4
Entropy-Guided Structure Learning Framework

We show in this chapter that the proposed entropy-guided feature generation method is naturally integrated to the general structure learning framework, thus extending it. We again make use of dependency parsing as an illustrative application to show that the system presented in the previous chapters is an instantiation of the extended framework.

Structure learning is to learn a function that maps an input $\boldsymbol{x}$ to the correct output structure $\boldsymbol{y}$. The output is an arbitrary structure, i.e., it comprises many variables with complex interdependencies. In the SL framework, the prediction problem is recast as an optimization problem of the form

$$F(\boldsymbol{x}; \boldsymbol{w}) = \arg \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{x})} s(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}), \tag{4-1}$$

where $\boldsymbol{w} = (w_1, \ldots, w_M)$ is the parameter vector of some learned model, $\mathcal{Y}(\boldsymbol{x})$ is the set of feasible predictions for the given input, and $s(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w})$ is a $\boldsymbol{w}$-parameterized scoring function that measures how well an output $\boldsymbol{y}$ fits the input $\boldsymbol{x}$. The prediction output space is arbitrary, but the scoring function must have the following strict form

$$s(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) = \langle \boldsymbol{w}, \boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y}) \rangle, \tag{4-2}$$

where $\langle \cdot, \cdot \rangle$ is the scalar product operator and $\boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y}) = (\phi_1(\boldsymbol{x}, \boldsymbol{y}), \ldots, \phi_M(\boldsymbol{x}, \boldsymbol{y}))$ is some joint feature vector representation of the input-output pair. That is, the scoring function is linear on the feature representation.

4.1
Feature Factorization

Each value $\phi_m(\boldsymbol{x}, \boldsymbol{y})$ in the feature vector is called the *global* feature m, which is the value of a specific feature m on the whole structure. For dependency parsing, we define the global feature m as $\phi_m(\boldsymbol{x}, \boldsymbol{y}) = \sum_{(i,j) \in \boldsymbol{y}} \phi_m(\boldsymbol{x}, i, j)$, which is the sum of the *local* feature values over all *edges* in the tree $\boldsymbol{y}$. In that way, we can rewrite the dependency tree scoring function

in the general form of (4-2)

$$\begin{aligned} s(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) &= \sum_{(i,j)\in\boldsymbol{y}} \langle \boldsymbol{w}, \boldsymbol{\Phi}(\boldsymbol{x}, i, j)\rangle \\ &= \sum_{(i,j)\in\boldsymbol{y}} \sum_{m=1}^{M} w_m \cdot \phi_m(\boldsymbol{x}, i, j) \\ &= \sum_{m=1}^{M} w_m \sum_{(i,j)\in\boldsymbol{y}} \phi_m(\boldsymbol{x}, i, j) \\ &= \sum_{m=1}^{M} w_m \cdot \phi_m(\boldsymbol{x}, \boldsymbol{y}) \\ &= \langle \boldsymbol{w}, \boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y})\rangle. \end{aligned}$$

Thus, global DP features are factored along the edges of the dependency tree. Consequently, the score of a dependency tree in the prediction function is also factored along its edges. Feature factorization is a key point in SL modeling and must give rise to efficient prediction algorithms. For DP, for instance, we end up with a maximum branching problem that is efficiently solved by the Chu-Liu-Edmonds algorithm.

4.2 Entropy-Guided Feature Generation

The derived feature vector $\boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y})$ is automatically generated by means of the proposed entropy-guided feature generation method. EFG induces feature templates by conjoining the available basic features and then instantiates these templates to generate the derived feature vectors. Basic features are factored in the same way as derived features. This factorization determines the prediction scoring function and, consequently, directly affects the prediction problem, which is the core of the SL framework. Therefore, we use the same factorization to derive the EFG basic dataset. For each factor in a structured example, we generate an example in the basic dataset, which comprises a vector of basic features and a decision variable. These variables correspond to local decision variables in the prediction problem. Hence, EFG conjoins basic features that help to discriminante the prediction problem variables.

For instance, dependency parsing features are factored along candidate edges. Thus, for each edge in a training sentence, we generate an example in the basic dataset. And, a binary decision variable is associated with each candidate edge and determines whether an edge is present in the corresponding decision

tree. Hence, EFG generates feature templates that are highly discriminative with respect to dependency edge prediction, which corresponds to the local decision in dependency parsing.

Each generated template is used to instantiate a feature in $\boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y})$. Then, this derived feature vector is used to train the structured model. Therefore, the structured model is linear on the derived feature representation, which corresponds to a non-linear combination of the basic features. EFG is completely aligned to the SL framework and it is naturally integrated in it.

4.3 Training Algorithm

There are some training algorithms that learn the parameter vector $\boldsymbol{w}$ from a given training dataset $\mathcal{D} = \{(\boldsymbol{x}, \boldsymbol{y})\}$ of correct input-output pairs. For instance, Collins (2002b) proposed the structured perceptron algorithm, a generalization of the well known binary perceptron algorithm for sequence labeling problems. The structured perceptron can be easily applied to any structured problem. Collins (2002b) also proved that the structured perceptron converges to a zero-error solution, if one exists. Crammer and Singer (2003) proposed the margin infused relaxed algorithm (MIRA), an online algorithm to train structured models for multiclass problems. MIRA can also be extended for virtually any structured problem and, for instance, is used in MSTParser. Crammer and Singer (2003) also proved some mistake bounds for an algorithm class called *ultraconservative*, which includes MIRA and structure perceptrons. SVM^{struct} Tsochantaridis et al. (2004) formulates the structure learning problem through a regularized max-margin framework, inspired on the binary support vector machine formulation. They also proposed a cutting plane method to efficiently solve this problem. However, this method still requires more computational power and memory than online algorithms like structure perceptron and MIRA. Additionally, the online algorithms are much simpler to implement than SVM^{struct}.

In this work, we use the structure perceptron algorithm (Collins, 2002b). Given a training sample $\mathcal{D} = \{(\boldsymbol{x}, \boldsymbol{y})\}$ of correct input-output pairs, the algorithm generates a sequence of models until convergence. At each iteration, a training instance is drawn from $\mathcal{D}$ and two major steps are performed: prediction using the current model and model update based on the difference between the correct and the predicted outputs. We use the large-margin structure perceptron Fernandes and Brefeld (2011).

During training, instead of the ordinary prediction problem in (4-1), we

use the following loss-augmented version

$$F_\ell(\boldsymbol{x};\boldsymbol{w}) = \arg\max_{\boldsymbol{y}'\in\mathcal{Y}(\boldsymbol{x})} \left[\langle \boldsymbol{w}, \boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y}')\rangle + C\cdot\ell(\boldsymbol{y},\boldsymbol{y}')\right],$$

where $\ell(\cdot,\cdot) \geq 0$ is an appropriate loss function that measures the difference between a candidate output and the correct one. For dependency trees, as presented in Section 2.2, we use a loss function that just counts the number of incorrect edges in the predicted tree. This loss function factorizes along the dependency tree edges, just like the global features. Thus, the nature of the underlying optimization problem is not modified when using the loss-augmented prediction. This is a desirable, yet *not necessary*, characteristic of loss functions in the SL framework. The model update usually is also factored along the output structure and efficient algorithms can be used. For dependency trees, for instance, this update can be performed in linear time, since a tree has no more than N edges (for a sentence with N tokens plus the artificial token).

We further extend the SL framework by introducing the EFG method into it. EFG is naturally integrated in the SL framework as a preprocessing step. In Figure 4.1, we present the pseudo-code of the entropy-guided large-margin structure perceptron. EFG is used to generate the derived feature vectors

$\boldsymbol{\Phi}(\mathcal{D}) \leftarrow \texttt{EFG}(\mathcal{D})$
$\boldsymbol{w}^0 \leftarrow \mathbf{0}$
$t \leftarrow 0$
while no convergence
  for each $(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}$
    $\hat{\boldsymbol{y}} \leftarrow \arg\max_{\boldsymbol{y}'\in\mathcal{Y}(\boldsymbol{x})} \left[\langle \boldsymbol{w}^t, \boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y}')\rangle + C\cdot\ell(\boldsymbol{y},\boldsymbol{y}')\right]$
    $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y}) - \boldsymbol{\Phi}(\boldsymbol{x},\hat{\boldsymbol{y}})$
    $t \leftarrow t+1$
return $\frac{1}{t}\sum_{k=1}^{t} \boldsymbol{w}^k$

Figure 4.1: ESL training algorithm – the entropy-guided large-margin structure perceptron.

$\boldsymbol{\Phi}(\mathcal{D}) = \{\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y})\}_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}}$. The derived features are then used to train the structured model. Note that, when a correct prediction is made, that is $\hat{\boldsymbol{y}} = \boldsymbol{y}$, the model does not change, that is $\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^k$. When the prediction is wrong, the update rule favors the correct output $\boldsymbol{y}$ over the predicted one $\hat{\boldsymbol{y}}$. Regarding binary features, for instance, the update rule increases the weights of features that are present in $\boldsymbol{y}$ but missing in $\hat{\boldsymbol{y}}$ and decreases the weights of features that are present in $\hat{\boldsymbol{y}}$ but not in $\boldsymbol{y}$. The weights of features that are present in both $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are not changed.

A simple extension of Novikoff's theorem (Novikoff, 1962) shows that the structure perceptron is guaranteed to converge to a zero loss solution, if one exists, in a finite number of steps (Altun et al., 2003; Collins, 2002a). The convergence theorem for SPerc is stated in Theorem 1. Crammer and Singer (2003) further prove some mistake bounds for the structure perceptron algorithm.

Theorem 1 (Structure Perceptron Convergence) *For any training dataset $\mathcal{D}$ that is separable by margin δ, the structure perceptron algorithm makes no more than $\frac{R^2}{\delta^2}$ prediction errors, where $R = \max_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D};\boldsymbol{y}'\in\mathcal{Y}(\boldsymbol{x})} ||\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y}) - \boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y}')||$ is the radius of a hypersphere that, centered at $\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y})$, encloses the joint feature vectors for all alternative outputs $\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{x})$, for all training examples $(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}$.*

4.4 Kernelization

We argue that EFG has two main advantages over kernel functions. EFG training algorithm is much faster than kernelized algorithms, and EFG makes generalization performance control easier than with kernel methods. In this section, we present the kernelized structure perceptron in order to better understand the differences between this method and EFG.

Analogously to the binary perceptron algorithm, its structure generalization can be easily kernelized. Given the sequence $(\boldsymbol{x}^1,\boldsymbol{y}^1,\hat{\boldsymbol{y}}^1),\ldots,(\boldsymbol{x}^T,\boldsymbol{y}^T,\hat{\boldsymbol{y}}^T)$ of inputs, correct outputs and predicted outputs considered by the training algorithm up to iteration T, the parameter vector at this point can be defined as

$$\boldsymbol{w}^T = \sum_{t=1}^{T-1} \left[\boldsymbol{\Phi}(\boldsymbol{x}^t,\boldsymbol{y}^t) - \boldsymbol{\Phi}(\boldsymbol{x}^t,\hat{\boldsymbol{y}}^t)\right].$$

The algorithm can keep track of how many times each alternative output $\hat{\boldsymbol{y}}$ has been predicted instead of the correct output $\boldsymbol{y}$ for each example pair $(\boldsymbol{x},\boldsymbol{y})$ by means of counters $\alpha_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{y}}}$. Thus, the parameter vector can be rewritten as

$$\boldsymbol{w} = \sum_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{y}}} \alpha_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{y}}} \cdot \left[\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y}) - \boldsymbol{\Phi}(\boldsymbol{x},\hat{\boldsymbol{y}})\right], \tag{4-3}$$

which is called the *dual model representation*. The output space $\mathcal{Y}(\boldsymbol{x})$ of most SL problems is exponential on the input size or even infinity. Thus, the dual model representation may comprise an intractable number of parameters. However, these parameters are initially *zero* for all $\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{y}}$ and only need to be instantiated once the respective triple is actually seen.

Using (4-3), the objective function of the prediction problem can also be rewritten as

$$\langle \boldsymbol{w}, \boldsymbol{\Phi}(\boldsymbol{x}', \boldsymbol{y}') \rangle = \sum_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{y}}} \alpha_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{y}}} \cdot \left[\langle \boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y}), \boldsymbol{\Phi}(\boldsymbol{x}', \boldsymbol{y}') \rangle - \langle \boldsymbol{\Phi}(\boldsymbol{x}, \hat{\boldsymbol{y}}), \boldsymbol{\Phi}(\boldsymbol{x}', \boldsymbol{y}') \rangle \right],$$

which depends only on inner products of feature vectors of the form $\langle \boldsymbol{\Phi}^a, \boldsymbol{\Phi}^b \rangle$, where $\boldsymbol{\Phi}^a$ and $\boldsymbol{\Phi}^b$ are shortcuts to, respectively, $\boldsymbol{\Phi}(\boldsymbol{x}^a, \boldsymbol{y}^a)$ and $\boldsymbol{\Phi}(\boldsymbol{x}^b, \boldsymbol{y}^b)$ for any two input-output pairs $(\boldsymbol{x}^a, \boldsymbol{y}^a)$ and $(\boldsymbol{x}^b, \boldsymbol{y}^b)$. Following the *kernel trick* (Vapnik, 1998), the inner products of feature vectors can then be replaced by an appropriate kernel function

$$K(\boldsymbol{\Phi}^a, \boldsymbol{\Phi}^b) = \langle \boldsymbol{\Psi}(\boldsymbol{\Phi}^a), \boldsymbol{\Psi}(\boldsymbol{\Phi}^b) \rangle,$$

where $\boldsymbol{\Psi}(\cdot)$ expands elements from the original feature vector space $\boldsymbol{\Phi}(\cdot, \cdot)$ to a much higher dimensional space. The kernel trick relies on the kernel function $K(\cdot, \cdot)$ to efficiently compute inner products in the high dimensional space of $\boldsymbol{\Psi}$ without explicitly expanding the original feature space.

The most successful kernel function family for NLP problems is the *polynomial kernel*. Considering binary features, a polynomial kernel of degree d conjoins all possible combinations with up to d original features. The polynomial kernel of degree d can be efficiently computed by

$$K_d(\boldsymbol{\Phi}^a, \boldsymbol{\Phi}^b) = \left(\langle \boldsymbol{\Phi}^a, \boldsymbol{\Phi}^b \rangle + 1 \right)^d,$$

which involves only an inner product in the original feature space, a sum of a constant, and an exponentiation. For instance, if $d = 2$ and the original space has exactly 3 *binary* features, then the *explicit* polynomial kernel expansion of $\boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y}) = (\phi_1, \phi_2, \phi_3)$ corresponds to $\boldsymbol{\Psi}(\boldsymbol{\Phi}(\boldsymbol{x}, \boldsymbol{y})) = (1,\ \phi_1,\ \phi_2,\ \phi_3,\ \phi_1\phi_2,\ \phi_1\phi_3,\ \phi_2\phi_3)$, if we omit redundant permutations.

The polynomial kernel of degree d is equivalent to generating all possible templates with length up to d. The problem with these kernels is that the only way to control which combinations are used is through the parameter d. For some SL task, for instance, $d = 2$ can be not enough to capture all relevant contextual patterns, but $d = 3$ can bring so many patterns that it is harmful to the generalization performance. This is a known issue with kernel methods and is related to overfitting. Another issue with kernel functions is training time. Performing predictions with the dual model is much slower than with the primal, because the former is represented by a list of dual variables that usually keeps growing during training. Since the prediction problem is constantly solved during training, the training algorithm becomes very slow.

Just like features, kernel functions can also be decomposed along the output structures. Thus, the dual model representation can be even more sparse by using α counters for each factor that appears in $(\boldsymbol{x}, \boldsymbol{y})$ but not in $(\boldsymbol{x}, \hat{\boldsymbol{y}})$, and vice versa. For DP, for instance, we can store a counter for each possible edge within a training sentence.

4.5 Empirical results

We compare ESL to polynomial kernels on two text chunking tasks. We use ESL to train a text chunking system on the Portuguese dataset provided by Fernandes et al. (2010b). We also train a kernelized SPerc system on the same data using a second-degree polynomial kernel. Previous work (Kudo and Matsumoto, 2001; Wu et al., 2006) report that this is the optimum degree for text chunking. Again, in this experiment, we use the same basic features, training algorithm, and datasets for both systems. In the first row of Table 4.1, we report the performances of these two systems. ESL outperforms the kernel

Task	**Kernel Method**		**ESL**	
	Learning	**F_1**	**F_1**	**Error Reduction**
Portuguese Chunking	SPerc	86.67	87.72	7.9%
English Chunking	SVM	93.48	94.12	9.8%

Table 4.1: Comparison of ESL to second-degree polynomial kernel.

method, reducing its error by 7.9%. That is an impressive achievement, since polynomial kernels present state-of-the-art results on many tasks, and training time for kernel methods is more than one order of magnitude longer than for ESL.

We also compare ESL to a kernel system on the CoNLL-2000 text chunking dataset. The second row of Table 4.1 presents the performances of ESL and a second-degree polynomial kernel system. The kernel method result is reported by Kudo and Matsumoto (2001). They use an SVM algorithm to train their system. However, they employ a training strategy that considers sequential interdependencies among output variables and also use Viterbi decoding during test. ESL outperforms this system, reducing its error by 9.8%.