FNLP - Week 4: Logistic Regression, Morphological Parsing & POS Tagging

Antonio León Villares

Feburary 2022

Contents

1 Text Classification with Logistic Regression **2**
1.1 Comparing NB and Logistic Regression for Classification . . . 2
1.2 Classifying with Logistic Regression . . . 2
1.2.1 Binary Classification with Logistic Regression . . . 2
1.2.2 Multinomial Logistic Regression . . . 3
1.2.3 Features in Logistic Regression . . . 4
1.3 Training Logistic Regression . . . 7
1.3.1 Loss Function . . . 7
1.3.2 Gradient Descent . . . 7
1.3.3 Computing the Gradient for CMLE (TODO if time, in lecture notes) . . . 8
1.4 Evaluating Logistic Regression . . . 8

2 Morphological Parsing **9**
2.1 Linguistics: Morphemes, and Building Words . . . 9
2.2 Morphological Parsing . . . 11
2.3 Finite State Transducers . . . 12

3 POS Tagging **16**
3.1 Markov Chains . . . 16
3.2 Hidden Markov Models . . . 17
3.3 POS Tagging . . . 18
3.4 HMMs for POS Tagging . . . 19
3.5 The Viterbi Algorithm . . . 21
3.5.1 Worked Example: Viterbi Algorithm . . . 24
3.6 The Forward Algorithm . . . 27
3.7 HMMs for Unsupervised Estimation . . . 29

1 Text Classification with Logistic Regression

1.1 Comparing NB and Logistic Regression for Classification

- **What is logistic regression?**
  - a **binary** classifier
  - part of the **Maximum Entropy Classifiers**
- **What are the key difference between NB and logistic regression?**
  - LR doesn't assume **feature independence**
  - LR is **discriminative**: instead of learning $P(c \mid d)$ through Bayes Rule, it **models it directly**

1.2 Classifying with Logistic Regression

- **What are the components of LR?**
  - **Feature Representations**
    - each input is a **vector of features**:

$$\underline{x} = [x_1, x_2, \ldots, x_n]$$

    - these features are more varied than in NB (for example, it can be counts, text length, strings, etc ...)
  - **Classification Function**
    - LR uses the **sigmoid function** to model $P(y \mid \underline{x})$ (for binary classification)
    - we can generalise for **multiclass** classification using **softmax**
  - **Error Function**
    - used to train LR
    - includes **cross-entropy loss**, **conditional maximum likelihood estimation**
    - can include a **regularisation** term to prevent overfitting
  - **Learning Algorithm**
    - used to minimise the **error function**
    - generally **gradient descent** (either **stochastic gradient descent** or **batch gradient descent**)

1.2.1 Binary Classification with Logistic Regression

- **What are weights and biases in LR?**
  - these are the **model parameters**
- **How are weights and biases used in binary classification?**
  - we learn a **vector** of weights, and a single **bias** term
  - given an input vector of **features** $\underline{x} = [x_1, \ldots, x_n]$, the first step of LR involves computing:

$$z = \underline{w} \cdot \underline{x} + b = b + \sum_{i=1}^{n} w_i x_i$$

- to turn this into a probability, we use the **sigmoid** function:

$$\sigma(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

which maps $z \in (-\infty, \infty)$ to $\sigma(z) \in (0,1)$

- σ defines a probability distribution, since:

$$P(y=0) = \sigma(-x) = 1 - \sigma(x) = 1 - P(y=1)$$

- **How is classification performed with binary logistic regression?**
  - since:

  $$P(y=1 \mid x) = \sigma(\underline{w} \cdot \underline{x} + b)$$

  we pick $\hat{y} = 1$ if $\sigma(\underline{w} \cdot \underline{x} + b) > 0.5$, and $\hat{y} = 0$ otherwise

- **How can we use matrices to perform a batch of classifications?**
  - take all the feature vectors $\underline{x}^{(1)}, \dots, \underline{x}^{(m)}$, and place them into a matrix (as row vectors):

  $$\boldsymbol{X} = \begin{pmatrix} \underline{x}^{(1)} \\ \vdots \\ \underline{x}^{(m)} \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \dots & x_n^{(m)} \end{pmatrix}$$

  - define the **bias vector** as $\underline{b} \in \mathbb{R}^n$:

  $$\begin{pmatrix} b \\ b \\ \vdots \\ b \end{pmatrix}$$

  where b is the binary classification bias
  - then, we obtain a **prediction vector** $\underline{\hat{y}}$, where the ith component corresponds to the classification of $\underline{x}^{(i)}$:

  $$\underline{\hat{y}} = \boldsymbol{X}\underline{w} + \underline{b}$$

1.2.2 Multinomial Logistic Regression

- **How does LR change from binary to multinomial classification?**
  - we have a bunch of classes that can be predicted: $c_1, \dots, c_k$
  - instead of producing a single probability, we produce a **vector** $\underline{\hat{y}}$, where component i is the probability of input $\underline{x}$ being c_i:

  $$\hat{y}_i = P(y = c_i \mid \underline{x})$$

  - instead of the **sigmoid**, uses **softmax**
  - instead of a single **weight** vector and **bias** term, we now need to train a **weight** vector and **bias** term **for each class**

- **What is the softmax function?**

- consider an input vector:

$$\underline{z} = [z_1, \dots, z_n]$$

- the **softmax** functions takes in $\underline{z}$, and returns a vector $softmax(\underline{z})$, where the ith component is given by:

$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{n} exp(z_j)}$$

- because of the denominator, clearly this produces a probability distribution

- **What can we classify in multinomial LR?**
  - for each class $c_i, i \in [1, k]$, we define a weight vector $\underline{w}_i$ and a **bias** b_i
  - for an input $\underline{x}$, the probability of it being classified c_i is:

$$\hat{y}_i = P(y = c_i \mid \underline{x}) = softmax(\underline{w}_i \cdot \underline{x} + b_i) = \frac{exp(\underline{w}_i \cdot \underline{x} + b_i)}{\sum_{j=1}^{n} exp(\underline{w}_j \cdot \underline{x} + b_j)}$$

  - if we want the full probability vector $\underline{\hat{y}}$, we can define:
    * a **matrix of weights**:

$$\boldsymbol{W} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k1} & w_{k2} & \dots & w_{kn} \end{pmatrix}$$

    * a **bias** vector:

$$\underline{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_k \end{pmatrix}$$

    Then:

$$\underline{\hat{y}} = \boldsymbol{W}\underline{x} + \underline{b}$$

1.2.3 Features in Logistic Regression

- **What is the meaning of weights in terms of features?**
  - in **binary classification**, a weight indicates the importance of a feature in classification:
    * a **positive** weight indicates a feature is useful for classifying the positive feature ($y = 1$)
    * conversely, a **negative** weight indicates $y = 0$
    * the **magnitude** of the weight indicates the **importance**
  - in **multinomial classification**, since each class has its own weights, features indicate evidence **against** or **for** a specific class

- **Can we treat features as functions in multinomial logsitic regression?**
  - in NB, we saw that **features** came directly from the data
  - in LR, features are a **representation** of data
  - in LR, we can think of features as **functions** of **observations** and **data**

- **How can we treat features as functions?**

- currently, LR is defined by converting an observation x into a set of features $\underline{x}$.
  - we can modify this, by defining n functions f_i such that:

$$\underline{x} = [f_1(x), f_2(x), \ldots, f_n(x)]$$

  - that is, each f_i maps an observation to a feature
- each feature $f_i(x)$ has been defined to have a weight **which depends on a class**.
  - class c_k has a weight vector $\underline{w}_k$
  - the **weight** which interacts with $f_i(x)$ is $w_{k,i}$
- an alternative view to this (as presented in lectures) is to "flatten" this
  - instead of having different weights for different classes, just define a single vector of weights
  - to do this, we modify the way in which we think about features, so that they **also** depend on class:

$$f_i(x, c)$$

    where:

$$f_i(x, c) = \begin{cases} f_i(x), c = c_j \\ 0, c \neq c_j \end{cases}$$

  - this is better exemplified by:

$$\begin{aligned} f_1 &: \quad \texttt{contains('ski')} \ \& \ c = 1 \\ f_2 &: \quad \texttt{contains('ski')} \ \& \ c = 2 \\ f_3 &: \quad \texttt{contains('ski')} \ \& \ c = 3 \end{aligned}$$

    In the previous interpretation, we would have had $f_i(x) = x.contains(\text{"}ski\text{"})$, and then, for each class $c = 1, 2, 3$, we would have trained 3 different weights $w_{1,i}, w_{2,i}, w_{3,i}$.
  - in this new interpretation $f_i(x) = x.contains(\text{"}ski\text{"})$ gets split into the 3 features defined above f_1, f_2, f_3. We then define 3 different weights w_1, w_2, w_3 for each.
  - in the new interpretation, if we try to compute $P(y = 1 \mid x)$, then f_1 will be activated, so its weight w_1 will contribute towards classification, but we will have $f_2 = f_3 = 0$, so their weights won't contribute to classification
- under this interpretation, we can define:

$$\underline{x}(x, c) = [f_1(x, c), \ldots, f_n(x, c)]$$

  so that LR becomes:

$$P(c \mid x) = \frac{1}{Z} exp(\underline{w} \cdot \underline{x}(x, c)) = \frac{1}{Z} exp\left(\sum_{i=1}^{n} w_i f_i(x, c)\right)$$

$$Z = \sum_{c' \in C} exp(\underline{w} \cdot \underline{x}(x, c')) = \sum_{c' \in C} exp\left(\sum_{i=1}^{n} w_i f_i(x, c')\right)$$

  (for some reason in lectures they ignore the weight bias)

$$
\begin{array}{lll}
f_1: & \texttt{contains('ski')}\ \&\ c=1 & w_1 = 1.2 \\
f_2: & \texttt{contains('ski')}\ \&\ c=2 & w_2 = 2.3 \\
f_3: & \texttt{contains('ski')}\ \&\ c=3 & w_3 = -0.5 \\
f_4: & \texttt{link\_to('expedia.com')}\ \&\ c=1 & w_4 = 4.6 \\
f_5: & \texttt{link\_to('expedia.com')}\ \&\ c=2 & w_5 = -0.2 \\
f_6: & \texttt{link\_to('expedia.com')}\ \&\ c=3 & w_6 = 0.5 \\
f_7: & \texttt{num\_links}\ \&\ c=1 & w_7 = 0.0 \\
f_8: & \texttt{num\_links}\ \&\ c=2 & w_8 = 0.2 \\
f_9: & \texttt{num\_links}\ \&\ c=3 & w_9 = -0.1
\end{array}
$$

Figure 1: Consider the following features. We observe a document with the word "ski", and 6 outgoing links. For this document, the numerator of the probability for each class is:

$$\sum_i w_i f_i(x, c=1) = 1.2 + (0)6 = 1.2$$

$$\sum_i w_i f_i(x, c=2) = 2.3 + (0.2)6 = 3.5$$

$$\sum_i w_i f_i(x, c=3) = -0.5 + (-0.1)6 = -1.1$$

Notice, we don't need to compute the denominator, or even the exponential. The denominator is constant, and the exponential is **monotonic**. Hence, for classification all we really need to do is compute the dot product $\underline{w} \cdot \underline{x}(x, c)$.

- **What are feature templates?**
  - in practice, instead of defining **all** the features, we use **feature templates**
  - for example, if we want a feature to see if a document contains a word w, instead of explicitly writing $contains(aardvark), contains(america), etc...$, we would use:

    $$contains(w)\&c$$

    and apply the template for each possible word and class
  - typically a few templates, but 1000s of features

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon words $\in$ doc)	3
x_2	count(negative lexicon words $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

Figure 2: Possible features for a sentiment analyser.

$$x_1 = \begin{cases} 1 & \text{if ``} Case(w_i) = \text{Lower''} \\ 0 & \text{otherwise} \end{cases}$$

$$x_2 = \begin{cases} 1 & \text{if ``} w_i \in \text{AcronymDict''} \\ 0 & \text{otherwise} \end{cases}$$

$$x_3 = \begin{cases} 1 & \text{if ``} w_i = \text{St. \& } Case(w_{i-1}) = \text{Cap''} \\ 0 & \text{otherwise} \end{cases}$$

Figure 3: Possible features for a end of sentence detector.

1.3 Training Logistic Regression

1.3.1 Loss Function

- **What is a loss function?**
  - a function indicating how well our model performs
- **What is conditional maximum likelihood estimation?**
  - recall, LR is a **discriminative model**, which learns weights to model:

$$P(c \mid x)$$

  - in particular, we want to find $\underline{\hat{w}}$ such that:

$$\underline{\hat{w}} = \underset{\underline{w}}{argmax} \prod_j P(c^{(j)} \mid x^{(j)})$$

    where $c^{(j)}$ is the class associated with document $x^{(j)}$
  - taking the log won't change the weights (since logarithms are monotonic):

$$\underline{\hat{w}} = \underset{\underline{w}}{argmax} \sum_j \log\left(P(c^{(j)} \mid x^{(j)})\right)$$

- **What is categorical cross-entropy loss?**
  - an alternative loss function (which can be derived from the above):

$$L_{CE}(\hat{y}, y) = -\sum_j y^{(j)} \log(\hat{y}^{(j)}) = -\sum_j y^{(j)} \log(softmax(x^{(j)}))$$

    where $y^{(j)}$ is 1 when we correctly classify instance $x^{(j)}$

1.3.2 Gradient Descent

- Link for gradient of binary classification
- **What is gradient descent?**
  - we can't analytically obtain an optimal set of weights
  - use **gradient descent** to numerically approximate the weights

– initialise $\underline{w}^0$ randomly, and then we iterate the following until convergence:

$$\underline{w}^{t+1} = \underline{w}^t + \eta \nabla_{\underline{w}} \sum_{j=1}^{N} \log\left(P(c^{(j)} \mid x^{(j)})\right)$$

(technically this is **gradient ascent**, since we want to **maximise** the CMLE)

– the above is slow (we iterate through **all** training examples for a single gradient update), so alternatively:

  * **stochastic gradient descent**: update weights using a single training instance
  * **mini-batch gradient descent**: pick a random subset of the data, and adapt gradient after seeing the subset:

$$B = RandomSubset([1, \ldots, N])$$

$$\underline{w}^{t+1} = \underline{w}^t + \eta \nabla_{\underline{w}} \sum_{j \in B} \log\left(P(c^{(j)} \mid x^{(j)})\right)$$

1.3.3 Computing the Gradient for CMLE (TODO if time, in lecture notes)

We get that the gradient with respect to weight w_l, corresponding to feature f_l, which is active only when $c = k$, is:

$$\frac{d}{dw_l} \log(P(c^{(j)} \mid x^{(j)}) = (\mathcal{X}_k(x^{(j)}) - P(c = k \mid x^{(j)}))f_l(x^{(j)}, k)$$

When the classifier is confident of its prediction, $P(c = k \mid x^{(j)} \approx 1$ (if $k = c^{(j)}$), so the gradient will be close to zero.

1.4 Evaluating Logistic Regression

- **Can we express NB as logistic regression?**

  – yes, set the weights to the probability predicted by NB:

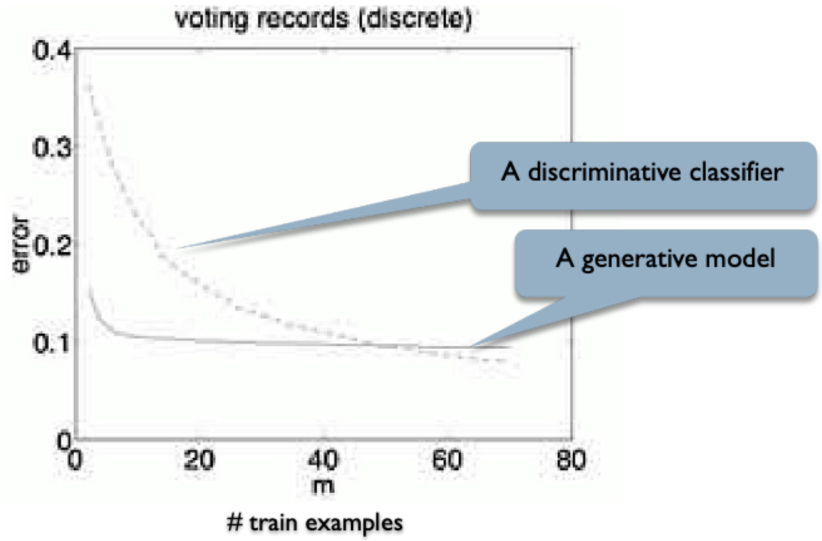
$$\begin{array}{lll}
f_1: & \texttt{contains('ski')} \ \& \ c=1 & w_1 = \log \hat{P}(\texttt{'ski'}|c=1) \\
f_2: & \texttt{contains('ski')} \ \& \ c=2 & w_2 = \log \hat{P}(\texttt{'ski'}|c=2) \\
f_3: & \texttt{contains('ski')} \ \& \ c=3 & w_3 = \log \hat{P}(\texttt{'ski'}|c=3) \\
f_4: & \texttt{contains('beach')} \ \& \ c=1 & w_4 = \log \hat{P}(\texttt{'beach'}|c=1) \\
f_5: & \texttt{contains('beach')} \ \& \ c=2 & w_5 = \log \hat{P}(\texttt{'beach'}|c=2) \\
f_6: & \texttt{contains('beach')} \ \& \ c=3 & w_6 = \log \hat{P}(\texttt{'beach'}|c=3) \\
f_7: & c=1 & w_7 = \log \hat{P}(c=1) \\
f_8: & c=2 & w_8 = \log \hat{P}(c=2) \\
f_9: & c=3 & w_9 = \log \hat{P}(c=3)
\end{array}$$

Figure 4: If the feature were independent, NB and LR will converge to the same solution, given sufficient training data.

- **How do generative and discriminative classifiers compare?**

  – as the number of **training** examples increases, **discriminative** models **outperform** the **generative** ones

– however, **generative** classifiers converge faster to **their optimal error**



- **What are some downsides of MaxEnt models (compared to NB)?**
  - **Difficulty**
    * NB easy to train (compute counts + normalisation)
    * LR: GD is expensive, need to compute probabilities $P(c^{(j)} \mid x^{(j)})$ **for each class**
  - **Robustness**
    * LR can learn to rely on a single, very frequent, predictive feature, which might not appear in testing - means such a feature will have very large weights compared to other features
    * NB relies on **all** features, so relevance of one doesn't take from others
    * NB more robust than **basic** LR when testing data has different distribution than training

2 Morphological Parsing

2.1 Linguistics: Morphemes, and Building Words

- **What is a morpheme?**
  - a **minimal, meaning-bearing** unit of a language
    * "foxes" = "fox" + "-es" (composed of 2 morphemes)
    * "jump" = "jump" (composed of a single morpheme)
- **What are morphological rules?**
  - rules defining the structure of words
    * "fish" is a null plural
    * the plural of "goose" is built by changing the vowel
  - **morphology** changes depending on the language
    * English has (simple) poor morphology

    - Turkish is **agglutinative**: can concatenate many morphemes together (to the point that sentences in English like "from your houses" can be expressed as a single word "evlerinizden")

- **What are stems and affixes?**
  - the types of **morphemes** used to construct words
  - **stems** are the **main** part of the word: it's what conveys meaning
  - **affixes** are morphemes added with grammatical purpose: they modify the main meaning provided by stems

- **What are the different types of affixes?**
  - depending on how an **affix** "joins" with a stem, it is a:
    - **prefix** (before)
      - "un-" + "buckle" = "unbuckle"
    - **suffix** (after)
      - "eat" + "-s" = "eats"
    - **infix** (middle)
    - **circumfix** (before and after)
      - "ge-" + "sagen" + "-t" = "gesagt"
  - words can have more than 1 affix (i.e "un-" + "believe" + "-able" + "-y" = "unbeliaevably")

- **What are the 4 ways to combine stems and affixes?**
  1. **Inflection**
     - combines a **stem** and an **affix** to produce a word in the **same grammatical category**
     - for example, *walk* → *walking* (verb), *agreement* → *agreements* (noun)
  2. **Derivation**
     - combines a **stem** and an **affix** to produce a word in a **different grammatical category**
     - for example, $different \rightarrow differentiate$ (*adjective* → *verb*), *computerise* → *computerisation* (*verb* → *noun*)
  3. **Compounding**
     - combine multiple **stems**
     - for example, *dog* + *house* → *doghouse*
  4. **Cliticisation**
     - "'ve" is a **clitic** in "I've"
     - "l'" is a **clitic** in "l'opera"

  What types of words can be inflected in English?
  - **nouns** have 2 inflections:
    - **plural**
      - for **regular** nouns, append the affix "-s" or "-es"
      - "-es" added for words ending in *s,z,sh,ch* and sometimes *x*; if a word ends in *y*, change for *ies* (i.e $butterfly \rightarrow butterflies$)
      - irregular nouns include *mouse* → *mice* and *ox* → *oxen*
    - **possesive**

        · add "'s" for regular singular nouns and plural nouns not ending in "-s"
        · add "'" after regular plural nouns, and some names ending in "-s" or "-z"
  - **verbs** are inflected for **person** and **tense**
    * for example "You read" (2nd person, present/past), "she reads" (3rd person, present), "she read" (3rd person past)
    * verbs can be **regular** (i.e *walk, inspect*) or **irregular** (i.e *be, hit, cut, eat, catch*)
    * verbs can be **main verbs** (*eat, sleep, impeach*), **modal verbs** (can, will, should) or **primary verbs** (*be, have, do*)
    * main and primary verbs can have inflectional endings; when **regular**, there are 4 forms (stem, -s form, -ing participle and past form/-ed participle)

- **How does English differ from other languages in inflection?**
  - **German** inflects nouns for number and case (nominative, genitive, dative and accusative)
  - **Spanish** inflects on gender
  - in **Luganda**, nouns have 10 genders

- **What is concatenative morphology?**
  - word construction based on **concatenating morphemes**
  - in **non-concatenative morphology**, morphemes are combined in different ways (for example, in **hebrew** they use **templactic morphology**; the **root** of a verb is 3 consonants which carry meaning; a **template** orders the consonants/vowels; in this way "lmd" means to "learn/study"; then:
    * $CaCaC \rightarrow lamad$ ("he studied")
    * $CiCeC \rightarrow limed$ ("he taught")
    * $CuCaC \rightarrow lumad$ ("he was taught")

2.2 Morphological Parsing

- **What is morphological parsing?**
  - breaking down a word in **surface form** into its **lexical form** (that is, its set of **component morphemes**)
  - in particular, its **stem** and the set of **affixes** carrying grammatical information

Surface Form	cats	walking	smoothest
Lexical Form	cat + N + PL	walk + V + PresPart	smooth + Adj + Sup

- **What is generation?**
  - the opposite of parsing: going from **lexical** to **surface** form:

$$fox + N + PL \rightarrow foxes$$

- **What are intermediate forms?**
  - form of words including **morphemes** before applying **orthographic rules**:

$$foxes \rightarrow fox\hat{}\,s\ \# \rightarrow fox + N + PL$$

- ˆ represents a **morpheme boundary**
- # represents a **word boundary**

- **Which issues can arise during morphological parsing?**
  - **irregular forms** (*goose* → *geese*)
  - accounting for **rules** (i.e plurals of words ending in s or z)
  - affix looking things (i.e **pro**tect)
  - **blocking** due to **semi-productive** morphological rules (i.e generally adding "-ful" creates an adjective, like *graceful*; but certain words have specific adjective forms $intelligence \to intelligent \not\to intelligenceful$)

- **Why is morphological parsing useful?**
  - prerequesite for **grammatical parsing**
  - **search engines** (if I search for "foxes", I'd be interested in articles containing "fox")
  - **spell-checking** (*sleeped* → *slept*)
  - makes POS tagging easier
  - easy to add/learn new words

- **Can we keep a corpus of words and their derivations instead of applying morphological parsing?**
  - potentially in English
  - impossible for **agglutinative** languages (too many possibilities)
  - impossible for languages like German, where noun compounding is very **productive**

- **Why are FSMs useful for morphological parsing?**
  - morphemes are "glued" in regular manner
  - this is independent of previous morphemes

2.3 Finite State Transducers

- **What is a nondeterministic finite state automaton?**
  - a FSM where each element of the vocabulary can have more than one (or none) arcs from each state
  - ε-NFA allow the empty string as state transitions

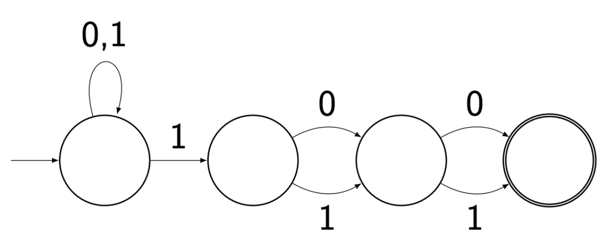


Figure 5: An NFA representing the regex $(0|1) * 1(0|1)^2$

- **What is a finite state transducer?**

– instead of simply **accepting** symbols, a FST **maps** between 2 sets of symbols

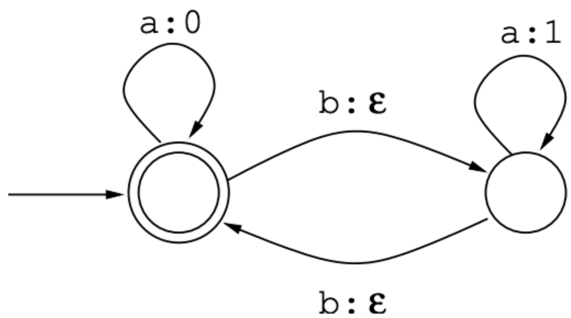


Figure 6: A FST mapping from $\{a, b\}$ to $\{0, 1\}$. For example, we get mappings like $abba \to 00$ or $aaabaaabb \to 000111$.

– this is useful for mapping between **surface**, **intermediate** and **lexical** forms

- **What are the 4 interpretations of a FST?**

  1. **Recogniser**: determine if string pair belongs to a language
  2. **Generator**: produce string pairs
  3. **Translator**: read input string, and produce translated output
  4. **Set Relator**: determine relation between sets

- **How can we describe FSTs mathematically?**

  1. Q - finite set of N states
  2. Σ - finite set, input alphabet
  3. Π - finite set, output alphabet
  4. $q_0 \in Q$ - start state
  5. $F \subseteq Q$ - set of final states
  6. $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Pi \cup \{\varepsilon\}) \times Q$ - transition relation between 2 states
  7. $\hat{\Delta} \subseteq Q \times \Sigma^* \times \Pi^* \times Q$ - many-step transition relation between 2 states (for example, $(q, x, y, q') \in \hat{\Delta}$ can represent $abba \to 00$)

- **What operations are FSTs closed under?**

  – **inversion**: switch input and output alphabets. Convert a **parser** into a **generator**
  – **composition**: chain FSTs

- **How can FSTs be constructed for morphological parsing?**

  1. **Lexical to Intermediate Form**: in this example, we consider an FST for **nominal number inflection** (i.e working with plurals). For example, performs:

  $$fox + N + PL \to fox\,\hat{}\,s\ \#$$

  and accounts for **irregular forms** ($goose \to geese$)

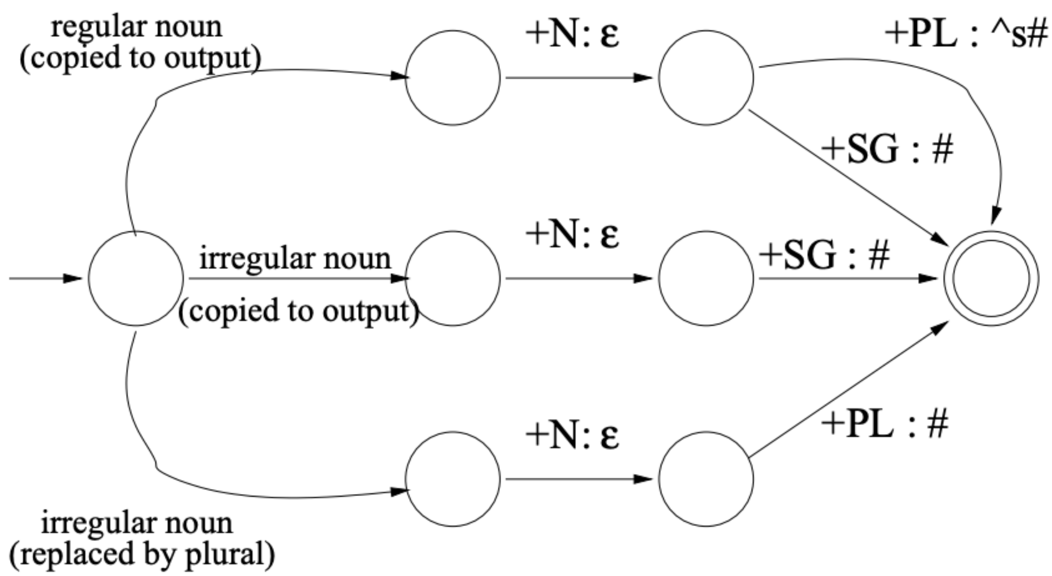


Figure 7: This FST works in the following way.
$(cat) = cat + N + SG \rightarrow cat + N + SG \rightarrow cat + SG \rightarrow cat\#$
$(cats) = cat + N + PL \rightarrow cat + N + PL \rightarrow cat + PL \rightarrow cat\hat{}s\ \#$
$(goose) = goose + N + SG \rightarrow goose + N + SG \rightarrow goose + SG \rightarrow goose\#$
$(geese) = goose + N + PL \rightarrow geese + N + PL \rightarrow geese + PL \rightarrow geese\#$
It is important to note that the transition $+PL :\hat{}s\#$ can be represented as 3 transitions, but easier in this way.

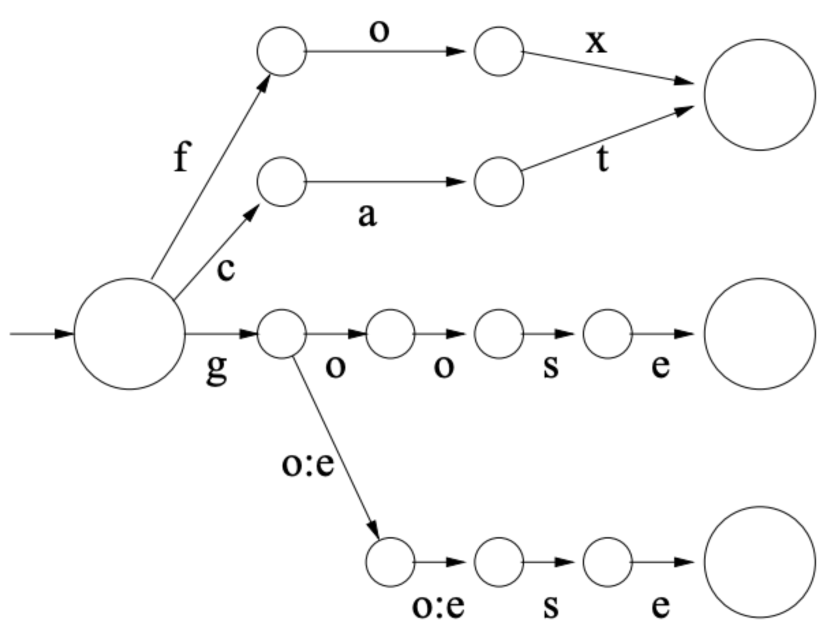


Figure 8: The FST used to copy the inputs.

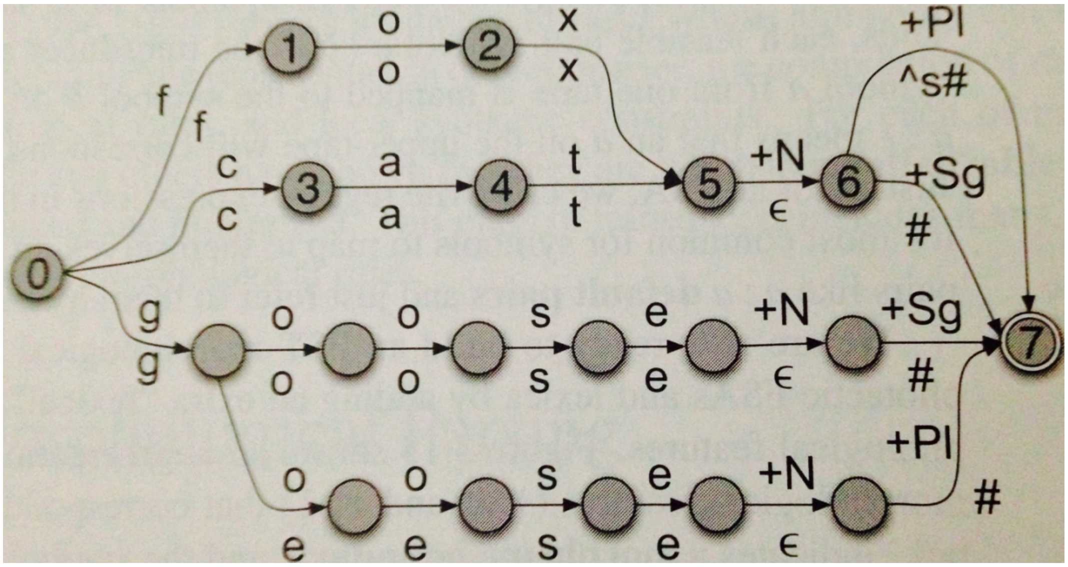


Figure 9: The full FST for nominal number inflection.

2. **From Intermediate to Surface Form**: this relies on applying orthographic rules like:
   - **E-insertion** (-es as plural for words ending with s,z,x, etc...)
   - **E-deletion** (remove -e before suffix starting with *i,e* (*love* $\rightarrow$ *loving*)
   - **Consonant Doubling**: s b,s,g,k,l,m,n,p,r,s,t,v doubled before suffix -ed, -ing (*beg* $\rightarrow$ *begged*)

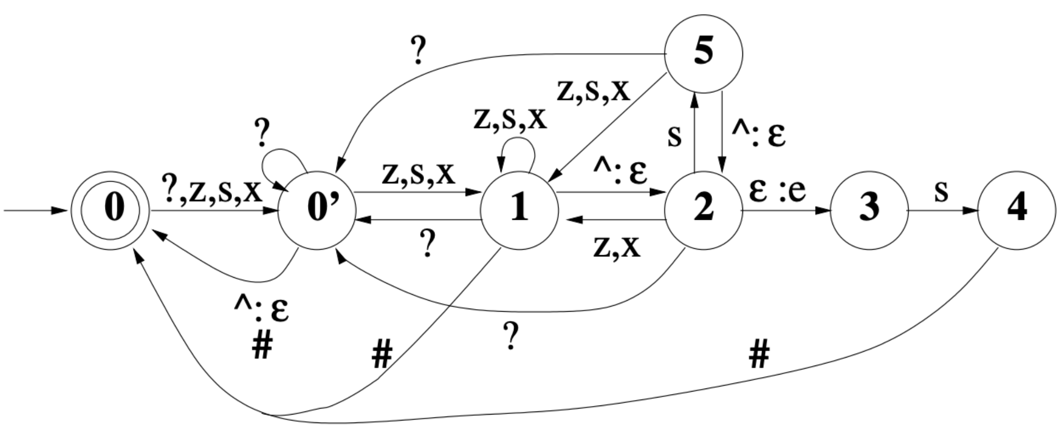


Figure 10: Simplified FST for E-insertion.
? stands for every symbol excluding $z, s, x,$' $\#$.
Converts $fox\hat{}s\ \# \rightarrow foxes$ and $ex\hat{}service\hat{}men\ \# \rightarrow exservicemen$

3. **Combining The Two**
   - for **generation**, compose *lexical* $\rightarrow$ *intermediate* $\rightarrow$ $surface$; even if FSTs are non-deterministic, this typically is deterministic (unique surface form given lexical form)
   - for **parsing** just reverse the above (there is ambiguity, so this is non-deterministic; nonetheless, can be solved later on)

- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{e} 3 \xrightarrow{s} 4$, output ass^s
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{e} 0' \xrightarrow{s} 1$, output ass^es
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{e} 0' \xrightarrow{s} 1$, output asses
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{\epsilon} 2 \xrightarrow{e} 3 \xrightarrow{s} 4$, output as^s^s
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{\epsilon} 2 \xrightarrow{e} 0' \xrightarrow{s} 1$, output as^s^es
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{e} 0' \xrightarrow{s} 1$, output as^ses
- Four of these can also be followed by $1 \xrightarrow{\epsilon} 2$ (output ^).

- **What is the Porter Stemmer?**
  - an efficient **lexicon-free** method of extracting the stem of a word:

$$ational \rightarrow ate \implies relational \rightarrow relate$$

  - still makes errors:

$$organisation \rightarrow organ$$

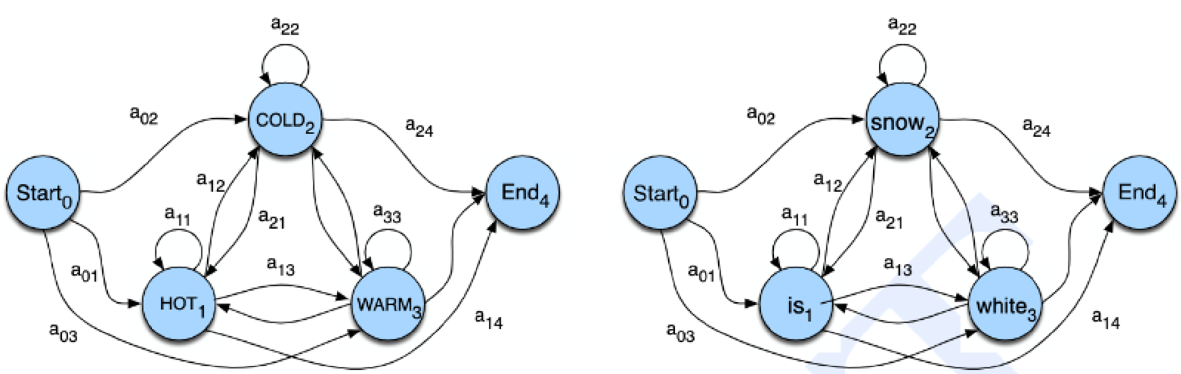
$$policy \rightarrow police$$

(errors derived from *computerisation* → *computer* and *juicy* → *juice*)

3 POS Tagging

3.1 Markov Chains

- **What is a Markov Chain?**
  - a type of **finite state automaton**
  - the **arcs** leaving a **node/state** have **probabilities** assigned
  - the probabilities indicate the likelihood of "jumping" between states
- **Why are Markov chains used?**
  - they allow us to **assign** probabilities to sequences of occurrences
  - for example, a weather sequence, or a sequence of words (this is a bigram model!)



- **How can we represent a Markov chain mathematically?**

1. A set of N states:
$$Q = \{q_1, q_2, \ldots, 2_N\}$$
2. A **transition probability matrix** A, where a_{ij} is the transition probability of moving from q_i to q_j and with:
$$\sum_{j=1} a_{ij} = 1, \forall i \in [1, N]$$
3. 2 special states, q_0, q_F not associated with observations (i.e the start/end of a sentence)

- **What is a first-order Markov chain?**
  - a **Markov chain** in which the probability of reaching a state depends only on the previous state:
$$P(q_i \mid q_1, \ldots, q_{i-1}) = P(q_i \mid q_{i-1})$$

3.2 Hidden Markov Models

- **What are Hidden Markov Models?**
  - **Hidden Markov Models** allow us to talk about **observed events**, which we think might be caused by **hidden, unobserved events**
    * **POS-Tagging**: we observe words, and infer POS tags
    * we see ice-cream consumption, and infer weather
  - can be thought as a **generative** model
- **How are HMMs formalised mathematically?**

1. A set of N states:
$$q_1, \ldots, q_N$$
(these represent the **hidden** observations)
2. A **transition probability matrix** A, where a_{ij} is the transition probability of moving from q_i to q_j and with:
$$\sum_{j=1} a_{ij} = 1, \forall i \in [1, N]$$
3. A **sequence** of T observations:
$$O = o_1, o_2, \ldots, o_T$$
4. A **sequence** of **emission probabilities**:
$$B = b_i(o_t)$$
indicating the probability of an observation o_t being generated by (hidden) state q_i
5. 2 special states q_0, q_F indicating **start** and **final** states. These aren't associated with the observation. Contain transition probabilities out of q_0, and transition probabilities into q_F

- **What are the assumptions for a first-order HMM?**
  - the probability of a state depends only on the previous state:
$$P(q_i \mid q_1, \ldots, q_{i-1}) = P(q_i \mid q_{i-1})$$
  - an **output observation** depends only on the state which produces it:
$$P(o_i \mid q_1, \ldots, q_T, o_1, \ldots, o_i, \ldots, o_T) = P(o_i \mid q_i)$$

- **What is a second-order HMM?**
  - the probability of a state depends on the **two** previous states:

$$P(q_i \mid q_{i-2}, q_{i-1})$$

  - the **transition matrix** will then be $N \times N \times \boldsymbol{N}$
- **Which three fundamental problems can be solved by HMMs?**
  1. **Likelihood**: compute the likelihood of an observation sequence
  2. **Decoding**: compute the best hidden state sequence producing an observation sequence
  3. **Learning**: given an observation sequence, and the states of a HMM, learn the parameters (i.e transition and emission matrices) of the HMM

3.3 POS Tagging

- **What is POS tagging?**
  - determinining the **parts of speech** or **syntactic categories** of the words conforming a sentence:

$$This/DET\ is/VB\ a/DET\ simple/ADJ\ sentence/NOUN$$

- **Why is POS tagging important?**
  - first step towards syntactic analysis
  - simpler than parsing, but still useful (i.e as features for text classification)
- **What other types of tagging are there?**
  - **Named Entity Recognition**: determining whether a word belongs to a person/organisation/location or neither
  - **Information Field Segmentation**: identify the "fields" in a given type of text
    - for example, in an ad for houses, determining which words are prices, sizes, location, etc...
  - **Gesture Recognition**: understand and predict gestures in a video sequence
- **How do open class words differ from closed class words?**
  - OCW (**content words**) are **content bearing** (i.e nouns, verbs, adjectives, adverbs), and **illimited** (new ones added all the time, like "email")
  - CCW (**function words**) tie concepts of sentences together, and are limited (i.e pronouns, prepositions, connectives, etc ...)
- **How many POS tags are there?**
  - depends on linguistic and practical considerations; annotators decide. For example:
    - include names and common nouns?
    - include singulars and plurals?
    - include past and present tense verbs?
  - in other languages, this is more complex (i.e agglutinative), since thousands of possibilities
- **Why is POS tagging hard?**

- **ambiguity**
  - **water** can be a noun (*I drink water*) or a verb (*I water the plants*)
  - **wind** (*Please wind down* vs *There was a strong wind*)
- **sparse data**
  - there are words we haven't seen/seen in a given context
  - there are word-tag pairs **never** before seen/created (i.e creating verbs like "google")
- **labels**
  - the same word can have **different labels**
  - looking at the previous word is not enough to determine the true tag
  - require additional context information to diambiguate

3.4 HMMs for POS Tagging

- **What does the tag of a word depend on?**
  - the **word** being labelled (i.e **chair** is likely a noun)
  - the tags of the surrounding words (i.e word after an adjective is likely a noun)
- **Why are HMMs good for POS tagging?**
  - the HMM assumptions are well suited for POS tagging
  - we can think of a sentence as being **probabilistically generated**, where:
    - tag t_i depends on the previous tag:

      $$P(t_i \mid t_{i-1})$$

      (first-order Markov assumption, **transition probability**)
    - word w_i is conditioned on its tag:

      $$P(w_i \mid t_i)$$

      (words occur as **emissions** of their tag, independent of other words or tags; **emission probability**)
- **Where do the emission and transition probabilities come from?**
  - these can be computed from corpora
  - each sentence in the corpus is annotated with the POS tags of its words
  - for **transition probabilities**:

    $$a_{ij} = P(t_i \mid t_{i-1}) = \frac{C(t_{i-1}, t_i)}{\sum_k C(t_{i-1}, t_k)}$$

    that is, the proportion of times in which t_i followed t_{i-1}, out of all the times that t_{i-1} appeared before any tag
  - for **emission probabilities**

    $$b_{ij} = P(w_i \mid t_i) = \frac{C(t_i, w_i)}{\sum_k C(t_i, w_k)}$$

    that is, the proportion of times in which tag t_i emitted w_i, out of all the emissions produced by t_i

$t_{i-1} \backslash t_i$	NNP	MD	VB	JJ	NN	...
<s>	0.2767	0.0006	0.0031	0.0453	0.0449	...
NNP	0.3777	0.0110	0.0009	0.0084	0.0584	...
MD	0.0008	0.0002	0.7968	0.0005	0.0008	...
VB	0.0322	0.0005	0.0050	0.0837	0.0615	...
JJ	0.0306	0.0004	0.0001	0.0733	0.4509	...
...	...	...	...	...	...	...

Figure 11: Transition probabilities from the WSJ corpus. If we had the whole table, each row should add up to 1. These tell us that:
- **proper nouns** (NNP) typically begin sentences (probability of about 0.28)
- **modal verbs** (MD) are typically followed by **bare verbs** (VB)
- **adjectives** (JJ) are often followed by **nouns** (NN)

$t_i \backslash w_i$	Janet	will	back	the	...
NNP	0.000032	0	0	0.000048	...
MD	0	0.308431	0	0	...
VB	0	0.000028	0.000672	0	...
DT	0	0	0	0.506099	...
...	...	...	...	...	...

Figure 12: Emission probabilities from the WSJ corpus. We can see that:
- more than halr of **determiners** (DT) are "the"
- there is a potential annotation error, since "the" has a probability of being a proper noun

- **Can we apply smoothing to emission probabilities?**
  - applying **Good Turing** could be smart
  - if w_i has never been seen, the probability of it being emitted by a highly popular POS tag (like noun or adjective) is low
  - smart to estimate using an unlikely taga
- **How are HMMs used for POS tagging?**
  - if we have a HMM, POS tagging gets reduced to, given a sentence, what is the **most probable path across the HMM** which produces the sentence?
- **What is the probability of a tagged sentence, using HMMs**
  - this isn't exactly what we want, since we want to **find** untagged sequences, but it is a useful basis
  - say we have a sentence $S = w_1, \ldots, w_n$ and its tags $T = t_1, \ldots, t_n$

- the probability of S havings tags T is the joint probability:

$$P(S,T) = P(S \mid T)P(T) = \prod_{i=1}^{n} P(t_i \mid t_{i-1})P(w_i \mid t_i)$$

- this is a simple application of the independence assumptions
- for example:

$$This/DET\ is/VB/a/DET/simple/JJ/sentence/NN$$

  has probability:

$$\begin{aligned} P(S,T) = &P(DET|\texttt{<s>})P(VB|DET)P(DET|VB)P(JJ|DET)P(NN|JJ)P(\texttt{</s>}|NN) \\ &P(This|DET)P(is|VB)P(a|DET)P(simple|JJ)P(sentence|NN) \end{aligned}$$

- **How do HMMs relate to the previous models?**
  - **N-Grams**: model sequences using the Markov assumption (dependence only on history), but no hidden variables
  - **Naive Bayes**: no sequential dependence, but there are hidden variables (since we assume that classes generate words)
- **What is the tagging problem with HMMs, assuming we have tagged training data, but untagged testing data?**
  - given an untagged sentence S, we want to find:

$$\hat{T} = \underset{T}{argmax} P(T \mid S)$$

  - we can apply Bayes Rule, and:

$$\underset{T}{argmax} P(T \mid S) = \underset{T}{argmax} P(S \mid T)P(T)$$

  - notice:
    * $P(T)$ is, by the independence assumption, the **state transition** probability:

$$P(T) = \prod_{i} P(t_i \mid t_{i-1})$$

    * $P(S \mid T)$, again by independence, is a product of **emission probabilities**:

$$P(S \mid T) = \prod_{i} P(w_i \mid t_i)$$

  - hence, given some T^*, we can easily determine $P(T \mid S)$
  - however, notice, it is **not** efficient to iterate through **all** possible state sequences: with c tags and n words, there are c^n possible tag sequences

3.5 The Viterbi Algorithm

- **What is the Viterbi Algorithm?**
  - a **dynamic programming** algorithm, used to compute the **most likely** transition sequence T which can generate S

- with c tags and n words:
$$\mathcal{O}(c^2n)$$
- it is efficient, since it doesn't consider **all** sequences
- a great video on this can be found here

- **How are Viterbi and the noisy channel model related?**
  - we can think of T as a signal someone wants to send; when it gets altered with noise with $P(S \mid T)$, we get S
  - we need to find the model which can decode the words back into the tags
  - **decoding** is the general procedure by which we **infer hidden variables** from an instance (like spelling correction or POS tagging)

- **What is the Viterbi Algorithm Recursion?**
  - the most likely POS tagging of $w_1, \dots, w_n$ should only depend on the most likely POS tagging of $w_1, \dots, w_{n-1}$
  - we keep a table with entries $v_t(j)$, which gives the probability of word w_t having tag j, given that we have traversed states:
  $$q_0, q_1, \dots, q_{t-1}$$
  and words:
  $$w_1, w_2, \dots, w_t$$
  Hence:
  $$v_t(j) = \max_{q_0,\dots,q_{t-1}} P(q_0, \dots q_{t_1}, w_1, \dots, w_t, q_t = j \mid \lambda)$$
  where λ is the HMM model
  - notice, this probability only depends on:
    * the **previous Viterbi path probability**, with which we reached state q_{t-1}
    * the **transition probability** of jumping from q_{t-1} to j
    * the **emission probability** of being at j and emitting the observed word w_t

  Thus, we obtain our dynamic programming recursion:
  $$v_t(j) = \left(\max_{i=1,\dots,c} v_{t-1}(i) \times a_{ij} \right) \times b_j(w_t)$$

function VITERBI(*observations* of len T, *state-graph* of len N) **returns** *best-path*

create a path probability matrix *viterbi[N+2,T]*
for each state s **from** 1 **to** N **do** ;initialization step
  $viterbi[s,1] \leftarrow a_{0,s} * b_s(o_1)$
  $backpointer[s,1] \leftarrow 0$
for each time step t **from** 2 **to** T **do** ;recursion step
  for each state s **from** 1 **to** N **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} \; viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} \; viterbi[s',t-1] * a_{s',s}$
$viterbi[q_F,T] \leftarrow \max_{s=1}^{N} \; viterbi[s,T] * a_{s,q_F}$; termination step
$backpointer[q_F,T] \leftarrow \operatorname{argmax}_{s=1}^{N} \; viterbi[s,T] * a_{s,q_F}$; termination step
return the backtrace path by following backpointers to states back in time from $backpointer[q_F,T]$

Figure 13: Notice, in the actual implementation, we keep track of **backpointers**, which tell us the POS tag from which we come from. This is needed so that when the algorithm is completed, we have the full POS tagging of the sentence.

- **What are the formulae for computing the probabilities in Viterbi?**
  1. **Initialisation**
  $$v_1(j) = a_{START,j} \times b_j(w_1), \qquad \forall j \in [1, N]$$
  where N is the total number of POS tags
  2. **Recomputation**
  $$v_t(j) = \left(\max_{i=1,\ldots,c} v_{t-1}(i) \times a_{ij} \right) \times b_j(w_t), \qquad j \in [1, N],\ t \in [2, |\underline{w}|]$$
  3. **Final**
  $$v_{|\underline{w}|+1}(STOP) = \max_{i=1,\ldots,c} v_{|\underline{w}|}(i) \times a_{i,STOP}$$
  where notice that we don't multiply by the emission probabilities (since when you transition to the last step, no emission is made)
- **In practice, how are the Viterbi probabilities computed?**
  - we use **log-space**
  - if we define:
  $$g_t(\underline{w}, i, j) = \begin{cases} \log(a_{ij}) + \log(b_j(w_t)), & t \in [1, |\underline{w}|] \\ \log(a_{ij}), & t = |w| + 1 \end{cases}$$
  then, the computations can be redefined as:

– **Initialisation**

$$v_1(j) = g_1(\underline{w}, START, j), \qquad \forall j \in [1, N]$$

– **Recomputation**

$$v_t(j) = \max_{i=1,\ldots,c} \left(v_{t-1}(i) + g_t(\underline{w}, i, j)\right), \qquad j \in [1, N],\ t \in [2, |\underline{w}|]$$

– **Final**

$$v_{|\underline{w}|+1}(STOP) = \max_{i=1,\ldots,c} \left(v_{|\underline{w}|}(i) + g_{|w|+1}(\underline{w}, i, STOP)\right)$$

3.5.1 Worked Example: Viterbi Algorithm

a_{ij}	STOP	NN	VB	JJ	RB
START	0	0.5	0.25	0.25	0
NN	0.25	0.25	0.5	0	0
VB	0.25	0.25	0	0.25	0.25
JJ	0	0.75	0	0.25	0
RB	0.5	0.25	0	0.25	0

b_{ik}	time	flies	fast	...	...	...
NN	0.1	0.01	0.01	...	...	...
VB	0.01	0.1	0.01	...	...	...
JJ	0	0	0.1	...	...	...
RB	0	0	0.1	...	...	...

Figure 14: We consider the following emission and transition probabilities, and we want to determine th ePOS tags for “time flies fast”.

	$time_1$	$flies_2$	$fast_3$	-
NN	0.5x0.1=0.05			
VB				
JJ				
RB				
STOP	-	-	-	

Figure 15:
- The probability of the first tag being NN is: 0.5
- The probability of NN emitting “time” is: 0.1
- Hence, total probability: $0.5 \times 0.1 = 0.05$.

	time$_1$	*flies*$_2$	*fast*$_3$	-
NN	0.5x0.1=0.05			
VB	0.25x0.01=0.0025			
JJ	0			
RB	0			
STOP	-	-	-	

Figure 16: Similarly, VB is the first tag with probability 0.25, and it emits "time" with probability 0.01, so total probability is 0.0025.
"Time" is never emitted as a JJ or RB, so probability 0.

	time$_1$	*flies*$_2$	*fast*$_3$	-
NN	0.05 x 0.25			
VB	0.0025 x 0.25			
JJ	0			
RB	0			
STOP	-	-	-	

Figure 17:
- Probability of NN emitting "flies": 0.01

We consider the first transition $NN \rightarrow NN$:
- Probability of NN following NN: 0.25
- Hence, the probability that the NN "time" occurs before the NN "flies" is: $0.05 \times 0.25 \times 0.01 = 0.000125 = 1.25 \times 10^{-4}$

We consider the second transition $VB \rightarrow NN$:
- Probability of VB following NN: 0.25
- Hence, the probability that the VB "time" occurs before the NN "flies" is: $0.0025 \times 0.25 \times 0.01 = 6.25 \times 10^{-6}$

	$time_1$	$flies_2$	$fast_3$	-
NN	0.05 x 0.25	0.05 x 0.25 x 0.01		
VB	0.0025			
JJ	0			
RB	0			
STOP	-	-	-	

Figure 18: If “flies” is a NN, we have seen above that it is more likely that it was preceded by the NN “time”, so we keep this as the probability.

	$time_1$	$flies_2$	$fast_3$	-
NN	0.05	1.25E-4		
VB	0.0025 x 0	x 0.5		
JJ	0			
RB	0			
STOP	-	-	-	

Figure 19: We keep a backpointer, to indicate that the NN “flies” comes after the NN “times”.
Now notice that the transition $VB \rightarrow VB$ has probability 0. Moreover, since “time” can’t be a JJ or a RB, the only possibilitiy is that, when “flies” is a VB, it must have come from the NN “time”. Hence:
- Probability of VB emitting “flies”: 0.1
- Probability of transition $NN \rightarrow VB$: 0.5
- Hence, the probability that the VB “flies” occurs after the NN “time” is: $0.05 \times 0.5 \times 0.1 = 0.0025$

	$time_1$	$flies_2$	$fast_3$	-
NN	0.05	1.25E-4	6.25E-6	
VB	0.0025	0.0025	6.25E-7	
JJ	0	0	6.25E-5	
RB	0	0	6.25E-5	
STOP	-	-	-	

Figure 20: We continue filling in the table, keeping track of the most probable path.

	$time_1$	$flies_2$	$fast_3$	-
NN	0.05	1.25E-4	6.25E-6 x 0.25	-
VB	0.0025	0.0025	6.25E-7 x 0.25	-
JJ	0	0	6.25E-5 x 0.0	-
RB	0	0	6.25E-5 x 0.5	-
STOP	-	-	-	

Figure 21: Finally, we need to consider the transition probabilities to the end state, from each of a NN, VB, JJ and RB.

	$time_1$	$flies_2$	$fast_3$	-
NN	0.05	1.25E-4	6.25E-6	-
VB	0.0025	0.0025	6.25E-7	-
JJ	0	0	6.25E-5	-
RB	0	0	6.25E-5	-
STOP	-	-	-	3.125E-5

Figure 22: We can then see that the most probable path is that with "time flies fast" as $NN \to VB \to RB$ (with probability 3.125×10^{-5})

3.6 The Forward Algorithm

- **What is the forward algorithm?**
  - the way in which HMMs can be used to compute the **likelihood** of a word sequence
- **How can we compute the likelihood of a sequence?**
  - recall, we showed that the probability of a sentence S having tags T is:

$$P(S,T) = \prod_{i=1}^{n} P(t_i \mid t_{i-1}) P(w_i \mid t_i)$$

  - the **Law of Total Probability** then tells us that $P(S)$ is:

$$P(S) = \sum_{T'} P(S, T')$$

  - that is, a sum of the probability of S, given all possible tags
- **How is the Forward Algorithm similar to Viterbi?**

– determining all possible tag sequences is exponential
– the **Forward Algorithm** is a dynamic programming approach, which is very similar to Viterbi
– keep a table with entries $\alpha_t(j)$: the probability of being in state j after observing $w_1, \ldots, w_t$:

$$\alpha_t(j) = P(w_1, \ldots, w_t, q_t = j \mid \lambda)$$

– to compute this, we just sum over all posible paths which can be used to reach j:

$$\alpha_t(j) = \sum_{i=1}^{n} \alpha_{t-1}(i) a_{ij} b_j(w_t)$$

function FORWARD(*observations* of len T, *state-graph* of len N) **returns** *forward-prob*

create a probability matrix *forward[N+2,T]*
for each state s **from** 1 **to** N **do** ;initialization step
$\quad forward[s,1] \leftarrow a_{0,s} * b_s(o_1)$
for each time step t **from** 2 **to** T **do** ;recursion step
$\quad$ **for** each state s **from** 1 **to** N **do**

$$forward[s,t] \leftarrow \sum_{s'=1}^{N} forward[s',t-1] * a_{s',s} * b_s(o_t)$$

$forward[q_F,T] \leftarrow \sum_{s=1}^{N} forward[s,T] * a_{s,q_F}$; termination step
return $forward[q_F,T]$

- **What are the formulae for the Forward Algorithm?**
  – for Viterbi, we had:
    1. **Initialisation**
    $$v_1(j) = a_{START,j} \times b_j(w_1), \qquad \forall j \in [1, N]$$
    2. **Recomputation**
    $$v_t(j) = \left(\max_{i=1,\ldots,c} v_{t-1}(i) \times a_{ij}\right) \times b_j(w_t), \qquad j \in [1, N],\ t \in [2, |\underline{w}|]$$
    3. **Final**
    $$v_{|\underline{w}|+1}(STOP) = \max_{i=1,\ldots,c} v_{|\underline{w}|}(i) \times a_{i,STOP}$$
  – for the **Forward Algorithm**, instead of finding the most likely path, we just **sum** over all possible paths:
    1. **Initialisation**
    $$v_1(j) = a_{START,j} \times b_j(w_1), \qquad \forall j \in [1, N]$$
    2. **Recomputation**
    $$v_t(j) = \left(\sum_{i=1}^{N} v_{t-1}(i) \times a_{ij}\right) \times b_j(w_t), \qquad j \in [1, N],\ t \in [2, |\underline{w}|]$$
    3. **Final**
    $$v_{|\underline{w}|+1}(STOP) = \sum_{i=1}^{N} v_{|\underline{w}|}(i) \times a_{i,STOP}$$

3.7 HMMs for Unsupervised Estimation

- **Can we develop a HMM model even when the corpus of sentences is annotated?**
  - we can only see the ouputs (words), but no state sequence
  - situation ideal for **expectation maximisation**:
    * with state sequences, we can compute the emission and transition probabilities
    * with emission and transition probabilities, we can find the most likely state sequence (Viterbi)
  - in practice, EM not too good; better to apply **semi-supervised learning**
- **How is EM applied to HMMs?**
  1. Randomly initialise the **transition probabilities** (A) and the **emission probabilities** (B)
  2. At each iteration:
     (a) **Expectation**: use A, B to compute the **expected counts**
     (b) **Maximisation**: use **expected counts** to update A, B
  3. Repeat until convergence

function FORWARD-BACKWARD(*observations* of len T, *output vocabulary* V, *hidden state set* Q) **returns** $HMM=(A,B)$

initialize A and B

iterate until convergence

E-step

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)} \ \forall\, t \text{ and } j$$

$$\xi_t(i,j) = \frac{\alpha_t(i)\,a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(N)} \ \forall\, t,\ i, \text{ and } j$$

M-step

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

$$\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t.\ O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

return A, B

Figure 23: This is known as the **Forward-Backward Algorithm**, which computes the **expected counts** with a **dynamic programming approach**. Here, the α are the forward probabilities defined above. The β are the backward probabilities, which are the probability of, given that we are at state i at time t, observing $w_{t+1}, \ldots, w_n$:

$$\beta_t(i) = P(w_{t+1}, \ldots, w_n \mid q_t = i, \lambda)$$

To compute this, see below:

1. **Initialization:**

(6.29) $$\beta_T(i) \;=\; a_{i,F}, \quad 1 \le i \le N$$

2. **Recursion** (again since states 0 and q_F are non-emitting):

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

3. **Termination:**

(6.31) $$P(O|\lambda) = \alpha_T(q_F) = \beta_1(0) = \sum_{j=1}^{N} a_{0j}\, b_j(o_1)\, \beta_1(j)$$

- **What are expected counts?**
  - we are dealing with **probabilistic EM**
  - consider counting the transitions of the form $q_{t-1} \rightarrow q_t$
  - with **real counts**, $C(q_{t-1}, q_t)$ counts 1 each time the pair are seen together
  - with expected counts, if a sequence of states has a probability p of appearing, we count p each time $q_{t-1} \rightarrow q_t$ appears within the sequence

Possible tag sequence			Probability of the sequence
N	N	N	p_1
N	V	N	p_2
N	N	V	p_3
aa	bb	cc	- Sequence of observations (words)

$C_T(\text{N, N}) = 2\, p_1 + p_3$