Contents

1 NLP with Deep Learning **2**

1.1 Word Vector Representations (Lec 2) 3

1.2 GloVe (Lec 3) 6

2 Speech and Language Processing **9**

2.1 Introduction (Ch. 1 2nd Ed.) 10

2.2 Morphology (Ch. 3 2nd Ed.) 11

2.3 N-Grams (Ch. 6 2nd Ed.) 12

2.4 Naive Bayes and Sentiment (Ch. 6 3rd Ed.) 14

2.5 Hidden Markov Models (Ch. 9 3rd Ed.) 16

2.6 POS Tagging (Ch. 10 3rd Ed.) 19

2.7 Formal Grammars (Ch. 11 3rd Ed.) 22

2.8 Vector Semantics (Ch. 15) 23

2.9 Semantics with Dense Vectors (Ch. 16) 26

2.10 Information Extraction (Ch. 21 3rd Ed) 29

NLP with Deep Learning

Contents

1.1 Word Vector Representations (Lec 2) 3
1.2 GloVe (Lec 3) 6

NLP with Deep Learning		**May 08**
	Word Vector Representations (Lec 2)	
Table of Contents Local		*Written by Brandon McKinzie*

Meaning of a word. Common answer is to use a *taxonomy* like WordNet that has hypernyms (is-a) relationships. Problems with this discrete representation: misses nuances, e.g. the words in a set of synonyms can actually have quite different meanings/connotations. Furthermore, viewing words as atomic symbols is a bit like using one-hot vectors of words in a vocabulary space (inefficient).

Distributed representations. Want a way to encode word vectors such that two similar words have a similar structure/representation. The **distributional similarity-based**[1] approach represents words by means of its *neighbors* in the sentences in which it appears. You end up with a dense "vector for each word type, chosen so that it is good at predicting other words appearing in its context."

Skip-gram prediction. Given a word w_t at position t in a sentence, learn to predict [probability of] some number of surrounding words, given w_t. Standard minimization with negative log-likelihood:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m \leq j \leq m} \log \Pr(w_{t+j}|w_t) \tag{1}$$

$$\Pr(o \mid c) = \frac{e^{\boldsymbol{u}_o^T \boldsymbol{v}_c}}{\sum_{w=1}^{\text{vocab size}} e^{\boldsymbol{u}_w^T \boldsymbol{v}_c}} \tag{2}$$

I cannot believe this actually works.

where

- The params θ are the vector representation of the words (they are the *only* learnable parameters here).
- m is our radius/window size.
- o and c are indices into our vocabulary (somewhat inconsistent notation).
- Yes, they are using different vector representations for $\boldsymbol{u}$ (context words) and $\boldsymbol{v}$ (center words). I'm assuming one reason this is done is because it makes the model architecture simpler/easier to build.

Some subtleties:

[1]Note that this is distinct from the way "distributed" is meant in "distributed representation." In contrast, distributional similarity-based representations refers to the notion that you can describe the meaning of words by understanding the context in which they appear.

- Looks like e.g. $Pr(w_{t+j} \mid w_t)$ doesn't really care what the value of j is, it is just modeling the probability that it is *somewhere* in the context window. The w_t are one-hot vectors into the vocabulary. Standard tricks for simplifying the cross-entropy loss apply.
- Equation 2 should be interpreted as the probability that the oth word in our vocabulary occurs in the context window of the cth word in our vocabulary.
- The model architecture is *identical* to an autoencoder. However, the (big) difference is the training procedure and interpretation of the model parameters going "in" versus the parameters going "out".

Sentence Embeddings. It turns out that simply taking a weighted average of word vectors and doing some PCA/SVD is a competitive way of getting unsupervised word embeddings. Apparently it *beats* supervised learning with LSTMs (?!). The authors claim the theoretical explanation for this method lies in a latent variable generative model for sentences (of course). Approach:

Discussion based on paper by Arora et al., (2017).

1. Compute the weighted average of the word vectors in the sentence:

$$\frac{1}{N}\sum_{i}^{N}\frac{a}{a+p(\boldsymbol{w}_i)}\boldsymbol{w}_i \tag{3}$$

The authors call their weighted average the **Smooth Inverse Frequency (SIF)**.

   where $\boldsymbol{w}_i$ is the word vector for the ith word in the sentence, a is a parameter, and $p(\boldsymbol{w}_i)$ is the (estimated) word frequency [over the entire corpus].
2. Remove the projections of the average vectors on their first principal component ("common component removal") (y tho?).

1

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$
1: **for all** sentence s in $\mathcal{S}$ **do**
2: $\quad v_s \leftarrow \frac{1}{|s|}\sum_{w\in s}\frac{a}{a+p(w)}v_w$
3: **end for**
4: Compute the first principal component u of $\{v_s : s \in \mathcal{S}\}$
5: **for all** sentence s in $\mathcal{S}$ **do**
6: $\quad v_s \leftarrow v_s - uu^\top v_s$
7: **end for**

Further Reading.

- Learning representations by back-propagating errors (Rumelhard et al., 1986)
- A Neural Probabilistic Language Model (Bengio et al., 2003)
- NLP (almost) from Scratch (Collobert & Watson, 2008)
- Word2Vec (Miklov et al. 2013)

Skip-gram and negative sampling. Main idea:

- Split the loss function from last lecture into two (additive) terms corresponding to the numerator and denominator respectively (you've done this a trillion times).
- The second term is an expectation over all the words in your vocab space. That is huge, so instead we only use a subsample of size k (the negative samples).
- Interpretation: First term is maximizing $\Pr(o \mid c)$, the probability that the true outside word (given by index o) occurs given context (index) c. Second term is minimizing the probability of random words (the negative samples) occurring around the center (context) word given by c.

To sample the negative samples, draw from $P(w) = U(w)^{3/4}/Z$, where U is the **unigram distribution**.

GloVe (Global Vectors). Given some co-occurrence matrix we computed with previous methods, we can use the following GloVe loss function over all pairs of co-occurring words in our matrix:

$$J(\theta) = \sum_{i,j=1}^{W} f(P_{ij})(u_i^T v_j - \log P_{ij})^2 \tag{4}$$

where P_{ij} is computed simply from the counts of words i and j co-occurring (empirical probability) and f is some squashing function that really isn't discussed in this lecture (**TODO**).

Evaluating word vectors.

- **Word Vector Analogies**: Basically, determining if we can do standard analogy fill-in-the-blank problems: "*man [a] is to woman [b] as king [c] is to <blank>*" (if you answered "queen", you'd make a good AI). We can determine this using a standard cosine distance measure:

$$d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||} \tag{5}$$

Woah that is pretty neat. The solution is x_i = queen. $x_b - x_a$ is the vector pointing from man to woman, which encodes the type of similarity we are looking for with the other pair. Therefore, we take the vector to "king" and *add* the aforementioned difference vector – the resultant vector should point to "queen". Neat!

Derivation. Based on the descriptions in the original paper[2] We want to develop a model for learning word vectors.

1. The authors argue that "the appropriate starting point for word vector learning should be with ratios of co-occurrence probabilities rather than the probabilities themselves." The most general such model takes the form,

$$F(w_i, w_j, \widetilde{w}_k) = \frac{\Pr\left[\widetilde{w}_k \mid w_i\right]}{\Pr\left[\widetilde{w}_k \mid w_j\right]} \equiv \frac{P_{ik}}{P_{jk}} \quad \text{where all} \quad w \in \mathbb{R}^d \tag{6}$$

   and the tilde in $\widetilde{w}_k$ denotes that $\widetilde{w}_k$ is a *context* word vector, which are given a distinct space from the word vectors w_i and w_j being compared. We compute all P_{ik} via frequency counts over the corpus.
2. Now that we've specified the inputs and ratio of interest, we can start specifying some desirable constraints on the function F that we're trying to find. The authors speculate that, since vector spaces are linear structures, we should have F encode the information of the ratio in the vector space via vector differences:

$$F(w_i - w_j, \widetilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{7}$$

   which basically says "our representation of the word vectors should be s.t. the **relative** probability of some word $\widetilde{w}_k$ occurring in the context of a word w_i compared to it occurring in the context of a different word w_j can be captured by $w_i - w_j$ and $\widetilde{w}_k$ alone."
3. Next we notice that F maps arguments in $\mathbb{R}^d$ to a scalar in $\mathbb{R}$. The most straightforward way of doing so while maintaining the linear structure we are trying to capture is via a dot product:

$$F((w_i - w_j)^T \widetilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{8}$$

   Note that now $F : \mathbb{R} \mapsto \mathbb{R}$.
4. We want our model to be invariant under the exchanges $w \leftrightarrow \widetilde{w}$ and $X \leftrightarrow X^T$. We can restore this symmetry by first requiring that F be a **homomorphism**[3] between the groups $(\mathbb{R}, +)$ and $(\mathbb{R}_{>0}, \times)$ (in our case, negation and division would be better symbols, but it's equivalent). This requires that the following relation hold[4]

$$F(w_i^T \widetilde{w}_k - w_j^T \widetilde{w}_k) = \frac{F(w_i^T \widetilde{w}_k)}{F(w_j^T \widetilde{w}_k)} \tag{9}$$

   where I've grouped terms on the LHS to emphasize how this is the definition of homomorphism. The solution for this equation is that $F(\cdot) \equiv \exp(\cdot)$. Combining this realization with equations 8 and 9 yields,

$$F(w_i^T \widetilde{w}_k) = e^{w_i^T \widetilde{w}_k} = P_{ik} \tag{10}$$

$$\Rightarrow w_i^T \widetilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i) \tag{11}$$

[2]Pennington et al., "GloVe: Global Vectors for Word Representation."

[3]In more detail, $F : (\mathbb{R}, +) \mapsto (\mathbb{R}_{>0}, \times)$, which reads "the function F maps elements in $\mathbb{R}$ and any summation of elements in $\mathbb{R}$ to elements in $\mathbb{R}$ that are greater than zero or any product of positive elements in $\mathbb{R}$."

[4]Note that we do not need to know anything about the RHS of the equations above to state this relation. We write it by definition of homomorphism.

where X_{ik} is the number of times word k appears in the context of word i, and $X_i = \sum_k X_{ik}$ is the number of times any word appears in the context of i.

5. Restore symmetry under the exchanges $w \leftrightarrow \widetilde{w}$ and $X \leftrightarrow X^T$. We absorb X_i into a bias b_i for w_i since it is independent of k.

$$w_i^T \widetilde{w}_k + b_i + \widetilde{b}_k = \log(X_{ik}) \tag{12}$$

6. A main drawback to this model is that it weighs all co-occurrences equally, even those that happen rarely or never. The authors propose a new weighted least squares regression model, introducing a weighting function $f(X_{ij})$ into the cost function of our model:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \widetilde{w}_j + b_i + \widetilde{b}_j - \log X_{ij})^2 \tag{13}$$

Speech and Language Processing

Contents

2.1 Introduction (Ch. 1 2nd Ed.) 10
2.2 Morphology (Ch. 3 2nd Ed.) 11
2.3 N-Grams (Ch. 6 2nd Ed.) 12
2.4 Naive Bayes and Sentiment (Ch. 6 3rd Ed.) 14
2.5 Hidden Markov Models (Ch. 9 3rd Ed.) 16
2.6 POS Tagging (Ch. 10 3rd Ed.) 19
2.7 Formal Grammars (Ch. 11 3rd Ed.) 22
2.8 Vector Semantics (Ch. 15) 23
2.9 Semantics with Dense Vectors (Ch. 16) 26
2.10 Information Extraction (Ch. 21 3rd Ed) 29

Speech and Language Processing **July 10, 2017**

Introduction (Ch. 1 2nd Ed.)

Table of Contents Local *Written by Brandon McKinzie*

Overview. Going to rapidly jot down what seems most important from this chapter.

- **Morphology**: captures information about the shape and behavior of words in context (Ch. 2/3).
- **Syntax**: knowledge needed to order and group words together.
- **Lexical semantics**: knowledge of the meanings of individual words.
- **Compositional semantics**: knowledge of how these components (words) combine to form larger meanings.
- **Pragmatics**: the appropriate use of polite and indirect language.
- The knowledge of language needed to engage in complex language behavior can be separated into the following 6 distinct categories:
    1. Phonetics and Phonology – The study of linguistic sounds.
    2. Morphology – The study of the meaningful components of words.
    3. Syntax – The study of the structural relationships between words.
    4. Semantics – The study of meaning.
    5. Pragmatics – The study of how language is used to accomplish goals.
    6. Discourse – The study of linguistic units larger than a single utterance.
- Methods for **resolving ambiguity**: pos-tagging, word sense disambiguation, probabilistic parsing, and speech act interpretation.
- **Models and Algorithms**. Among the most important are **state space search** and **dynamic programming** algorithms.

Morphology (Ch. 3 2nd Ed.)

Written by Brandon McKinzie

English Morphology. Morphology is the study of the way words are built up from smaller meaning- bearing units, **morphemes**. A morpheme is often defined as the minimal meaning-bearing unit in a language[5]. The two classes of morphemes are **stems** (the "main" morpheme of the word) and **affixes** (the "additional" meanings of various kinds).

Affixes are further divided into prefixes (precede stem), suffixes (follow stem), circumfixes (do both), and infixes (inside the stem).

Two classes of ways to form words from morphemes: **inflection** and **derivation**. Inflection is the combination of a word stem with a grammatical morpheme, usually resulting in a word of the same class as the original stem, and usually filling some syntactic function like agreement. Derivation is the combination of a word stem with a grammatical morpheme, usually resulting in a word of a different class, often with a meaning hard to predict exactly.

[5]Examples: "fox" is its own morpheme, while "cats" consists of the morpheme "cat" and the morpheme "-s".

Counting Words. Most N-gram based systems deal with the *wordform*, meaning they treat words like "cats" and "cat" as distinct. However, we may want to treat the two as instances of a single abstract word, or **lemma**: a set of lexical forms having the same stem, the same major part of speech, and the same word-sense.

Simple (Unsmoothed) N-Grams. An N-gram is a N-1th order Markov model (because it looks N-1 steps in the past). Notation: the authors use the convention that $w_1^n \triangleq w_1, w_2, \dots, w_n$ to denote a sequence of n words. Given this, we can write the general equation for the N-gram approximation for the probability of the nth word ($n > N$) in a sentence:

$$\Pr\left[w_n \mid w_1^{n-1}\right] \approx \Pr\left[w_n \mid w_{n-N+1}^{n-1}\right] \tag{14}$$

for $N \geq 2$. We can compute these probabilities by simply counting:

$$\Pr\left[w_n \mid w_1^{n-1}\right] = \frac{C(w_{n-N+1}^{n-1} w_n)}{C(w_{n-N+1}^{n-1})} \tag{15}$$

where $C(\cdot)$ is the number of times the sequence, denoted as $\cdot$, occurred in the corpus.

Entropy. Denote the random variable of interest as $\mathbf{x}$ with probability function $p(\mathbf{x})$. The entropy of this random variable is:

$$H(\mathbf{x}) = -\sum_x p(\mathbf{x} = x) \log_2 p(\mathbf{x} = x) \tag{16}$$

which should be thought of as a lower bound on the number of bits it would take to encode a certain decision or piece of information in the optimal coding scheme. The value 2^H is the **perplexity**, which can be interpreted as the weighted average number of choices a random variable has to make.

Cross Entropy for Comparing Models. Useful when we don't know the actual probability distribution p that generated some data. Assume we have some model m that's an approximation of p. The cross-entropy of m on p is defined by:

$$H(p, m) = \lim_{n \to \infty} \frac{1}{n} \sum_{W \in L} p(w_1, \ldots, w_n) \log m(w_1, \ldots, w_n) \tag{17}$$

That is we draw sequences according to the probability distribution p, but sum the log of their probability according to m.

[3rd Ed.] [Quick Review]

Overview. This chapter is concerned with **text categorization**, the task of classifying an entire text by assigning it a label drawn from some set of labels. *Generative classifiers* like naive Bayes build a model of each class. Given an observation, they return the class most likely to have generated the observation. *Discriminative classifiers* like logistic regression instead learn what features from the input are most useful to discriminate between the different possible classes. Notation: we will assume we have a training set of N documents, each hand-labeled with some class: $\{(d_1, c_1), \ldots, (d_N, c_N)\}$.

Discriminative systems are often more accurate and hence more commonly used.

Naive Bayes. A multinomial[6] classifier with a naive assumption about how the features interact. We model a text document as a **bag of words**, meaning we store (1) the words that occurred and (2) their frequencies. It's a probabilistic classifier, meaning it estimates the label/class of a document d as

$$\begin{aligned} \hat{c} &= \arg\max_{c \in C} \Pr[c \mid d] && (19) \\ &= \arg\max_{c \in C} \frac{\Pr[d \mid c] \Pr[c]}{\Pr[d]} && (20) \\ &= \arg\max_{c \in C} \Pr[d \mid c] \Pr[c] && (21) \end{aligned}$$

Computing the class-conditional distribution (the likelihood) $\Pr[d \mid c]$ over all possible $d \in D$ is typically intractable; we must introduce some simplifying assumptions and use an approximation of it. Our assumptions in this section will be:

- **Bag-of-Words**: Assume that word position within a document is irrelevant. (Counts still matter)

---

[6] Rapid review: **multinomial distribution**. Let $\mathbf{x} = (x_1, \ldots, x_k)$ denote the result of an experiment with n independent trials ($n = \sum_i^k x_i$) and k possible outcomes for any given trial. i.e. x_i is the number of trials that had outcome i ($1 \leq i \leq k$). The pmf of this multinomial distribution, over all possible $\mathbf{x}$ constrained by $n = \sum_i^k x_i$, is:

$$\Pr[\mathbf{x} = (x_1, \ldots, x_k); n] = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \times \cdots \times p_k^{x_k} \quad (18)$$

where p_i is the probability of outcome i for any single trial.

- **Naive Bayes Assumption**: First, recall that d is typically modeled as a (random) vector consisting of features $f_1, \ldots, f_n$, each of which has an associated probability distribution $\Pr[f_i \mid c]$. The NB assumption is that the features are mutually independent given the class c:

$$\Pr[f_1, \ldots, f_n \mid c] = \Pr[f_1 \mid c] \cdots \Pr[f_n \mid c] \tag{22}$$

$$c_{NB} = \arg\max_{c \in C} \Pr[c] \prod_{f \in F} \Pr[f \mid c] \tag{23}$$

  where 23 is the final equation for the class chosen by the naive Bayes classifier.

In text classification we typically use the word at position i in the document as f_i, and move to log space to avoid underflow/increase speed:

$$c_{NB} = \arg\max_{c \in C} \log \Pr[c] + \sum_{i}^{len(d)} \log \Pr[w_i \mid c] \tag{24}$$

Classifiers that use a linear combination of the inputs to make a classification decision (e.g. NB, logistic regression) are called linear classifiers.

Training the NB Classifier. No real "training" in my opinion, just simple counting from the data:

$$\hat{P}[c] = \frac{N_c}{N_{docs}} \tag{25}$$

$$\hat{P}[w_i \mid c] = \frac{\text{count}(w_i, c) + 1}{\left(\sum_{w \in V} \text{count}(w, c)\right) + |V|} \tag{26}$$

The Laplace smoothing is added to avoid the occurrence of zero-probabilities in equation 23.

Optimizing for Sentiment Analysis.

- It often improves performance [for sentiment] to clip word counts in each document to 1 (“binary NB”).
- deal with *negations* in some way.
- Use sentiment lexicons, lists of words that are pre-annotated with positive or negative sentiment.

Hidden Markov Models (Ch. 9 3rd Ed.)

 Written by Brandon McKinzie

Overview. Here we will first go over the math behind HMMs: the **Viterbi**, **Forward**, and **Baum-Welch** (EM) algorithms for unsupervised or semi-supervised learning. Recall that a HMM is defined by specifying the set of N states Q, transition matrix A, sequence of T observations O, sequence of observation likelihoods B, and the initial/final states. They can be characterized by three fundamental problems:

1. **Likelihood**. Given an HMM $\lambda = (A, B)$ and observation sequence, compute the likelihood (prob. of the observations given the model). (**Forward**)
2. **Decoding**. Given an HMM $\lambda = (A, B)$ and observation sequence, discover the best hidden state sequence. (**Viterbi**)
3. **Learning**. Given an observation sequence and the set of states in the HMM, learn the HMM parameters A and B. (**Baum-Welch/Forward-Backward/EM**)

The Forward Algorithm. For likelihood computation. We want to compute the probability of some sequence of observations O, without knowing the sequence of hidden states (that emitted the observations) Q. In general, this can be expressed by summing over all possible hidden state sequences:

$$\Pr[O] = \sum_Q \Pr[Q] \Pr[O \mid Q] \tag{27}$$

However, for N hidden states and T observations, this summation involves N^T terms, which becomes intractable rather quickly. Instead, we can use the $\mathcal{O}(N^2 T)$ **Forward Algorithm**. The forward algorithm can be defined by initialization, recursion definition, and termination, shown respectively as follows:

$$\alpha_1(j) = a_{0j} b_j(o_1) \qquad 1 \le j \le N \tag{28}$$

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t) \quad 1 \le j \le N, 1 \le t \le T \tag{29}$$

$$\Pr[O \mid \lambda] = \alpha_T(q_F) = \sum_{i=1}^{N} \alpha_T(i) a_{iF} \tag{30}$$

Viterbi Algorithm. For decoding. Want the most likely hidden state sequence given observations. Let $v_t(j)$ represent the probability that we are in state j after t observations and passing through the most probable state sequence $q_0, q_1, \ldots, q_{t-1}$. Similar to the forward algorithm, we show the defining equations for the Viterbi algorithm below:

$$v_1(j) = a_{0j} b_j(o_1) \quad 1 \le j \le N \tag{31}$$

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(j) a_{ij} b_j(o_t) \tag{32}$$

$$P^* = v_T(q_F) = \max_{i=1}^{N} v_T(i) a_{iF} \tag{33}$$

N.B.: At each step, the best path up to that point can be found by taking the *argmax* instead of max.

Baum-Welch Algorithm. AKA forward-backward algorithm, a special case of the EM algorithm. Given an observation sequence O and the set of best possible states in the HMM, learn the HMM parameters A and B. First, we must define some new notation. The **backward probability** β is defined as:

$$\beta_t(i) \triangleq \Pr\left[o_{t+1}, \ldots, o_T \mid q_t = i, \lambda\right] \tag{34}$$

Remember $\lambda \equiv (A, B)$

As usual, we can compute its values inductively as follows:

$$\beta_T(i) = a_{iF} \qquad 1 \le i \le N \tag{35}$$

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \quad 1 \le i \le N, 1 \le t \le T \tag{36}$$

$$\Pr\left[O \mid \lambda\right] = \alpha_T(q_F) = \beta_1(q_0) \tag{37}$$

$$= \sum_{j=1}^{N} a_{0j} b_j(o_1) \beta_1(j) \tag{38}$$

We can use the forward and backward probabilities α and β to compute the transition probabilities a_{ij} and observation probabilities $b_i(o_t)$ from an observation sequence. The derivation steps are as follows:

1. **Estimating** $\hat{a}_{ij}$. Begin by defining quantities that will prove useful:

$$\xi_t(i,j) \triangleq \Pr\left[q_t = i, q_{t+1} = j \mid O, \lambda\right] \tag{39}$$

$$\widetilde{\xi}_t(i,j) \triangleq \Pr\left[q_t = i, q_{t+1} = j, O \mid \lambda\right] \tag{40}$$

$$= \alpha_t(i) a_{ij} b_j(t+1) \beta_{t+1}(j) \tag{41}$$

Remember, knowing the observation sequence does NOT give us the sequence of hidden states.

where you should be able to derive eq. 41 in your head using just logic. If you cannot, review before continuing. We can then derive $\xi_t(i,j)$ using basic definitions of conditional probability, combined with eq. 37. Finally, we estimate $\hat{a}_{ij}$ as the expected number of transitions $q_i \to q_j$ divided by the expected number of transitions from q_i total:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{k=1}^{N} \xi_t(i,k)} \tag{42}$$

2. **Estimating** $\hat{b}_j(v_k)$. We define our estimate as the expected number of times we are in q_j and emit observation v_k, divided by the expected number of times we are in state j. Similar to our approach for $\hat{a}_{ij}$ we define helper quantities for these values at a given timestep, then sum over them (all t) to obtain our estimate.

$$\gamma_t(j) \triangleq \Pr\left[q_t = j \mid O, \lambda\right] \tag{43}$$

$$= \frac{\Pr\left[q_t = j, O \mid \lambda\right]}{\Pr\left[O \mid \lambda\right]} \tag{44}$$

$$= \frac{\alpha_t(j)\beta_t(j)}{\Pr\left[O \mid \lambda\right]} \tag{45}$$

Thus, we obtain $\hat{b}_j(v_k)$ by summing over all timesteps where $o_t = v_k$, denoted as the set T_{v_k}, divided by the summation over all t regadless of o_t:

$$\hat{b}_j(v_k) = \frac{\sum_{t \in T_{v_k}} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{46}$$

At last we can finally define the **Forward-Backward Algorithm** as follows:

1. Initialize A and B.
2. **E-step**. Compute $\gamma_t(j)$ $(\forall t, j)$, and compute $\xi_t(i,j)$ $(\forall t, i, j)$.
3. **M-step**. Update all $\hat{a}_{ij}$ and $\hat{b}_j(v_k)$.
4. Upon convergence, return A and B.

Speech and Language Processing **July 30, 2017**

POS Tagging (Ch. 10 3rd Ed.)

Table of Contents Local *Written by Brandon McKinzie*

English Parts-of-Speech. POS are traditionally defined based on syntactic and morphological function, grouping words that have similar neighboring words or take similar affixes.

	Part of Speech	Definition	Properties
Open classes:	noun	people, places, things	occur with determiners, take possessives,
	verb	actions, processes	3rd-person-sg, progressive, past participle
	adjective	properties, qualities	
	adverb	modify verbs, adverbs, verb phrases	directional, locative, degree, manner, tem

Common nouns can be divided into *count* (e.g. goat/goats) and *mass* (e.g. snow) nouns.

Closed classes. POS with relatively fixed membership. Some of the most important in English are:

prepositions: on, under, over, near, by, at, from, to, with
determiners: a, an, the
pronouns: she, who, I, others
conjunctions: and, but, or, as, if, when
auxiliary verbs: can, may, should, are
particles: up, down, on, off, in, out, at, by
numerals: one, two, three, first, second, third

Some subtleties: the **particle** resembles a preposition or an adverb and is used in combination with a verb. An example case where "over" is a particle: "she turned the paper over." When a verb and a particle behave as a single syntactic and/or semantic unit, we call the combination a **phrasal verb**. Phrasal verbs cause widespread problems with NLP because they often behave as a semantic unit with a noncompositional meaning – one that is not predictable from the distinct meanings of the verb and the particle. Thus, "turn down" means something like "reject", "rule out" means "eliminate", "find out" is "discover", and "go on" is "continue".

HMM POS Tagging. Since we typically train on labeled data, we need only use the Viterbi algorithm for decoding[7]. In the POS case, we wish to find the sequence of n tags, $\hat{t}_1^n$, given the observation sequence of n words w_1^n.

$$\hat{t}_1^n = \arg\max_{t_1^n} \Pr\left[t_1^n \mid w_1^n\right] = \arg\max_{t_1^n} \Pr\left[w_1^n \mid t_1^n\right] \Pr\left[t_1^n\right] \tag{47}$$

where we've dropped the denominator after using Bayes' rule (since argmax is the same). HMM taggers made two further simplifying assumptions:

$$\Pr\left[w_1^n \mid t_1^n\right] \approx \prod_{i=1}^{n} \Pr\left[w_i \mid t_i\right] \tag{48}$$

$$\Pr\left[t_1^n\right] \approx \prod_{i=1}^{n} \Pr\left[t_i \mid t_{i-1}\right] \tag{49}$$

We can thus plug-in these values into eq. 47 to obtain the equation for $\hat{t}_1^n$.

$$\hat{t}_1^n = \arg\max_{t_1^n} \prod_{i=1}^{n} \Pr\left[w_i \mid t_i\right] \Pr\left[t_i \mid t_{i-1}\right] \tag{50}$$

$$= \arg\max_{t_1^n} \prod_{i=1}^{n} b_i(w_i) a_{i-1,i} \tag{51}$$

where I've written a "translated" version on the second line using the familiar syntax from the previous chapter. In practice, we can obtain quick estimates for the two probabilities on the RHS by taking counts/averages over our tagged training data. We then run through the Viterbi algorithm to find all the argmaxes over states for the most likely hidden state sequence.

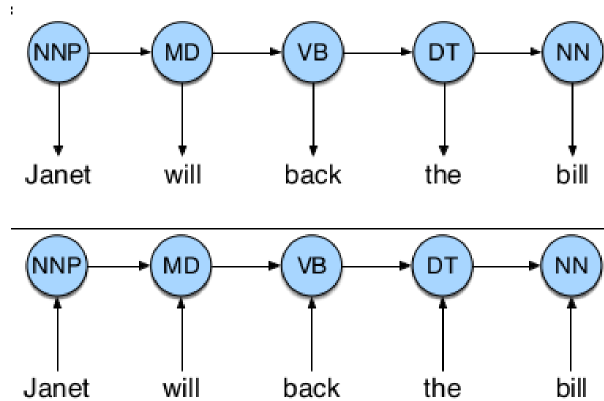
Maximum Entropy Markov Models (MEMMs). A sequence model adaptation of the MaxEnt (multinomial logistic regression) classifier[8]. Since HMMs are generative models, they decompose $\Pr[T \mid W]$ into $\Pr[W \mid T] \Pr[T]$ when computing the best tag sequence $\hat{T}$. Since MEMMs are discriminative, they compute/model $\Pr[T \mid W]$ directly:

$$\hat{T} = \arg\max_{T} \Pr\left[T \mid W\right] = \arg\max_{T} \prod_{i} \Pr\left[t_i \mid w_i, t_{i-1}\right] \tag{52}$$

Visually, we can think of the difference between HMMs and MEMMs via the direction of arrows, as illustrated below.

[7] Recall that decoding is the problem of finding the best hidden state sequence, given $\lambda = (A, B)$ and observation sequence O.

[8] Because it is based on logistic regression, the MEMM is a **discriminative sequence model**. By contrast, the HMM is a **generative sequence model**.



The top shows the HMM representation, while the bottom is MEMM.

> *The reason to use a discriminative sequence model is that discriminative models make it easier to incorporate a much wider variety of features.*

Bidirectionality. The one problem with the MEMM and HMM models as presented is that they are exclusively run left-to-right. MEMMs[9] have a weakness known as the **label bias problem**. Consider the tagged fragment: "*will/NN to/TO fight/VB*[10]." Even though the word "*will*" is followed by "*to*", which strongly suggests "*will*" is a NN, a MEMM will incorrectly label "*will*" as MD (modal verb). The culprit lies in the fact that $\Pr\left[\text{TO} \mid to, t_{will}\right]$ is essentially 1 $\underline{\text{regardless of } t_{will}}$; i.e. the fact that "*to*" must have the tag TO has **explained away** the presence of TO and so the model doesn't learn the importance of the previous NN tag for predicting TO.

One way to implement bidirectionality (and thus allowing e.g. the link between TO being available when tagging the NN) is to use a **Conditional Random Field** (CRF) model. However, CRFs are much more computationally expensive than MEMMs and don't work better for tagging.

[9] And other non-generative finite-state models based on next-state classifiers

[10] Note on the tag meanings: TO literally means "to". MD means "modal" and refers to modal verbs such as *will*, *shall*, etc.

Constituency and CFGs. Discovering the inventory of constituents present in the language. Groups of words like *noun phrases* or *prepositional phrases* can be thought of as single units which can only be placed within certain parts of a sentence.

The most widely used formal system for modeling constituent structure in English is the **Context-Free Grammar**[11]. A CFG consists of a set of **productions** (rules), e.g.

$$\text{NP} \longrightarrow \text{Det Nominal} \tag{53}$$

$$\text{NP} \longrightarrow \text{ProperNoun} \tag{54}$$

$$\text{Nominal} \longrightarrow \text{Noun} \mid \text{Nominal Noun} \tag{55}$$

where the arrow is to be read "is composed of" or "consists of."

The sequence of rule expansions going from left to right is called a **derivation** of the string of words, commonly represented by a **parse tree**. The formal language defined by a CFG is the set of strings that are derivable from the designated **start symbol**.

[11] Also called Phrase-Structure Grammars. Equiv formalism as Backus-Naur Form (BNF)

Words and Vectors. Vector models are generally based on a **co-occurrence**, an example of which is a **term-document matrix**: Each row is identified by a word, and each column a document. A given cell value is the number of times the assoc. word occurred in the assoc. document. Can also view each column as a document vector.

Information Retrieval: task of finding document d, from D docs total, that best matches a query q.

For individual word vectors, however, it is most common to instead use a **term-term** matrix[12], in which columns are also identified by individual words. Now, cell values are the number of times the row (target) word and the column (context) word co-occur in some context in some training corpus. The context is most commonly a window around the row/target word, meaning a cell gives the number of times the column word occurs in a window of $\pm N$ words from the row word.

- **Q**: What about the co-occurrence of a word with itself (row i, col i)?
  - **A**: It is included, yes. Source: "The size of the window ... is generally between 1 and 8 words on each side of the target word (*for a total context of 3-17 words*)."
- **Q**: Why is the size of each vector generally $|V|$ (vocab size)? Shouldn't this vary substantially with window and corpus size?
  - **A**: idk

(**TODO**: revisit end of page 5 in my pdf of this)

Pointwise Mutual Information (PMI). Motivation: raw frequencies in a co-occurrence matrix aren't that great, and words like "the" (which aren't useful and occur everywhere) can really skew things. *The best weighting or measure of association between words should tell us how much more often than chance the two words co-occur.* PMI is such a measure.

[Mutual Information] $$I(X,Y) = \sum_x \sum_y P(X=x, Y=y) \mathrm{PMI}(x,y) \tag{56}$$

[PMI] $$\mathrm{PMI}(x,y) = \ln \frac{P(x,y)}{P(x)P(y)} \tag{57}$$

which can be applied for our specific use case as $\mathrm{PMI}(w,c) = \ln \frac{P(w,c)}{P(w)P(c)}$. The interpretation is simple: the denominator tells the joint probability of the given target word w occurring with context word c if they were independent of each other, while the numerator tells us how often we observed the two words together (assuming we compute probability by using the MLE).

[12] Also called the word-word or term-context matrix

Therefore, the ratio gives us how an estimate of how much more the target and feature co-occur than we expect by chance[13]. Most people use **Positive PMI**, which is just $\max(0, \text{PMI})$. We can compute a **PPMI matrix** (to replace our co-occurrence matrix), where PPMI_{ij} gives the PPMI value of word w_i with context c_j. The authors show a few formulas which is really distracting since all we actually need is the counts $f_{ij} = \text{counts}(w_i, c_j)$, and from there we can use basic probability and Bayes rule to get the PPMI formula.

- **Q**: Explain why the following is true: very rare words tend to have very high PMI values.
  - **A**: hi
- **Q**: What is the range of α used in PPMI_α? What is the intuition behind doing this?
  - **A**: For reference:

$$\text{PPMI}_\alpha(w, c) = \max\left(\ln \frac{P(w,c)}{P(w)P_\alpha(c)}, 0\right) \tag{58}$$

$$P_\alpha(c) = \frac{\text{count}(c)^\alpha}{\sum_{c'} \text{count}(c')^\alpha} \tag{59}$$

Although there are better methods than PPMI for weighted co-occurrence matrices, most notably **TF-IDF**, things like tf-idf are <u>not</u> generally used for measuring *word similarity*. For that, PPMI and significance-testing metrics like t-test and likelihood-ratio are more common. The **t-test** statistic, like PMI, measures how much more frequent the association is than chance.

$$t = \frac{\bar{x} - \mu}{\sqrt{s^2/N}} \tag{60}$$

$$\text{t-test}(a, b) = \frac{P(a,b) - P(a)P(b)}{\sqrt{P(a)P(b)}} \tag{61}$$

where $\bar{x}$ is the observed mean, while μ is the expected mean [under our null-hypothesis of independence].

Measuring Similarity. By far the most common similarity metric is the **cosine** of the angle between the vectors:

$$\text{cosine}(\boldsymbol{v}, \boldsymbol{w}) = \frac{\boldsymbol{v} \cdot \boldsymbol{w}}{|\boldsymbol{v}|\ |\boldsymbol{w}|} \tag{62}$$

Note that, since we've been defining vector elements as frequencies/PPMI values, they won't have negative elements, and thus our cosine similarities will be between 0 and 1 (not -1 and 1).

[13]Computing PMI this way can be problematic for word pairs with small probability, especially if we have a small corpus. Recognize that PMI should never really be negative, but in practice this happens for such cases

Alternatives to cosine:

- **Jaccard measure**: Described as "weighted number of overlapping features, normalized", but looks like a silly hack in my opinion:

$$\text{sim}_{Jac}(\boldsymbol{v}, \boldsymbol{w}) = \frac{\sum_{i=1}^{N} \min(\boldsymbol{v}_i, \boldsymbol{w}_i)}{\sum_{i=1}^{N} \max(\boldsymbol{v}_i, \boldsymbol{w}_i)} \tag{63}$$

- **Dice measure**: Another hack. This displeases me.

$$\frac{2 \times \sum_{i=1}^{N} \min(\boldsymbol{v}_i, \boldsymbol{w}_i)}{\sum_{i=1}^{N} (\boldsymbol{v}_i + \boldsymbol{w}_i)} \tag{64}$$

- **Jensen-Shannon Divergence**: An alternative to the KL-divergence[14], which represents the divergence of each distribution from the mean of the two:

$$\text{sim}_{JS}(\boldsymbol{v}||\boldsymbol{w}) = D\left(\boldsymbol{v}\middle|\middle|\frac{\boldsymbol{v}+\boldsymbol{w}}{2}\right) + D\left(\boldsymbol{w}\middle|\middle|\frac{\boldsymbol{v}+\boldsymbol{w}}{2}\right) \tag{66}$$

[14] Idea:if two vectors, $\boldsymbol{v}$ and $\boldsymbol{w}$, each express a probability distribution (their values sum to one), then they are are similar to the extent that these probability distributions are similar. The basis of comparing two probability distributions P and Q is the **Kullback-Leibler** divergence or relative entropy, defined as:

$$D(P||Q) = \sum_{x} P(x) \log \frac{P(x)}{Q(x)} \tag{65}$$

Overview. This chapter introduces three methods for generating short, dense vectors: (1) dimensionality reduction like SVD, (2) neural networks like skip-gram or CBOW, and (3) Brown clustering.

Dense Vectors via SVD. Method for finding more important dimensions of a dataset, "important" defined as dimensions wherein the data most varies. First applied (for language) for generating embeddings from term-document matrices in a model called **Latent Semantic Analysis** (LSA). LSA is just SVD on a $|V| \times c$ term-document matrix $\boldsymbol{X}$, factorized into $\boldsymbol{W\Sigma C}^T$. By using only the top $k < m$ dimensions of these three matrices, the product becomes a least-squares approx. to the original X. It also gives us the reduced $|V| \times k$ matrix $\boldsymbol{W}_k$, where each row (word) is a k-dimensional vector (embedding). Voilà, we have our dense vectors!

$\boldsymbol{W} \in \mathbb{R}^{|V| \times m}$

$\boldsymbol{\Sigma} \in \mathbb{R}^{m \times m}$

$\boldsymbol{C}^T \in \mathbb{R}^{m \times c}$

Note that LSA implementations typically use a particular weighting of each cell in the term-document matrix called the **local** and **global** weights.

$$\text{[local]} \qquad \log \mathrm{f}(i,j) + 1 \tag{67}$$

$$\text{[global]} \qquad 1 + \frac{\sum_j p(i,j) \log p(i,j)}{\log D} \tag{68}$$

$f(i,j)$ is the raw frequency of word i in context j. D is number of docs.

For the case of a word-word matrix, it is common to use PPMI weighting.

Skip-Gram and CBOW. Neural models learn an embedding by starting with a random vector and then iteratively shifting a word's embeddings to be more like the embeddings of neighboring words, and less like the embeddings of words that don't occur nearby[15] Word2vec, for example, learns embeddings by training to predict neighboring words[16].

- **Skip-Gram**: Learns two embeddings for each word w: the **word embedding** v (within matrix $\boldsymbol{W}$) and **context embedding** c (within matrix $\boldsymbol{C}$). Visually:

$$\boldsymbol{W} = \begin{pmatrix} \boldsymbol{v}_0^T \\ \boldsymbol{v}_1^T \\ \vdots \\ \boldsymbol{v}_{|V|}^T \end{pmatrix} \qquad \boldsymbol{C} = \begin{pmatrix} \boldsymbol{c}_0 & \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_{|V|} \end{pmatrix} \tag{69}$$

[15] Why? Why is this a sensible assumption? I see no reason a priori why it ought to be true.

[16] Note that the prediction task is not the goal – it just happens to result in good word embeddings. Hacky.

For a context window of $L = 2$, and at a given word $\boldsymbol{v}^{(t)}$ inside the corpus[17], our goal is to predict the context [words] denoted as $\left[\, \boldsymbol{c}^{(t-2)}, \boldsymbol{c}^{(t-1)}, \boldsymbol{c}^{(t+1)}, \boldsymbol{c}^{(t+2)} \,\right]$.

- Example: Consider one of the context words, say $\boldsymbol{c}^{(t+1)} \triangleq \boldsymbol{c}_k$, where we also assume it's the kth word in our vocab. Also assume that our target word $\boldsymbol{v}^{(t)} \triangleq \boldsymbol{v}_j$ is the jth word in our vocab.
- Our task is to compute $\Pr\left[\boldsymbol{c}_k \mid \boldsymbol{v}_j\right]$. We do this with a softmax:

$$\Pr\left[\boldsymbol{c}_k \mid \boldsymbol{v}_j\right] = \frac{e^{\boldsymbol{c}_k^T \boldsymbol{v}_j}}{\sum_{i \in |V|} e^{\boldsymbol{c}_i^T \boldsymbol{v}_j}} \tag{70}$$

- **CBOW**: Continuous bag of words. Basically the mirror-image of skip-gram. Goal is to predict current word $\boldsymbol{v}^{(t)}$ from the context window of 2L words $\left[\, \boldsymbol{c}^{(t-2)}, \boldsymbol{c}^{(t-1)}, \boldsymbol{c}^{(t+1)}, \boldsymbol{c}^{(t+2)} \,\right]$.

As usual, the denominator of the softmax is computationally expensive, and usually we approximate it with **negative sampling**.

> *In the training phase, the algorithm walks through the corpus, at each target word choosing the surrounding context words as positive examples, and for each positive example also choosing k noise samples or negative samples: non-neighbor words. The goal will be to move the embeddings toward the neighbor words and away from the noise words.*

Example: Suppose we come along the following window (in "[]") (L=2) in our corpus:

```
lemon, a [tablespoon of apricot preserves or] jam
```

Ultimately, we want dot products, $\boldsymbol{c}_i \cdot \mathit{vector}(\text{"apricot"})$, to be high for all four of the context words $\boldsymbol{c}_i$. We do negative sampling by sampling k random noise words according to their [unigram] frequency. So here, for e.g. $k = 2$, this would amount to 8 noise words, 2 for each context word. We want the dot products between "apricot" and these noise words to be low. For a given single context-word pair (w, c), our training objective is to maximize:

$$\log \sigma(c \cdot w) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim p(w)} \left[\log \sigma(-w_i \cdot w)\right] \tag{71}$$

In practice, common to use $p^{3/4}(w)$ instead of $p(w)$

Again, the above is for a single context-target word-pair and, accordingly, the summation is only over $k = 2$ (for our example). Don't try to split the expectation into a summation or anything – just view it as an expected value. To iteratively shift parameters, we use an optimizer like SGD.

[17] Note that, technically, the position t of $\boldsymbol{v}^{(t)}$ is irrelevant for our computation; we are predicting those words based on which word $\boldsymbol{v}^{(t)}$ is in the vocabulary, not it's position in the corpus.

The actual model architecture is a typical neural net, progressing as follows:

$$
\begin{aligned}
\text{“apricot”} \;&\rightarrow\; \boldsymbol{w}^{\text{one-hot}} = \begin{bmatrix} 0 \; 0 \cdots 1 \cdots 0 \end{bmatrix} & (72)\\
&\rightarrow\; \boldsymbol{h} = \boldsymbol{W}^T \boldsymbol{w}^{\text{one-hot}} & (73)\\
&\rightarrow\; \boldsymbol{o} = \boldsymbol{C}^T \boldsymbol{h} = \begin{bmatrix} \boldsymbol{c}_0^T \boldsymbol{h}, \; \boldsymbol{c}_1^T \boldsymbol{h}, \; \cdots \; \boldsymbol{c}_{|V|}^T \boldsymbol{h} \end{bmatrix}^T & (74)\\
&\rightarrow\; \boldsymbol{y} = \text{softmax}(\boldsymbol{o}) = \begin{bmatrix} \Pr[\boldsymbol{c}_0 | \boldsymbol{h}], \; \Pr[\boldsymbol{c}_2 | \boldsymbol{h}], \; \cdots \; \Pr\big[\boldsymbol{c}_{|V|} | \boldsymbol{h}\big] \end{bmatrix}^T & (75)\\
& & (76)
\end{aligned}
$$

Brown Clustering. An agglomerative clustering algorithm for deriving vector representations of words by clustering words based on their associations with the preceding or following words. Makes use of the **class-based language model** (CBLM), wherein each word w belongs to some class $c \in C$ via the probability $P(w \mid c)$. CBLMs define

$$
\begin{aligned}
P(w_i \mid w_{i-1}) &= P(c_i \mid c_{i-1}) P(w_i \mid c_i) & (77)\\
P(\text{corpus} \mid C) &= \prod_{i-1}^{n} P(w_i \mid w_{i-1}) & (78)
\end{aligned}
$$

A naive and extremely inefficient version of Brown clustering, a hierarchical clustering algorithm, is as follows:

1. Each word is initially assigned to its own cluster.
2. For each cluster pair $(c_i, c_{j \neq i})$, compute the value of eq 78 that would result from merging c_i and c_j into a single cluster. The pair whose merger results in the smallest decrease in eq 78 is merged.
3. Clustering proceeds until all words are in one big cluster.

This process builds a binary tree from the bottom-up, and the binary string corresp. to a word's traversal from leaf-to-root is its representation.

Overview. The first step in most IE tasks is **named entity recognition** (NER). Next we can do **relation extraction**: finding and classifying semantic relations among the entities, e.g. "spouse-of." **Event extraction** is finding the events in which the entities participate, and **event coreference** for figuring out which event mentions actually refer to the same event. It's also common to extract dates/times (temporal expression) and perform **temporal expression normalization** to map them onto specific calendar dates. Finally, we can do **template filling**: finding recurring/stereotypical situations in documents and filling the template slots with appropriate material.

Named Entity Recognition. Standard algorithm is word-by-word sequence labeling task by a MEMM or CRF, trained to label tokens with tags indicating the presence of particular kinds of NEs. It is common to label with **BIO tagging**, for beginning, inside, and outside of entities. If we have n unique entity types, then we'd have $2n + 1$ BIO tags[18]. A helpful illustration is shown below:

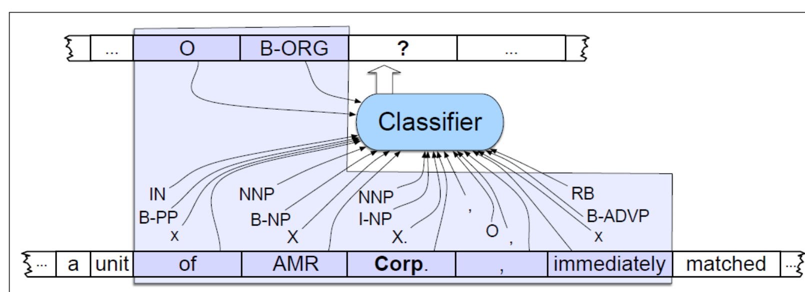


Figure 21.7 Named entity recognition as sequence labeling. The features available to the classifier during training and classification are those in the boxed area.

Here we see a classifier determining the label for *Corp.* with a context window of size 2 and various features shown in the boxed region. For evaluation of NER, we typically use the familiar **recall**, **precision**, and **F1 measure**.

[18] $2n$ for B-<NE> and I-<NE>, with +1 for the blanket O tag (not any of our NEs)

Relation Extraction. The four main algorithm classes used are (1) hand-written patterns, (2) supervised ML, (3) semi-supervised, and (4) unsupervised. Terminology:

- **Infobox**: structured tables associated with certain articles/topics/etc. For example, the Wikipedia infobox for Stanford includes structured facts like state = 'California'.
- **Resource Description Framework (RDF)**: a metalanguage of RDF triples, tuples of (entity, relation, entity), called a subject-predicate-object expression. For example: (Golden Gate Park, location, San Francisco).
- **hypernym**: the "is-a" relation.
- **hyponym**: the "kind-of" relation. *Gelidium is a kind of red algae.*

Overview of the four algorithm classes:

1. **Patterns**. Consider a sentence that has the following form:

   $$NP_0 \textit{ such as } NP_1\{, NP_2, \ldots, (and|or) NP_i\}, i \geq 1$$

   also known as a *lexico-syntactic pattern*, which implies $\forall NP_i, i \geq 1$,hyponym(NP_i, NP_0)[19]. Patterns typically have high precision but low-recall.
2. **Supervised**. The general approach for finding relations in a given sequence of words is the following:
   (a) Find all pairs of named entities in the sequence (typically a single sentence).
   (b) For each pair, use a trained binary classifier to predict whether or not the entities in the pair are indeed related.
   (c) If related, use a classifier trained to predict the relation given the entity-pair.

   As with NER, **the most important step** in this process is to identify useful surface features that will be useful for relation classification, including word features, NER features, syntactic paths (chunk seqs, constituent paths, dependency-tree paths), and more.
3. **Semi-supervised** via bootstrapping. Suppose we have a few high-precision **seed patterns** (or seed tuples)[20]. **Bootstrapping** proceeds by taking the entities in the seed pair, and then finding sentences (on the web, or whatever dataset we are using) that contain both entities. From all such sentences, we extract and generalize the context around the entities to learn new patterns.

```
function BOOTSTRAP(Relation R) returns new relation tuples

  tuples ← Gather a set of seed tuples that have relation R
  iterate
    sentences ← find sentences that contain entities in seeds
    patterns ← generalize the context between and around entities in sentences
    newpairs ← use patterns to grep for more tuples
    newpairs ← newpairs with high confidence
    tuples ← tuples + newpairs
  return tuples
```

Figure 21.14 Bootstrapping from seed entity pairs to learn relations.

[19]Here, hyponym(A, B) means "A is a kind-of (hyponym) of B."

[20]seed tuples are tuples of the general form (M1, M2) where M1 and M2 are each specific named entities we know have the relation of interest R.

4. **Unsupervised**. The Re Verb system extracts a relation from a sentence s in 4 steps:

    1. Run a part-of-speech tagger and entity chunker over s
    2. For each verb in s, find the longest sequence of words w that start with a verb and satisfy syntactic and lexical constraints, merging adjacent matches.
    3. For each phrase w, find the nearest noun phrase x to the left which is not a relative pronoun, wh-word or existential "there". Find the nearest noun phrase y to the right.
    4. Assign confidence c to the relation $r = (x, w, y)$ using a confidence classifier and return it.

Event Extraction. An event mention is any expression denoting an event or state that can be assigned to a particular point, or interval, in time. Note that this is quite different than the colloquial usage of the word "event," you should think of the two as distinct. Here, most event mentions correspond to verbs, and most verbs introduce events. Event extraction is typically modeled via ML, detecting events via sequence models with BIO tagging, and assigning event classes/attributes with multi-class classifiers.

Template Filling. The task is creation of one template for each event in the input documents, with the slots filled with text from the document. For example, an event could be "Fare-Raise Attempt" with corresponding template (slots to be filled) "(<Lead Airline>, <Amount>, <Effective Date>, <Follower>)". This is generally modeled by training two separate supervised systems:

1. **Template recognition**. Trained to determine if template T is present in sentence S. Here, "present" means there is a sequence within the sentence that could be used to fill a slot within template T.
2. **Role-filler extraction**. Trained to detect each role (slot-name), e.g. "Lead Airline".