CSCI 6509

Advanced Topics in Natural Language Processing

Vlado Keselj

Alex Safatli

Monday, January 5, 2015

Contents

Introduction

Course Information

General Information. Lectures take place on Mondays, Wednesdays, and Fridays at 1:35 PM. Labs are at 2:35 to 3:25 PM on Wednesdays in Teaching Lab 4 (for graduate students). Labs will not be held every week (only around 7 will be held). The first lab starts next week with subversion (SVN). The website is located at `http://web.cs.dal.ca/~vlado/csci6509` and an e-mail list is found at `nlp-course@lists.dnlp.ca`.

Only a single examination is required for this course, scheduled by the Registrar's Office at the end of term (April, 2015). A required textbook is available in the bookstore (Jurafsky and Martin, 2008). Assignments (3-5) in this course will focus on Perl, but projects are more flexible in this regard (e.g., Python, Java, C, C++).

The password protected section of the website requires user `nlp` and password `discourse`.

Course Project. For the graduate-level course, this will be a research project. They are individual or can be comprised of forms of up to four students. Presentations, however, will be individual for CSCI 6509. The final report is due on the 10th of April, 2015. A number of deliverables comprise the project including a topic proposal (P0), project statement (P1), presentation (P), and report (R). Some parts may be submitted using SVN. For e-mail submission, use the e-mail subjects as defined in the slide (9-Jan-2015, Slide 12).

What Is Natural Language Processing?

What is a **natural** language? In computer science, this is what people usually call a language in common communication – in computer science, we distinguish **artificial** (formal) programming and markup languages. Artificial languages are designed in a manner to only have one meaning (are not ambiguous). In natural language, ambiguity saves us effort and allow jokes to be made.

Applications. What are applications of NLP? What kind of useful programs can we make? Machine translation (of natural languages), speech analysis, spell checking, document generation, information retrieval, question answering (more or less solved for factual questions), support applications such as stemming, etc.

As a Research Area. Relatively old for computer science research areas (almost as old as the discipline). The problems are challenging and therefore the area is still active. The area can be seen as being part of artificial intelligence (AI). It is related to several other areas such as machine learning and text mining.

NOTE Useful links are found on the website for finding research articles.
NOTE Assignment 0 has been passed out in paper copy during the lecture on Wed Jan 7, 2015.

Methodology. There is no single tool to perform all applications in NLP. There is a number of different methodologies that can be divided into two larger groups: *Knowledge-Driven* and *Data-Driven* approaches.

Knowledge-driven and symbolic approaches are older and came first – they use crafted rules but have scalability issues. They are appropriate for more controlled language formats. Example applications include information extraction where from a body of text, we wish to extract fields of data into a database. The methodology involves regular expressions (matching), unification-based methods, etc.

Data-driven and stochastic approaches use machine learning – they are newer, scalable, and are for open-ended applications. With more data, the methodology does not typically fail. Examples are classification (e-mail spam) and clustering. Methodology includes probabilistic models, Bayesian classifiers, etc.

Short History of NLP. Refer to slides (7-Jan-2015, Slide 4).

Levels of NLP. There are a number of levels of abstraction for natural language (a stack, such as in networking).

1. *phonetics* (speech): physical sounds,
2. *phonology* (speech): sound system (phonemes) of a spoken language,
3. *morphology* (words): word structure,
4. *syntax* (words): inter-word structure (up to sentences),
5. *semantics* (meanings): literal meaning (up to sentences),
6. *pragmatics* (meanings): implied meaning extended from literal sentence meaning,
7. *discourse* (meanings): units larger than an utterance (e.g., inter-sentence meaning, references).

Phonetics is processing that is mostly performed using signal processing methodology. It is divided into speech generation and speech analysis. *Phonology* is linguistic processing of sounds of spoken language at a higher level, mainly concerned with elementary sound units of a language (phonemes).

Morphology involves morphological processes (word transformation), where there are three main processes: (1) inflection, (2) derivation, and (3) compounding. Inflection is the change of a word that is systematic and does not change the type of word (e.g., noun). Derivation is not so systematic and can have different meaning (be a different type of word). Compounding involves joining two words together. An example of processing is stemming.

Syntax is concerned with sentence structure, or are the rule for arranging words within a sentence. One of the main tasks is PARSING, a task of producing a parse tree given a sentence as input. The GRAMMAR of a language is a set of rules for deriving syntactic structure. Furthermore, there are different types of parse trees: context-free parse trees and dependency parse trees. A dependency tree helps map dependencies of words between others, where typically a root of the sentence does not depend on any other word. A handful of parsers are available; the better known one is the Stanford parser which allow for both trees.

Semantics and higher levels become less understood than those below. These become much more application-dependent. Meaning is everything in communication. Semantics is interpreting literal meaning up to a sentence level. We start with LEXICAL SEMANTICS, which is the semantics of words, and every word has semantics (example resource is WordNet). To build higher semantic representations, methodology includes using first-order logic (FOPC) and unification.

Pragmatics is concerned with intended, practical meaning of language. For example, the sentence "Could you print this document?" implies that the person should actually print the document. *Discourse* is concerned with language structure beyond the sentence level (inter-sentence relations, references, document structure).

Why Is NLP Hard? Some evidence that processing natural language is hard include:

- *Turing Test* — When asked about the intelligence of computers, Alan Turing proposed a test. The test primarily tries to prove whether a computer can behave in a way that is indistinguishable from a human. At its core, this is merely a language test.
- *Evidence from Neural Sciences* (language part of human brain) — Human brain is not very different from that of a monkey's, in terms of physical structure. Language facilities is really the most distinguishable element of the structure.

Some reasons why it is hard:

- *Highly Ambiguous* — Language is very ambiguous. Even for us, language can be ambiguous. Concepts of metaphors are difficult to ascertain for a computer.

- *Vague* (the principle of minimal effort) — Trying to speak precisely is difficult. Context can make up a lot of communication.
- *Universal* (Domain Independent) — Same words for different disciplines. Many knowledge bases, not built into a program, may be tapped into.

Ambiguities in Language. They exist at all levels of NLP. For example, phonological ambiguities include words that sound the same (e.g., "two" and "too"). At the lexical and syntactic levels, you can have lexical ambiguities (e.g., the word "hot") or syntactic ambiguities ("time flies like an arrow" or "I made her duck"). Semantic-level ambiguities are where meanings may be different. For example, what does "coast road" mean? Is it a road following a coast, or leading to one? A "carriage return", too, is an idiom. Furthermore, we can discuss the concept of *referential ambiguity*, a kind of semantic ambiguity also known as a discourse ambiguity – examples are "it" or "he" in a text and what they refer to. Pragmatic ambiguities include dates (10/11/12), "do you have a quarter?", and "can you pass the salt?".

NOTE An ambiguity at the lexical level is a homonym.

Stream-based Text Processing

Perl

Elements of Morphology

NOTE Reading: Section 3.1 ("Survey of (Mostly) English Morphology").

Morphemes. *Morphemes* are the smallest meaning-bearing units in morphology. For example, the word cats contains two morphemes `cat` and `s`, and the word unbelievably contains the four morphemes `un`, `believ`, `ab`, and `ly`. It could be sometimes debatable what is the proper way of breaking a word into morphemes, but not having a clear correct answer is not uncommon in analysis of natural languages.

Other important terms are *stems* and *affixes*, where stems provide "main" meaning and affixes act as modifiers (prefixes, suffixes, infixes, and circumfixes). Stacking multiple affixes is possible, but the number of which you can stack depends on the language. *Cliticization* refers to the notion of clitics appearing as orthographic or phonological parts of a word (but syntactically act as words; e.g., 'm, 're).

Tokenization. Text processing in which the plain text is broken into words. It may not be a simple process, depending on the type of text and kind of tokens that we want to recognize.

Stemming. Type of word processing in which a word is mapped into its stem, which is a part of the word that represents the main meaning of the word. For example, foxes is mapped to the stem fox, or the word semantically is mapped to the stem semanti.

It is used in Information Retrieval due to the property that if two words have the same stem, they are typically semantically very related. Hence, if words in documents and queries are replaced by their stems, the resulting indices are smaller, and words in a query can be easily matched with their morphological variations.

Lemmatization is a word processing method in which a surface word form, i.e., the word form as it appears in text, is mapped to its lemma, i.e., the canonical form as it appears in a dictionary. For example, the word working would be mapped into the verb work, or the word semantically would be mapped to the lemma semantics.

Elements of Morphology continued on next page...

Morphological Processes. A *morphological process* is a word transformation that happens as a regular language transformation. There are tree main morphological processes in English: (1) inflection, (2) derivation, and (3) compounding. Typically, these are done using finite automata.

An *inflection* is a transformation for when adding a single feature to a word, for example, when "work" becomes "working" or "worked". It is a small, systematic, change where a word remains in the same category; it is relatively regular and uses suffixes and prefixes. Different inflection forms are not usually entries in a dictionary; they are typically able to be done with any word.

Derivation is when a new word is derived from an existing word. This is less systematic and typically uses suffixes. Derivations are more radical changes (change word class) and do not necessarily work with any word. Unless people adopt these words, these typically are not accepted. For example, "wide" (adjective) can become "widely" (adverb).

When *compounding*, one combines words together. For example, "playground".

Characters, Words, and N-Grams

We looked at code for counting letters/words/sentences. We can look again at counting words. We can observe Zipf's law (1929): $r \times f \approx \text{const}$ (where r is rank and f is frequency).

A **word or character n-gram** is a sequence of tokens of fixed length in order they appear in the text; tokens are either words or characters.

NOTE See slide for Perl list operations and a program to read N-grams. On attempt 3, the `join` method is used – this method requires a string and an array and constructs a string that is the joining of all these elements with the separator specified between them. An opposite function to this is `split` which uses regular expressions. Notice that <> is assumed to read all lines – in Perl, this is known as **context polymorphism**. In Perl, when an expression is given, it will behave differently in different contexts (array vs. scalar). Furthermore, in Perl, functions are called sub-routines (`sub`).

N-grams could be of any token. However, generally, any token may span multiple lines. This could be handled, but leads to more complex code; special cases are necessary. It may even be relevant to compute n-gram frequencies, which could be useful for segmentation of texts (more frequency n-grams tend to be words or affixes). It turns out that Zipf's Law still relatively holds on these n-grams, as well.

NOTE There is a Perl module known as `Text::Ngrams`, a module available and written by the professor. There is also a collection of Perl modules known as `CPAN`, which can be installed on systems.

Elements of Information Retrieval and Text Mining

NOTE Reading: [JM] Sec 23.1, ([MS] Ch.15)

Information Retrieval is an area of computer science concerned with finding a set of relevant documents from a document collection given some user query (from a user whom has some sort of *information need*). The basic task definition (a task known as **ad hoc retrieval**) involves:

- *User*: information need expressed as a query

- *Document Collection*
- *Result*: set of relevant documents

This does not necessarily involve the internet. A typical IR system architecture is present on the slides; what must be done for a search to occur is **Indexing** of the documents in the Document Collection. Indexing takes a relatively long period of time. An open-source Apache Hadoop processing system performs this task. Searching, on the other hand, must be done quickly.

Steps in Document and Query Processing.

- A "bag-of-words" model – treating the query as a set of independent words typically works in such an operation.
- Stop-word Removal – a list of words that are present in most queries (e.g., articles).
- Rare Word Removal (optional) – there are a number of very rare words occurring only once.
- Stemming – removing affixes; it is questionable whether or not this is critical in the English language but it is very critical in other languages.
- Optional Query Expansion – adding more words to increase chance of getting matches. Typically not a problem with the internet. For example, could add synonyms.
- Document Indexing
- Document and Query Representation (e.g., sets — Boolean model, vectors) – Want to do efficient matching. One method is inverted word lists. Queries can be expressed as boolean expressions, also. Many older systems used to work like this. Another model, the vector space model, eventually came about.

Vector Space Model in IR. Choosing a global set of terms $\{t_1, t_2, ..., t_m\}$, documents and queries are represented as vectors of weights where weights are corresponding to respective terms. Weights can be binary (1 or 0), frequency, etc. A standard choice is *tfidf* (term frequency inverse document frequency weights) where *tf* is a frequency of a term (sometimes log) and *df* is document frequency (number of documents in collection with the term). The inverse document frequency involves the term N in the numerator where N is the total number of documents.

$$tfidf = tf \cdot \log\left(\frac{N}{df}\right) \tag{1}$$

This is typically state of the art. Using this model, we can represent queries or documents are points or vectors. How do we measure if a query is similar to a document or not? A number of similarity measures exist (see slides).

NOTE Lucene is an open-source search engine in Java that uses this model.

To evaluate IR, we have measures **precision** and **recall**. We can also consider some truly relevant documents (a gold standard). Precision is the percentage of true positives out of all returned documents (returned by the search) and recall is the percentage of true positives out of all relevant documents. Typically care the most about precision, but sometimes good recall is wanted (e.g., when want some useful e-mails). We can also refer to an **F-measure**, a weighted harmonic mean. Other similarity measures are also defined.

Some Text Mining Tasks. These include:

- Text Classification
- Text Clustering
- Information Extraction – filling out a table based on documents.

- Text Visualization
- Filtering Tasks, Event Detection
- Terminology Extraction

Text Classification, also known as Text Categorization, is defined by the problem of classification of a document into a class (category) of documents. The typical approach uses machine learning to learn the classification model from previously labelled documents (an example of supervised learning). *Types*: topic categorization, sentiment classification, authorship attribution (plagiarism detection), authorship profiling (age/gender, etc.), spam detection and e-mail classification, encoding and language detection, and automatic essay grading.

Creating Classifiers. Can be created manually (typically rule-based classifier), make programs that learn to classify (generated based on labeled data; supervised learning). A Common N-Gram Analysis (CNG Method) is a simple method that was initiall used for authorship attribution. It's based on a kNN method that is language independent with a similarity measure as defined in the slides (another method is to do something similar to the vector-based method).

Evaluation Measures. A confusion matrix, or contingency table, allows us to tabulate correctness of the classification, either at a global scale or by class. General issues in classification (and machine learning) are the notions of *overfitting* and *underfitting*. Underfitting means you are not correctly using the signal present in the data and representing it. Overfitting involves finding a function that perfectly represents that data you are trained on, but not properly classifying later data. The main methods for evaluating text classifiers involves (1) calculating training error — evaluating the classifier on the same training data which typically detects underfitting (biased), (2) use a train and test approach where results are evaluated on testing data that can help detect overfitting (unbiased), and (3) N-fold cross-validation (unbiased). Train and test can be useful because you can hide the testing data until you are done formalizing the classifier.

Probabilistic Approach to NLP

Logical vs. Plausible Reasoning

As part of AI (Artificial Intelligence), NLP follows two main approaches to *computer reasoning* or *computer inference*.

1. **Logical Reasoning** — a.k.a. Classical, Symbolic, Knowledge-Based AI. This reasoning is *monotonic* (once a conclusion is drawn, never retracted; with more facts, we can acquire more conclusions) and *certain* (conclusions are certain, given assumptions).
2. **Plausible Reasoning** — Probabilistic, Fuzzy Logic, Neural Networks. This reasoning is non-monotonic and uncertain; sometimes, when we know more facts, we can make less conclusions. In general, it is the process of combining ambiguous, incomplete, and possibly contradicting evidence to draw reasonable conclusions. Frequently approached as a task of making plausible inferences of hidden structures from observations. For example, with symptoms (observations), one can try to find an illness (hidden structure).

Probabilistic NLP as a Plausible Reasoning Approach. This is only a single form of plausible reasoning. Regular expressions and finite automata are examples of logical or knowledge-based approaches to NLP. Probabilistic NLP, however, makes use of the Theory of Probabilistic (stochastic, statistical NLP). Other forms include neutral networks, kernel methods, fuzzy sets/logic, Dempster-Shafer theory, rough sets, default logic, ...

Bayesian Inference uses generative models (forward generative models). It is one way of representing knowledge with a probabilistic model implying some manner of hidden structure. This is generative in the sense that it is

attempting to capture the generation of observations from some hidden truth. Bayesian inference is a principle of combining evidence. The probability of possible truth is typically referred to as *a priori* probability (before anything is "said").

NOTE For the formulation of bayesian inference, because the denominator is constant amount different comparisons for maximum argument, it can be disregarded. Another way of writing the numerator is as a joint probability $P(\text{evidence}, \text{possible truth})$.

NOTE Bayesian inference can be applied to speech recognition — acoustic model and language model where $P(\text{sound}|\text{digit})$ is an acoustic model and $P(\text{digit})$ is a language model (for any sentence in English, give a probability of that sentence).

NOTE P0 is due this Monday (by e-mail), an informal proposal of the project.

Probabilistic Modelling

How do we create and use **probabilistic models**? Key elements of a model are *random variables*, *variable dependencies*, and *model configurations*. A **random variable** V defines an event as $V = x$; this is usually not a basic event due to more variables being present. When establishing a set of them, we call this a *full configuration* (e.g., $P(V_1 = x_1, ..., V_n = x_n)$) or a *partial configuration* if only a subset of variables are identified. We can also speak about *observable* and *hidden* variables.

Probabilistic modelling in NLP can be described as a general framework for modelling NLP problems using random variables, random configurations, and finding effective ways of reasoning about probabilities of such these configurations. The computational tasks in this modelling include:

1. *Evaluation*: compute probability of a full configuration,
2. *Simulation*: generate random configurations,
3. *Inference*: with subtasks
    - Marginalization: computing probabilistic of a partial configuration,
    - Conditioning: computing conditional probability of a completion given an observation (typically, this involves finding the probability of hidden variables given observable ones.), and
    - Completion: finding the most probable completion, given an observation.
4. *Learning*: learning parameters of a model from data.

Example. An example can be found in spam detection (see slides) where random variables, simply, could be if the subject line contains all uppercase letters, if the word "free" appears in the subject line, and whether or not the message is spam. One random configuration represents a single e-mail message.

A random sample can be obtained by selecting a number of random messages and representing them as random configurations to generate a joint distribution model. To perform simulation with this model, an interval can be created and random numbers can be obtained (roulette wheel).

Probabilistic Modelling continued on next page...

Marginalization can involve identifying what messages are spam. Conditioning can typically be reduced to marginalization tasks; for instance, the probability of a spam message, given that it contained "free", could involve computing the conditional probability. Finding a maximum for probability of spam given certain other variables would be completion.

All of this together can provide a simple spam classifier. Learning, in this case, would also be relatively simple.

Naive Bayes Model

NOTE Graphically, the Naive Bayes Model can be remembered as a (set of) class(es), in the sense of a Bayesian Network, where each node is associated with a conditional probability table (CPT) related to the class(es).

From the point of view of a probabilistic model, the main drawbacks of the joint distribution model is the memory cost to store the table, the running-time cost to do summations, and the sparse data problem found in learning. Other models are found by specifying specialized joint distributions satisfying certain independence assumptions. The goal is to impose structure on a joint distribution; one key tool for imposing structure is variable independence. Therefore, the joint probabilities no longer need to be calculated as we are only concerned with their products.

The **fully independent model** is of little practical use but is useful from a theoretical point of view. The assumption here is that all variables are independent. This is an efficient model with a small number of parameters = $O(nm)$ where n is the number of variables and m is the number of different values each respective variable can have (the domain of each variable). Unfortunately, this is usually too strong an assumption.

In the **Naive Bayes classification model**, *all variables are independent except the class variable.* If we assume that the variable V_1 is the output variable, and other variables are input variables, the classification problem is expressed as a conditional probability computation problem. Another derivation involves the chain rule of probability where $P(V_1, V_2, ..., V_n) = P(V_n|V_1, ..., V_{n-1}) \cdot P(V_1, ...V_{n-1})$. Approximately, by the Naive Bayes assumption, this means that $P(V_1, ...V_n) = P(V_1)P(V_2|V_1)$.... The table configuration thus involves a single table containing columns for V_1 and corresponding probabilities for it occurring, and then a number of other tables equal to the number of other independent variables (CPT — conditional probability tables). Learning is done by estimating parameters using a corpus (maximum likelihood estimation). This classification model possesses an order $O(m^2 n)$ complexity.

This model is useful as it is efficient, there is no sparse data problem, and it has surprisingly good performance (accuracy) in text classification, etc. Unfortunately, it can be over-simplifying and cannot model more than one "output" variable.

N-gram Model and Smoothing

An important task in probabilistic NLP is language modelling where you estimate the probability of arbitrary NL in a sentence – P(sentence). For example, this is found in speech recognition using the acoustic or language models. In an N-gram model, we can investigate a sentence with n words and model the probability that this sentence is comprised of a set of words by modelling using the expression:

$$P(V_1 = w_1, V_2 = w_2, ..., V_n = w_n) \tag{2}$$

N-gram Model and Smoothing continued on next page...

This can be expanded using the chain rule (see above) of probability expressions. Consequently, for any given word, we must know the probability of any previous word in the sentence. In reality, it is practical to assume that words earlier on in the sentence become less as important as near the end of the sentence in the probability of successive words. In essence, we are considering a Markov model here. Any Markov chain over a finite domain can be represented using a deterministic finite automaton (DFA). This also has application in page ranking used by the Google search engine.

Notice that, given some language model, we can consider the probability of a certain construction of text to occur in that language. Evaluation of this sort of language model is using extrinsic or intrinsic methods. Extrinsic evaluation is when the model is embedded in an application, whereas intrinsic evaluation involves direct evaluation using a measure. The **perplexity** measure evaluates some text W of length L where the evaluation is independent of text length (a weighted average branching factor).

Smoothing refers to methods that are used to avoid a probability of zero due to sparse data (e.g., a probability of zero which implies impossibility, but this is not typically true and only an indication of no observation found). Some smoothing methods include add-one smoothing (Laplace smoothing), which is typically too simple, Bell-Witten smoothing, which is state of the art, and Good-Turing smoothing.

Add-one smoothing starts with a count of 1 for all events, regardless of if it was observed or not. Smoothed unigram probabilities would be equal to

$$P(w) = \frac{\text{number}(w) + 1}{N + V} \tag{3}$$

where N is the length of text in tokens and V is the vocabulary size (unique tokens). However, recognize this may be giving too much probability for unseen events. To fix this, we move to Bell-Witten smoothing.

Bell-Witten smoothing follows from an idea of data compression by Witten and Bell (1991) where tokens are encoded as numbers when they are read, and using special (escape) codes to introduce new tokens. The frequency of "escape", or encoding a new number, is the probability of an unseen event which can be distributed evenly across all other possible tokens.

Example. Character Unigram Probabilities. Imagine we have the word "mississippi". What are letter unigram probabilities? What would be the probability of the word "river" based on this model? This would be the product of the probabilities of any of its comprising letters, which will include zero probabilities. Using add-one smoothing, the probability of observing the letter `r` would be $\frac{1}{37}$. Using Bell-Witten smoothing, it would be $\frac{4}{15}$ divided by 22 (the number of other possible tokens).

Part-of-Speech (POS) Tagging

Words of natural languages are classified into part-of-speech (POS) classes, also known as syntactic categories, grammatical categories, or lexical categories. Consider, as an example, the following sentence. This example also illustrates ambiguities in part-of-speech tagging.

Time flies like an arrow.

Example. In the sentence "time flies like an arrow", there are a number of different ways of tagging its comprising words. Two ways are: (1) N V P D N, (2) N N V D N.

In the example above,

- "N" denotes a *noun*,
- "V" denotes a *verb*,
- "P" denotes a *preposition*, and
- "D" denotes a *determiner*.

The problem of associating each word in a text with its tag is called **POS tagging** but may depend on the text interpretation of a reader.

Open and Closed Categories. Word categories are divided into two sets. (1) *Closed* or *functional categories* (fixed set, small set, frequent words, incl. articles, auxiliaries, prepositions) and (2) *open categories* (dynamic set, larger set, content words, incl. nouns, verbs, adjectives).

CLOSED CATEGORIES are lexical categories consisting of fixed sets of words used frequently in text and typically used in a *functional fashion*, i.e. as syntactic fillers and with less information content. Examples include determiners, pronouns, prepositions, particles, auxiliaries and modal verbs, qualifiers, conjunctions, and interjections.

Typically, these words have no morphological transformations.

- *Determiners* (DT) include articles "the", "a", and "an", demonstratives "this", "that", "those", "some", "any", "either", and "neither", and quantifiers "all" or "some".
- *Interrogative Determiners* (WDT) are tagged as a separate class and include "what", "which", "that", "whatever", and "whichever".
- *Predeterminers* (PDT) include "both", "quite", "many", "all", "such", and "half". An interesting classification of determiners was made by Bond 2001 (see lecture notes).
- *Personal Pronouns* (PRP) (used to be PP) include "I", "you", "he", "she", "it", "we", "you", and "they". The Brown Corpus has a special tag PPS for plural personal pronouns ("we", "you", "they"). Personal pronouns in accusative cash have tag PPO in the Brown Corpus. Reflexive pronouns are PPL and PPLS in the Brown Corpus. The pronouns have the following features:
  - Number: singular (sg), plural (pl).
  - Person: first (1st), second (2nd), third (3rd).
  - Case: nominative (subject), accusative (object).
  - Gender: masculine (he), feminine (she), neuter (it)
- *Possessive Pronouns* (PRP$) include "your", "my", "her", "our", "his", "their", and "its". The second possessives ("ours", "mine", "yours", ...) are tagged PRP or PP$$ in Brown Corpus.
- *Wh-Pronouns* (WP) include "who", "what", "whom", and "whomever" and *Wh-Possessive* (WP$) include "whose".
- *Prepositions* (IN) reflect spatial or time relationships and include "of", "in", "for", "on", "at", "by", and "concerning".
- *Particles* (RP) are frequently ambiguous and confused with prepositions. They are used to create compound verbs. For example, "put off", "take off", "give in", and "take on".
- *Possessive Ending* (POS) involve a possessive clitic ("'s").
- *Modal Verbs* (MD) include "can", "may", "could", "might", "should", and "will" as well as their abbreviations (with "'d" and "'ll"). Negative forms are separated into a modal verb and an adverb "not" (such as "couldn't"). Auxiliary verbs are "be", "have" and "do".

- *Infinitive Word* "to" (TO) is used to denote an infinitive (to call). Similarly, "na" is marked as TO in "gonna", "wanna", and similar.
- *Qualifiers* (RB) are closed adverbs and are tagged as adverbs (RB). For example, "not" and "very". Postqualifiers are "enough" and "indeed".
- *Wh-Adverbs* (WRB) include "how", "when", "where", "whichever", and "whenever".
- *Conjunctions* (CC) are words connecting phrases; coordinate conjunctions (CC) connect coordinate phrases. Subordinate conjunctions connect phrases where one is subordinate to another and are tagged as prepositions (IN) in Penn Tree bank.
- *Interjections* (UH) include "yes", "no", "OK", and "please".

We can also discuss the Existential "there" (EX), such as the "there" in the sentence:

There are three classes per week.

OPEN CATEGORIES are lexical categories dynamic in a sense that their content changes over time or depending on dialect/domain of usage. Along with being larger sets, these words bear most of the information content in a text. These include nouns (NN, NNS, NNP, NNPS), adjectives (JJ, JJR, JJS), numbers (CD), verbs (VB, VBP, VBZ, VBG, VBD, VBN), and adverbs (RB, RBR, RBS).

Tag Sets. There are traditionally eight parts of speech (nouns, verbs, pronouns, prepositions, adverbs, adjectives, conjunctions, and articles). In practice, with computer processing, there is a need for a larger set of categories. Various POS tag sets (in NLP) are found in Brown Corpus, Penn Treebank, CLAWS, C5, C7, ... We will look in detail at the Pen Treebank system of tags, with mentioning to some of the Brown Corpus tags.

WSJ Dataset. Wall Street Journal (WSJ) dataset is most commonly used to train and test POS taggers. It consists of 25 sections (of about 1.2 million words).

REMAINING POS CLASSES include foreign words (FW), list items (LS), and punctuation.

Example. Referring to the previous sentence: "There are three classes per week". Respectively, the POS tags in this sentence would be: EX VBP CD NNS IN NN.

Hidden Markov Models

A **Hidden Markov Model** (HMM) is typically used to annotate *sequences of tokens* with the most common annotation being part-of-speech (POS) tags. While a regular expression could be used to determine some parts of speech, how would we use regular expressions to identify personal names?

The input to part-of-speech tagging is a sentence while the output should be the same sentence with each token associated with one of the POS tags. To solve this problem using probabilistic modeling, it is natural to associate all tokens to probabilistic variables along with their tags. There are dependencies between words and their associated tags, but it also seems that tags form some typical sequences, so there are dependencies from each tag to the following tag. This is the motivation for the HMM model being introduced here.

Recall *Markov Chains* from earlier in this course. If we assume states in such model are *not observable* (hidden) and we can only observed "emitted" symbols, based on another distribution for producing observable symbols, we obtain the HMM model, a kind of generative (ordered tasks) model.

HMMs are used in language modelling, acoustic modelling, part-of-speech tagging, and many kinds of sequence tagging (e.g., extracting bio-medical terms).

Hidden Markov Models continued on next page...

Formally, we can define an HMM as a five tuple (Q, π, a, C, b). Let $Q = \{q_1, q_2, ..., q_N\}$ be the set of N hidden states, π as the initial distribution $\pi(q)$ for each state q, a as transition probabilities for any two states q and s, V as the output vocabulary $V = \{o_1, o_2, ..., o_m\}$ and b as the output probability for each state s and observable o.

In more precise terms, an HMM includes a finite set of hidden states Q. A probability distribution $\pi(q)$ specifies, for each state, the probability it will be the initial state where $\sum_s \pi(s) = 1$. Instead of a deterministic transition from state to state, for each pair of states there is some probability of transition. From each visited state, an observable o is generated where $o \in V$ with some probability b.

Assumptions. Given an HMM, we can generate samples by generating an initial state, producing an observable, and then creating another state, and so on. For a length n, a graph can be devised to illustrate operation of an HMM (similar to a previous graphical representation of the Naive Bayes model known as the Belief Network or Bayesian Network representation). The value of some X_1 is the initial state and each consecutive variable X_i are consecutive states with produced observables.

The HMM assumption formula is:

$$P(X_1, O_1, ..., X_n, O_n) = P(X_1)P(O_1|X_1)P(X_2|X_1)P(O_2|X_2)...P(X_n|X_{n-1})P(O_n|X_n) \tag{4}$$

NOTE See [JM] Section 5.5 for more on HMM POS Tagging.

POS Tagging. For POS tagging, hidden states are *tags* and observable variables are words. In practice, higher-order taggers are used with a bit more history (e.g., two previous tags).

Computational Tasks. Evaluation requires use of the assumption formula. Generation is done in order dictated by the "unrolled" graphical representation. Inference involves marginalization, conditioning, and completion. Learning involves MLE if labelled examples are given.

Bayesian Networks

Hidden Markov Models, Naive Bayes Models, etc. are all specialized forms of more general model distinguished as **Bayesian Networks**, a generative probabilistic model. Graphs in these networks are directed, acyclic graphs where each node represents a (random) variables and edges represent dependencies, allowing us to see how we set parameters. The probability of a complete configuration is a product of conditional probabilities, and each node corresponds to a conditional probability.

Computational Tasks. Evaluation uses the Bayes assumption. Simulation simulates a note, one after another and following edges. Learning involves counts. Inference is not trivial; one way it can be solved is with a brute force technique. In some Bayesian Networks, inference is always expensive (e.g., joint distribution has a very large number of parameters). Can we be more efficient if number of parent nodes is limited? Naive Bayes or HMM has a limit of parents to 1. If we limit them to two, this may lead to an NP-hard inference problem (proof: a reduction to this problem from Circuit Satisfiability Problem). Typically, if a BN does not have loops, it can computed efficiently.

Inference in Tree Bayesian Networks. The basic idea here is optimizing *sub-product calculation* (Kschishang et al., 2000) and has two major components:

- Construction of a factor graph
- Message-passing algorithms

Within the network, *factor nodes* will be introduced, one for each variable, meant to represent single conditional probability tables of those variables. The factor graph is claimed to capture the structure of computation. A factor

Bayesian Networks continued on next page...

graph allows efficient calculation of sums of products of these tables. *Message passing* is used to facilitate this process.

Example. In the factor graph example, *f1* represents a table for $P(B)$ and, similarly, *f3* represents $P(A|B,E)$.

A *message* summarizes computation in the corresponding part of the graph, where messages are vectors of real numbers. Each node passes to each neighbor node a message exactly once; to pass one to a neighbor, a node needs to receive messages from all other neighbor nodes (this is why loops are a problem). An important property is that tree-structured Bayesian Networks lead to tree factor graphs, a type of *bipartite graph*. Even if the structure has loops, this algorithm could still be useful (random message sending where there is a high chance of convergence to a local maximum).

There are two main types of computation:

- Summation of resulting overall products where variables take different domain values.
- Finding variable values for which overall product is maximized (max-args).

And there are two main situations, a result of the bipartite graph:

- Factor node passing a message to variable node (most expensive),
- Variable node passing a message to factor node (trivial). A vector of 1s are used by default.

Marginalization with message passing, in general, considers the calculation of $P(V_1 = x_1, ..., V_k = x_k)$ where $V_1, ..., V_k$ are evidence variables and instantiated values $x_1, ..., x_k$ are observed evidence. Conditioning requires hardwiring certain numbers of variables.

HMM as Bayesian Network. Revisiting the previous topic of HMMs, HMM can be seen as a *tree-structured* (there are no loops) Bayesian Network. From an HMM unrolled visual representation, we can construct a factor graph. Initial probability tables (factors) connect to parents of the variables for states. Realize that the factor tables parent to all leaf variables can be referred to as the same factor table. In HMMs, hidden layers have the same probability tables. Message passing gives an efficient solution here; all messages need to be passed such that all nodes receive all messages.

Parsing (Syntactic Processing)

Syntax

Syntax is the study of the structure of the natural language sentence, and how words are connected into a richer structure. Words are not randomly ordered; word order is important and non-trivial. While there are "free order" languages (Latin, Russian), they are still not completely order free. Comparably, English has a very strict format. Usually, the tradeoff is how inflect-able the language is and the role of these inflections.

There are two ways of organizing sentence structure:

- Phrase Structure
- Dependency Structure

Phrase Structure. Phrase Structure Grammars or Context-Free Grammars. A hierarchical view of sentence structure (words form phrases, phrases form clauses, and clauses form sentences). The main NLP problem in syntax is

Syntax continued on next page...

parsing. Given a sentence, find a correct structure of the sentence (typically, in the form of a parse tree). Sometimes parse trees can also be simplified by ignoring parts of the structure (see slide example).

Recall we can define *rules* or *productions* for context-free grammars. In these rules, we would like to know exactly what symbols are the terminals (leaves). We also need to know non-terminals or variables and the root start symbol (`S`). This allows us to make derivations ($\Rightarrow$) of the rules and acquire a list of terminals. Direct derivations involve replacing a non-terminal with a set of other non-terminals or terminals. A (full) derivation is a set of direct derivations. Recall LL and LR, corresponding to left-most and right-most derivations.

Context-Free Grammars are equivalent to BNF (Backus-Naur form). Formally, they were defined by Chomsky and Backus in the 50s. Once you possess a CFG, you can generate different sentences, language (a context-free language). A parsing task involves taking, as input, a sentence and answering whether or not the sentence belongs to the language and what the parse tree for that sentence is. Programming languages usually have a single parse tree (designed that way; no ambiguity).

Typical Phrase Structures Rules in English. For a sentence, we can have *declarative sentences* (saying what you want), *imperative sentences* (issuing a command; skip noun phrase), and *questions* of which include syntactic structures yes-no questions, Wh-subject questions, and Wh-non-subject questions among other types. Sentences inside sentences are *relative clauses* ($\bar{S}$ or `SBAR`). Recognize the presence of a *gap* (`*gap*`) inside sentences (optional parts of sentences).

- *Noun Phrases* (NP) are typically pronouns, proper nouns, or determiner-nominal constructions. The main noun is a head noun which may have modifiers before or after.
- *Relative Clauses* (sentence-like phrases) follow a noun phrase.
- *Verb Phrases* (VP) organize arguments around a verb. These include intransitive verbs, transitive verbs, ditransitive verbs, and sentential complements (clauses modifying a verb).
- *Preposition Phrases* (PP) are noun phrases following a preposition.
- *Adjective Phrases* (ADJP) and *Adverbial Phrases* (ADVP) are less common.

Are Natural Languages Context-Free? Can we use CFGs directly to model the syntax? It is surprisingly effective in many cases but it is, however, not considered sufficient. Some NL are provably not context-free due to $ww = w^2$ forms (repeating same sequence to be grammatically correct). Additionally, we have what is known as an *NL Phenomena* where are some features of NLs cannot be captured properly or easily:

- *Agreement* is a phenomenon where constituents are required to agree on some features before being combined to larger constituents. For example: "this book" vs. "these book" (subject-verb agreement). Relevant features are propagating from child nodes to parent nodes. Agreement can also be a non-local dependency (do not have to be close).
- *Movement* is a phenomenon where a constituent in a grammatically valid sentence can sometimes be moved to another position and continue to yield a grammatically valid sentence. For example: "are you well?" from "you are well" or wh-movement.
- *Subcategorization* denotes the tendency of verbs to have certain preferences regarding arguments they can take. Verbs may allow or disallow certain arguments in a verb phrase or sentence. For example, some verbs do not take a noun-phrase object while some verbs do take an object, or two objects (direct and indirect) – this depends entirely on what verb is being used. This divides the category of verbs into finer grained subcategories of different types of verbs based on arguments they can take. For example, "he disappeared the eraser." The verb "disappeared" does not like objects (verb valence). This is analogous to function signatures in programming languages.

Syntax continued on next page...

Heads and Depedency. A phrase typically has a central word (a head or governor) where other words are direct or indirect dependents. These typically get propagated up a parse tree; the head of a verb phrase is usually made the head of a sentence (surprisingly has a lot of semantic meaning, and for purposes of agreement, etc.). See examples on slide(s).

Head-Feature Principle. A principle that a set of characteristic features of a head word are transferred to the containing phrase. A head child distinguishes itself among other children, and some of these features are passed (e.g., a word being plural). Can annotate a head in a context-free rule. Can also talk about *head-driven phrase structure grammars*. These grammars are richer than CFGs. In this grammar, when simple context-free rules are written, will use square brackets (e.g., a `H` is in front of the respective square brackets).

Dependency Trees. Can talk about *dependency grammars* as well which comes out of annotation of dependencies via heads (with subscripts on non-terminals). Head of the entire sentence is pointed to by all other heads. This sort of structure creates a dependency tree or structure. To encode this, all we need are indices for every word and associate these indices with their heads (a sort of graph structure). This representation is very descriptive of the semantics of a sentence. Many parsers today produce parse trees and dependency structures.

Arguments and Adjuncts. There are two kinds of dependents:

- *Arguments* are mandatory dependents (without them, the sentence loses its meaning). For instance, because of subcategorization.
- *Adjuncts* are not required (still would be grammatical) and can be moved around (more) easily. Can contribute meaning but are not essential. They have a "less tight" link to the head.

Parsing Natural Languages. Must deal with possible ambiguities. Cannot just use recursive descent parsers. Can also talk about partial parsing (noun-phrase chunking). Decide whether to make a phrase structure or dependency parser. Algorithm is usually based on charts (*chart parsing*) – a simple chart algorithm is **Cocke-Younger-Kasami** (CYK) algorithm and can only be applied to a Chomsky Normal Form (CNF) grammar.

Chart Parsing. Chart parsing involves partial parsing pieces of the sentence using a chart and chart entries covering parts of the sentence gradually. This is a form of dynamic programming.

We cannot just generate all possible parse trees. Some grammars will have a very large amount.

Chomsky Normal Form (CNF) grammars are grammars where all rules are in one of the forms: (1) $A \rightarrow BC$ where A, B, C are non-terminals or (2) $A \rightarrow w$ where w is a terminal (not ϵ). If a grammar is not CNF, it can be converted to one. Only ϵ symbol that is allowed is for the start symbol.

CYK Algorithm. Use a three-dimensional table β (starting index i, length of span being covered l, non-terminal in question k). Can see pseudocode on slides.

Probabilistic CFGs

NOTE Reading for this section is Chapters 13 and 14.

PCFGs, or Stochastic Context-Free Grammars (SCFGs), are an idea driven by the motivation for resolving ambiguities by applying a probabilistic framework to grammars. It involves creating a generative model where derivations can be modelled as a generative stochastic process. Probabilities are given to each rule in the grammar, assigned according to this generative process.

Computational Tasks for PCFG Model. How do we do evaluation, ..., and completion in this model? The problem with these grammars is that they are more complex than previous models. These trees can grow in complexity and

possesses a very dynamic model for configurations.

Evaluation. The probability of a tree. Acquired from the product of all probabilities for the generative processes involved in the tree, starting from the basic derivation. See slide for example.

Generation. Or sampling/simulation. This involves continuously making derivations until we get a terminal sentence. This can be repeated multiple times. This process, however, can go forever. There is no guarantee we will get all terminals. Our sentential form could get larger and larger without any sight of a terminal. There are therefore two types of PCFGs: PROPER and NON-PROPER grammars. Proper grammars will stop with probability 1. A grammar learned from a corpus is always proper.

Inference. Marginalization tends to refer to the probability of a sentence. Computing all trees could be the way to go here, but this is likely inefficient – a parsing algorithm is necessary (adapt CYK algorithm by changing two lines in order to accumulate probabilities). Conditioning refers to parsing and whether or not a tree is probable for that sentence. Finally, completion would be the best tree. This is similar to marginalization but effectively always takes the maximum possibility.

An interesting open problem is whether inference in PCFGs can be reduced to a message passing style algorithm as used in Bayesian Networks (reduce it to Bayesian Network).

Issues with PCFGs. They:

1. Do not capture structural dependencies, and
2. Do not capture lexical dependencies.

Structural Dependencies are dependencies on position in a tree. Example: consider rules NP → PRP and NP → DT NN. PRP is more likely as a subject than an object (on the left side of the tree, usually). NL parse trees are usually deeper on their right side.

Lexical Dependencies include the example of the PP attachment problem. We usually resolve using probabilities of NP → NP PP vs. VP → VB NP PP. But it is very important that sort of verb is there – they depend on the verbs and prepositions. A solution could involve the use of probabilistic lexicalized CFGs using heads of phrases, which is relatively new research.

Parser Evaluation. Still a matter of discussion and research. Required labelled data. PARSEVAL measures are commonly used for context-free parsing.

Semantics and Unification-based NLP

Semantics and Unification-based Approach to NLP

Semantic Analysis. Recall this is a way to extract meaning and represent it for natural language (as a language or data structure). It is typically syntax-driven and has the following computational requirements: (1) verifiability, (2) unambiguous representation, (3) canonical form, (4) inference, and (5) expressiveness. An example of a semantic representation language that is typically used is first-order logic (FOL). However, this practically too ambitious.

Lexical Semantics. The basic level of semantics is word or lexical semantics – word meaning is the basic element for compositional semantics. What is a word? A *wordform* is a word as it appears in text or speech (orthographic or phonological representation). A *lexeme* is a (wordform, meaning) pair with optionally more information. A *lexicon* is a set of lexemes (or database). A lemma or citation form refers to its appearance in a dictionary entry.

Word Senses. One word can have more senses. There are homonymns such as "bank" or homophones ("wood" and "would"). Furthermore, we have homographs – words written the same way but pronounced differently. While

homonymy is a problem, a bigger problem is polysemy where some words can adapt new meanings in real-time ("he went *flying* out the classroom"). A very useful resource for these relations is WordNet with many synonyms (synsets) that are connected with relations.

Semantic Compositionality. How meanings of the pieces (words) combine into meaning of the whole. There are a number levels of compositionality: (1) compositional semantics (e.g., "white paper"), (2) collocations (e.g., "white wine" has its own meaning), and (3) idioms (e.g., "the tip of the iceberg" are collocations that are even less compositional).

Semantic Roles. Syntax is closely related to semantics. Subcategorization frames can be used to assign *semantic roles*, for instance of sender, recipient, and object. An example is the verb "send" that can be placed into a semantic frame to assign these roles. Constructions these frames is not just important for semantics. Usually, you cannot do proper syntactical analysis without semantics.

Unification or Knowledge-Based NLP. Involves fine engineering of parsing and not depending on probabilities. Mostly started with roots in logic (Aristotle) and first-order predicate logic. With computers, this mathematical reasoning could be done computationally but this comes with the issue of a lack of intuition. Prolog came out of this and immediately was recognized as being useful for semantics (and syntax). Grammar formalisms followed.

First-Order Predicate Logic or first-order logic or calculus describes mathematical logic and reasoning. It is constructed of constants (atoms), variables (can take on values of constants), and functions (symbols or operators that take arguments).

With these, terms are constructed; variables and constants are terms and more complex terms are built by providing a function with terms as arguments. Predicates look like functions but are assumed to be statements – they return only true or false. True and false symbols can be regarded as zero-arity predicates. Logical formulae involve quantifiers (for all, there exists) and logic operators which are defined recursively. There are atomic formulae (predicate with terms) or recursively by combining formulae.

We can also discuss the notion of bound and free variables, open and closed formulae (non-bound vs. bound variables), and sentences.

Inference rules are templates for deriving new facts from known facts. Some sentences are chosen to be axioms (assumed to be true) and along with these inference rules, we can derive new true sentences to define formal proofs.

Resolution-Based Inference System. Only came with a single inference rule (otherwise, too many to encode).

Definite Clause Grammars

NOTE The last lab is this Wednesday. Assignment 8 is due this Thursday.

Parsing with Prolog. More details are provided in the lab. We can talk about an **implicative normal form** where $p_1 \wedge p_2 \wedge ... \wedge p_n \Rightarrow q_1 \vee q_2 \vee ... \vee q_m$ that is logically true and can be verified – can capture any other inference rule. If $m \leq 1$, the clause is called a **Horn clause**. The right side causes large complexity because you must try a number of different options (all are possible); therefore, if we have all Horn clauses, inference can be done relatively efficiently.

A value of $m = 0$ implies a truth statement (only the value $\top$) and is called a **fact**. A Horn clause with $m = 1$ is a **rule**. In Prolog, a fact is expressed as

```
p1, p2, ..., pn
```

and a rule is expressed as (where :- is analogous to $\Leftarrow$ and a comma is analogous to $\wedge$)

Definite Clause Grammars continued on next page...

```
q1 :- p1, p2, ..., pn
```

Prolog is very convenient for databases. Databases are merely sets of facts. The interpreter essentially queries these database(s). The two important features that Prolog offers, that other more common languages do not provide, are *unification* and *backtracking*. One problem with it, though, is that it forces you to think in a different way.

When an interpreter executes a statement, it tries to prove it. It will go to all facts and rules to satisfy that statement. Anything not compatible is ignored until necessary. Matching or *unification* is the process of trying to reduce the rules/facts to the statement. This may involve backtracking. "Functions" are known as goals in Prolog. Purists in Prolog suggest that "cuts" and other techniques should never be used.

Parsing leverages backtracking to parse CFGs. The developer of Prolog actually used it to parse languages. A simple CFG is on the slides (involving sentences "the dog runs" and "the dogs run"). We want to parse efficiently and want a mechanism to take words out and try other variations of the sentence (possibly formatted as a list). To do this, we use a *difference list*; a returning from a goal the remainder of a list. The use of difference lists is efficient because it would not have to try every possible split of collections of words.

Prolog has a built-in mechanism, **Definite Clause Grammars** (DCGs), for parsing and have implemented a special, simplified notation for predicates. This is merely syntactic sugar but allows you to leverage other parts of Prolog as well. Using this formalism, we can also build parse trees. The easiest way in Prolog to do this does not involve tree structures; we use terms instead. Recognize the symbols within the terms are not the same (e.g., `s` and `s(Tn,Tv)`).

Recognize this also allows us to *handle agreement* and encode agreement information! With DCGs, we can also define additional code within braces (protected from being interpreted as a non-terminal). With this embedded code, that means we can also easily express PCFGs.

Unification

> **NOTE** Unification techniques can be implemented in other languages, but it tends to be more elegant in Prolog. Furthermore, in other languages, backtracking already tends to be inefficient and other mechanisms would be more useful. Prolog remains to be the most elegant method to perform this sort of task. For a short time, microprocessors were going to be generated to work hand-in-hand with the language, but that never came about.

Classical unification comes about from the problem of making two terms equal by substituting some variables with terms. A useful reading is Kevin Knight's 1989 paper in *ACM Computing Surveys*. Classical unification is the foundation for several important extensions: (1) unification in Prolog, (2) matching and merging structured information, and (3) graph unification.

See slides for example(s). On slide 28 (20-Mar-2015 slides), the second last statement would unify to $f(g(x), g(g(x)))$. The last statement would unify to $f(g(f(T), f(T)), g(f(T), f(T)), g(f(T), f(T)))$.

Some important concepts include **substitution** (replacing a *finite* set of variables with terms – a mapping (function) from variables to terms), **unifier** (substitution that makes two terms equal), **unifiable terms** (terms that can be unified), and unification as an operation on terms. Unification is thus the search for a unifier for unifiable terms. Can define a procedure UNIF, a procedure that returns either a substitution (binding list) or a failure (`fail`) if no substitutions exist. There can be more than one substitution that unifies two terms – we are looking for the *most general unifier*.

To define the **most general unifier** (MGU), we need to define the *composition of substitutions*, identical to the notion of function composition. Two substitutions can be composed by applying one and then another on a term; we can talk about compositions of these substitutions that is equivalent to applying both (order of substitutions mattering). The MGU of two terms is a unifier such that any other unifier can be expressed as a composition of

Unification continued on next page...

some other substitution and it. It makes the least commitments, and if terms are unifiable, then MGU exists and is unique up to variable renaming.

Important Unification Algorithms include the Robinson 1965 exponential algorithm, Huet's 1976 almost linear algorithm (relatively simple), and an improved Huet's algorithm that is linear from Paterson in 1976.

Robinson's algorithm functions by considering two terms and checking them to see if one of them are variables. Otherwise, if either of them are constants, the other must be the same constant. The easiest solution is substitution. If neither of these are true, go on to more complex checks (with recursion). When applied on certain expressions, changes can begin to accumulate. The problem with this algorithm is its complexity – it is exponential and thus $O(2^n)$ in the worst-case.

We can improve this by considering a graph representation where nodes are symbols and arity are edges from nodes. Unique nodes ensure that we can identify those symbols that point to the same entity. When unification occurs, we perform efficient graph unification of graphs. Recognize, though, that "printing" or traversing this graph would be exponential (DFS with repetitions).

Huet's algorithm works with this concept and uses a stack. The structure is a union-find one. To make it more efficient, we perform weighting and keep track of which are deeper or more shallow to ensure trees are balanced. Also, long paths can be compressed. Amortized analysis means finding representatives in the union-find is constant time amortized.

Feature Structures

Most unification grammars after DCG are based not on classical FOPL terms, but on **feature structures**. The two main differences are:

1. Add named arguments (to manage growing numbers of them), and
2. Introduce types (to model inheritance of linguistic features among syntactic constituents).

An example of such a structure is the term `np(np(Td,Tn),A)` which could be replaced with an attribute-value matrix (AVM) a.k.a. feature structure, we have attributes related to values (akin to JSON format). AVMs have the following basic elements: (1) types, (2) attributes, and (3) values. It is very useful for semantic representation and representing information in general. We can also discuss the notion of a top-most general type (e.g., the most general Object). We can also express DCGs as AVMs.

Feature paths is an idea of following paths from the root to an element. Feature structures can also be expressed as graphs, as we had seen previously with classical unification. Cyclic AVMs may also be permitted – if we think about file systems, these are similar to symbol links. The PATR-II style of representing rules is convenient for representing rules in that it is similar to context-free grammars using dot notation (similar to yac grammar).

Feature structure unification works similar to classical unification – the same algorithm works but there is no equivalent for an arity of nodes. Type unification is used when unifying nodes. Outgoing edges with the same label are unified by pushing on the stack σ ending nodes.

NOTE A collection of all course notes are located in the `Misc` section on the website as a single PDF. Additionally, a sample examination is present there (from last year).

NOTE General rule for Bayesian Networks is, for marginalization, use summation. For completion, use maximization.