CS6030 - NATURAL LANGUAGE PROCESSING
ASSIGNMENT - DEVELOPING A POS TAGGER IN NLTK

- **GOKUL. S**
2018103026

Github Page: https://github.com/gokul-sunilkumar/NLP-Projects

PROBLEM AT HAND:

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. Here, 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 DESCRIPTION:

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

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

TEXT PRE-PROCESSING

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. The most common algorithm for stemming English, and one that has repeatedly been shown to be empirically very effective, 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.

apples → apple
flying → fly

SPLIT 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

CODE SNAPSHOTS

MODULE I – EXPLORING AND VISUALISING DATASET

Importing Packages

'nltk' (Natural Language ToolKit) is the library package that contains the corpus for stop words and for lemmatization.

```
1 import nltk
2 nltk.download('all')
```

```
[nltk_data] Downloading collection 'all'
[nltk_data]    |
[nltk_data]    | Downloading package abc to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/abc.zip.
[nltk_data]    | Downloading package alpino to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/alpino.zip.
[nltk_data]    | Downloading package biocreative_ppi to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping corpora/biocreative_ppi.zip.
[nltk_data]    | Downloading package brown to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/brown.zip.
[nltk_data]    | Downloading package brown_tei to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/brown_tei.zip.
```

OBTAINING THE DATASET

The Penn-TreeBank dataset is available in the nltk package. It is downloaded in-order to start the preprocessing required to start the model training. Further the data is restructured to separate the words from the tags.

Downloading Dataset

```
[2]  1 import nltk
     2 tagged_sentences = nltk.corpus.treebank.tagged_sents()
```

```
[3]  1 print(tagged_sentences[3])
     2 print("Tagged sentences: ", len(tagged_sentences))
     3 print("Tagged words:", len(nltk.corpus.treebank.tagged_words()))
```

```
[('A', 'DT'), ('form', 'NN'), ('of', 'IN'), ('asbestos', 'NN'), ('once', 'RB'),
Tagged sentences:  3914
Tagged words: 100676
```

```
[4]  1 import numpy as np
     2 sentences, sentence_tags =[], []
     3 for tagged_sentence in tagged_sentences:
     4     sentence, tags = zip(*tagged_sentence)
     5     sentences.append(np.array(sentence))
     6     sentence_tags.append(np.array(tags))
```

```
[6]  1 print(sentences[30])
     2 print(sentence_tags[30])
```

```
['``' 'There' "'s" 'no' 'question' 'that' 'some' 'of' 'those' 'workers'
 'and' 'managers' 'contracted' 'asbestos-related' 'diseases' ',' "''"
 'said' '*T*-1' 'Darrell' 'Phillips' ',' 'vice' 'president' 'of' 'human'
 'resources' 'for' 'Hollingsworth' '&' 'Vose' '.']
['``' 'EX' 'VBZ' 'DT' 'NN' 'IN' 'DT' 'IN' 'DT' 'NNS' 'CC' 'NNS' 'VBD' 'JJ'
 'NNS' ',' "''" 'VBD' '-NONE-' 'NNP' 'NNP' ',' 'NN' 'NN' 'IN' 'JJ' 'NNS'
 'IN' 'NNP' 'CC' 'NNP' '.']
```

Before training the model, the data is split into training and testing data using the train_test_split function from Scikit-Learn. Since Keras works with numbers, 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.

MODEL CONFIGURATION

```
[25]  1 from sklearn.model_selection import train_test_split
      2 (train_sentences,  test_sentences, train_tags, test_tags) = train_test_split(sentences, sentence_tags, test_size=0.2)
```

```
[26]  1 from nltk.tokenize import sent_tokenize, word_tokenize
      2 from nltk.stem.porter import PorterStemmer
      3
      4 st = PorterStemmer()
      5 words, tags = set([]), set([])
      6
      7 for s in train_sentences:
      8     for w in s:
      9         words.add(w.lower())
     10
     11 for ts in train_tags:
     12     for t in ts:
     13         tags.add(t)
```

```
[27]  1 word2index = {w: i + 2 for i, w in enumerate(list(words))}
      2 word2index['-PAD-'] = 0
      3 word2index['-OOV-'] = 1
      4
      5 tag2index = {t: i + 1 for i, t in enumerate(list(tags))}
      6 tag2index['-PAD-'] = 0
```

```
[28]  1 train_sentences_X, test_sentences_X, train_tags_y, test_tags_y = [], [], [], []
      2
      3 for s in train_sentences:
      4     s_int = []
      5     for w in s:
      6         try:
      7             s_int.append(word2index[w.lower()])
      8         except KeyError:
      9             s_int.append(word2index['-OOV-'])
     10
     11     train_sentences_X.append(s_int)
```

```
[29]  1 for s in test_sentences:
      2     s_int = []
      3     for w in s:
      4         try:
      5             s_int.append(word2index[w.lower()])
      6         except KeyError:
      7             s_int.append(word2index['-OOV-'])
      8
      9     test_sentences_X.append(s_int)
```

```
[30]  1 for s in train_tags:
      2     train_tags_y.append([tag2index[t] for t in s])
      3
      4 for s in test_tags:
      5     test_tags_y.append([tag2index[t] for t in s])
```

```
[31]  1 print(train_sentences_X[0])
      2 print(test_sentences_X[0])
      3 print(train_tags_y[0])
      4 print(test_tags_y[0])
```

```
[4120, 8229, 3806, 4208, 5951, 7441, 5532, 3697, 357, 5984, 
[3829, 4137, 4128, 5532, 6786, 5532, 977, 2932, 10070, 6939, 
[39, 6, 33, 38, 45, 43, 16, 33, 38, 6, 43, 45, 26, 39, 26, 2
[4, 4, 4, 16, 27, 16, 45, 26, 2, 43, 43, 43, 6, 33, 43, 23]
```

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 which is found to be 271 as calculated below.

```
[32]  1 MAX_LENGTH = len(max(train_sentences_X, key=len))
      2 print(MAX_LENGTH)
```

```
271
```

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

```
[33]  1 from keras.preprocessing.sequence import pad_sequences
      2
      3 train_sentences_X = pad_sequences(train_sentences_X, maxlen=MAX_LENGTH, padding='post')
      4 test_sentences_X = pad_sequences(test_sentences_X, maxlen=MAX_LENGTH, padding='post')
      5 train_tags_y = pad_sequences(train_tags_y, maxlen=MAX_LENGTH, padding='post')
      6 test_tags_y = pad_sequences(test_tags_y, maxlen=MAX_LENGTH, padding='post')
```

```
1 print(train_sentences_X[0])
2 print(test_sentences_X[0])
3 print(train_tags_y[0])
4 print(test_tags_y[0])
```

```
[4120 8229 3806 4208 5951 7441 5532 3697  357 5984  699 3529 6500  975
 2565 5738 3580 3697 3117 3661 1427 3426 1562 5984 5992  145 9582 5532
 4425 4970 5036 4240 2403    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0]
[ 3829  4137  4128  5532  6786  5532   977  2932 10070  6939  8220  2885
  5984  3697  3082  2403     0     0     0     0     0     0     0     0
```

NETWORK ARCHITECTURE

The model to be trained as a POS Tagger 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=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.

```
[35]
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)          1294208
_________________________________________________________________
bidirectional (Bidirectional (None, 271, 512)          788480
_________________________________________________________________
time_distributed (TimeDistri (None, 271, 47)           24111
_________________________________________________________________
activation (Activation)      (None, 271, 47)           0
=================================================================
Total params: 2,106,799
Trainable params: 2,106,799
Non-trainable params: 0
_________________________________________________________________
```

Before Training the sequence of tags are transformed into a sequence of One-Hot Encoded tags, which is output by the Dense Layer.

```
[36]
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)
```

```
[37]
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. The training loss, accuracy and the validation loss and accuracy are noted in the variable history.

In [18]:
```
history = model.fit(train_sentences_X, to_categorical(train_tags_y, len(tag2index)), batch_size=128, epochs=40, validation_split
```

```
Epoch 1/40
20/20 [==============================] - 155s 8s/step - loss: 1.1935 - accuracy: 0.8585 - val_loss: 0.3678 - val_accuracy: 0.
9096
Epoch 2/40
20/20 [==============================] - 227s 11s/step - loss: 0.3359 - accuracy: 0.9057 - val_loss: 0.3167 - val_accuracy:
0.9057
Epoch 3/40
20/20 [==============================] - 234s 12s/step - loss: 0.3143 - accuracy: 0.9113 - val_loss: 0.3042 - val_accuracy:
0.9175
Epoch 4/40
20/20 [==============================] - 238s 12s/step - loss: 0.3035 - accuracy: 0.9166 - val_loss: 0.2960 - val_accuracy:
0.9179
Epoch 5/40
20/20 [==============================] - 241s 12s/step - loss: 0.2958 - accuracy: 0.9168 - val_loss: 0.2902 - val_accuracy:
0.9180
Epoch 6/40
20/20 [==============================] - 247s 12s/step - loss: 0.2892 - accuracy: 0.9169 - val_loss: 0.2813 - val_accuracy:
0.9181
Epoch 7/40
Epoch 7/40
20/20 [==============================] - 250s 12s/step - loss: 0.2817 - accuracy: 0.9184 - val_loss: 0.2751 - val_accuracy:
0.9215
Epoch 8/40
20/20 [==============================] - 250s 12s/step - loss: 0.2763 - accuracy: 0.9208 - val_loss: 0.2707 - val_accuracy:
0.9227
Epoch 9/40
20/20 [==============================] - 260s 13s/step - loss: 0.2719 - accuracy: 0.9226 - val_loss: 0.2666 - val_accuracy:
0.9248
Epoch 10/40
20/20 [==============================] - 249s 12s/step - loss: 0.2671 - accuracy: 0.9254 - val_loss: 0.2615 - val_accuracy:
0.9287
Epoch 11/40
20/20 [==============================] - 250s 13s/step - loss: 0.2603 - accuracy: 0.9292 - val_loss: 0.2539 - val_accuracy:
0.9308
Epoch 12/40
20/20 [==============================] - 249s 12s/step - loss: 0.2491 - accuracy: 0.9362 - val_loss: 0.2412 - val_accuracy:
0.9415
Epoch 13/40
20/20 [==============================] - 247s 12s/step - loss: 0.2314 - accuracy: 0.9446 - val_loss: 0.2201 - val_accuracy:
0.9460
Epoch 14/40
20/20 [==============================] - 247s 12s/step - loss: 0.2053 - accuracy: 0.9503 - val_loss: 0.1917 - val_accuracy:
0.9514
Epoch 15/40
20/20 [==============================] - 270s 14s/step - loss: 0.1744 - accuracy: 0.9551 - val_loss: 0.1624 - val_accuracy:
0.9584
Epoch 16/40
20/20 [==============================] - 286s 14s/step - loss: 0.1448 - accuracy: 0.9626 - val_loss: 0.1361 - val_accuracy:
0.9660
Epoch 17/40
20/20 [==============================] - 293s 15s/step - loss: 0.1189 - accuracy: 0.9709 - val_loss: 0.1152 - val_accuracy:
0.9709
Epoch 18/40
20/20 [==============================] - 305s 15s/step - loss: 0.0974 - accuracy: 0.9771 - val_loss: 0.0979 - val_accuracy:
0.9768
```

```
Epoch 19/40
20/20 [==============================] - 303s 15s/step - loss: 0.0801 - accuracy: 0.9827 - val_loss: 0.0846 - val_accuracy:
0.9803
Epoch 20/40
20/20 [==============================] - 312s 16s/step - loss: 0.0660 - accuracy: 0.9867 - val_loss: 0.0739 - val_accuracy:
0.9835
Epoch 21/40
20/20 [==============================] - 309s 15s/step - loss: 0.0546 - accuracy: 0.9894 - val_loss: 0.0657 - val_accuracy:
0.9853
Epoch 22/40
20/20 [==============================] - 301s 15s/step - loss: 0.0455 - accuracy: 0.9914 - val_loss: 0.0594 - val_accuracy:
0.9868
Epoch 23/40
20/20 [==============================] - 291s 15s/step - loss: 0.0382 - accuracy: 0.9928 - val_loss: 0.0544 - val_accuracy:
0.9878
Epoch 24/40
20/20 [==============================] - 294s 15s/step - loss: 0.0323 - accuracy: 0.9939 - val_loss: 0.0512 - val_accuracy:
0.9883
Epoch 25/40
20/20 [==============================] - 290s 15s/step - loss: 0.0276 - accuracy: 0.9947 - val_loss: 0.0475 - val_accuracy:
0.9893
Epoch 26/40
20/20 [==============================] - 266s 13s/step - loss: 0.0239 - accuracy: 0.9954 - val_loss: 0.0447 - val_accuracy:
0.9896
Epoch 27/40
20/20 [==============================] - 265s 13s/step - loss: 0.0207 - accuracy: 0.9960 - val_loss: 0.0430 - val_accuracy:
0.9899
Epoch 28/40
20/20 [==============================] - 260s 13s/step - loss: 0.0181 - accuracy: 0.9964 - val_loss: 0.0415 - val_accuracy:
0.9902
Epoch 29/40
20/20 [==============================] - 259s 13s/step - loss: 0.0161 - accuracy: 0.9968 - val_loss: 0.0407 - val_accuracy:
0.9903
Epoch 30/40
20/20 [==============================] - 259s 13s/step - loss: 0.0143 - accuracy: 0.9971 - val_loss: 0.0393 - val_accuracy:
0.9907
Epoch 31/40
20/20 [==============================] - 264s 13s/step - loss: 0.0130 - accuracy: 0.9974 - val_loss: 0.0384 - val_accuracy:
0.9908
Epoch 32/40
20/20 [==============================] - 269s 13s/step - loss: 0.0117 - accuracy: 0.9976 - val_loss: 0.0380 - val_accuracy:
0.9908
Epoch 33/40
20/20 [==============================] - 279s 14s/step - loss: 0.0106 - accuracy: 0.9979 - val_loss: 0.0377 - val_accuracy:
0.9909
Epoch 34/40
20/20 [==============================] - 260s 13s/step - loss: 0.0097 - accuracy: 0.9980 - val_loss: 0.0371 - val_accuracy:
0.9910
Epoch 35/40
20/20 [==============================] - 251s 12s/step - loss: 0.0090 - accuracy: 0.9982 - val_loss: 0.0369 - val_accuracy:
0.9911
Epoch 36/40
20/20 [==============================] - 249s 12s/step - loss: 0.0083 - accuracy: 0.9983 - val_loss: 0.0373 - val_accuracy:
0.9910
Epoch 37/40
20/20 [==============================] - 261s 13s/step - loss: 0.0077 - accuracy: 0.9984 - val_loss: 0.0375 - val_accuracy:
0.9910
Epoch 38/40
20/20 [==============================] - 267s 13s/step - loss: 0.0072 - accuracy: 0.9985 - val_loss: 0.0369 - val_accuracy:
0.9911
Epoch 39/40
20/20 [==============================] - 258s 13s/step - loss: 0.0067 - accuracy: 0.9987 - val_loss: 0.0363 - val_accuracy:
0.9913
Epoch 40/40
20/20 [==============================] - 259s 13s/step - loss: 0.0063 - accuracy: 0.9988 - val_loss: 0.0364 - val_accuracy:
0.9912
```

Now we evaluate the test set to find the metrics obtained. It is observed that we obtain an accuracy of 99.13 % in test set having a loss of 0.0355

```
predictions = model.predict(test_sentences_X)
```

```
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)
```

```
25/25 [==============================] - 17s 670ms/step - loss: 0.0355 - accuracy: 0.9913
accuracy: 99.13380742073059
['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:

```
[42]  1 from nltk.tokenize import sent_tokenize, word_tokenize
      2 input_text = "Will he cheat hari in the park? I shall call the police here. The culprit has been caught! He must surrender."
      3 test_samples = sent_tokenize(input_text)
```

```
[43]  1 test_samples = [word_tokenize(x) for x in test_samples]
```

```
[44]  1 test_samples_X = []
      2 for s in test_samples:
      3     s_int = []
      4     for w in s:
      5         try:
      6             s_int.append(word2index[w.lower()])
      7         except KeyError:
      8             s_int.append(word2index['-OOV-'])
      9     test_samples_X.append(s_int)
     10
     11 test_samples_X = pad_sequences(test_samples_X, maxlen=MAX_LENGTH, padding='post')
     12 print(test_samples_X)
```

```
[[4032 2406 1025 ...    0    0    0]
 [3267 2399 3242 ...    0    0    0]
 [3697    1 1767 ...    0    0    0]
 [2406 5918 3838 ...    0    0    0]]
```

```
[45]  1 predictions = model.predict(test_samples_X)
      2 #print(predictions, predictions.shape)
```

```
[46]  1 def logits_to_tokens(sequences, index):
      2     token_sequences = []
      3     for categorical_sequence in sequences:
      4         token_sequence = []
      5         for categorical in categorical_sequence:
      6             if index[np.argmax(categorical)] == '-PAD-':
      7                 break
      8             token_sequence.append(index[np.argmax(categorical)])
      9
     10         token_sequences.append(token_sequence)
     11
     12     return token_sequences
```

```
[47]  1 print(logits_to_tokens(predictions, {i: t for t, i in tag2index.items()}))
```

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

```
[48]  1 "Will he cheat hari in the park? I shall call the police here. The culprit has been caught! He must surrender."
```

```
'Will he cheat hari in the park? I shall call the police here. The culprit has been caught! He must surrender.'
```

SAMPLE INPUT AND OUTPUT PRODUCED:

Will	he	cheat	hari	in	the	park	?
'MD'	'PRP'	'VB'	'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'	'.'

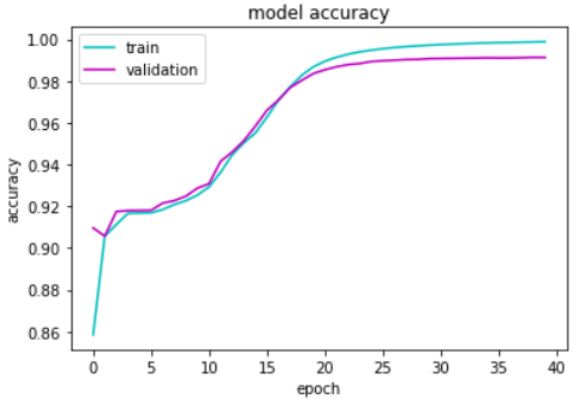
He	must	surrender	.
'PRP'	'MD'	'VB'	'.'

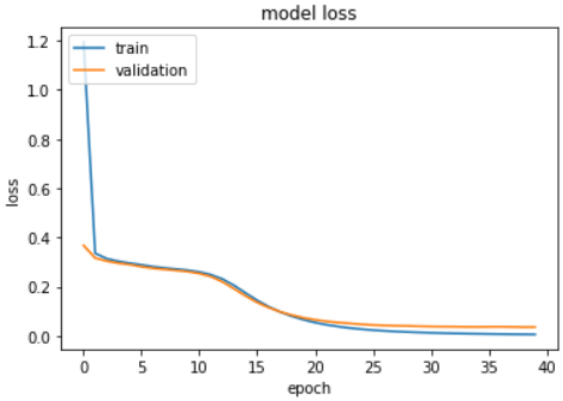
RESULTS

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.

```
[39]  1 import matplotlib.pyplot as plt
      2 print(history.history.keys())
      3 plt.axes().set(facecolor ="white")
      4 plt.plot(history.history['accuracy'],color='c')
      5 plt.plot(history.history['val_accuracy'],color='m')
      6 plt.title('model accuracy').set_color('black')
      7 plt.ylabel('accuracy').set_color('black')
      8 plt.xlabel('epoch').set_color('black')
      9 plt.legend(['train', 'validation'], loc='upper left')
     10
     11 plt.show()
     12
     13 plt.axes().set(facecolor ="white")
     14 plt.plot(history.history['loss'])
     15 plt.plot(history.history['val_loss'])
     16 plt.legend(['train', 'validation'], loc='upper left')
     17 plt.title('model loss').set_color('black')
     18 plt.ylabel('loss').set_color('black')
     19 plt.xlabel('epoch').set_color('black')
     20 plt.show()
```

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





The noted values for the accuracy and loss are as follows:

	Accuracy	Loss
Training	99.88%	0.0063
Validation	99.12%	0.0364
Testing	99.13%	0.0315

CONCLUSION

As seen from the output screenshots, the model performs well in the dataset. It managed to achieve a training accuracy of 99.88 % at the end of the 40th epoch. The reported validation accuracy improved over the training to record 99.12% by the 40th epoch. On testing 99.13% accuracy is achieved.

REFERENCES

https://nlpforhackers.io/lstm-pos-tagger-keras/

https://towardsdatascience.com/part-of-speech-tagging-for-beginners-3a0754b2ebba