Unit 3 Language Modelling

Contents

- Probabilistic language modelling
- Markov models
- Generative models of language
- Log Liner Models
- Graph-based Models N-gram models: Simple n-gram models
- Estimation parameters and smoothing
- Evaluating language models
- Word Embeddings/ Vector Semantics: Bag-of-words, TFIDF, word2vec, doc2vec
- Contextualized representations (BERT) Topic Modelling: Latent Dirichlet Allocation (LDA), Latent Semantic Analysis, Non Negative Matrix Factorization

Language Modelling

- **Building language models** is a **fundamental task** in natural language processing (NLP) that involves creating computational models capable of **predicting the next word** in a sequence of words.

- These models are essential for various **NLP applications**, such as machine translation, speech recognition, and text generation.

- A language model is a **statistical model** that is used to **predict the probability of a sequence of words.**

- It **learns the structure and patterns of a language** from a given text corpus and can be used to **generate new text** that is similar to the original text.

Probabilistic language modelling

- Probabilistic Language Modeling in Natural Language Processing (NLP) refers to the **process of assigning probabilities** to sequences of words in a language.

- It helps in **predicting the likelihood of a word sequence** and **estimating the probability of the next word**, given the preceding words.

- A probabilistic language model **assigns a probability** P(W) to a sequence of words W=w1,w2,...,wn representing **how likely that sequence is in the given language.**

Types of Probabilistic Models

- **N-gram Models**
  - Approximate the probability of a word sequence using the Markov assumption.
  - Example:

A bigram model estimates $P(wn \mid wn-1)$,

while a trigram model estimates $P(wn \mid wn-2, wn-1)$

Probability formula:

$P(W)=P(w1)P(w2 \mid w1)P(w3 \mid w2,w1)...P(wn \mid wn-1,...,wn-(N-1))$

Neural Language Models

- **Word2Vec** and **FastText**: Word embeddings that capture semantic relationships.
- **Recurrent Neural Networks (RNNs)** and **LSTMs**: Sequence models that improve context retention.
- **Transformer-based Models (BERT, GPT, T5, etc.)**: Use self-attention mechanisms for better contextual understanding.

Bayesian Language Models

- **Latent Dirichlet Allocation (LDA)**: Used for topic modelling.
- **Hidden Markov Models (HMMs)**: Useful in POS tagging and speech recognition.

Markov models

- Markov Models are widely used in Natural Language Processing (NLP) **for probabilistic modeling of sequential data**, such as text and speech.

Markovian Assumption

- These models assume that the **probability of a word or state** depends **only on the previous word(s) or state(s),** making them useful for tasks like speech recognition, part-of-speech tagging, and text generation.

- This simplifying assumption is known as the **Markovian assumption**, and it is a special case of a **one-Markov** or **first-order Markov assumption**.

Types of Markov Models in NLP

1. Markov Chains (MC)

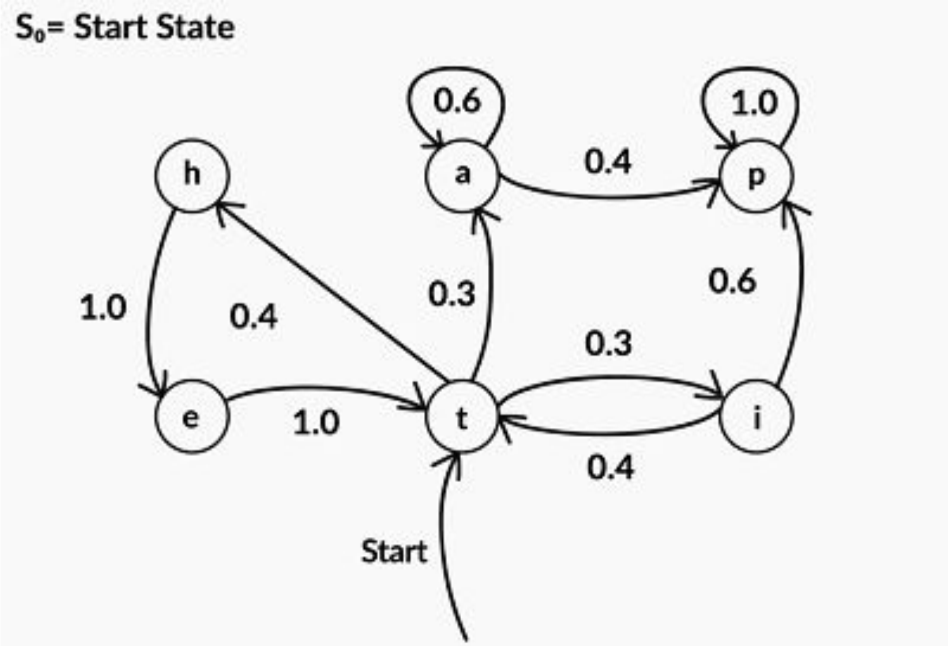
- A **simple** probabilistic model where the **next state depends only on** the **current state**.
- Used in **text generation** and **word prediction**.
- Example: **Bigram and Trigram Language Models**.

2. Hidden Markov Models (HMM)

- Extends Markov Chains by **introducing hidden states** that generate observable outputs.
- Used in **Part-of-Speech (POS) tagging**, **Named Entity Recognition (NER)**, and **speech recognition**.
- Example: **Tagging words as nouns, verbs, etc., based on context**

Markov Chain

- A Markov chain is a mathematical model that represents a process where the system transitions from one state to another.
- The transition assumes that the probability of moving to the next state is solely dependent on the current state.
- Please refer to the figure below for an illustration:



Letters- States

Numbers- Probabilities of going from one state to another

- Markov processes are commonly used to **model sequential data**, **like text** and speech.

- For instance, if you want to build an application that **predicts the next word** in a sentence, you can represent **each word** in a sentence as **a state.**

- The transition probabilities can be learned from a corpus and represent the probability of moving from the current word to the next word.

- Predictions: For example, the **transition probability from the state 'San' to 'Francisco'** will be higher than the probability of transitioning to the state **'Delhi'.**

Hidden Markov Model

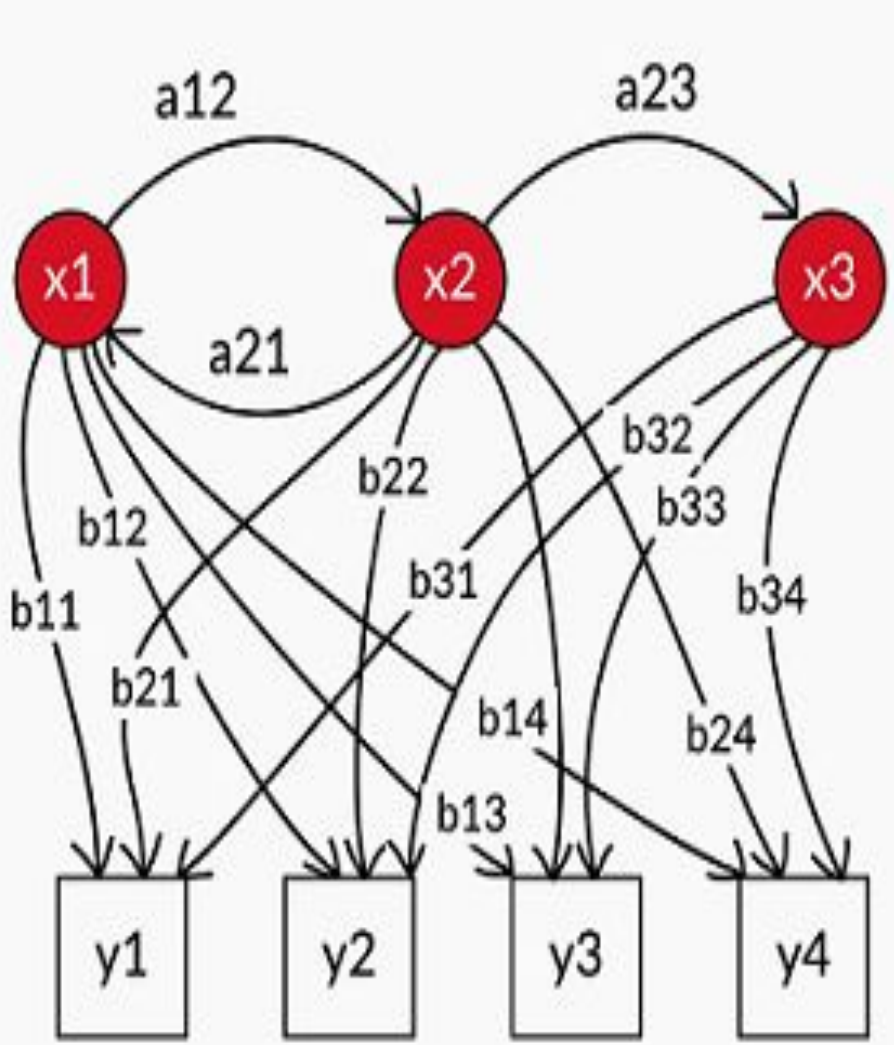
- The **Hidden Markov Model (HMM)** is an extension of the Markov process used to model phenomena where the **states are hidden** or latent, but they **emit observations**.

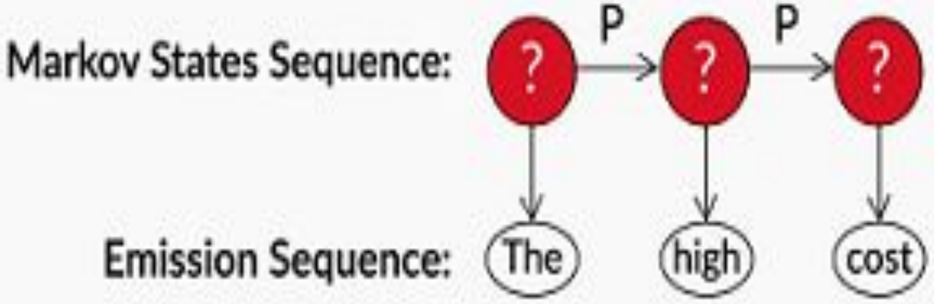
- For instance, in a **speech recognition system** like a **speech-to-text converter**, the **states represent the actual text words to predict**, but they are not directly observable (i.e., the states are hidden).

- Rather, **you only observe the speech** (audio) signals corresponding to each word and need to deduce the states using the observations.

- Similarly, in POS tagging, you observe the words in a sentence, but the **POS tags themselves are hidden.**
- Thus, the **POS tagging task** can be **modeled as a Hidden Markov Model** with the hidden states representing POS tags that emit observations, i.e., words.
- The **hidden states emit observations** with a certain probability.

- Therefore, **Hidden Markov Model has emission probabilities**, which represent the probability that a particular state emits a given observation.
- Along with the **transition and initial state probabilities**, these emission probabilities are used to model HMMs.
- The figure below illustrates the emission and transition probabilities for a hidden Markov process with three hidden states and four observations.





1. States: x1, x2, x3
2. Transition probability: a12, a21, a23
3. Output: y1, y2, y3, y4
4. Emission probability: b11, b21, b12, b13 ...
5. Sequence of words: y1, y2, y3, y4
6. POS tags: x1, x2, x3

- HMM can be trained using a variety of algorithms, including the **Baum-Welch algorithm** and the **Viterbi algorithm**.

- The **Baum-Welch algorithm** is an **unsupervised** learning algorithm that iteratively **adjusts the probabilities** of events occurring in each state **to fit the data better.**

- The **Viterbi algorithm is a dynamic programming algorithm** that finds the **most likely sequence of hidden states** given a sequence of observable events.

Viterbi Algorithm

- The Viterbi algorithm is a dynamic programming algorithm used to determine the most probable sequence of hidden states in a Hidden Markov Model (HMM) **based on a sequence of observations**.

- It is a **widely used** algorithm in **speech recognition, natural language processing**, and other areas that involve sequential data.

- The algorithm works by **recursively computing the probability of the most likely sequence of hidden states** that ends in each state for each observation.

- **At each step**, the **algorithm computes the probability of being in each state** and **emits the current observation** based on the probabilities of being in the previous states and making a transition to the current state.

- The Viterbi algorithm is an efficient and powerful tool that can handle long sequences of observations using dynamic programming.

Applications of the Hidden Markov Model

- Part-of-Speech (POS) Tagging
- Named Entity Recognition (NER)
- Speech Recognition
- Machine Translation

Limitations of the Hidden Markov Model

- HMM **assumes that the probability** of an event occurring in a certain state **is fixed**, which may not always be the case in real-world data.

- Additionally, HMM is **not well-suited for modeling long-term dependencies** in language, as it **only considers the immediate past.**

- There are alternative models to HMM in NLP, including recurrent neural networks (RNNs) and transformer models like BERT and GPT.

- These models have shown promising results in a variety of NLP tasks, but they also have their own limitations and challenges.

Implementation of the Hidden Markov Model in Python

In this Python implementation of HMM, we will try to code the Viterbi heuristic using the **tagged Treebank corpus**.

Exploring Treebank Tagged Corpus

```
#Importing libraries
import nltk, re, pprint
import numpy as np
import pandas as pd
import requests
import matplotlib.pyplot as plt
import seaborn as sns
import pprint, time
import random
from sklearn.model_selection import train_test_split
from nltk.tokenize import word_tokenize

# reading the Treebank tagged sentences
wsj = list(nltk.corpus.treebank.tagged_sents())

# first few tagged sentences
print(wsj[:40])
```

```
[[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'), ('old', 'JJ'), (',', ','), ('will', 'M
D'), ('join', 'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'
('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')], [('Mr.', 'NNP'), ('Vinken', 'NNP'), ('is', 'VBZ'), ('chairman', 'NN'), ('of', 'I
N'), ('Elsevier', 'NNP'), ('N.V.', 'NNP'), (',', ','), ('the', 'DT'), ('Dutch', 'NNP'), ('publishing', 'VBG'), ('group', 'N
N'), ('.', '.')], [('Rudolph', 'NNP'), ('Agnew', 'NNP'), (',', ','), ('55', 'CD'), ('years', 'NNS'), ('old', 'JJ'), ('and',
'CC'), ('former', 'JJ'), ('chairman', 'NN'), ('of', 'IN'), ('Consolidated', 'NNP'), ('Gold', 'NNP'), ('Fields', 'NNP'), ('PL
C', 'NNP'), (',', ','), ('was', 'VBD'), ('named', 'VBN'), ('*-1', '-NONE-'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('directo
r', 'NN'), ('of', 'IN'), ('this', 'DT'), ('British', 'JJ'), ('industrial', 'JJ'), ('conglomerate', 'NN'), ('.', '.')], [('A',
'DT'), ('form', 'NN'), ('of', 'IN'), ('asbestos', 'NN'), ('once', 'RB'), ('used', 'VBN'), ('*', '-NONE-'), ('*', '-NONE-'),
('to', 'TO'), ('make', 'VB'), ('Kent', 'NNP'), ('cigarette', 'NN'), ('filters', 'NNS'), ('has', 'VBZ'), ('caused', 'VBN'),
('a', 'DT'), ('high', 'JJ'), ('percentage', 'NN'), ('of', 'IN'), ('cancer', 'NN'), ('deaths', 'NNS'), ('among', 'IN'), ('a',
'DT'), ('group', 'NN'), ('of', 'IN'), ('workers', 'NNS'), ('exposed', 'VBN'), ('*', '-NONE-'), ('to', 'TO'), ('it', 'PRP'),
('more', 'RBR'), ('than', 'IN'), ('30', 'CD'), ('years', 'NNS'), ('ago', 'IN'), (',', ','), ('researchers', 'NNS'), ('reporte
d', 'VBD'), ('0', '-NONE-'), ('*T*-1', '-NONE-'), ('.', '.')], [('The', 'DT'), ('asbestos', 'NN'), ('fiber', 'NN'), (',',
','), ('crocidolite', 'NN'), (',', ','), ('is', 'VBZ'), ('unusually', 'RB'), ('resilient', 'JJ'), ('once', 'IN'), ('it', 'PR
P'), ('enters', 'VBZ'), ('the', 'DT'), ('lungs', 'NNS'), (',', ','), ('with', 'IN'), ('even', 'RB'), ('brief', 'JJ'), ('expos
ures', 'NNS'), ('to', 'TO'), ('it', 'PRP'), ('causing', 'VBG'), ('symptoms', 'NNS'), ('that', 'WDT'), ('*T*-1', '-NONE-'),
('show', 'VBP'), ('up', 'RP'), ('decades', 'NNS'), ('later', 'JJ'), (',', ','), ('researchers', 'NNS'), ('said', 'VBD'),
('0', '-NONE-'), ('*T*-2', '-NONE-'), ('.', '.')], [('Lorillard', 'NNP'), ('Inc.', 'NNP'), (',', ','), ('the', 'DT'), ('uni
```

As we can observe from the above output, each word in the sentence is tagged with its corresponding POS tag.

Train Test Split

In this step, we will split the dataset into a 70:30 ratio i.e., 70% of the data for the training set and the rest 30% for the test set.

```
# Splitting into train and test
random.seed(1234)
train_set, test_set = train_test_split(wsj,test_size=0.3)

print(len(train_set))
print(len(test_set))
print(train_set[:40])
```

```
2739
1175
[[('Primerica', 'NNP'), ('closed', 'VBD'), ('at', 'IN'), ('$', '$'), ('28.25', 'CD'), ('*U*', '-NONE-'), (',', ','), ('down',
'RB'), ('50', 'CD'), ('cents', 'NNS'), ('.', '.')], [('The', 'DT'), ('recent', 'JJ'), ('quarter', 'NN'), ('includes', 'VBZ'),
('pretax', 'NN'), ('gains', 'NNS'), ('of', 'IN'), ('$', '$'), ('98', 'CD'), ('million', 'CD'), ('*U*', '-NONE-'), ('from', 'I
N'), ('asset', 'NN'), ('sales', 'NNS'), (',', ','), ('while', 'IN'), ('like', 'JJ'), ('gains', 'NNS'), ('in', 'IN'), ('the',
'DT'), ('year-earlier', 'JJ'), ('quarter', 'NN'), ('totaled', 'VBD'), ('$', '$'), ('61', 'CD'), ('million', 'CD'), ('*U*', '-
NONE-'), ('.', '.')], [('But', 'CC'), ('rather', 'RB'), ('than', 'IN'), ('*', '-NONE-'), ('sell', 'VB'), ('new', 'JJ'), ('30-
year', 'JJ'), ('bonds', 'NNS'), (',', ','), ('the', 'DT'), ('Treasury', 'NNP'), ('will', 'MD'), ('issue', 'VB'), ('$', '$'),
('10', 'CD'), ('billion', 'CD'), ('*U*', '-NONE-'), ('of', 'IN'), ('29year', 'JJ'), (',', ','), ('nine-month', 'JJ'), ('bond
s', 'NNS'), ('--', ':'), ('*-2', '-NONE-'), ('essentially', 'RB'), ('increasing', 'VBG'), ('the', 'DT'), ('size', 'NN'), ('o
f', 'IN'), ('the', 'DT'), ('current', 'JJ'), ('benchmark', 'NN'), ('30-year', 'JJ'), ('bond', 'NN'), ('that', 'WDT'), ('*T*-
3', '-NONE-'), ('was', 'VBD'), ('sold', 'VBN'), ('*-1', '-NONE-'), ('at', 'IN'), ('the', 'DT'), ('previous', 'JJ'), ('refundi
ng', 'NN'), ('in', 'IN'), ('August', 'NNP'), ('.', '.')], [('France', 'NNP'), ('can', 'MD'), ('boast', 'VB'), ('the', 'DT'),
('lion', 'NN'), ("'s", 'POS'), ('share', 'NN'), ('of', 'IN'), ('high-priced', 'JJ'), ('bottles', 'NNS'), ('.', '.')], [('Rich
ard', 'NNP'), ('Driscoll', 'NNP'), (',', ','), ('vice', 'NN'), ('chairman', 'NN'), ('of', 'IN'), ('Bank', 'NNP'), ('of', 'I
N'), ('New', 'NNP'), ('England', 'NNP'), (',', ','), ('told', 'VBD'), ('the', 'DT'), ('Dow', 'NNP'), ('Jones', 'NNP'), ('Prof
essional', 'NNP'), ('Investor', 'NNP'), ('Report', 'NNP'), (',', ','), ('``', '``'), ('Certainly', 'RB'), (',', ','), ('ther
e', 'EX'), ('are', 'VBP'), ('those', 'DT'), ('outside', 'IN'), ('the', 'DT'), ('region', 'NN'), ('who', 'WP'), ('*T*-159', '-
```

From the above output, we can observe that the total number of training records is 2739, and the test set has 1175. The top 40 sentences are also shown in which each word is tagged to its POS tag.

Next, we will check the number of tagged words in the training set to understand how much data will be used for training the POS tagger.

```
# Getting list of tagged words
train_tagged_words = [tup for sent in train_set for tup in sent]
len(train_tagged_words)
```

Output

```
70503
```

Next, we will create a tokens variable that will contain all the tokens from the train_tagged_words. Furthermore, we also need to create a vocabulary and set of all unique tags in the training data.

```
# tokens
tokens = [pair[0] for pair in train_tagged_words]
# vocabulary
V = set(tokens)
print("Total vocabularies: ",len(V))
# number of tags
T = set([pair[1] for pair in train_tagged_words])
print("Total tags: ",len(T))
```

Output

```
Total vocabularies: 10236
Total tags: 46
```

Next, we will use HMM algorithm to tag the words.

- Given a sequence of words to be tagged, the task is to assign the most probable tag to the word.

- In other words, to every word w, **assign the tag t that maximizes the likelihood P(t/w).** Since P(t/w) = P(w/t). P(t) / P(w), after ignoring P(w), we have to compute P(w/t) and P(t).

- P(w/t) is basically the **probability that given a tag (say NN), what is the probability of it being w (say 'building').** This can be computed by computing the fraction of all NNs which are equal to w, i.e.
- P(w/t) = count(w, t) / count(t).

Limitations of Markov Models in NLP

1. **Limited Context Awareness**: Markov models **rely only on recent history** and struggle with **long-term dependencies.**
2. **Data Sparsity**: Higher-order models (e.g., trigram models) require large datasets.
3. **Inability to Capture Deep Semantic Meaning**: Unlike transformers (BERT, GPT), Markov models **do not understand context deeply**.

Comparison with Modern NLP Models

While Markov Models remain useful for simple NLP tasks, they have largely been replaced by **neural networks and transformers** (e.g., BERT, GPT) for complex language understanding.

Feature	Markov Models	Deep Learning (Transformers)
Context Length	Short-term	Long-term
Interpretability	High	Lower
Accuracy	Lower	Higher
Scalability	Limited	Better with more data

- The term P(t) is the probability of tag t, and in a tagging task, we assume that a tag will depend only on the previous tag.
- In other words, the **probability of a tag being NN will depend only on the previous tag t(n-1).**
- So for e.g. if t(n-1) is a JJ, then t(n) is **likely to be an NN since adjectives often precede a noun (blue coat, tall building, etc.).**
- Given the Penn treebank tagged dataset, we can compute the two terms P(w/t) and P(t) and store them in two large matrices.
- The matrix of P(w/t) will be sparse since each word will not be seen with most tags ever, and those terms will thus be zero.

Emission probabilities

```
# computing P(w/t) and storing in T x V matrix
t = len(T)
v = len(V)
w_given_t = np.zeros((t, v))

# compute word given tag: Emission Probability
def word_given_tag(word, tag, train_bag = train_tagged_words):
    tag_list = [pair for pair in train_bag if pair[1]==tag]
    count_tag = len(tag_list)
    w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]
    count_w_given_tag = len(w_given_tag_list)

    return (count_w_given_tag, count_tag)

# examples

# large
print("\n", "large")
print(word_given_tag('large', 'JJ'))
print(word_given_tag('large', 'VB'))
print(word_given_tag('large', 'NN'), "\n")

# will
print("\n", "will")
print(word_given_tag('will', 'MD'))
print(word_given_tag('will', 'NN'))
print(word_given_tag('will', 'VB'))

# book
print("\n", "book")
print(word_given_tag('book', 'NN'))
print(word_given_tag('book', 'VB'))
```

```
 large
(20, 4056)
(0, 1809)
(0, 9240)

 will
(210, 681)
(0, 9240)
(0, 1809)

 book
(4, 9240)
(0, 1809)
```

Transition Probabilities

```
# compute tag given tag: tag2(t2) given tag1 (t1), i.e. Transition Probability

def t2_given_t1(t2, t1, train_bag = train_tagged_words):
    tags = [pair[1] for pair in train_bag]
    count_t1 = len([t for t in tags if t==t1])
    count_t2_t1 = 0
    for index in range(len(tags)-1):
        if tags[index]==t1 and tags[index+1] == t2:
            count_t2_t1 += 1
    return (count_t2_t1, count_t1)

# examples
print(t2_given_t1(t2='NNP', t1='JJ'))
print(t2_given_t1('NN', 'JJ'))
print(t2_given_t1('NN', 'DT'))
print(t2_given_t1('NNP', 'VB'))
print(t2_given_t1(',', 'NNP'))
print(t2_given_t1('PRP', 'PRP'))
print(t2_given_t1('VBG', 'NNP'))
```

```
(139, 4056)
(1840, 4056)
(2696, 5707)
(65, 1809)
(1035, 6687)
(2, 1155)
(6, 6687)
```

```
#Please note P(tag|start) is same as P(tag|'.')
print(t2_given_t1('DT', '.'))
print(t2_given_t1('VBG', '.'))
print(t2_given_t1('NN', '.'))
print(t2_given_t1('NNP', '.'))
```

```
(597, 2712)
(11, 2712)
(111, 2712)
(507, 2712)
```

Next, we will create a transition matrix of tags of dimension txt

```python
# creating t x t transition matrix of tags
# each column is t2, each row is t1
# thus M(i, j) represents P(tj given ti)

tags_matrix = np.zeros((len(T), len(T)), dtype='float32')
for i, t1 in enumerate(list(T)):
    for j, t2 in enumerate(list(T)):
        tags_matrix[i, j] = t2_given_t1(t2, t1)[0]/t2_given_t1(t2, t1)[1]

tags_matrix
```

```
array([[ 6.04043379e-02,  3.42702158e-02,  2.24358980e-02, ...,
         4.19132132e-03,  9.86193307e-04,  0.00000000e+00],
       [ 9.27172136e-03,  3.81785542e-01,  1.94407050e-02, ...,
         2.84133386e-03,  1.49543892e-04,  1.49543892e-04],
       [ 3.01348139e-02,  1.50674069e-02,  1.84377477e-01, ...,
         1.18953211e-03,  3.96510703e-04,  0.00000000e+00],
       ...,
       [ 2.07114071e-01,  1.25109509e-01,  2.48817243e-02, ...,
         1.40178727e-03,  0.00000000e+00,  0.00000000e+00],
       [ 1.21739127e-01,  7.82608688e-02,  1.73913036e-02, ...,
         2.69565225e-01,  0.00000000e+00,  8.69565178e-03],
       [ 4.16666679e-02,  2.77777780e-02,  1.38888890e-02, ...,
         2.01388896e-01,  0.00000000e+00,  0.00000000e+00]], dtype=float32)
```

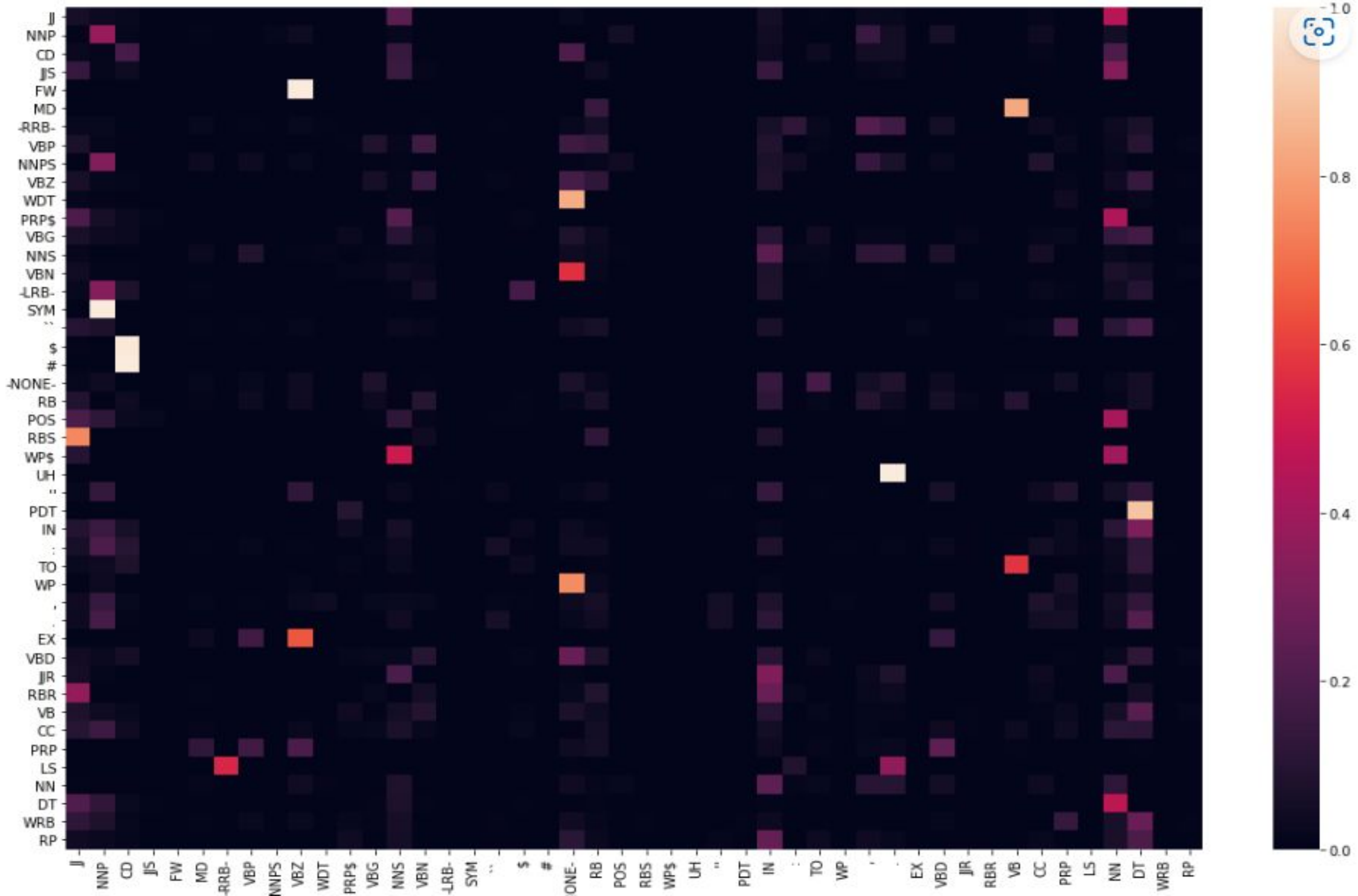
As tags are not visible in this matrix, we will now convert it into pandas dataframe for better readability.

```
# convert the matrix to a df for better readability
tags_df = pd.DataFrame(tags_matrix, columns = list(T), index=list(T))
tags_df
```

	JJ	NNP	CD	JJS	FW	MD	-RRB-	VBP	NNPS	VBZ	...	JJR	RBR	VB	CC
JJ	0.060404	0.034270	0.022436	0.000493	0.000000	0.000000	0.000247	0.000493	0.000986	0.000986	...	0.000740	0.000247	0.000000	0.014793
NNP	0.009272	0.381786	0.019441	0.000000	0.000000	0.010917	0.003739	0.003739	0.017945	0.036489	...	0.000150	0.000000	0.000748	0.038582
CD	0.030135	0.015067	0.184377	0.000793	0.000000	0.001983	0.000793	0.002379	0.000000	0.001586	...	0.001190	0.000397	0.000000	0.013878
JJS	0.147287	0.015504	0.038760	0.000000	0.000000	0.000000	0.000000	0.007752	0.007752	0.007752	...	0.000000	0.000000	0.000000	0.007752
FW	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	...	0.000000	0.000000	0.000000	0.000000
MD	0.001468	0.001468	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.001468	0.825257	0.000000
-RRB-	0.021053	0.021053	0.000000	0.000000	0.000000	0.021053	0.000000	0.010526	0.000000	0.021053	...	0.000000	0.000000	0.000000	0.031579
VBP	0.074157	0.014607	0.004494	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001124	...	0.005618	0.004494	0.001124	0.003371
NNPS	0.000000	0.331461	0.000000	0.000000	0.000000	0.033708	0.000000	0.033708	0.000000	0.022472	...	0.000000	0.000000	0.000000	0.089888
VBZ	0.063165	0.019282	0.014628	0.000665	0.000000	0.000000	0.000000	0.000000	0.000000	0.000665	...	0.007314	0.003324	0.002660	0.003324
WDT	0.015674	0.009404	0.006270	0.003135	0.000000	0.003135	0.000000	0.000000	0.000000	0.006270	...	0.000000	0.000000	0.000000	0.000000

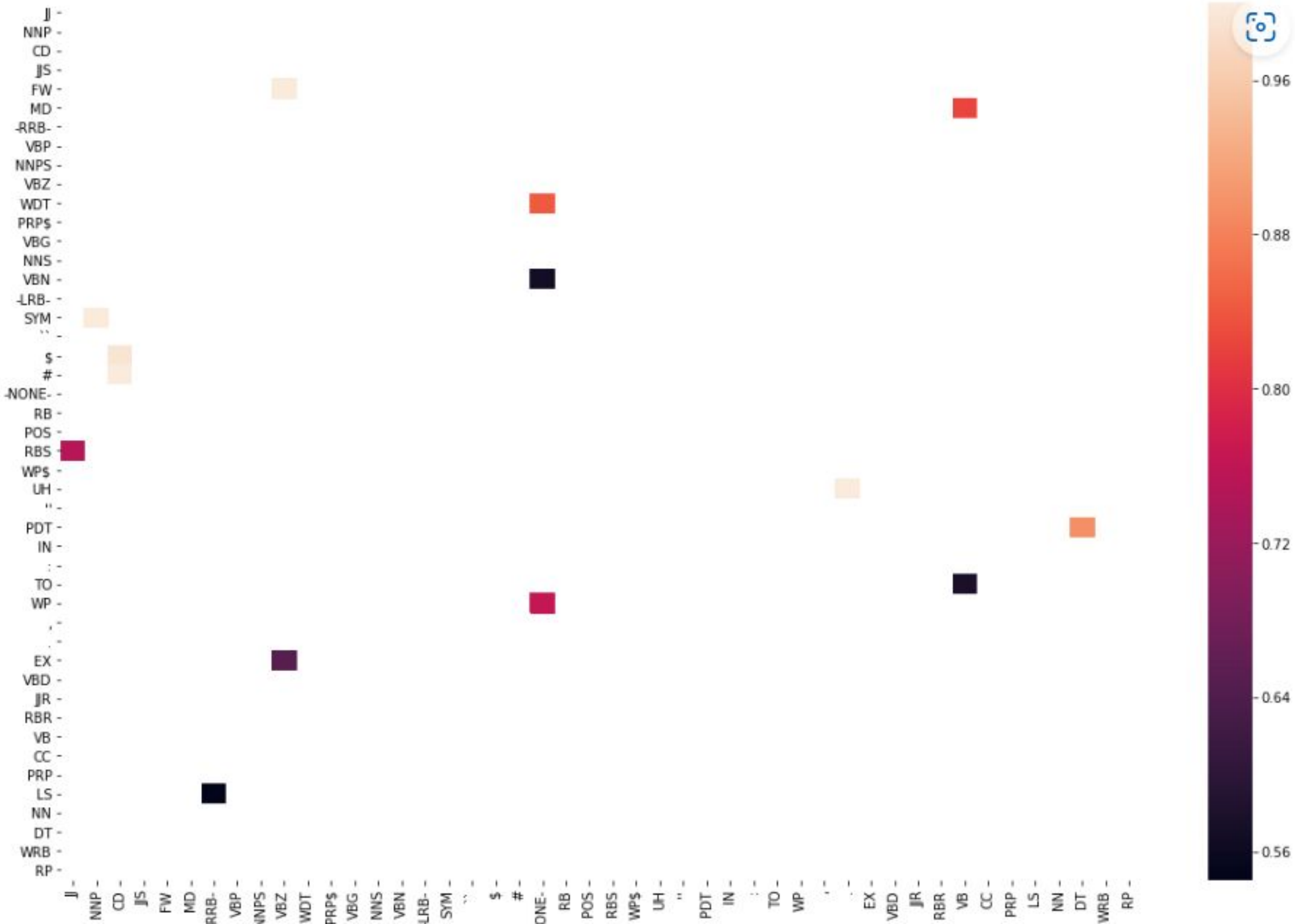
Next will create a heatmap of the tag matrix

```
# heatmap of tags matrix
# T(i, j) means P(tag j given tag i)
plt.figure(figsize=(18, 12))
sns.heatmap(tags_df)
plt.show()
```



Now, in order to see the most frequent tags we have to filter the tags with >0.5 probability

```
# frequent tags
# filter the df to get P(t2, t1) > 0.5
tags_frequent = tags_df[tags_df>0.5]
plt.figure(figsize=(18, 12))
sns.heatmap(tags_frequent)
plt.show()
```



```python
# Viterbi Heuristic
def Viterbi(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        pmax = max(p)
        # getting state for which probability is maximum
        state_max = T[p.index(pmax)]
        state.append(state_max)
    return list(zip(words, state))
```

Evaluating on Test Set

```python
# Running on entire test dataset would take more than 3-4hrs.
# Let's test our Viterbi algorithm on a few sample sentences of test dataset

random.seed(1234)

# choose random 5 sents
rndom = [random.randint(1,len(test_set)) for x in range(5)]

# list of sents
test_run = [test_set[i] for i in rndom]

# list of tagged words
test_run_base = [tup for sent in test_run for tup in sent]

# list of untagged words
test_tagged_words = [tup[0] for sent in test_run for tup in sent]
test_run
```

```
[[('Individuals', 'NNS'),
  ('familiar', 'JJ'),
  ('with', 'IN'),
  ('the', 'DT'),
  ('Justice', 'NNP'),
  ('Department', 'NNP'),
  ("'s", 'POS'),
  ('policy', 'NN'),
  ('said', 'VBD'),
  ('that', 'IN'),
  ('Justice', 'JJ'),
  ('officials', 'NNS'),
  ('had', 'VBD'),
  ("n't", 'RB'),
  ('any', 'DT'),
  ('knowledge', 'NN'),
  ('of', 'IN'),
  ('the', 'DT'),
  ('IRS', 'NNP'),
```

now, we will tag the test sentences using the Viterbi algorithm

```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi(test_tagged_words)
end = time.time()
difference = end-start
print("Time taken in seconds: ", difference)
print(tagged_seq)
```

```
Time taken in seconds:  122.65850806236267
[('Individuals', 'JJ'), ('familiar', 'JJ'), ('with', 'IN'), ('the', 'DT'), ('Justice', 'NNP'), ('Department', 'NNP'), ("'s", 'P
OS'), ('policy', 'NN'), ('said', 'VBD'), ('that', 'IN'), ('Justice', 'NNP'), ('officials', 'NNS'), ('had', 'VBD'), ("n't", 'R
B'), ('any', 'DT'), ('knowledge', 'NN'), ('of', 'IN'), ('the', 'DT'), ('IRS', 'NNP'), ("'s", 'POS'), ('actions', 'NNS'), ('in',
'IN'), ('the', 'DT'), ('last', 'JJ'), ('week', 'NN'), ('.', '.'), ('In', 'IN'), ('her', 'PRP$'), ('wake', 'NN'), ('she', 'PR
P'), ('left', 'VBD'), ('the', 'DT'), ('bitterness', 'JJ'), ('and', 'CC'), ('anger', 'JJ'), ('of', 'IN'), ('a', 'DT'), ('princip
al', 'NN'), ('who', 'WP'), ('*T*-81', 'JJ'), ('was', 'VBD'), ('her', 'PRP$'), ('friend', 'JJ'), ('and', 'CC'), ('now', 'RB'),
('calls', 'VBZ'), ('her', 'PRP$'), ('a', 'JJ'), ('betrayer', 'JJ'), (';', ':'), ('of', 'IN'), ('colleagues', 'NNS'), ('who', 'W
P'), ('*T*-82', 'JJ'), ('say', 'VBP'), ('0', '-NONE-'), ('she', 'PRP'), ('brought', 'VBD'), ('them', 'PRP'), ('shame', 'NN'),
(';', ':'), ('of', 'IN'), ('students', 'NNS'), ('and', 'CC'), ('parents', 'NNS'), ('who', 'WP'), ('*T*-83', 'JJ'), ('defended',
'JJ'), ('her', 'PRP'), ('and', 'CC'), ('insist', 'VBP'), ('0', '-NONE-'), ('she', 'PRP'), ('was', 'VBD'), ('treated', 'JJ'),
('*-1', '-NONE-'), ('harshly', 'RB'), (';', ':'), ('and', 'CC'), ('of', 'IN'), ('school-district', 'JJ'), ('officials', 'NNS'),
('stunned', 'JJ'), ('that', 'IN'), ('despite', 'IN'), ('the', 'DT'), ('bald-faced', 'JJ'), ('nature', 'NN'), ('of', 'IN'), ('he
r', 'PRP$'), ('actions', 'NNS'), (',', ','), ('she', 'PRP'), ('became', 'VBD'), ('something', 'NN'), ('of', 'IN'), ('a', 'DT'),
('local', 'JJ'), ('martyr', 'JJ'), ('.', '.'), ('At', 'IN'), ('the', 'DT'), ('same', 'JJ'), ('time', 'NN'), (',', ','), ('an',
'DT'), ('increase', 'NN'), ('of', 'IN'), ('land', 'NN'), ('under', 'IN'), ('cultivation', 'JJ'), ('after', 'IN'), ('the', 'D
T'), ('drought', 'NN'), ('has', 'VBZ'), ('boosted', 'VBN'), ('production', 'NN'), ('of', 'IN'), ('corn', 'NN'), (',', ','), ('s
oybeans', 'JJ'), ('and', 'CC'), ('other', 'JJ'), ('commodities', 'NNS'), (',', ','), ('*', '-NONE-'), ('causing', 'VBG'), ('a',
'DT'), ('fall', 'NN'), ('in', 'IN'), ('prices', 'NNS'), ('that', 'IN'), ('*T*-2', '-NONE-'), ('has', 'VBZ'), ('been', 'VBN'),
('only', 'RB'), ('partly', 'RB'), ('cushioned', 'JJ'), ('*-3', '-NONE-'), ('by', 'IN'), ('heavy', 'JJ'), ('grain', 'NN'), ('buy
ing', 'NN'), ('by', 'IN'), ('the', 'DT'), ('Soviets', 'NNPS'), ('.', '.'), ('Pacific', 'NNP'), ('Sierra', 'NNP'), (',', ','),
('based', 'VBN'), ('*', '-NONE-'), ('in', 'IN'), ('Los', 'NNP'), ('Angeles', 'NNP'), (',', ','), ('has', 'VBZ'), ('about', 'I
N'), ('200', 'CD'), ('employees', 'NNS'), ('and', 'CC'), ('supplies', 'NNS'), ('professional', 'JJ'), ('services', 'NNS'), ('an
d', 'CC'), ('advanced', 'VBD'), ('products', 'NNS'), ('to', 'TO'), ('industry', 'NN'), ('.', '.'), ('A', 'DT'), ('spokesman',
'NN'), ('declined', 'VBD'), ('*-1', '-NONE-'), ('to', 'TO'), ('speculate', 'JJ'), ('about', 'IN'), ('possible', 'JJ'), ('reduct
ions', 'NNS'), ('in', 'IN'), ('force', 'NN'), ('.', '.')]
```

As we can see it has taken around 122 seconds and it has tagged all the words in the test sentences. Now in order to check the accuracy we have

```
# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq)
print(accuracy)
```

Output

```
0.8736263736263736
```

Our POS tagger model, which is based on HMM, achieves a reasonably good accuracy of 87.36% for POS tagging.