Next Word Prediction

importing the libraries

In [3]:

```
import tensorflow as tf
import string
import requests
```

get the dataset

In [4]:

```
response=requests.get("http://www.gutenberg.org/cache/epub/5200/pg5200.txt")
```

In [5]:

```
response.text[:1500]
```

Out[5]:

```
'\ufeffThe Project Gutenberg EBook of Metamorphosis, by Franz Kafka\r\nTranslated by David Wyllie.\r\n\r\nThis eBook is for the use of anyone anywhere at no c
ost and with\r\nalmost no restrictions whatsoever. You may copy it, give it away or\r\nre-use it under the terms of the Project Gutenberg License included\r \nwith this eBook or online at www.gutenberg.org\r\n\r\n** This is a COPYRIGHTED Project Gutenberg eBook, Details Below **\r\n** Please follow the copyrig ht guidelines in this file. **\r\n\r\n\r\nTitle: Metamorphosis\r\n\r\nAuthor: Franz Kafka\r\n\r\nTranslator: David Wyllie\r\n\r\nRelease Date: August 16, 2005 [EBook #5200]\r\nFirst posted: May 13, 2002\r\nLast updated: May 20, 2012\r\n\r\nLanguage: English\r\n\r\n\r\n*** START OF THIS PROJECT GUTENBERG EBOOK M ETAMORPHOSIS ***\r\n\r\n\r\n\r\n\r\n\r\nCopyright (C) 2002 David Wyllie.\r\n\r\n\r\n\r\n\r\n\r\n\r\n Metamorphosis\r\n Franz Kafka\r\n\r\nTranslated by David Wyllie \r\n\r\n\r\n\r\nI\r\n\r\n\r\n\r\nOne morning, when Gregor Samsa woke from troubled dreams, he found\r\nhimself transformed in his bed into a horrible vermin. He lay on\r\nhis armour-like back, and if he lifted his head a little he could\r\nsee his brown belly, slightly domed and divided by arches into stiff\r\nsection s. The bedding was hardly able to cover it and seemed ready\r\nto slide off any moment. His many legs, pitifully thin compared\r\nwith the size of the rest of him, waved about helplessly as he\r\nlooked.\r\n\r\n"What\'s happened to me?" he thought. It wasn\'t a dream. His room,\r\na proper human room although a little too small, lay peacefully\r\nbetwee'
```

Split the data set into lines

In [6]:

```
data = response.text.split('\n')
data[0]
```

Out[6]:

```
'\ufeffThe Project Gutenberg EBook of Metamorphosis, by Franz Kafka\r'
```

In [7]:

```
data = data[253:]
data[0]
```

Out[7]:

```
'away from the bed, bend down with the load and then be patient and\r'
```

In [8]:

```
len(data)
```

Out[8]:

```
2110
```

Right now we have a list of the lines in the data. Now we are going to join all the lines and create a long string consisting of the data in continuous format.

In [9]:

```
data = " ".join(data)
data[:1000]
```

Out[9]:

```
'away from the bed, bend down with the load and then be patient and\r careful as he swang over onto the floor, where, hopefully, the\r little legs would find
a use. Should he really call for help\r though, even apart from the fact that all the doors were locked?\r Despite all the difficulty he was in, he could not suppress a smile\r at this thought.\r \r After a while he had already moved so far across that it would have\r been hard for him to keep his balance if he roc ked too hard. The\r time was now ten past seven and he would have to make a final\r decision very soon. Then there was a ring at the door of the flat.\r "Th at\'ll be someone from work", he said to himself, and froze very\r still, although his little legs only became all the more lively as\r they danced around. F or a moment everything remained quiet.\r "They\'re not opening the door", Gregor said to himself, caught in\r some nonsensical hope. But then of course, the
maid\'s firm steps\r went to the door as ever and opened it. Gregor on'
```

we can see that after passing data to clean_text we get the data in the required format without punctuations and special characters.

In [10]:

```
def clean_text(doc):
 tokens = doc.split()
 table = str.maketrans('', '', string.punctuation)
 tokens = [w.translate(table) for w in tokens]
 tokens = [word for word in tokens if word.isalpha()]
 tokens = [word.lower() for word in tokens]
 return tokens

tokens = clean_text(data)
print(tokens[:50])
```

```
['away', 'from', 'the', 'bed', 'bend', 'down', 'with', 'the', 'load', 'and', 'then', 'be', 'patient', 'and', 'careful', 'as', 'he', 'swang', 'over', 'onto', 'the', 'floor', 'where', 'hopefully', 'the', 'little', 'legs', 'would', 'find', 'a', 'use', 'should', 'he', 'really', 'call', 'for', 'help', 'though', 'even', 'apart', 'from', 'the', 'fact', 'that', 'all', 'the', 'doors', 'were', 'locked', 'despite']
```

In [11]:

```
len(tokens)
```

Out[11]:

```
22607
```

we are going to use a set of previous words to predict the next word in the sentence. To be precise we are going to use a set of 50 words to predict the 51st word. Hence we are going to divide our data in chunks of 51 words and at the last we will separate the last word from every line. We are going to limit our dataset to 200000 words.

In [12]:

```
length = 50 + 1
lines = []

for i in range(length, len(tokens)):
 seq = tokens[i-length:i]
 line = ' '.join(seq)
 lines.append(line)
 if i > 200000:
 break

print(len(lines))
```

```
22556
```

Build LSTM Model and Prepare X and y

import all the necessary libraries used to pre-process the data and create the layers of the neural network.

In [13]:

```
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Embedding
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

We are going to create a unique numerical token for each unique word in the dataset.fit_on_texts() updates internal vocabulary based on a list of texts. texts_to_sequences() transforms each text in texts to a sequence of integers.

In [14]:

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
```

sequences containes a list of integer values created by tokenizer. Each line in sequences has 51 words. Now we will split each line such that the first 50 words are in X and the last word is in y.

In [15]:

```
sequences = np.array(sequences)
X, y = sequences[:, :-1], sequences[:,-1]
X[0]
```

Out[15]:

```
array([ 103, 29, 1, 245, 2883, 98, 14, 1, 1435, 3, 48,

 30, 618, 3, 756, 13, 6, 1434, 107, 165, 1, 149,
 86, 2880, 1, 78, 225, 21, 530, 12, 156, 193, 6,
 142, 754, 17, 180, 116, 49, 1433, 29, 1, 753, 11,
 26, 1, 455, 58, 617, 329])
```

vocab_size contains all the uniques words in the dataset. tokenizer.word_index gives the mapping of each unique word to its numerical equivalent. Hence len() of tokenizer.word_index gives the vocab_size

In [16]:

```
vocab_size = len(tokenizer.word_index) + 1
```

to_categorical() converts a class vector (integers) to binary class matrix. num_classes is the total number of classes which is vocab_size.

In [17]:

```
y = to_categorical(y, num_classes=vocab_size)
```

In [18]:

```
seq_length = X.shape[1]
seq_length
```

Out[18]:

```
50
```

LSTM Model

A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.

In [19]:

```python
model = Sequential()
model.add(Embedding(vocab_size, 50, input_length=seq_length))
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))
```

In [20]:

```python
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
embedding (Embedding) (None, 50, 50) 144250
_________________________________________________________________
lstm (LSTM) (None, 50, 100) 60400
_________________________________________________________________
lstm_1 (LSTM) (None, 100) 80400
_________________________________________________________________
dense (Dense) (None, 100) 10100
_________________________________________________________________
dense_1 (Dense) (None, 2885) 291385
=================================================================
Total params: 586,535
Trainable params: 586,535
Non-trainable params: 0
_________________________________________________________________
```

In [21]:

```python
model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
```

After compiling the model we will now train the model using model.fit() on the training dataset. We will use 100 epochs to train the model. An epoch is an iteration over the entire x and y data provided. batch_size is the number of samples per gradient update i.e. the weights will be updates after 256 training examples.

In [22]:

```python
model.fit(X, y, batch_size = 256, epochs = 100)
```

```
Epoch 1/100
89/89 [==============================] - 32s 286ms/step - loss: 6.6286 - accuracy: 0.0533
Epoch 2/100
89/89 [==============================] - 28s 312ms/step - loss: 6.1863 - accuracy: 0.0540
Epoch 3/100
89/89 [==============================] - 29s 324ms/step - loss: 6.1646 - accuracy: 0.0540
Epoch 4/100
89/89 [==============================] - 30s 335ms/step - loss: 6.0548 - accuracy: 0.0540
Epoch 5/100
89/89 [==============================] - 32s 358ms/step - loss: 5.9604 - accuracy: 0.0546
Epoch 6/100
89/89 [==============================] - 29s 333ms/step - loss: 5.8097 - accuracy: 0.0606
Epoch 7/100
89/89 [==============================] - 31s 354ms/step - loss: 5.7121 - accuracy: 0.0674
Epoch 8/100
89/89 [==============================] - 31s 345ms/step - loss: 5.6420 - accuracy: 0.0727
Epoch 9/100
89/89 [==============================] - 34s 384ms/step - loss: 5.5737 - accuracy: 0.0771
Epoch 10/100
89/89 [==============================] - 34s 381ms/step - loss: 5.5113 - accuracy: 0.0814
Epoch 11/100
89/89 [==============================] - 32s 361ms/step - loss: 5.4571 - accuracy: 0.0869
Epoch 12/100
89/89 [==============================] - 32s 359ms/step - loss: 5.4046 - accuracy: 0.0900
Epoch 13/100
89/89 [==============================] - 32s 361ms/step - loss: 5.3515 - accuracy: 0.0951
Epoch 14/100
89/89 [==============================] - 35s 391ms/step - loss: 5.2971 - accuracy: 0.0994
Epoch 15/100
89/89 [==============================] - 31s 353ms/step - loss: 5.2420 - accuracy: 0.1012
Epoch 16/100
89/89 [==============================] - 31s 344ms/step - loss: 5.1879 - accuracy: 0.1050
Epoch 17/100
89/89 [==============================] - 31s 348ms/step - loss: 5.1422 - accuracy: 0.1076
Epoch 18/100
89/89 [==============================] - 31s 351ms/step - loss: 5.0895 - accuracy: 0.1127
Epoch 19/100
89/89 [==============================] - 31s 349ms/step - loss: 5.0384 - accuracy: 0.1159
Epoch 20/100
89/89 [==============================] - 31s 349ms/step - loss: 4.9907 - accuracy: 0.1197
Epoch 21/100
89/89 [==============================] - 31s 345ms/step - loss: 4.9425 - accuracy: 0.1231
Epoch 22/100
89/89 [==============================] - 33s 371ms/step - loss: 4.8975 - accuracy: 0.1242
Epoch 23/100
89/89 [==============================] - 31s 348ms/step - loss: 4.8493 - accuracy: 0.1282
Epoch 24/100
89/89 [==============================] - 31s 350ms/step - loss: 4.7985 - accuracy: 0.1312
Epoch 25/100
89/89 [==============================] - 31s 350ms/step - loss: 4.7511 - accuracy: 0.1360
Epoch 26/100
89/89 [==============================] - 31s 353ms/step - loss: 4.7086 - accuracy: 0.1386
Epoch 27/100
89/89 [==============================] - 32s 355ms/step - loss: 4.6627 - accuracy: 0.1423
Epoch 28/100
89/89 [==============================] - 32s 360ms/step - loss: 4.6216 - accuracy: 0.1446
Epoch 29/100
89/89 [==============================] - 33s 368ms/step - loss: 4.5774 - accuracy: 0.1502
Epoch 30/100
89/89 [==============================] - 28s 317ms/step - loss: 4.5345 - accuracy: 0.1517
Epoch 31/100
89/89 [==============================] - 29s 326ms/step - loss: 4.4967 - accuracy: 0.1579
Epoch 32/100
89/89 [==============================] - 30s 339ms/step - loss: 4.4517 - accuracy: 0.1585
Epoch 33/100
89/89 [==============================] - 30s 336ms/step - loss: 4.4139 - accuracy: 0.1627
Epoch 34/100
89/89 [==============================] - 33s 369ms/step - loss: 4.3756 - accuracy: 0.1644
Epoch 35/100
89/89 [==============================] - 33s 371ms/step - loss: 4.3356 - accuracy: 0.1671
Epoch 36/100
89/89 [==============================] - 33s 368ms/step - loss: 4.2984 - accuracy: 0.1655
Epoch 37/100
89/89 [==============================] - 34s 379ms/step - loss: 4.2631 - accuracy: 0.1698
Epoch 38/100
89/89 [==============================] - 34s 379ms/step - loss: 4.2237 - accuracy: 0.1719
Epoch 39/100
89/89 [==============================] - 34s 379ms/step - loss: 4.1897 - accuracy: 0.1750
Epoch 40/100
89/89 [==============================] - 34s 381ms/step - loss: 4.1546 - accuracy: 0.1766
Epoch 41/100
89/89 [==============================] - 34s 376ms/step - loss: 4.1202 - accuracy: 0.1791
Epoch 42/100
89/89 [==============================] - 34s 380ms/step - loss: 4.0842 - accuracy: 0.1827
Epoch 43/100
89/89 [==============================] - 33s 371ms/step - loss: 4.0563 - accuracy: 0.1830
Epoch 44/100
89/89 [==============================] - 33s 366ms/step - loss: 4.0208 - accuracy: 0.1851
Epoch 45/100
89/89 [==============================] - 33s 373ms/step - loss: 3.9841 - accuracy: 0.1880
Epoch 46/100
89/89 [==============================] - 34s 383ms/step - loss: 4.0119 - accuracy: 0.1854
Epoch 47/100
89/89 [==============================] - 34s 379ms/step - loss: 4.6439 - accuracy: 0.1425
Epoch 48/100
89/89 [==============================] - 34s 379ms/step - loss: 4.2285 - accuracy: 0.1657
Epoch 49/100
89/89 [==============================] - 34s 380ms/step - loss: 4.0601 - accuracy: 0.1766
Epoch 50/100
89/89 [==============================] - 33s 375ms/step - loss: 3.9604 - accuracy: 0.1887
Epoch 51/100
89/89 [==============================] - 34s 377ms/step - loss: 3.9000 - accuracy: 0.1937
Epoch 52/100
89/89 [==============================] - 34s 381ms/step - loss: 3.8429 - accuracy: 0.1990
Epoch 53/100
89/89 [==============================] - 34s 380ms/step - loss: 3.8056 - accuracy: 0.2029
Epoch 54/100
89/89 [==============================] - 34s 380ms/step - loss: 3.8391 - accuracy: 0.1978
Epoch 55/100
89/89 [==============================] - 34s 381ms/step - loss: 3.8186 - accuracy: 0.2002
Epoch 56/100
89/89 [==============================] - 34s 377ms/step - loss: 3.7560 - accuracy: 0.2077
Epoch 57/100
89/89 [==============================] - 34s 379ms/step - loss: 3.7023 - accuracy: 0.2109
Epoch 58/100
89/89 [==============================] - 34s 381ms/step - loss: 3.6516 - accuracy: 0.2193
Epoch 59/100
89/89 [==============================] - 34s 380ms/step - loss: 3.6059 - accuracy: 0.2254
Epoch 60/100
89/89 [==============================] - 35s 393ms/step - loss: 3.5610 - accuracy: 0.2298
Epoch 61/100
89/89 [==============================] - 34s 381ms/step - loss: 3.5158 - accuracy: 0.2357
Epoch 62/100
89/89 [==============================] - 34s 381ms/step - loss: 3.4702 - accuracy: 0.2402
Epoch 63/100
89/89 [==============================] - 34s 379ms/step - loss: 3.4297 - accuracy: 0.2471
Epoch 64/100
89/89 [==============================] - 34s 381ms/step - loss: 3.3787 - accuracy: 0.2532
Epoch 65/100
89/89 [==============================] - 34s 383ms/step - loss: 3.3360 - accuracy: 0.2606
Epoch 66/100
89/89 [==============================] - 34s 382ms/step - loss: 3.2913 - accuracy: 0.2649
Epoch 67/100
89/89 [==============================] - 34s 379ms/step - loss: 3.2408 - accuracy: 0.2743
Epoch 68/100
89/89 [==============================] - 34s 382ms/step - loss: 3.1928 - accuracy: 0.2818
Epoch 69/100
89/89 [==============================] - 34s 380ms/step - loss: 3.1544 - accuracy: 0.2872
Epoch 70/100
89/89 [==============================] - 34s 379ms/step - loss: 3.1073 - accuracy: 0.2967
Epoch 71/100
89/89 [==============================] - 34s 382ms/step - loss: 3.0578 - accuracy: 0.3033
Epoch 72/100
```

```
89/89 [==============================] - 34s 379ms/step - loss: 3.0159 - accuracy: 0.3102
Epoch 73/100
89/89 [==============================] - 34s 381ms/step - loss: 2.9814 - accuracy: 0.3133
Epoch 74/100
89/89 [==============================] - 34s 381ms/step - loss: 2.9322 - accuracy: 0.3246
Epoch 75/100
89/89 [==============================] - 33s 372ms/step - loss: 2.8932 - accuracy: 0.3305
Epoch 76/100
89/89 [==============================] - 34s 384ms/step - loss: 2.8588 - accuracy: 0.3352
Epoch 77/100
89/89 [==============================] - 34s 380ms/step - loss: 2.8155 - accuracy: 0.3459
Epoch 78/100
89/89 [==============================] - 34s 382ms/step - loss: 2.7867 - accuracy: 0.3518
Epoch 79/100
89/89 [==============================] - 34s 381ms/step - loss: 2.8494 - accuracy: 0.3391
Epoch 80/100
89/89 [==============================] - 34s 380ms/step - loss: 2.7980 - accuracy: 0.3495
Epoch 81/100
89/89 [==============================] - 34s 379ms/step - loss: 2.9486 - accuracy: 0.3259
Epoch 82/100
89/89 [==============================] - 34s 382ms/step - loss: 2.7581 - accuracy: 0.3537
Epoch 83/100
89/89 [==============================] - 34s 379ms/step - loss: 2.7084 - accuracy: 0.3623
Epoch 84/100
89/89 [==============================] - 34s 380ms/step - loss: 2.6768 - accuracy: 0.3687
Epoch 85/100
89/89 [==============================] - 34s 380ms/step - loss: 2.6200 - accuracy: 0.3807
Epoch 86/100
89/89 [==============================] - 34s 382ms/step - loss: 2.6001 - accuracy: 0.3861
Epoch 87/100
89/89 [==============================] - 34s 382ms/step - loss: 2.5829 - accuracy: 0.3881
Epoch 88/100
89/89 [==============================] - 34s 383ms/step - loss: 2.6642 - accuracy: 0.3777
Epoch 89/100
89/89 [==============================] - 34s 381ms/step - loss: 2.6497 - accuracy: 0.3776
Epoch 90/100
89/89 [==============================] - 34s 381ms/step - loss: 2.7969 - accuracy: 0.3494
Epoch 91/100
89/89 [==============================] - 34s 385ms/step - loss: 2.7195 - accuracy: 0.3664
Epoch 92/100
89/89 [==============================] - 34s 382ms/step - loss: 2.8789 - accuracy: 0.3356
Epoch 93/100
89/89 [==============================] - 34s 384ms/step - loss: 2.9671 - accuracy: 0.3139
Epoch 94/100
89/89 [==============================] - 34s 384ms/step - loss: 2.8647 - accuracy: 0.3333
Epoch 95/100
89/89 [==============================] - 34s 385ms/step - loss: 2.8068 - accuracy: 0.3487
Epoch 96/100
89/89 [==============================] - 34s 382ms/step - loss: 2.7667 - accuracy: 0.3527
Epoch 97/100
89/89 [==============================] - 34s 383ms/step - loss: 2.8761 - accuracy: 0.3300
Epoch 98/100
89/89 [==============================] - 34s 381ms/step - loss: 2.8692 - accuracy: 0.3283
Epoch 99/100
89/89 [==============================] - 34s 382ms/step - loss: 2.7949 - accuracy: 0.3441
Epoch 100/100
89/89 [==============================] - 34s 380ms/step - loss: 2.7516 - accuracy: 0.3507
```

Out[22]:

```
<tensorflow.python.keras.callbacks.History at 0x2979c8d47f0>
```

We are now going to generate words using the model. For this we need a set of 50 words to predict the 51st word. So we are taking a random line.

In [23]:

```
seed_text=lines[12343]
seed_text
```

Out[23]:

```
'condition seemed serious enough to remind even his father that gregor despite his current sad and revolting form was a family member who could not be treated as an enemy on the contrary as a family there was a duty to swallow any revulsion for him and to be patient just'
```

generate_text_seq() generates n_words number of words after the given seed_text. We are going to pre-process the seed_text before predicting. We are going to encode the seed_text using the same encoding used for encoding the training data. Then we are going to convert the seed_textto 50 words by using pad_sequences(). Now we will predict using model.predict_classes(). After that we will search the word in tokenizer using the index in y_predict. Finally we will append the predicted word to seed_text and text and repeat the process.

In [24]:

```
def generate_text_seq(model, tokenizer, text_seq_length, seed_text, n_words):
  text = []

  for _ in range(n_words):
  encoded = tokenizer.texts_to_sequences([seed_text])[0]
  encoded = pad_sequences([encoded], maxlen = text_seq_length, truncating='pre')

  y_predict = model.predict_classes(encoded)

  predicted_word = ''
  for word, index in tokenizer.word_index.items():
  if index == y_predict:
  predicted_word = word
  break
  seed_text = seed_text + ' ' + predicted_word
  text.append(predicted_word)
  return ' '.join(text)
```

We can see that the next 100 words are predicted by the model for the seed_text.

In [25]:

```
generate_text_seq(model, tokenizer, seq_length, seed_text, 100)
```

```
C:\Drivers\anaconda\lib\site-packages\tensorflow\python\keras\engine\sequential.py:455: UserWarning: `model.predict_classes()` is deprecated and will be remov ed after 2021-01-01. Please use instead:* `np.argmax(model.predict(x), axis=-1)`, if your model does multi-class classification (e.g. if it uses a `softma x` last-layer activation).* `(model.predict(x) > 0.5).astype("int32")`, if your model does binary classification (e.g. if it uses a `sigmoid` last-layer a ctivation).
  warnings.warn('`model.predict_classes()` is deprecated and '
```

Out[25]:

```
'the first word to him the door and wipe himself on the floor he had been knocked at all the door and was already begun to speak to him and the door and slid down in front of the couch shivering the living room and pressed up against the door to him he had been working for them and smoking though the practise he wa s already able to speak to him mixed from by hope that he was stretching him out the door to bear the door to bear the door to him he had woke to attract the door'
```

We have got a accuracy of 46%. To increase the accuracy we can increase the number of epochs or we can consider the entire data for training. For this model we have only considered 1/4th of the data for training.