_textkernel

VU University Amsterdam

Constrained Conditional Models for improving the accuracy of a sequence tagger

Author:
Tim Stokman

Supervisors:
Remko Bonnema &
Guszti Eiben

May 20, 2011

Contents

Summary

We attempt to evaluate the application of a constrained conditional model in an environment with time limitations. With this model you can impose prior knowledge in the form of rules or constraints on the output of an NLP model. We constructed a solver to find solutions for such models with added constraints. This solver was constructed with a combination heuristic depth first search procedures and an evolutionary algorithm both with anytime behavior. We evaluated this solver on a real world data set, which consists of the professional experience sections of resumes at Textkernel. We were able to realize small improvements for this dataset.

1 Acknowledgment

I would like to thank Jakub Zavrel for the opportunity to execute this project and his suggestions on the project and the presentation. I would like to my mentor at Textkernel Remko Bonnema for his advice, insight and guidance. I would like to thank Mihai Rotaru for his help and his excellent suggestions and advice. I would like to thank all the other members of Textkernel for their help and support at Textkernel. And finally I would like to thank my mentor Guszti Eiben at the VU University for his excellent advice on my thesis at the end of my internship.

2 Introduction

Textkernel is a company that specializes in data mining text documents. It provides products and customized solutions to businesses that want to automatize their data entry efforts.

One of the products Textkernel sells is resume parsing. This product is sold to job agencies, HR departments, unemployment bureaus and other organizations that deal with a lot of resumes. It provides a way to automatize the data entry effort associated with entering a lot of resumes in a database. Because resumes are mostly provided in an unstructured way (as a doc or pdf file), with no structured semantic information about what various parts of the text means, it is not trivial to automatically find the information you need.

Textkernel uses a number of machine learning algorithms to learn how most resumes are put together. They provide their algorithms with examples of resume with extra semantic information provided by human "annotators". These semantic tags contain information like:

- Sections (personal section, experience section, education section)
- Items (experience item, education item)
- Name, Telephone number (inside the personal section)
- Job title, Organization Name (inside the experience section)

With these annotated examples (for different languages) and a number of different algorithms the system can add semantic information to new resumes it has not seen before. Because it is impossible to anticipate all the possible types of resumes that can be encountered, errors in the way these resumes are tagged can happen. These errors can be caused by a number of things:

- "Weird" resumes with a structure that has not or rarely been encountered before

- Limitations in the machine learning algorithms that learn the annotated resumes. Some algorithms only look back a limited amount of words or have other limitations that limit the way they can learn the structure of the resumes
- Incorrect annotations (which are rare)

Sometimes, this system makes an error in the output that is obvious to a human but might be beyond the capabilities of the algorithm used to generate this output. For example, the algorithm may mark two different and separate pieces of text within an experience item as a "job title". This is incorrect in most cases but the algorithm might be unable to learn this.

Since Textkernel sells this product to critical and demanding customers, it is important that there are as few errors in the output of their product as possible.

2.1 The Approach

For some of these errors, some sort of rule or *constraint* could be formulated that would forbid certain situations that you know are incorrect. One of the more common tools used to apply constraints to the output of NLP[1] algorithms is called a constrained conditional model[1]. A constrained conditional model augments probabilistic or conditional algorithms with declarative constraints. With a constrained conditional model you can add rules and a goal function to which the output of the base algorithm has to confirm.

There needs to be a way to "score" different solutions and choices, so that a solver (which tries to find a good enough solution within the perimeter defined by the constraints) can compare different solutions and generate them in an order defined by these scores. These scores are usually the probability the NLP model assigns to the specific solutions. The goal of the solver is to find a solution that has a large score (in most cases, probability of being the right solution) while still satisfying all the constraints posed (for an example of this, see Figure 1).

To test the practicality of using a constrained conditional model to correct output of the systems Textkernel uses, we will do the following things:

- Define a constrained conditional model together for a dataset used within Textkernel that will forbid certain common errors from happening in the output
- Implement a solver that attempts to find a good enough solution within a reasonable time frame

[1]Natural Language Processing

Token	Job Title	Job Date	Job Description	Ignore
From	0.0	0.2	0.2	0.6
June	0.0	0.9	0.05	0.05
2006	0.0	0.8	0.1	0.1
until	0.3	0.3	0.0	0.4
August	0.0	0.9	0.0	0.1
2008	0.0	0.8	0.2	0.1
I	0.0	0.0	0.6	0.4
Worked	0.0	0.0	0.4	0.6
As	0.0	0.0	0.7	0.3
Secretary	0.8	0.0	0.2	0.0

Table 1: Example output with a probability distribution on the possible choices that can be made. The HMM gives one solution that might need to be corrected. We attempt to find a solution that satisfies the constraint that there should be only one consecutive job date and job description section. The discarded tag assignments are also highlighted

2.2 The Question

Constrained conditional models are fundamentally hard to solve. The amount of possible solutions to check for a solver is very large, which means that it can take a long time to find an optimal solution for a problem instance.

The questions we are examining in this paper are the following:

1. Can a constrained conditional model be used to enhance the accuracy of the existing system within Textkernel?
2. Could the constrained conditional model be made fast enough to incorporate into the existing system. What kind of search algorithms and heuristics could be used to find a good enough solution within a reasonable time span?
3. What kind of constraints are practical enough to be used in the system? Practical meaning that the constraints are easy enough to solve within a certain time limit while showing real improvement

3 Existing Pipeline

Textkernel has a large pipeline[2] of several components that takes an input resume in any format (pdf, html, doc) and outputs a tagged version of the resume. It also contains several methods to learn and test the various machine learning algorithms that it uses. This pipeline contains components like:

[2]A Chain of components that process data such that the output of one component is the input of the next component

- A Preprocessor that converts the several possible input formats to one common format
- Language guessing for determining which language model should be applied
- Tokenisation to split the text into tokens which can be words, commas or other punctuation marks
- Various phrase rules to handle specific cases
- Segmentation algorithms to split the resumes in different sections or items
- Sequence Tagging to find the appropriate semantic and part-of-speech tags for the text

The constrained conditional model is applied on the output of the sequence tagger. Because we are trying to improve the performance of the sequence tagging component, we will expand on how it theoretically works. We will also give some theory on how the segmentation with the conditional random field is done because this component might influence the types of errors the constrained conditional model encounters.

We integrated the Constrained Conditional Model as an extra step in this pipeline. It is build as a server component that talks to the sequence tagger with a predefined protocol. This gives us access to a number of tools to test the implementation of the constrained conditional model in a proper way.

3.1 Sequence Tagging

After the document has been segmented, each segment will be tagged by a sequence tagger. This algorithm tags each token with its part of speech and with its semantic significance within the segment. For example, the token "Philips" in the sentence "In 1994 I worked for Philips" would be tagged as a *singular noun*[3] of the sentence while semantically it would be tagged as the *organization name*. The part of speech tagging is done to give the sequence tagger more context in the sentence, we are mainly interested in the correctness of the semantic tags. The fact that the token is a singular noun helps to determine if it is an organization name.

The sequence tagging is done with a second order hidden markov model. A hidden markov model[10][4] is a markov chain where the states (in this case the semantic and part of speech tags) are not directly observed. The actual observations (words and tokens in the text) depend on these hidden states and the states themselves depend on one (first order) or more past states like normal markov models. Hidden markov models is a generative model[4] and it is a type of a bayesian network[5].

These dependencies (see Figure 1) can be used to infer the actual states from a set of observations, the HMM tries to find a sequence of states that maximizes the joint likelihood of these observations and observations occurring. We will give a short explanation how a simple first order hidden markov model can be used. Most practical HMM have a higher order (they look back more steps), use scaling (normalization/log-transforming of the probabilities to prevent floating point underflow), use smoothing algorithms and have extensions for things like unknown words (using suffix analysis or other techniques).

Given:

[3] a content word that can be used to refer to a person, place, thing, quality or action

[4] Generative models learn the joint distribution of label and observation sequences while discriminative models only learn the conditional distribution of the labels given the observations

[5] A Bayesian network is a graphical way of representing a joint distribution of labels and observations. Each variable (observation or label) is a vertex in the graph, each conditional dependency is represented as an edge. A graph representing a bayesian network must be acyclic

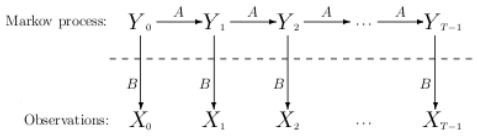

Figure 1: Conditional dependencies of the first order HMM, image source[10]

- $Q = q_i, i = 1 \dots N$ the set of possible states (the tags that can be assigned in this case)
- $O = \{o_i\}, i = 1 \dots M$ the set of possible observations (tokens in this case)
- $Y_t, t = 1 \dots T$ random variable giving the state at time t
- $X_t, t = 1 \dots T$ random variable giving the observation at time t
- $A = \{a_{ij}\}, a_{ij} = P(Y_{t+1} = q_i | Y_t = q_j)$ the row-stochastic state transition probability matrix of the markov model
- $B = \{b_{ij}\}, b_{ij} = P(X_t = o_i | Y_t = q_j)$ the row-stochastic probability matrix of an observation occurring conditioned on the current state
- $\pi = \{\pi_i\}$ the initial probability distribution of the states

You can define a complete HMM as $\lambda = \{A, B, \pi\}$. With this HMM there are different types of problems that need to be solved:

- The evaluation problem: Finding $P(X|\lambda)$ the probability of a certain observation sequence occurring within a HMM
- The decoding problem: Finding the set of states that maximizes the joint likelihood of the states and observations occurring
- The learning problem: Finding a complete HMM λ that maximizes $\max_\lambda P(X|\lambda)$ from a set of observations with prelabeled states that can be used to classify new observations

3.1.1 The Evaluation Problem

The first problem you need to solve when evaluating an HMM is calculating $P(X|\lambda)$, this probability is required to solve the decoding and learning problem. You can expand this equation as:

$$P(X|\lambda) = \sum_Y P(Y, X|\lambda) = \sum_Y P(X|Y, \lambda) P(Y|\lambda) = \sum_Y \pi_{y_0} b_{y_0, o_0} a_{y_0, y_1} b_{x_1, o_1} \dots a_{x_{T-2}, x_{T-1}} b_{x_{T-1}, o_{T-1}} \tag{1}$$

Computing this directly is infeasible for most practical cases. We can use a forward recurrence relation to calculate this probability more efficiently with a dynamic programming algorithm. This is called the forward algorithm. Given:

$$\alpha_t(i) = P(X_0 = x_0, X_1 = x_1, \dots, X_{t-1} = x_{t-1}, Y_t = q_i | \lambda) \tag{2}$$

Which is the probability of the observation sequence up to time t with current state being q_i. This has the following recurrence relation:

$$\alpha_{t+1}(i) = b_{t+1,i} \sum_{j \in Q} \alpha_t(j) a_{ji} \tag{3}$$

$$\alpha_1(i) = \pi_i b_{1i} \tag{4}$$

After a "forwards pass" calculating the α probabilities for all i, we can calculate the probability of a given sequence of observations occurring as:

$$P(X|\lambda) = \sum_{i \in Q} \alpha_{T-1}(i) \tag{5}$$

Being able to calculate this probability is important when decoding or learning an HMM.

3.1.2 The Decoding Problem

Given a HMM λ and a series of observations X, you would want to find the sequence of states that maximizes the joint likelihood of the state and observation sequence occurring:

$$y^* = \max_{y_i \in Q} P(Y_0 = y_0) P(X_0 = x_0 | Y_0 = y_0) \Pi_{i \in Y} P(Y_i = y_i | Y_{i-1} = y_{i-1}) P(X_i = x_i | Y_i = y_i) \tag{6}$$

$$= \max_{y_i \in Q} \pi_{y_0} b_{x_0, y_0} \Pi_{i \in Y} a_{y_i, y_{i-1}} b_{x_i, y_i} \tag{7}$$

Instead of the naive approach of calculating this directly, we can solve this more efficient using dynamic programming with the backward-forward algorithm. Lets define a backwards recurrence variable:

$$\beta_t(i) = P(X | y_t = q_i, \lambda) \tag{8}$$

Which is the maximum probability which an observation sequence and state sequence from time t can have. It can be calculated with the recurrence relation:

$$\beta_t(i) = \sum_{j \in Q} a_{ij} b_{j, x_{t+1}} \beta_{t+1}(j) \tag{9}$$

$$\beta_{T-1}(i) = 1 \tag{10}$$

To solve the decoding problem we need to calculate *both* the backward and forward variables (α, β) with a backwards and forward dynamic programming pass. Now define:

$$\gamma_t(i) = P(y_t = q_i | X, \lambda) = \frac{\alpha_t(i) \beta_t(i)}{P(X|\lambda)} \tag{11}$$

The most likely state y_t is the state q_i for which γ_t is maximum. There are other algorithms to find the state sequence. One of them is the Viterbi algorithm. The decoding algorithms can also be heuristically evaluated by not taking in account any probabilities under a certain cutoff probability δ. This type of heuristic can be problematic when using a constrained conditional model because it needs to have each option available to evaluate.

3.1.3 The Learning Problem

Estimating λ directly is not possible, but there are iterative algorithms available to approximate A,B and π. Let's define:

$$\gamma_t(i,j) = P(y_t = q_i, y_{t+1} = q_j | O, \lambda) = \frac{\alpha_t(i) a_{ij} b_{j,x_{t+1}} \beta_{t+1}(j)}{P(X|\lambda)} \tag{12}$$

The probability of going from one state to another at time t. Then $\gamma(i)$ can be calculated as:

$$\gamma_t(i) = \sum_{j \in Q} \gamma_t(i,j) \tag{13}$$

The procedure to estimate the model involves iterating the various estimation procedures until the model converges or an iteration limit is reached. The initial distribution can be estimated with:

$$\pi_i = \gamma_0(i) \tag{14}$$

The state transition matrix can be estimated with:

$$a_{ij} = \frac{\sum_{t \in \{0,\ldots,T-2\}} \gamma_t(i,j)}{\sum_{t \in \{0,\ldots,T-2\}} \gamma_t(i)} \tag{15}$$

And the observation dependence matrix can be estimated with:

$$b_{jk} = \frac{\sum_{t \in \{0,\ldots,T-2, O_t = k\}} \gamma_t(j)}{\sum_{t \in \{0,\ldots,T-2\}} \gamma_t(j)} \tag{16}$$

The idea is to iterate and maximize $P(X|\lambda)$. The iterative estimation process is:

1. Randomly generate $\lambda = (A, B, \pi)$ that satisfies the row stochastic constraints
2. Compute $\alpha_t(i), \beta_t(i), \gamma_t(i,j)$ and $\gamma(i)$
3. Re-estimate $\lambda = (A, B, \pi)$
4. Stop when the difference in $P(X|\lambda)$ decreases below a certain convergence threshold or until a iteration limit is reached

There more efficient heuristic methods to find answers to these three questions. A lot of issues like floating point underflow or unknown words are not considered here (these usually require log-transformations or scaling). Since the sequence tagger already exists, we only want to illustrate the principles of the underlying model.

3.1.4 Label Bias

Besides being limited in the amount of positions a HMM can effectively take into account, it has another problem. Due to the markov chain structure of the state transition probabilities, each state has to transfer the incoming probability mass to its output states. This can give problems with states that have a low amount of output states (low entropy). Take for example a HMM that attempts to classify the words rib (with labels "111") or rob (with labels "222"). Each letter is considered an observation. When the letter "r" is observed, the HMM will give both labels the same probability. Due to the training, there were no observations that saw a transition from label 1 to 2 and visa versa. That

means that no matter the observations, both labels will get the probability 0.5. If during the training one word was observed more then the other word, then that word will always be assigned the highest probability, despite the observations. This problem also exists with a small output states instead of one output state, which can also result in a large part of the input being ignored. In practice, these states are quite rare, but the bias exists and can be problematic.

3.2 Segmentation

Segmentation splits the resume in different sections and items. Segmentation is done using a line-based conditional random field[6][4][11]. A conditional random field is a discriminative maximum entropy model that uses features on each line (or token) to determine what label to assign. These models can be used to segment or tag sequential data. It is a type of markov random field conditioned on the observations that can model certain dependencies that ordinary hidden markov models cannot. It also is not affected by the label bias problem. Another advantage is that it can work at the line-level instead of the token-level, this makes sure the model respects the implicit constraint that item and section boundaries should happen on line-boundaries. In Textkernel this model is used to segment the resumes in sections (personal section, experience section, etc. . .) and items.

In Textkernel this model works on a line-level, meaning the problem it tries to solve is assigning tags to individual lines. It uses calculated features and features from the sequential HMM tagger and other sources to classify each line and to properly segment the data. The current setup uses the sequential tagger and the segmentation algorithm in tandem:

1. Classify the whole document with the HMM sequence tagger
2. Use the data from the sequence tagger as features in the conditional random field to segment the data in sections
3. Classify the individual sections with the sequence tagger
4. Use the data from the sequence tagger to segment the data into items

Given:

- X, Y the random variable representing the observations and labels
- $G = \{V, E\}$ a graph such that it indexes that the random variable $Y = Y_v, v \in V$
- Y respects the markov property with respect to G and is conditioned on X: $P(Y_v|X, Y_w, w \neq v) = P(Y_v, X, Y_w, (w, v) \in E)$

Then (X, Y) is a conditional random field. Given that:

- f_k is a feature on an edge e, with λ_k being the weight for this feature
- g_k is a feature on a vertex v, with μ_k being the weight for this feature
- $\phi = \{\lambda_1, \lambda_2, \ldots, \mu_1, \mu_2, \ldots\}$ is the set of parameters

If G is a tree (For graphs which are not a tree, exact inference is intractable) then the joint distribution (derived from it being a random field) has the form:

$$P(y|x) \propto \exp(\sum_{e \in E,k} \lambda_k f_k(e, y|_e, x) + \sum_{v \in V,k} \mu_k g_k(v, y|_v, x)) \tag{17}$$

These features are properties of the observations. Most of the features in the model at Textkernel are derived from the labeled HMM data.

3.2.1 The Evaluation Problem

We assume that G is a chain (it can have other forms), this means that the conditional probability can be expressed as a transition matrix Y with the extra states $Y_0 = \text{start}, Y_{n+1} = \text{stop}$. For each position i in the observation sequence X, define:

$$M_i(y', y|x) = \exp(\sum_k \lambda_k f_k(e_i, Y|_{e_i} = (y', y), x) + \sum_k u_k g_k(v_i, Y|_{v_i}, x)) \tag{18}$$

While normalization factor is the product of M_i:

$$Z_\Phi(x) = (M_1(x), M_2(x), \ldots, M_{n+1}(x))_{0,n+1} \tag{19}$$

The conditional probability of a label sequence y given an observation sequence then is:

$$P_\phi(y|x) = \frac{\Pi M_i(y_{i-1}, y_i|x)}{(\Pi(x))_{0,n+1}} \tag{20}$$

3.2.2 The Learning Problem

The learning problem is defined as finding the set ϕ that maximizes the log-likelihood function:

$$O(\phi) = \sum_i \log p_\phi(y_i|x_i) \tag{21}$$

$$\propto \sum_{x,y} \tilde{P}(x, y) \log p_\phi(y|x) \tag{22}$$

These parameters can be found with iterative scaling algorithms. Meaning that we start with a set of default weights each initialized to 0, which are then iteratively updated with the following rules:

$$\lambda_k := \lambda_k + \delta\lambda_k \tag{23}$$

$$\mu_k := \mu_k + \delta\mu_k \tag{24}$$

The update delta $\delta\lambda_k$ is the solution to the equation:

$$\tilde{E}(f_k) = \sum_{x,y} \tilde{P}(x, y) \sum_i f_k(e_i, y|x) \tag{25}$$

$$= \sum_{x,y} \tilde{P}(x) p(y|x) \sum_i f_k(e_i, y|_{e_i}, x) e^{\delta\lambda_k T(x,y)} \tag{26}$$

Where $T(X, Y)$ is the total feature count defined as:

$$T(x, y) = \sum_{i,k} f_k(e_i, y|_{e_i}, x) + \sum_{i,k} g_k(v_i, y|_{v_i}, x) \tag{27}$$

The equations for μ_k are almost the same. To calculate these equation using dynamic programming can be inefficient because T can vary. One way to deal with this is using slack features:

$$s(x,y) = S - \sum_{i,k} f_k(e_i|_{e_i}, x) - \sum_{i,k} g_k(v_i, y|_{v_i}, x) \tag{28}$$

Where S is chosen such that $T(x,y) = S$, the new feature number. We define forward and backward recurrence vectors just like with the HMM algorithm:

$$\alpha_0(y|x) = \begin{cases} 1 \text{ if } y = start \\ 0 \text{ else} \end{cases} \tag{29}$$

$$\alpha_i(x) = \alpha_{i-1}(x) M_i(x) \tag{30}$$

$$\beta_{n+1}(y|x) = \begin{cases} 1 \text{ if } y = stop \\ 0 \text{ else} \end{cases} \tag{31}$$

$$\beta_i(x) = M_{i+1}(x)\beta_{i+1}(x) \tag{32}$$

We can use these recurrence vectors to solve the equations for $\delta\lambda, \delta\mu$:

$$\delta\lambda_k = \frac{1}{S} \log \frac{\tilde{E}f_k}{Ef_k} \tag{33}$$

$$\delta\mu_k = \frac{1}{S} \log \frac{\tilde{E}g_k}{Eg_k} \tag{34}$$

$$Ef_k = \sum_{i,y',y} f_k(e_i, y|_{e_i} = (y', y), x) \frac{\alpha_{i-1}(y'|x) M_i(y', y|x) \beta_i(y|x)}{Z_\Phi(x)} \tag{35}$$

$$Eg_k = \sum_{i,y} g_k(v_i, y|_{v_i} = y, x) \frac{\alpha_i(y|x)\beta_i(y|x)}{Z_\Phi(x)} \tag{36}$$

$$\tag{37}$$

Which then gives the update rules necessary to find the feature weights. You usually iterate until the consecutive changes are below a certain convergence threshold and the model states converge.

4 Constrained Conditional Models

Constrained conditional models[1][8][9] are a way of imposing rules or constraints on NLP algorithms. The NLP algorithm is reformulated as a constraint optimization problem in such a way that additional constraints can be added. This is done by reformulating the original NLP algorithm in the form of a goal function with a set basic consistency constraints. Any additional constraints can be considered prior knowledge about how the output is "supposed" to look and what is not supposed to happen. There are two types of constraints that can be used:

- Hard constraints: Constraints that contain *absolute* prior knowledge about the dataset. The solver will not be able to find solutions that do not satisfy these constraints
- Soft constraints: Constraints that contain prior knowledge that is *mostly* true. Solutions that violate these constraints are penalized with a penalty ρ and the solver will try to avoid these situations with a degree depending on the penalty

These constraints are imposed on a number of local features that come from the underlying NLP model(s) (it is possible to combine multiple NLP in a CCM, we do not do this however) or manually defined features. The manually defined features can be used to incentive or disincentive certain local choices the solver has to make. The goal function tries to optimize the local features (depending on the underlying model) while satisfying all the hard constraints and trying to avoid the soft constraints depending on their penalty.

Given:

- $\mathcal{X}$ a set of tokens
- $\mathcal{Y}$ a set of tags
- $\theta_i : \mathcal{X} \times \mathcal{Y} \rightarrow \mathcal{R}$ a set of local properties of token and tags pairs. These can be probabilistic from conditional models or local calculated features. To incorporate more backwards looking models, it can also be defined as $\theta_i = \mathcal{X}^n \times \mathcal{Y} \rightarrow \mathcal{R}$
- $C = \{C_i(\cdot)\}, C_i : \mathcal{X}^M \times \mathcal{Y}^M \rightarrow \{0, 1\}$ a (usually small) set of constraints defined as predicate statements over the current tag assignment
- $d_{C_i} = \mathcal{X}^M \times \mathcal{Y}^M \rightarrow \mathcal{R}$ a set of distance functions that measure how much a solution violates constraint i
- $\rho = \{\rho_i\}$ a set of weights for each constraint that indicate how much each constraint violation is penalized. If a constraint i is hard then $\rho_i = \infty$
- $W = \{w_i\}$ a set of weights for each local feature

The goal of a constrained conditional model is to find:

$$y^* = argmax_{\mathbf{y}} \sum_i w_i \theta_i(\mathbf{x}, \mathbf{y}) - \sum_i \rho_i d_{c_i}(\mathbf{x}, \mathbf{y}) \tag{38}$$

That is, the goal is to find a solution that maximizes the local features with weights $\mathbf{w}$ while avoiding violating constraints with weight ρ.

4.1 Integer Linear Programming Formulation

We formulated the constrained conditional model as an integer linear programming model. The formulation of an integer linear programming model consists of a linear goal function (that has to maximized or minimized) and a set of linear equations or inequalities that represent the constraints. These optimization and constraint functions are defined on a set of integer variables. We define the constrained conditional model in the form of an integer linear programming model because:

- The constraints are in a form that is easy and fast to check (with vector multiplication)
- There are a lot of programs and theory available to solve these models in an exact or heuristic fashion

Because we want to define constraints on tag transitions and sequences as well as tag assignments, we define an integer linear programming model with the following set of boolean decision variables:

$$\mathcal{V} = \{\forall i \in \mathcal{X}, y' \in \mathcal{Y}, y \in \mathcal{Y} : \mathbf{1}_{y_{i-1}=y' \wedge y_i = y} \in \{0, 1\}\} \tag{39}$$

These variables indicate if token i is assigned to y and token $i-1$ to y'. The variable is 1 if both these assignments exists and it is 0 if one or both of these assignments do not exists. These type of "joint" variables are a bit more powerful than indicator variables for single tag assignments. These variables make it possible to put constraints on tag sequences (consecutive assignments of the same tag), tag transitions and not just on single tag assignments.

We want to rewrite the original HMM decoding problem as an ILP with the set variables of variables $\mathcal{V}$. For a first order HMM, the decoding problem is to find a set of tags that maximize the joint probability of the tokens and tags:

$$y^* = \max_{y_i \in Q} P(Y_0 = y_0)P(X_0 = x_0|Y_0 = y_0)\Pi_{i \in Y} P(Y_i = y_i|Y_{i-1} = y_{i-1})P(X_i = x_i|Y_i = y_i) \tag{40}$$

This optimization problem cannot be directly represented as an ILP because it is not a *linear* function, it has to be log-transformed. The equivalent representation of the first order HMM (based on the principle that $\log(a) + \log(b) = \log(a \cdot b)$) as an integer linear programming model is:

$$argmax_y \sum_{y \in \mathcal{Y}} \lambda_{0,y,start} \mathbf{1}_{y_{-1}=start \wedge y_0 = y} + \sum_{i \in \mathcal{X} \setminus \{0\}, y \in \mathcal{Y}, y' \in \mathcal{Y}} \lambda_{i,y',y} \mathbf{1}_{y_i = y \wedge y_{i-1} = y'} \tag{41}$$

under

$$\forall i \in \mathcal{X} : \sum_{y' \in \mathcal{Y}, y \in \mathcal{Y}} \mathbf{1}_{y_{i-1}=y' \wedge y_i = y} = 1 \tag{42}$$

$$\forall y \in \mathcal{Y}, i \in \mathcal{X} : \sum_{y' \in \mathcal{Y}} \mathbf{1}_{y_{i-1}=y' \wedge y_i = y} = \sum_{y'' \in \mathcal{Y}} \mathbf{1}_{y_i = y \wedge y_{i+1} = y''} \tag{43}$$

$$\sum_{y \in \mathcal{Y}} \mathbf{1}_{y_{-1}=start \wedge y_0 = y} = 1 \tag{44}$$

The log-transformed weightings in the goal function for a first order HMM are:

$$\lambda_{0,y,start} = \log(P(Y_0 = y_0) \cdot P(X_0 = x_0|Y_0 = y_0)) \tag{45}$$

$$\lambda_{i,y,y'} = \log(P(Y_i = y|Y_{i-1} = y') \cdot P(X_i = x_i|Y_i = y_i)) \tag{46}$$

We cannot use these log-transformed weights in our model because the sequence tagger used at Textkernel:

- Is a second order HMM
- Is not able to output all the probabilities conditioned on the previous taggings due to the specific algorithms used within the sequence tagger

Instead we use a probability estimate that is conditioned on the most probable previous choices instead of the actual previous choice:

$$\lambda_{0,y,start} = \log(P(Y_0 = y_0) \cdot P(X_0 = x_0|Y_0 = y_0)) \tag{47}$$

$$\lambda_{i,y,y'} = \log(P(Y_i = y|Y_{i-1} = \hat{y}', Y_{i-2} = \hat{y}'') \cdot P(X_i = x_i|Y_i = y_i)) \tag{48}$$

Where $\hat{y}$ is the tag value that was the best solution according to the HMM. This estimate depends on the fact that most of the correct solutions for the CCM contain the most likely choice given by the HMM.

The initial constraints in the ILP formulation (see equation (41)) are all "consistency" constraints that make sure that ensure that the solution vectors satisfy some very basic rules:

1. There is exactly one tag for each token (see equation (42))
2. The previous tag value matches up with the actual previous tag (see equation (43))
3. The tag at position -1 is the special "start" tag (see equation (44))

In our solver, these consistency constraints are not checked explicitly but used only implicitly when building the different possible solutions.

For the purpose of solving a formulated constrained conditional model we store the constraints and the optimization function in the canonical ILP form:

$$\max cx \tag{49}$$
$$under \tag{50}$$
$$Ax \geq b \tag{51}$$
$$x \in \{0, 1\} \tag{52}$$

Where:

$$\mathbf{1}_{y_i=y, y_{i-1}=y'} = x_{i \cdot |\mathcal{Y}|^2 + y' \cdot |\mathcal{Y}| + y} \tag{53}$$
$$\sum w_j \theta_j(x_i, y_i) = c_{i \cdot |\mathcal{Y}|^2 + y' \cdot |\mathcal{Y}| + y} \tag{54}$$
$$\tag{55}$$

ILP models can be formulated in different forms. The most general form of the ILP model can contain equality and different types of inequality constraints. The canonical form is simplified and only contains $\geq$ or equality constraints. The standard form contains only equality constraints with extra slack variables added that compensates for changing the inequality constraints. All these representations can represent exactly the same model. The standard form can be used for the general exact ILP solvers, the canonical form is easier to handle for the solvers we build (we do not have to handle slack variables explicitly this way) while constraints are defined in the general form for readability purposes.

To convert the constraints to the canonical or standard form from the general form, two steps have to be taken:

- Normalize all inequalities to the $\geq$ or equality constraint type. This gives us the canonical form of the ILP model
- Give all the inequality constraints a slack variable and convert them to equality constraints. This gives us the standard form of the ILP model

Because we need the ability to use soft constraints, we also need to define the "degree" a constraint is violated. Given that all constraints in the canonical form are either equality constraints or $\geq$ constraints, the constraint distance is given by:

$$d_{c_i}(x) = \begin{cases} b_i - A_i \cdot x^T & \text{if constraint violated} \\ 0 & \text{else} \end{cases} \tag{56}$$

This distance function allows us to create soft constraints by assigning penalties based on on this degree of violation. Any soft or in this case "relaxed" constraints (defined as $\mathcal{L}$) can be put in the goal function as:

$$argmax_y \sum_{y \in \mathcal{Y}} \lambda_{0,y,start} \mathbf{1}_{y_{-1}=start \wedge y_0=y} + \sum_{i \in \mathcal{X} \setminus \{0\}, y \in \mathcal{Y}, y' \in \mathcal{Y}} \lambda_{i,y',y} \mathbf{1}_{y_i=y \wedge y_{i-1}=y'} - \sum_{i \in \mathcal{L}} \rho_i d_{c_i}(\overline{y}) \tag{57}$$

One of the main problems when finding solutions for a model is discarding (or pruning) large sets of wrong solutions. Some search strategies involve building solutions step by step, with a lot of intermediate partial solutions. A partial solution is defined as a solution that does not have an assigned tag for each token, meaning it violates the first consistency constraint:

$$\forall i \in \mathcal{X} : \sum_{y' \in \mathcal{Y}, y \in \mathcal{Y}} \mathbf{1}_{y_{i-1}=y' \wedge y_i=y} = 1 \tag{58}$$

You want to quickly prune any partial solutions that violate constraints and you want to know how much constraints on those partial solutions are violated. Since partial solutions violate a consistency constraint, you cannot generally state that all constraints in a model can be applied to partial solutions.

The set of solutions that can be evaluated partially are in the set $\mathcal{P}$. A constraint i is in $\mathcal{P}$ if it satisfies two rules:

1. An empty solution x satisfies constraint i
2. Any partial solution x which does not satisfy constraint i cannot be "completed" in such a way that it now satisfies constraint i

For example the constraint "there should be more then one token labeled as Job Title" cannot be evaluated partially while the constraint "there cannot be more then one tag sequence with Job Title" can be evaluated partially. The difference is that empty solutions do not satisfy the first constraint while they do satisfy the second constraint. Therefore the first constraint only applies to complete solutions while the second constraint can be applied to any partial solution.

If the constraint "there should be more tag sequences of type Job Title then tag sequences of type Experience Description" is evaluated partially then solutions that might become correct when completing the solution can be considered to be violating this constraint. For example a partial solution that only contains the tag "Experience Description" is considered invalid, while the completed solution might also contain the tag "Job Title". So depending on the way solutions are build, evaluating this constraint partially will influence the set of solutions an "incremental" strategy can find. Now solutions must first contain "Job Title" tags before the "Experience Description" tag can be added. This might not matter much depending on the circumstances, but it will still disallow a set of solutions that do not violate the constraint.

Luckily, a lot of practical constraints are in $\mathcal{P}$. Because it is hard to automatically determine if constraints are in $\mathcal{P}$, these constraints can be marked as such in the constraint definition file.

4.1.1 Constraint Formulation

This ILP model (see equation (41)) defines the baseline optimization problem. This formulation allows us to inject prior knowledge in this model in the form of additional constraints added to this model. The constraints can be defined in a number of different ways. The raw form of the constraints are the linear equations in the canonical integer linear programming model. Defining the constraints like this might be very easy to process for the solver program, but it might not be a natural way to define these rules for most people.

Linear Constraints The linear constraints are basically (in)equality equations on the set of boolean variables $\mathcal{V}$. They are the "raw" form any other type of constraint ends up as. This means there is no translation step and the constraints you define are directly incorporated in the model as such. For some examples of such constraints, see Table 2.

Constraint	Expression
For token i, tag j should not precede tag k	$\mathbf{1}_{y_{i-1}=j \wedge y_i=k} = 0$
Token i should be tagged j	$\sum_{k \in \mathcal{Y}} \mathbf{1}_{y_{i-1}=k \wedge y_i=j} = 1$
The presence of tag i implies that there should be a tag j	$\sum_{k \in \mathcal{X}, l \in \mathcal{Y}} \mathbf{1}_{y_{k-1}=l \wedge y_k=j} - \sum_{k \in \mathcal{X}, l \in \mathcal{Y}} \mathbf{1}_{y_{k-1}=l \wedge y_k=i} \geq 0$
The presence of a sequence of tags i implies that there should be a sequence of tag j	$\sum_{k \in \mathcal{X}, l \in \mathcal{Y} \setminus \{i\}} \mathbf{1}_{y_{k-1}=l \wedge y_k=j} - \sum_{k \in \mathcal{X}, l \in \mathcal{Y} \setminus \{y\}} \mathbf{1}_{y_{k-1}=l \wedge y_k=i} \geq 0$

Table 2: set of constraints defined in terms of the linear model

First Order Logic Constraints First order logic constraints are statements in formal logic over the variables used in the model that are either true or not true. The first order logic sentences can be written with a combination of quantifiers, variables, truth operators, and implication arrows. These statements need to be translated to linear equations so they can be checked faster and more efficiently during the search procedure. This can be done with the following steps:

1. Convert the constraints to conjunctive normal form (NP-Hard)
2. Normalize and redistribute the constraints
3. Convert them to vectors from the normalized form

This form of the constraints is recommended in the literature about constrained conditional models, but it has a number of practical disadvantages:

- The translation algorithm has exponential time complexity and involves a lot of steps where the (sometimes very large) constraint vectors need to be copied. This translation step must be done for each instance because each time the number and type of the tokens differ from each other. Some of these steps can be cached but it still takes a large amount of time which is precious in this environment

- For some statements the translation step generates extra constraints and auxiliary variables. The existence of auxiliary variables makes sure that any heuristic solvers need a lot of extra logic to deal with these variables, while for most constraints they are not necessary
- Avoiding the generation of auxiliary variables and extra constraints leads to a restricted set of possible constraint statements. While most practical constraints do not need auxiliary variables, the restrictions of constraints without auxiliary variables are quite unintuitive and difficult to understand
- The statements in first order logic are not much more simple then the linear statements. There are other methods to somewhat simplify the constraints without these disadvantages

For these reasons we decided not to use these constraints to define the constraints in the CCM. For examples on how these constraints would look, see Table 3.

Constraint	Expression
For token i, tag j should not precede tag k	$\neg \mathbf{1}_{y_{i-1}=j \wedge y_i=k}$
Token i should be tagged j	$\exists k \in \mathcal{Y} : \mathbf{1}_{y_{i-1}=k \wedge y_i=j}$
The presence of tag i implies that there should be a tag j	$(\exists k \in \mathcal{X}, l \in \mathcal{Y} : \mathbf{1}_{y_{k-1}=l \wedge y_k=i}) \implies (\exists k \in \mathcal{X}, l \in \mathcal{Y} : \mathbf{1}_{y_{k-1}=l \wedge y_k=i})$
The presence of a sequence of tags i implies that there should be a sequence of tag j	$(\exists k \in \mathcal{X}, l \in \mathcal{Y} \setminus i : \mathbf{1}_{y_{k-1}=l \wedge y_k=i}) \implies (\exists k \in \mathcal{X}, l \in \mathcal{Y} \setminus j : \mathbf{1}_{y_{k-1}=l \wedge y_k=j})$

Table 3: set of constraints defined in terms of first order logic

Templated Constraints Because raw constraints are typically a bit difficult to enter, and we want other people to enter new constraints into the system, we use a templating system to abstract away common types of constraints. These have the following advantages:

- You don't need to repeat the definition of common types of constraints
- Other people will be able to enter new constraints or define new template types
- The translation step to the linear constraints is efficient

For examples of templated constraints, see Table 4, note that real templates are a bit more verbose, but these examples should show the general idea.

Constraint	Expression
For token i, tag j should not precede tag k	$ShouldNotPrecede(i, j, k)$
Token i should be tagged j	$ShouldBeTagged(i, j)$
The presence of tag i implies that there should be a tag j	$TagImpliesTag(i, j)$
The presence of a sequence of tags i implies that there should be a sequence of tag j	$SequenceImpliesSequence(i, j)$

Table 4: set of constraints defined in terms of templates

4.2 Solving the constrained conditional model

We build a solver that attempts to find solutions for the model. We attempt to maximize the defined goal function while satisfying all the states strict constraints. The solver has the following design goals:

- Finding the best solution we can in the time we are given
- Make sure we can interrupt the algorithm at any point and take the best solution so far (anytime behavior)
- Make sure we find an initial solution that satisfies the constraints as fast as possible

4.2.1 Complexity of the search space

A solution of the ILP model is defined as a particular assignment of the decision variables. Under the default ILP model (with consistency constraints), the amount of possible solutions is $\|\mathcal{Y}\|^{\|\mathcal{X}\|}$. The worst case time to find the optimal solution is proportional to the size of this search space (see Figure 2).

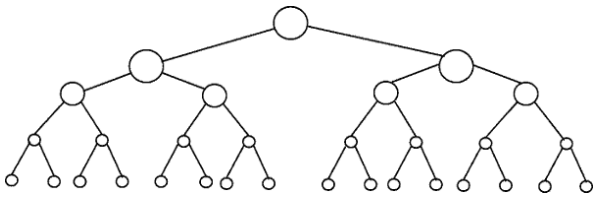

Figure 2: The size of the search space increases exponentially with regards to the number of tokens in the problem instance, for each token a choice needs to be made

Because the search space increases exponentially with the amount of tokens, we use different algorithms for a large number of tokens and a small number of tokens. When the search space is small

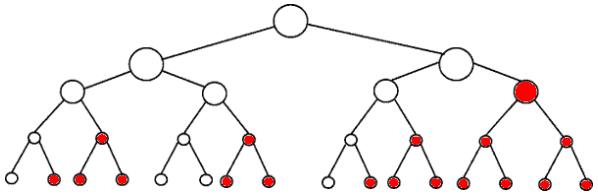

Figure 3: The approach of the search algorithm is to prune as much of the search space as possible

we can search a well defined partition systematically. But because we have a limited amount of time, we use a different strategy when there are instances that contain a large amount of tokens.

4.2.2 Limited Discrepancy Search

For smaller search spaces we use a combination of limited discrepancy search[5] and branch & bound[7]. The aim here is to search only the subset of the most likely solutions and prune the search space aggressively (see Figure 3). We use the value of the goal function to guide the solver.

The strategy of limited discrepancy search is to give a penalty (called the discrepancy value) for each heuristically suboptimal choice. If this penalty exceeds some sort of limit, the search node is pruned. When the search fails to find anything, this limit is increased and the search is retried. This heuristic ensures that the search procedure does not end up in a local maximum. If the algorithm made a bad choice which later leads to being forced to violate a number of constraints, then it would be forced to backtrack from this local minimum.

Each choice that can be made has a certain score. We define discrepancy of each choice as the difference with the maximum possible score. Since the model is log-linear, with a maximum of 0 (if all ρ are positive and you disregard any additional features with positive weight). This means we can define the discrepancy as the goal function of the canonical ILP model:

$$discrepancy = \sum_{y \in \mathcal{Y}} \lambda_{0,y,start} \mathbf{1}_{y_{-1}=start \wedge y_0=y} + \sum_{i \in \mathcal{X} \setminus \{0\}, y \in \mathcal{Y}, y' \in \mathcal{Y}} \lambda_{i,y',y} \mathbf{1}_{y_i=y \wedge y_{i-1}=y'} - \sum_{i \in \mathcal{L}} \rho_i d_{c_i}(\overline{y}) \quad (59)$$

The idea of branch & bound when maximizing a function is to use a known global minimum score to prune branches for which the maximum score they can attain is smaller then this global minimum. Whenever we find a solution, we have a new "known" best score, this score is used as the minimum score that solutions should be able to attain. This can be used to prune nodes for which we know the maximum score they could possibly attain, which is the current discrepancy value. This heuristic is added to the limited discrepancy search by adjusting the allowed discrepancy value when a new solution is found.

The branch & bound heuristic can be problematic for certain problem instances. Positive feature weights can increase the score of a partial solutions in subsequent choices. This could result in potentially better solutions with a higher goal function value being thrown away. Because we try to avoid heavy feature weights in the constrained conditional model, this should have a limited impact.

We start with a maximum allowed discrepancy of $M = \alpha \cdot \|\mathcal{X}\| \cdot m^i$ where α is the token multiplier, M is the maximum globally allowed discrepancy, m is the iteration multiplier and i is the current iteration (starts at zero). Because you need to make a choice for each token, the reasoning behind this discrepancy formula is that you want to restrict the search from deviating from a maximum allowed

average penalty for each token. When the first search fails to find any solution, the iteration count i is increased. When it does find a solution it restricts the search space further by setting the discrepancy to the found goal value (branch & bound heuristic). Additionally, we restrict the allowed number of choices that can be tried (the branching factor). We start of with in an initial maximum branching factor b, which is increased each iteration with δb.

In short, we restrict the search space in the following ways:

- A large part of the "suboptimal" choices are restricted by the initial discrepancy maximum
- When a solution is found the discrepancy is further limited with the branch & bound heuristic, making sure no worse solutions are found
- The branching factor is limited by a maximum that increases each iteration

Because these restrictions can be "loosened", the algorithm will find a solution eventually if there exists one. Here is an implementation of the extended discrepancy search algorithm for solving a constrained conditional model:

```
// get all the valid successors in order of the most likely choices
def prune(successors, threshold, beamSize) {
  successors.map(successor => (successor, score(successor))) // score each possibility
            .filter((successor, score) => score >= threshold) // prune solutions below
                threshold
            .sort((successor, score) => score) // sort by the score
            .take(beamSize) // cut of any solutions above the allowed beamSize
}

// try a search procedure
def probe(currentSolution, currentDepth, previousChoice) {
  if(currentDepth < tokenCount) {
    currentBest = None
    successors = generateSuccessors(previousChoice, currentDepth)
    pruned = prune(successors, threshold, beamSize)
    for(choice <- pruned) {
      if(timerDone)
        return currentBest
      result = probe(currentSolution + choice, currentDepth + 1, choice, threshold,
          beamSize)
      if(result > currentBest) {
        currentBest = result
      }
    }
    return currentBest
  } else {
    threshold = max(threshold, score(currentSolution))
    return currentSolution
  }
}

def search(startThreshold, multiplier, startBeamSize) {
  bestSolution = None
```

```
  // set the global search space restriction variables
  threshold = startThreshold
  beamSize = startBeamSize

  // search until we've found something or until the timer is done
  while(!timerDone && bestSolution = None) {
    bestSolution = probe({}, 0, 0)
    // widen the allowed search space
    threshold *= multiplier
    beamSize += 1
  }

  return bestSolution
}
```

Having a fast way to check constraints and calculate the discrepancy is essential for finding solutions fast. The way the constraints and the goal function are formulated allow us to incrementally calculate constraint violations and the current value of the goal function in a fast way. We store a number of partial constraint values (as $\forall i \in \mathcal{P} : pc_i$) and the current value of the features (as pv). These are updated each time we make a choice or when we backtrack on that choice. We use the following update rule when assigning j:

$$pc := pc + A_{:,j} \tag{60}$$
$$pv := pv + c_j \tag{61}$$

And the following rules when backtracking on j:

$$pc := pc - A_{:,j} \tag{62}$$
$$pv := pv - c_j \tag{63}$$

The equations should be guarded against infinity or other values that could cause numeric instability. These incrementally calculated values allow us to check the constraints quickly for any partial solution:

```
def constraintCorrect(constraint, solution) {
  if(isComplete(solution) || constraint in P) {
    if(hasSlack(constraint)) { // do not assign slack variables explicitly, use canonical
        form
      return pc(constraint) >= b(constraint)
    } else {
      return pc(constraint) == b(constraint)
    }
  }
}

def constraintDistance(constraint, solution) {
  if(!constraintCorrect(constraint, solution) && (isComplete? solution || constraint in P))
      {
    return b(constraint) - pc(constraint)
  } else {
    return 0
  }
}
```

```
def constraintPenalty(solution) {
  return sum(constraints.map(constraint => gamma(constraint) *
      constraintDistance(constraint, solution)))
}

def disprecancy(solution) {
  return pv - constraintPenalty(solution)
}
```

In literature beam search is usually recommended, but this extended limited discrepancy search has a number of clear advantages over beam search:

- Since it is a depth first search, the time to find the first solution is a lot smaller then beam search, which is a breath first search. Having a large initial setup time without any solutions runs counter to the strategy of using a anytime algorithm. We would be unable to interrupt the algorithm for a long time while it was searching for an initial solution. It would also limit the usefulness of the branch & bound heuristic
- It uses a long distance heuristic that prunes nodes more intelligently then the standard beam search and is not much more costly to calculate
- It is guaranteed to find a solution eventually by expanding the allowed bounds of the search and retrying. The initial bounds should be chosen in such a way that most problems can be solved in the initial iteration. A large amount of unnecessary iterations increases the time the algorithm cannot be interrupted because it has no initial solution

4.2.3 Evolutionary Algorithm

Initialization	best first search result + mutation
Representation	sparse vector with length $\|Y\|^2\|X\|$
Recombination	2-point crossover on tag boundaries
Mutation	section or border flip
Parent Selection	uniform random
Survivor Selection	$\mu + \lambda$
Constraint Handling	implicitly handling consistency constraints, with increasing penalty for strict constraints

Table 5: Overview of the evolutionary algorithm

For larger instances of the problem, the limited discrepancy search can only search a very limited section of the pruned search space, even with a very small α it cannot search the whole pruned search space which means that it is reduced to a mostly best-first search.

There are a number of meta-heuristic algorithms which might be able to improve on best first search with such a large search space. Possible choices for such algorithms might be hill climbing, simulated annealing, tabu search, particle swarm optimization or evolutionary algorithms. We choose to create an evolutionary algorithm to solve this problem because we have the ability to customize the algorithm to fit this specific problem.

Evolutionary algorithms[3] are a class of population-based heuristic optimization algorithms that are inspired by evolution. They use a combination of mutation, recombination and selection operators generate new solutions from a population and to navigate the search space. These operators allow us to customize this search to fit our specific problem. Our implementation of the evolutionary algorithm exploits a number of biases in typical solutions:

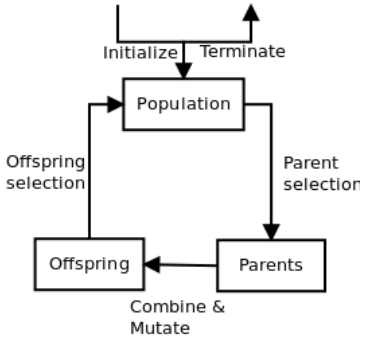

Figure 4: Evaluation loop of a evolutionary algorithm

- Tags are usually applied to tokens in large sequences at the same time
- The right solution can often be found by a limited shifts of the border of tag sequences or by flipping an entire sequence to a different type of tag

These biases can used by modifying the operators to take into account the tag borders in the solutions. This allows us to skip a lot of mostly irrelevant solutions.

Our implementation of the evolutionary algorithm is fairly typical (see Table 5):

1. Find a solution with best-first search that satisfies the hard constraints. Apply the mutation operator a number of times to get an initial population to start the search with
2. Evaluate the population (score each solution with the goal function) and select parents from the population using the selection operator
3. Use the recombination operator to generate new offspring, and use the mutation operator to change a limited number of offspring
4. Select the survivors from the pool of parents and offspring (a $\mu + \lambda$ strategy). We make sure to keep the best best solution and the best solution that satisfies all constraints in the population (this is called elitism)
5. Go to step 2 until a time limit is reached

Recombination & Mutation The recombination and mutation operators (see Figure 5) generate each new generations of solutions. Both operators use biases inherent in the typical structure of a solution. The operators obey the consistency constraints implicitly by generating only solutions that satisfy these constraints. The only constraint violations that need to be corrected are violations on the consistency constraint of the previous-next variables:

$$\forall y \in \mathcal{Y}, i \in \mathcal{X} : \sum_{y' \in \mathcal{Y}} \mathbf{1}_{y_{i-1}=y' \wedge y_i=y} = \sum_{y'' \in \mathcal{Y}} \mathbf{1}_{y_i=y \wedge y_{i+1}=y''} \tag{64}$$

When the solution is mutation or combined, the edges of these solutions need to be consistent with regards to the previous value pointers of the variables. The correction only involves setting the current border variable to zero and setting the variable with the correct previous label to one.

Both operators are based on the tag transition points. The set of transition points of solution s_j is given by:

$$T_j = \{\forall i : \mathbf{1}_{y_{i-1}=y' \wedge y_i=y} = 1 \text{ where } y' \neq y\} \tag{65}$$

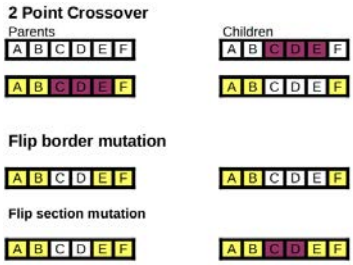

Figure 5: Visual representation of the effect of the various operators

Any point where the tag sequence changes can be used as a mutation or recombination point. The recombination operator (on solution s_i and s_j) is a 2-point crossover operator defined as:

1. Pick a random tag transition point t_k from T_i
2. Swap the contents of s_i between t_k and t_{k+1} (inclusive, exclusive) with s_j, if there is no next transition, swap until the end of the solution
3. Rewrite the border variable such that they satisfy the consistency constraints

The mutation operator (which is applied with probability β to offspring) can do two different things.

1. With probability ω it shifts the tag transition point t_k one position backwards, meaning it shifts the tag "border" one position
2. With probability $1 - \omega$ it flips the section between point t_k and t_{k+1} (inclusive, exclusive) to a random section

After this mutation has been processed, it corrects the solution for the consistency constraints by adjusting the transition points.

Selection We use a linear ranking scheme to assign selection probabilities to individual solutions based on their rank:

$$P_{rank}(i) = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)} \tag{66}$$

Where $s \in \{1.0 \dots 2.0\}$ is the selection pressure[6], i is the rank of the solution and μ is the size of the population. We use the universal sampling algorithm to select solutions (each with a selection probability) from a pool of possible candidates. Conceptually this algorithm is similar to making a spin with a roulette wheel with n equally spaced arms on the list of candidate solutions.

```
def universalStochasticSampling(solutions, n) {
  selected = {}
  r = randomValue(0, 1 / n)
  i = 0
  while(selected.size <= n) {
    while(r < cumProb(solutions(i))) {
      selected += solutions(i)
      r += 1 / n
    }
```

[6]The higher the number, the more biased the selection probability is to better ranked solutions

```
    i += 1
  }

  return selected
}
```

This algorithm is used to select both the survivors and parents.

Constraint Handling The operators in the previous section implicitly generate solutions that satisfy basic consistency constraints. Since these constraints are rather simple to satisfy in practice, it makes a lot of sense to do it this way instead of generating solutions and throwing a large portion of them away after they fail to satisfy these constraints.

Additional hard constraints can be problematic with the evolutionary search. The weight of ∞ on these constraints means the candidate solutions violating these constraints are most likely immediately pruned from the population. The problem with that is that often the path to a more optimal solution contains intermediate solutions that violate these hard constraints. Therefore we dynamically apply different weightings on the hard constraints depending on how long the evolutionary search has been running:

$$\forall i \notin \mathcal{L} : \rho_i = \gamma_s + i * \gamma_\delta \tag{67}$$

Where γ_s is the start weight, i is the current iteration and γ_δ is the difference for each iteration.

5 Results

5.1 Experimental Setup

We test the constrained conditional model on one particular section of a resume, the experience section. The Textkernel pipeline does the following steps (roughly) in order:

- Use the sequence tagger to tag the whole resume
- Use the CRF to segment the resume into sections
- Use the sequence tagger to tag each section
- Use the CRF to segment each section into items

Because the CCM cannot process a large amount of tokens, we apply the CCM to individual items. To do this, we add the following steps to the pipeline:

- Use the sequence tagger to tag the individual items, this sequence tagger is trained on a different dataset then the dataset we are testing it on. The output is restricted to the set of tags we are testing
- Apply the CCM on the output of the sequence tagger to find a solution that satisfies the included constraints

The aim of the experiments on these data sets is to test the performance of the model on several key tags within the experience items, these tags are:

- Job Title
- Experience Date
- Organization Name
- Experience Description

We use the following performance measures to check for improvement:

$$precision = \frac{true\text{-}positives}{true\text{-}positives + false\text{-}positives} \tag{68}$$

$$recall = \frac{true\text{-}positives}{true\text{-}positives + false\text{-}negatives} \tag{69}$$

$$F_{measure} = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{70}$$

The precision measures how much of the tags that were found are actually correct. The recall measures how of the tags that should have been found were actually found and the $F_{measure}$ is the harmonic mean (which is the appropriate way to take an average of two rates) of the precision and recall.

These performance indicators are measured at Textkernel in different ways:

- An exact measure tests the correctness (false/true positive/negative rate) of tag sections (consecutive section of tokens that are tagged in the same way), only tag sections that *exactly* match the gold test sample tag sections are marked as correct
- The line-based overlap test measures the fraction of the found taggings that are correct by calculating how many lines overlap between the found tagging and the annotated tags found in the test set. The fraction correct is given by: $\text{line-overlap} = \frac{\#(\text{found-lines} \cap \text{test-lines})}{\#(\text{found-lines} \cup \text{test-lines})}$
- Character-based overlap test measures the fraction of the found taggings that are correct by calculating the amount of individual characters (letters, punctuation marks, spaces, etc. . .) that overlap between the found taggings and annotated taggings. Non-alphanumeric characters are not taken into account. The fraction correct is given by: $\text{char-overlap} = \frac{\#(\text{found-characters} \cap \text{test-characters})}{\#(\text{found-characters} \cup \text{test-characters})}$

The best to measure precision and recall from these choices is the character-based overlap measure. This measure is used at Textkernel to test most of their improvements to the models for the following reasons:

- The line-based and exact measures are inadequate in several ways. The exact way to test correctness discards many almost correct solutions that would be adequate otherwise. The line based measure is too lenient and would accept taggings that are mostly or completely incorrect but on the same line
- It is an measure that takes into account the amount of overlap with the correct solution
- A character based test also measures errors outside the tokenisation level, which is important for Textkernel (because errors when tokenizing can happen and have to be tracked down) but does not really matter in these tests
- It weights small words and punctuation marks less than larger words (which is usually correct)

The parameters for the base setup are given by Table 6.

Parameter	Value
Border between discrepancy search and evolutionary search	60 tokens
The allowed branching factor b	3
The branching factor increase δb	1
Token multiplier α	-1.3
Iteration multiplier m	1.5
Mutation Rate β	0.25
Selection Pressure s	1.5
Population Size λ	30
Sequence Mutation Rate ω	0.9
Start weight hard constraints γ_s	2.0
Difference for weight hard constraints γ_δ	1.0
Given run time (excluding preprocessing/postprocessing)	800ms

Table 6: Parameters for the base setup

5.2 The Tests

Languages To test if constraints found for one language generalize over multiple languages, we test a set of constraints that work well on resumes from one language, on another language:

- We first find a set of constraints that work well for the English resumes
- Then we test if those constraints perform well for the German resumes

Cascading Errors The constrained conditional model might have problems with cascading errors. This means that errors in previous layers of the pipeline (more specifically, the segmentation layer) can multiply and cause additional errors in the output of the HMM and the CCM. We will use two different setups to test this issue:

- One setup where the segmentation is done normally
- One setup where the segmentation data is taken from the gold dataset, which means we can assume we have perfectly segmented data as input

Evolutionary Algorithm To test the improvements (or lack of them) made by the evolutionary algorithm, we test the following setups against the original setup:

- With only the evolutionary algorithm
- With only the disprecancy search

Wrong paths We will also show some test instances where we did not get the results we wanted:

- When using different types of constraints: We have tried a lot of different constraints to correct errors encountered. Most of those constraints did not show improvements, here we show the results of one such example
- When using absolute constraints: Here we test the CCM with absolute constraints
- When using absolute constraints and local features with high weight: Using hard constraints necessitates using local features with high weights to force the solver to make certain choices. Here we show the results of such high weights
- When using no extra features: Local features are a useful tool for incentivizing certain choices. Here we use a test the usefulness of local constraints

5.3 Resulting Data

Base Performance *REDACTED*

Languages The CCM for which we found the best improvements (see Table 7) had the following constraints and features:

- Constraint: The should be at most one consecutive description tag with weight 1.3
- Feature: A transition from any kind of description tag to any other tag then a IGN tag penalized with weight -0.9
- Feature: A transition from any kind of description tag to any other kind of description tag promoted with weight 0.01
- Feature: A transition from any kind of Job Title to any other kind of Job title promoted with weight 0.05

The best improvements found are for the description tags (which makes sense because those are the errors we were trying to correct), but the correction of the description tags has as residual effect that the performance of the other tags are improved. If incorrect description tags are removed, then they might be replaced by the correct tags, improving the recall of other tags. If incorrect tags are replaced by a correct description tag, then that would improve the precision of other tags. The improvements are rather small. Since the model is not really trained, we cannot use the usual cross-folded validation to further validate these improvements.

Looking at the results for German with the same set of constraints (see Table 8), the constraints and features do not really generalize well over different languages. The model corrects a lot less errors then for the English model. The types of errors the German experience section seem to be somewhat different then the errors the English experience section has. This means that some (or most) constraints cannot be used as across languages but would have to be developed for each language individually.

Tag	Precision Change (Absolute)	Recall Change (Absolute)	F_{measure} Change (Absolute)
Experience Description	+1.78%	+0.44%	+1.46%
Job Title	+0.16%	+0.13%	+0.14%
Experience Date	+0.05%	+0.00%	+0.03%
Organization Name	+0.01%	+0.10%	+0.04%

Table 7: English model improvements with constraints

Tag	Precision Change (Absolute)	Recall Change (Absolute)	F_{measure} Change (Absolute)
Experience Description	+0.02%	+0.04%	+0.03%
Job Title	+0.01%	+0.01%	+0.01%
Experience Date	+0.0%	+0.0%	+0.0%
Organization Name	+0.0%	+0.0%	+0.0%

Table 8: German model improvements with constraints

Cascading Errors At the start of our tests, we assumed that cascading errors in the segmentation layer had a large effect on the amount of improvements the model could make. We assumed that incorrect item or section segmentation would invalidate some of the constraints and would decrease accuracy. If you look at the improvement results with perfect sections and items (see Table 9), then it seems our assumptions were wrong.

Some parts of the "perfect items" test perform better, other worse, but overall there is not much difference. The main difficulties given by segmentation errors may have been reduces by switching the CCM to relaxed constraints instead of hard constraints.

Tag	Precision Change (Absolute)	Recall Change (Absolute)	F_{measure} Change (Absolute)
Experience Description	+1.38%	+0.32%	+1.11%
Job Title	+0.14%	+0.20%	+0.17%
Experience Date	+0.07%	+0.02%	+0.04%
Organization Name	+0.50%	+0.07%	+0.32%

Table 9: English model improvements with constraints with perfectly tagged sections and items

Evolutionary Search Now we want to test the addition of the evolutionary search to the mix of the algorithms. Our assumption was that the strategy of finding local improvements to a set of starting solutions is better then a depth first search for large instances, but worse for small instances. Therefore we used a mixed strategy with the boundary set at 60 Tokens. In this test we check if this assumption is true. It seems the mixed strategy is overall better then both the complete evolutionary strategy (see Table 10) and the strategy without evolutionary programming (see Table 11), although there is one relatively large regression for the Job Title tag when measuring the addition of the evolutionary search. This validates our initial assumptions. The precise boundary between the two strategies could probably be changed to get further improvements.

Tag	Precision Change (Absolute)	Recall Change (Absolute)	F_{measure} Change (Absolute)
Description	+0.40%	+0.20%	+0.36%
Job Title	+0.03%	+0.07%	+0.05%
Date	+0.05%	+0.0%	+0.03%
Organization	+0.02%	+0.03%	+0.02%

Table 10: Mixed evolutionary strategy vs All evolutionary strategy

Tag	Precision Change (Absolute)	Recall Change (Absolute)	F_{measure} Change (Absolute)
Description	+0.35%	+0.12%	+0.30%
Job Title	-0.03%	+0.01%	-0.01%
Date	+0.0%	+0.0%	+0.0%
Organization	+0.02%	+0.03%	+0.02%

Table 11: Mixed evolutionary strategy vs No evolutionary strategy

Wrong Paths When using hard constraints for the Experience Description tag, the CCM fails to find any solution within the time-out period for 9 of the 312 problem instances. To force the solver in the right direction, we use higher weights for our local features:

- Any description tag to any other description tag is now promoted with weight 4.0
- Any description tag to any tag other then IGN is now penalized with weight -9.0

These weights are necessary to solve the 9 instances without solutions. With these high weights, the resulting solutions are more biased towards Experience Description tags, this results in the performance seen in Table 12. The model does correct a lot more errors, but also decreases the performance for all the other tags.

The high local features bias the solutions towards the Experience Description tags and the hard constraint forces a lot of suboptimal choices to be made. Only the Experience Date has a large increase in the precision, but at the cost of a high decrease in the recall.

Tag	Precision Change (Absolute)	Recall Change (Absolute)	F_{measure} Change (Absolute)
Experience Description	+4.84%	+7.72%	+5.95%
Job Title	-3.37%	-7.62%	-5.36%
Experience Date	+2.54%	-2.86%	+0.11%
Organization Name	-0.32%	-3.81%	-1.71%

Table 12: English model improvements with constraints with hard constraints and extra features

There were a lot of other constraints then the "unique description tag sequence" constraint we have tried. Here are the results of one example where we added the constraint "There should be at most one consecutive organization name tag sequence" with weight 1.3 (see Table 13). This particular constraint did not lead to any real improvements. The performance on the first three tags went slightly down compared to the normal improvements (see Table 7). The constraint did lead to an improvement in the precision of the organization name, but also leads to a decrease in the recall of this tag, which overall is not very impressive. We found similar results for other types of constraints. These include:

- The Job Title should exists before the Description Tag
- There should be at most one consecutive Job Title and Experience Date tag sequence
- There should be at least one job title and organization name tag sequence

Tag	Precision Change (Absolute)	Recall Change (Absolute)	F_{measure} Change (Absolute)
Experience Description	+1.68%	+0.62%	+1.45%
Job Title	+0.30%	+0.12%	+0.22%
Experience Date	+0.005%	+0.000%	+0.003%
Organization Name	+1.31%	-0.14%	0.76%

Table 13: English model improvements with constraints with extra constraint (singular organization name)

The features added to the model were added to eliminate a specific set of flaws:

- Feature: A transition from any kind of description tag to any other tag then a IGN tag penalized with weight -0.9. This feature was added to prevent the description tag from ending prematurely or to prevent the description tag from existing at all
- Feature: A transition from any kind of description tag to any other kind of description tag promoted with weight 0.01. This feature was added to ensure the solver considers continuing the description tag
- Feature: A transition from any kind of Job Title to any other kind of Job title promoted with weight 0.05. This feature was added to prevent the description tag from overwriting the Job Title tag

To test if these local features are useful, we compare it with a model without those features (see Table 14). It is quite clear that without these local features, the constraint fixes less problems

Tag	Precision Change (Absolute)	Recall Change (Absolute)	F_{measure} Change (Absolute)
Experience Description	0.06%	0.03%	0.05%
Job Title	-0.12%	0.23%	0.04%
Experience Date	0.08%	0.09%	0.08%
Organization Name	-0.06%	0.00%	-0.04%

Table 14: English model improvements with constraints without any features

6 Conclusion

Here we answer the research questions posed at the start of the paper:

1. We were only able to realize small improvements on the test dataset. This was after trying a large number of constraints and weights. There are a number of reasons for this:
   - The dataset Textkernel uses is very diverse. This makes it hard to find constraints that are useful (correct common errors) and universally true for all or most of the test samples. Counter examples for these constraints can severely impact the performance because corrections can wipe out a large number of correct and important labellings. Due to my inexperience with this dataset it is also quite possible I was not able to find the most useful constraints. People more experienced with their resume parsing system should be able to find better constraints.
   - The constrained conditional model depends on the results of the hidden markov model. It optimizes the likelihood of the results of the HMM under constraints. It is important that the correct solution that satisfies the constraints in the CCM is marked as somewhat likely by the HMM, otherwise the CCM cannot chose this option because it will be marked with a large negative weight. The sequence tagger at Textkernel is optimized for finding the most likely solution. The heuristics used in the sequence tagger (which cannot be easily turned off) discards choices below a certain probability threshold, these choices are sometimes essential to find a solution for certain constraint violations. We are also not able to get a completely accurate probability estimate for each choice that depends on the last choices made. These probability values were not available from the sequence tagger, incorporating these values into the CCM would make it a more accurate representation of the actual hidden markov model which may improve the accuracy of the CCM
   - Although we were unable to test the difference of our solutions with solutions from an exact ILP solver (an exact ILP solver would take a very long time to solve ILP models of this size), it is quite likely that due to the large instance sizes and the short evaluation time available, the quality of the solutions that we found suffers. It is quite probable that using an exact solver would improve the accuracy of the found solutions by a significant amount
   - There are a lot of parameters that determine the decisions the search algorithms makes. The was very little time available to actually tune these parameters. It is likely that with better parameter settings, the search algorithm would find better quality solutions
2. With a combination of heuristic algorithms it is possible to evaluate the model in a short time. The use of a class of anytime algorithms that find an initial solution fast and can be aborted at any point in time, seems to be a good fit for a time constrained environment
3. Only a small amount of soft constraints and light features are practical. Hard constraints or constraints with a high weight force certain choices to be made. When the constraint is not always true, or the right choice is not indicated as probable by the HMM, bad alternatives can be chosen which means that the weights for both constraints and features have to be low

A Constrained conditional model is a useful technique but it is difficult to apply effectively to the resume data set and sequence tagger at Textkernel. With different constraints, better tuning and a different

dataset, Textkernel might be able to realize more significant improvements. We have shown that a constrained conditional model can be used to improve the performance of the base model, but it might not be worth the investment in manpower and the relative increase in model evaluation time to implement this technique properly.

7 Possible Future Work

Here we indicate several methods that could be used to improve the results we realized:

7.1 Learning Constraint and Feature Weights

There are methods to automatically tune the weights of the constraints and features in the CCM[1][8]. Because this weight learning process would quite a long time, we were unable to attempt this within the time limitations of our project. There also might be some practical concerns about the viability of this approach in our situation.

There are two main approaches that are used to find better weights in a supervised setup:

- Inference Based Training that learns both w and ρ
- Learning + Inference that learns w and ρ independently

Both these approaches can automatically tune the weights by repeatedly solving the model and looking at the difference of the amount of constraint violations and features in the training examples and the output of the solver. The learning algorithms attempt to find a set of weights such the amount of constraint violations and features found in the solutions are about the same as the amount of constraint violations found in the training examples. Inference based training is supposed to be better when the classification problem is hard to solve and the constraints and features are not separable while the Learning+Inference approach is supposed to be better when the features and constraints are separable and the classification problem is easy to learn[8].

The inference based learning algorithm seems to be the best candidate for our situation and is given by:

```
for(i from 1 to #constraints) {
  rho_i = if(hardConstraint(i)) infty else 0
}

for(i from 1 to #iterations, instance from dataset) {
  (tokens, labels) = instance
  label-solution = solve(instance)
  weights = weights + features(tokens, labels) - features(tokens, label-solution)
  for(j from 1 to #constraints) {
    if(!hardConstraint(i)) {
      rho_i = rho_i + dc(i,tokens,labels) - dc(i,tokens,labels-solution)
    }
  }
}
```

These learning algorithms which are recommended by the literature might not be the best approach to finding good weights when there is no way to find a guaranteed solutions for a subset of problem instances. When using the Learning based inference approach, the learning algorithms would probably find weights that are too large to use in practice. Not all problems have a "good" solutions for constraint violations and setting the weights of our constraints and features at a point such that all or most constraint violations would have to be solved, would probably decrease the performance of the CCM.

Given the specific situation at Textkernel, there are two methods to incorporate a weight learning algorithm:

- The algorithms would have to be tweaked to be more "conservative" with the weight assignments and more aimed at optimizing the performance of the model instead of fine-tuning the amount of constraint violations
- The problems with consistently finding good alternatives solutions for constraint violations would have to be solved to incorporate these algorithms unchanged

There is also the possibility to do semi-supervised learning with constrained conditional models, but since there is a large amount of annotated resumes already available, the effect of this approach would be limited.

7.2 Modifying the sequence tagger

The current HMM sequence tagger that is being used at Textkernel can be problematic when a constrained conditional model is used. There are a number of problems with it that restrict the CCM in some ways:

- Currently the sequence tagger cannot give the probability estimate depending on the previous taggings as output. Being able to incorporate those probabilities would probably give a more accurate model to optimize
- The current sequence tagger uses a number of heuristics that cannot be completely turned of. Those heuristics result in taggings with very small probabilities being assigned the probability 0

Changing the sequence tagger to correct these problems would probably improve the performance of the CCM in a number of ways.

7.3 Automatic Parameter Tuning for Evolutionary Algorithms

Reasonable parameters for the evolutionary algorithms are usually picked by hand. While this mostly results in acceptable improvements, automatic tuning algorithms can be used to find close to optimal parameters that result in better performance. Tuning the evolutionary algorithm would take a lot CPU time, but it would improve the speed at which it will find better solutions. An overview of tuning evolutionary algorithms can be found in[2].

References

[1] M. Chang, L. Ratinov, and D. Roth. Constraints as prior knowledge. In *ICML Workshop on Prior Knowledge for Text and Language Processing*, pages 32–39, 2008.

[2] A. E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, to appear, 2011.

[3] A.E. Eiben and J.E. Smith. *Introduction to evolutionary computing*. Springer Verlag, 2003.

[4] L. Getoor and B. Taskar. *Introduction to statistical relational learning*. The MIT Press, 2007.

[5] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 607–615. Citeseer, 1995.

[6] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, pages 282–289. Citeseer, 2001.

[7] E.L. Lawler and D.E. Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[8] V. Punyakanok, D. Roth, W. Yih, and D. Zimak. Learning and inference over constrained output. In *International Joint Conference on Artificial Intelligence*, volume 19, page 1124. Citeseer, 2005.

[9] D. Roth and W. Yih. Integer linear programming inference for conditional random fields. In *Proceedings of the 22nd international conference on Machine learning*, pages 736–743. ACM, 2005.

[10] M. Stamp. A revealing introduction to hidden Markov models. *Department of Computer Science San Jose State University*, 2004.

[11] H.M. Wallach. Conditional random fields: An introduction. *Rapport technique MS-CIS-04-21, Department of Computer and Information Science, University of Pennsylvania*, 50, 2004.