Kapitola 24

Státnice I3: Automatická analýza jazyka

24.1 Morphology & Tagging

- Task, formally: $A^+ \to T$ (simplified), split morphology & tagging (disambiguation): $A^+ \to 2^{(L,C_1,C_2,\ldots,C_n)} \to T$. Tagging must look at context.
- Tagset: influenced by linguistics as well as technical decisions.
  - Tag ˜ n-tuple of grammar information, often thought of as a flat list.
  - Eng. tagsets ˜ 45 (PTB), 87 (Brown), presentation: short names.
  - Other: bigger tagsets – only positional tags, size: up to 10k.
- Tagging inside morphology: first, find the right tag, then the morphological categories: $A^+ \to T \to (L, C_1, \ldots, C_n)$.
  - Doable for poor flection languages only.
  - Possibly only decrease the ambiguity for the purposes of tagging (i.e. morphology doesn't have to be so precise).
- Lemmatization: normally a part of morphology, sometimes (for searching) done separately.
  - Stem – simple code for Eng., no need of a dictionary, now out-dated.
- Possible methods for morphology:
  - Word lists: lists of possible tags for each word form in a language
    - Works well for Eng. (avg. ca. 2.5 tags/word), not so good for Cze. (avg. ca. 12-15 tags/word).
  - Direct coding: splitting into morphemes (problem: split and find possible combinations)
  - Finite State Machinery (FSM)
  - CFG, DATR, Unification: better for generation than analysis

Finite State Machinery

Finite-State-Automata

- Smarter word form lists: compression of a long word list into a compact automaton.
  - Trie + Grammar information, minimize the automaton (automaton reduction)
  - Need to minimize the automaton & not overgenerate

Two-Level-Morphology

- phonology + morphotactics, two-level rules, converted to FSA
- solves e.g. Eng. doubling (stop**p**ing), Ger. umlaut, Cze. "ský" –>pl. "čtí" etc.
- **Finite State Transducer**: an automaton, where the input symbols are tuples
  - run modes: check (for a sequence of pairs, gives out Y/N) + analysis (computes the "resolved" (upper) member of the pair for a sequence of "surface" symbols) + synthesis (the other way round)
  - used mostly for analysis
  - ususaly, one writes small independent rules (watch out for collisions), one FST for one rule – they're run in parallel and all must hold

- zero-symbols, one side only, check for max. count for a language
- we may eliminate zero-symbols using ordinary FSA on lexicon entries (upper layer alphabet; prefixes: according to them, the possible endings are treated specially)

- FSTs + FSAs may be combined, concatenated; the result is always an FST

Two-level morphology: analysis

1. initialize paths $P := \{\}$
2. read input surface symbols, one at a time, repeat (this loop consumes one char of the input):
   (a) at each symbol, until max. consecutive zeroes for given language reached, repeat (this loop just adds one possible zero):
      i. generate all lexical (upper-level) symbols possibly corresponding to zero (+all possible zeroes on surface)
      ii. prolong all paths in P by all such possible $(x : 0)$ pairs (big expansion)
      iii. check each new path extension against the phonological FST and lexical FSA (lexical symbols only), delete impossible path prefixes.
   (b) generate all possible lexical symbols (get from all FSTs) for the current input symbol, create pairs
   (c) extend all paths with the pairs
   (d) check all paths (next step in FST/FSA) and delete the impossible ones
3. collect lexical "glosses" from all surviving paths

Rule-Based Tagging

- Rules using: word forms/tags/tag sets for context and current position, combinations
  - If-then / regular expression style (blocking negative)
  - Implementation: FSA, for all rules – intersection
  - algorithm ˜ similar to Viterbi (dynamic programming: if the FSA rejects a path, throw it away)
  - May even work, sometimes does not disambiguate enough
- Tagging by parsing: write rules for syntactic analysis and get morphological analysis as a by-product
  - in a way, this is the linguistically correct approach
  - better, cleaner rules
  - more difficult than tagging itself, nobody has ever done it right

HMM Tagging

- probabilistic methods, also applies to feature-based
- noisy channel: input (tags) -> output (words), goal: discover channel input given the output.
  - $p(T|W) = p(W|T) \cdot \frac{p(T)}{p(W)}$, $\arg\max_T p(T|W) = \arg\max_T p(W|T)p(T)$.
- two models – simplification:
  - tags depend on limited history (4-5 grams)
  - word depends on tag only (1 gram!)
- almost the general HMM
  - output words emitted by states (not arcs), states are $(n-1)$-tuples of tags for an n-gram tag model
  - (S, s_0, Y, P_S, P_Y) – set of states, initial state, output alphabet (words), set of prob. distributions of transitions, set of prob. distributions for emissions
- **supervised learning**: use MLE, smoothing:
  - p(w|t) – "add 1" for all possible tag+word pairs using a predefined dictionary (i.e. some 0's kept: p(word|impossible-tag) = 0)
  - p(t|context) – linear interpolation, up to uniform (as for language model)
- old and simple, but still accurate enough (only slower than e.g. neuron networks)

- may be trained even with **unsupervised** data: unambiguous words help get the disambiguation for the others (improvement depends on language and tagset)
  - Baum-Welch algorithm, minimizing the entropy; use heldout data
  - training always decreases the entropy, smoothing increases it again (in case of no bigger tagged corpus available, it's a good step to try; supervised is always better)
- **Out-of-Vocabulary**
  - no lists of possible tags
  - try all / open class tags (good for non-flective languages), or:
  - try to guess possible tags based on suffix/ending or both ends of the word (e.g. for Cze. – first 3 and last 8 letters) – train the classifier using rare words from the training data only!
- **Running the tagger**
  - Viterbi, remember to handle unknown words, or:
  - assign always the best tag at each word, but consider all possibilities for previous tags; introduces some errors, but might get better accuracy

Transformation-Based Tagging

- Not noisy channel, not probabilistic, but statistical – uses training data (combination of supervised/unsupervised), learning rules of type context + tag_1 : $\text{tag}_1 \rightarrow \text{tag}_2$
- criterion: accuracy – "objective function"
- **training**: stepwise greedy-select
  - iterate: pre-anotate using current rules (intermediate data), select the rule from the pool of possible ones (from templates) that contributes best to the improvement of the error rate
  - stopping criterion: no or too small improvement possible; prone to overtraining!
  - heldout possible (afterwards, remove rules that degrade performance on heldout data)
- rule types: context, lexical (looks at parts of the word)
  - application of the rules – left-to-right (a rule may be applied on part of its output) / delayed
- improved version: Fast-TBL(Transformation-Based Learning), there's no parallelized version
- old method (90's – was the best one), faster than HMM
  - tested for Cze. in the late 90's, not very good results, too many rules – uncomputable (no way to parallelize it, the rules are weird in the beggining)
  - may be used e.g. for named entity recognition (less rules, more effective)
- **tagger**: input = untagged data + rules from the learner
  - applies the rules one-by-one to all the data –> creates n iterations of intermediate data, the last one of which is the output
- n-best modification (criterion: accuracy + number of tags per word), unsupervised modification (use only unambiguous words for evaluation)

That's more than sneisble! That's a great post!
6IE1X3 cxlcxqqmlidk

Tagger Evaluation

- A must: **Test data (S)**, previously unseen, change frequently if possible
- Formally: Out(w) / True(w) – for a given word
  - Errors(S) = $\sum_{i=1}^{|S|} \delta(\text{Out}(w_i) \neq \text{True}(w_i))$
  - Correct(S) = $\sum_{i=1}^{|S|} \delta(\text{True}(w_i) \in \text{Out}(w_i))$
  - Generated(S) = $\sum_{i=1}^{|S|} |\text{Out}(w_i)|$ – how many outputs the tagger produced (sum over all data)

Metrics

- Error rate: Err(S) = Errors(S) / |S|
- Accuracy: Acc(S) = 1 — Err(S)
- Multiple / no output:
  - Recall: R(S) = Correct(S) / |S| – must select the right one (possibly among others)
  - Precision: P(S) = Correct(S) / Generated(S) – against too much noise
  - no way to improve P + R at the same time, but also no way to tell what's better (depends on the application: Google – P, Medical test – R)
    * systems with a big difference in P/R are (empirically) worse
  - F-Measure: $F = \frac{1}{\frac{\alpha}{P} + \frac{1-\alpha}{R}}$, usually $F = \frac{2PR}{R+P}$ (for $\alpha = 0.5$)

24.2 Parsing

Chomsky Hierarchy

1. Type 0 Grammars/Languages – $\alpha \to \beta$, where α, β are any strings of nonterminals
2. Context-Sensitive Grammars/Languages – $\alpha X \beta \to \alpha \gamma \beta$, where X is a nonterminal, α, β, γ any string of terminals and nonterminals (and γ must not be empty)
3. Context-Free Grammars/Languages – $X \to \gamma$, where X is a nonterminal and γ is any string of terminals and nonterminals
4. Regular Grammmars/Languages – $X \to \alpha Y$, where X, Y are nonterminals and α a string of terminals, Y might be missing

Chomsky's Normal Form

- for CFG's – each CFG convertible to normal form
- rules only $A \to BC$ (two nonterminals), $A \to \gamma$ (one terminal), $S \to \varepsilon$ (empty string)

Parsing grammars

- Regular Grammars – FSA, constant space, linear time
- CFG – widely used for surface syntax description of natural languages, needed: stack space, $O(n^3)$ time – CKY/CYK algorithm

Shift-Reduce CFG Parsing

- CFG with no empty rules ($N \to \varepsilon$) – any CFG may be converted; recursion is OK
- Bottom-up, construction of a push-down automaton (non-deterministic); delay rule acceptance until all of it is parsed
- Asymptotically slower than CKY, but fast for usual grammars
- Builds upon a **state parsing table** ˜ graph, edges = transitions (defined by one terminal or nonterminal symbol)
  - each state: a special function telling if we output the rule number (even more rules – ambiguity) – this is what separates a shift-reduce parser from an FSA
- lex/yacc / flex/bison – shift-reduce parser generators

Table construction – dot mechanism

- dots = remember where we are in all the rules which possibly could go through this set, used only for table construction

1. take the starting rule and add it to the 1st state (put all rules with the starting symbol on left-hand side into the 1st state, mark the dot at the beginning of the right-hand side of all of them)
2. state expansion: for all nonterminals right after the dot in any rule in this state, add the rewriting rules (in which the given nonterminal is on the left-hand side) into this state (and do this recursively until there are no more nonterminals that have not been expanded)

3. construction of following states: for each terminal / nonterminal after the dot, create a new state (if there are several rules for the same symbol, create only one state over the transition for this symbol)

4. into the new state: add all the rules with the transition symbol just after the dot and move the dot after it + perform state expansion

5. reduction states: if there is a rule with the dot at the end in the state, this state is a so-called reduction-state – in this state, the rule that caused the possibility of moving forward shall be printed (such rule has no expansions –> this leads to finish)

6. merge identical states: if the created state has the same rules (with dots at the same positions) as another state, merge the two (otherwise this would never finish for a recursive grammar; merge only after the whole state has been created!)

- problems:
  - shift-reduce conflict – a state is ambiguous (there is a rule with the dot at the end + some rules with dots in the middle ˜ state may be reductional, but doesn't have to) – this leads to backtracking, ambiguous parses
  - reduce-reduce conflict – another kind of ambiguity (more different rules with dot at the end)
  - the ambiguity does not occur for special kind of grammars – $LR(n)$: for bottom-up parsing, we only need to look n symbols ahead to prevent backtracking, $LR(0)$ never get a conflict in a table
    * there's no simple algorithm for obtaining the n for which a given grammar is $LR(n)$, but we may try for all n's
  - the algorithm complexity copies the complexity of the grammar – it's only expensive at the points of ambiguity

Parsing

- using parsing stack for states and backtrack stack for whole parser configurations at the points of ambiguity
  - backtracking may be implemented in such a way that only the position in the parsing stack and the input need to be stored on the backtrack stack

1. we have an empty backtrack stack, the 1st state on the parsing stack and the original string at the input

2. from a shift state, follow the transition using the input symbol:
   (a) if there is no such transition and there's nothing on the backtrack stack, FAIL; otherwise take something out of the backtrack stack – keep the stack saved if there still are more possibilities to follow!)
   (b) if we find the transition, eat one symbol from the input, follow it to the new state and put the state on the parsing stack

3. if we are in a reduce state:
   (a) remove as many elements from the parsing stack as there are on the right-hand side of the rule we're reducing over
   (b) put the nonterminal from the left-hand side of the rule on the input

4. for conflicts: follow the first path + save the current configuration to the backtrack stack

5. PASS condition: empty parsing stack and end of input (possibly continue looking for some other parses if there's something on the backtrack stack)

Probabilistic CFG

- relations among mother & daughter nodes in terms of derivation probability
- probability of a parse tree: $p(T) = \prod_{i=1}^{n} p(r(i))$, where $p(r(i))$ is a probability of a rule used to generate the sentence of which the tree T is a parse
  - probability of a string is not as trivial, as there may be many trees resulting from parsing the string: $p(W) = \sum_{j=1}^{n} p(T_j)$, where T_j's are all possible parses of the string W.
- assumptions (very strong!):
  - independence of context (neighboring subtrees)
  - independence of ancestors (upper levels)

  - place independence (regardless where the rule appears in the tree) ~ similar to time invariance in HMM

- probability of a rule – distribution $r(i) : A \to \alpha$ ~ $0 \leq p(r) \leq 1$; $\sum_{r \in \{A \to \dots\}} p(r) = 1$
  - may be estimated by MLE from a treebank following a CFG grammar: counting rules & counting nonterminals

- inside probability: $\beta_N(p,q) = p(N \to^\star \; w_{pq})$ (probability that the nonterminal N generates the part of the sentence $w_p \dots w_q$)
  - for CFG in Normal Form may be computed recursively: $\beta_N(p,q) = \sum_{A,B} \sum_{d=p}^{q-1} p(N \to A\ B) \beta_A(p,d) \beta_B(d+1,q)$

Computing string probability – Chart Parsing (CYK algorithm)

- create a table $n \times n$, where n is the length of the string
- initialize on the diagonal, using $N \to \alpha$ rules (tuples: nonterminal + probability), compute along the diagonal towards the upper right corner
  - fill the cell with a nonterminal + probability that the given part of the string, i.e. row = from, col = to, is generated by the particular rule
  - consider more probabilities that lead to the same results & sum them (here: for obtaining the probability of a string, not parsing)
- for parsing: need to store what was the best (most probable) tree – everything is computable from the table, but it's slow: usually, you want a list of cells / rules that produced this one
  - if the CFG is in Chomsky Normal Form, the reverse computation is much more effective

External links

- `http://ufal.mff.cuni.cz/~novak/presentPraha/` — slides in Czech
- `http://nlp.stanford.edu/fsnlp/pcfg/fsnlp-pcfg-slides.pdf` — slides in English

Statistical Parsing

- parsing model: $p(s|W) = \frac{p(W,s)}{p(W)} = \frac{p(s)}{p(W)}$ since $p(W,s) = p(s)$ (where s is a parse and W the corresponding string – a parse defines a sentence!)
  - therefore: $\text{argmax}_s p(s|W) = \text{argmax}_s p(s)$ – we just select the most probable parse
  - similar to language model; we don't consider trees, but all the possible parses

Parser Creation

1. extract (all used) rules from a treebank
2. convert the grammar to the normal form
3. apply this back to the treebank (keep track of which rules were affected by the conversion)
4. get the counts of the rules –> probability
5. smoothen

Smoothing

- the extracted rules cover an infinite number of sentences, but certainly not the whole language
- add poor (missing) rules – get all possible combinations on the right side
  - but ensure their probability is really very low!
- there are many ways of smoothing
  - e.g. tie several probabilities to one: $B \to N\ V$ –> split to first and second nonterminal: $p(B \to N\ \star)$, then: $p(B \to N\ V) = p(B \to N\ \star) \cdot p(V|N)$ (only V following N on the right-hand side of any rule, regardless of the left-hand side)
  - or, use some linear combination of similar tied probabilities

  - may be combined with the original ones before smoothing
- the smoothing is often tuned on data, the best way is selected according to the performance
- if we don't use Chomsky's Normal Form, we may have much more sofisticated ways of smoothing, perhaps even reflecting the linguistic properties of the language, but it'll slow down the process

Lexicalization

- obtain more distinct nonterminals: use lexicalized parse tree (˜ dependency tree + phrase labels; no lexicalization needed for dep. trees)
  - process of filling in words that were originally only on the terminal nodes – so that the word is taken from the head of the phrase

1. pre-terminals (right above the leaves): assign the word below
2. recursive step (up one level – bottom-up): select one node and copy it up (the "more important one", eg. the preposition for PP, the noun for NP; there are no clean rules, it's a linguistic problem)

- increases the number of rules (up to 100k), but helps – the smoothing must be very precise (e.g. using the non-lexicalized distribution)
  - particular words are important for the parsing of the sentence –> CFG development paradigm
- it's possible to use POS-tags with the words, or POS-tags only
- conditional probabilities: there are too many rules, the data are sparse –> we need to simplify – assumptions:
  - total independence ($p(\alpha B(head_B)\gamma \ldots | A(head_A)) = p(\alpha | A(head_A)) \cdot p(B(head_B)|A(head_A)) \ldots$) is too strong, too inaccurate
  - best known heuristics – decomposition: split the right side of the rule into head + left-of-head + right-of-head
    * technical terminal STOP at both sides of the head **(?)**
    * $p_H(H(head_A)|A(head_A))$, $p_L(L_i(l_i)|A(head_A), H)$, $p_R(R_i(r_i)|A(head_A), H)$
  - more conditioning – distance: absolute is non-zero ? path goes over verb ? over commas ?
  - other: complement/adjunct, subcategorization **(?)**

Remarks

- parsing is still not solved properly, the results are not sufficient

Dependency parsing

- until 2005, done via phrase-tree parsing, the trees were then converted
- McDonald's Parser
- result: a tree – each word has its parent (or is root)
- initialize: make a total graph, where each edge is rated with a probability (using a perceptron) + find the maximum spanning tree

Parsing Evaluation

Dependency parser metrics:

- dependency recall: $R_D = \text{Correct}(D)/|S|$, where Correct(D) is the number of correct dependencies (correct head / marked root), |S| is the size of the test data in words
- dependency precision: if output is not a tree – $P_D = \text{Correct}(D)/\text{Generated}(D)$, where Generated(D) is the number of output dependencies

Parse tree metrics:

- number of nonterminals may not be the same as in the "truth" –> more complicated
- crossing brackets measure: number of crossing brackets between the truth and the result
- labeled precision/recall – usual computation using bracket labels (phrase markers)
  - the bigger label coverage, the better – recall
  - the less brackets, the better – precision