approaches that essentially "reinvent" an existing programming language to support probabilistic reasoning at the first order level. IBAL [Pfeffer, 2001] derives from O'Caml; Church [Goodman et al., 2008] derives from LISP. IBAL uses a (highly optimized) form of variable elimination for inference that takes strong advantage of the structure of the program; Church uses MCMC techniques, coupled with a different type of structural reasoning to improve efficiency.

It is worth noting that most of these approaches have a different goal than we have. Our goal is to build a framework that allows a developer to solve a quite general, but still specific type of problem: learning to solve sequential decision-making problems ("learning to search"). The goal of (most) probabilistic programming languages is to provide a flexible framework for specifying graphical models and performing inference in those models. While these two goals are similar, they are different enough that the minimalistic library approach we have provided is likely to be insufficient for general graphical model inference.

8   Discussion

We have shown a new abstraction for a structured prediction library which achieves better accuracies than many widely used specialized software toolkits that are specialized to sequence labeling.[11] Moreover, it achieves this with runtimes up to two orders of magnitude faster than competing approaches. Moreover, we achieve this with minimal programming effort on the part of the developer who must implement RUN. Our sequence labeling implementation is 6 lines of code; compared to: CRF SGD at 1068 LOC, CRF++ at 777 LOC and Structured SVM at 876 LOC.[12] Somewhat surprisingly, this is all possible through a very simple (three function) library interface which does *not* require the development of an entirely new programming language. This is highly advantageous as it allows very easy adoption. Moreover, since our library functions in a reduction stack, as base classifiers and reductions for cost-sensitive classification improve, so does structured prediction performance.

---

[11]It is worth noting that better accuracies *have* been reported in the literature for these two tasks, even without using external data sources (e.g., unlabeled data or gazetteers). For instance, guided learning [Shen et al., 2007] achieves an accuracy of 97.33% on POS tagging and Ratinov and Roth [2009] achieves an F-score of 85.57% on NER.

[12]All LOC computations *exclude* lines of code for "base learning" algorithms and "optimizer" implementations: this is a measure of how much code has gone in to implementing the specific structured prediction task (in this case: sequence labeling). The code complexity for alternatives come from implementing dynamic programming; or for SVMs from the requirement for finding maximally violated constraints.

Acknowledgments

References

Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *arXiv preprint arXiv:1110.4198*, 2011.

Alina Beygelzimer, Varsha Dani, Tom Hayes, John Langford, and Bianca Zadrozny. Error limiting reductions between classification tasks. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 49–56, 2005.

Leon Bottou. crfsgd project, 2011. http://leon.bottou.org/projects/sgd.

Kai-Wei Chang, Vivek Srikumar, and Dan Roth. Multi-core structural SVM training. In *Proceedings of the European Conference on Machine Learning (ECML)*, 2013.

Michael Collins. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2002.

Koby Crammer and Yoram Singer. Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research (JMLR)*, 2003.

Hal Daumé III, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine Learning Journal*, 2009.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research (JMLR)*, 12:2121–2159, 2011.

Jason Eisner, Eric Goldlust, and Noah A. Smith. Compiling comp ling: Practical weighted dynamic programming and the dyna language. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2005.

Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Josh Tenenbaum. Church: a language for generative models. In *Proceedings of the Converence on Uncertainty in Artificial Intelligence (UAI)*, 2008.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE, FOSE track)*, 2014.

Thorsten Joachims, Thomas Finley, and Chun-Nam Yu. Cutting-plane training of structural SVMs. *Machine Learning Journal*, 2009.

Nikos Karampatziakis and John Langford. Online importance weight aware updates. In *UAI*, pages 392–399, 2011.

Angelika Kimmig, Stephen Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *Proceedings of the NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.

Taku Kudo. CRF++ project, 2005. http://crfpp.googlecode.com.

John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 282–289, 2001.

John Langford, Alex Strehl, and Lihong Li. Vowpal wabbit, 2007. http://hunch.net/~vw.

Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation. In *Proceedings of CoNLL*, 2002.

Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

Ryan McDonald, Koby Crammer, and Fernando Pereira. Large margin online learning algorithms for scalable structured classification. In *NIPS Workshop on Learning with Structured Outputs*, 2004.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. BLOG: probabilistic models with unknown objects. *Statistical relational learning*, 2007.

Tom Minka, John Winn, John Guiver, and David Knowles. Infer .net 2.4, 2010. microsoft research cambridge, 2010.

S.G. Nash and J. Nocedal. A numerical study of the limited memory BFGS method and the truncated Newton method for large scale optimization. *SIAM Journal of Optimization*, 1:358–372, 1991.

Andrew Ng and Michael Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. In *Proceedings of the Converence on Uncertainty in Artificial Intelligence (UAI)*, pages 406–415, 2000.

Avi Pfeffer. Ibal: A probabilistic rational programming language. In *IJCAI*, 2001.

Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Conference on Natural Language Learning (CoNLL)*, 2009.

Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62(1-2), 2006.

Stephane Ross, Geoff J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Workshop on Artificial Intelligence and Statistics (AI-Stats)*, 2011.

Stéphane Ross, Paul Mineiro, and John Langford. Normalized online learning. In *UAI*, 2013.

Libin Shen, Giorgio Satta, and Aravind Joshi. Guided learning for bidirectional sequence classification. In *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*, 2007.

Ben Taskar, Carlos Guestrin, and Daphne Koller. Max-margin Markov networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.

Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasmine Altun. Support vector machine learning for interdependent and structured output spaces. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2004.

Algorithm 4 SEQUENTIAL_SNAPSHOT_RUN(*examples*)

1: **for** $i = 1$ **to** LEN(*examples*) **do**
2:    SNAPSHOT(*i*, &*i*)   // snapshot or restore the example counter. First argument: snapshot id, second argument: state to snapshot.
3:    *prediction* ← PREDICT(*examples*[*i*], *examples*[*i*].*label*)   // make a prediction on the *i*th example
4:    **if** *output*.*good* **then**
5:      *output* « ' ' « *prediction*   // if we should generate output, append our prediction
6:    **end if**
7: **end for**

Algorithm 5 SEQUENTIAL_DETECTION_RUN(*examples*, *false_negative_loss*)

1: Let *max_value* = 1
2: **for** $i = 1$ **to** LEN(*examples*) **do**
3:    *max_value* ← MAX(*max_value*, *examples*[*i*].*label*)
4: **end for**
5: Let *max_prediction* = 1
6: **for** $i = 1$ **to** LEN(*examples*) **do**
7:    SNAPSHOT(*i*, &*i*)   // snapshot example counter
8:    SNAPSHOT(*i*, &*max_prediction*)   // snapshot max accumulator
9:    *max_prediction* ← MAX(*max_prediction*, PREDICT(*examples*[*i*], *examples*[*i*].*label*)) // maintain max
10: **end for**
11: **if** *max_label* > *max_prediction* **then**
12:    LOSS(*false_negative_loss*)   // The loss is asymmetric
13: **else**
14:    **if** *max_label* < *max_prediction* **then**
15:      LOSS(1)
16:    **else**
17:      LOSS(0)
18:    **end if**
19: **end if**
20: **if** *output*.*good* **then**
21:    *output* « *max_prediction*   // if we should generate output, append our prediction
22: **end if**

A   Example TDOLR programs

In this section, we show a few TDOLR programs which illustrate the ease and flexibility of programming. The first is algorithm A which is a variant of algorithm 1 that enables the snapshotting optimization.

Algorithm A is for a sequential *detection* task where the goal is to detect whether or not a sequence contains some rare element. This illustrates outputs of lengths other than the number of examples, explicit loss functions, and a slightly less trivial use of snapshot.

B   Hardware Used

All timing results were obtained on the same machine with the following configuration. Nothing else was run on this machine concurrently:

```
2 * Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz
6144 KB cache
8 GB RAM, 4 GB Swap
Red Hat Enterprise Linux Workstation release 6.5 (Santiago)
Linux 2.6.32-431.17.1.el6.x86_64 #1 SMP
  from Fri Apr 11 17:27:00 EDT 2014 x86_64 x86_64 x86_64 GNU/Linux
```

C   Software Used

The precise software versions used for comparison are:

- Vowpal Wabbit version 7.6.1, commit 57deef6d2503ee90836ce5a73f03b971e04e1680

- CRF++ version 0.58

- crfsgd version 2.0

- svm_hmm_learn version 3.10, 14.08.08
  includes SVM-struct V3.10 for learning complex outputs, 14.08.08
  includes SVM-light V6.20 quadratic optimizer, 14.08.08

D   Hyperparameters Tuned

The hyperparameters tuned and the values we considered for each system are:

CRF++

```
termination parameters:
  number of passes (--max_iter)    { 2, 4, 8, 16, 32, 64, 128 }
  termination criteria (--eta)     0.000000000001 (to prevent termination)

tuned hyperparameters (default is *):
  learning algorithm (--algorithm) { CRF*, MIRA }
  cost parameter (--cost)          { 0.0625, 0.125, 0.25, 0.5, 1*, 2, 4, 8, 16 }
```

CRF SGD

```
termination parameters:
  number of passes (-r)            { 1, 2, 4, 8, 16, 32, 64, 128 }

tuned hyperparameters (default is *):
  regularization parameter (-c)    { 0.0625, 0.125, 0.25, 0.5, 1*, 2, 4, 8, 16 }
  learning rate (-s)               { auto*, 0.1, 0.2, 0.5, 1, 2, 5 }
```

Structured SVM

```
termination parameters:
  epsilon tolerance (-e)           { 4, 2, 1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001 }

tuned hyperparameters (default is *):
  regularization parameter (-c)    { 0.0625, 0.125, 0.25, 0.5, 1*, 2, 4, 8, 16 }
```

Structured Perceptron

```
termination parameters:
  number of passes (MAX_NUM_ITER)  { 1, 2, 4, 8, 16, 32, 64, 128 }

tuned hyperparameters (default is *):
  NONE
```

Structured SVM (DEMI-DCD)

```
termination parameters:
  number of passes (MAX_NUM_ITER)  { 1, 2, 4, 8, 16, 32, 64, 128 }

tuned hyperparameters (default is *):
  regularization (C_FOR_STRUCTURE) { 0.01, 0.05, 0.1*, 0.5, 1.0 }
```

VW Search

```
termination parameters:
  number of passes (--passes)      { 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 4 }
  (note: a number of passes < 1 means that we perform one full pass, but
   _subsample_ the training positions for each sequence at the given rate)

tuned hyperparameters (default is *):
  base classifier                  { csoaa*, wap }
  interpolation rate               10^{-10, -9, -8, -7, -6 }
```

VW Classifier

```
termination parameters:
  number of passes (--passes)      { 1, 2, 4 }

tuned hyperparameters (default is *):
  learning rate (-l)               { 0.25, 0.5*, 1.0 }
```

E   Templates Used

For part of speech tagging (CRF++):

```
U00:%x[-2,0]
U01:%x[-1,0]
U02:%x[0,0]
U03:%x[1,0]
U04:%x[2,0]
```

For named entity recognition (CRF++):

```
U00:%x[-2,0]
U01:%x[-1,0]
U02:%x[0,0]
U03:%x[1,0]
U04:%x[2,0]

U10:%x[-2,1]
U11:%x[-1,1]
U12:%x[0,1]
U13:%x[1,1]
U14:%x[2,1]

U15:%x[-2,1]/%x[-1,1]
U16:%x[-1,1]/%x[0,1]
U17:%x[0,1]/%x[1,1]
U18:%x[1,1]/%x[2,1]
```

(where words are in position 0 and POS is in 1)

For VW Search (own fts) on POS Tagging:

```
--search_neighbor_features -1:w,1:w
--affix -2w,+2w
```

(where words are in namespace "w")

For VW Search (own fts) on NER:

```
--search_neighbor_features -2:w,-1:w,1:w,2:w,-1:p,1:p
--affix -1w
```

(where words are in namespace "w" and POS is in "p")