KENTUCKY STATE UNIVERSITY, COMPUTER SCIENCE DEPARTMENT

Building and comparing Parts Of Speech tagger models using Natural Language Processing Project Status Report

Anil Varma Bethalam
May 9, 2020

1 Introduction

The process of classifying words into their parts of speech and labeling them accordingly is known as part-of-speech tagging, or simply POS-tagging. In syntactic analysis, the main aim is to understand the role played by words in the sentence, the relationship between words and to parse the grammatical structure of sentences. One of the key tasks in syntactic analysis is parsing. It means to break down a sentence into its grammatic components to help us understand the linguistic structure of sentences. POS tagging is one such task of assigning the parts of speech tag to each word. The POS tags identify the linguistic role of the word in the sentence. The First level of syntactic analysis is POS tagging. A word can be tagged a noun, verb, adjective, adverb etc. depending on its role in the sentence. Assigning the correct tag is the most fundamental task in syntactic analysis.[1] The NLTK library has a number of corpora which contains word and its POS tag. The Aim of the project is to research and build different Parts of Speech tagger models using the natural Language Tool Kit library in Python and compare the different pros of cons of each model. The different model are built using different algorithms and methods and are described as follows. The assigned POS tag helps in understanding the intended meaning of a sentence or phrase and is the crucial part of syntactic processing. There are four common approaches to POS tagging,

- Lexicon-based
- Rule-based
- Probabilistic techniques
- Deep learning techniques

1.1 Lexicon-based POS tagging

The lexicon based approach uses simple statistical algorithm which assigns POS tags that most frequently occur for that word in some training corpus. But if there's a rule that is applied to the entire text, such as, 'replace VB with NN if the previous tag is DT', or 'tag all words ending with ing as VBG', the tag can be corrected. Rule-based tagging methods use such an approach. The disadvantage of this method is that such a tagging approach cannot handle unknown or ambiguous words.

1.2 Rule-based POS tagging

Rule-based taggers first assign the tag using the lexicon method and then apply predefined rules.

1.3 Probabilistic techniques

Probabilistic taggers don't naively assign the highest frequency tag to each word, instead, they look at slightly longer parts of the sequence and often use the tag(s) and the word(s) appearing before the target word to be tagged. The objective is to find the most probable POS tag sequence. We need to find the maximum of [P(tag sequence | observation sequence)] among the possible tag sequences.

1.4 Deep learning techniques

Apart from conventional sequence models such as HMMs, significant advancement has been made in the field of Neural Networks (or deep learning) based sequence models. Specifically, Recurrent Neural Networks (RNNs) have empirically proven to outperform many conventional sequence models for tasks such as POS tagging.

2 Methodology and Code

2.1 POS Tagging Model 1 - Lexicon and Rule Based Taggers

The two most basic tagging techniques - lexicon based (or unigram) and rule-based.A Lexicon tagger uses a simple statistical tagging algorithm. For each token, it assigns the most frequently assigned POS tag. Rule-based taggers first assign the tag using the lexicon method and then apply predefined rules. We use the Treebank corpus of NLTK to build a lexicon and rule-based tagger using this corpus as the training data.[2] This following code for the first model is divided into the following subsections

- Reading and understanding the tagged dataset
- Exploratory analysis
- Lexicon and rule-based models:
- Creating and evaluating a lexicon POS tagger
- Creating and evaluating a rule-based POS tagger

2.1.1 1. Reading and understanding the tagged dataset

[1]:
```
# Importing libraries
import nltk
import numpy as np
import pandas as pd
import pprint, time
import random
from sklearn.model_selection import train_test_split
from nltk.tokenize import word_tokenize
import math
```

[2]:
```
# reading the Treebank tagged sentences
wsj = list(nltk.corpus.treebank.tagged_sents())
```

[9]:
```
# samples: Each sentence is a list of (word, pos) tuples
wsj[:3]
```

[9]:
```
[[('Pierre', 'NNP'),
  ('Vinken', 'NNP'),
  (',', ','),
  ('61', 'CD'),
  ('years', 'NNS'),
  ('old', 'JJ'),
  (',', ','),
  ('will', 'MD'),
  ('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', 'IN'),
  ('Elsevier', 'NNP'),
  ('N.V.', 'NNP'),
  (',', ','),
  ('the', 'DT'),
  ('Dutch', 'NNP'),
  ('publishing', 'VBG'),
```

```
 ('group', 'NN'),
 ('.', '.')],
[('Rudolph', 'NNP'),
 ('Agnew', 'NNP'),
 (',', ','),
 ('55', 'CD'),
 ('years', 'NNS'),
 ('old', 'JJ'),
 ('and', 'CC'),
 ('former', 'JJ'),
 ('chairman', 'NN'),
 ('of', 'IN'),
 ('Consolidated', 'NNP'),
 ('Gold', 'NNP'),
 ('Fields', 'NNP'),
 ('PLC', 'NNP'),
 (',', ','),
 ('was', 'VBD'),
 ('named', 'VBN'),
 ('*-1', '-NONE-'),
 ('a', 'DT'),
 ('nonexecutive', 'JJ'),
 ('director', 'NN'),
 ('of', 'IN'),
 ('this', 'DT'),
 ('British', 'JJ'),
 ('industrial', 'JJ'),
 ('conglomerate', 'NN'),
 ('.', '.')]]
```

In the list mentioned above, each element of the list is a sentence. Also, note that each sentence ends with a full stop '.' whose POS tag is also a '.'. Thus, the POS tag '.' demarcates the end of a sentence.

Also, we do not need the corpus to be segmented into sentences, but can rather use a list of (word, tag) tuples. Let's convert the list into a (word, tag) tuple.

[34]:
```
# converting the list of sents to a list of (word, pos tag) tuples
tagged_words = [tup for sent in wsj for tup in sent]
print(len(tagged_words))
tagged_words[:10]
```

```
100676
```

[34]:
```
[('Pierre', 'NNP'),
 ('Vinken', 'NNP'),
 (',', ','),
 ('61', 'CD'),
```

```
 ('years', 'NNS'),
 ('old', 'JJ'),
 (',', ','),
 ('will', 'MD'),
 ('join', 'VB'),
 ('the', 'DT')]
```

We now have a list of about 100676 (word, tag) tuples. Let's now do some exploratory analyses.

2.1.2 2. Exploratory Analysis

We conduct some basic exploratory analysis to understand the tagged corpus. Few simple questions to start with: 1. How many unique tags are there in the corpus? 2. Which is the most frequent tag in the corpus? 3. Which tag is most commonly assigned to the following words: - "bank" - "executive"

[48]:
```
# question 1: Find the number of unique POS tags in the corpus
# using the set() function on the list of tags to get a unique set of tags,
# and compute its length
tags = [pair[1] for pair in tagged_words]
unique_tags = set(tags)
len(unique_tags)
```

[48]:
```
46
```

[49]:
```
# question 2: Which is the most frequent tag in the corpus
# to count the frequency of elements in a list, the Counter() class from␣
 ↪collections
# module is very useful, as shown below

from collections import Counter
tag_counts = Counter(tags)
tag_counts
```

[49]:
```
Counter({'#': 16,
         '$': 724,
         "'": 694,
         ',': 4886,
         '-LRB-': 120,
         '-NONE-': 6592,
         '-RRB-': 126,
         '.': 3874,
         ':': 563,
         'CC': 2265,
         'CD': 3546,
         'DT': 8165,
         'EX': 88,
         'FW': 4,
```

```
 'IN': 9857,
 'JJ': 5834,
 'JJR': 381,
 'JJS': 182,
 'LS': 13,
 'MD': 927,
 'NN': 13166,
 'NNP': 9410,
 'NNPS': 244,
 'NNS': 6047,
 'PDT': 27,
 'POS': 824,
 'PRP': 1716,
 'PRP$': 766,
 'RB': 2822,
 'RBR': 136,
 'RBS': 35,
 'RP': 216,
 'SYM': 1,
 'TO': 2179,
 'UH': 3,
 'VB': 2554,
 'VBD': 3043,
 'VBG': 1460,
 'VBN': 2134,
 'VBP': 1321,
 'VBZ': 2125,
 'WDT': 445,
 'WP': 241,
 'WP$': 14,
 'WRB': 178,
 '``': 712})
```

[51]:
```
# the most common tags can be seen using the most_common() method of Counter
tag_counts.most_common(5)
```

[51]:
```
[('NN', 13166), ('IN', 9857), ('NNP', 9410), ('DT', 8165), ('-NONE-', 6592)]
```

Thus, NN is the most common tag followed by IN, NNP, DT, -NONE- etc.

[63]:
```
# question 3: Which tag is most commonly assigned to the word w.
bank = [pair for pair in tagged_words if pair[0].lower() == 'bank']
bank
```

[63]:
```
[('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
```

```
('Bank', 'NNP'),
('bank', 'NN'),
('Bank', 'NNP'),
('bank', 'NN'),
('Bank', 'NNP'),
('bank', 'NN'),
('bank', 'NN'),
('bank', 'NN'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('bank', 'NN'),
('bank', 'NN'),
('bank', 'NN'),
('bank', 'NN'),
('bank', 'NN'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('bank', 'NN'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('bank', 'NN'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('bank', 'NN'),
('bank', 'NN'),
('Bank', 'NNP'),
('bank', 'NN'),
('bank', 'NN'),
('bank', 'NN'),
('Bank', 'NNP'),
('bank', 'NN'),
('bank', 'NN'),
('bank', 'NN'),
('bank', 'NN'),
('bank', 'NN'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('bank', 'NN'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('Bank', 'NNP'),
('Bank', 'NNP'),
```

```
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN')]
```

[64]:
```
# question 3: Which tag is most commonly assigned to the word w.
executive = [pair for pair in tagged_words if pair[0].lower() == 'executive']
executive
```

[64]:
```
[('executive', 'NN'),
 ('executive', 'JJ'),
 ('executive', 'JJ'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'JJ'),
 ('executive', 'JJ'),
 ('executive', 'JJ'),
 ('executive', 'NN'),
 ('executive', 'JJ'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
 ('executive', 'NN'),
```

```
('executive', 'NN'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'NN'),
('executive', 'JJ'),
('executive', 'JJ'),
('executive', 'NN'),
('executive', 'JJ')]
```

2.1.3 2. Exploratory Analysis Contd.

Until now, we were looking at the frequency of tags assigned to particular words, which is the basic idea used by lexicon or unigram taggers. Let's now try observing some rules which can potentially be used for POS tagging.

Let's see if the following questions reveal something useful:

4. What fraction of words with the tag 'VBD' (verb, past tense) end with the letters 'ed'
5. What fraction of words with the tag 'VBG' (verb, present participle/gerund) end with the letters 'ing'

[105]:
```
# 4. how many words with the tag 'VBD' (verb, past tense) end with 'ed'
past_tense_verbs = [pair for pair in tagged_words if pair[1]=='VBD']
ed_verbs = [pair for pair in past_tense_verbs if pair[0].endswith('ed')]
print(len(ed_verbs) / len(past_tense_verbs))
ed_verbs[:20]
```

```
0.3881038448899113
```

[105]:
```
[('reported', 'VBD'),
 ('stopped', 'VBD'),
 ('studied', 'VBD'),
 ('led', 'VBD'),
 ('worked', 'VBD'),
 ('explained', 'VBD'),
 ('imposed', 'VBD'),
 ('dumped', 'VBD'),
 ('poured', 'VBD'),
 ('mixed', 'VBD'),
 ('described', 'VBD'),
 ('ventilated', 'VBD'),
 ('contracted', 'VBD'),
 ('continued', 'VBD'),
 ('eased', 'VBD'),
 ('ended', 'VBD'),
 ('lengthened', 'VBD'),
 ('reached', 'VBD'),
 ('resigned', 'VBD'),
 ('approved', 'VBD')]
```

[104]:
```
# 5. how many words with the tag 'VBG' end with 'ing'
participle_verbs = [pair for pair in tagged_words if pair[1]=='VBG']
ing_verbs = [pair for pair in participle_verbs if pair[0].endswith('ing')]
print(len(ing_verbs) / len(participle_verbs))
ing_verbs[:20]
```

```
0.9972602739726028
```

[104]:
```
[('publishing', 'VBG'),
 ('causing', 'VBG'),
 ('using', 'VBG'),
 ('talking', 'VBG'),
 ('having', 'VBG'),
 ('making', 'VBG'),
 ('surviving', 'VBG'),
 ('including', 'VBG'),
 ('including', 'VBG'),
 ('according', 'VBG'),
 ('remaining', 'VBG'),
 ('according', 'VBG'),
 ('declining', 'VBG'),
 ('rising', 'VBG'),
 ('yielding', 'VBG'),
 ('waiving', 'VBG'),
 ('holding', 'VBG'),
 ('holding', 'VBG'),
 ('cutting', 'VBG'),
 ('manufacturing', 'VBG')]
```

Now we try to check the tag patterns using the fact the some tags are more likely to apper after certain other tags. For e.g. most nouns NN are usually followed by determiners DT ("The/DT constitution/NN"), adjectives JJ usually precede a noun NN (" A large/JJ building/NN"), etc.

1. What fraction of adjectives JJ are followed by a noun NN?
2. What fraction of determiners DT are followed by a noun NN?
3. What fraction of modals MD are followed by a verb VB?

[84]:
```
# question: what fraction of adjectives JJ are followed by a noun NN

# create a list of all tags (without the words)
tags = [pair[1] for pair in tagged_words]

# create a list of JJ tags
jj_tags = [t for t in tags if t == 'JJ']

# create a list of (JJ, NN) tags
jj_nn_tags = [(t, tags[index+1]) for index, t in enumerate(tags)
              if t=='JJ' and tags[index+1]=='NN']

print(len(jj_tags))
print(len(jj_nn_tags))
print(len(jj_nn_tags) / len(jj_tags))
```

```
5834
2611
0.4475488515598217
```

[97]:
```
# question: what fraction of determiners DT are followed by a noun NN
dt_tags = [t for t in tags if t == 'DT']
dt_nn_tags = [(t, tags[index+1]) for index, t in enumerate(tags)
              if t=='DT' and tags[index+1]=='NN']

print(len(dt_tags))
print(len(dt_nn_tags))
print(len(dt_nn_tags) / len(dt_tags))
```

```
8165
3844
0.470789957134109
```

[100]:
```
# question: what fraction of modals MD are followed by a verb VB?
md_tags = [t for t in tags if t == 'MD']
md_vb_tags = [(t, tags[index+1]) for index, t in enumerate(tags)
              if t=='MD' and tags[index+1]=='VB']

print(len(md_tags))
print(len(md_vb_tags))
print(len(md_vb_tags) / len(md_tags))
```

```
927
756
0.8155339805825242
```

Thus, we see that the probability of certain tags appearing after certain other tags is quite high, and this fact can be used to build quite efficient POS tagging algorithms. Let us now see lexicon and rule-based models for POS tagging. We first split the corpus into training and test sets and then use built-in NLTK taggers.

2.1.4 3.1 Splitting into Train and Test Sets

[85]:
```
# 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[:2])
```

```
2739
1175
[[('Negotiable', 'JJ'), (',', ','), ('bank-backed', 'JJ'), ('business', 'NN'),
('credit', 'NN'), ('instruments', 'NNS'), ('typically', 'RB'), ('financing',
'VBG'), ('an', 'DT'), ('import', 'NN'), ('order', 'NN'), ('.', '.')], [('Where',
'WRB'), ('they', 'PRP'), ('disagree', 'VBP'), ('*T*-1', '-NONE-'), ('is',
'VBZ'), ('on', 'IN'), ('the', 'DT'), ('subject', 'NN'), ('of', 'IN'), ('U.S.',
```

```
'NNP'), ('direct', 'JJ'), ('investment', 'NN'), ('in', 'IN'), ('Japan', 'NNP'),
('.', '.')]]
```

2.1.5 3.2 Lexicon (Unigram) Tagger

Let's now try training a lexicon (or a unigram) tagger which assigns the most commonly assigned tag to a word.

In NLTK, the `UnigramTagger()` can be used to train such a model.

[120]:
```
# Lexicon (or unigram tagger)
unigram_tagger = nltk.UnigramTagger(train_set)
unigram_tagger.evaluate(test_set)
```

[120]:
```
0.870780747954553
```

Even a simple unigram tagger seems to perform fairly well.

2.1.6 3.3. Rule-Based (Regular Expression) Tagger

Now we build a rule-based, or regular expression based tagger. In NLTK, the `RegexpTagger()` can be provided with handwritten regular expression patterns, as shown below.

In the example below, we specify regexes for gerunds and past tense verbs (as seen above), 3rd singular present verb (creates, moves, makes etc.), modal verbs MD (should, would, could), possesive nouns (partner's, bank's etc.), plural nouns (banks, institutions), cardinal numbers CD and finally, if none of the above rules are applicable to a word, we tag the most frequent tag NN.

[112]:
```
# specify patterns for tagging
# example from the NLTK book
patterns = [
    (r'.*ing$', 'VBG'),               # gerund
    (r'.*ed$', 'VBD'),                # past tense
    (r'.*es$', 'VBZ'),                # 3rd singular present
    (r'.*ould$', 'MD'),               # modals
    (r'.*\'s$', 'NN$'),               # possessive nouns
    (r'.*s$', 'NNS'),                 # plural nouns
    (r'^-?[0-9]+(.[0-9]+)?$', 'CD'), # cardinal numbers
    (r'.*', 'NN')                     # nouns
]
```

[113]:
```
regexp_tagger = nltk.RegexpTagger(patterns)
# help(regexp_tagger)
```

[116]:
```
regexp_tagger.evaluate(test_set)
```

[116]:
```
0.21931829474311834
```

2.1.7 3.4 Combining Taggers

Let's now try combining the taggers created above. We saw that the rule-based tagger by itself is quite ineffective since we've only written a handful of rules. However, if we could combine the lexicon and the rule-based tagger, we can potentially create a tagger much better than any of the individual ones.

NLTK provides a convenient way to combine taggers using the 'backup' argument. In the following code, we create a regex tagger which is used as a backup tagger to the lexicon tagger, i.e. when the tagger is not able to tag using the lexicon (in case of a new word not in the vocabulary), it uses the rule-based tagger.[3]

Also, note that the rule-based tagger itself is backed up by the tag 'NN'.

[117]:
```
# rule based tagger
rule_based_tagger = nltk.RegexpTagger(patterns)

# lexicon backed up by the rule-based tagger
lexicon_tagger = nltk.UnigramTagger(train_set, backoff=rule_based_tagger)

lexicon_tagger.evaluate(test_set)
```

[117]:
```
0.9049985093908377
```

2.2 POS tagging using modified Viterbi

Markov processes are commonly used to model sequential data, such as text and speech. For e.g., say you want to build an application which predicts the next word in a sentence. You can represent each word in a sentence as a state. The transition probabilities (which can be learnt from some corpus, more on that later) would represent the probability that the process moves from the current word to the next word. Transition probability is the probabilities of transitioning from one state to another.

The Hidden Markov Model (HMM) is an extension to the Markov process which is used to model phenomena where the states are hidden (or latent) and they emit observations. For example, in a speech recognition system (a speech-to-text converter), the states represent the actual text words which you want to predict, but you do not directly observe them (i.e. the states are hidden). Similarly, in POS tagging, what you observe are the words in a sentence, while the POS tags themselves are hidden. Thus, you can model the POS tagging task as an HMM with the hidden states representing POS tags which emit observations, i.e. words.

A Markov chain is used to represent a process which performs a transition from one state to other. This transition makes an assumption that the probability of transitioning to the next state is dependent solely on the current state. The start state is a special state which represents the initial state of the process (e.g. the start of a sentence). The hidden states emit observations with a certain probability. Therefore, along with the transition and initial state probabilities, Hidden Markov Models also have emission probabilities which represent the probability that an observation is emitted by a particular state. The sequence of words are the observations while the POS tags are the hidden states. Also, the HMM is represented by its initial state probabilities (i.e. the probability of transition into any state from the initial state), the transition and the emission probabilities.[4]

The Hidden Markov Model is used to model phenomena where the states are hidden and they emit observations. The transition and the emission probabilities specify the probabilities of transition between states and emission of observations from states, respectively. In POS tagging, the states are the POS tags while the words are the observations. after learning the HMM model parameters, i.e. computing the transition and the emission probabilities, the Viterbi algorithm is used to assign POS tags efficiently. The Viterbi algorithm, although greedy in nature, is a computationally viable alternative to choosing a tag sequence from a large set of possible sequences.

The Penn Treebank is a manual of around a million words taken from 1989 Wall Street Journal's articles. This manual is available in NLTK toolkit of Python. You can explore this data by importing NLTK and then run the following code:

```
wsj = list(nltk.corpus.treebank.tagged_sents())
```

Below are the steps that will be used for building the model.

- Conduct exploratory analysis on the tagged corpus
- Sample the data into 70:30 train-test sets
- Train an HMM model using the tagged corpus:
  - Calculating the emission probabilities
  - Calculating the transition probabilities
- Write the Viterbi algorithm to POS tag sequence of words (sentence)
- Evaluate the model predictions against the ground truth

2.2.1 Data Preparation

[308]:
```
#Importing libraries
import nltk
import numpy as np
import pandas as pd
import random
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
import nltk.tag, nltk.data
from nltk.tag import CRFTagger
```

[309]:
```
# reading the Treebank tagged sentences
nltk_data = list(nltk.corpus.treebank.tagged_sents(tagset='universal'))
```

[310]:
```
# first few tagged sentences
print(nltk_data[:5])
```

```
[[('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years',
'NOUN'), ('old', 'ADJ'), (',', '.'), ('will', 'VERB'), ('join', 'VERB'), ('the',
'DET'), ('board', 'NOUN'), ('as', 'ADP'), ('a', 'DET'), ('nonexecutive', 'ADJ'),
('director', 'NOUN'), ('Nov.', 'NOUN'), ('29', 'NUM'), ('.', '.')], [('Mr.',
'NOUN'), ('Vinken', 'NOUN'), ('is', 'VERB'), ('chairman', 'NOUN'), ('of',
'ADP'), ('Elsevier', 'NOUN'), ('N.V.', 'NOUN'), (',', '.'), ('the', 'DET'),
('Dutch', 'NOUN'), ('publishing', 'VERB'), ('group', 'NOUN'), ('.', '.')],
[('Rudolph', 'NOUN'), ('Agnew', 'NOUN'), (',', '.'), ('55', 'NUM'), ('years',
'NOUN'), ('old', 'ADJ'), ('and', 'CONJ'), ('former', 'ADJ'), ('chairman',
'NOUN'), ('of', 'ADP'), ('Consolidated', 'NOUN'), ('Gold', 'NOUN'), ('Fields',
'NOUN'), ('PLC', 'NOUN'), (',', '.'), ('was', 'VERB'), ('named', 'VERB'),
('*-1', 'X'), ('a', 'DET'), ('nonexecutive', 'ADJ'), ('director', 'NOUN'),
('of', 'ADP'), ('this', 'DET'), ('British', 'ADJ'), ('industrial', 'ADJ'),
('conglomerate', 'NOUN'), ('.', '.')], [('A', 'DET'), ('form', 'NOUN'), ('of',
'ADP'), ('asbestos', 'NOUN'), ('once', 'ADV'), ('used', 'VERB'), ('*', 'X'),
('*', 'X'), ('to', 'PRT'), ('make', 'VERB'), ('Kent', 'NOUN'), ('cigarette',
'NOUN'), ('filters', 'NOUN'), ('has', 'VERB'), ('caused', 'VERB'), ('a', 'DET'),
('high', 'ADJ'), ('percentage', 'NOUN'), ('of', 'ADP'), ('cancer', 'NOUN'),
('deaths', 'NOUN'), ('among', 'ADP'), ('a', 'DET'), ('group', 'NOUN'), ('of',
'ADP'), ('workers', 'NOUN'), ('exposed', 'VERB'), ('*', 'X'), ('to', 'PRT'),
('it', 'PRON'), ('more', 'ADV'), ('than', 'ADP'), ('30', 'NUM'), ('years',
'NOUN'), ('ago', 'ADP'), (',', '.'), ('researchers', 'NOUN'), ('reported',
'VERB'), ('0', 'X'), ('*T*-1', 'X'), ('.', '.')], [('The', 'DET'), ('asbestos',
'NOUN'), ('fiber', 'NOUN'), (',', '.'), ('crocidolite', 'NOUN'), (',', '.'),
('is', 'VERB'), ('unusually', 'ADV'), ('resilient', 'ADJ'), ('once', 'ADP'),
('it', 'PRON'), ('enters', 'VERB'), ('the', 'DET'), ('lungs', 'NOUN'), (',',
'.'), ('with', 'ADP'), ('even', 'ADV'), ('brief', 'ADJ'), ('exposures', 'NOUN'),
('to', 'PRT'), ('it', 'PRON'), ('causing', 'VERB'), ('symptoms', 'NOUN'),
('that', 'DET'), ('*T*-1', 'X'), ('show', 'VERB'), ('up', 'PRT'), ('decades',
'NOUN'), ('later', 'ADJ'), (',', '.'), ('researchers', 'NOUN'), ('said',
'VERB'), ('0', 'X'), ('*T*-2', 'X'), ('.', '.')]]
```

2.2.2 Exploratary data analysis on the universal data set

[311]:
```
# Splitting the data into train and test
random.seed(1234)
train_set, test_set = train_test_split(nltk_data,train_size=0.95,test_size=0.
 ↪05,random_state=101)

print(len(train_set))
print(len(test_set))
print(train_set[:5])
```

```
3718
196
[[('Reliance', 'NOUN'), ('confirmed', 'VERB'), ('the', 'DET'), ('filing',
'NOUN'), ('but', 'CONJ'), ('would', 'VERB'), ("n't", 'ADV'), ('elaborate',
```

```
'VERB'), ('.', '.')], [('*', 'X'), ('Encouraging', 'VERB'), ('long-term',
'ADJ'), ('investing', 'NOUN'), ('.', '.')], [('Because', 'ADP'), ('of', 'ADP'),
('the', 'DET'), ('rulings', 'NOUN'), (',', '.'), ('the', 'DET'), ('Commerce',
'NOUN'), ('Department', 'NOUN'), ('will', 'VERB'), ('continue', 'VERB'), ('*-1',
'X'), ('to', 'PRT'), ('investigate', 'VERB'), ('complaints', 'NOUN'),
('*ICH*-2', 'X'), ('by', 'ADP'), ('U.S.', 'NOUN'), ('sweater', 'NOUN'),
('makers', 'NOUN'), ('that', 'ADP'), ('the', 'DET'), ('imports', 'NOUN'),
('are', 'VERB'), ('reaching', 'VERB'), ('the', 'DET'), ('U.S.', 'NOUN'), ('at',
'ADP'), ('unfairly', 'ADV'), ('low', 'ADJ'), ('prices', 'NOUN'), ('in', 'ADP'),
('violation', 'NOUN'), ('of', 'ADP'), ('the', 'DET'), ('U.S.', 'NOUN'), ('anti-
dumping', 'ADJ'), ('act', 'NOUN'), ('.', '.')], [('What', 'PRON'), ('she',
'PRON'), ('did', 'VERB'), ('*T*-97', 'X'), ('was', 'VERB'), ('like', 'ADP'),
('*', 'X'), ('taking', 'VERB'), ('the', 'DET'), ('law', 'NOUN'), ('into',
'ADP'), ('your', 'PRON'), ('own', 'ADJ'), ('hands', 'NOUN'), ('.', '.'), ("''",
'.')], [('Los', 'NOUN'), ('Angeles', 'NOUN'), ('is', 'VERB'), ('a', 'DET'),
('sprawling', 'ADJ'), (',', '.'), ('balkanized', 'ADJ'), ('newspaper', 'NOUN'),
('market', 'NOUN'), (',', '.'), ('and', 'CONJ'), ('advertisers', 'NOUN'),
('seemed', 'VERB'), ('*-1', 'X'), ('to', 'PRT'), ('feel', 'VERB'), ('0', 'X'),
('they', 'PRON'), ('could', 'VERB'), ('buy', 'VERB'), ('space', 'NOUN'), ('in',
'ADP'), ('the', 'DET'), ('mammoth', 'ADJ'), ('Times', 'NOUN'), (',', '.'),
('then', 'ADV'), ('target', 'VERB'), ('a', 'DET'), ('particular', 'ADJ'),
('area', 'NOUN'), ('with', 'ADP'), ('one', 'NUM'), ('of', 'ADP'), ('the',
'DET'), ('regional', 'ADJ'), ('dailies', 'NOUN'), ('.', '.')]]
```

[312]:
```
# Getting the list of tagged words
train_tagged_words = [tup for sent in train_set for tup in sent]
len(train_tagged_words)
```

[312]:
```
95547
```

[313]:
```
# tokens
tokens = [pair[0] for pair in train_tagged_words]
tokens[:10]
```

[313]:
```
['Reliance',
 'confirmed',
 'the',
 'filing',
 'but',
 'would',
 "n't",
 'elaborate',
 '.',
 '*']
```

[314]:
```
# vocabulary
V = set(tokens)
```

```
print(len(V))
```

```
12100
```

[315]:
```
# number of tags
T = set([pair[1] for pair in train_tagged_words])
len(T)
```

[315]:
```
12
```

[316]:
```
print(T)

#tags = [pair[1] for pair in train_tagged_words]
#print(tags)
#unique_tags = set(tags)
#len(unique_tags)

#from collections import Counter
#tag_counts = Counter(tags)
#tag_counts
```

```
{'.', 'NOUN', 'VERB', 'ADV', 'PRON', 'NUM', 'ADJ', 'ADP', 'PRT', 'CONJ', 'DET',
'X'}
```

2.2.3 Calculating the Emission Probabilities

The process of learning the probabilities from a tagged corpus is called training an HMM model. For this, we calculate the transition probability (from one state to another), the emission probability (of a state emitting a word), and the initial state probabilities (of a state appearing at the start of a sentence).

Emission Probability is the probability that a tag 't' will 'emit' a word w.Emission Probability of a word 'w' for tag 't':

P(w | t) = Number of times w has been tagged t/Number of times t appears

2.2.4 computing P(w/t) and storing in T x V matrix

[317]:
```
t = len(T)
v = len(V)
w_given_t = np.zeros((t, v))
w_given_t
```

[317]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
```

```
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]])
```

[318]:
```
# 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)
```

2.2.5 Calculating the Transition probabilities

Transition probability is the probability of moving from one state to another. Transition Probability of tag t1 followed by tag t2 is given as

P(t2 | t1) = Number of times t1 is followed by tag t2/ Number of times t1 appears

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

[319]:
```
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)
```

[320]:
```
# 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
```

[320]:
```
array([[9.33201462e-02, 2.22242206e-01, 8.90946686e-02, 5.23240119e-02,
        6.63490072e-02, 8.10033232e-02, 4.39629592e-02, 9.13422629e-02,
        2.42740265e-03, 5.75384349e-02, 1.73334539e-01, 2.69711409e-02],
       [2.40603983e-01, 2.63563901e-01, 1.47667453e-01, 1.70737058e-02,
        4.60661016e-03, 9.54226404e-03, 1.22477328e-02, 1.76513597e-01,
        4.33971919e-02, 4.26659845e-02, 1.29423812e-02, 2.91751977e-02],
       [3.49341594e-02, 1.10069714e-01, 1.69248641e-01, 8.19519758e-02,
```

```
        3.57862115e-02, 2.28505041e-02, 6.49883822e-02, 9.20216888e-02,
        3.06738969e-02, 5.57707204e-03, 1.34391949e-01, 2.17505813e-01],
       [1.37131497e-01, 3.14673744e-02, 3.43491226e-01, 8.04902315e-02,
        1.49055980e-02, 3.04736663e-02, 1.29181847e-01, 1.18582316e-01,
        1.42431268e-02, 6.95594586e-03, 6.98906928e-02, 2.31864862e-02],
       [4.09647785e-02, 2.10949466e-01, 4.85451758e-01, 3.40735056e-02,
        7.65696773e-03, 6.50842255e-03, 7.31240436e-02, 2.29709037e-02,
        1.30168451e-02, 5.35987737e-03, 9.95405857e-03, 8.99693742e-02],
       [1.17331743e-01, 3.50208461e-01, 1.87611673e-02, 2.97796307e-03,
        1.48898154e-03, 1.84931502e-01, 3.42465751e-02, 3.60333547e-02,
        2.65038721e-02, 1.36986300e-02, 3.27575929e-03, 2.10541993e-01],
       [6.39314577e-02, 6.99621022e-01, 1.16987973e-02, 4.77838190e-03,
        3.29543574e-04, 2.12555602e-02, 6.64030313e-02, 7.82665983e-02,
        1.07101668e-02, 1.69714950e-02, 4.94315382e-03, 2.10907888e-02],
       [3.90249118e-02, 3.20966542e-01, 8.33956990e-03, 1.40062012e-02,
        7.00310096e-02, 6.22260235e-02, 1.07024483e-01, 1.68929752e-02,
        1.38992839e-03, 9.62258084e-04, 3.24708641e-01, 3.44274566e-02],
       [4.38220762e-02, 2.47775942e-01, 4.05271828e-01, 1.02141676e-02,
        1.77924223e-02, 5.66721596e-02, 8.30313042e-02, 2.00988464e-02,
        1.64744642e-03, 2.30642501e-03, 9.78583172e-02, 1.35090612e-02],
       [3.48675027e-02, 3.49139929e-01, 1.56671315e-01, 5.53231053e-02,
        5.81125058e-02, 3.99814025e-02, 1.18084610e-01, 5.25337048e-02,
        4.64900024e-03, 4.64900048e-04, 1.21338911e-01, 8.83310102e-03],
       [1.79929957e-02, 6.38087213e-01, 3.98502611e-02, 1.24381110e-02,
        3.74350930e-03, 2.22195387e-02, 2.04323143e-01, 9.53991059e-03,
        2.41516726e-04, 4.83033451e-04, 5.67564322e-03, 4.54051457e-02],
       [1.63590074e-01, 6.23806491e-02, 2.03851044e-01, 2.49840859e-02,
        5.55378757e-02, 2.86441762e-03, 1.71865057e-02, 1.42584339e-01,
        1.85232341e-01, 1.06619988e-02, 5.47422022e-02, 7.63844699e-02]],
      dtype=float32)
```

[321]:
```
# convert the matrix to a df for better readability
tags_df = pd.DataFrame(tags_matrix, columns = list(T), index=list(T))
tags_df
```

[321]:

	.	NOUN	VERB	ADV	PRON	NUM	ADJ	\
.	0.093320	0.222242	0.089095	0.052324	0.066349	0.081003	0.043963	
NOUN	0.240604	0.263564	0.147667	0.017074	0.004607	0.009542	0.012248	
VERB	0.034934	0.110070	0.169249	0.081952	0.035786	0.022851	0.064988	
ADV	0.137131	0.031467	0.343491	0.080490	0.014906	0.030474	0.129182	
PRON	0.040965	0.210949	0.485452	0.034074	0.007657	0.006508	0.073124	
NUM	0.117332	0.350208	0.018761	0.002978	0.001489	0.184932	0.034247	
ADJ	0.063931	0.699621	0.011699	0.004778	0.000330	0.021256	0.066403	
ADP	0.039025	0.320967	0.008340	0.014006	0.070031	0.062226	0.107024	
PRT	0.043822	0.247776	0.405272	0.010214	0.017792	0.056672	0.083031	
CONJ	0.034868	0.349140	0.156671	0.055323	0.058113	0.039981	0.118085	
DET	0.017993	0.638087	0.039850	0.012438	0.003744	0.022220	0.204323	

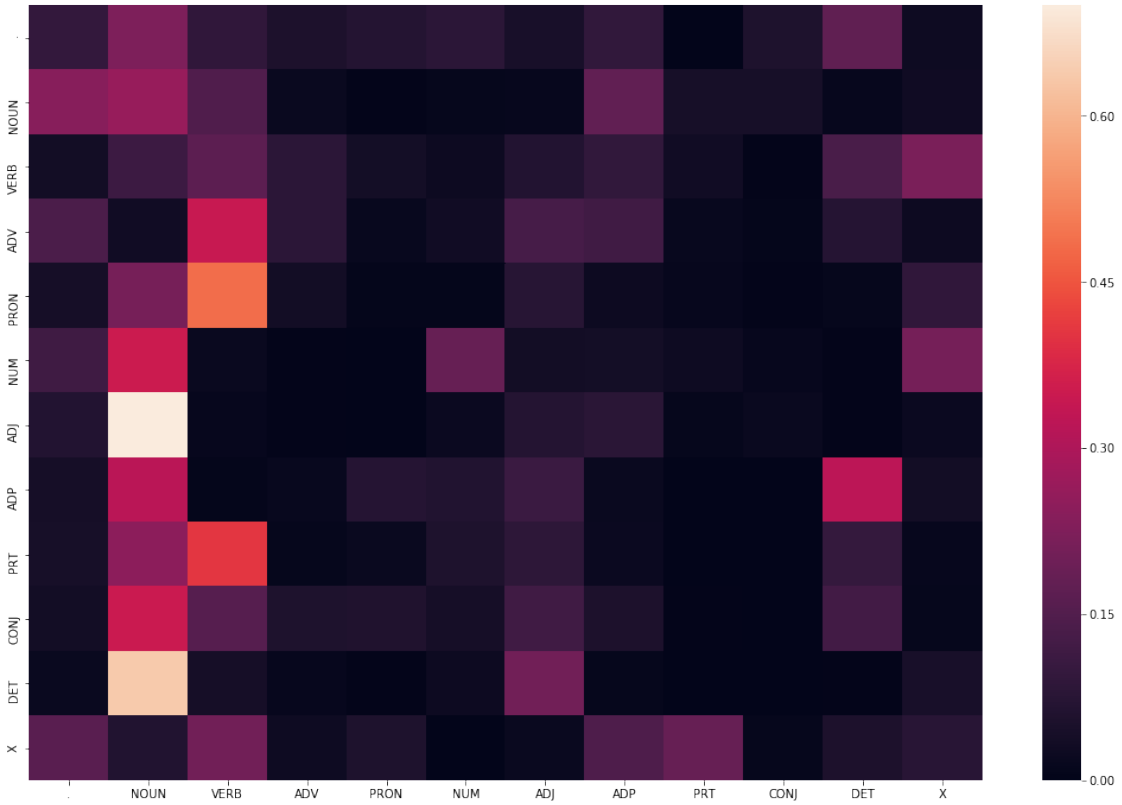
```
X     0.163590  0.062381  0.203851  0.024984  0.055538  0.002864  0.017187

            ADP       PRT       CONJ       DET         X
.      0.091342  0.002427  0.057538  0.173335  0.026971
NOUN   0.176514  0.043397  0.042666  0.012942  0.029175
VERB   0.092022  0.030674  0.005577  0.134392  0.217506
ADV    0.118582  0.014243  0.006956  0.069891  0.023186
PRON   0.022971  0.013017  0.005360  0.009954  0.089969
NUM    0.036033  0.026504  0.013699  0.003276  0.210542
ADJ    0.078267  0.010710  0.016971  0.004943  0.021091
ADP    0.016893  0.001390  0.000962  0.324709  0.034427
PRT    0.020099  0.001647  0.002306  0.097858  0.013509
CONJ   0.052534  0.004649  0.000465  0.121339  0.008833
DET    0.009540  0.000242  0.000483  0.005676  0.045405
X      0.142584  0.185232  0.010662  0.054742  0.076384
```

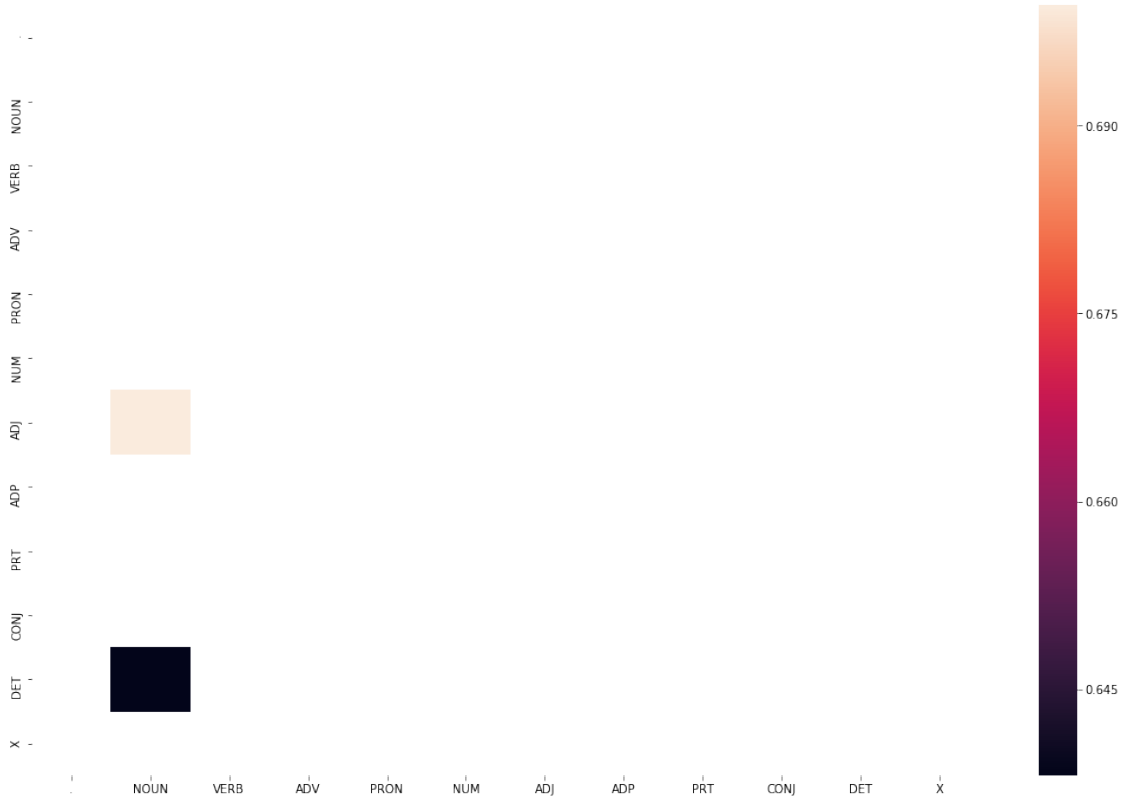
[322]:
```
tags_df.loc['.', :]
```

[322]:
```
.       0.093320
NOUN    0.222242
VERB    0.089095
ADV     0.052324
PRON    0.066349
NUM     0.081003
ADJ     0.043963
ADP     0.091342
PRT     0.002427
CONJ    0.057538
DET     0.173335
X       0.026971
Name: ., dtype: float32
```

[323]:
```
# 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()
```



[324]:
```
# 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()
```



[ ]:

2.2.7 Build the vanilla Viterbi based POS tagger

[325]:
```
len(train_tagged_words)
```

[325]: 95547

[326]:
```
# 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))
```

2.2.8 Evaluating on Test Set using Vanilla Viterbi

[327]:
```
# 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
```

[327]:
```
[[('The', 'DET'),
  ('Contra', 'NOUN'),
  ('military', 'ADJ'),
  ('command', 'NOUN'),
  (',', '.'),
  ('in', 'ADP'),
  ('a', 'DET'),
  ('statement', 'NOUN'),
  ('from', 'ADP'),
  ('Honduras', 'NOUN'),
  (',', '.'),
  ('said', 'VERB'),
  ('0', 'X'),
```

```
 ('Sandinista', 'NOUN'),
 ('troops', 'NOUN'),
 ('had', 'VERB'),
 ('launched', 'VERB'),
 ('a', 'DET'),
 ('major', 'ADJ'),
 ('offensive', 'NOUN'),
 ('against', 'ADP'),
 ('the', 'DET'),
 ('rebel', 'NOUN'),
 ('forces', 'NOUN'),
 ('.', '.')],
[('*-1', 'X'),
 ('Bucking', 'VERB'),
 ('the', 'DET'),
 ('market', 'NOUN'),
 ('trend', 'NOUN'),
 (',', '.'),
 ('an', 'DET'),
 ('issue', 'NOUN'),
 ('of', 'ADP'),
 ('$', '.'),
 ('130', 'NUM'),
 ('million', 'NUM'),
 ('*U*', 'X'),
 ('general', 'ADJ'),
 ('obligation', 'NOUN'),
 ('distributable', 'ADJ'),
 ('state', 'NOUN'),
 ('aid', 'NOUN'),
 ('bonds', 'NOUN'),
 ('from', 'ADP'),
 ('Detroit', 'NOUN'),
 (',', '.'),
 ('Mich.', 'NOUN'),
 (',', '.'),
 ('apparently', 'ADV'),
 ('drew', 'VERB'),
 ('solid', 'ADJ'),
 ('investor', 'NOUN'),
 ('interest', 'NOUN'),
 ('.', '.')],
[('Ralston', 'NOUN'),
 ('said', 'VERB'),
 ('0', 'X'),
 ('its', 'PRON'),
 ('Eveready', 'NOUN'),
```

```
 ('battery', 'NOUN'),
 ('unit', 'NOUN'),
 ('was', 'VERB'),
 ('hurt', 'VERB'),
 ('*-1', 'X'),
 ('by', 'ADP'),
 ('continuing', 'VERB'),
 ('economic', 'ADJ'),
 ('problems', 'NOUN'),
 ('in', 'ADP'),
 ('South', 'NOUN'),
 ('America', 'NOUN'),
 ('.', '.')],
[('I', 'PRON'),
 ('feel', 'VERB'),
 ('pretty', 'ADV'),
 ('good', 'ADJ'),
 ('about', 'ADP'),
 ('it', 'PRON'),
 ('.', '.')],
[('Mr.', 'NOUN'),
 ('Felten', 'NOUN'),
 ('said', 'VERB'),
 (',', '.'),
 ('``', '.'),
 ('We', 'PRON'),
 ('got', 'VERB'),
 ('what', 'PRON'),
 ('*T*-252', 'X'),
 ('amounted', 'VERB'),
 ('to', 'PRT'),
 ('a', 'DET'),
 ('parking', 'NOUN'),
 ('ticket', 'NOUN'),
 (',', '.'),
 ('and', 'CONJ'),
 ('by', 'ADP'),
 ('*-1', 'X'),
 ('complaining', 'VERB'),
 ('about', 'ADP'),
 ('it', 'PRON'),
 (',', '.'),
 ('we', 'PRON'),
 ('ended', 'VERB'),
 ('up', 'PRT'),
 ('with', 'ADP'),
 ('a', 'DET'),
```

```
  ('sizable', 'ADJ'),
  ('fine', 'NOUN'),
  ('and', 'CONJ'),
  ('suspension', 'NOUN'),
  ('.', '.'),
  ("''", '.')]]
```

[328]:
```
# tagging the test sentences
start = time.time()
tagged_seq = Viterbi(test_tagged_words)
end = time.time()
difference = end-start
```

[329]:
```
print("Time taken in seconds: ", difference)
print(tagged_seq)
#print(test_run_base)
```

```
Time taken in seconds:  28.097516536712646
[('The', 'DET'), ('Contra', '.'), ('military', 'ADJ'), ('command', 'VERB'),
(',', '.'), ('in', 'ADP'), ('a', 'DET'), ('statement', 'NOUN'), ('from', 'ADP'),
('Honduras', '.'), (',', '.'), ('said', 'VERB'), ('0', 'X'), ('Sandinista',
'.'), ('troops', 'NOUN'), ('had', 'VERB'), ('launched', 'VERB'), ('a', 'DET'),
('major', 'ADJ'), ('offensive', '.'), ('against', 'ADP'), ('the', 'DET'),
('rebel', '.'), ('forces', 'NOUN'), ('.', '.'), ('*-1', 'X'), ('Bucking', '.'),
('the', 'DET'), ('market', 'NOUN'), ('trend', 'NOUN'), (',', '.'), ('an',
'DET'), ('issue', 'NOUN'), ('of', 'ADP'), ('$', '.'), ('130', 'NUM'),
('million', 'NUM'), ('*U*', 'X'), ('general', 'ADJ'), ('obligation', 'NOUN'),
('distributable', 'ADJ'), ('state', 'NOUN'), ('aid', 'NOUN'), ('bonds', 'NOUN'),
('from', 'ADP'), ('Detroit', 'NOUN'), (',', '.'), ('Mich.', 'NOUN'), (',', '.'),
('apparently', 'ADV'), ('drew', '.'), ('solid', 'ADJ'), ('investor', 'NOUN'),
('interest', 'NOUN'), ('.', '.'), ('Ralston', 'NOUN'), ('said', 'VERB'), ('0',
'X'), ('its', 'PRON'), ('Eveready', '.'), ('battery', 'NOUN'), ('unit', 'NOUN'),
('was', 'VERB'), ('hurt', 'VERB'), ('*-1', 'X'), ('by', 'ADP'), ('continuing',
'VERB'), ('economic', 'ADJ'), ('problems', 'NOUN'), ('in', 'ADP'), ('South',
'NOUN'), ('America', 'NOUN'), ('.', '.'), ('I', 'PRON'), ('feel', 'VERB'),
('pretty', 'ADV'), ('good', 'ADJ'), ('about', 'ADP'), ('it', 'PRON'), ('.',
'.'), ('Mr.', 'NOUN'), ('Felten', 'NOUN'), ('said', 'VERB'), (',', '.'), ('``',
'.'), ('We', 'PRON'), ('got', 'VERB'), ('what', 'PRON'), ('*T*-252', '.'),
('amounted', 'VERB'), ('to', 'PRT'), ('a', 'DET'), ('parking', 'NOUN'),
('ticket', 'NOUN'), (',', '.'), ('and', 'CONJ'), ('by', 'ADP'), ('*-1', 'X'),
('complaining', '.'), ('about', 'ADP'), ('it', 'PRON'), (',', '.'), ('we',
'PRON'), ('ended', 'VERB'), ('up', 'ADV'), ('with', 'ADP'), ('a', 'DET'),
('sizable', 'ADJ'), ('fine', 'NOUN'), ('and', 'CONJ'), ('suspension', 'NOUN'),
('.', '.'), ("''", '.')]
```

[330]:
```
# accuracy
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]
```

```
accuracy = len(check)/len(tagged_seq)
accuracy
```

[330]: 0.8938053097345132

[331]:
```
incorrect_tagged_cases = [[test_run_base[i-1],j] for i, j in␣
 ↪enumerate(zip(tagged_seq, test_run_base)) if j[0]!=j[1]]
incorrect_tagged_cases
```

[331]:
```
[[('The', 'DET'), (('Contra', '.'), ('Contra', 'NOUN'))],
 [('military', 'ADJ'), (('command', 'VERB'), ('command', 'NOUN'))],
 [('from', 'ADP'), (('Honduras', '.'), ('Honduras', 'NOUN'))],
 [('O', 'X'), (('Sandinista', '.'), ('Sandinista', 'NOUN'))],
 [('major', 'ADJ'), (('offensive', '.'), ('offensive', 'NOUN'))],
 [('the', 'DET'), (('rebel', '.'), ('rebel', 'NOUN'))],
 [('*-1', 'X'), (('Bucking', '.'), ('Bucking', 'VERB'))],
 [('apparently', 'ADV'), (('drew', '.'), ('drew', 'VERB'))],
 [('its', 'PRON'), (('Eveready', '.'), ('Eveready', 'NOUN'))],
 [('what', 'PRON'), (('*T*-252', '.'), ('*T*-252', 'X'))],
 [('*-1', 'X'), (('complaining', '.'), ('complaining', 'VERB'))],
 [('ended', 'VERB'), (('up', 'ADV'), ('up', 'PRT'))]]
```

We observe that the Viterbi algorithm chooses a random tag on encountering an unknown word especially in the case of nouns where the emission probability is zero.

[332]:
```
# Cheking accuraccy using only unigram tagger
unigram=nltk.UnigramTagger(train_set)
unigram.evaluate(test_set)
```

[332]: 0.906999415090661

[333]:
```
# Cheking accuraccy using only bigram tagger
bigram=nltk.BigramTagger(train_set)
bigram.evaluate(test_set)
```

[333]: 0.2177812439071944

[334]:
```
# Cheking accuraccy using only rule based tagger
patterns = [
    (r'.*ing$', 'VERB'),              # gerund
    (r'.*ed$', 'VERB'),               # past tense
    (r'.*es$', 'VERB'),               # 3rd singular present
    (r'.*ould$', 'MD'),              # modals
    (r'.*\'s$', 'NOUN'),              # possessive nouns
    (r'.*s$', 'NOUN'),                # plural nouns
    (r'^-?[0-9]+(.[0-9]+)?$', 'NUM'), # cardinal numbers
```

```
    (r'.*', 'NOUN')                    # nouns
 ]
regexp=nltk.RegexpTagger(patterns)
regexp.evaluate(test_set)
```

[334]: 0.34080717488789236

[335]:
```
# using Rules based tagger as a backoff to unigram tagger and tagging the test␣
 ↪sentences
rule_based_tagger=nltk.RegexpTagger(patterns)
lexicon_tagger=nltk.NgramTagger(1,train_set,backoff=rule_based_tagger)
lexicon_tagger.evaluate(test_set)
```

[335]: 0.9487229479430688

2.2.9 Solving the problem of unknown words

[337]:
```
# Function to use lexicom/Rule based tagger in case of unknown word
def tag_unknown_word(word):
    res = [val[1] for val in lexicon_tagger.tag([word])]
    return(str(res[0]))
```

2.2.10 Modified Viterbi Heuristic I

[338]:
```
# Modified Viterbi Heuristic - Adding Unigram and Rule based tagger in case of␣
 ↪unknown word
def Viterbi_rule_based(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)
```

```
        # If pmax is zero that means it is a UNKNOWN WORD

        if pmax == 0 :
                state_max = tag_unknown_word(words[key])
        else :
            # getting state for which probability is maximum
            state_max = T[p.index(pmax)]
        state.append(state_max)

    return list(zip(words, state))
```

[339]:
```
# tagging the test sentences
start = time.time()
tagged_seq2 = Viterbi_rule_based(test_tagged_words)
end = time.time()
difference = end-start
```

[340]:
```
print("Time taken in seconds: ", difference)
print(tagged_seq2)
#print(test_run_base2)
```

```
Time taken in seconds:  29.301992654800415
[('The', 'DET'), ('Contra', 'NOUN'), ('military', 'ADJ'), ('command', 'VERB'),
(',', '.'), ('in', 'ADP'), ('a', 'DET'), ('statement', 'NOUN'), ('from', 'ADP'),
('Honduras', 'NOUN'), (',', '.'), ('said', 'VERB'), ('0', 'X'), ('Sandinista',
'NOUN'), ('troops', 'NOUN'), ('had', 'VERB'), ('launched', 'VERB'), ('a',
'DET'), ('major', 'ADJ'), ('offensive', 'NOUN'), ('against', 'ADP'), ('the',
'DET'), ('rebel', 'NOUN'), ('forces', 'NOUN'), ('.', '.'), ('*-1', 'X'),
('Bucking', 'VERB'), ('the', 'DET'), ('market', 'NOUN'), ('trend', 'NOUN'),
(',', '.'), ('an', 'DET'), ('issue', 'NOUN'), ('of', 'ADP'), ('$', '.'), ('130',
'NUM'), ('million', 'NUM'), ('*U*', 'X'), ('general', 'ADJ'), ('obligation',
'NOUN'), ('distributable', 'ADJ'), ('state', 'NOUN'), ('aid', 'NOUN'), ('bonds',
'NOUN'), ('from', 'ADP'), ('Detroit', 'NOUN'), (',', '.'), ('Mich.', 'NOUN'),
(',', '.'), ('apparently', 'ADV'), ('drew', 'NOUN'), ('solid', 'ADJ'),
('investor', 'NOUN'), ('interest', 'NOUN'), ('.', '.'), ('Ralston', 'NOUN'),
('said', 'VERB'), ('0', 'X'), ('its', 'PRON'), ('Eveready', 'NOUN'), ('battery',
'NOUN'), ('unit', 'NOUN'), ('was', 'VERB'), ('hurt', 'VERB'), ('*-1', 'X'),
('by', 'ADP'), ('continuing', 'VERB'), ('economic', 'ADJ'), ('problems',
'NOUN'), ('in', 'ADP'), ('South', 'NOUN'), ('America', 'NOUN'), ('.', '.'),
('I', 'PRON'), ('feel', 'VERB'), ('pretty', 'ADV'), ('good', 'ADJ'), ('about',
'ADP'), ('it', 'PRON'), ('.', '.'), ('Mr.', 'NOUN'), ('Felten', 'NOUN'),
('said', 'VERB'), (',', '.'), ('``', '.'), ('We', 'PRON'), ('got', 'VERB'),
('what', 'PRON'), ('*T*-252', 'NOUN'), ('amounted', 'VERB'), ('to', 'PRT'),
('a', 'DET'), ('parking', 'NOUN'), ('ticket', 'NOUN'), (',', '.'), ('and',
'CONJ'), ('by', 'ADP'), ('*-1', 'X'), ('complaining', 'VERB'), ('about', 'ADP'),
('it', 'PRON'), (',', '.'), ('we', 'PRON'), ('ended', 'VERB'), ('up', 'ADV'),
('with', 'ADP'), ('a', 'DET'), ('sizable', 'ADJ'), ('fine', 'NOUN'), ('and',
```

```
'CONJ'), ('suspension', 'NOUN'), ('.', '.'), ("''", '.')]
```

[341]:
```
# accuracy
check2 = [i for i, j in zip(tagged_seq2, test_run_base) if i == j]

accuracy2 = len(check2)/len(tagged_seq2)
accuracy2
```

[341]:
```
0.9646017699115044
```

[342]:
```
incorrect_tagged_cases2 = [[test_run_base[i-1],j] for i, j in␣
 ↪enumerate(zip(tagged_seq2, test_run_base)) if j[0]!=j[1]]
incorrect_tagged_cases2
```

[342]:
```
[[('military', 'ADJ'), (('command', 'VERB'), ('command', 'NOUN'))],
 [('apparently', 'ADV'), (('drew', 'NOUN'), ('drew', 'VERB'))],
 [('what', 'PRON'), (('*T*-252', 'NOUN'), ('*T*-252', 'X'))],
 [('ended', 'VERB'), (('up', 'ADV'), ('up', 'PRT'))]]
```

Evaluating tagging accuracy

[380]:
```
# accuracy for Vanilla Viterbi
check = [i for i, j in zip(tagged_seq, test_run_base) if i == j]

accuracy = len(check)/len(tagged_seq)
accuracy
```

[380]:
```
0.8938053097345132
```

2.2.11 Comparing the tagging accuracies of the modifications with the vanilla Viterbi algorithm

[381]:
```
# accuracy using Vanilla Viterbi with Unigram and Rule based taggers
accuracy2
```

[381]:
```
0.9646017699115044
```

Accuracy using Vanilla Viterbi algorithm : 89%

Accuracy from using Vanilla Viterbi algorithm with unigram and Rule based taggers : 96%

2.2.12 Cases which were incorrectly tagged by original POS tagger and got corrected by the modifications

[383]:
```
## Testing with test file with sample sentences
sentence_test = 'Android is a mobile operating system developed by Google.␣
 ↪Android has been the best-selling OS worldwide on smartphones since 2011 and␣
 ↪on tablets since 2013. Google and Twitter made a deal in 2015 that gave Google␣
 ↪access to Twitters firehose. Twitter is an online news and social networking␣
 ↪service on which users post and interact with messages known as tweets. Before␣
 ↪entering politics, Donald Trump was a domineering businessman and a television␣
 ↪personality. The 2018 FIFA World Cup is the 21st FIFA World Cup, an␣
 ↪international football tournament contested once every four years. This is the␣
 ↪first World Cup to be held in Eastern Europe and the 11th time that it has␣
 ↪been held in Europe. Show me the cheapest round trips from Dallas to Atlanta.␣
 ↪I would like to see flights from Denver to Philadelphia. Show me the price of␣
 ↪the flights leaving Atlanta at about 3 in the afternoon and arriving in San␣
 ↪Francisco. NASA invited social media users to experience the launch of␣
 ↪ICESAT-2 Satellite.'
words = word_tokenize(sentence_test)
```

[384]:
```
# using Vanilla Viterbi algorithm
start = time.time()
tagged_sent_seq = Viterbi(words)
end = time.time()
difference = end-start
```

[385]:
```
print(tagged_sent_seq)
print(difference)
```

```
[('Android', '.'), ('is', 'VERB'), ('a', 'DET'), ('mobile', 'ADJ'),
('operating', 'NOUN'), ('system', 'NOUN'), ('developed', 'VERB'), ('by', 'ADP'),
('Google', '.'), ('.', '.'), ('Android', '.'), ('has', 'VERB'), ('been',
'VERB'), ('the', 'DET'), ('best-selling', 'ADJ'), ('OS', '.'), ('worldwide',
'.'), ('on', 'ADP'), ('smartphones', '.'), ('since', 'ADP'), ('2011', '.'),
('and', 'CONJ'), ('on', 'ADP'), ('tablets', 'NOUN'), ('since', 'ADP'), ('2013',
'.'), ('.', '.'), ('Google', '.'), ('and', 'CONJ'), ('Twitter', '.'), ('made',
'VERB'), ('a', 'DET'), ('deal', 'NOUN'), ('in', 'ADP'), ('2015', '.'), ('that',
'DET'), ('gave', 'VERB'), ('Google', '.'), ('access', 'NOUN'), ('to', 'PRT'),
('Twitters', '.'), ('firehose', '.'), ('.', '.'), ('Twitter', '.'), ('is',
'VERB'), ('an', 'DET'), ('online', '.'), ('news', 'NOUN'), ('and', 'CONJ'),
('social', 'ADJ'), ('networking', 'NOUN'), ('service', 'NOUN'), ('on', 'ADP'),
('which', 'DET'), ('users', 'NOUN'), ('post', 'NOUN'), ('and', 'CONJ'),
('interact', '.'), ('with', 'ADP'), ('messages', '.'), ('known', 'VERB'), ('as',
'ADP'), ('tweets', '.'), ('.', '.'), ('Before', 'ADP'), ('entering', 'VERB'),
('politics', 'NOUN'), (',', '.'), ('Donald', 'NOUN'), ('Trump', 'NOUN'), ('was',
'VERB'), ('a', 'DET'), ('domineering', '.'), ('businessman', 'NOUN'), ('and',
'CONJ'), ('a', 'DET'), ('television', 'NOUN'), ('personality', '.'), ('.', '.'),
('The', 'DET'), ('2018', '.'), ('FIFA', '.'), ('World', 'NOUN'), ('Cup', '.'),
```

```
('is', 'VERB'), ('the', 'DET'), ('21st', '.'), ('FIFA', '.'), ('World', 'NOUN'),
('Cup', '.'), (',', '.'), ('an', 'DET'), ('international', 'ADJ'), ('football',
'NOUN'), ('tournament', '.'), ('contested', '.'), ('once', 'ADV'), ('every',
'DET'), ('four', 'NUM'), ('years', 'NOUN'), ('.', '.'), ('This', 'DET'), ('is',
'VERB'), ('the', 'DET'), ('first', 'ADJ'), ('World', 'NOUN'), ('Cup', '.'),
('to', 'PRT'), ('be', 'VERB'), ('held', 'VERB'), ('in', 'ADP'), ('Eastern',
'NOUN'), ('Europe', 'NOUN'), ('and', 'CONJ'), ('the', 'DET'), ('11th', 'ADJ'),
('time', 'NOUN'), ('that', 'ADP'), ('it', 'PRON'), ('has', 'VERB'), ('been',
'VERB'), ('held', 'VERB'), ('in', 'ADP'), ('Europe', 'NOUN'), ('.', '.'),
('Show', 'NOUN'), ('me', 'PRON'), ('the', 'DET'), ('cheapest', 'ADJ'), ('round',
'NOUN'), ('trips', '.'), ('from', 'ADP'), ('Dallas', 'NOUN'), ('to', 'PRT'),
('Atlanta', 'NOUN'), ('.', '.'), ('I', 'PRON'), ('would', 'VERB'), ('like',
'ADP'), ('to', 'PRT'), ('see', 'VERB'), ('flights', 'NOUN'), ('from', 'ADP'),
('Denver', 'NOUN'), ('to', 'PRT'), ('Philadelphia', 'NOUN'), ('.', '.'),
('Show', 'NOUN'), ('me', 'PRON'), ('the', 'DET'), ('price', 'NOUN'), ('of',
'ADP'), ('the', 'DET'), ('flights', 'NOUN'), ('leaving', 'VERB'), ('Atlanta',
'NOUN'), ('at', 'ADP'), ('about', 'ADP'), ('3', 'NUM'), ('in', 'ADP'), ('the',
'DET'), ('afternoon', 'NOUN'), ('and', 'CONJ'), ('arriving', '.'), ('in',
'ADP'), ('San', 'NOUN'), ('Francisco', 'NOUN'), ('.', '.'), ('NASA', '.'),
('invited', '.'), ('social', 'ADJ'), ('media', 'NOUN'), ('users', 'NOUN'),
('to', 'PRT'), ('experience', 'NOUN'), ('the', 'DET'), ('launch', 'NOUN'),
('of', 'ADP'), ('ICESAT-2', '.'), ('Satellite', '.'), ('.', '.')]
39.458866596221924
```

[386]:
```
# using Vanilla Viterbi algorithm with unigram and Rule based taggers
start = time.time()
sent_tagged_seq2 = Viterbi_rule_based(words)
end = time.time()
difference = end-start
```

[387]:
```
print(sent_tagged_seq2)
print(difference)
```

```
[('Android', 'NOUN'), ('is', 'VERB'), ('a', 'DET'), ('mobile', 'ADJ'),
('operating', 'NOUN'), ('system', 'NOUN'), ('developed', 'VERB'), ('by', 'ADP'),
('Google', 'NOUN'), ('.', '.'), ('Android', 'NOUN'), ('has', 'VERB'), ('been',
'VERB'), ('the', 'DET'), ('best-selling', 'ADJ'), ('OS', 'NOUN'), ('worldwide',
'NOUN'), ('on', 'ADP'), ('smartphones', 'VERB'), ('since', 'ADP'), ('2011',
'NUM'), ('and', 'CONJ'), ('on', 'ADP'), ('tablets', 'NOUN'), ('since', 'ADP'),
('2013', 'NUM'), ('.', '.'), ('Google', 'NOUN'), ('and', 'CONJ'), ('Twitter',
'NOUN'), ('made', 'VERB'), ('a', 'DET'), ('deal', 'NOUN'), ('in', 'ADP'),
('2015', 'NUM'), ('that', 'ADP'), ('gave', 'VERB'), ('Google', 'NOUN'),
('access', 'NOUN'), ('to', 'PRT'), ('Twitters', 'NOUN'), ('firehose', 'NOUN'),
('.', '.'), ('Twitter', 'NOUN'), ('is', 'VERB'), ('an', 'DET'), ('online',
'NOUN'), ('news', 'NOUN'), ('and', 'CONJ'), ('social', 'ADJ'), ('networking',
'NOUN'), ('service', 'NOUN'), ('on', 'ADP'), ('which', 'DET'), ('users',
'NOUN'), ('post', 'NOUN'), ('and', 'CONJ'), ('interact', 'NOUN'), ('with',
'ADP'), ('messages', 'VERB'), ('known', 'VERB'), ('as', 'ADP'), ('tweets',
```

```
'NOUN'), ('.', '.'), ('Before', 'ADP'), ('entering', 'VERB'), ('politics',
'NOUN'), (',', '.'), ('Donald', 'NOUN'), ('Trump', 'NOUN'), ('was', 'VERB'),
('a', 'DET'), ('domineering', 'VERB'), ('businessman', 'NOUN'), ('and', 'CONJ'),
('a', 'DET'), ('television', 'NOUN'), ('personality', 'NOUN'), ('.', '.'),
('The', 'DET'), ('2018', 'NUM'), ('FIFA', 'NOUN'), ('World', 'NOUN'), ('Cup',
'NOUN'), ('is', 'VERB'), ('the', 'DET'), ('21st', 'NOUN'), ('FIFA', 'NOUN'),
('World', 'NOUN'), ('Cup', 'NOUN'), (',', '.'), ('an', 'DET'), ('international',
'ADJ'), ('football', 'NOUN'), ('tournament', 'NOUN'), ('contested', 'VERB'),
('once', 'ADV'), ('every', 'DET'), ('four', 'NUM'), ('years', 'NOUN'), ('.',
'.'), ('This', 'DET'), ('is', 'VERB'), ('the', 'DET'), ('first', 'ADJ'),
('World', 'NOUN'), ('Cup', 'NOUN'), ('to', 'PRT'), ('be', 'VERB'), ('held',
'VERB'), ('in', 'ADP'), ('Eastern', 'NOUN'), ('Europe', 'NOUN'), ('and',
'CONJ'), ('the', 'DET'), ('11th', 'ADJ'), ('time', 'NOUN'), ('that', 'ADP'),
('it', 'PRON'), ('has', 'VERB'), ('been', 'VERB'), ('held', 'VERB'), ('in',
'ADP'), ('Europe', 'NOUN'), ('.', '.'), ('Show', 'NOUN'), ('me', 'PRON'),
('the', 'DET'), ('cheapest', 'ADJ'), ('round', 'NOUN'), ('trips', 'NOUN'),
('from', 'ADP'), ('Dallas', 'NOUN'), ('to', 'PRT'), ('Atlanta', 'NOUN'), ('.',
'.'), ('I', 'PRON'), ('would', 'VERB'), ('like', 'ADP'), ('to', 'PRT'), ('see',
'VERB'), ('flights', 'NOUN'), ('from', 'ADP'), ('Denver', 'NOUN'), ('to',
'PRT'), ('Philadelphia', 'NOUN'), ('.', '.'), ('Show', 'NOUN'), ('me', 'PRON'),
('the', 'DET'), ('price', 'NOUN'), ('of', 'ADP'), ('the', 'DET'), ('flights',
'NOUN'), ('leaving', 'VERB'), ('Atlanta', 'NOUN'), ('at', 'ADP'), ('about',
'ADP'), ('3', 'NUM'), ('in', 'ADP'), ('the', 'DET'), ('afternoon', 'NOUN'),
('and', 'CONJ'), ('arriving', 'VERB'), ('in', 'ADP'), ('San', 'NOUN'),
('Francisco', 'NOUN'), ('.', '.'), ('NASA', 'NOUN'), ('invited', 'VERB'),
('social', 'ADJ'), ('media', 'NOUN'), ('users', 'NOUN'), ('to', 'PRT'),
('experience', 'NOUN'), ('the', 'DET'), ('launch', 'NOUN'), ('of', 'ADP'),
('ICESAT-2', 'NOUN'), ('Satellite', 'NOUN'), ('.', '.')]
40.238938331604004
```

2.2.13 Sample cases where we can see incorrect tagging from using Vanilla Viterbi algorithm

('Android', '.')

('NASA', '.')

('Google', '.')

2.2.14 Sample cases where we can see correct tagging from using Vanilla Viterbi algorithm with unigram and Rule based taggers

('Android', 'NOUN')

('NASA', 'NOUN')

('Google', 'NOUN')

2.3 pos-tagging-using-RNN models

Building POS tagger using Long short-term memory (LSTM) and Bidirectional LSTM, these are recurrent neural networks architecture. Below are the steps for building the model. We will use a combination of the Tree-bank, brown and CONLL 2000 Chunking Corpus corpus available in NLTK library to implement the RNN models. Below are the steps that are used for building the model.

- Preprocess data
- Vanilla RNN
- Word Embeddings
- LSTM
- Bidirectional LSTM
- Model Evaluation

[9]:
```
# import necessary libraries
import warnings
warnings.filterwarnings("ignore")

import numpy as np

from matplotlib import pyplot as plt

from nltk.corpus import brown
from nltk.corpus import treebank
from nltk.corpus import conll2000

import seaborn as sns

from gensim.models import KeyedVectors

from keras.preprocessing.sequence import pad_sequences
from keras.utils.np_utils import to_categorical
from keras.models import Sequential
from keras.layers import Embedding
from keras.layers import Dense, Input
from keras.layers import TimeDistributed
from keras.layers import LSTM, Bidirectional, SimpleRNN, RNN
from keras.models import Model
from keras.preprocessing.text import Tokenizer

from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
```

3 1. Preprocess data

3.1 Load data

[10]:
```
# load POS tagged corpora from NLTK
treebank_corpus = treebank.tagged_sents(tagset='universal')
brown_corpus = brown.tagged_sents(tagset='universal')
conll_corpus = conll2000.tagged_sents(tagset='universal')
tagged_sentences = treebank_corpus + brown_corpus + conll_corpus
```

[11]:
```
# let's look at the data
tagged_sentences[11]
```

[11]:
```
[('``', '.'),
 ('We', 'PRON'),
 ('have', 'VERB'),
 ('no', 'DET'),
 ('useful', 'ADJ'),
 ('information', 'NOUN'),
 ('on', 'ADP'),
 ('whether', 'ADP'),
 ('users', 'NOUN'),
 ('are', 'VERB'),
 ('at', 'ADP'),
 ('risk', 'NOUN'),
 (',', '.'),
 ("''", '.'),
 ('said', 'VERB'),
 ('*T*-1', 'X'),
 ('James', 'NOUN'),
 ('A.', 'NOUN'),
 ('Talcott', 'NOUN'),
 ('of', 'ADP'),
 ('Boston', 'NOUN'),
 ("'s", 'PRT'),
 ('Dana-Farber', 'NOUN'),
 ('Cancer', 'NOUN'),
 ('Institute', 'NOUN'),
 ('.', '.')]
```

3.2 Divide data in words (X) and tags (Y)

Since this is a **many-to-many** problem, each data point will be a different sentence of the corpora.

Each data point will have multiple words in the **input sequence**. This is what we will refer to as **X**.

Each word will have its correpsonding tag in the **output sequence**. This what we will refer to as **Y**.

Sample dataset:

X	Y
Mr. Vinken is chairman of Elsevier	NOUN NOUN VERB NOUN ADP NOUN
We have no useful information	PRON VERB DET ADJ NOUN

[12]:
```
X = [] # store input sequence
Y = [] # store output sequence

for sentence in tagged_sentences:
    X_sentence = []
    Y_sentence = []
    for entity in sentence:
        X_sentence.append(entity[0])  # entity[0] contains the word
        Y_sentence.append(entity[1])  # entity[1] contains corresponding tag

    X.append(X_sentence)
    Y.append(Y_sentence)
```

[13]:
```
num_words = len(set([word.lower() for sentence in X for word in sentence]))
num_tags   = len(set([word.lower() for sentence in Y for word in sentence]))
```

[14]:
```
print("Total number of tagged sentences: {}".format(len(X)))
print("Vocabulary size: {}".format(num_words))
print("Total number of tags: {}".format(num_tags))
```

```
Total number of tagged sentences: 72202
Vocabulary size: 59448
Total number of tags: 12
```

[7]:
```
# let's look at first data point
# this is one data point that will be fed to the RNN
print('sample X: ', X[0], '\n')
print('sample Y: ', Y[0], '\n')
```

```
sample X:  ['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join',
'the', 'board', 'as', 'a', 'nonexecutive', 'director', 'Nov.', '29', '.']

sample Y:  ['NOUN', 'NOUN', '.', 'NUM', 'NOUN', 'ADJ', '.', 'VERB', 'VERB',
'DET', 'NOUN', 'ADP', 'DET', 'ADJ', 'NOUN', 'NOUN', 'NUM', '.']
```

[8]:
```
# In this many-to-many problem, the length of each input and output sequence␣
 ↪must be the same.
# Since each word is tagged, it's important to make sure that the length of␣
 ↪input sequence equals the output sequence
```

```
print("Length of first input sequence  : {}".format(len(X[0])))
print("Length of first output sequence : {}".format(len(Y[0])))
```

```
Length of first input sequence  : 18
Length of first output sequence : 18
```

3.3 Vectorise X and Y

Encode X and Y to integer values We will use the Tokenizer() function from Keras library to encode text sequence to integer sequence

[9]:
```
# encode X

word_tokenizer = Tokenizer()                      # instantiate tokeniser
word_tokenizer.fit_on_texts(X)                    # fit tokeniser on data
X_encoded = word_tokenizer.texts_to_sequences(X)  # use the tokeniser to encode␣
 ↪input sequence
```

[10]:
```
# encode Y

tag_tokenizer = Tokenizer()
tag_tokenizer.fit_on_texts(Y)
Y_encoded = tag_tokenizer.texts_to_sequences(Y)
```

[11]:
```
# look at first encoded data point

print("** Raw data point **", "\n", "-"*100, "\n")
print('X: ', X[0], '\n')
print('Y: ', Y[0], '\n')
print()
print("** Encoded data point **", "\n", "-"*100, "\n")
print('X: ', X_encoded[0], '\n')
print('Y: ', Y_encoded[0], '\n')
```

```
** Raw data point **
 ----------------------------------------------------------------------------------
--------------------

X:  ['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the',
'board', 'as', 'a', 'nonexecutive', 'director', 'Nov.', '29', '.']

Y:  ['NOUN', 'NOUN', '.', 'NUM', 'NOUN', 'ADJ', '.', 'VERB', 'VERB', 'DET',
'NOUN', 'ADP', 'DET', 'ADJ', 'NOUN', 'NOUN', 'NUM', '.']


** Encoded data point **
 ----------------------------------------------------------------------------------
--------------------
```

```
X:  [6423, 24231, 2, 7652, 102, 170, 2, 47, 1898, 1, 269, 17, 7, 13230, 619,
1711, 2761, 3]

Y:  [1, 1, 3, 11, 1, 6, 3, 2, 2, 5, 1, 4, 5, 6, 1, 1, 11, 3]
```

[12]:
```
# make sure that each sequence of input and output is same length

different_length = [1 if len(input) != len(output) else 0 for input, output in␣
 ↪zip(X_encoded, Y_encoded)]
print("{} sentences have disparate input-output lengths.".
 ↪format(sum(different_length)))
```

```
0 sentences have disparate input-output lengths.
```

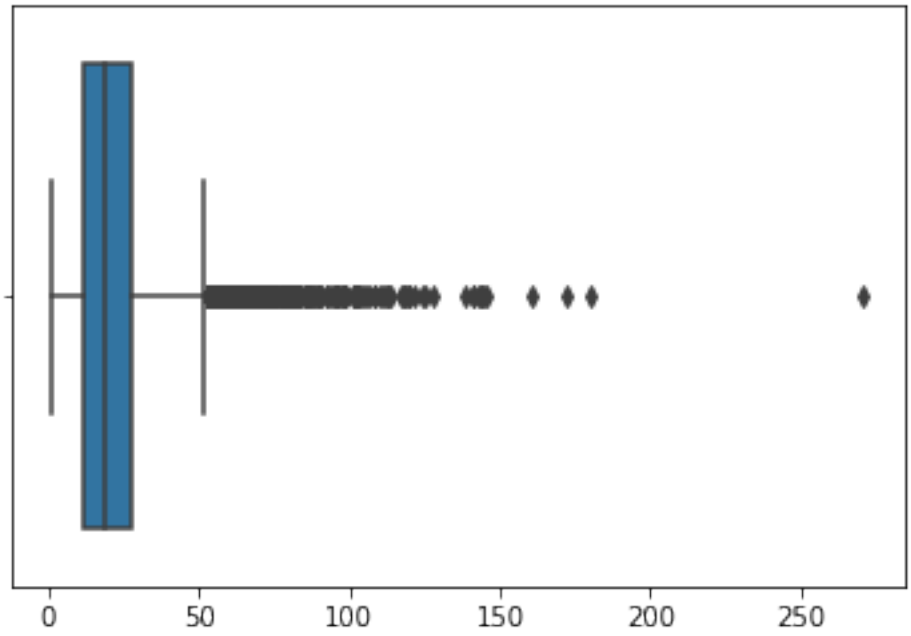
3.4 Pad sequences

The next step after encoding the data is to **define the sequence lengths**. As of now, the sentences present in the data are of various lengths. We need to either pad short sentences or truncate long sentences to a fixed length. This fixed length, however, is a **hyperparameter**.

[13]:
```
# check length of longest sentence
lengths = [len(seq) for seq in X_encoded]
print("Length of longest sentence: {}".format(max(lengths)))
```

```
Length of longest sentence: 271
```

[14]:
```
sns.boxplot(lengths)
plt.show()
```



[15]:
```
# Pad each sequence to MAX_SEQ_LENGTH using KERAS' pad_sequences() function.
# Sentences longer than MAX_SEQ_LENGTH are truncated.
# Sentences shorter than MAX_SEQ_LENGTH are padded with zeroes.

# Truncation and padding can either be 'pre' or 'post'.
# For padding we are using 'pre' padding type, that is, add zeroes on the left␣
 ↪side.
# For truncation, we are using 'post', that is, truncate a sentence from right␣
 ↪side.

MAX_SEQ_LENGTH = 100  # sequences greater than 100 in length will be truncated

X_padded = pad_sequences(X_encoded, maxlen=MAX_SEQ_LENGTH, padding="pre",␣
 ↪truncating="post")
Y_padded = pad_sequences(Y_encoded, maxlen=MAX_SEQ_LENGTH, padding="pre",␣
 ↪truncating="post")
```

[16]:
```
# print the first sequence
print(X_padded[0], "\n"*3)
print(Y_padded[0])
```

```
[   0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0
```

```
    0     0     0     0     0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0     0     0     0     0
    0     0     0     0     0     0     0     0     0     0  6423 24231
    2  7652   102   170     2    47  1898     1   269    17     7 13230
  619  1711  2761     3]
```

```
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  1  1  3 11  1  6  3  2  2  5  1  4  5  6
  1  1 11  3]
```

RNN will learn the zero to zero mapping while training. So we don't need to worry about the padded zeroes. Please note that zero is not reserved for any word or tag, it's only reserved for padding.

[17]:
```
# assign padded sequences to X and Y
X, Y = X_padded, Y_padded
```

3.5 Word embeddings

Currently, each word and each tag is encoded as an integer.

We will use a more sophisticated technique to represent the input words (X) using what's known as **word embeddings**.

However, to represent each tag in Y, we will simply use **one-hot encoding** scheme since there are only 13 tags in the dataset and the LSTM will have no problems in learning its own representation of these tags. We're using the word2vec model . Dimensions of a word embedding is: (VOCABULARY_SIZE, EMBEDDING_DIMENSION)

3.5.1 Use word embeddings for input sequences (X)

[18]:
```
# word2vec download link (Size ~ 1.5GB): https://drive.google.com/file/d/
↪0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit

path = 'storage/word-embeddings/GoogleNews-vectors-negative300.bin'

# load word2vec using the following function present in the gensim library
word2vec = KeyedVectors.load_word2vec_format(path, binary=True)
```

[19]:
```
# word2vec effectiveness
word2vec.most_similar(positive = ["King", "Woman"], negative = ["Man"])
```

[19]:
```
[('Queen', 0.4929388165473938),
 ('Tupou_V.', 0.45174291729927063),
 ('Oprah_BFF_Gayle', 0.4422132670879364),
```

```
 ('Jackson', 0.440250426530838),
 ('NECN_Alison', 0.43312832713127136),
 ('Whitfield', 0.42834725975990295),
 ('Ida_Vandross', 0.42084527015686035),
 ('prosecutor_Dan_Satterberg', 0.42075902223587036),
 ('martin_Luther_King', 0.42059648036956787),
 ('Coretta_King', 0.42027339339256287)]
```

[20]:
```
# assign word vectors from word2vec model

EMBEDDING_SIZE  = 300  # each word in word2vec model is represented using a 300␣
 ↪dimensional vector
VOCABULARY_SIZE = len(word_tokenizer.word_index) + 1

# create an empty embedding matix
embedding_weights = np.zeros((VOCABULARY_SIZE, EMBEDDING_SIZE))

# create a word to index dictionary mapping
word2id = word_tokenizer.word_index

# copy vectors from word2vec model to the words present in corpus
for word, index in word2id.items():
    try:
        embedding_weights[index, :] = word2vec[word]
    except KeyError:
        pass
```

[21]:
```
# check embedding dimension
print("Embeddings shape: {}".format(embedding_weights.shape))
```

```
Embeddings shape: (59449, 300)
```

[22]:
```
# let's look at an embedding of a word
embedding_weights[word_tokenizer.word_index['joy']]
```

[22]:
```
array([ 0.4453125 , -0.20019531,  0.20019531, -0.03149414,  0.078125  ,
       -0.390625  ,  0.13671875, -0.13867188,  0.05395508,  0.10546875,
       -0.05029297, -0.23730469,  0.19921875,  0.12597656, -0.12695312,
        0.34179688,  0.06347656,  0.26757812, -0.07324219, -0.29101562,
        0.10498047,  0.11914062,  0.23730469,  0.00640869,  0.12451172,
       -0.00939941, -0.02770996,  0.03076172,  0.07421875, -0.22851562,
       -0.08056641, -0.05273438,  0.16894531,  0.19824219, -0.15625   ,
       -0.08740234,  0.10742188, -0.07177734,  0.05200195,  0.25976562,
        0.171875  , -0.13574219,  0.06738281,  0.00531006,  0.15527344,
       -0.03515625,  0.08789062,  0.3359375 , -0.12890625,  0.17578125,
       -0.08642578,  0.32421875, -0.09033203,  0.35351562,  0.24316406,
       -0.07470703, -0.06640625, -0.17578125,  0.06689453, -0.03833008,
```

```
 0.0100708 , -0.21484375, -0.03686523,  0.04394531,  0.02209473,
 0.00219727, -0.22460938,  0.03015137, -0.21582031,  0.16015625,
 0.23339844, -0.16699219, -0.09228516,  0.10644531,  0.19335938,
-0.26757812,  0.15722656, -0.08691406,  0.11181641,  0.14941406,
-0.20507812,  0.04882812, -0.07519531, -0.21582031, -0.10107422,
-0.13378906, -0.06103516,  0.05444336,  0.07470703,  0.09521484,
-0.0144043 ,  0.27929688, -0.25585938, -0.05273438, -0.22460938,
 0.10253906, -0.15136719,  0.21289062, -0.04711914, -0.12109375,
 0.04663086,  0.25976562,  0.13574219,  0.00799561,  0.02001953,
 0.1796875 ,  0.30664062,  0.06152344,  0.13574219, -0.09619141,
-0.07421875,  0.38671875,  0.20800781,  0.12695312,  0.05200195,
 0.17675781, -0.16796875, -0.19335938, -0.06152344, -0.07568359,
-0.18457031,  0.06030273, -0.15136719, -0.1953125 , -0.23339844,
 0.00738525, -0.02478027, -0.09765625, -0.06054688,  0.20214844,
-0.2734375 ,  0.00595093, -0.34570312, -0.12988281,  0.00418091,
 0.09960938,  0.0246582 ,  0.15917969, -0.02038574,  0.30273438,
-0.20800781, -0.20214844, -0.03930664, -0.06494141,  0.00436401,
-0.02270508, -0.171875  ,  0.30273438, -0.16113281, -0.49414062,
 0.3515625 ,  0.39257812,  0.09814453,  0.41796875,  0.05371094,
 0.02392578, -0.03710938, -0.08251953, -0.38671875, -0.40625   ,
-0.05664062,  0.203125  , -0.01782227,  0.3359375 ,  0.19140625,
-0.44335938,  0.00927734,  0.24804688, -0.05102539,  0.19726562,
 0.03881836,  0.03442383, -0.40039062, -0.09912109, -0.07128906,
 0.21484375, -0.01422119,  0.04907227, -0.07421875, -0.21582031,
-0.41992188,  0.02172852,  0.11083984, -0.33398438, -0.2734375 ,
-0.05322266, -0.16601562, -0.28515625, -0.12207031,  0.04882812,
-0.0625    , -0.04077148, -0.16503906,  0.0480957 , -0.21191406,
 0.20019531, -0.2109375 ,  0.10839844, -0.14648438, -0.07958984,
-0.05151367, -0.16601562, -0.24902344, -0.375     ,  0.05664062,
-0.13671875, -0.2578125 ,  0.28515625, -0.04736328,  0.13574219,
-0.14550781,  0.19433594, -0.21972656,  0.08447266, -0.10791016,
-0.11816406, -0.16015625,  0.12060547, -0.10888672,  0.04345703,
 0.11474609, -0.08447266, -0.00720215,  0.03662109, -0.38671875,
-0.03881836, -0.03198242,  0.00344849,  0.22558594, -0.06787109,
-0.16699219,  0.2421875 ,  0.05712891,  0.27539062, -0.0456543 ,
-0.19042969, -0.17285156,  0.00836182, -0.03271484,  0.16992188,
-0.18554688, -0.0703125 , -0.32617188, -0.00668335, -0.02770996,
 0.3359375 ,  0.125     , -0.2109375 ,  0.06005859, -0.07080078,
 0.11132812,  0.125     ,  0.25390625,  0.29296875, -0.03125   ,
 0.09033203, -0.20507812, -0.07861328,  0.02062988, -0.0546875 ,
-0.23339844,  0.00096893, -0.04516602,  0.16894531, -0.22167969,
 0.08105469,  0.33398438,  0.09619141,  0.00866699, -0.03271484,
 0.05493164,  0.12109375,  0.16210938, -0.10302734,  0.27148438,
-0.03344727, -0.30273438,  0.04223633,  0.08496094, -0.15527344,
 0.10107422, -0.11474609, -0.13085938,  0.22949219,  0.12988281,
 0.09863281, -0.03588867,  0.10693359, -0.24902344,  0.19238281,
-0.05322266, -0.09033203, -0.31640625, -0.5703125 , -0.15917969,
```

```
       0.0291748 , -0.0246582 , -0.07714844, -0.04663086, -0.17578125])
```

3.5.2 Use one-hot encoding for output sequences (Y)

[24]:
```
# use Keras' to_categorical function to one-hot encode Y
Y = to_categorical(Y)
```

[26]:
```
# print Y of the first output sequqnce
print(Y.shape)
```

```
(72202, 100, 13)
```

3.6 Split data in training, validation and tesing sets

[27]:
```
# split entire data into training and testing sets
TEST_SIZE = 0.15
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=TEST_SIZE,␣
 ↪random_state=4)
```

[28]:
```
# split training data into training and validation sets
VALID_SIZE = 0.15
X_train, X_validation, Y_train, Y_validation = train_test_split(X_train,␣
 ↪Y_train, test_size=VALID_SIZE, random_state=4)
```

[29]:
```
# print number of samples in each set
print("TRAINING DATA")
print('Shape of input sequences: {}'.format(X_train.shape))
print('Shape of output sequences: {}'.format(Y_train.shape))
print("-"*50)
print("VALIDATION DATA")
print('Shape of input sequences: {}'.format(X_validation.shape))
print('Shape of output sequences: {}'.format(Y_validation.shape))
print("-"*50)
print("TESTING DATA")
print('Shape of input sequences: {}'.format(X_test.shape))
print('Shape of output sequences: {}'.format(Y_test.shape))
```

```
TRAINING DATA
Shape of input sequences: (52165, 100)
Shape of output sequences: (52165, 100, 13)
--------------------------------------------------
VALIDATION DATA
Shape of input sequences: (9206, 100)
Shape of output sequences: (9206, 100, 13)
--------------------------------------------------
TESTING DATA
```

```
Shape of input sequences: (10831, 100)
Shape of output sequences: (10831, 100, 13)
```

Before using RNN, we must make sure the dimensions of the data are what an RNN expects. In general, an RNN expects the following shape

Shape of X: (#samples, #timesteps, #features)

Shape of Y: (#samples, #timesteps, #features)

Now, there can be various variations in the shape that you use to feed an RNN depending on the type of architecture. Since the problem we're working on has a many-to-many architecture, the input and the output both include number of timesteps which is nothing but the sequence length. But notice that the tensor X doesn't have the third dimension, that is, number of features. That's because we're going to use word embeddings before feeding in the data to an RNN, and hence there is no need to explicitly mention the third dimension. That's because when you use the Embedding() layer in Keras, you the training data will automatically be converted to (#samples, #timesteps, #features) where #features will be the embedding dimention (and note that the Embedding layer is always the very first layer of an RNN). While using the embedding layer we only need to reshape the data to (#samples, #timesteps) which is what we have done. However,we need to shape it to (#samples, #timesteps, #features) in case you don't use the Embedding() layer in Keras.

4 2. Vanilla RNN

The RNN based models we are going to build are as follows:

- RNN with arbitrarily initialised, untrainable embeddings: In this model, we will initialise the embedding weights arbitrarily. Further, we'll freeze the embeddings, that is, we won't allow the network to train them.
- RNN with arbitrarily initialised, trainable embeddings: In this model, we'll allow the network to train the embeddings.
- RNN with trainable word2vec embeddings: In this experiment, we'll use word2vec word embeddings and also allow the network to train them further.

4.0.1 Uninitialised fixed embeddings

First let's try running a vanilla RNN. For this RNN we won't use the pre-trained word embeddings. We willl use randomly inititalised embeddings. Moreover, we won't update the embeddings weights.

[30]:
```
# total number of tags
NUM_CLASSES = Y.shape[2]
```

[42]:
```
# create architecture

rnn_model = Sequential()

# create embedding layer - usually the first layer in text problems
```

```
rnn_model.add(Embedding(input_dim     =  VOCABULARY_SIZE,          # vocabulary␣
 ↪size - number of unique words in data
                        output_dim    =  EMBEDDING_SIZE,          # length of␣
 ↪vector with which each word is represented
                        input_length  =  MAX_SEQ_LENGTH,          # length of␣
 ↪input sequence
                        trainable     =  False                    # False -␣
 ↪don't update the embeddings
))

# add an RNN layer which contains 64 RNN cells
rnn_model.add(SimpleRNN(64,
              return_sequences=True  # True - return whole sequence; False -␣
 ↪return single output of the end of the sequence
))

# add time distributed (output at each sequence) layer
rnn_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax')))
```

4.1 Compile model

[43]:
```
rnn_model.compile(loss      =  'categorical_crossentropy',
                  optimizer =  'adam',
                  metrics   =  ['acc'])
```

[44]:
```
# check summary of the model
rnn_model.summary()
```

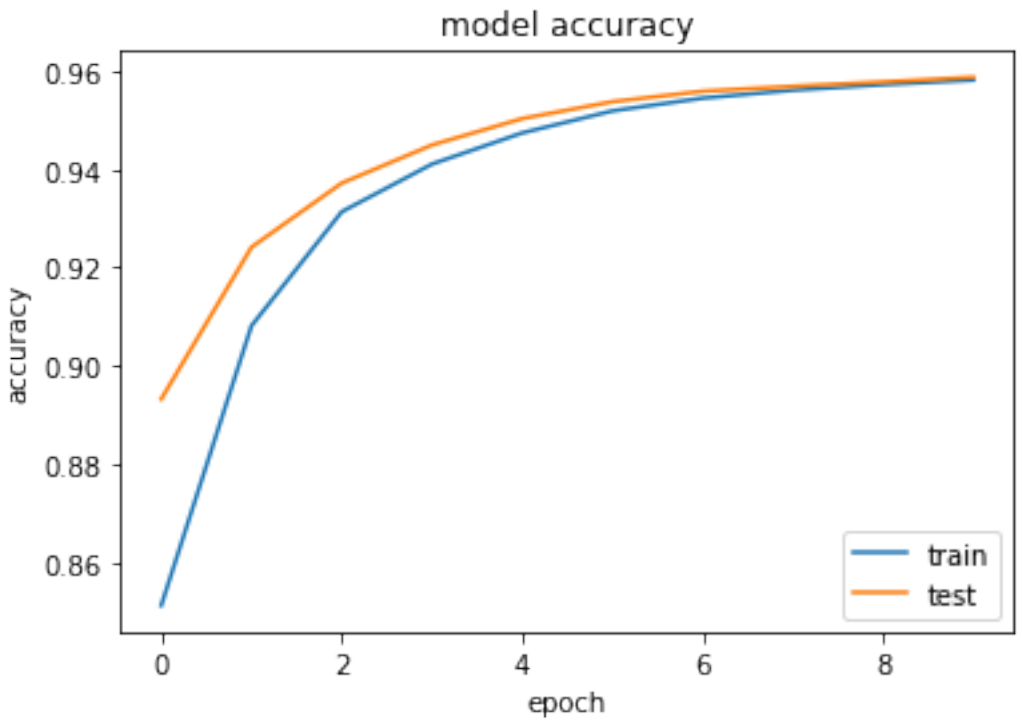
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (Embedding)      (None, 100, 300)          17834700
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, 100, 64)           23360
_________________________________________________________________
time_distributed_3 (TimeDist (None, 100, 13)           845
=================================================================
Total params: 17,858,905
Trainable params: 24,205
Non-trainable params: 17,834,700
_________________________________________________________________
```

4.2 Fit model

[45]:
```
rnn_training = rnn_model.fit(X_train, Y_train, batch_size=128, epochs=10,␣
↪validation_data=(X_validation, Y_validation))
```

```
Train on 52165 samples, validate on 9206 samples
Epoch 1/10
52165/52165 [==============================] - 27s 511us/step - loss: 0.5250 -
acc: 0.8513 - val_loss: 0.3503 - val_acc: 0.8933
Epoch 2/10
52165/52165 [==============================] - 26s 505us/step - loss: 0.2959 -
acc: 0.9082 - val_loss: 0.2468 - val_acc: 0.9241
Epoch 3/10
52165/52165 [==============================] - 25s 486us/step - loss: 0.2214 -
acc: 0.9313 - val_loss: 0.1978 - val_acc: 0.9372
Epoch 4/10
52165/52165 [==============================] - 24s 466us/step - loss: 0.1852 -
acc: 0.9410 - val_loss: 0.1711 - val_acc: 0.9449
Epoch 5/10
52165/52165 [==============================] - 25s 481us/step - loss: 0.1634 -
acc: 0.9474 - val_loss: 0.1537 - val_acc: 0.9503
Epoch 6/10
52165/52165 [==============================] - 25s 473us/step - loss: 0.1492 -
acc: 0.9518 - val_loss: 0.1425 - val_acc: 0.9537
Epoch 7/10
52165/52165 [==============================] - 30s 583us/step - loss: 0.1399 -
acc: 0.9544 - val_loss: 0.1351 - val_acc: 0.9558
Epoch 8/10
52165/52165 [==============================] - 27s 510us/step - loss: 0.1333 -
acc: 0.9561 - val_loss: 0.1300 - val_acc: 0.9568
Epoch 9/10
52165/52165 [==============================] - 26s 504us/step - loss: 0.1287 -
acc: 0.9572 - val_loss: 0.1260 - val_acc: 0.9577
Epoch 10/10
52165/52165 [==============================] - 25s 484us/step - loss: 0.1251 -
acc: 0.9581 - val_loss: 0.1229 - val_acc: 0.9587
```

[46]:
```
# visualise training history
plt.plot(rnn_training.history['acc'])
plt.plot(rnn_training.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc="lower right")
plt.show()
```



4.2.1 Uninitialised trainable embeddings

[48]:
```
# create architecture

rnn_model = Sequential()

# create embedding layer - usually the first layer in text problems
rnn_model.add(Embedding(input_dim     =  VOCABULARY_SIZE,          # vocabulary␣
 ↪size - number of unique words in data
                        output_dim    =  EMBEDDING_SIZE,           # length of␣
 ↪vector with which each word is represented
                        input_length  =  MAX_SEQ_LENGTH,           # length of␣
 ↪input sequence
                        trainable     =  True                      # True -␣
 ↪update the embeddings while training
))

# add an RNN layer which contains 64 RNN cells
rnn_model.add(SimpleRNN(64,
              return_sequences=True  # True - return whole sequence; False -␣
 ↪return single output of the end of the sequence
))
```

```
# add time distributed (output at each sequence) layer
rnn_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax')))
```

4.3 Compile model

[49]:
```
rnn_model.compile(loss      =  'categorical_crossentropy',
                  optimizer =  'adam',
                  metrics   =  ['acc'])
```

[50]:
```
# check summary of the model
rnn_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_7 (Embedding)      (None, 100, 300)          17834700
_________________________________________________________________
simple_rnn_6 (SimpleRNN)     (None, 100, 64)           23360
_________________________________________________________________
time_distributed_5 (TimeDist (None, 100, 13)           845
=================================================================
Total params: 17,858,905
Trainable params: 17,858,905
Non-trainable params: 0
_________________________________________________________________
```

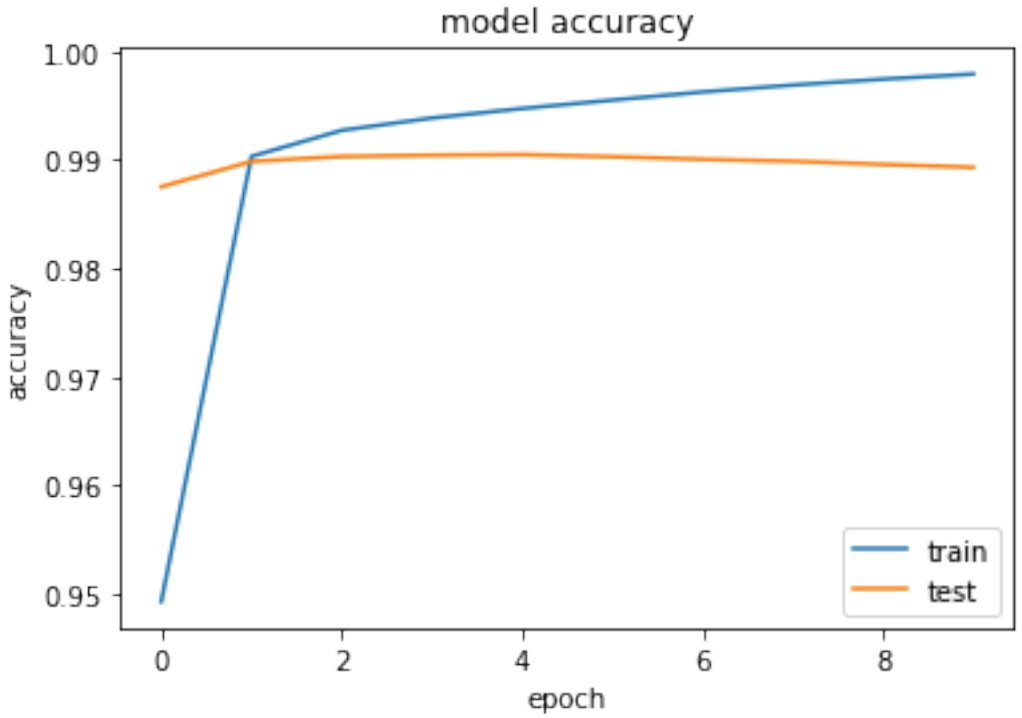
4.4 Fit model

[51]:
```
rnn_training = rnn_model.fit(X_train, Y_train, batch_size=128, epochs=10,␣
 ↪validation_data=(X_validation, Y_validation))
```

```
Train on 52165 samples, validate on 9206 samples
Epoch 1/10
52165/52165 [==============================] - 35s 676us/step - loss: 0.2149 -
acc: 0.9493 - val_loss: 0.0417 - val_acc: 0.9875
Epoch 2/10
52165/52165 [==============================] - 31s 602us/step - loss: 0.0301 -
acc: 0.9903 - val_loss: 0.0295 - val_acc: 0.9899
Epoch 3/10
52165/52165 [==============================] - 31s 592us/step - loss: 0.0209 -
acc: 0.9927 - val_loss: 0.0273 - val_acc: 0.9903
Epoch 4/10
52165/52165 [==============================] - 29s 564us/step - loss: 0.0173 -
acc: 0.9939 - val_loss: 0.0270 - val_acc: 0.9905
Epoch 5/10
52165/52165 [==============================] - 30s 569us/step - loss: 0.0149 -
acc: 0.9948 - val_loss: 0.0272 - val_acc: 0.9905
```

```
Epoch 6/10
52165/52165 [==============================] - 40s 768us/step - loss: 0.0128 -
acc: 0.9955 - val_loss: 0.0282 - val_acc: 0.9903
Epoch 7/10
52165/52165 [==============================] - 32s 607us/step - loss: 0.0110 -
acc: 0.9963 - val_loss: 0.0298 - val_acc: 0.9901
Epoch 8/10
52165/52165 [==============================] - 31s 600us/step - loss: 0.0093 -
acc: 0.9969 - val_loss: 0.0315 - val_acc: 0.9899
Epoch 9/10
52165/52165 [==============================] - 29s 559us/step - loss: 0.0078 -
acc: 0.9975 - val_loss: 0.0339 - val_acc: 0.9896
Epoch 10/10
52165/52165 [==============================] - 30s 569us/step - loss: 0.0065 -
acc: 0.9979 - val_loss: 0.0359 - val_acc: 0.9893
```

[52]:
```
# visualise training history
plt.plot(rnn_training.history['acc'])
plt.plot(rnn_training.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc="lower right")
plt.show()
```



4.4.1 Using pre-trained embedding weights

[53]:
```
# create architecture

rnn_model = Sequential()

# create embedding layer - usually the first layer in text problems
rnn_model.add(Embedding(input_dim     =  VOCABULARY_SIZE,         # vocabulary␣
 ↪size - number of unique words in data
                        output_dim    =  EMBEDDING_SIZE,          # length of␣
 ↪vector with which each word is represented
                        input_length  =  MAX_SEQ_LENGTH,          # length of␣
 ↪input sequence
                        weights       = [embedding_weights],      # word␣
 ↪embedding matrix
                        trainable     =  True                     # True -␣
 ↪update the embeddings while training
))

# add an RNN layer which contains 64 RNN cells
rnn_model.add(SimpleRNN(64,
              return_sequences=True  # True - return whole sequence; False -␣
 ↪return single output of the end of the sequence
))

# add time distributed (output at each sequence) layer
rnn_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax')))
```

4.5 Compile model

[54]:
```
rnn_model.compile(loss      =  'categorical_crossentropy',
                  optimizer =  'adam',
                  metrics   =  ['acc'])
```

[55]:
```
# check summary of the model
rnn_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_8 (Embedding)      (None, 100, 300)          17834700
_________________________________________________________________
simple_rnn_7 (SimpleRNN)     (None, 100, 64)           23360
_________________________________________________________________
time_distributed_6 (TimeDist (None, 100, 13)           845
=================================================================
Total params: 17,858,905
```

```
Trainable params: 17,858,905
Non-trainable params: 0
_________________________________________________________________
```

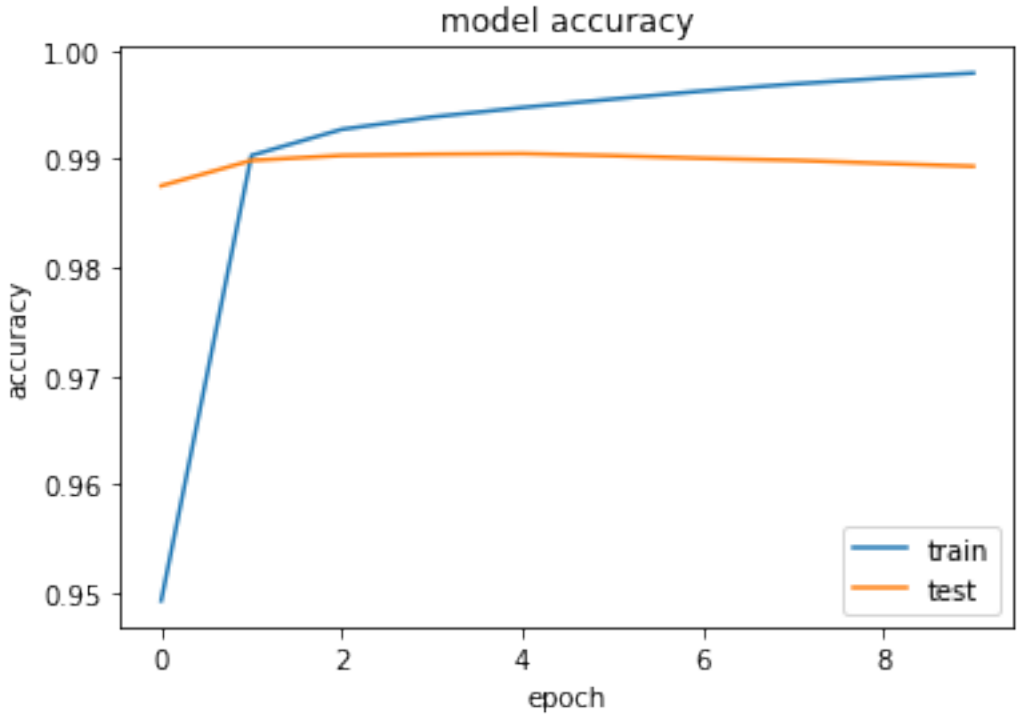
4.6 Fit model

[56]:
```
rnn_training = rnn_model.fit(X_train, Y_train, batch_size=128, epochs=10,␣
 ↪validation_data=(X_validation, Y_validation))
```

```
Train on 52165 samples, validate on 9206 samples
Epoch 1/10
52165/52165 [==============================] - 29s 547us/step - loss: 0.1853 -
acc: 0.9621 - val_loss: 0.0345 - val_acc: 0.9892
Epoch 2/10
52165/52165 [==============================] - 28s 534us/step - loss: 0.0266 -
acc: 0.9911 - val_loss: 0.0268 - val_acc: 0.9907
Epoch 3/10
52165/52165 [==============================] - 28s 540us/step - loss: 0.0198 -
acc: 0.9930 - val_loss: 0.0250 - val_acc: 0.9911
Epoch 4/10
52165/52165 [==============================] - 28s 538us/step - loss: 0.0169 -
acc: 0.9940 - val_loss: 0.0247 - val_acc: 0.9913
Epoch 5/10
52165/52165 [==============================] - 28s 537us/step - loss: 0.0149 -
acc: 0.9946 - val_loss: 0.0246 - val_acc: 0.9913
Epoch 6/10
52165/52165 [==============================] - 28s 541us/step - loss: 0.0133 -
acc: 0.9953 - val_loss: 0.0249 - val_acc: 0.9913
Epoch 7/10
52165/52165 [==============================] - 28s 537us/step - loss: 0.0117 -
acc: 0.9959 - val_loss: 0.0259 - val_acc: 0.9911
Epoch 8/10
52165/52165 [==============================] - 28s 542us/step - loss: 0.0103 -
acc: 0.9964 - val_loss: 0.0270 - val_acc: 0.9910
Epoch 9/10
52165/52165 [==============================] - 28s 542us/step - loss: 0.0090 -
acc: 0.9970 - val_loss: 0.0284 - val_acc: 0.9908
Epoch 10/10
52165/52165 [==============================] - 42s 799us/step - loss: 0.0077 -
acc: 0.9974 - val_loss: 0.0301 - val_acc: 0.9907
```

[52]:
```
# visualise training history
plt.plot(rnn_training.history['acc'])
plt.plot(rnn_training.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
```

```
plt.legend(['train', 'test'], loc="lower right")
plt.show()
```



5 2. LSTM

We will use pre-trained word embeddings in following models and allow them to be updated as well.

5.1 Create model architecture

[57]:
```
# create architecture

lstm_model = Sequential()
lstm_model.add(Embedding(input_dim     = VOCABULARY_SIZE,          # vocabulary␣
↪size - number of unique words in data
                         output_dim    = EMBEDDING_SIZE,           # length of␣
↪vector with which each word is represented
                         input_length  = MAX_SEQ_LENGTH,           # length of␣
↪input sequence
                         weights       = [embedding_weights],      # word␣
↪embedding matrix
```

```
                        trainable     = True                        # True -␣
 ↪update embeddings_weight matrix
))
lstm_model.add(LSTM(64, return_sequences=True))
lstm_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax')))
```

5.2 Compile model

[58]:
```
lstm_model.compile(loss      =  'categorical_crossentropy',
                   optimizer =  'adam',
                   metrics   =  ['acc'])
```

[59]:
```
# check summary of the model
lstm_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_9 (Embedding)      (None, 100, 300)          17834700
_________________________________________________________________
lstm_1 (LSTM)                (None, 100, 64)           93440
_________________________________________________________________
time_distributed_7 (TimeDist (None, 100, 13)           845
=================================================================
Total params: 17,928,985
Trainable params: 17,928,985
Non-trainable params: 0
_________________________________________________________________
```

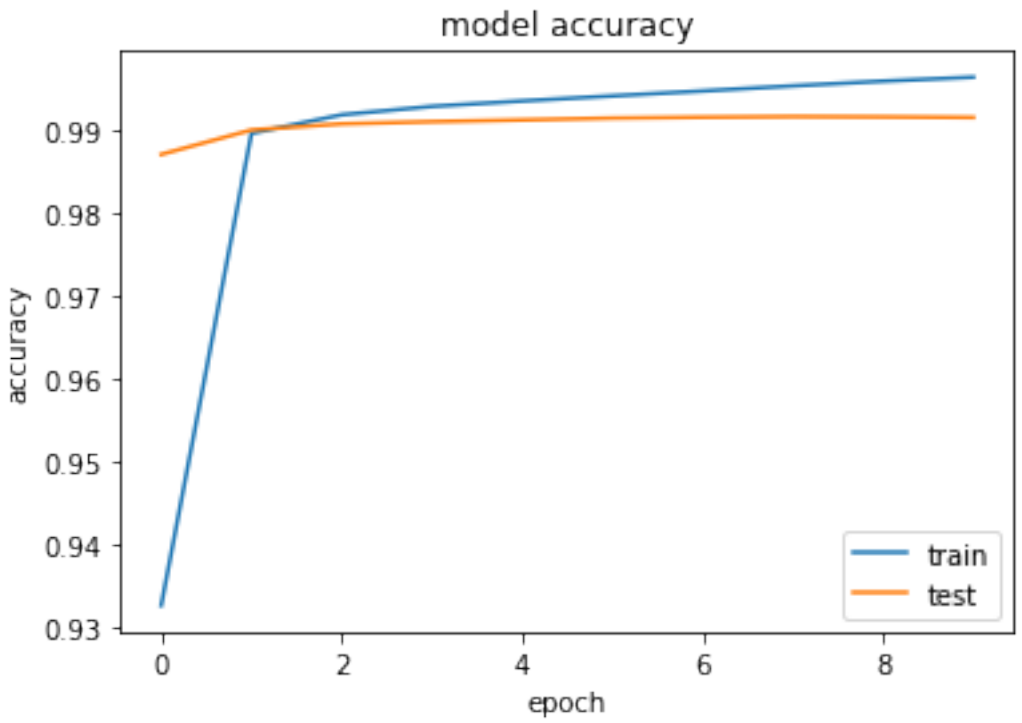
5.3 Fit model

[60]:
```
lstm_training = lstm_model.fit(X_train, Y_train, batch_size=128, epochs=10,␣
 ↪validation_data=(X_validation, Y_validation))
```

```
Train on 52165 samples, validate on 9206 samples
Epoch 1/10
52165/52165 [==============================] - 89s 2ms/step - loss: 0.3125 -
acc: 0.9327 - val_loss: 0.0463 - val_acc: 0.9871
Epoch 2/10
52165/52165 [==============================] - 74s 1ms/step - loss: 0.0333 -
acc: 0.9896 - val_loss: 0.0296 - val_acc: 0.9900
Epoch 3/10
52165/52165 [==============================] - 75s 1ms/step - loss: 0.0234 -
acc: 0.9919 - val_loss: 0.0261 - val_acc: 0.9908
Epoch 4/10
52165/52165 [==============================] - 77s 1ms/step - loss: 0.0198 -
acc: 0.9929 - val_loss: 0.0248 - val_acc: 0.9910
```

```
Epoch 5/10
52165/52165 [==============================] - 77s 1ms/step - loss: 0.0176 -
acc: 0.9935 - val_loss: 0.0240 - val_acc: 0.9912
Epoch 6/10
52165/52165 [==============================] - 77s 1ms/step - loss: 0.0159 -
acc: 0.9941 - val_loss: 0.0238 - val_acc: 0.9915
Epoch 7/10
52165/52165 [==============================] - 78s 1ms/step - loss: 0.0144 -
acc: 0.9947 - val_loss: 0.0238 - val_acc: 0.9916
Epoch 8/10
52165/52165 [==============================] - 77s 1ms/step - loss: 0.0129 -
acc: 0.9953 - val_loss: 0.0239 - val_acc: 0.9916
Epoch 9/10
52165/52165 [==============================] - 77s 1ms/step - loss: 0.0115 -
acc: 0.9959 - val_loss: 0.0246 - val_acc: 0.9916
Epoch 10/10
52165/52165 [==============================] - 77s 1ms/step - loss: 0.0103 -
acc: 0.9964 - val_loss: 0.0255 - val_acc: 0.9915
```

[61]:
```
# visualise training history
plt.plot(lstm_training.history['acc'])
plt.plot(lstm_training.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc="lower right")
plt.show()
```



6 2. Bidirectional LSTM

6.1 Create model architecture

[71]:
```
# create architecture

bidirect_model = Sequential()
bidirect_model.add(Embedding(input_dim     = VOCABULARY_SIZE,
                             output_dim    = EMBEDDING_SIZE,
                             input_length  = MAX_SEQ_LENGTH,
                             weights       = [embedding_weights],
                             trainable     = True
))
bidirect_model.add(Bidirectional(LSTM(64, return_sequences=True)))
bidirect_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax')))
```

6.2 Compile model

[72]:
```
bidirect_model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['acc'])
```

[73]:
```
# check summary of model
bidirect_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_12 (Embedding)     (None, 100, 300)          17834700
_________________________________________________________________
bidirectional_1 (Bidirection (None, 100, 128)          186880
_________________________________________________________________
time_distributed_10 (TimeDis (None, 100, 13)           1677
=================================================================
Total params: 18,023,257
Trainable params: 18,023,257
Non-trainable params: 0
_________________________________________________________________
```

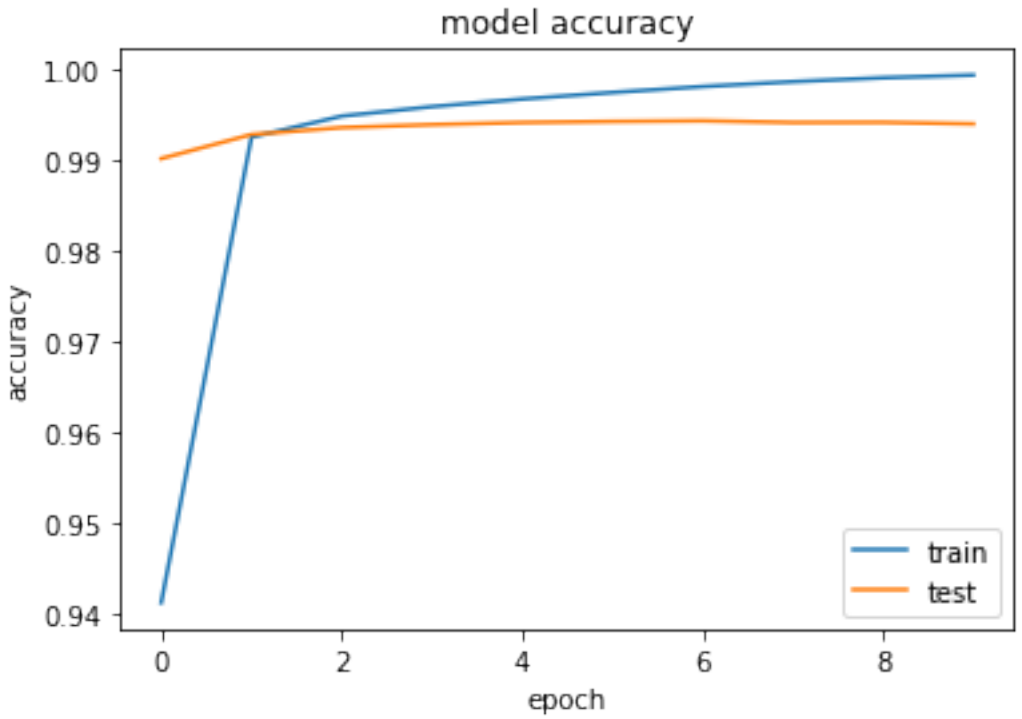
6.3 Fit model

[74]:
```
bidirect_training = bidirect_model.fit(X_train, Y_train, batch_size=128,␣
↪epochs=10, validation_data=(X_validation, Y_validation))
```

```
Train on 52165 samples, validate on 9206 samples
Epoch 1/10
52165/52165 [==============================] - 155s 3ms/step - loss: 0.2523 -
acc: 0.9413 - val_loss: 0.0322 - val_acc: 0.9901
Epoch 2/10
52165/52165 [==============================] - 156s 3ms/step - loss: 0.0234 -
acc: 0.9924 - val_loss: 0.0217 - val_acc: 0.9927
Epoch 3/10
52165/52165 [==============================] - 140s 3ms/step - loss: 0.0158 -
acc: 0.9948 - val_loss: 0.0192 - val_acc: 0.9935
Epoch 4/10
52165/52165 [==============================] - 139s 3ms/step - loss: 0.0125 -
acc: 0.9958 - val_loss: 0.0182 - val_acc: 0.9938
Epoch 5/10
52165/52165 [==============================] - 136s 3ms/step - loss: 0.0101 -
acc: 0.9966 - val_loss: 0.0178 - val_acc: 0.9941
Epoch 6/10
52165/52165 [==============================] - 138s 3ms/step - loss: 0.0081 -
acc: 0.9974 - val_loss: 0.0179 - val_acc: 0.9942
Epoch 7/10
52165/52165 [==============================] - 136s 3ms/step - loss: 0.0063 -
acc: 0.9980 - val_loss: 0.0184 - val_acc: 0.9943
Epoch 8/10
52165/52165 [==============================] - 137s 3ms/step - loss: 0.0048 -
acc: 0.9986 - val_loss: 0.0197 - val_acc: 0.9941
Epoch 9/10
```

```
52165/52165 [==============================] - 139s 3ms/step - loss: 0.0036 -
acc: 0.9990 - val_loss: 0.0208 - val_acc: 0.9941
Epoch 10/10
52165/52165 [==============================] - 141s 3ms/step - loss: 0.0027 -
acc: 0.9993 - val_loss: 0.0221 - val_acc: 0.9939
```

[75]:
```
# visualise training history
plt.plot(bidirect_training.history['acc'])
plt.plot(bidirect_training.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc="lower right")
plt.show()
```



7 4. Model evaluation

[76]:
```
loss, accuracy = rnn_model.evaluate(X_test, Y_test, verbose = 1)
print("Loss: {0},\nAccuracy: {1}".format(loss, accuracy))
```

```
10831/10831 [==============================] - 9s 812us/step
Loss: 0.030307781009474555,
Accuracy: 0.9906518321135909
```

[77]:
```
loss, accuracy = lstm_model.evaluate(X_test, Y_test, verbose = 1)
print("Loss: {0},\nAccuracy: {1}".format(loss, accuracy))
```

```
10831/10831 [==============================] - 17s 2ms/step
Loss: 0.025485771876122578,
Accuracy: 0.9915363299484542
```

[79]:
```
loss, accuracy = bidirect_model.evaluate(X_test, Y_test, verbose = 1)
print("Loss: {0},\nAccuracy: {1}".format(loss, accuracy))
```

```
10831/10831 [==============================] - 34s 3ms/step
Loss: 0.02203728499372694,
Accuracy: 0.9938629862344471
```

8 Conclusions

- In the RNN model, the results improved marginally in case of using word vectors. That's because the model was already performing very well. You'll see much more improvements by using pre-trained embeddings in cases where you don't have such a good model performance.
- The LSTM model also provided some marginal improvement. The time taken by an LSTM is greater than an RNN. This was expected since the parameters in an LSTM are 4x of a normal RNN.
- The bidirectional LSTM did increase the accuracy substantially (considering that the accuracy was already hitting the roof). This shows the power of bidirectional LSTMs. However, this increased accuracy comes at a cost. The time taken was almost double than a normal LSTM network.

References

[1] http://www.nltk.org/book/ch05.html.

[2] https://web.stanford.edu/~jurafsky/slp3/8.pdf.

[3] https://stackoverflow.com/questions/17259970/tagging-pos-in-nltk-using-backoff-ngrams.

[4] https://www.cs.jhu.edu/~jason/papers/jurafsky+martin.slp3draft.ch9.pdf.