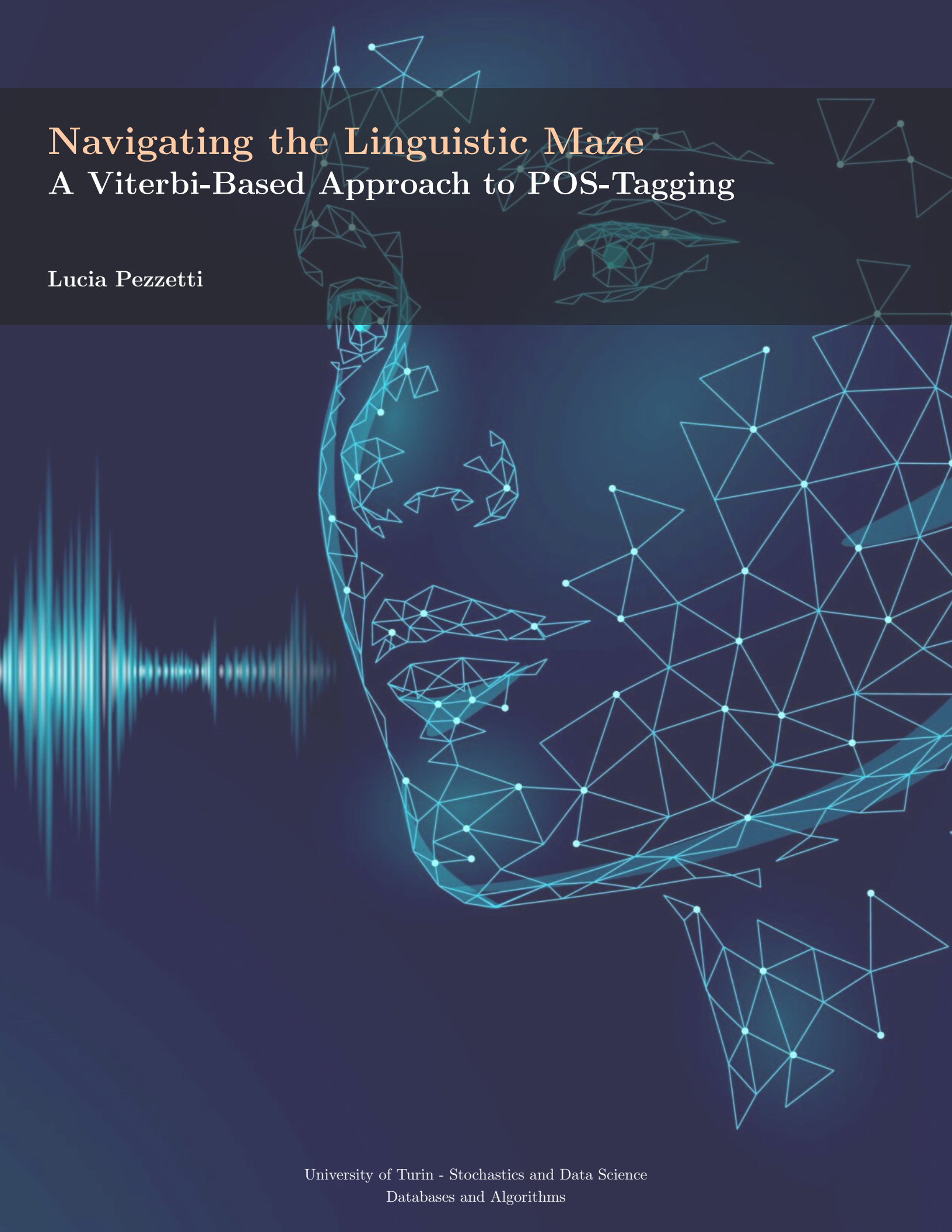
Navigating the Linguistic Maze

A Viterbi-Based Approach to POS-Tagging

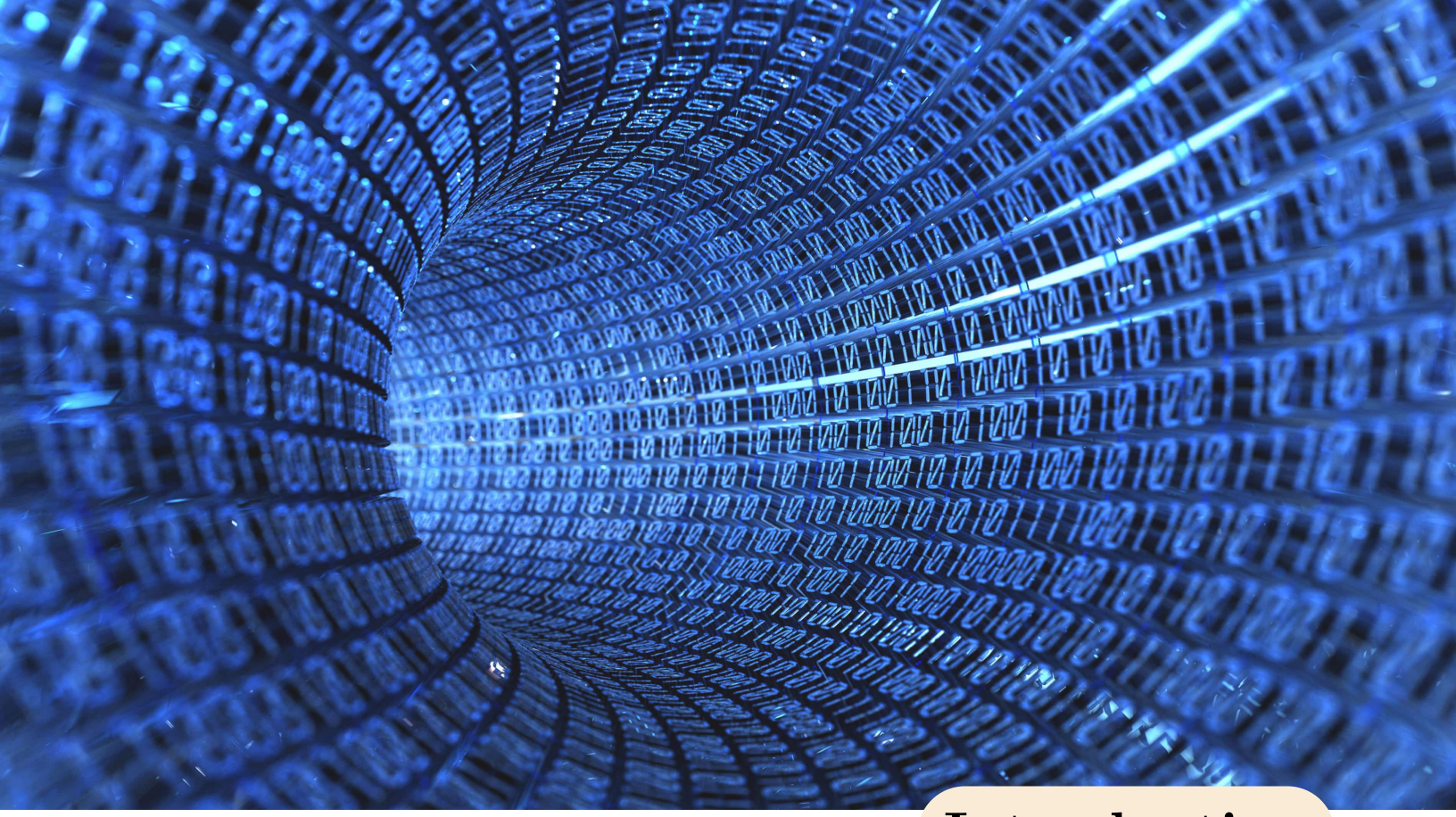
Lucia Pezzetti



University of Turin - Stochastics and Data Science
Databases and Algorithms

Table of Contents

Contents



Introduction

1. Overview

Part-of-speech (POS) tagging - also known as grammatical tagging or word category disambiguation - is a popular Natural Language Processing process that aims to categorize words in a text on the basis of their syntactical role. In order to do so, it assigns to each word the specific grammatical category or part-of-speech tag that best describes the use of that word in the given sentence.

POS tagging is essential in understanding the structure and meaning of a sentence, as different parts of speech have distinct grammatical functions and semantic properties. By identifying the correct POS tag for each word, NLP systems can analyze the relationships between words, extract meaningful information, and perform more advanced language processing tasks like parsing, information retrieval, machine translation, sentiment analysis, and text generation.

Interestingly enough, part of speech is something we have started to learn since a very young age and a very fundamental part of our everyday life. As humans we understand several nuances of the natural language and this allows us - for example - to unambiguously use the same word in very different contexts.

I like his watch

The man fans the flame

The fans watch the race

Looking at the above examples it is clear how watch has been used as a *noun* in the first sentence and as a *verb* in the third one. Similarly, fans is a *verb* in the second phrase and a *noun* in the last one.

It is these very intricacies in natural language understanding that we want to teach to a machine, starting from the ability to understand the syntactical role of a word in a sentence.

1.1 Data

For this project I have used the Brown Corpus data set of NLTK with the 'universal' tagset.

Specifically, The Brown Corpora contains 57340 POS annotated sentences and 1161192 tuples. Analyzing further the data set, it contains 56057 distinct tokens and 66939 distinct bigram tags. Therefore there are 10882 "ambiguous" words having more than one tag.

Whilst the Universal tagset refers to a simplified tagset comprising only the following 12 coarse tag classes:

Universal tagset		
Tag	Description	Examples
ADJ	adjective	new, good, high, special, big, local
ADP	adposition	on, of, at, with, by, into, under
ADV	adverb	really, already, still, early, now
CONJ	conjunction	and, or, but, if, while, although
DET	determiner, article	the, a, some, most, every, no, which
NOUN	noun	year, home, costs, time, Africa
NUM	numeral	twenty-four, fourth, 1991, 14:24
PRT	particle	at, on, out, over per, that, up, with
PRON	pronoun	he, their, her, its, my, I, us
VERB	verb	is, say, told, given, playing, would
.	punctuation marks	. , ; !
X	other	ersatz, esprit, dunno, gr8, univeristy

An example of how the data are structured is reported:

```
from nltk.corpus import brown

print(brown.tagged_words(tagset='universal'))

>> [('The', 'DET'), ('Fulton', 'NOUN'), ('County',
   'NOUN'), ('Grand', 'ADJ'), ('Jury', 'NOUN'), ('said', 'VERB'), ('Friday', 'NOUN'), ('an', 'DET'
    ), ('investigation', 'NOUN'), ('of', 'ADP'), ("Atlanta's", 'NOUN'), ('recent', 'ADJ'), ('
    primary', 'NOUN'), ('election', 'NOUN'), ('produced', 'VERB'), (''', '.'), ('no', 'DET'), ('
    evidence', 'NOUN'), ("''", '.'), ('that', 'ADP'), ('any', 'DET'), ('irregularities', 'NOUN'),
    ('took', 'VERB'), ('place', 'NOUN'), ('.', '.')]
```

The data set has been split into training and test data. Precisely, the training data correspond to the 80% of the whole original data set. It has been verified that among the training data all the 12 POS tags appears at least once.

2. POS tagging with HMMs and Viterbi algorithm

As already argued, POS tagging is a fundamental prerequisite to several and complex NLP problems, for this reason an efficient strategy to address this problem is required.

Clearly, trying all the possible tags combinations is not feasible, except for very simple - and hence quite unrealistic - inputs. Indeed, given a set of words of length n_{words} and n_{tags} tags, the total number of possible combinations is $n_{tags}^{n_{words}}$ - exponential in n_{words}.

Thus different strategies are needed. The types of POS taggers that will be discussed in this project are:

- Rule-based POS taggers
- Stochastic POS taggers

Rule-based POS tagging assigns POS tags to the words based on some predefined linguistic rules - usually created by linguists or language experts. These rules generally rely on linguistic patterns and observations about how words behave in context. An example may be:

if a word ends with -ing, tag it as a verb

Clearly, this method is all the more efficient the more detailed the rules are. Consequently this approach may be time-consuming and unable to handle all the subtleties of the natural language.

Stochastic taggers provide a different approach that is gaining a lot of popularity lately: leveraging the power of machine learning algorithms to learn from labeled training data. In other words, by observing the patterns and co-occurrences of words and their tags in the training data, a stochastic tagger builds a statistical model that estimates the probability of a word having a particular POS tag based on its context.

There are several different approaches and algorithms that can be used for stochastic POS tagging. Here, the one based on Hidden Markov Models (HMMs) will be discussed.

Briefly, at the very heart of this technique there is the assumption that the data - i.e. the sequence of words and the associated POS tags - satisfy the dependencies assumptions of an HMM (see section 2.1). Given this choice, probability needs to be incorporated in the model. As it will be better explained below, HMMs require both:

- the probability that a word occurs with a particular tag - computed using the frequency that the tag has been encountered in the training set with that particular word.
- the probability that a sequence of tags occurs. This approach is also called the *n-grams* approach, as the "best" tag for a word is determined also by computing the probability that it occurs given the last n tags. In this project, a 1-gram approach will be used.

Combining these two features together allows to consider both the nature of the words in itself and the context in which it is used.

2.1 Hidden Markov Models

The following will be a very briefly and not exhaustive introduction to HMMs. In particular, the models will be presented referring to their application in POS tagging.

One can think of Hidden Markov Models as a class of models in which an underlying stochastic process - the state process X - is unobserved and constitutes the object of inference, whilst a related process - the observation process Y - is observed and provides information about X. In order for this couple of processes to be an HMM, they need to satisfy the following conditional dependencies properties:

- X should be a Markov Chain;
- each value of the observation process Y should depend only on the value of the state process at that particular time:
$$P(Y_t|Y_{0:t-1}, X_{0:t}) = P(Y_t|X_t)$$

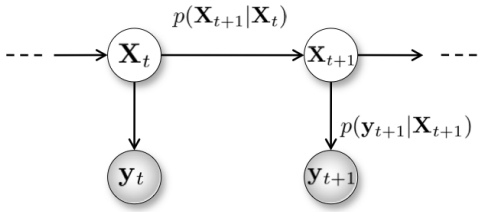
As anticipated, the object of inference is the state process or, more precisely, the motivating question is

"given the sequence of observed states $Y_{0:t}$,
which is the most probable sequence of underlying states $X_{0:t}$?"

The theoretical answer exploits Bayes theorem[1]:

$$p(X_{0:t+1}|y_{0:t+1}) \propto p(X_{0:t}|y_{0:t})p(X_{t+1}|X_t)p(y_{t+1}|X_{t+1})$$

where $P(y_{t+1}|X_{t+1})$ is the so-called **emission** probability and $P(X_{t+1}|X_t)$ the **transition** probability.



The usage of these models in POS tagging is quite straightforward: the sequence of words is modeled as the sequence of observations, whilst the part-of-speech tags correspond to the underlying state values. Hence, for modelling purposes, the n^{th} word of the input text is considered as the state at time n of a stochastic process.

Following this interpretation, the transition probabilities $P(tag_t|tag_{t-1})$ represent the probability of finding a certain syntactic structure, tag_t after a given POS tag, tag_{t-1} - for example $P('NOUN'|'DET')$. Whereas, the emission probabilities $P(word|tag)$ correspond to the probabilities of having the particular word, $word$, given the syntactic role that it needs to play inside the sentence - for instance $P(watch|'VERB')$.

Note that both the transition and emission probabilities are to be determined on the basis of the training data and therefore will be assumed to be known in the following explanation of the model.

Having established the theoretical foundations of the model, the issue is now **how** to efficiently determine the most likely sequence of POS tags for a given sequence of words based on this probabilistic HMM setup. A possible answer can be found in the Viterbi algorithm.

[1] Actually, for convenience, both the state transitions and the conditional distribution of the observations given the states are assumed to admit a density w.r.t. some Lebesgue measure.

2.2 The Viterbi algorithm: heuristics

The Viterbi algorithm is a dynamic programming algorithm that is commonly used to find the most likely sequence of hidden states in a Hidden Markov Model. It efficiently computes the maximum a posteriori[2] (MAP) estimation of the hidden state sequence given an observed sequence of symbols.

Focusing on the POS tagging problem, it can be applied to find the most likely sequence of POS tags for a given sequence of words in one or more sentences.

Loosely speaking, at each time step - i.e. for any given word - the algorithm scans all the possible POS tags and, thoughtfully, keep track of the possible paths with the associated probabilities. The ultimate optimal sequence of tags is determined only at the end of the procedure and corresponds to the stored path with the highest probability. Why is the procedure 'thoughtful' in exploring possible POS sequences? Because it only examines a meaningful subset of them, returning the result in polynomial time.

To gain a first insight on the algorithm behavior, the following simple example is considered:

`The fans watch the race`

In order to keep the example tractable, the possible POS tags are limited to $'DET'$, $'NOUN'$, $'VERB'$ and the zero-probability options for each word are ruled out. The considered transition and emission probability matrices are available and displayed below.

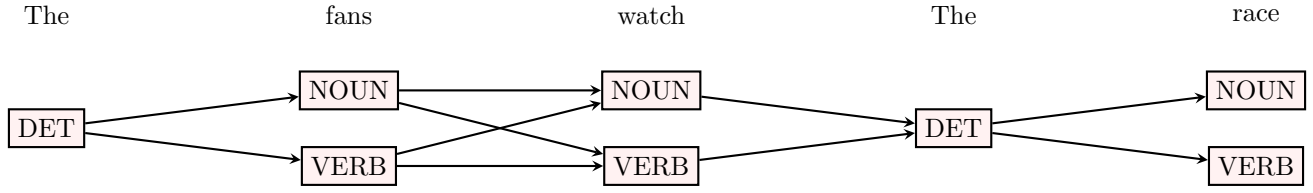
Transition probability matrix[3]

	DET	NOUN	VERB
START	0.8	0.2	0
DET	0	0.9	0.1
NOUN	0	0.5	0.5
VERB	0.5	0.5	0

Emission probability matrix[4]

	the	fans	watch	race
DET	0.2	0	0	0
NOUN	0	0.1	0.3	0.1
VERB	0	0.2	0.15	0.3

So, the example assumes that $'The'$ is for sure a determinant ($'DET'$), whereas for each of the other three words there are two available options.



Remark 2.1. *Although the example has been simplified and contains very few words, there are still eight possible ways to assign the POS tags: for more complex cases in which the whole set of tags is considered, the number of possibilities will quickly explode.*

Stochastic POS tagging based on HMMs tackles this problem by looking for

$$\max_{tag_1,tag_2,tag_3,tag_4,tag_5} P(tag_1, tag_2, tag_3, tag_4, tag_5, 'The', 'fans', 'watch', 'the', 'race')$$

or equivalently

$$\max_{tag_i} \underbrace{P(tag_1)}_{\text{initial prob.}} \underbrace{\prod_{k=2}^{5} P(tag_k|tag_{k-1})}_{\text{transitions } tag_{k-1} \to tag_k} \underbrace{\prod_{k=1}^{5} P(w_k|tag_k)}_{\text{emissions } tag_k \to w_k}$$

Before proceeding, it is worth-noticing that the transition probabilities play a key role in determining the correct tags. Indeed they allow to make predictions based not only on the single words, but also on the structure of the phrase itself: after a determinant, for example, it is more likely to find a noun than another determinant or even a verb. For this reason, looking at the emission matrix only is (other than not theoretically well-founded) misleading: in the example the outcome would have been

[2]The maximum a posteriori is one of the most used Bayesian point estimates and corresponds to the mode of the posterior distribution.

DET VERB NOUN DET VERB

Clearly different than the right answer.

The mechanism of the Viterbi algorithm, for this example, is shown with reference to the above diagram.
The algorithm starts by answering two questions:

which is the probability that the first input word is a determinant? 0.8

which is the probability that, given a determinant, the corresponding word is 'the'? 0.2

Hence the global probability of observing the information $('the',' DET')$ is simply $0.8 \times 0.2 = 0.16$. The following step requires the analysis of the word $'fans'$. In this example it can either be a noun or a verb, so both cases need to be evaluated. Exploiting Bayes theorem, the joint probability of observing all the information $(('the',' DET'),('fans',' NOUN'))$ is

$$\underbrace{P(('the',' DET'))}_{0.16} \times \underbrace{P('NOUN'|'DET')}_{0.9} \times \underbrace{P('fans'|'NOUN')}_{0.1} = 0.0144$$

and similarly for the joint probability of having $(('the',' DET'),('fans',' VERB'))$:

$$\underbrace{P(('the',' DET'))}_{0.16} \times \underbrace{P('VERB'|'DET')}_{0.1} \times \underbrace{P('fans'|'VERB')}_{0.3} = 0.0032$$

At this stage, the branch with the lower probability can't simply be discarded. The reason is that there is no guarantees that the evaluation of the following states will leave the path with the highest probability unchanged.

Let's try to go a step further analysing the transition from the observed word $'fans'$ to $'watch'$. Limiting at the set of non-zero emission probabilities, $'watch'$ can either be generated by the POS tag $'NOUN'$ or $'VERB'$. Exactly as before, the desired joint probabilities can be computed using the transition and the emission probability matrices, together with the already stored probabilities related to the previous stage. In particular, for the joint probability of $(('the',' DET'),('fans',' NOUN'),('watch',' NOUN'))$:

$$\underbrace{P(('the',' DET'),('fans',' NOUN'))}_{0.0144} \times \underbrace{P('NOUN'|'NOUN')}_{0.5} \times \underbrace{P('watch'|'NOUN')}_{0.3} = 0.00216$$

Whereas the joint probability of $(('the',' DET'),('fans',' VERB'),('watch',' NOUN'))$ is

$$\underbrace{P(('the',' DET'),('fans',' VERB'))}_{0.0032} \times \underbrace{P('NOUN'|'VERB')}_{0.5} \times \underbrace{P('watch'|'VERB')}_{0.15} = 0.00024$$

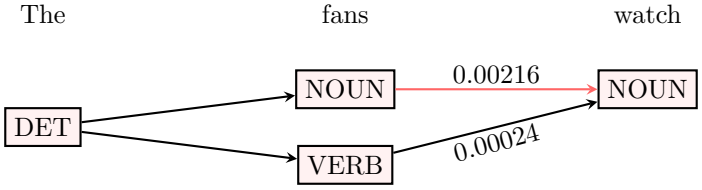
At this very stage, one can start to appreciate the power of the Viterbi algorithm and the reason why it is so powerful with respect to a "brute force" approach. Indeed, here two different paths allow to reach the same tuple $('watch',' NOUN')$ and the less probable path can be "cut-off". More precisely, the following observation - which is at the very heart of the Viterbi algorithm - can be made:

"Due to the dependency structure of HMMs, if there are two or more paths all merging to the same node at time n, then - regardless of the following observations (words) - the first n steps of the most probable path including that node will match the path with the highest probability entering the considered node."

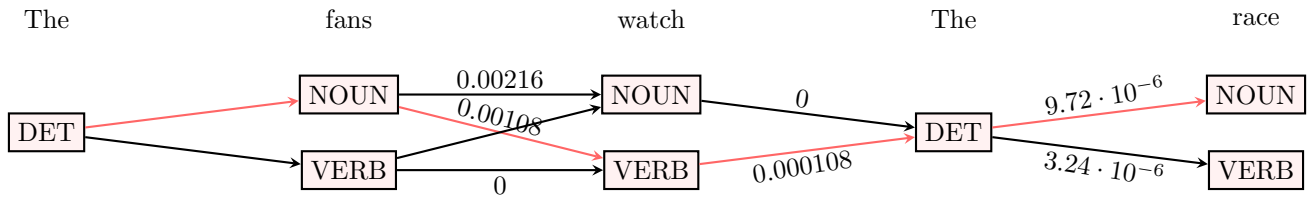
Heuristically, the reason being that every time a new state is added to a path, the update of the previous probability only depends on the very last stored POS tag. Indeed the emission probability is independent of all the previous sequences of states, whilst the transition probability only depends on the previous tag.

Clearly, this mechanism allows to explore only, hopefully, the most relevant subset among all possible sequences of tags, reducing significantly the computational cost of the procedure.

Visually,



Repeating the same steps, the whole sentence is evaluated:



2.3 The Viterbi Algorithm: theoretical justification

Setting

Let $(X_n)_{n\geq 0}$ be a Markov chain, the *state process*, whose transition probability matrix T is known.

Let $(Y_n)_{n\geq 0}$ be a stochastic process, the *observation process* s.t.

$$P(Y_n|X_{0:n}, Y_{0:n-1}) = P(Y_n|X_n)$$

The probabilities in the right hand-side of the expression above are called *emission probabilities* and are also assumed to be available.

Goal

Compute the most probable sequence of states $x_{0:n}$ given the observations $y_{0:n}$. Precisely

$$x^* = \underset{x}{\operatorname{argmax}}\, P(x|y), \quad \text{where } x = x_{0:n},\ y = y_{0:n}$$

Before going into the discussion, a couple of remarks:

Remark 2.2. *If* $f(a) \geq 0,\ \forall a$ *and* $g(a,b) \geq 0\, \forall a, b \implies \max_{a,b} f(a)g(a,b) = \max_a\{f(a) \max_b g(a,b)\}$

Remark 2.3. $P(x,y) = P(x|y)P(y)$, *hence* $\underset{x}{\operatorname{argmax}}\, P(x|y) = \underset{x}{\operatorname{argmax}}\, P(x,y)$ *and working with the joint probability is actually easier.*

Taking this into account, the idea is to firstly try to express the maximum (not the argmax) of the joint distribution in a recursive way. Exploiting Markov property:

$$\begin{aligned}
\mu_n(x_n) &= \max_{x_{0:n-1}} P(x_{0:n}, y_{0:n}) \\
&= \max_{x_{0:n-1}} P(y_n|x_{0:n}, y_{0:n-1})P(x_{0:n}, y_{0:n-1}) \\
&= \max_{x_{0:n-1}} P(y_n|x_{0:n}, y_{0:n-1})P(x_n|x_{0:n-1}, y_{0:n-1})P(x_{0:n-1}, y_{0:n-1}) \\
&= \max_{x_{0:n-1}} P(y_n|x_n)P(x_n|x_{n-1})P(x_{0:n-1}, y_{0:n-1}) \quad \text{by Markov property and conditional dependency property between X and Y} \\
&= \max_{x_{n-1}} \left\{ \underbrace{P(y_n|x_n)}_{emission} \underbrace{P(x_n|x_{n-1})}_{transition} \underbrace{\max_{x_{0:n-2}} P(x_{0:n-1}, y_{0:n-1})}_{\mu_{n-1}(x_{n-1})} \right\}
\end{aligned}$$

where the last inequality follows immediately from 2.2 with $a = x_{n-1}$ and $b = x_{0:n-2}$

So, the desired maximum can be expressed in the following recursive way:

$$\mu_n(x_n) = \max_{x_{n-1}} P(y_n|x_n)P(x_n|x_{n-1})\mu_{n-1}(x_{n-1})$$
$$\mu_0(x_0) = P(x_0, y_0) = P(x_0)P(y_0|x_0)$$

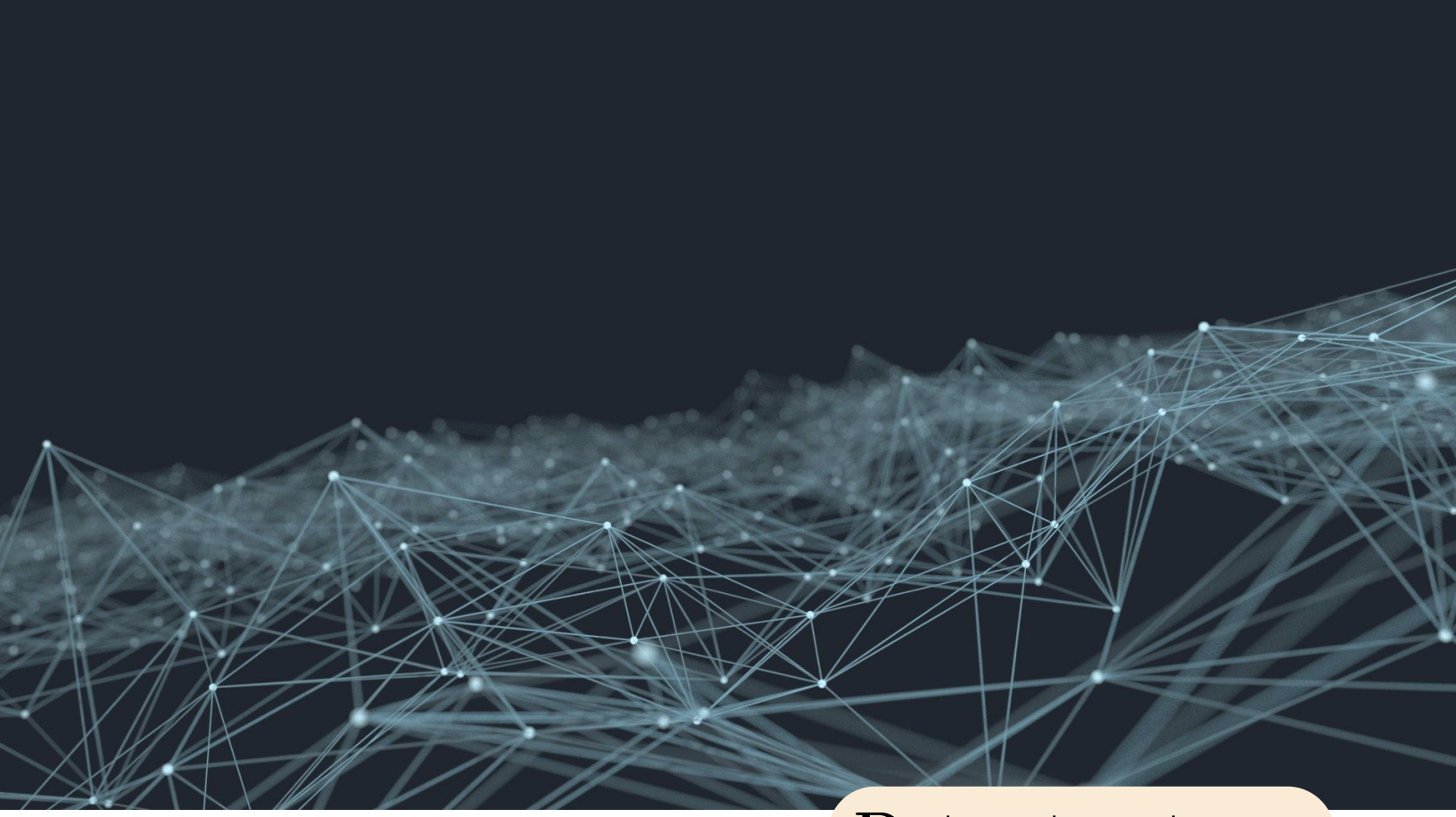
Finally, once $\mu_n(x_n)$ has been computed via recursion, the total maximum is simply obtained as:

$$\max_{x_n} \mu_n(x_n) = \max_{x_{0:n}} P(x_{0:n}, y_{0:n})$$

The argmax can be found exploiting the just enhanced recursive mechanism: at each time step the algorithm stores

$$x^*_{n-1} = \operatorname*{argmax}_{x_{n-1}} P(y_n|x_n)P(x_n|x_{n-1})\mu_{n-1}(x_{n-1})$$

that is, the $(n-1)^{th}$ state that leads to the highest probability $\mu_n(x_n)$.
Keeping everything into account, the Viterbi algorithm naturally exploits the dependency assumptions of HMMs to perform sequential updates of the "best" sequences of tags. Moreover, during the process, each tuple (x_n, y_n) is associated with a pointer to the previous state x^*_{n-1} in the current optimal path ending with it. Consequently, at the end of the procedure, the most probable sequence of tags can be retrieved by simply identifying the final most probable sequence and then 'backtracking' it by following the pointers.



Data structures

3. Data preparation

First of all, let's try to visualize better the data and the patterns among them, as well as define data structure that will be crucial in the following training and developing of the model.

Four dictionaries will be defined:

1. VOCABULARY:
   its keys are all the distinct words - sorted in lexicographic order - that are present in the training corpus. For each key, the value stores the correspondent index in the ordered list of training words.

2. TRANSITION_DICT:
   counts the number of times a tags happened next to another tag. The keys are tuples of the form $(prev\_tag, tag)$ and, as the name suggests, it will be crucial in computing the transition probability matrix.

3. EMISSION_DICT:
   stores how many times each pair $(word, tag)$ appears in the data. Those very pairs are the keys of the dictionary. It will be helpful to compute the emission probability matrix.

4. COUNTING_DICT:
   its keys are simply the 12 tags and the value associated to each tag is the corresponding frequency in the data.

Dictionaries have been chosen since in Python they are implemented as Hash Tables and this allows to perform all the basic operations - especially the ones of `get` and `in` - in $\mathcal{O}(1)$ average time-complexity.

Exploiting these dictionaries some summaries of the data can be computed. Bar plots are used to visualize them.

Figure 1 allows to visualize the frequencies of the different POS tags in the training set, while, next to it, there is the same graph but considering all the words in the Brown corpus. Then figure 3 represents the frequency of words in the training set grouped with respect to the number of different tags with which they appear in the data.

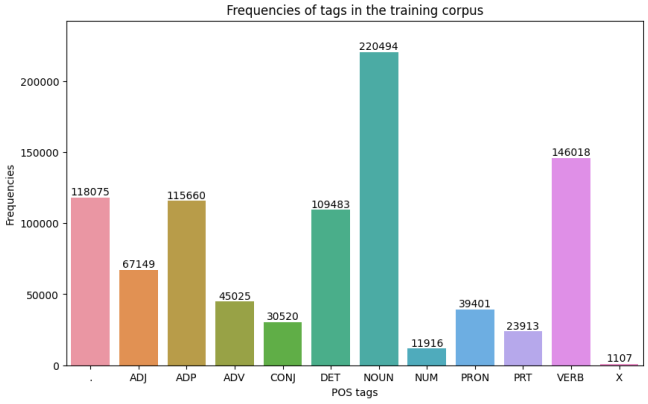


Figure 1: number of tokens in the training set associated to each POS tag.

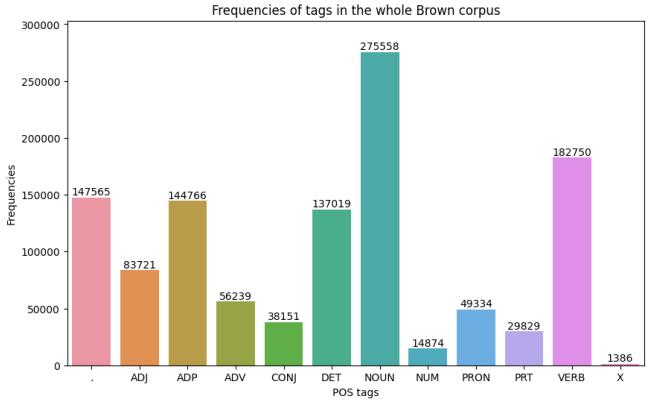


Figure 2: number of tokens in the whole data set associated to each POS tag.

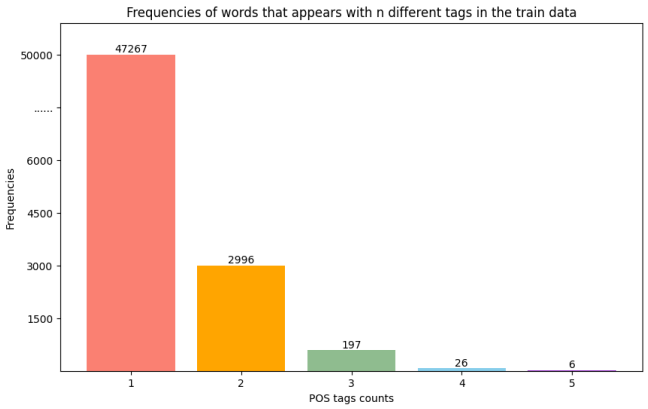


Figure 3: distribution of words associated with different numbers of POS tags in the training corpus

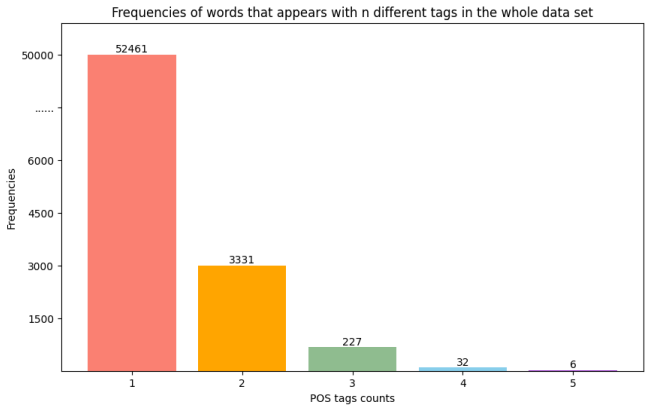


Figure 4: distribution of words associated with different numbers of POS tags in the whole corpus

Again, this is compared directly with the corresponding graph constructed using all the words in the data set. These comparisons make it possible to state that the training data are well representative of the entire dataset.

Similarly, in figure 5 we can visualize the most frequent tuples $(word, tag)$ and the most frequent transitions $(prev\_tag, tag)$ in figure 6:

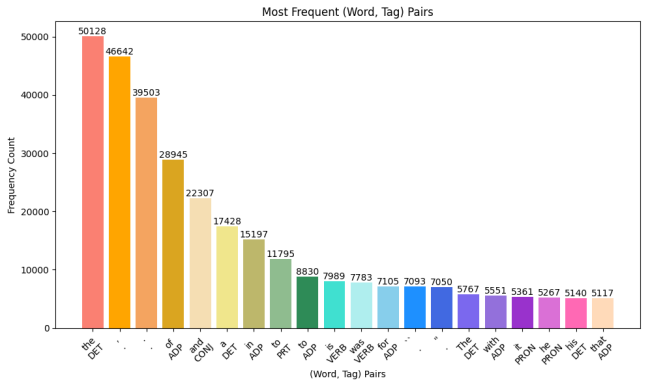


Figure 5: most frequent transitions $(prev\_tag, tag)$ evaluated on the training data.

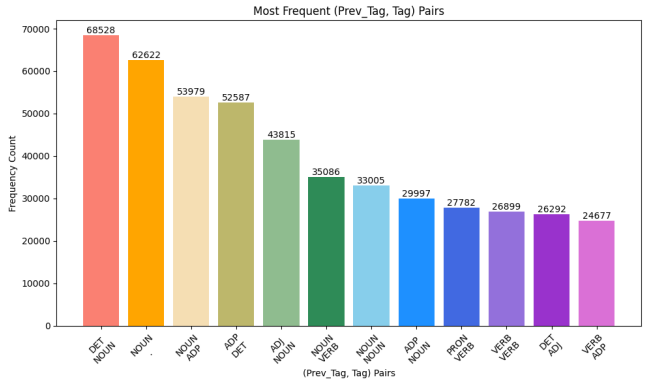


Figure 6: most frequent tuples $(word, tag)$.

Finally, relevant is also the presence of "unique" words - i.e. words that appear just one time in all the training corpus. In the analyzed case, there are 23374 unique words, most of which are proper names or compound names.

The presence of words with such a small frequency may pose challenges to the accuracy of the Viterbi algorithm, especially since the procedure relies on statistical patterns learned from the training data. If a token is unique, there is limited or even misleading statistical information available to guide the tag assignment process.

3.1 Transition and emission probability matrix

The transition and emission probability matrices are the main outcome of the training step and are both computed using the above dictionaries. As their names suggest, they store the two types of probabilities - the emission and the transition ones - needed by the Viterbi algorithm in its sequential procedure. More precisely:

1. Transition probability matrix:
   It is a $n_{tags} \times n_{tags}$ matrix that stores the probability that given a certain tag, $prev\_tag$, the following one is tag. Each entry of the matrix is evaluated as:

$$p(tag|prev\_tag) = \frac{\#\{\text{times the tag } tag \text{ is observed after } prev\_tag\}}{\#\{\text{times the tag } prev\_tag \text{ is observed}\}}$$

   Where $\#\{\text{times the tag } tag \text{ is observed after } prev\_tag\}$ corresponds to the value of the key $(prev\_tag, tag)$ in the dictionary `transition_dict` - if present, else it is zero - and $\#\{\text{times the tag } prev\_tag \text{ is observed}\}$ is the value of the key $prev\_tag$ in `counting_dict`.

2. Emission probability matrix:
   It is a $n_{tags} \times n_{words}$ matrix that stores the probability that given a certain tag, tag the corresponding word is $word$. The emission probability of $word$ given tag is computed as:

$$p(word|tag) = \frac{\#\{\text{times the pair } (word, tag) \text{ is observed}\}}{\#\{\text{times any pair } (\cdot, tag) \text{ is observed}\}}$$

   Where $\#\{\text{times the pair } (word, tag) \text{ is observed}\}$ corresponds to the value of the key $(word, tag)$ in dictionary `emission_dict` - if present, else it is zero - and $\#\{\text{times any pair } (\cdot, tag) \text{ is observed}\}$ corresponds to the value of the key tag in `counting_dict`.

The results can be visualized using heat maps.

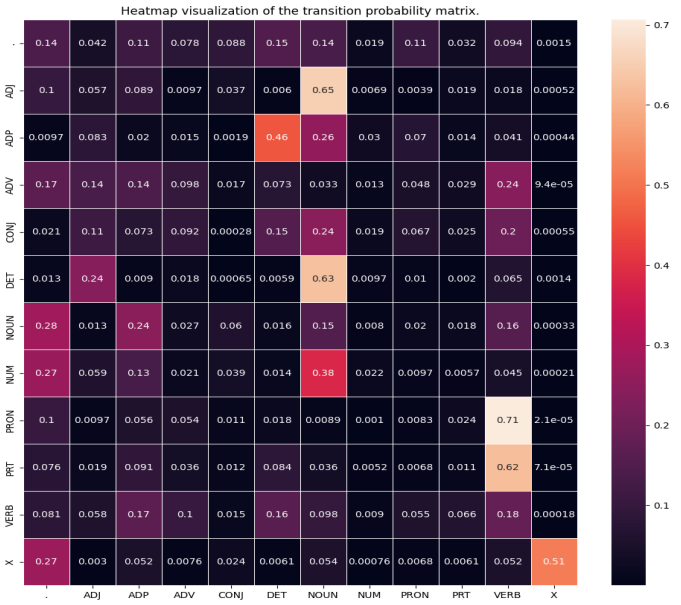


Figure 7: Heatmap visualization of the transition probability matrix

Figure 8: Heatmap visualisation of a portion of the natural logarithm of the emission probability matrix

For the emission probability matrix the natural logarithm of the entries have been considered - so that the stored numbers are more treatable - and only some columns of the matrix have been taken into account.

As it can be seen, the transition probability matrix is full for the considered training data. This may not be always the case, nevertheless, for a training data set to be considered adequate, each column is required to have at least one non-zero element.

3.2 Trellis graph

A trellis is a time-indexed graph in which each node at a certain time step is linked with at least a node at the previous step and at the following step. Due to this peculiar pattern a Trellis is able to capture the dependencies among the POS tags data and avoid redundant calculations. Focusing on its application in the Viterbi algorithm, each 'column' of the trellis grid-like structure represents a position (or a time step) within the input sentence, while each 'row' corresponds to a possible POS tag for that position. Each node also stores the probability associated to the 'current most probable path' up to that step. Finally, a trellis graph can also be used to obtain a meaningful visual representation of the states and the probabilities returned by the Viterbi procedure.

Practically, a trellis graph can be defined in Python by means of two classes:

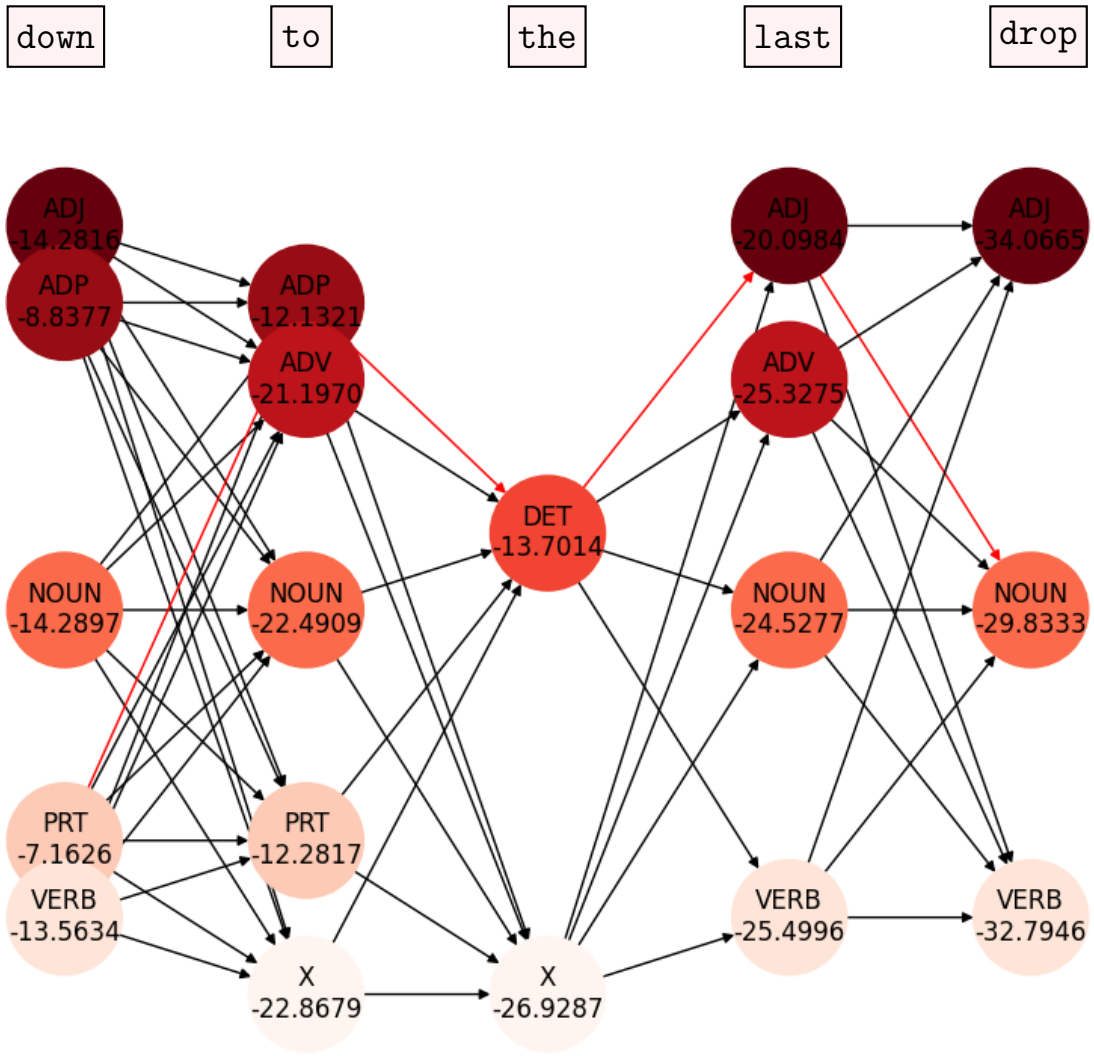
1. TrellisNode
   which depicts the structure of each node of the graph. A node has three different attributes:
   - state: corresponding to the POS tag represented by the node;
   - path_prob: the (natural logarithm of the) probability of the most probable path up to that state
   - backpointer: a reference to the previous node in the most probable path.
2. Trellis
   that contains - besides the constructor of the trellis graph - also some useful methods to add and get nodes at given time steps as well as to visualize the full graph.

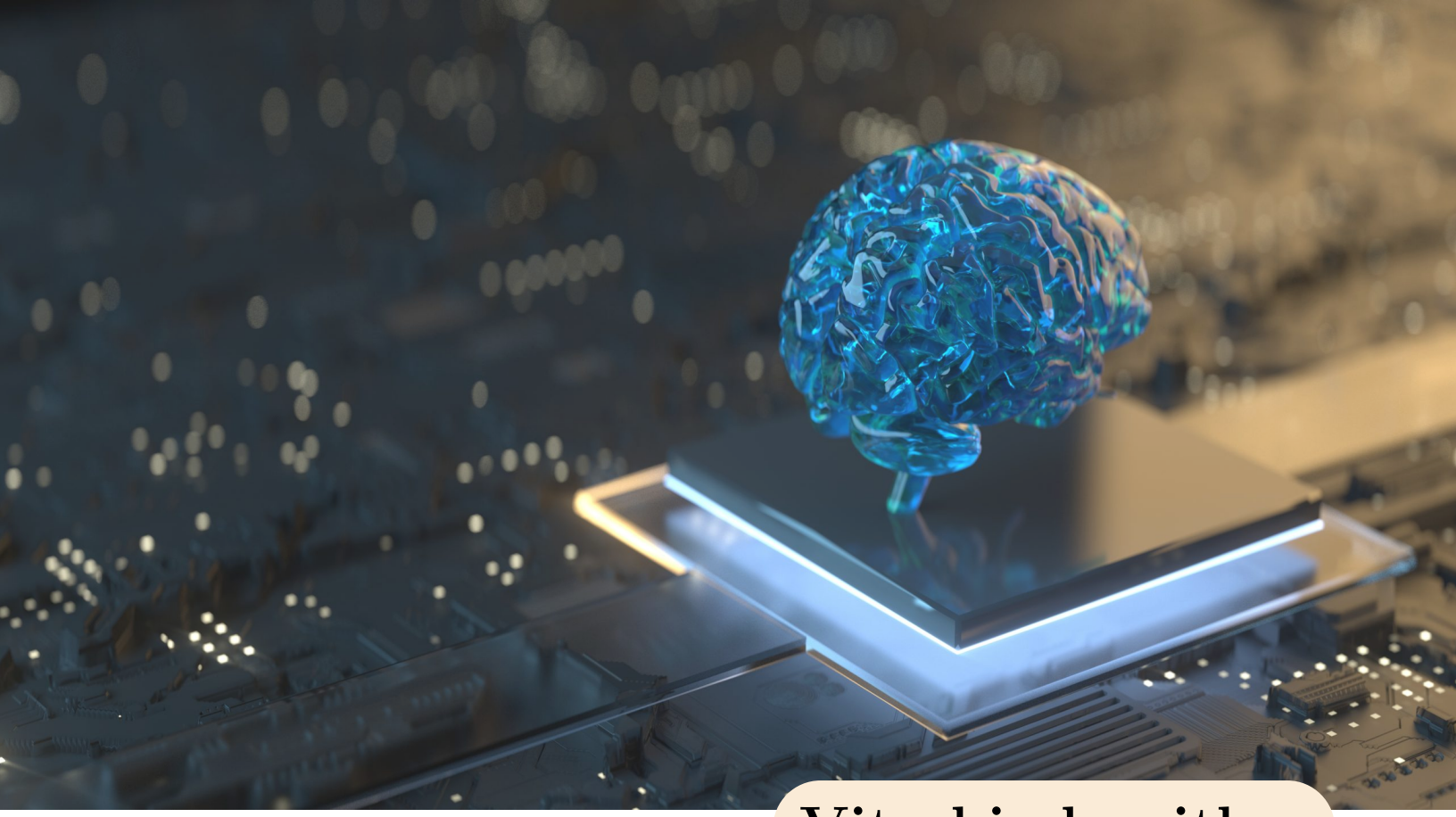
   Worth-noticing is the fact that, the `nodes` attribute of the `Trellis` class is a list of lists, where each inner list represents the nodes at a specific time step.

```python
class TrellisNode:
    def __init__(self, state, path_prob):
        self.state = state  # Represents the state (POS tag) at this node
        self.path_prob = path_prob  # Probability of the most probable path up to this node
        self.backpointer = None  # Pointer to the previous node with the highest probability

class Trellis:
    def __init__(self, num_time_steps):
        self.num_time_steps = num_time_steps
        self.nodes = [[] for _ in range(num_time_steps)]  # List of nodes at each time step

    def add_node(self, time_step, node):
        self.nodes[time_step].append(node)

    def get_nodes(self, time_step):
        return self.nodes[time_step]

    def visualize(self, tags, X):
        G = nx.DiGraph()

        # Add nodes to the graph
        for t in range(self.num_time_steps):
            for node in self.get_nodes(t):
                G.add_node((t, tags.index(node.state)), path_prob=node.path_prob)

        # Add edges to the graph and saves the "solution" path
        sol_path = []
        for t in range(1, self.num_time_steps):
            for current_node in self.get_nodes(t):
                for prev_node in self.get_nodes(t - 1):
                    G.add_edge((t - 1, tags.index(prev_node.state)),
                    (t, tags.index(current_node.state)))
                    if current_node.state == X[t] and prev_node.state == X[t-1] :
                        sol_path.append(((t - 1, tags.index(prev_node.state)),
                        (t, tags.index(current_node.state))))
```

```
        # Create layout and draw the graph
        pos = {node: (node[0], -node[1]) for node in G.nodes()}
        node_color = [node[1] for node in G.nodes()]
        cmap = plt.cm.Reds_r
        node_labels = {node: f"{tags[node[1]]}\n{G.nodes[node]['path_prob']:.4f}"
        for node in G.nodes()}
        edge_labels = {edge: "" for edge in G.edges()}
        edge_colors = ['black' if not edge in sol_path else 'red' for edge in G.edges()]
        plt.figure(figsize=(10, 8))
        nx.draw_networkx(G, pos, labels=node_labels, with_labels=True, node_color = node_color,
                         cmap = cmap, node_size=3000, edge_color = edge_colors)
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
        plt.xticks(range(self.num_time_steps))
        plt.yticks(range(-len(self.get_nodes(0)), 0))
        plt.axis("off")
        plt.xlabel("Time Step")
        plt.ylabel("State (POS Tag)")
        plt.savefig("Graph_ex.png", bbox_inches='tight')
        plt.show()
```

A visual representation over a small toy example is provided. The nodes corresponding to the same time step are drawn in columns and their height depends on the POS tag they represent. The edges represent the transitions between them. Finally, the 'global' most probable path returned by the POS tagging procedure has been highlighted in red.





Viterbi algorithm

4. Viterbi algorithm

Once both the emission and the transition probability matrices have been computed, the real heart of this POS tagging approach - the VITERBI ALGORITHM - can be implemented.

Two different versions of the Viterbi algorithm will be developed and compared. The first one is a "canonical" dynamical algorithm which exploits two $n_{tags} \times n_{words}$ matrices:

- the first one, T, stores in each entry $T[tag, word]$ the highest among the joint probabilities associated to all the possible paths up to the state $(word, tag)$.
- the second one, P, can be regarded as a *'backpointer' matrix* since $P[tag, word]$ saves the index of the previous tag leading to the maximum probability in $T[tag, word]$

The second one exploits the `Trellis` class described above.

4.1 Matrix-based implementation

The pseudo code is reported below after having introduced some useful notations.

The algorithm returns the most probable sequence of tags $X = (x_0, \dots, x_n)$, where $x_i \in tags = \{t_1, \dots, t_K\}$, $\forall i = 0, \dots n$ and $tags$ is the state space - i.e. the set of all possible POS tags.

Let $words = (w_0, \dots, w_n)$ be the sequence of observations, where $w_i \in vocab = \{v_1, \dots, v_N\}$, $\forall i = 1, \dots, n$ and $vocab$ is the observation space - i.e. the vocabulary of all *"known"* words.

Finally, consider $\pi = (\pi_1, \dots, \pi_K)$ the vector of initial probabilities, meaning that $\pi_i = P(x_0 = t_i)$, $\forall i = 1, \dots, K$.

Remark 4.1. *Actually, instead of arbitrarily assigning the initial probability of each tag t, one can simply consider it as the probability of having t given that the previous label is a punctuation, $'.'$. The reason being that the input sentences are assumed to be coherent and thus the first word can be considered as a normal word found after an end-of-sentence punctuation mark.*

Finally, the transition and emission probability matrices constructed above are respectively denoted as *tmp* and *emp*.

As already anticipated, the computation of the most probable sequence of tags is made by means of two matrices:

- T whose entry $T[i, j]$ stores the probability of the most probable path so far $\hat{X} = (\hat{x}_1, \ldots, \hat{x}_j)$ with $\hat{x}_j = t_i$, that generates the sequence of observations $(w_1, \ldots, w_j)$.
- P whose element $P[i, j]$ stores the tag $\hat{x}_{j-1}$ of the above most probable path. The entries of the first column are set all equal to 0.

Algorithm 1 Viterbi algorithm

```
procedure VANILLA_VITERBI(words, tags, pi, epm, tpm, vocab)
    for each tag t = 1, ..., k do
        T[t, 0] ← π[t] * epm[t, 0]
        P[t, 0] ← 0
    for each observation w = 1, ..., n do
        for each tag t = 1, ..., K do
            T[t, w] ← max{T[k, w − 1] * tpm[k, t] : k = 1, ..., K} * epm[t, w]
            P[t, w] ← argmax{T[k, w − 1] * tpm[k, t] : k = 1, ..., K}
    Z[n] ← argmax{T[k, n] : k = 1, ..., K}
    X[n] ← tags[Z[n]]
    for j = n, n − 1, ..., 1 do
        Z[j − 1] ← P[Z[j], j]
        X[j − 1] ← tags[Z[j − 1]]
    return X
```

The actual implementation of the algorithm has been divided into three parts:

1. Initialization;
2. Viterbi forward;
3. Viterbi backward.

Moreover, considerations linked to the particular context of interest have lead to some different implementation choices with respect to the ones presented in the pseudo code above.

Among all, the possibility of encountering new words - i.e. words that are not in the training set - needs to be taken into account. The problem is addressed in two different ways:

- The first one is highly naive: all unknown words are categorized as nouns, regardless of their features or context. The quite weak justification being that $'NOUN'$ is by far the most frequent POS tag and hence it is expected to be the most various and present category.
- The second approach, instead, incorporates a rule-based POS tagger in the structure of the Viterbi algorithm presented above, so that new words are no more classified as nouns by default, but are analyzed using the rule-based tagger.

Finally, in order to avoid storing very small numbers, the algorithm considers the natural logarithm of the most probable path probabilities. This allows to work with more treatable numbers without having any effect on the identification of the argmax.

4.1.1 Initialization

With reference to the notations introduced above, the objective of this first part is to properly initialize the matrices T and P.

Specifically, at the end of this step:

- P will be a $n_{tags} \times n_{words}$ empty matrix;
- T will be a $n_{tags} \times n_{words}$ whose only non-empty entries are the ones of the first column, corresponding to the vector of the initial probabilities for the first word, $word\_0$. For each tag t, these are evaluated as the sum of the natural logarithm of the transition probability from $'.'$ to t and the emission probability of $word\_0$ given t. If any of these two probabilities is zero, then the entry is simply set equal to $float('-inf')$. As already mentioned, for now unknown words are assumed to have $'NOUN'$ as the only possible associated tag - so with an emission probability equal to 1.

```python
def initializer(words, tags = tags, tpm = tpm_df, epm = epm_df, vocabulary = vocabulary) :
    # save the dimensions of tags and words
    n_tags = len(tags)
    n_words = len(words)

    # define the matrices
    T = np.empty((n_tags, n_words), dtype=np.float64)
    P = np.empty((n_tags, n_words), dtype=np.int64)

    # initialize the first column of T with the initial probabilities
    if words[0] not in vocabulary :
        T[:,0] = float('-inf')
        # unknown words up to know are treated as nouns
        ind_tag = tags.index('NOUN')
        T[ind_tag, 0] = math.log(tpm.loc['.', 'NOUN'])

    else :
        for ind_tag, tag in enumerate(tags) :
            em_prob = epm.loc[tag, words[0]]
            tr_prob = tpm.loc['.', tag]
            if (em_prob == float('-inf')) or (tr_prob == 0):
                T[ind_tag, 0] = float('-inf')
            else :
                # the initial probability T[ind_tag, 0] is equal to the sum of the
                # natural logarithms of the transition probability '.' -> tag and of the
                # emission tag -> words[0]
                T[ind_tag, 0] = math.log(tr_prob) + em_prob

    return T, P
```

4.1.2 Viterbi Forward

In this second part, the two matrices T and P are populated, following the recursive strategy explained above.

Algorithm 2 Viterbi forward algorithm

procedure VITERBI_FORWARD(words, T, P, epm, tpm)
  for each word in the corpus **do**
    for each POS tag type, tag, that this word may be **do**
      for each POS tag type, prev_tag, that the previous word could be **do**
        multiply the probability that the previous word is tagged prev_tag by the transition probability
        of tag given prev_tag, and by the probability that tag would emit the current word.
        retain the highest probability computed for the current word in tmp_prob
        retain the corresponding previous tag in tmp_prev
      $T[tag, word] \leftarrow tmp\_prob$
      $P[tag, word] \leftarrow tmp\_prev$
  return T, P

Briefly, the core of the algorithm consists in evaluating, sequentially for each word, the updated most probable path including also that word. To do so, for each tag tag and previous tag $prev\_tag$, the natural logarithm of the current transition $prev\_tag \rightarrow tag$ and emission probability $tag \rightarrow word$ are added to the best probability up to the tuple $(prev\_word, prev\_tag)$. The highest among all these is then retained in $T[current\_word, tag]$ and the corresponding previous tag is stored in $P[current\_word, tag]$.

Unknown words are still treated simply as nouns, consequently the emission probability for every other tag is assumed to be zero.

```python
def Viterbi_forward(words, T, P, tags = tags, tpm = tpm, epm = epm, vocabulary = vocabulary) :
    # number of POS tags and words
    n_tags = len(T[:,0])
    n_words = len(words)

    for j in range(1, n_words) :
        # if the word is known - hence at least one of its emission probabilities is not zero
        if words[j] in vocabulary :
            # for all tags
            for i in range(n_tags) :
                # save the logarithm of the emission probability
                em_prob = epm[i,vocabulary[words[j]]]
                # if the emission probability is 0, then also the joint probability of the path
                # up to this state will be 0
                if em_prob == float('-inf') :
                    T[i,j] = float('-inf')

                # if the emission probability is not 0
                else :
                    # initialize the tmp_prob and the tmp_prev for word i to -inf and None
                    tmp_prob_i = float('-inf')
                    tmp_prev_i = None

                    # For each POS tag that the previous word can be:
                    for k in range(n_tags):

                        if tpm[k,i] != 0 :
                            # Calculate the probability =
                            # T of POS tag k and previous word j-1 +
                            # log(prob of transition from POS k to POS i) +
                            # log(prob that emission of POS i is word k)
                            prob = T[k,j-1] + math.log(tpm[k,i]) + em_prob

                            # check if this path's probability is greater than the tmp probability
                            if prob > tmp_prob_i:

                                # Keep track of the best probability
                                tmp_prob_i = prob

                                # keep track of the index of the corresponding previous POS tag
                                tmp_prev_i = k

                    # Save the best probability up to the current tag i and word j
                    T[i,j] = tmp_prob_i
                    # Save the index of the previous tag in to the saved best path (if present)
                    if tmp_prev_i != None:
                        P[i,j] = tmp_prev_i

        else :
            # unknown words are tagged 'NOUN' with emission probability 1
            ind_tag = tags.index('NOUN')
            for i in range(n_tags) :
                T[i,j] = float('-inf')
                # if the tag is 'NOUN', the best probability and the corresponding previous tag
                # are saved in T and P respectively
                if i == ind_tag :
                    tmp_prob_i = float('-inf')
                    tmp_prev_i = None

                    for k in range(n_tags):
                        if tpm[k,i] != 0 :
                            prob = T[k,j-1] + math.log(tpm[k,i])

                            if prob > tmp_prob_i:
                                tmp_prob_i = prob
                                tmp_prev_i = k

                    T[i,j] = tmp_prob_i
                    P[i,j] = tmp_prev_i

    return T, P
```

4.1.3 Viterbi backward

Given T and P, returns the most probable sequence of POS tags for the considered words.

The sequence is retrieved by identifying the argmax of the last column of T and then tracing back the sequence using the backpointers stored in P. The reason being that, by construction, each entry $T[tag, last\_word]$ stores the probability of the most probable path ending with that tuple. Therefore their maximum is exactly the probability of the wanted sequence.

```
def Viterbi_backward(T, P, tags = tags) :

    # number of words
    n_words = len(T[0,:])

    X = np.empty(n_words, dtype='object')

    # compute the highest joint probability
    z = np.argmax(T[:,n_words-1])
    X[n_words-1]= tags[z]

    # retrieve the most probable sequence of POS tags using the backpointers stored in matrix P
    for w in range(n_words-1,0,-1) :
        z = P[z, w]
        X[w-1] = tags[z]

    return X
```

Remark 4.2. *Once the matrices P and T have been filled in the previous step, it is possible to reconstruct not only the most probable sequence of tags associated to all the n words, but also to retrieve the one that would have been obtained only providing the first, let's say, $m < n$ words. Simply run the* `Viterbi_backward` *algorithm with inputs $T[:, : m]$ and $P[:, : m]$*

4.1.4 Accuracy

Despite the quite naive choice of categorizing all unknown words as nouns, the obtained accuracy of the Viterbi algorithm is high:

```
# evaluate the accuracy of the Viterbi algorithm
correct = [t for t, t_vit in zip(test_tags, X) if t == t_vit]
accuracy = len(correct)/len(test_words)

print("Viterbi algorithm accuracy: ", accuracy)

>> Viterbi algorithm accuracy:  0.9622812791753251
```

Remark 4.3. *POS tagging algorithms are evaluated in terms of the percentage of correct tags. The standard assumption is that every word should be tagged with exactly one tag, which is scored as correct or incorrect: there are no marks for near misses.*

Generally there are some words which can be tagged in only one way, so are automatically counted as correct. Therefore, the error rate on a particular problem will be distributed very unevenly. For instance, a POS tagger will never confuse the tag $'PUN'$ with the tag $'VERB'$, but might confuse $'VERB'$ with $'ADJ'$ because there's a systematic ambiguity for many forms.

4.2 Viterbi algorithm + rule-based tagger

4.2.1 Rule-based tagger via nltk.RegexpTagger

There are several strategies to tackle better the problem of new words. As previously mentioned, the one adopted here consists in combining the stochastic approach to POS tagging with a rule-based one.

Precisely, the decision about unknown words is delegated to a different tagger relying on the presence, in the natural language, of some 'typical' features shared by a relevant portion of words in a certain POS class. Linguistic patterns, suffixes, prefixes or other cues are used to this end: for example, words ending with *'-ful', '-less', '-able', '-ous', ...* are likely to be adjectives, whereas the suffixes *'-ed', '-ing', '-es', ...* are usually used for verbs. Finally,

some quite frequent but particular words - like modal verbs or simple prepositions - are identified and categorized "manually".

In practice, such an approach is performed by means of the class `nltk.RegexpTagger`. Its constructor takes as input a list of handcrafted rules, each assigning to a linguistic pattern the related tag. The method `tag` can then be called: it takes as input a list of tokens, determines the most appropriate tag sequence by comparing each word to the series of provided regular expressions and returns the corresponding list of tagged words.

The set of handcrafted patterns is:

```
# specify patterns for tagging
patterns = [
    (r'(.|,|:|;|!)$', '.'),             # punctuations
    (r'(I|You|you|He|he|She|she|It|it|We|we|They|they)$', 'PRON'),  # pronomes
    (r'(But|but|And|and|Or|or)$', 'CONJ'),                          # common conjunctions
    (r'(Not|not)$', 'ADV'),             # adverbs
    (r'(That|that|This|this|Those|those|These|these)$', 'ADP'),     # common adpositions
    (r'(In|in|To|to|At|at|For|for|By|by|From|from|With|with)$', 'ADP'),
    (r'.*ing$', 'VERB'),                # gerund
    (r'.*ed$', 'VERB'),                 # past tense verbs
    (r'.*es$', 'VERB'),                 # singular present verbs
    (r'.*en$', 'VERB'),                 # verbs that mean 'cause to be'
    (r'.*ould$', 'VERB'),               # modal verbs
    (r'(Can|can|Will|will|May|may|might|might)$', 'VERB'),           # modal verbs
    (r'.*\'s$', 'NOUN'),                # possessive nouns
    (r'.*s$', 'NOUN'),                  # plural nouns
    (r'^-?[0-9]+(.[0-9]+)?$', 'NUM'),   # cardinal numbers
    (r'(The|the|A|a|An|an)$', 'DET'),   # articles or determinants
    (r'.*ly$', 'ADV'),                  # adverbs
    (r'.*able$', 'ADJ'),                # adjectives
    (r'.*ible$', 'ADJ'),
    (r'.*ful$', 'ADJ'),
    (r'.*less$', 'ADJ'),
    (r'.*ous$', 'ADJ'),
    (r'.*ive$', 'ADJ'),
    (r'.*ness$', 'NOUN'),               # nouns formed from adjectives
    (r'.*ment$', 'NOUN'),
    (r'.*ion$', 'NOUN'),
    (r'.*ship$', 'NOUN'),
    (r'.*ity$', 'NOUN'),
    (r'.*er$', 'NOUN'),
    (r'.*or$', 'NOUN'),
    (r'.*ee$', 'NOUN'),                 # adverbs
    (r'.*', 'NOUN')                     # nouns (default)
]
```

Given so, the rule-based tagger is simply defined as an instance of the class `nltk.RegexpTagger`:

```
# rule based tagger
rule_based_tagger = nltk.RegexpTagger(patterns)
```

4.2.2 Modified Viterbi algorithm

Using the newly defined POS tagger, the only two steps of the Viterbi algorithm to be modified are the initialization and the Viterbi forward. More precisely, in both procedures, the only change regards the tagging of unknown words: they are now classified accordingly to the rule-based tagger. It is therefore sufficient to report this new variation:

```
def initializer_ruled(words, tags = tags, tpm = tpm_df, epm = epm, vocabulary = vocabulary) :
    ...
    # initialize the first column of T with the "starting probabilities"
    if words[0] not in vocabulary :
        T[:,0] = float('-inf')
        # unknown words are treated using a rule-based approach
        [(_,rule_tag)] = rule_based_tagger.tag([words[0]])
        ind_rule_tag = tags.index(rule_tag)
        T[ind_rule_tag, 0] = math.log(tpm.loc['.', rule_tag])
    ...
```

```
def Viterbi_forward_ruled(words, T, P, tags = tags, tpm = tpm, epm = epm, vocabulary=vocabulary):
    ...
    else :
```

```
        # unknown words are treated using a rule-based approach
        [(_,rule_tag)] = rule_based_tagger.tag([words[j]])
        ind_rule_tag = tags.index(rule_tag)
        for i in range(n_tags) :
            T[i,j] = float('-inf')
            if i == ind_rule_tag :
                tmp_prob_i = float('-inf')
                tmp_prev_i = None

                for k in range(n_tags):
                    if tpm[k,i] != 0:
                        prob = T[k,j-1] + math.log(tpm[k,i])

                        if prob > tmp_prob_i:
                            tmp_prob_i = prob
                            tmp_prev_i = k

                T[i,j] = tmp_prob_i
                P[i,j] = tmp_prev_i

    ...
```

4.2.3 Accuracy

```
# evaluate the accuracy of the Viterbi algorithm
correct = [t for t, t_vit in zip(test_tags, X_r) if t == t_vit]
accuracy = len(correct)/len(test_words)

print("Viterbi algorithm accuracy: ", accuracy)

>> Viterbi algorithm accuracy:  0.9654090891490378
```

As expected accuracy increased, however slightly. There are a couple of reasons:

- The natural language has several subtleties and it is hard to fully depict them with a handful of lexical rules. Hence it would be unrealistic to believe that the introduced rule-based POS-tag correctly solves all unknown cases.
- The accuracy of the model does not merely depend on the handling of unknown words, but also on the quality of the training data, the presence of ambiguous or rare words, the effectiveness of the algorithm itself, etc.
- The accuracy of the algorithm was already quite high, suggesting an efficient tagging procedure, regardless of how unknown words are treated.

Clearly, to better quantify the improvement in accuracy due to the incorporation of the rule-based tagger, analysing a single test set is not enough. A cross-validation technique can return a better accuracy evaluation. In particular, it helps evaluate the model's ability to generalize beyond the training data and mitigate overfitting. The results of a 10-fold cross validation are reported below:

```
>> Fold 1
   Viterbi algorithm accuracy without the rule-base tagger:  0.9602110193596486
   Viterbi algorithm accuracy with the rule-base tagger:  0.9625043557183491

   Fold 2
   Viterbi algorithm accuracy without the rule-base tagger:  0.9592781921722761
   Viterbi algorithm accuracy with the rule-base tagger:  0.9613418505186714

   Fold 3
   Viterbi algorithm accuracy without the rule-base tagger:  0.9521390259600944
   Viterbi algorithm accuracy with the rule-base tagger:  0.9542759064578417

   Fold 4
   Viterbi algorithm accuracy without the rule-base tagger:  0.9608099035146614
   Viterbi algorithm accuracy with the rule-base tagger:  0.9630805905295001

   Fold 5
   Viterbi algorithm accuracy without the rule-base tagger:  0.9620569001011137
   Viterbi algorithm accuracy with the rule-base tagger:  0.9636354378077967
```

```
Fold 6
Viterbi algorithm accuracy without the rule-base tagger:  0.9572261398265786
Viterbi algorithm accuracy with the rule-base tagger:  0.9600011777349213

Fold 7
Viterbi algorithm accuracy without the rule-base tagger:  0.9606858771052051
Viterbi algorithm accuracy with the rule-base tagger:  0.9638150641949763

Fold 8
Viterbi algorithm accuracy without the rule-base tagger:  0.9615207347370077
Viterbi algorithm accuracy with the rule-base tagger:  0.9638589430848612

Fold 9
Viterbi algorithm accuracy without the rule-base tagger:  0.9571680758152932
Viterbi algorithm accuracy with the rule-base tagger:  0.9606406206448016

Fold 10
Viterbi algorithm accuracy without the rule-base tagger:  0.9582783261038117
Viterbi algorithm accuracy with the rule-base tagger:  0.9607091217702898
```

As it can be seen, the cross-validation technique confirms an increase in the Viterbi accuracy when unknown words are treated using a rule-based POS tagger.

Finally, the results for the test data can be compared to the ones on training words:

```
>> Viterbi algorithm accuracy:  0.975367182730541
```

The model overfits a bit the training data, one could consider adding some smoothing or regularization techniques.

4.3 Trellis-based Viterbi algorithm

All the main ideas and procedures behind the Viterbi algorithm are unchanged, the modifications concern only the data structure chosen to model the data.

Instead of using the two matrices T and P, the data are represented by means of the `Trellis` class introduced above. In particular, the nodes of the graph represent only the "reachable" tuples $(word, tag)$ and contain both the probability of the most probable path so far, and the backpointer to the previous tuple in that path. Moreover, the unknown words are directly treated using the rule-based approach.

Firstly, the trellis is initialized as a list of empty lists. Only the one corresponding to the first word is filled.

```python
def initializer_trellis(words, tags = tags, tpm = tpm_df, epm = epm_df, vocabulary = vocabulary) :
    n_words = len(words)

    # Create a trellis with the specified number of time steps, at each time step is associated
    # an empty list of nodes
    trellis = Trellis(n_words)

    # fill the list of nodes for the first word
    if words[0] not in vocabulary :
        # unknown words are treated using a rule-based approach
        [(_,rule_tag)] = rule_based_tagger.tag([words[0]])
        # the only possible tag is the one returned by the rule-based approach
        node = TrellisNode(rule_tag, math.log(tpm.loc['.', rule_tag]))
        trellis.add_node(0, node)

    else :
        # if the word is known, the corresponding list of nodes is populated
        for tag in tags :
            em_prob = epm.loc[tag, words[0]]
            tr_prob = tpm.loc['.', tag]
            if (em_prob != float('-inf')) and (tr_prob != 0):
                # define a node containing the tag and the log of the probability of the most
                # probable path up to the current state
                node = TrellisNode(tag, math.log(tr_prob) + em_prob)
                # add the node to the trellis at the current time step
                trellis.add_node(0, node)

    return trellis
```

Once the trellis has been constructed, it is populated using the same scheme than in `Viterbi_forward_ruled`.

```
def Viterbi_forward_trellis(words, trellis, tags = tags, tpm = tpm_df, epm = epm_df, vocabulary =
    vocabulary) :
    # for every test word
    for j in range(1, len(words)) :
        word = words[j]
        if word in vocabulary :
            for tag in tags :
                # retrieve the logarithm of the emission probability of tag -> word
                em_prob = epm.loc[tag,word]
                # if the emission probability is zero, do nothing. Else
                if em_prob != float('-inf') :
                    # Initialize best_prob for word i to negative infinity
                    tmp_prob = float('-inf')
                    # Initialize best_path for current word i to 0
                    tmp_prev = None

                    # For each node at the previous time step:
                    for prev_node in trellis.get_nodes(j - 1) :
                        tr_prob = tpm.loc[prev_node.state, tag]
                        # if the transition probability is zero, do nothing. Else
                        if tr_prob != 0 :
                            # evaluate the logarithm of the updated most probable path:
                            # log of the most probable path up to prev_node +
                            # log of the transition probability prev_tag -> tag +
                            # log of the emission probability tag -> word
                            prob = prev_node.path_prob + math.log(tr_prob) + em_prob

                            # check if this path's probability is greater than
                            # the best probability stored up to now
                            if prob > tmp_prob:
                                # Keep track of the best probability and of the previous node
                                tmp_prob = prob
                                tmp_prev = prev_node

                    # add a new node
                    node = TrellisNode(tag, tmp_prob)
                    node.backpointer = tmp_prev
                    trellis.add_node(j, node)

        else :
            # unknown words are treated using a rule-based approach
            [(_,rule_tag)] = rule_based_tagger.tag([word])
            tmp_prob = float('-inf')
            tmp_prev = None

            # rule_tag is assumed to be the only possible tag for the unknown word
            for prev_node in trellis.get_nodes(j - 1) :
                tr_prob = tpm.loc[prev_node.state, rule_tag]
                if tr_prob != 0 :
                    prob = prev_node.path_prob + math.log(tr_prob)

                    if prob > tmp_prob:
                        tmp_prob = prob
                        tmp_prev = prev_node

                # add the new node
                node = TrellisNode(rule_tag, tmp_prob)
                node.backpointer = tmp_prev
                trellis.add_node(j, node)

    return trellis
```

Finally, in the backtracking step, the procedure looks for the node at the last time step with the greatest saved probability, then retrieves the corresponding sequence exploiting the backpointer node attributes.

```
def Viterbi_backward_trellis(trellis, n_words) :

    # Backtracking to find the most likely sequence of states
    max_final_prob = float('-inf')
    max_final_node = None
```

```
    # find the node in the last list with the highest associated probability
    for node in trellis.get_nodes(n_words - 1):
        if node.path_prob > max_final_prob:
            max_final_prob = node.path_prob
            max_final_node = node

    current_node = max_final_node

    X = np.empty(n_words, dtype='object')
    X[n_words-1] = current_node.state

    for w in range(n_words-1,0,-1) :
        # the attribute backpointer of any node references the previous node in the path
        current_node = current_node.backpointer
        X[w-1] = current_node.state

    return X
```

As expected (and desired) the accuracy of the Viterbi procedure is unchanged: a different choice of data structures does not affect the efficiency of the algorithm.

```
# evaluate the accuracy of the Viterbi algorithm
correct = [t for t, t_vit in zip(test_tags, X_trellis) if t == t_vit]
accuracy = len(correct)/len(test_words)

print("Viterbi algorithm accuracy: ", accuracy)

>> Viterbi algorithm accuracy:  0.9654090891490378
```

A comparison of the two methods is postponed to keep into account the computational cost.

Computational cost

5. Computational cost

In the previous sections different versions of the Viterbi algorithm have been motivated, introduced and described. In the following their computational cost is evaluated.

The computational cost of an algorithm is a crucial information that allows to gain insight into its efficiency, performance and resource utilisation. Moreover, it also helps to understand how the algorithm will behave as the input size grows larger and so if it is suitable for large-scale problems. Finally, taking into account the computational cost is fundamental when comparing two algorithms: different algorithms may allocate spatial and temporal resources differently and this need to be considered when choosing the procedure that best meets the problem specifications.

Throughout the whole discussion, the following notation is used:

$T = \#\{\text{tags}\}$

$N = \#\{\text{words in the test set}\}$
i.e. the length of the sequence of words in the input of the Viterbi algorithm.

$K = \#\{\text{different pairs } (word, tag) \text{ in the training data}\}$

Moreover, it is worth-noticing that clearly k is surely greater or equal than the total number of distinct words in the training set, where equality is attained only if the training set is made up of all different words. Clearly, this corresponds to the worst possible case in terms of space complexity since the emission probability will be sparse with only one non-empty value per column [5]. For this reason in the following analysis we can always consider the number of distinct words in the training set as a $\mathcal{O}(K)$.

[5] Actually it also leads to a very poor estimation, since we do not have a complex and variegated enough set to account for all the subtleties of the natural language.

5.1 Emission and transition probability matrices

The creation of the emission and transition probability matrices in the training phase exploits the already constructed and ordered dictionaries.

Thanks to this the time complexity - as well as the space complexity - of the `emission_prob_matrix` procedure is $\mathcal{O}(T \times K)$, whilst the one of the `transition_prob_matrix` procedure is $\mathcal{O}(T \times T)$.

5.2 Viterbi algorithm: matrix-based implementation

Only the complexity of Viterbi algorithm with the integrated rule-based tagger will be discussed.

Remark 5.1. *The only difference with the previous version, in terms of computational cost, is related to the call to the* `nltk.RegexTagger`*: whereas arbitrarily assigning* $'NOUN'$ *to all unknown words clearly has a per-token-time-complexity of* $\mathcal{O}(1)$*,* `nltk.RegexTagger` *has a time complexity of* $\mathcal{O}(r \times n)$*, where* $r = \#\{$*regular expressions used*$\}$ *and* $n = \#\{$*words to be tagged*$\}$*.*

5.2.1 Initialization

The procedure requires as inputs the N test words, the set of tags *tags*, the $T \times T$ transition and the $T \times K$ emission probability matrices and finally the set of all known words (of size $\leq K$).

The noteworthy steps are:

- Initialization of the empty matrices of "sequential optimal probabilities", T, and of "backpointers", P. These operations are $\mathcal{O}(T \times N)$.
- Filling of the first column of T in $\mathcal{O}(T)$, if the first word in known. Instead, if the first word is unknown we need $\mathcal{O}(T+r)$, where $\mathcal{O}(r)$ is the computational cost of the rule-based tagger and $r = \#\{$regular expressions used$\}$.

So globally, this first step has a time complexity of $\mathcal{O}(T \times N + r)$.

As a side-note, it is important to underline that the convenience of using dictionaries starts to be evident: dictionaries are implemented as Hash-maps in Phyton and therefore all their main operations run in $\mathcal{O}(1)$ in the average case.

5.2.2 Viterbi Forward

The procedure takes as inputs the N test words, the two $T \times K$ matrices initialized in the previous step, the transition and emission probability matrices and the vocabulary of all training words.

To fill the matrices T and P three nested loops are required: the external one over the set of N test words, the other two over the tags. Since inside the loops all the operations - if statements, comparisons or assignments - run in $\mathcal{O}(1)$, the total time complexity is $\mathcal{O}(N \times T^2)$. The above analysis actually gives a quite pessimistic estimate:

- For all the unknown words, the inner loop runs only for exactly one of the tags in the previous iteration. However, to determine that tag the rule-based tagger is called and, on a single word, its computational cost is $\mathcal{O}(r)$. Therefore, for each unknown word the time complexity of the two inner loops is $\mathcal{O}(r+(T-1)\times 1+1\times T) = \mathcal{O}(r+T)$.
- The case of known words is more complex: the innermost for loop is performed only when the considered emission probability is strictly positive[6]. Now, since the training data are actually fully available, investigating a bit further their "composition" allows the time complexity estimate to be refined. In particular, 3492 words are actually ambiguous - meaning that they appear in the training set with more than one associated tag - and $m = 5$ is the maximum number of tags associated with one word. As a consequence, the complexity for the worst case scenario - given the training data - is $\mathcal{O}((T-m) \times 1 + m \times T) = \mathcal{O}(m \times T)$.

So, the complexity cost of the Viterbi forward procedure is $\mathcal{O}(N \times (m \times T + r))$.

[6] hence its natural logarithm is not $float('-inf')$.

5.2.3 Viterbi Backward

Takes as inputs the filled matrices T and P and the set of all possible tags.

The overall time complexity is $\mathcal{O}(T + N)$. Indeed the most probable POS tags associated to the given set of N words is re-constructed by searching the argmax in the last T-dimensional column of the T matrix, and then tracing back and retrieving the N sequence exploiting the "backpointers" stored in P.

Actually, the algorithm does not require as input the full matrix T, but only its last column. Nevertheless it has been left to favor the readability of the code.

5.2.4 Overall Viterbi complexity

Overall, the time complexity of the Viterbi procedure is $\mathcal{O}(N \times (m \times T + r))$. The parameter $m = \max \#\{\text{different tags associated with a single word}\}$ depends heavily on the specific training data:

- in the worst possible case, reasonably quite unrealistic, $m = T$;
- for the specific case of the data used throughout this project, $m = 5$. Actually, as figure 3 highlights in the whole training corpus there are just 6 words that appear with 5 different tags, the vast majority of words are associated with just one or at most two different tags. Assuming that the training tagged words are representative of the whole data set (as 4 seems to confirm) and keeping into account how unknown words are handled, one could say that, on average, $m = \frac{47267 \cdot 1 + 2996 \cdot 2 + 197 \cdot 3 + 26 \cdot 4 + 6 \cdot 5}{50492} = 1.06916 \approx 1$.

Its space complexity is $\mathcal{O}(T(T + K + N))$.

5.3 Viterbi algorithm: trellis-based implementation

5.3.1 Initialization

Still takes as inputs the sequence of test tokens to be analysed, the one of tags, the transition and emission probability matrices and finally the vocabulary of all known words.

The most expensive operations are the construction of the trellis, which require a time and spatial cost of $\Theta(N)$, and the population of the list of nodes corresponding to the first test word. This is - in the most general and pessimistic case and recalling the above definitions of m and r - $\mathcal{O}(\max\{m, r\})$.

Therefore, the overall computational cost of the initialization procedure is $\mathcal{O}(N + \max\{m, r\}) = \mathcal{O}(N + m + r)$.

5.3.2 Viterbi forward

The procedure requires as inputs the corpus of test words, the trellis initialized in the previous step, the set of all tags, the transition and emission probability matrices and, finally, the vocabulary containing all known words.

As in the matrix-based approach, the case of known words requires three nested loops to populate the trellis. Moreover, the conditions for the innermost cycle to be executed are exactly the same. The difference is that once executed, the loop is not on all tags, but only on the possible tags that the previous word may be or, in other words, on the list of nodes associated with the previous time step. Therefore the time complexity for the two innermost loops of a known word is $\mathcal{O}(m^2)$.

Turning to the case of unknown words, only one nested loop is needed: the intermediate one of the three above is unnecessary. the reason being that the only admissible tag for the current token is the one returned by the rule-based tagger. These operations have $\mathcal{O}(r + m)$ time complexity.

Keeping everything into account, the time complexity of the worst case is $\mathcal{O}(N \times \max\{m^2, mr\}) = \mathcal{O}(N \times (m^2 + r))$.

5.3.3 Viterbi backward

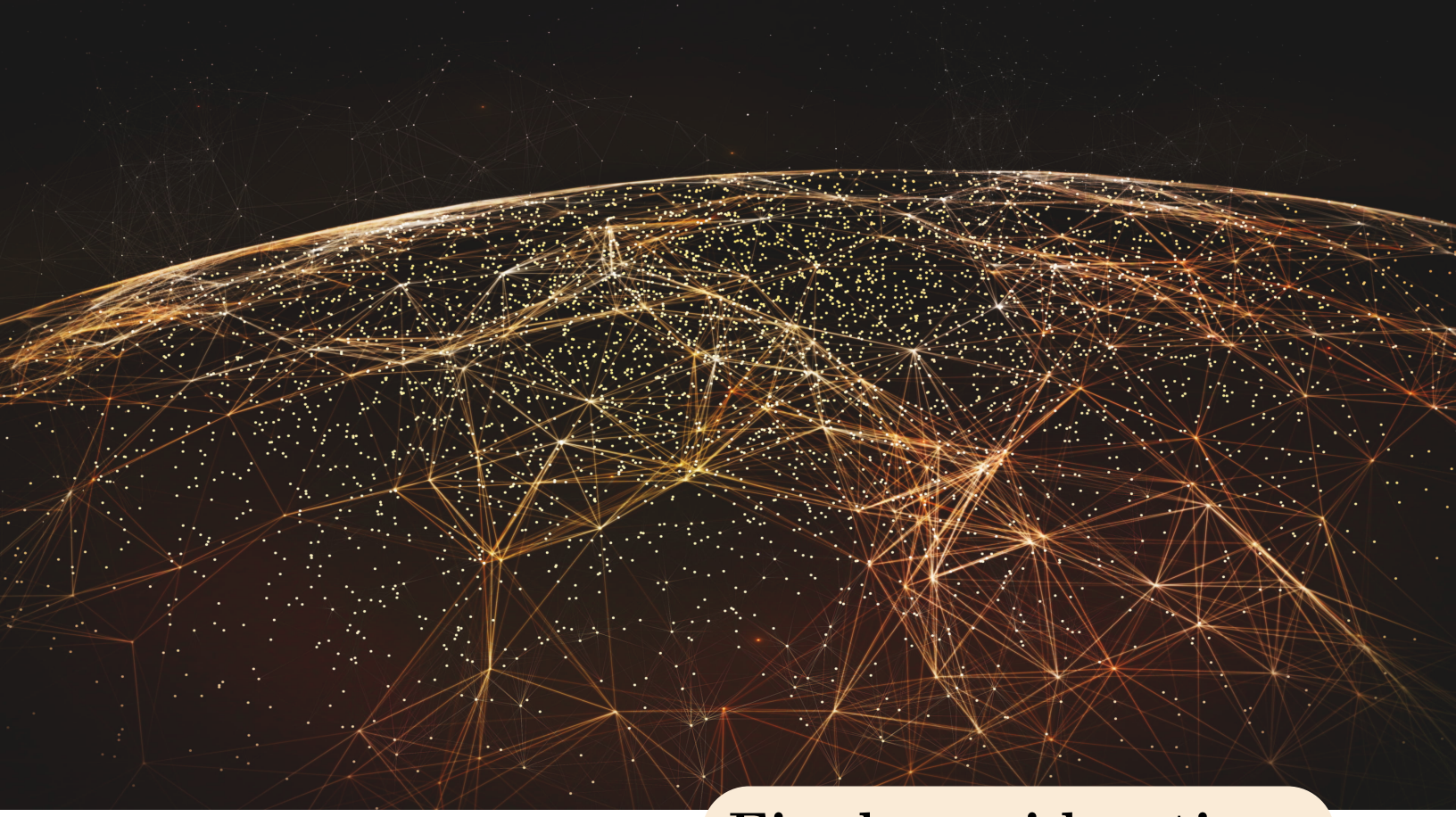
To retrive the most probable sequence of tags the trellis and the number of test words are required.

The procedure looks for the node at the last time step with the greatest saved probability in $\mathcal{O}(m)$, then backtracks the corresponding sequence in $\mathcal{O}(N)$.

5.3.4 Overall Viterbi complexity

Taking everything into account, the trellis-based Viterbi algorithm has a $\mathcal{O}(N \times (m^2 + r))$ computational cost, slightly better than the previous one. Also with regard to the space complexity, the trellis is $\mathcal{O}(N \times m)$, whereas the two matrices T and P above were $\mathcal{O}(T \times N)$.

Clearly, the same considerations apply to m as above.



Final considerations

6. Final considerations

The Viterbi algorithm has been implemented using both trellis graphs or matrices. Despite the underlying ideas and schemes being the same, some differences between the two approaches may be highlighted:

- Memory usage:
  The trellis-based implementation typically requires less memory since it only stores the reachable nodes.

- Computational cost:
  Despite the trellis implementation being theoretically slightly more efficient, other aspects need to be considered. Particularly important is the fact that in Python the numpy arrays are highly optimized and this positively affects the Viterbi matrix-based version with respect to the graph one. As a general guideline, it may be argued that for dense HMMs or situations where the transition and emission probabilities are readily available, the matrix implementation may be more efficient and hence preferable. Conversely, the usefulness of using a graph-based implementation becomes apparent when working with very complex models, like sparse or highly structured HMMs.

- Flexibility:
  In the current project, only a 'standard' HMM was applied to POS-tagging. However, extensions like higher-order Markov Models or lexicalized HMMs may improve model's performances and address some of its limitations. When handling such more complex models, the trellis graph implementation may provide more flexibility.

As far as the current project is concerned, since a simple HMM model with fixed state transitions and emission probabilities has been discussed, an implementation based on the Viterbi matrix can be opted for.

In conclusion, this POS tagging project employed Hidden Markov Models and the Viterbi algorithm to achieve accurate part-of-speech tagging. It involved learning the fundamentals of POS tagging and understanding how HMMs provide a statistical framework for modelling the sequential nature of language. Throughout the project some limitations of this approach were discussed and addressed. In particular, a rule-based tagger was integrated into the Viterbi algorithm to handle previously unseen words and improve model accuracy.

Conditional Random Fields

7. Extra: Conditional Random Fields

Despite HMMs being a powerful and widely used model, it turned out that it needs a number of augmentations to achieve high accuracy in POS tagging. Just think about unknown words or even newly created words: not only new acronyms and proper nouns, but even new common nouns and verbs enter the language at a surprising rate. To tackle these challenges, an idea might be to exploit arbitrary features based on morphological patterns. Despite it is possible to incorporate them inside a HMM, a more principled way is provided by Conditional Random Fields (CRFs).

Just like HMMs, CRFs provide a probabilistic approach to POS tagging. The strategies they use, however, are quite different:

- HMMs are *generative sequence models* and so aim to learn the joint probability distribution of input features and output variables.
- CRFs are *discriminative sequence models* and focus on learning the conditional probability distribution of the output variable given the input features. Therefore, they directly model the decision boundary between different classes or states without explicitly modeling the joint distribution.

To better internalize this difference, assume to have a sequence of test tokens $X = (x_1, \dots, x_n)$ for which a sequence of tags $Y = (y_1, \dots, y_n)$ has to be computed. In an HMM to compute the sequence of POS tags that maximizes $P(Y|X)$, one relies on Bayes' rule and the likelihood $P(X|Y)$:

$$\begin{aligned}\hat{Y} &= \underset{Y}{\operatorname{argmax}}\, p(Y|X) \\ &= \underset{Y}{\operatorname{argmax}}\, p(X|Y)p(Y) \\ &= \underset{Y}{\operatorname{argmax}} \prod_i p(x_i|y_i) \prod_i p(y_i|y_{i-1})\end{aligned}$$

In a CRF model, by contrast, the posterior $p(Y|X)$ is estimated directly, by training the CRF to discriminate among the possible tag sequences:

$$\hat{Y} = \underset{Y \in \mathcal{Y}}{\operatorname{argmax}}\, p(Y|X)$$

More precisely, a CRF is a log-linear model that assigns a probability to an entire sequence of tags Y, given the entire input word sequence X. Exactly as emphasized for HMMs, the power of this model lies in the ability of returning an accurate estimate of the solution by exploring only a significant subset of all possible sequences. Without going into all the details, CRFs satisfy an optimal substructure property that allows us to use a (polynomial-time) dynamic programming algorithm to find the optimal label, similar to the Viterbi algorithm for HMMs.

To build a CRF model, A set of *feature functions* to extract patterns for each input word needs to be defined. These are all the more detailed and specific the more complex the model. In this section, the focus is on *linear-chain CRFs* - i.e. models in which each local feature at position i can depend only on y_i (the tag of the current word), y_{i-1} (the tag of the previous word), X (the whole input sentence) and i (the position of the word in the sentence. Examples of possible features include evaluating prefixes and suffixes, asking if the word is capitalized and if it is the first or the last word of a sentence. The code used can be found at this Github repository.

In the training phase, a weight λ_j is assigned to each feature function f_j using (for example) a gradient descent with the L-BFGS method (lbfgs). Once obtained, the weights capture which are the most decisive and relevant characteristics to the task and determine the influence of each feature function on the CRF's decision-making process. In particular, a CRF model scores a labelling y for a given sentence x by adding up the k weighted features over all the n words in the sentence:

$$score(y|x) = \sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j f_j(x, i, y_i, y_{i-1})$$

The scores are then normalized.

To evaluate the accuracy, a F1 score has been used. The score on the test data is:

```
>> y_pred=crf.predict(X_test)
metrics.flat_f1_score(y_test,
    y_pred,average='weighted',labels=crf.classes_)
```

whereas the one on the training data:

```
>> y_pred_train=crf.predict(X_train)
metrics.flat_f1_score(y_train,
    y_pred_train,average='weighted',labels=crf.classes_)
```

It can be noticed that the considered CRF model overfit the training data, still it has higher accuracy than the previously considered HMM on the test data. This could be imagined, CRFs are in this context more powerful than HMMs: they can model everything HMMs does and more. Indeed, it is possible to build a CRF equivalent to any HMM.

In conclusion, this project offers a glimpse into the world of statistical machine learning, with a focus on natural language processing, showing the potential of statistical models in capturing language patterns and improving language analysis tasks, and fostering further exploration and understanding of the field.

References

Links

More information about the Brown Corpora dataset can be found here. NLTK documentation is also linked.

References

[1] Artzi Y. *Sequence Prediction and Part-of-speech Tagging.* Cornell University, 2017.

[2] Chen E. Introduction to conditional random fields, 2012.

[3] Copestake A. *Lectures on Natural Language Processes.* University of Cambridge, 2013.

[4] Johansen A. M. and Evers L. *Monte Carlo Methods.* Addison-Wesley, University of Bristol, Department of Mathematics – Statistics Group, University Walk, Bristol,BS8 1TW, UK., 2007.

[5] Jurafsky D. and Martin J.H. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition. Third Draft.* Prentice Hall, 2023.