Machine learning in NLP
Lecture 6: Predicting structured objects

UNIVERSITY OF GOTHENBURG

Richard Johansson

September 22, 2012

This lecture

- Instead of simple labels such as text categories, we'll predict complex objects such as sequences, trees, or translations
- We'll stick to the same basic framework

$$\text{guess} = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y)$$

- The "same" learning algorithms can be used
- The critical things: $\boldsymbol{f}(x, y)$ and arg max
  - Features
  - Searching

Again: some preliminaries

- **Inputs**, $x \in \mathcal{X}$: the things that we want to classify
  - e.g. x is a document, a word in context, an image, a query, ...
- **Outputs**, $y \in \mathcal{Y}$: the categories to which the x:s are classified
  - e.g. y is sentiment or topic label for a document, a word sense tag for a word in context, ...
- Previously: $\mathcal{Y}$ is a small set e.g. { positive, negative } or { line.1, ..., line.6 }
- This lecture: $\mathcal{Y}$ is a very large set e.g. the set of possible parse trees of a sentence, or the set of translations of a sentence
- Supervised learning: the training set consists of input–output pairs: $\mathcal{T} = \{(x_1, y_1), \ldots, (x_T, y_T)\}$
  - so in this lecture, the x_i are e.g. sentences and the y_i are e.g. parse trees

Predicting structured objects

- In many NLP tasks, the output is not just a class
- For a given input x, the set of legal outputs
  - is very large – typically exponential in the size of x
  - depends on x
  - consists of many small but **interdependent** parts

Example: POS tag sequences

Will plays golf	NNP VBZ NN NNP VBZ VB NNP NNS NN NNP NNS VB NN VBZ NN NN VBZ VB NN NNS NN NN NNS VB MD VBZ NN MD VBZ VB MD NNS NN MD NNS VB

Example: dependency parse trees

s = <D> Lisa walks home

$t^1 =$

$t^2 =$

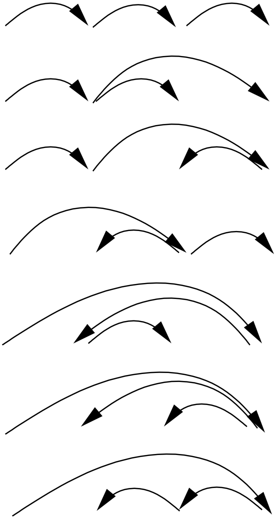
$t^3 =$

$t^4 =$

$t^5 =$

$t^6 =$

$t^7 =$



Example: translations

Kas sul kõht on tühi?

Is the stomach empty on you?
Do you have an empty stomach?
Are you starved?
. . .

recap: multiclass classification

- in the previous lecture, we discussed two basic ideas for training multiclass classifiers
  - break down the complex problem into simpler problems, train a classifier for each
  - make a more advanced model that tries to handle the complex problem directly
- we can apply the same ideas when predicting complex objects
- we start with the second idea and return to the first in the end

recap: multiclass linear classifiers

- the feature function $\boldsymbol{f}(x, y)$ creates a feature vector the represent the instance x and its class y

```
[ { 'label':'NP', ... }, SBJ          ]      [ 1 0 0 0 ... ]
[ { 'label':'PP', ... }, ADV          ]      [ 0 1 0 0 ... ]
[ { 'label':'S', ... }, (empty)       ]      [ 0 0 1 0 ... ]
[             ...                     ]  →   [       ...   ]
[             ...                     ]      [       ...   ]
[             ...                     ]      [       ...   ]
[ { 'label':'PP', ... }, TMP          ]      [ 0 0 0 1 ... ]
```

- then a one-vs-rest classifier can be written like this:

$$\text{guess} = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y)$$

recap: multiclass perceptron

$\boldsymbol{w} = (0, \ldots, 0)$
repeat N times
  for (x_i, y_i) in $\mathcal{T}$
    $g = \arg\max_y \boldsymbol{w} \cdot \boldsymbol{f}(x_i, y)$
    if g is not equal to y_i
      $\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{f}(x_i, y_i) - \boldsymbol{f}(x_i, g)$
return w

Adapting our algorithms

- The idea: adapt the algorithms we have seen:
  - Perceptron
  - SVM
  - Logistic regression
- Our algorithms still produce a $\boldsymbol{w}$
- But how to implement in practice?

Perceptron

- when training a perceptron, we make predictions and update the weights when our predictions are wrong
- how do we carry out the arg max line if there are 1,000,000 possible outputs?

$\boldsymbol{w} = (0, \ldots, 0)$
repeat N times
  for (x_i, y_i) in $\mathcal{T}$
    $g = \arg\max_y \boldsymbol{w} \cdot \boldsymbol{f}(x_i, y)$
    $\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{f}(x_i, y_i) - \boldsymbol{f}(x_i, g)$
return w

In logistic regression, we estimate $\boldsymbol{w}$ by maxizing a likelihood:

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} L(\boldsymbol{w}) = \arg\max_{\boldsymbol{w}} P(y_1|x_1) \cdot \ldots \cdot P(y_T|x_T)$$

where

$$P(y_k|x) = \frac{e^{score_{y_k}}}{e^{score_{y_1}} + \ldots + e^{score_{y_n}}} = \frac{e^{\boldsymbol{w}\cdot\boldsymbol{f}(x,y_k)}}{e^{\boldsymbol{w}\cdot\boldsymbol{f}(x,y_1)} + \ldots + e^{\boldsymbol{w}\cdot\boldsymbol{f}(x,y_n)}}$$

We don't want to sum over all possible outputs!

Prediction of complex objects

- The solution: break down the object into simple **parts**
  - There's an "infinite" set of outputs, but a finite set of parts
  - $\sim$ 50 possible POS tags, $\sim$ 2500 POS bigrams
  - $(n+1) \cdot n$ possible dependency links in a sentence of length n
- Apply a feature function to the parts independently
- Use some **problem-specific** method to find the best selection of parts, i.e. solving $\arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x_i, y)$
  - Sequence tagging: the Viterbi algorithm
  - Dependency parsing: maximum spanning tree algorithms
- (Jargon: the decomposition into parts is called a **factorization**)

Applying the scoring function

$$y = (y^1, \ldots, y^k)$$

$$\boldsymbol{w} \cdot \boldsymbol{f}(x, y) = \sum_{y^k} \boldsymbol{w} \cdot \boldsymbol{f}_{part}(x, y^k)$$

Case study: sequence tagging

<B>	Will	plays	golf	<E>
<B>	NNP	VBZ	NN	<E>
			↑	

- When we predict a tag, such as the POS tag for *golf*, our decision depends on the previous tags
- To make this problem solvable, we introduce a practical assumption: that it depends on **the previous tag only**
- This is called the *Markov* assumption
- Our parts are *tag bigrams*:

$$\mathbf{w} \cdot \mathbf{f}(x, y) = \sum_{y^k} \mathbf{w} \cdot \mathbf{f}_{part}(x, y^k)$$

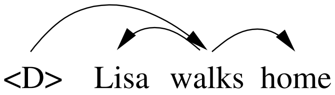
$$\mathbf{w} \cdot \mathbf{f}([\text{<B>,Will,plays,golf,<E>}], [\text{<B>,NNP,VBZ,NN,<E>}]) =$$
$$= \mathbf{w} \cdot \mathbf{f}_{part}(x, [\text{<B>,NNP}]) + \ldots + \mathbf{w} \cdot \mathbf{f}_{part}(x, [\text{NN,<E>}])$$

More sequence tagging

<B>	Prices	fell	in	New	York	.	<E>
<B>	C	NC	NC	C	C	.	<E>
<B>	O	O	O	B-LOC	I-LOC	O	<E>
					↑		

- The feature function f_{part} can use the previous tag and any information from the input
- See Collins, *Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms*, EMNLP 2002.

Case study: dependency parsing

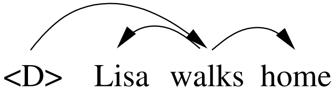


- In dependency parsing, we may use the dependency edges as the parts:

$$\begin{aligned} \boldsymbol{w} \cdot \boldsymbol{f}(x, y) &= \textstyle\sum_{y^k} \boldsymbol{w} \cdot \boldsymbol{f}_{part}(x, y^k) = \\ &= \boldsymbol{w} \cdot \boldsymbol{f}_{part}(x, \text{<D>}\rightarrow\text{walks}) + \\ &+ \boldsymbol{w} \cdot \boldsymbol{f}_{part}(x, \text{walks}\rightarrow\text{Lisa}) + \\ &+ \boldsymbol{w} \cdot \boldsymbol{f}_{part}(x, \text{walks}\rightarrow\text{home}) \end{aligned}$$

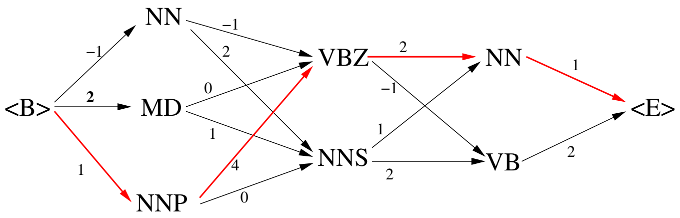
- See McDonald, Crammer and Pereira, *Online Large-Margin Training of Dependency Parsers*, ACL 2005.

Dependency parsing features



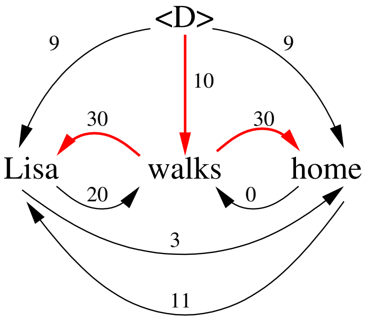
- head = "walks"
- dependent = "Lisa"
- head POS = "VBZ"
- dependent POS = "NNP"
- head+dependent POS pair = "VBZ+NNP"
- ...
- See McDonald, Crammer and Pereira, *Online Large-Margin Training of Dependency Parsers*, ACL 2005.

Finding the best sequence of tags



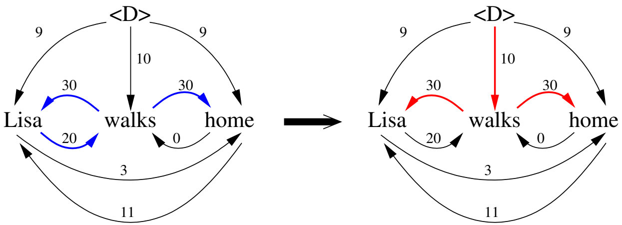
- We can visualize the search space as a graph
- The scores in the graph are given by $\boldsymbol{w} \cdot \boldsymbol{f}_{part}(x, \text{bigram})$
- Best tag sequence: the path from start to end that gives the highest sum
- Use the **Viterbi** algorithm
- Note the difference between Viterbi and greedy search

Finding the best tree



- The scores in the graph are given by $\boldsymbol{w} \cdot \boldsymbol{f}_{part}(x, \text{edge})$
- Our task: find the set of edges that
  - gives the highest sum of edge scores
  - includes all words
  - contains no cycles
- This is called the **maximum spanning tree**

Finding the best tree



- The **Chu–Liu/Edmonds** algorithm:
  1. For each node, find the top-scoring incoming edge
  2. If there are no cycles, we are done
  3. If there is a cycle, create a single node containing the cycle
  4. Find the MST in the new graph (recursion)
  5. Break the cycle...
- Also: the **Eisner** algorithm (see McDonald paper)

Perceptron

- The perceptron pseudocode looks exactly the same when we are predicting complex outputs!
- Of course, the arg max is implemented differently...

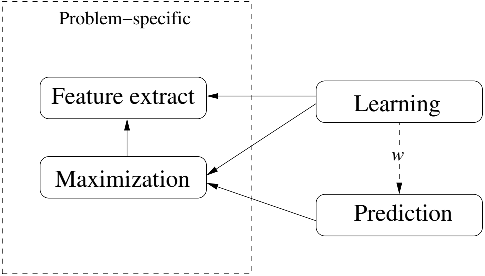
The perceptron learning algorithm:

$\boldsymbol{w} = (0, \ldots, 0)$
repeat N times
  for (x_i, y_i) in $\mathcal{T}$
    $g = \arg\max_y \boldsymbol{w} \cdot \boldsymbol{f}(x_i, y)$
    $\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{f}(x_i, y_i) - \boldsymbol{f}(x_i, g)$
return w

At prediction time: guess $= \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y)$

Loosely coupled software design

- The perceptron makes it easy to separate the problem-specific parts and the learning part



- more about this in the third assignment

What about logistic regression?

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} L(\boldsymbol{w}) = \arg\max_{\boldsymbol{w}} P(y_1|x_1) \cdot \ldots \cdot P(y_T|x_T)$$

where

$$P(y_k|x) = \frac{e^{score_{y_k}}}{e^{score_{y_1}} + \ldots + e^{score_{y_n}}} = \frac{e^{\boldsymbol{w}\cdot\boldsymbol{f}(x,y_k)}}{e^{\boldsymbol{w}\cdot\boldsymbol{f}(x,y_1)} + \ldots + e^{\boldsymbol{w}\cdot\boldsymbol{f}(x,y_n)}}$$

We rewrite a bit:

$$\log L(\boldsymbol{w}) = \sum_t \boldsymbol{w} \cdot \boldsymbol{f}(x_t, y_t) - Z$$

The term Z is called the **partition function** and is big and ugly:

$$Z = \sum_t \log \sum_i e^{\boldsymbol{w}\cdot\boldsymbol{f}(x_t,y_{ti})}$$

Conditional random fields

$$\log L(\boldsymbol{w}) = \sum_t \boldsymbol{w} \cdot \boldsymbol{f}(x_t, y_t) - \mathbf{Z}$$

$$Z = \sum_t \log \sum_i e^{\boldsymbol{w} \cdot \boldsymbol{f}(x_t, y_{ti})}$$

- LR for structured objects is called **conditional random fields** (CRFs)
- Implementation is more messy than for perceptrons: in addition to the maximization method, you must also compute Z and its derivative
- When computing Z, use the decomposition into parts
- At test time: as usual $\arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y)$

Sequence tagging with CRFs

<B>	Prices	fell	in	New	York	.	<E>
<B>	C	NC	NC	C	C	.	<E>
<B>	O	O	O	B-LOC	I-LOC	O	<E>
					↑		

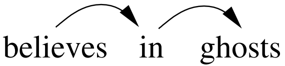
- For sequence tagging with the Markov assumption, Z can be computed efficiently
- CRF is probably the most popular learning method for this task
- Many implementations:
  - Mallet – a Java library that can be called from NLTK
  - CRF++
  - CRF-suite
  - CRF-SGD – the fastest
- Lafferty, McCallum, Pereira: *Conditional random fields: Probabilistic models for segmenting and labeling sequence data*. ICML 2001.

what about SVMs?

- similar ideas can be used to adapt the SVM
- multiclass Pegasos can be applied without change
- see also the paper by Tsochantaridis, Joachims, Hofmann, and Altun, *Large Margin Methods for Structured and Interdependent Output Variables*. JMLR, 2005.
- ...and their software at `http://svmlight.joachims.org/svm_struct.html`

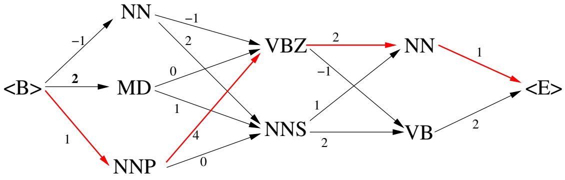
Using more complex parts

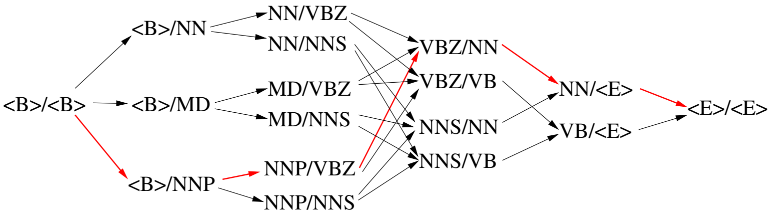
- Sometimes our parts are too restricted
- We may define more complex parts:
  - In tagging, let $\boldsymbol{f}_{part}$ use tag **trigrams**: NNP-VBZ-NN
  - In dependency parsing, let $\boldsymbol{f}_{part}$ use mini-trees larger than single links:



- **If we make f_{part} more complex, we also make the search more complex!**

Search spaces...





some Python libraries

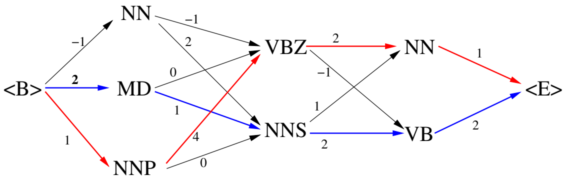
- PyStruct: `https://pystruct.github.io`
  - contains a number of learning algorithms as well as optimization tools to help implementing the $\arg\max$
  - designed to be compatible with scikit-learn
  - unfortunately, can't yet handle sparse feature vectors...
- seqlearn: `http://larsmans.github.io/seqlearn`
  - implemented by one of the designers of scikit-learn
  - only sequence tagging

again: two basic approaches

- how to predict a complex object?
  - break down the complex problem into simpler problems, train a classifier for each
  - make a more advanced model that tries to handle the complex problem directly
- we have now discussed the second of them
- how about the first?

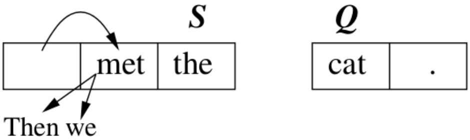
Greedy search

- use a **greedy** method: apply a classifier at each step



- Pros:
  - Easier to implement
  - Faster
  - No restriction on features
- Cons:
  - Less accurate (sometimes)
- Ratinov and Roth: *Design challenges and misconceptions in named entity recognition*. CoNLL 2009.
- Liang, Daumé III and Klein: *Structure compilation: trading structure for features*. ICML 2008.

- can we do something similar for a dependency parser?
- recall the Nivre parser from the StatNLP course
- this parser works in a left-to-right fashion
- at each step, we make a decision by using a classifier



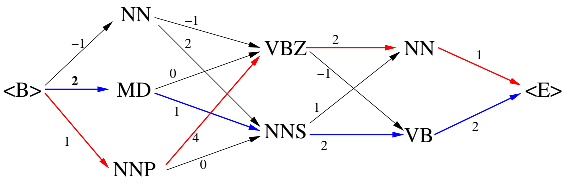
- this parser is much faster than graph-searching parsers:
  - Nivre (greedy left-to-right): linear time
  - McDonald (graph search): quadratic or cubic time

example: parser comparison

Sammanställning parsrar

parser	korrekthet	länkkorrekthet ▲	tid/mening	kommentar
LTH	82.43	88.58	0.193	2-ordning, pseudoprojektiv, Brown-kluster
Mate-tools	81.65	87.93	0.141	ickeprojektiv, Brown-kluster
TurboParser	79.91	87.31	0.053	
ZPar	80.78	87.26	0.190	projektiv
MSTParser	78.14	86.32	0.119	2-ordning, ickeprojektiv
MaltParser	78.42	85.17	0.005	ickeprojektiv, Brown-kluster, tränad enligt instruktioner av Johan Hall
Huang	-	84.74	0.017	projektiv, inga funktioner

Beam search

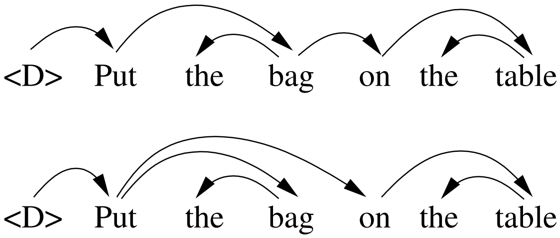


- Greedy search may be too imprecise
- A compromise between greedy and exact search: **beam search**
- At each step, remember the k best analyses
- Has been used to improve the performance of Nivre-like dependency parsers:
  - Johansson and Nugues: *Investigating multilingual dependency parsing*, CoNLL 2006.
  - Zhang and Nivre: *Transition-based dependency parsing with rich non-local features*, NAACL 2011.

Reranking

- Another compromise between greedy and exact: **reranking**
- First, use a "simple" system:
  - PCFG or edge-factored parser
  - Word-based machine translator
- Let the simple system generate the k top-scoring analyses
- In a second step, **rerank** the set: use a more careful scoring function
- The reranker can use almost any feature
- Charniak and Johnson: *Coarse-to-fine n-best parsing and MaxEnt discriminative reranking*. ACL 2005.

Reranking example



Summary

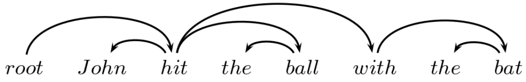
- We have seen how to apply classification methods when the things we want to predict are complex
- The general idea is similar:

$$\text{guess} = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y)$$

- We have adapted the learning algorithms: perceptron, LR$\rightarrow$CRF
- The critical things: $\boldsymbol{f}(x, y)$ and $\arg\max$
  - Features: the feature function operates on the **parts**
  - Searching
    - Search space complexity depends on the parts
    - Tailored search procedure for our problem: Viterbi, MST, ...
    - Cheating: greedy, beam, reranking

assignment 3

- implement the structured perceptron learning algorithm
- use it to train a dependency parser



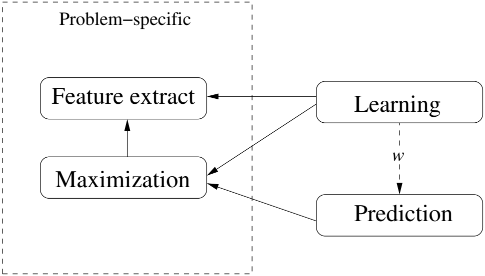
- . . . and a named entity recognizer

  United Nations official Ekeus heads for Baghdad.
  [ORG] [PER] [LOC]

- instructions will be up in a few days; you can prepare by reading McDonald's paper

recap: modularization

- in the assignment, the problem-specific parts (left box) will be provided
  - the Eisner algorithm for dependency parsing
  - the Viterbi algorithm for tagging
  - (and code for reading data, etc.)



- you implement the rest!