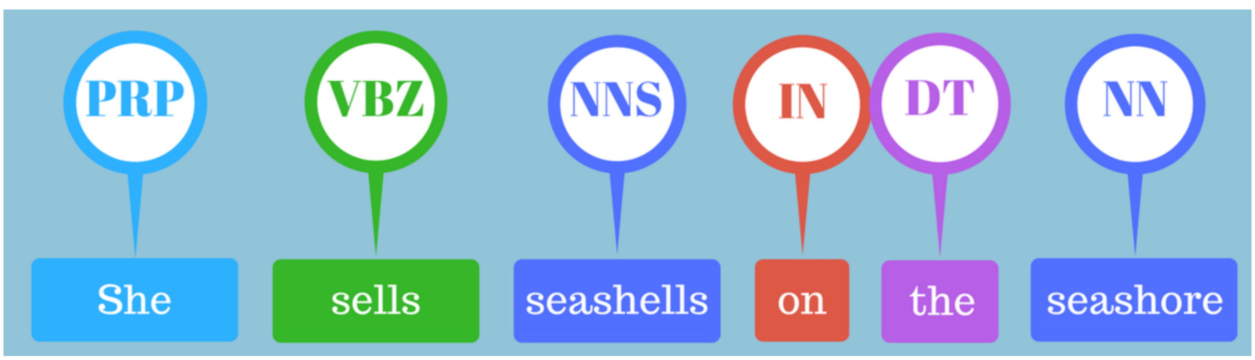
Natural Language Processing **Shakthi Prarthana V**
CS6030 **2018103062**
27.10.2021

POS TAGGER IN NLTK

Link to the GitHub Page:https://github.com/Vshakthiprarthana1570/NLP_assignments

PROBLEM STATEMENT:

Part-of-speech (POS) tagging is a popular Natural Language Processing process which refers to categorizing words in a text (corpus) in correspondence with a particular part of speech, depending on the definition of the word and its context.Part-of-speech tags describe the characteristic structure of lexical terms within a sentence or text, therefore we can use them for making assumptions about semantics. In this assignment, a POS Tagger is developed using deep learning techniques and nltk, as a result of which automatic text processing tools take into account which part of speech each word is.



DATASET USED:

The dataset used for developing the POS-Tagger is the **Penn-tree-bank dataset** which is maintained by the University of Pennsylvania.It has over four million and eight hundred thousand annotated words in it, all corrected by humans. It contains 36 POS tags and 12 other tags.The dataset is divided in different kinds of annotations, such as Piece-of-Speech, Syntactic and Semantic skeletons.

Link to the Dataset:https://www.kaggle.com/nltkdata/penn-tree-bank

PREPROCESSING THE TEXT:

REMOVING STOP WORDS AND NOISE

Here, we convert the words in text to lowercase and remove punctuations as they add noise to the text. Words are also tokenized so that they can be used in the following steps.

STEMMING

Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes.It is the process of reducing a word to its word stem that affixes to suffixes and prefixes or to the roots of words known as a lemma. The most common algorithm for stemming English is Porter's algorithm. Porter's algorithm consists of 5 phases of word reductions, applied sequentially. Within each phase there are various conventions to select rules, such as selecting the rule from each rule group that applies to the longest suffix.

For example,
Laughing → Laugh
Monkeys → Monkey

SPLITTING OF TRAINING AND VALIDATION SET

100676 tagged words are split with a percentage of 0.2. Hence, 80540 samples for training and 20135 for testing purposes.

EVALUATION METRICS

Accuracy and Loss of the trained model is measured for both the training and validation sets.

POS TAGGER CODING SNAPSHOTS:

Importing Packages

The Natural Language Toolkit (NLTK) is a platform used for building Python programs that work with human language data for applying in statistical natural language processing (NLP). It contains text processing libraries for tokenization, parsing, classification, stemming, tagging and semantic reasoning.

```
In [1]:  import nltk
         nltk.download('all')
```

```
[nltk_data] Downloading collection 'all'
[nltk_data]    |
[nltk_data]    | Downloading package abc to C:\Users\Shakthi
[nltk_data]    |     V\AppData\Roaming\nltk_data...
[nltk_data]    |   Package abc is already up-to-date!
[nltk_data]    | Downloading package alpino to C:\Users\Shakthi
[nltk_data]    |     V\AppData\Roaming\nltk_data...
[nltk_data]    |   Package alpino is already up-to-date!
[nltk_data]    | Downloading package biocreative_ppi to
[nltk_data]    |     C:\Users\Shakthi V\AppData\Roaming\nltk_data...
[nltk_data]    |   Package biocreative_ppi is already up-to-date!
[nltk_data]    | Downloading package brown to C:\Users\Shakthi
[nltk_data]    |     V\AppData\Roaming\nltk_data...
[nltk_data]    |   Package brown is already up-to-date!
[nltk_data]    | Downloading package brown_tei to C:\Users\Shakthi
[nltk_data]    |     V\AppData\Roaming\nltk_data...
[nltk_data]    |   Package brown_tei is already up-to-date!
[nltk_data]    | Downloading package cess_cat to C:\Users\Shakthi
[nltk_data]    |     V\AppData\Roaming\nltk_data...
[nltk_data]    |   Package cess_cat is already up-to-date!
```

DOWNLOADING THE DATASET:

The Penn-TreeBank dataset is available in the nltk package. It is downloaded and preprocessed for the model training. Further the data is restructured to separate the words from the tags.

```
In [15]:  import nltk
          tagged_sentences = nltk.corpus.treebank.tagged_sents()
```

```
In [16]:  print(tagged_sentences[2])
          print("Tagged sentences: ", len(tagged_sentences))
          print("Tagged words:", len(nltk.corpus.treebank.tagged_words()))
```

```
[('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'), ('.', '.')]
Tagged sentences:  3914
Tagged words: 100676
```

The data is restructured a bit.The words are separated from the tags.

```
In [17]:  import numpy as np
          sentences, sentence_tags =[], []
          for tagged_sentence in tagged_sentences:
              sentence, tags = zip(*tagged_sentence)
              sentences.append(np.array(sentence))
              sentence_tags.append(np.array(tags))
```

One of the sequences in the data is printed as follows:

In [18]:
```
print(sentences[29])
print(sentence_tags[29])
```

```
['Workers' 'described' '``' 'clouds' 'of' 'blue' 'dust' "''" 'that'
 '*T*-1' 'hung' 'over' 'parts' 'of' 'the' 'factory' ',' 'even' 'though'
 'exhaust' 'fans' 'ventilated' 'the' 'area' '.']
['NNS' 'VBD' '``' 'NNS' 'IN' 'JJ' 'NN' "''" 'WDT' '-NONE-' 'VBD' 'IN'
 'NNS' 'IN' 'DT' 'NN' ',' 'RB' 'IN' 'NN' 'NNS' 'VBD' 'DT' 'NN' '.']
```

Before training a model, the data is split into training and testing data.For that,train_test_split function from Scikit-learn is used.

In [19]:
```
from sklearn.model_selection import train_test_split
(train_sentences,  test_sentences, train_tags, test_tags) = train_test_split(sentences, sentence_tags, test_size=0.2)
```

Keras needs to work with numbers, not with words (or tags). Hence each word (and tag) is assigned a unique integer.The set of unique words and tags are computed by transforming it in a list and indexing them in a dictionary. These dictionaries are the word vocabulary and the tag vocabulary. We specially indicate the padded value and the Out-Of-Vocabulary words.

In [20]:
```
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.stem.porter import PorterStemmer

st = PorterStemmer()
words, tags = set([]), set([])

for s in train_sentences:
    for w in s:
        words.add(w.lower())

for ts in train_tags:
    for t in ts:
        tags.add(t)
```

In [21]:
```
word2index = {w: i + 2 for i, w in enumerate(list(words))}
word2index['-PAD-'] = 0
word2index['-OOV-'] = 1

tag2index = {t: i + 1 for i, t in enumerate(list(tags))}
tag2index['-PAD-'] = 0
```

In [22]:

```
train_sentences_X, test_sentences_X, train_tags_y, test_tags_y = [], [], [], []

for s in train_sentences:
    s_int = []
    for w in s:
        try:
            s_int.append(word2index[w.lower()])
        except KeyError:
            s_int.append(word2index['-OOV-'])

    train_sentences_X.append(s_int)

for s in test_sentences:
    s_int = []
    for w in s:
        try:
            s_int.append(word2index[w.lower()])
        except KeyError:
            s_int.append(word2index['-OOV-'])

    test_sentences_X.append(s_int)
```

In [23]:

```
for s in train_tags:
    train_tags_y.append([tag2index[t] for t in s])

for s in test_tags:
    test_tags_y.append([tag2index[t] for t in s])
```

In [24]:

```
print(train_sentences_X[0])
print(test_sentences_X[0])
print(train_tags_y[0])
print(test_tags_y[0])
```

```
[2875, 2082, 3591, 1957, 827, 6806, 5765, 8889, 8442, 1000, 8791, 1596, 9883, 5007, 1596, 8042, 2638, 8653, 9067, 8001, 819
8, 1120, 9883, 1756, 2009, 9265, 4186, 5865, 2875, 4158, 2818, 1202]
[2875, 4961, 2939, 1, 5172, 2939, 1, 1756, 1000, 3791, 3449, 76, 2939, 1756, 1707, 7923, 76, 6899, 8875, 1316, 2875, 1, 120
2, 7172]
[17, 42, 10, 21, 21, 21, 21, 28, 42, 2, 37, 42, 30, 37, 42, 39, 38, 40, 5, 40, 21, 21, 30, 5, 4, 17, 42, 30, 17, 35, 42, 8]
[17, 42, 13, 21, 21, 13, 10, 5, 2, 40, 30, 6, 13, 5, 4, 5, 6, 38, 27, 40, 17, 42, 8, 31]
```

PADDING:

Since Keras can only deal with fixed size sequences, all the sequences are padded with a special value (0 as the index and “-PAD-“as the corresponding word/tag) to the length of the longest sequence in the dataset is found to be 271 as calculated below.

In [25]:

```
MAX_LENGTH = len(max(train_sentences_X, key=len))
print(MAX_LENGTH)
```

```
271
```

This task of padding is accomplished using pad_sequences found in keras.

In [26]:

```
from keras.preprocessing.sequence import pad_sequences

train_sentences_X = pad_sequences(train_sentences_X, maxlen=MAX_LENGTH, padding='post')
test_sentences_X = pad_sequences(test_sentences_X, maxlen=MAX_LENGTH, padding='post')
train_tags_y = pad_sequences(train_tags_y, maxlen=MAX_LENGTH, padding='post')
test_tags_y = pad_sequences(test_tags_y, maxlen=MAX_LENGTH, padding='post')
```

In [27]:

```
print(train_sentences_X[0])
print(test_sentences_X[0])
print(train_tags_y[0])
print(test_tags_y[0])
```

```
[2875 2082 3591 1957  827 6806 5765 8889 8442 1000 8791 1596 9883 5007
 1596 8042 2638 8653 9067 8001 8198 1120 9883 1756 2009 9265 4186 5865
 2875 4158 2818 1202    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    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    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]
[2875 4961 2939    1 5172 2939    1 1756 1000 3791 3449   76 2939 1756
 1707 7923   76 6899 8875 1316 2875    1 1202 7172    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
```

NETWORK ARCHITECTURE

The POS tagger model has the following structure

- ☐ An embedding layer to compute a word vector model for the words.
- ☐ An LSTM layer with a Bidirectional modifier. The bidirectional modifier inputs to the LSTM the next values in the sequence, not just the previous.
- ☐ The return_sequences is set to True so that the LSTM outputs a sequence, not only the final value.
- ☐ After the LSTM Layer, a Dense Layer (or fully-connected layer) picks the appropriate POS tag. Since this dense layer needs to run on each element of the sequence, a TimeDistributed modifier is added.

In [29]:

```
from keras.models import Sequential
from keras.layers import Dense, LSTM, InputLayer, Bidirectional, TimeDistributed, Embedding, Activation
from tensorflow.keras.optimizers import Adam

model = Sequential()
model.add(InputLayer(input_shape=(MAX_LENGTH, )))
model.add(Embedding(len(word2index), 128))
model.add(Bidirectional(LSTM(256, return_sequences=True)))
model.add(TimeDistributed(Dense(len(tag2index))))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=Adam(0.001),
              metrics=['accuracy'])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 271, 128)          1293696
_________________________________________________________________
bidirectional (Bidirectional (None, 271, 512)          788480
_________________________________________________________________
time_distributed (TimeDistri (None, 271, 47)           24111
_________________________________________________________________
activation (Activation)      (None, 271, 47)           0
=================================================================
Total params: 2,106,287
Trainable params: 2,106,287
Non-trainable params: 0
```

There's one more thing to do before training. The sequences of tags are transformed to sequences of One-Hot Encoded tags. This is what the Dense Layer outputs. Here's a function that does that:

```
In [30]:  def to_categorical(sequences, categories):
              cat_sequences = []
              for s in sequences:
                  cats = []
                  for item in s:
                      cats.append(np.zeros(categories))
                      cats[-1][item] = 1.0
                  cat_sequences.append(cats)
              return np.array(cat_sequences)
```

```
In [31]:  cat_train_tags_y = to_categorical(train_tags_y, len(tag2index))
          print(cat_train_tags_y[0])

          [[0. 0. 0. ... 0. 0. 0.]
           [0. 0. 0. ... 0. 0. 0.]
           [0. 0. 0. ... 0. 0. 0.]
           ...
           [1. 0. 0. ... 0. 0. 0.]
           [1. 0. 0. ... 0. 0. 0.]
           [1. 0. 0. ... 0. 0. 0.]]
```

The model training is carried out for 40 epochs with batch size of 128. The training loss, accuracy and the validation loss and accuracy are shown in the below snapshots.

```
In [*]:  history = model.fit(train_sentences_X, to_categorical(train_tags_y, len(tag2index)), batch_size=128, epochs=40, validation_sp

         Train on 2504 samples, validate on 627 samples
         Epoch 1/40
         2504/2504 [==============================] - ETA: 0s - loss: 1.2323 - accuracy: 0.8590

         C:\Users\Shakthi V\AppData\Roaming\Python\Python38\site-packages\keras\engine\training.py:2470: UserWarning: `Model.state_up
         dates` will be removed in a future version. This property should not be used in TensorFlow 2.0, as `updates` are applied aut
         omatically.
           warnings.warn('`Model.state_updates` will be removed in a future version. '

         2504/2504 [==============================] - 107s 43ms/sample - loss: 1.2323 - accuracy: 0.8590 - val_loss: 0.3698 - val_acc
         uracy: 0.9070
         Epoch 2/40
         2504/2504 [==============================] - 140s 56ms/sample - loss: 0.3338 - accuracy: 0.9063 - val_loss: 0.3238 - val_acc
         uracy: 0.9040
         Epoch 3/40
         2504/2504 [==============================] - 260s 104ms/sample - loss: 0.3149 - accuracy: 0.9085 - val_loss: 0.3121 - val_ac
         curacy: 0.9156
         Epoch 4/40
         2504/2504 [==============================] - 336s 134ms/sample - loss: 0.3048 - accuracy: 0.9167 - val_loss: 0.3038 - val_ac
         curacy: 0.9165
         Epoch 5/40
         2504/2504 [==============================] - 350s 140ms/sample - loss: 0.2960 - accuracy: 0.9171 - val_loss: 0.2952 - val_ac
         curacy: 0.9164
         Epoch 6/40
         2504/2504 [==============================] - 363s 145ms/sample - loss: 0.2877 - accuracy: 0.9171 - val_loss: 0.2881 - val_ac
         curacy: 0.9168
```

```
Epoch 7/40
2504/2504 [==============================] - 361s 144ms/sample - loss: 0.2812 - accuracy: 0.9182 - val_loss: 0.2832 - val_ac
curacy: 0.9191
Epoch 8/40
2504/2504 [==============================] - 348s 139ms/sample - loss: 0.2765 - accuracy: 0.9206 - val_loss: 0.2785 - val_ac
curacy: 0.9243
Epoch 9/40
2504/2504 [==============================] - 370s 148ms/sample - loss: 0.2718 - accuracy: 0.9240 - val_loss: 0.2742 - val_ac
curacy: 0.9242
Epoch 10/40
2504/2504 [==============================] - 375s 150ms/sample - loss: 0.2665 - accuracy: 0.9293 - val_loss: 0.2662 - val_ac
curacy: 0.9314
Epoch 11/40
2504/2504 [==============================] - 367s 146ms/sample - loss: 0.2589 - accuracy: 0.9358 - val_loss: 0.2568 - val_ac
curacy: 0.9376
Epoch 12/40
2504/2504 [==============================] - 375s 150ms/sample - loss: 0.2476 - accuracy: 0.9405 - val_loss: 0.2441 - val_ac
curacy: 0.9446
Epoch 13/40
2504/2504 [==============================] - 369s 147ms/sample - loss: 0.2320 - accuracy: 0.9445 - val_loss: 0.2258 - val_ac
curacy: 0.9443
Epoch 14/40
2504/2504 [==============================] - 376s 150ms/sample - loss: 0.2130 - accuracy: 0.9479 - val_loss: 0.2054 - val_ac
curacy: 0.9505
Epoch 15/40
2504/2504 [==============================] - 384s 153ms/sample - loss: 0.1925 - accuracy: 0.9523 - val_loss: 0.1846 - val_ac
curacy: 0.9537
Epoch 16/40
2504/2504 [==============================] - 389s 155ms/sample - loss: 0.1706 - accuracy: 0.9560 - val_loss: 0.1635 - val_ac
curacy: 0.9570
Epoch 17/40
2504/2504 [==============================] - 384s 153ms/sample - loss: 0.1488 - accuracy: 0.9610 - val_loss: 0.1435 - val_ac
curacy: 0.9625
Epoch 18/40
2504/2504 [==============================] - 359s 144ms/sample - loss: 0.1278 - accuracy: 0.9675 - val_loss: 0.1242 - val_ac
curacy: 0.9682
Epoch 19/40
2504/2504 [==============================] - 269s 107ms/sample - loss: 0.1081 - accuracy: 0.9736 - val_loss: 0.1074 - val_ac
curacy: 0.9726
Epoch 20/40
2504/2504 [==============================] - 330s 132ms/sample - loss: 0.0910 - accuracy: 0.9787 - val_loss: 0.0934 - val_ac
curacy: 0.9766
Epoch 21/40
2504/2504 [==============================] - 166s 66ms/sample - loss: 0.0763 - accuracy: 0.9831 - val_loss: 0.0813 - val_acc
uracy: 0.9808
Epoch 22/40
2504/2504 [==============================] - 275s 110ms/sample - loss: 0.0638 - accuracy: 0.9867 - val_loss: 0.0720 - val_ac
curacy: 0.9836
Epoch 23/40
2504/2504 [==============================] - 394s 157ms/sample - loss: 0.0532 - accuracy: 0.9894 - val_loss: 0.0632 - val_ac
curacy: 0.9852

Epoch 24/40
2504/2504 [==============================] - 297s 119ms/sample - loss: 0.0442 - accuracy: 0.9913 - val_loss: 0.0566 - val_ac
curacy: 0.9868
Epoch 25/40
2504/2504 [==============================] - 170s 68ms/sample - loss: 0.0370 - accuracy: 0.9927 - val_loss: 0.0510 - val_acc
uracy: 0.9881
Epoch 26/40
2504/2504 [==============================] - 188s 75ms/sample - loss: 0.0312 - accuracy: 0.9938 - val_loss: 0.0469 - val_acc
uracy: 0.9888
Epoch 27/40
2504/2504 [==============================] - 179s 71ms/sample - loss: 0.0266 - accuracy: 0.9947 - val_loss: 0.0440 - val_acc
uracy: 0.9892
Epoch 28/40
2504/2504 [==============================] - 182s 73ms/sample - loss: 0.0230 - accuracy: 0.9954 - val_loss: 0.0411 - val_acc
uracy: 0.9899
Epoch 29/40
2504/2504 [==============================] - 191s 76ms/sample - loss: 0.0201 - accuracy: 0.9959 - val_loss: 0.0393 - val_acc
uracy: 0.9903
Epoch 30/40
2504/2504 [==============================] - 199s 80ms/sample - loss: 0.0178 - accuracy: 0.9963 - val_loss: 0.0382 - val_acc
uracy: 0.9906
Epoch 31/40
2504/2504 [==============================] - 203s 81ms/sample - loss: 0.0158 - accuracy: 0.9968 - val_loss: 0.0363 - val_acc
uracy: 0.9908
```

```
Epoch 32/40
2504/2504 [==============================] - 213s 85ms/sample - loss: 0.0141 - accuracy: 0.9971 - val_loss: 0.0357 - val_
accuracy: 0.9910
Epoch 33/40
2504/2504 [==============================] - 200s 80ms/sample - loss: 0.0128 - accuracy: 0.9974 - val_loss: 0.0348 - val_
accuracy: 0.9910
Epoch 34/40
2504/2504 [==============================] - 186s 74ms/sample - loss: 0.0117 - accuracy: 0.9976 - val_loss: 0.0341 - val_
accuracy: 0.9913
Epoch 35/40
2504/2504 [==============================] - 207s 83ms/sample - loss: 0.0107 - accuracy: 0.9978 - val_loss: 0.0341 - val_
accuracy: 0.9914
Epoch 36/40
2504/2504 [==============================] - 211s 84ms/sample - loss: 0.0098 - accuracy: 0.9980 - val_loss: 0.0333 - val_
accuracy: 0.9914
Epoch 37/40
2504/2504 [==============================] - 204s 81ms/sample - loss: 0.0091 - accuracy: 0.9981 - val_loss: 0.0331 - val_
accuracy: 0.9914

Epoch 38/40
2504/2504 [==============================] - 216s 86ms/sample - loss: 0.0085 - accuracy: 0.9982 - val_loss: 0.0333 - val_
accuracy: 0.9913
Epoch 39/40
2504/2504 [==============================] - 209s 83ms/sample - loss: 0.0079 - accuracy: 0.9984 - val_loss: 0.0326 - val_
accuracy: 0.9915
Epoch 40/40
2504/2504 [==============================] - 223s 89ms/sample - loss: 0.0073 - accuracy: 0.9985 - val_loss: 0.0329 - val_
accuracy: 0.9916
```

Now we evaluate the test set to find the metrics obtained. It is observed that we obtain an accuracy of 99.1521% in test set.

In [35]:

```
scores = model.evaluate(test_sentences_X, to_categorical(test_tags_y, len(tag2index)))
print(f"{model.metrics_names[1]}: {scores[1] * 100}")
print(model.metrics_names)
```

```
accuracy: 99.15218949317932
['loss', 'accuracy']
```

The model is now fed with new sentences and the predictions are made. The sentences provided as input and the corresponding outputs are as follows:

```
from nltk.tokenize import sent_tokenize, word_tokenize
input_text = "Will he cheat hari in the park? I shall call the police here. The culprit has been caught! He must surrender."
test_samples = sent_tokenize(input_text)
```

In [38]:
```python
test_samples_X = []
for s in test_samples:
    s_int = []
    for w in s:
        try:
            s_int.append(word2index[w.lower()])
        except KeyError:
            s_int.append(word2index['-OOV-'])
    test_samples_X.append(s_int)

test_samples_X = pad_sequences(test_samples_X, maxlen=MAX_LENGTH, padding='post')
print(test_samples_X)
```

```
[[5581 1081 6700    1 4379 1137    1 6238 6256  799 7658 8039 3138 6169
  1202    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
     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    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    0    0
     0    0    0    0    0]
 [2875 7940 2622 5527  906 6767    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
```

In [39]:
```python
predictions = model.predict(test_samples_X)
```

In [40]:
```python
def logits_to_tokens(sequences, index):
    token_sequences = []
    for categorical_sequence in sequences:
        token_sequence = []
        for categorical in categorical_sequence:
            if index[np.argmax(categorical)] == '-PAD-':
                break
            token_sequence.append(index[np.argmax(categorical)])

        token_sequences.append(token_sequence)

    return token_sequences
```

```python
print(logits_to_tokens(predictions, {i: t for t, i in tag2index.items()}))
```

```
[['MD', 'PRP', 'VBG', 'NN', 'IN', 'DT', 'NN', '.'], ['PRP', 'MD', 'VB', 'DT', 'NN', 'RB', '.'], ['DT', 'NN', 'VBZ', 'VBN', 'VBN', '.'],
['PRP', 'MD', 'VBG', '.']]
```

SAMPLE INPUT AND OUTPUT PRODUCED:

Will	**he**	**cheat**	**hari**	**in**	**the**	**park**	**?**
‘MD’	**‘PRP’**	**‘VBG’**	**‘NN’**	**‘IN’**	**‘DT’**	**‘NN’**	**‘.’**

I	**shall**	**call**	**the**	**police**	**here**	**.**
‘PRP’	**‘MD’**	**‘VB’**	**‘DT’**	**‘NN’**	**‘RB’**	**‘.’**

The	**culprit**	**has**	**been**	**caught**	**!**
‘DT’	**‘NN’**	**‘VBZ’**	**‘VBN’**	**‘VBN’**	**‘.’**

He	**must**	**surrender**	**.**
‘PRP’	**‘MD’**	**‘VBG’**	**‘.’**

RESULTS:

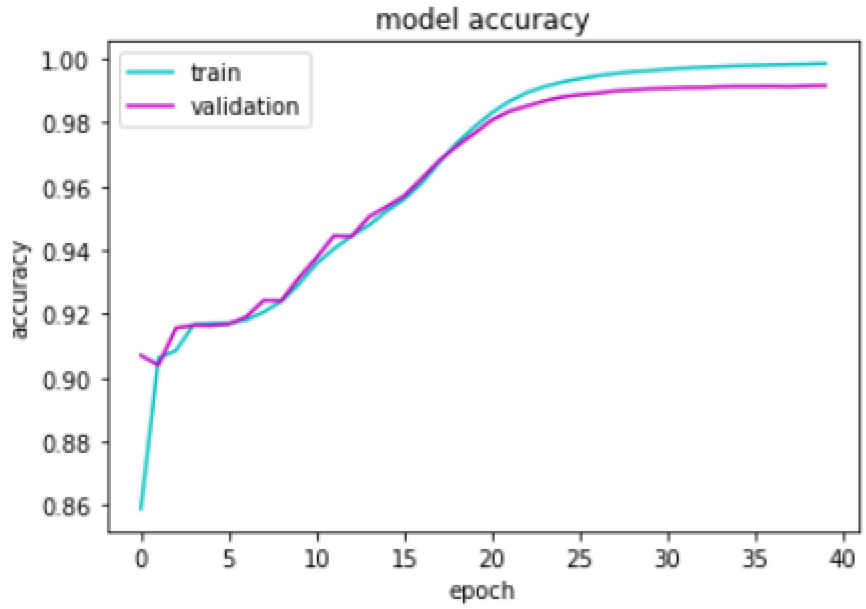
The loss values and the accuracy values are plotted in a graph to enable better visualization. It is observed from the graph that the generalization gap reduces as training proceeds and there isn't overfitting.

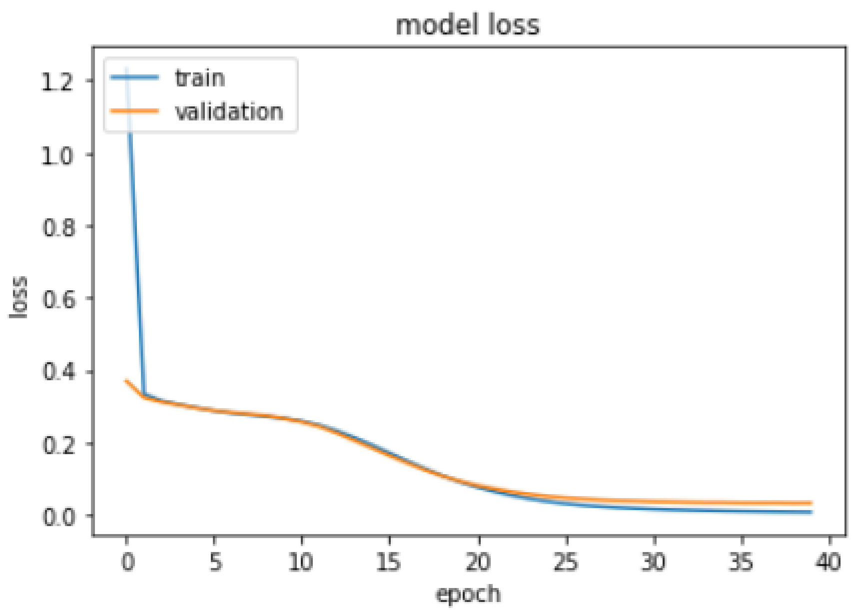
```
In [33]:
import matplotlib.pyplot as plt
print(history.history.keys())
plt.axes().set(facecolor ="white")
plt.plot(history.history['accuracy'],color='c')
plt.plot(history.history['val_accuracy'],color='m')
plt.title('model accuracy').set_color('black')
plt.ylabel('accuracy').set_color('black')
plt.xlabel('epoch').set_color('black')
plt.legend(['train', 'validation'], loc='upper left')

plt.show()

plt.axes().set(facecolor ="white")
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['train', 'validation'], loc='upper left')
plt.title('model loss').set_color('black')
plt.ylabel('loss').set_color('black')
plt.xlabel('epoch').set_color('black')
plt.show()

dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```





The noted values for the accuracy is as follows.

Training	99.85
Validation	99.16
Testing	99.15

INFERENCE:

As seen from the output screenshots, the model performs well in the dataset. It managed to achieve a training accuracy of 99.85 % at the end of the 40th epoch. The validation accuracy improved over the training to record 99.16% by the 40th epoch. On testing 99.15% accuracy is achieved.

REFERENCES USED:

https://towardsdatascience.com/part-of-speech-tagging-for-beginners-3a0754b2ebba
https://nlpforhackers.io/lstm-pos-tagger-keras/
https://medium.com/analytics-vidhya/pos-tagging-using-conditional-random-fields-92077e5eaa31