

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

NLP LAB -II

[POS tagging using RNN and HMM]

DATE: 15/04/2023 TIME: 02 Hours

The purpose of lab sheet-2 is learning the concepts of POS tagging using RNN and HMM. We will learn the process of classifying words into their **parts of speech, which is known as POS-tagging**.

POS-tagging using RNN

The NLTK library has corpora that contain words and their POS tags. We will be using the POS tagged corpora i.e **treebank**, **conll2000**, and **brown** from NLTK to demonstrate the key concepts.

The following table provides information about some of the major tags:

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

First section contains the following steps:

1. Pre-process data
2. Word Embeddings
3. RNN
4. Model Evaluation

Importing the dataset

Let's begin with importing the necessary libraries and loading the dataset. This is a requisite step in every data analysis process. We'll be loading the data first using three well-known text corpora and taking the union of those.

```
# import necessary libraries
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.utils 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, GRU, 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

```

1. Preprocess data

As a part of preprocessing, we'll be performing various steps such as dividing data into words and tags, Vectorise X and Y, and Pad sequences.

Let's look at the data first. For each of the words below, there is a tag associated with it.

```

# Importing and Loading the data into data frame
# 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')
# Merging the dataframes to create a master df
tagged_sentences = treebank_corpus + brown_corpus + conll_corpus

# let's look at the data
tagged_sentences[7]

```

```

[('A', 'DET'),
 ('Lorillard', 'NOUN'),
 ('spokewoman', 'NOUN'),
 ('said', 'VERB'),
 (',', ','),
 (''', '''),
 ('This', 'DET'),
 ('is', 'VERB'),
 ('an', 'DET'),
 ('old', 'ADJ'),
 ('story', 'NOUN'),
 ('.', '.')]

```

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 corresponding 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

```
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)

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]))
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
```

```
# 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', '.']
```

```
# 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
```

The next thing we need to figure out is how are we going to feed these inputs to an RNN. If we have to give the words as input to any neural networks then we essentially have to convert them into numbers. We need to create a word embedding or one-hot vectors i.e. a vector of numbers form of each word. To start with this we'll first encode the input and output which will give a blind unique id to each word in the entire corpus for input data. On the other hand, we have the Y matrix(tags/output data). We have twelve POS tags here, treating each of them as a class and each pos tag is converted into one-hot encoding of length twelve. We'll use the `Tokenizer()` function from Keras library to encode text sequence to integer sequence.

Vectorise X and Y

```
# encode X
word_tokenizer = Tokenizer()          # instantiate tokeniser
word_tokenizer.fit_on_texts(X)         # fit tokeniser on data
# use the tokeniser to encode input sequence
X_encoded = word_tokenizer.texts_to_sequences(X)
# encode Y
tag_tokenizer = Tokenizer()
tag_tokenizer.fit_on_texts(Y)
Y_encoded = tag_tokenizer.texts_to_sequences(Y)
# 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]
```

Make sure that each sequence of input and output is of the same length.

Pad sequences

The sentences in the corpus are not of the same length. Before we feed the input in the RNN model we need to fix the length of the sentences. We cannot dynamically allocate memory required to process each sentence in the corpus as they are of different lengths. Therefore the next step after encoding the data is to **define the sequence lengths**. We need to either pad short sentences or truncate long sentences to a fixed length. This fixed length, however, is a **hyperparameter**.

```
# check length of longest sentence
```

```
lengths = [len(seq) for seq in X_encoded]
```

```
print("Length of longest sentence: {}".format(max(lengths)))
```

```
sns.boxplot(lengths)
```

```
plt.show()
```

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.

sequences greater than 100 in length will be truncated

```
MAX_SEQ_LENGTH = 100
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")
```

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
  2 7652 102 170 2 47 1898 1 269 17 6423 24231
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 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.

```
# assign padded sequences to X and Y
```

```
X, Y = X_padded, Y_padded
```

2. Word embeddings

You know that a better way (than one-hot vectors) to represent text is **word embeddings**. Currently, each word and each tag is encoded as an integer. We'll 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'll simply use one-hot encoding scheme since there are only 12 tags in the dataset.

We're using the word2vec model for no particular reason. Both of these are very efficient in representing words. You can try both and see which one works better.

The dimension of a word embedding is: (VOCABULARY_SIZE, EMBEDDING_DIMENSION)

Use word embeddings for input sequences (X)

```
# word2vec
```

```
path = '../input/wordembeddings/GoogleNews-vectors-negative300.bin'
```

```
# load word2vec using the following function present in the gensim library
```

```

word2vec = KeyedVectors.load_word2vec_format(path, binary=True)

# assign word vectors from word2vec model
# each word in word2vec model is represented using a 300 dimensional vector
EMBEDDING_SIZE = 300
VOCABULARY_SIZE = len(word_tokenizer.word_index) + 1# create an empty embedding matrix
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

# check embedding dimension

print("Embeddings shape: {}".format(embedding_weights.shape))

# let's look at an embedding of a word

embedding_weights[word_tokenizer.word_index['joy']]

```

Use one-hot encoding for output sequences (Y)

```

# use Keras' to_categorical function to one-hot encode Y
Y = to_categorical(Y)

```

```

print(Y.shape)

```

All the data preprocessing is now complete. Let's now come to the modeling part by **splitting the data to train, validation, and test sets**.

Split data in training, validation and testing sets

```

# 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)

```

```

# 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)

```

```

# 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))

```

. 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, the training data will automatically be converted to (**#samples, #timesteps, #features**) where **#features** will be the embedding dimension (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, note that you'll need to shape it to (**#samples, #timesteps, #features**) in case you don't use the Embedding() layer in Keras.

3. RNN

Next, let's build the RNN model. We're going to use word embeddings to represent the words. Now, while training the model, you can also **train the word embeddings** along with the network weights. These are often called the **embedding weights**. While training, the embedding weights will be treated as normal weights of the network which are updated in each iteration.

In the next few sections, we will try the following three RNN models:

- RNN with **arbitrarily initialized, untrainable embeddings**: In this model, we will initialize the embedding weights arbitrarily. Further, we'll **freeze the embeddings**, that won't allow the network to train them.
- RNN with **arbitrarily initialized, 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.

Uninitialized fixed embeddings

Let's start with the first experiment: a vanilla RNN with **arbitrarily initialized, untrainable embedding**. For this RNN we won't use the pre-trained word embeddings. We'll use randomly initialized embeddings. Moreover, we won't update the embeddings weights.

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

# create architecture
rnn_model = Sequential()
# create embedding layer — usually the first layer in text problems
# vocabulary size — number of unique words in data
rnn_model.add(Embedding(input_dim = VOCABULARY_SIZE,
# length of vector with which each word is represented
output_dim = EMBEDDING_SIZE,
# length of input sequence
input_length = MAX_SEQ_LENGTH,
# False — don't update the embeddings
trainable = False
# add an RNN layer which contains 64 RNN cells
# True — return whole sequence; False — return single output of the end of the sequence
rnn_model.add(SimpleRNN(64, return_sequences=True))# add time distributed (output at each
sequence) layer
rnn_model.add(TimeDistributed(Dense(NUM_CLASSES, activation='softmax'))))

#compile model
rnn_model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics = ['acc'])
# check summary of the model
rnn_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 100, 300)	17834700
simple_rnn (SimpleRNN)	(None, 100, 64)	23360
time_distributed (TimeDistri	(None, 100, 13)	845

Total params: 17,858,905
Trainable params: 24,205
Non-trainable params: 17,834,700

```
#fit model
rnn_training = rnn_model.fit(X_train, Y_train, batch_size=128, epochs=10,
validation_data=(X_validation, Y_validation))
```

```

Epoch 1/10
408/408 [=====] - 23s 56ms/step - loss: 0.4801 - acc: 0.8604 - val_loss: 0.3241 - val_acc: 0.9006
Epoch 2/10
408/408 [=====] - 22s 54ms/step - loss: 0.2717 - acc: 0.9173 - val_loss: 0.2297 - val_acc: 0.9302
Epoch 3/10
408/408 [=====] - 22s 54ms/step - loss: 0.2075 - acc: 0.9359 - val_loss: 0.1862 - val_acc: 0.9418
Epoch 4/10
408/408 [=====] - 23s 56ms/step - loss: 0.1742 - acc: 0.9448 - val_loss: 0.1612 - val_acc: 0.9484
Epoch 5/10
408/408 [=====] - 22s 54ms/step - loss: 0.1543 - acc: 0.9502 - val_loss: 0.1460 - val_acc: 0.9523
Epoch 6/10
408/408 [=====] - 22s 54ms/step - loss: 0.1424 - acc: 0.9534 - val_loss: 0.1366 - val_acc: 0.9558
Epoch 7/10
408/408 [=====] - 23s 56ms/step - loss: 0.1348 - acc: 0.9553 - val_loss: 0.1307 - val_acc: 0.9564
Epoch 8/10
408/408 [=====] - 22s 55ms/step - loss: 0.1296 - acc: 0.9567 - val_loss: 0.1266 - val_acc: 0.9575
Epoch 9/10
408/408 [=====] - 23s 56ms/step - loss: 0.1258 - acc: 0.9578 - val_loss: 0.1240 - val_acc: 0.9584
Epoch 10/10
408/408 [=====] - 23s 56ms/step - loss: 0.1230 - acc: 0.9585 - val_loss: 0.1211 - val_acc: 0.9591

```

We can see here, after ten epoch, it is giving fairly decent **accuracy of approx 95%**. Also, we are getting a healthy growth curve below.

visualise training history

```
plt.plot(rnn_training.history['acc'])
```

```
plt.plot(rnn_training.history['val_acc'])
```

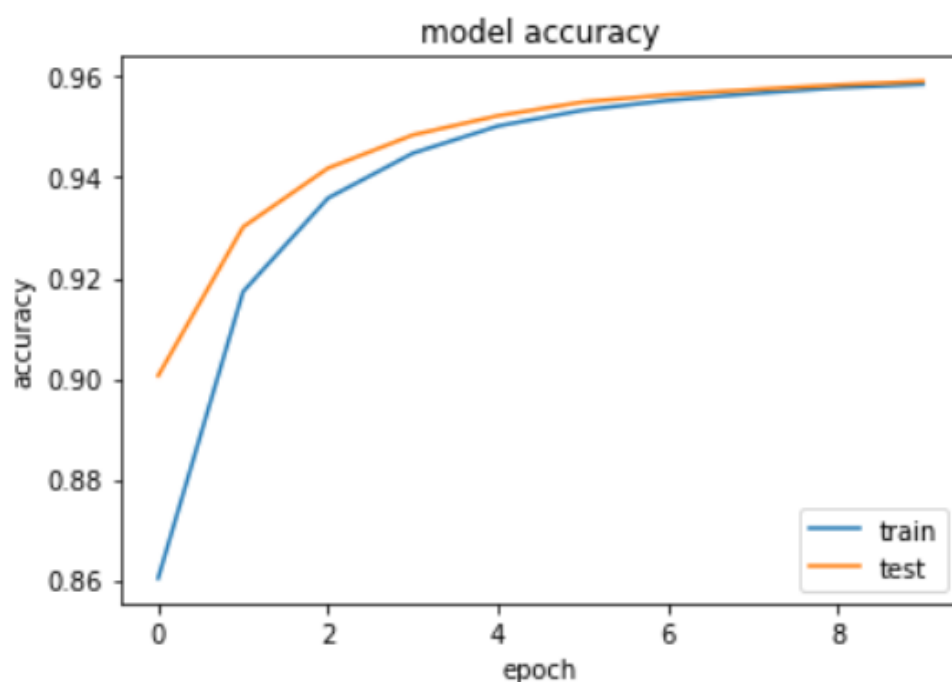
```
plt.title('model accuracy')
```

```
plt.ylabel('accuracy')
```

```
plt.xlabel('epoch')
```

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

```
plt.show()
```



Uninitialized trainable embeddings

Next, try the second model — RNN with **arbitrarily initialized, trainable embeddings**. Here, we'll allow the embeddings to get trained with the network. Change the parameter **trainable to true i.e trainable = True**. Rest all remains the same as above. On checking the model summary we can see that all the parameters have become trainable. i.e trainable params are equal to total params.

```
# 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'))))
```

Compile model

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

# check summary of the model
rnn_model.summary()
```

```
Model: "sequential_1"
```

```
-----  
Layer (type)                Output Shape              Param #  
-----  
embedding_1 (Embedding)     (None, 100, 300)         17834700  
-----  
simple_rnn_1 (SimpleRNN)     (None, 100, 64)          23360  
-----  
time_distributed_1 (TimeDist (None, 100, 13)      845  
-----  
Total params: 17,858,905  
Trainable params: 17,858,905  
Non-trainable params: 0  
-----
```

Fit model

```
rnn_training = rnn_model.fit(X_train, Y_train, batch_size=128, epochs=10,  
validation_data=(X_validation, Y_validation))
```

```
# 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()
```

Using pre-trained embedding weights

Let's now try the third experiment — RNN with **trainable word2vec embeddings**. Recall that we had loaded the word2vec embeddings in a matrix called 'embedding_weights'. Using word2vec embeddings is just as easy as including this matrix in the model architecture.

The network architecture is the same as above but instead of starting with an arbitrary embedding matrix, we'll use pre-trained embedding weights (**weights = [embedding_weights]**) coming from word2vec. The accuracy, in this case, has gone even further to **approx 99.04%**.

```
# 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'))))

    rnn_model.compile(loss      = 'categorical_crossentropy',
        optimizer = 'adam',
        metrics   = ['acc'])
    # check summary of the model
    rnn_model.summary()

```

Fit model

```

rnn_training = rnn_model.fit(X_train, Y_train, batch_size=128, epochs=10,
    validation_data=(X_validation, Y_validation))
# 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()

```

The results improved marginally in this case. 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. Pre-trained embeddings provide a real boost in many applications.

4. Model evaluation

```

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

```

```

339/339 [=====] - 3s 10ms/step - loss: 0.0310 - acc: 0.9905
Loss: 0.03096751496195793,
Accuracy: 0.9905309081077576

```

POS-tagging using HMM

Now, we will implement the POS tagging using hidden markov model and optimizing it using Viterbi algorithm. HMM (Hidden Markov Model) is a Stochastic technique for POS tagging.

1.Importing libraries

```
import nltk
import numpy as np
import pandas as pd
import random
from sklearn.model_selection import train_test_split
import pprint, time
```

download the treebank corpus from nltk

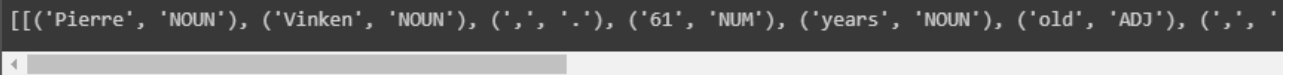
```
nltk.download('treebank')

#download the universal tagset from nltk
nltk.download('universal_tagset')

# reading the Treebank tagged sentences
nltk_data = list(nltk.corpus.treebank.tagged_sents(tagset='universal'))

#print the first two sentences along with tags
print(nltk_data[:2])
```

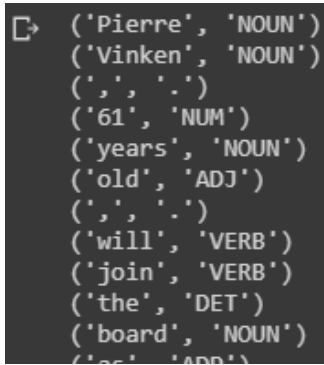
Output:



```
[(['Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years', 'NOUN'), ('old', 'ADJ'), (',', ' ')
```

```
#print each word with its respective tag for first two sentences
for sent in nltk_data[:2]:
    for tuple in sent:
        print(tuple)
```

Output:



```
(('Pierre', 'NOUN')
('Vinken', 'NOUN')
(',', '.')
('61', 'NUM')
('years', 'NOUN')
('old', 'ADJ')
(',', ' ')
('will', 'VERB')
('join', 'VERB')
('the', 'DET')
('board', 'NOUN')
('of', 'ADP')
```

Split data into training and validation set in the ratio 80:20

```
train_set, test_set =  
train_test_split(nltk_data, train_size=0.80, test_size=0.20, random_state = 101)  
# create list of train and test tagged words  
train_tagged_words = [ tup for sent in train_set for tup in sent ]  
test_tagged_words = [ tup for sent in test_set for tup in sent ]  
print(len(train_tagged_words))  
print(len(test_tagged_words))
```

Output:

```
80310  
20366
```

```
# check some of the tagged words.  
train_tagged_words[:5]
```

Output:

```
[('Drink', 'NOUN'),  
 ('Carrier', 'NOUN'),  
 ('Competes', 'VERB'),  
 ('With', 'ADP'),  
 ('Cartons', 'NOUN')]
```

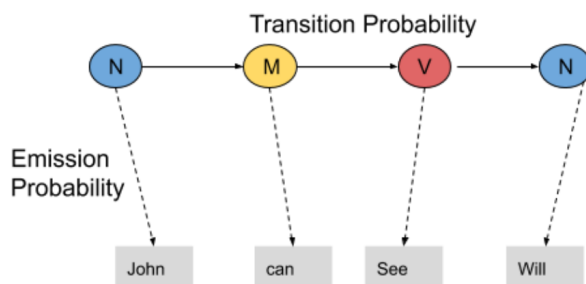
```
#use set datatype to check how many unique tags are present in training  
data
```

```
tags = {tag for word, tag in train_tagged_words}  
print(len(tags))  
print(tags)
```

```
# check total words in vocabulary  
vocab = {word for word, tag in train_tagged_words}
```

Output:

```
12  
{ 'NUM', 'PRT', 'CONJ', '.', 'DET', 'ADP', 'VERB', 'ADV', 'PRON', 'ADJ', 'X', 'NOUN' }
```



Compute 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)
    #total number of times the passed tag occurred in train_bag
    w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]

    #now calculate the total number of times the passed word occurred as the passed tag.
    count_w_given_tag = len(w_given_tag_list)
    return (count_w_given_tag, count_tag)
```

Compute Transition Probability

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

```
# creating t x t transition matrix of tags, t= no of tags
# Matrix(i, j) represents P(jth tag after the ith tag)
```

```
tags_matrix = np.zeros((len(tags), len(tags)), dtype='float32')
for i, t1 in enumerate(list(tags)):
    for j, t2 in enumerate(list(tags)):
        tags_matrix[i, j] = t2_given_t1(t2, t1)[0]/t2_given_t1(t2, t1)[1]

print(tags_matrix)
```

convert the matrix to a df for better readability

```
tags_df = pd.DataFrame(tags_matrix, columns = list(tags), index=list(tags))
display(tags_df)
```


Output:

	NUM	PRT	CONJ	.	DET	ADP	VERB	ADV	PRON	ADJ	X	NOUN
NUM	0.184220	0.026062	0.014281	0.119243	0.003570	0.037487	0.020707	0.003570	0.001428	0.035345	0.202428	0.351660
PRT	0.056751	0.001174	0.002348	0.045010	0.101370	0.019569	0.401174	0.009393	0.017613	0.082975	0.012133	0.250489
CONJ	0.040615	0.004391	0.000549	0.035126	0.123491	0.055982	0.150384	0.057080	0.060373	0.113611	0.009330	0.349067
.	0.078210	0.002789	0.060079	0.092372	0.172192	0.092908	0.089690	0.052569	0.068769	0.046132	0.025641	0.218539
DET	0.022855	0.000287	0.000431	0.017393	0.006037	0.009918	0.040247	0.012074	0.003306	0.206411	0.045134	0.635906
ADP	0.063275	0.001266	0.001012	0.038724	0.320931	0.016958	0.008479	0.014553	0.069603	0.107062	0.034548	0.323589
VERB	0.022836	0.030663	0.005433	0.034807	0.133610	0.092357	0.167956	0.083886	0.035543	0.066390	0.215930	0.110589
ADV	0.029868	0.014740	0.006982	0.139255	0.071373	0.119472	0.339022	0.081458	0.012025	0.130721	0.022886	0.032196
PRON	0.006834	0.014123	0.005011	0.041913	0.009567	0.022323	0.484738	0.036902	0.006834	0.070615	0.088383	0.212756
ADJ	0.021748	0.011456	0.016893	0.066019	0.005243	0.080583	0.011456	0.005243	0.000194	0.063301	0.020971	0.696893
X	0.003075	0.185086	0.010379	0.160869	0.056890	0.142226	0.206419	0.025754	0.054200	0.017682	0.075726	0.061695
NOUN	0.009144	0.043935	0.042454	0.240094	0.013106	0.176827	0.149134	0.016895	0.004659	0.012584	0.028825	0.262344

```
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))

#Here We will only test 10 sentences to check the accuracy
#as testing the whole training set takes huge amount of time
start = time.time()
```

```

tagged_seq = Viterbi(test_tagged_words)
end = time.time()
difference = end-start

print("Time taken in seconds: ", difference)

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

accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)

```

Output:

```

➤ Time taken in seconds: 26.02685284614563
  Viterbi Algorithm Accuracy: 93.77990430622009

```

Output:

```

➤ [('Will', 'NOUN'), ('can', 'VERB'), ('see', 'VERB'), ('Marry', 'NOUN')]
   [('Will', 'NUM'), ('can', 'VERB'), ('see', 'VERB'), ('Marry', 'NUM')]

```

As seen above, using the Viterbi algorithm along with rules can yield us better results.

```

#Code to test all the test sentences
#(takes alot of time to run s0 we wont run it here)
# tagging the test sentences()
test_tagged_words = [tup for sent in test_set for tup in sent]
test_untagged_words = [tup[0] for sent in test_set for tup in sent]
test_untagged_words

start = time.time()
tagged_seq = Viterbi(test_untagged_words)
end = time.time()
difference = end-start

print("Time taken in seconds: ", difference)

# accuracy
check = [i for i, j in zip(test_tagged_words, test_untagged_words) if i == j]

accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)

#To improve the performance,we specify a rule base tagger for unknown words
# specify patterns for tagging
patterns = [
    (r'.*ing$', 'VERB'),          # gerund
    (r'.*ed$', 'VERB'),          # past tense
    (r'.*es$', 'VERB'),          # verb

```

```

        (r'\.*s$', 'NOUN'),          # possessive nouns
        (r'\.*s$', 'NOUN'),          # plural nouns
        (r'\*T?\*?-[0-9]+$', 'X'),   # X
        (r'^-?[0-9]+(\.[0-9]+)?$', 'NUM'), # cardinal numbers
        (r'\.*', 'NOUN')             # nouns
    ]

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

#modified Viterbi to include rule based tagger in it
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)
        state_max = rule_based_tagger.tag([word]) [0] [1]

        if (pmax==0):
            state_max = rule_based_tagger.tag([word]) [0] [1] # assign based on
rule based tagger
        else:
            if state_max != 'X':
                # getting state for which probability is maximum
                state_max = T[p.index(pmax)]

            state.append(state_max)
    return list(zip(words, state))

#test accuracy on subset of test data
start = time.time()
tagged_seq = Viterbi_rule_based(test_tagged_words)
end = time.time()
difference = end-start

```

```

print("Time taken in seconds: ", difference)

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

accuracy = len(check)/len(tagged_seq)
print('Viterbi Algorithm Accuracy: ',accuracy*100)

```

Output:

```

➤ Time taken in seconds: 28.38091015815735
  Viterbi Algorithm Accuracy: 97.1291866028708

```

```

#Check how a sentence is tagged by the two POS taggers
#and compare them
test_sent="Will can see Marry"
pred_tags_rule=Viterbi_rule_based(test_sent.split())
pred_tags_withoutRules= Viterbi(test_sent.split())
print(pred_tags_rule)
print(pred_tags_withoutRules)
#Will and Marry are tagged as NUM as they are unknown words for Viterbi Al

```

```

➤ [('Will', 'NOUN'), ('can', 'VERB'), ('see', 'VERB'), ('Marry', 'NOUN')]
   [('Will', 'NUM'), ('can', 'VERB'), ('see', 'VERB'), ('Marry', 'NUM')]

```

Exercise: Implement the code for POS tagging using LSTM (follow the same manner as we did for RNN).