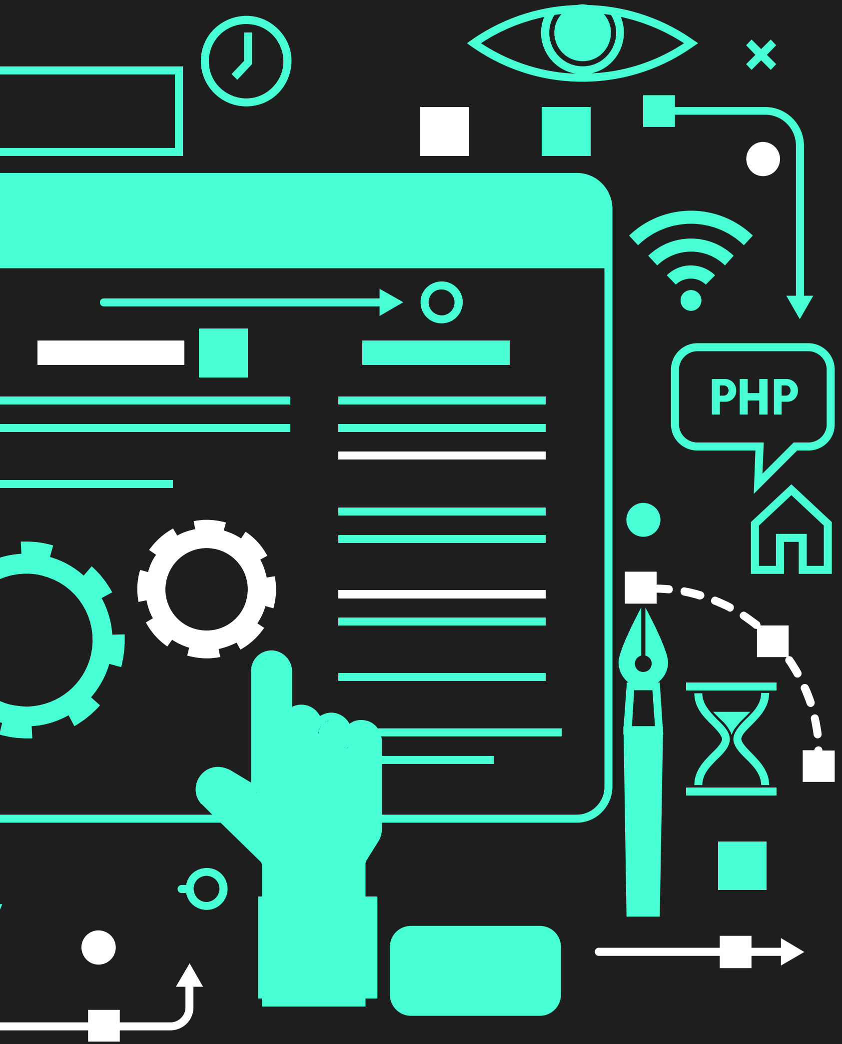



# NEXT WORD PREDICTION

PROJECT BY:
HARSH GUPTA

# TABLE OF CONTENTS

# PROBLEM FORMULATION

We are opening a blog for data scientists, where people can write articles and submit their ideas and solutions. To create a self made search algorithm, we are using our own algorithm to train the models on pre existing medium (https://medium.com/) articles and help us for predicting the next word according to the input in the search bar.

As we do more and more typing into non-traditional keyboards every advantage in ease of typing becomes more critical. Next word prediction is a means of helping the user to type faster by suggesting common words that are likely to follow, thus saving the user.

# LOGISTICS

**Libraries used:**
NLTK, Pandas, NumPy, Re, TensorFlow, Keras, Flask, Pickle, Matplotlib

**Languages:**
HTML, CSS, Python, javaScript

**Applications Used:**
Google Colab, Jupyter, Spyder, Ananconda

# DATA

Medium is one of the most famous tools for spreading knowledge about almost any field. It is widely used to published articles on ML, AI, NLP and data science. This dataset is the collection of about 350 articles in such fields.

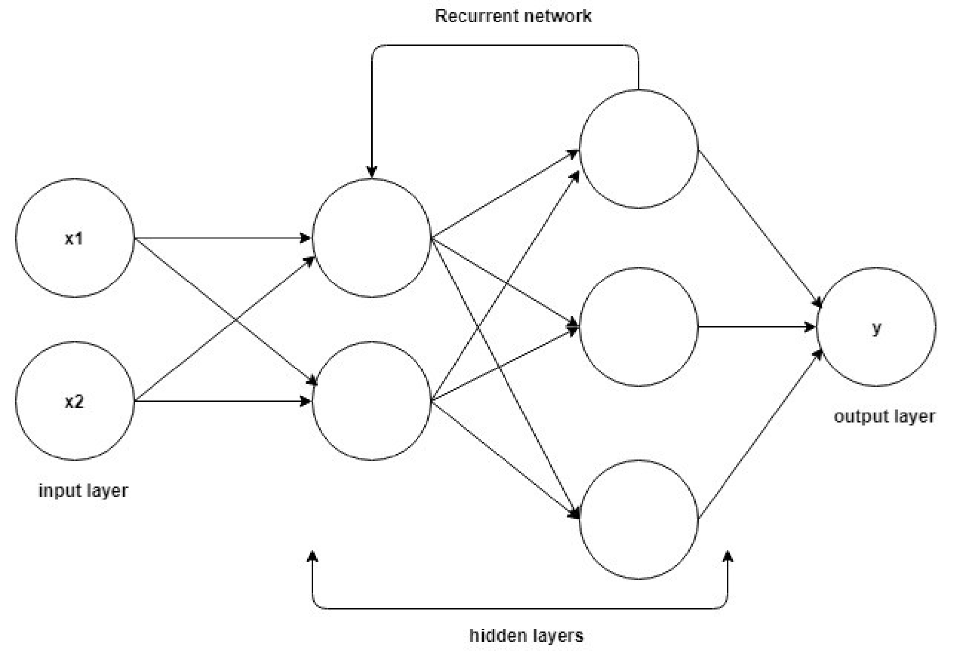
The dataset contains articles, their title, number of claps it has received, their links and their reading time in which we will focus on the articles. You can have a look at the dataset by clicking here.

# APPROACH

We have used the concept of RNN to create a model with LSTM layers to read a sequence of words converted into numbers. Using the model we have tried to capture the words which come after a particular sequence and generate the output accordingly

# WHAT IS RNN?

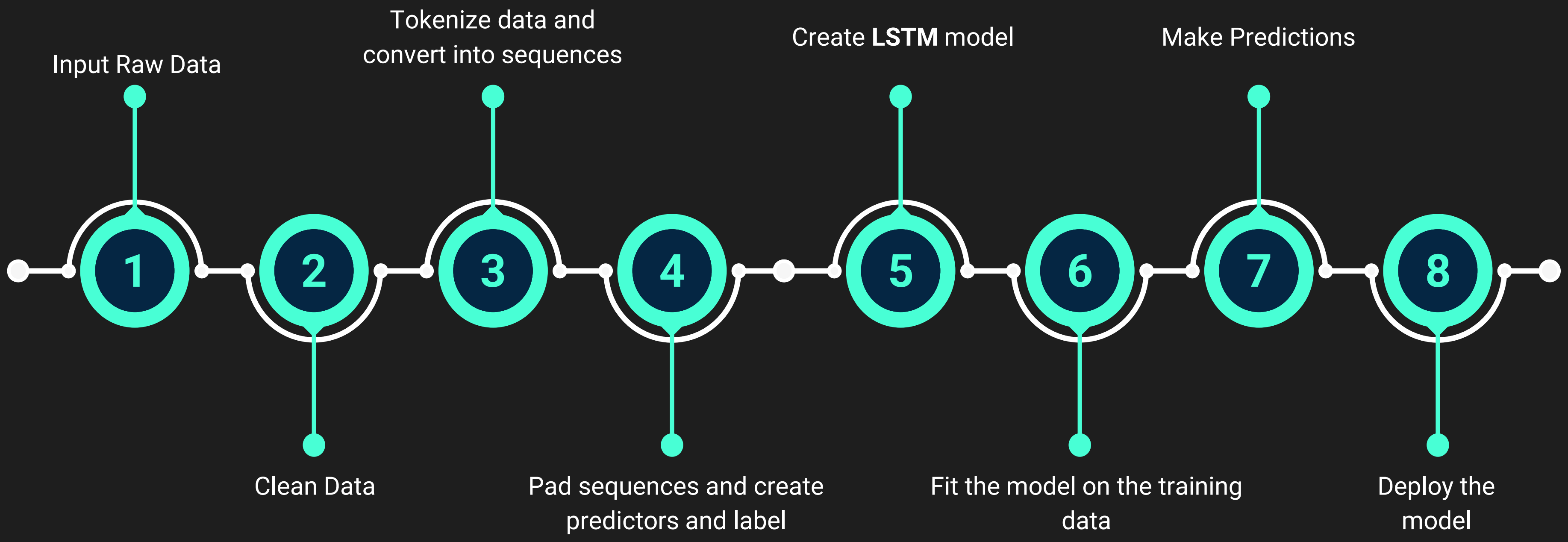
Unlike Feed-forward neural networks in which activation outputs are propagated only in one direction, the activation outputs from neurons propagate in both directions (from inputs to outputs and from outputs to inputs) in **Recurrent Neural Networks**. This creates loops in the neural network architecture which acts as a 'memory state' of the neurons. This state allows the neurons an ability to remember what have been learned so far.

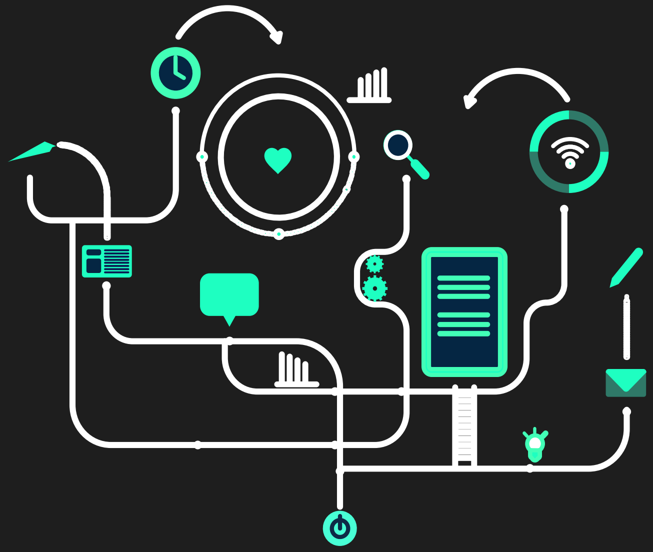


# HOW IS LSTM BETTER THAN RNN?

- The memory state in RNNs gives an advantage over traditional neural networks but a problem called Vanishing Gradient is associated with them. In this problem, while learning with a large number of layers, it becomes really hard for the network to learn and tune the parameters of the earlier layers. To address this problem, A new type of RNNs called LSTMs (**Long Short Term Memory**) Models have been developed.
- LSTM model is a special kind of RNN that learns long-term dependencies. It introduces new structure — the memory cell that is composed of four elements: input, forget and output gates and a neuron that connects to itself.

# SOLUTION FLOW



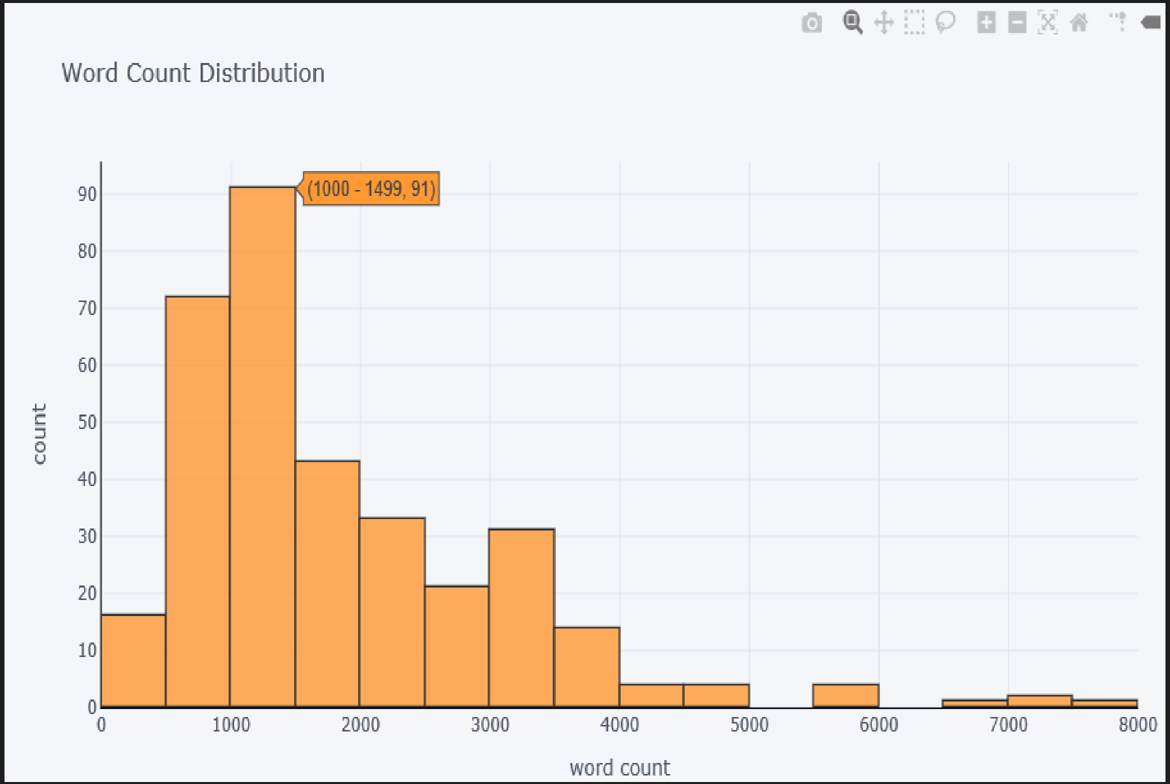


# EXPLORATORY DATA ANALYSIS

Natural Language Processing task needs more time on cleaning and exploring the text data. Exploratory data analysis (EDA) is an approach to analyzing data sets to summarize their main characteristics, often with visual methods.

For EDA we use .iplot which is an interactive plot and it helps us to interpret the plots in a better way and also save plots easily.

# Word Count Distribution




```
df['word_count'].iplot(
    kind='hist',
    xTitle='word count',
    linecolor='black',
    yTitle='count',
    title='Word Count Distribution')
```

# The distribution of top words before removing stop words

```
def get_top_n_words(corpus, n=None):
    vec = CountVectorizer().fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.vocabulary_.items()]
    words_freq =sorted(words_freq, key = lambda x: x[1], reverse=True)
    return words_freq[:n]
common_words = get_top_n_words(df['text'], 20)
for word, freq in common_words:
    print(word, freq)
df1 = pd.DataFrame(common_words, columns = ['text' , 'count'])
df1.groupby('text').sum()['count'].sort_values(ascending=False).iplot(
    kind='bar', yTitle='Count', linecolor='black', title='Top 20 words in text before removing stop words')
```

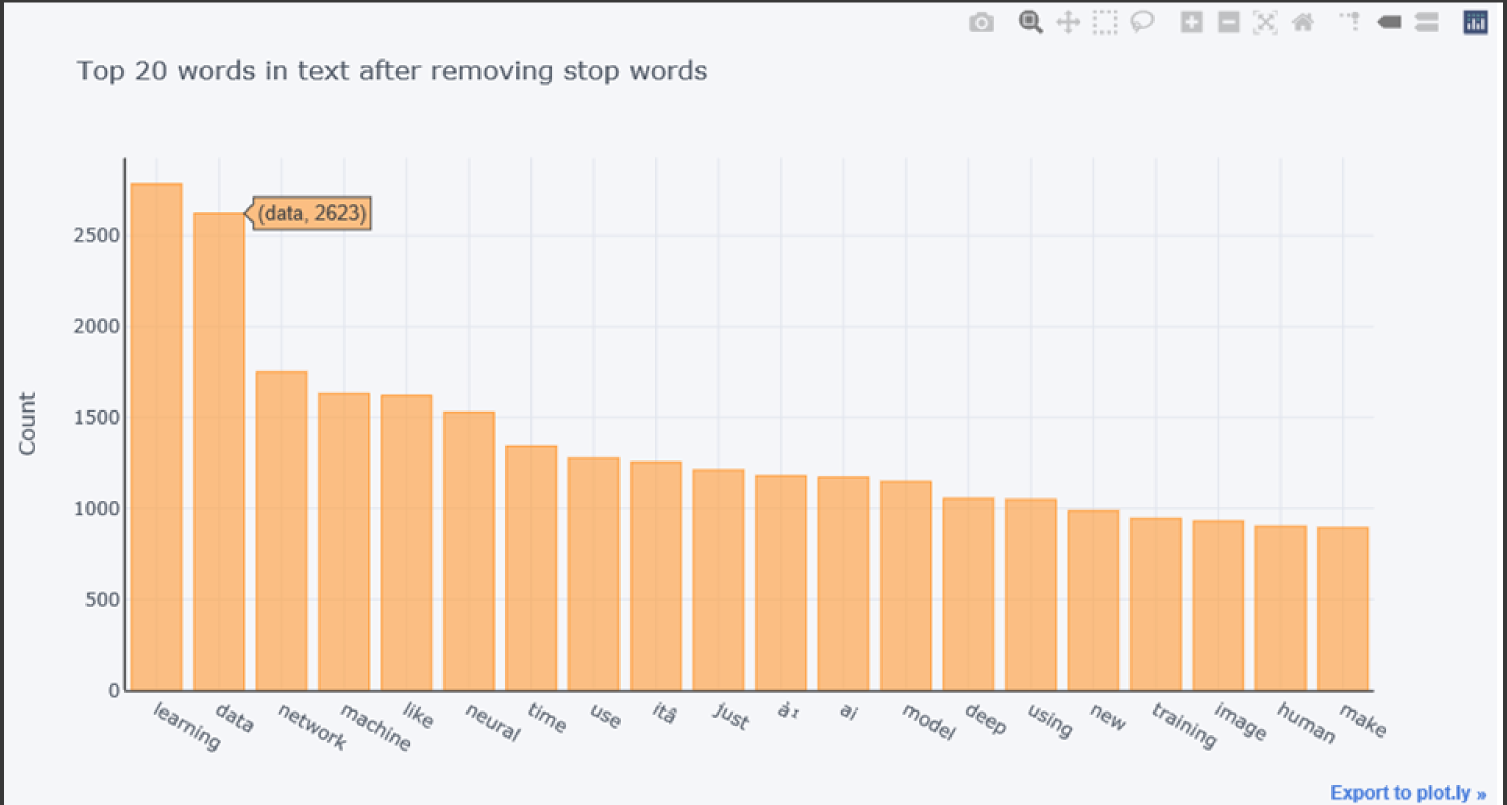
# The distribution of top words before removing stop words


the 31513
to 19239
of 16638
and 14915
in 10074
is 9114
that 8087
for 6176
this 6130
we 6022
it 5893
you 5143
with 4698
on 4607
are 4353
as 4180
be 3730
can 3352
learning 2783
an 2773

# The distribution of top words after removing stop words

```
def get_top_n_words(corpus, n=None):
    vec = CountVectorizer(stop_words = 'english').fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.vocabulary_.items()]
    words_freq =sorted(words_freq, key = lambda x: x[1], reverse=True)
    return words_freq[:n]
common_words = get_top_n_words(df['text'], 20)
for word, freq in common_words:
    print(word, freq)
df2 = pd.DataFrame(common_words, columns = ['text' , 'count'])
df2.groupby('text').sum()['count'].sort_values(ascending=False).iplot(
    kind='bar', yTitle='Count', linecolor='black', title='Top 20 words in text after removing stop words')
```

# The distribution of top words after removing stop words

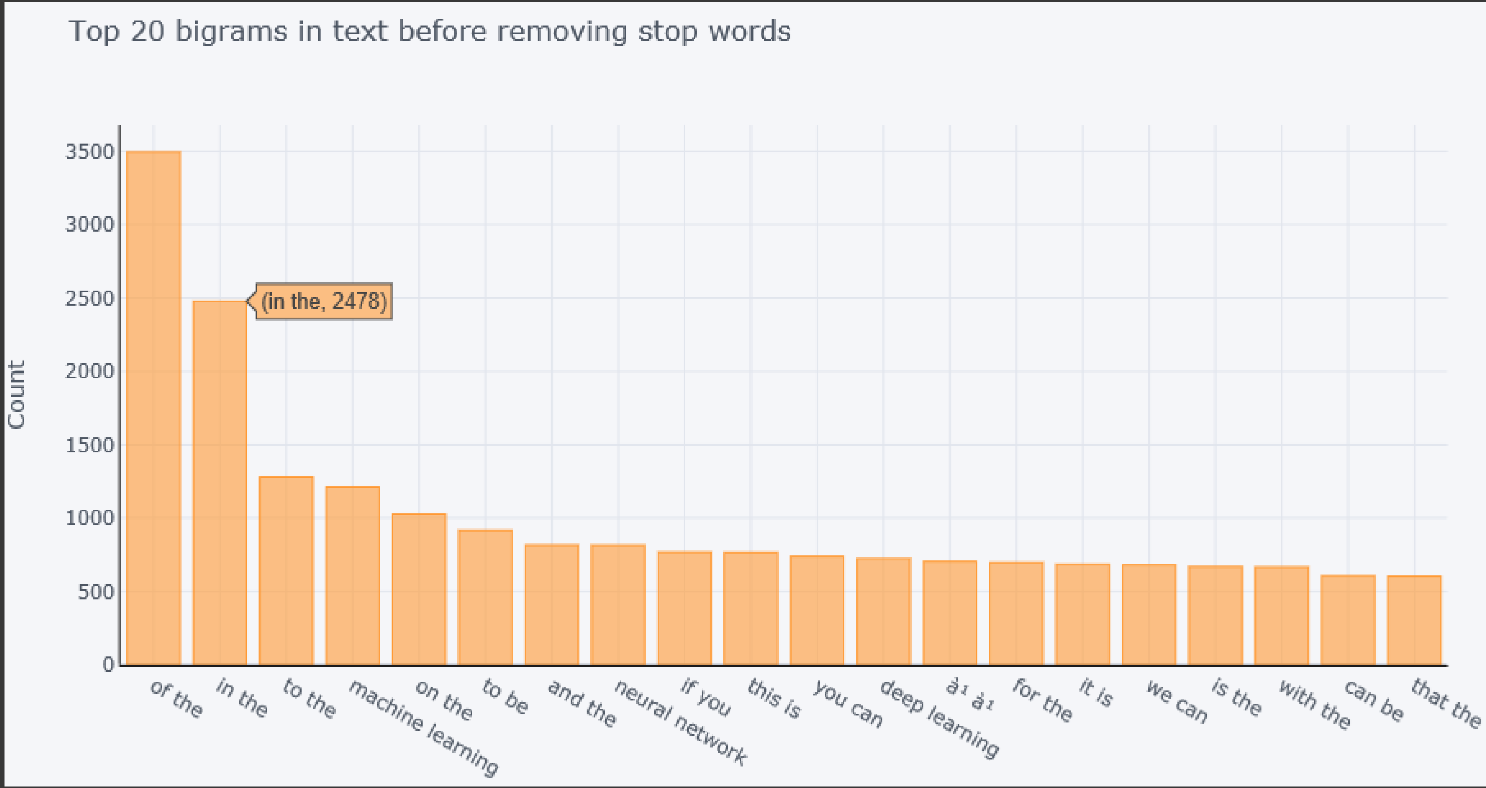



learning 2783
data 2623
network 1751
machine 1632
like 1621
neural 1529
time 1342
use 1277
itâ 1254
just 1210
à¹ 1181
ai 1173
model 1148
deep 1055
using 1049
new 987
training 945
image 930
human 903
make 895

# The distribution of top bigrams before removing stop words

```python
def get_top_n_bigram(corpus, n=None):
    vec = CountVectorizer(ngram_range=(2, 2)).fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.vocabulary_.items()]
    words_freq =sorted(words_freq, key = lambda x: x[1], reverse=True)
    return words_freq[:n]
common_words = get_top_n_bigram(df['text'], 20)
for word, freq in common_words:
    print(word, freq)
df3 = pd.DataFrame(common_words, columns = ['text' , 'count'])
df3.groupby('text').sum()['count'].sort_values(ascending=False).iplot(
    kind='bar', yTitle='Count', linecolor='black', title='Top 20 bigrams in text before removing stop words')
```

# The distribution of top bigrams before removing stop words

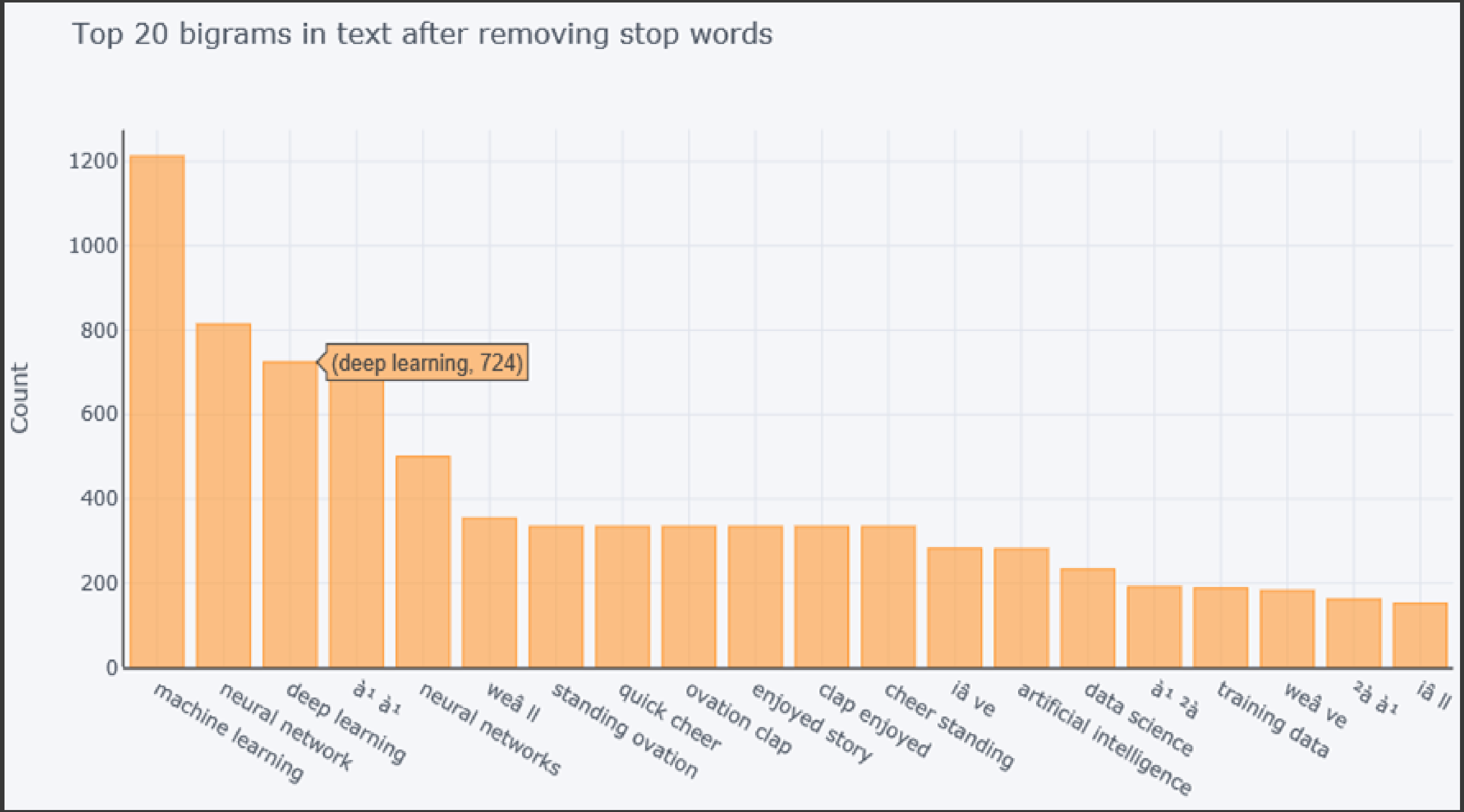



of the 3496
in the 2478
to the 1281
machine learning 1212
on the 1028
to be 916
and the 815
neural network 814
if you 768
this is 766
you can 739
deep learning 724
à¹ à¹ 704
for the 697
it is 685
we can 681
is the 668
with the 666
can be 606
that the 602

# The distribution of top bigrams after removing stop words

```
def get_top_n_bigram(corpus, n=None):
    vec = CountVectorizer(ngram_range=(2, 2), stop_words='english').fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.vocabulary_.items()]
    words_freq =sorted(words_freq, key = lambda x: x[1], reverse=True)
    return words_freq[:n]
common_words = get_top_n_bigram(df['text'], 20)
for word, freq in common_words:
    print(word, freq)
df4 = pd.DataFrame(common_words, columns = ['text' , 'count'])
df4.groupby('text').sum()['count'].sort_values(ascending=False).iplot(
    kind='bar', yTitle='Count', linecolor='black', title='Top 20 bigrams in text after removing stop words')
```

# The distribution of top bigrams after removing stop words




machine learning 1212
neural network 814
deep learning 724
à¹ à¹ 704
neural networks 500
weâ ll 354
quick cheer 334
cheer standing 334
standing ovation 334
ovation clap 334
clap enjoyed 334
enjoyed story 334
iâ ve 282
artificial intelligence 281
data science 233
à¹ ²à 191
training data 188
weâ ve 182
²à à¹ 162
iâ ll 152

# DATA PREPARATION

To prepare the data, we have first removed punctuations, symbols and stopwords. The data has then been tokenized and converted into sequences. These sequences are then padded and the labels are created by taking the last word of every sequence.

# Remove Punctuations, Symbols and Stop Words

```
REPLACE_BY_SPACE_RE = re.compile('[/(){}\[\]\|@,;]')
BAD_SYMBOLS_RE = re.compile('[^0-9a-z #+_]')
stop_words = set(stopwords.words('english'))

def clean_text(text):
    text = text.lower()
    text = REPLACE_BY_SPACE_RE.sub(' ', text)
    text = BAD_SYMBOLS_RE.sub(' ', text)
    text = ' '.join(word for word in text.split() if word not in stop_words)
    return text
```

You cannot go straight from raw text to fitting a machine learning or deep learning model. First we must clean your text first, which means splitting it into words and handling punctuation and case.

Here are a few key benefits of removing stopwords:

- On removing stopwords, dataset size decreases and the time to train the model also decreases
- Removing stopwords can potentially help improve the performance as there are fewer and only meaningful tokens left. Thus, it could increase classification accuracy
- Even search engines like Google remove stopwords for fast and relevant retrieval of data from the database

# Tokenization

We turn the texts into space-separated sequences of words in lowercase. These sequences are then split into lists of tokens. Hence we convert the corpus into sequence of tokens using tokenization.

```
t = Tokenizer(num_words=None, filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n', lower=True, split=' ',
              char_level=False, oov_token=None, document_count=0)
t.fit_on_texts(corpus)
print('Found %s unique tokens.' % len(t.word_index))
```

```
Found 13664 unique tokens.
```

# Convert the articles into sequences

**Now after Tokenization, we convert the corpus into sequence of tokens. We have defined the sequence length as 20. The output can be seen in the next slide.**

```
def get_sequence_of_tokens(corpus):
    total_words = len(t.word_index) + 1

    input_sequences = []
    for line in corpus:
        token_list = t.texts_to_sequences([line])[0]
        for i in range(1, len(token_list)-seq_length):
            n_gram_sequence = token_list[i:i+seq_length]
            input_sequences.append(n_gram_sequence)

    return input_sequences, total_words
input_sequences, total_words = get_sequence_of_tokens(corpus)
print(len(input_sequences))
```

# Output of the sequence of tokens

Here we can see the sequences generated, with a length of 20 words per sequence

```
['headlines blared chatbots next big thing hopes sky high bright eyed bushy tailed industry ripe new era innovation time start',
 'blared chatbots next big thing hopes sky high bright eyed bushy tailed industry ripe new era innovation time start socializing',
 'chatbots next big thing hopes sky high bright eyed bushy tailed industry ripe new era innovation time start socializing machines',
 'next big thing hopes sky high bright eyed bushy tailed industry ripe new era innovation time start socializing machines road',
 'big thing hopes sky high bright eyed bushy tailed industry ripe new era innovation time start socializing machines road signs',
 'thing hopes sky high bright eyed bushy tailed industry ripe new era innovation time start socializing machines road signs pointed',
 'hopes sky high bright eyed bushy tailed industry ripe new era innovation time start socializing machines road signs pointed towards',
 'sky high bright eyed bushy tailed industry ripe new era innovation time start socializing machines road signs pointed towards insane',
 'high bright eyed bushy tailed industry ripe new era innovation time start socializing machines road signs pointed towards insane success',
 'bright eyed bushy tailed industry ripe new era innovation time start socializing machines road signs pointed towards insane success mobile',
 'eyed bushy tailed industry ripe new era innovation time start socializing machines road signs pointed towards insane success mobile world',
 'bushy tailed industry ripe new era innovation time start socializing machines road signs pointed towards insane success mobile world congress',
 'tailed industry ripe new era innovation time start socializing machines road signs pointed towards insane success mobile world congress 2017',
 'industry ripe new era innovation time start socializing machines road signs pointed towards insane success mobile world congress 2017 chatbots',
 'ripe new era innovation time start socializing machines road signs pointed towards insane success mobile world congress 2017 chatbots main']
```

# Pad sequences and create predictors and label

The sequences are taken and padded to a uniform length of 20. The last word of every sequence is then taken as the label for the sequence.

```
def generate_padded_sequences(input_sequences):
    input_sequences = np.array(pad_sequences(input_sequences,
                                             maxlen = seq_length, padding = 'pre'))
    predictors, label = input_sequences[:,:-1],input_sequences[:,-1]
    label = ku.to_categorical(label, num_classes = total_words)

    return predictors, label

predictors, label= generate_padded_sequences(input_sequences)
```

# MODEL GENERATION

To build the the model, we have used Keras Sequential method and added embedding, LSTM and Dense layers. The final model is being evaluated on the basis of training loss and is being stored in a hdf5 file for deployment use.

# Creating LSTM Model

```
def create_model(sequence_len, total_words):
    model = Sequential()
    model.add(Embedding(total_words, sequence_len, input_length=sequence_len - 1))
    model.add(LSTM(100,return_sequences=True))
    model.add(LSTM(100))
    model.add(Dense(100,activation='relu'))
    model.add(Dense(total_words,activation='softmax'))
    model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
    model.summary()
    return model

model = create_model(seq_length, total_words)
model.summary()
cp=ModelCheckpoint('/content/drive/My Drive/NLP_Data/model_lstm_20.hdf5',
                   monitor='loss',verbose=1,save_best_only=True, period = 5)
```

Now that the data is ready, We start our sequential NN by adding neural network embeddings that are useful because they can reduce the dimensionality and meaningfully represent categories in the transformed space. Generally speaking, we use an embedding layer to compress the input feature space into a smaller one.

Then we add two stacked LSTM (Long Short Term Memory) layers with 100 units each. (LSTM Algorithm will be explained in the section later.)

Then comes the two fully connected or dense layers first layers have 100 units or neurons and the second dense layer which is our output layer have the number of units equal to the total words. As for every input, our model will predict the probability of every word in our total_words.
We have experimented with a couple of optimizers such as RMSprop, SGD, and Adam and in our case, Adam optimizer gave us the best results. As for the loss function, we have used categorical cross-entropy.
There is no test data set, we are modelling the entire training data to learn the probability of each word in a sequence.
We are ready to train our model, we have added a callback ModelCheckpoint to save the weights after every epoch.
Then finally we fit our model with 100 epochs.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 19, 20)            273300
_________________________________________________________________
lstm_2 (LSTM)                (None, 19, 100)           48400
_________________________________________________________________
lstm_3 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 13665)             1380165
=================================================================
Total params: 1,792,365
Trainable params: 1,792,365
Non-trainable params: 0
_________________________________________________________________
```


```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 19, 20)            273300
_________________________________________________________________
lstm_2 (LSTM)                (None, 19, 100)           48400
_________________________________________________________________
lstm_3 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 13665)             1380165
=================================================================
Total params: 1,792,365
Trainable params: 1,792,365
Non-trainable params: 0
```

# Results of training

Not “amazing”, but it should be fair enough for our test. Indeed, for a given sequence of word, there is no clear determinism in the word to be chosen. So we have to be careful not only pick-up the word with the biggest probability, but being able to choose another one with high probability too.

```
Epoch 1/100
6453/6453 [==============================] - 62s 10ms/step - loss: 8.0978 - accuracy: 0.0165
Epoch 2/100
6453/6453 [==============================] - 62s 10ms/step - loss: 7.7762 - accuracy: 0.0269
Epoch 3/100
6453/6453 [==============================] - 61s 10ms/step - loss: 7.5477 - accuracy: 0.0320
Epoch 4/100
6453/6453 [==============================] - 61s 10ms/step - loss: 7.3331 - accuracy: 0.0365
Epoch 5/100
6448/6453 [============================>.] - ETA: 0s - loss: 7.1282 - accuracy: 0.0416
Epoch 00005: loss improved from inf to 7.12815, saving model to /content/drive/My Drive/NLP_Data/model_lstm_20.hdf5

Epoch 99/100
6453/6453 [==============================] - 62s 10ms/step - loss: 2.4590 - accuracy: 0.5352
Epoch 100/100
6449/6453 [============================>.] - ETA: 0s - loss: 2.4526 - accuracy: 0.5352
Epoch 00100: loss improved from 2.49000 to 2.45264, saving model to /content/drive/My Drive/NLP_Data/model_lstm_20.hdf5
6453/6453 [==============================] - 62s 10ms/step - loss: 2.4526 - accuracy: 0.5352
```

# GENERATING PREDICTIONS

After training the model, we can enter the seed text and the number of words to be predicted. Since our aim is to create a model to predict the next word in a search option, we aim to generate only the next 1 or 2 words

```
def generate_text(seed_text, next_words, model, max_seq_len):
    for _ in range(next_words):
        token_list = t.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_seq_len-1, padding='pre')
        predicted = model.predict_classes(token_list, verbose=0)
        output_word = ''
        for word,index in t.word_index.items():
            if index == predicted:
                output_word = word
                break
        seed_text = seed_text + " " + output_word
    return seed_text.title()
```

# OUTPUTS

The output is generated by passing a seed text to the model along with the number of words to be predicted.

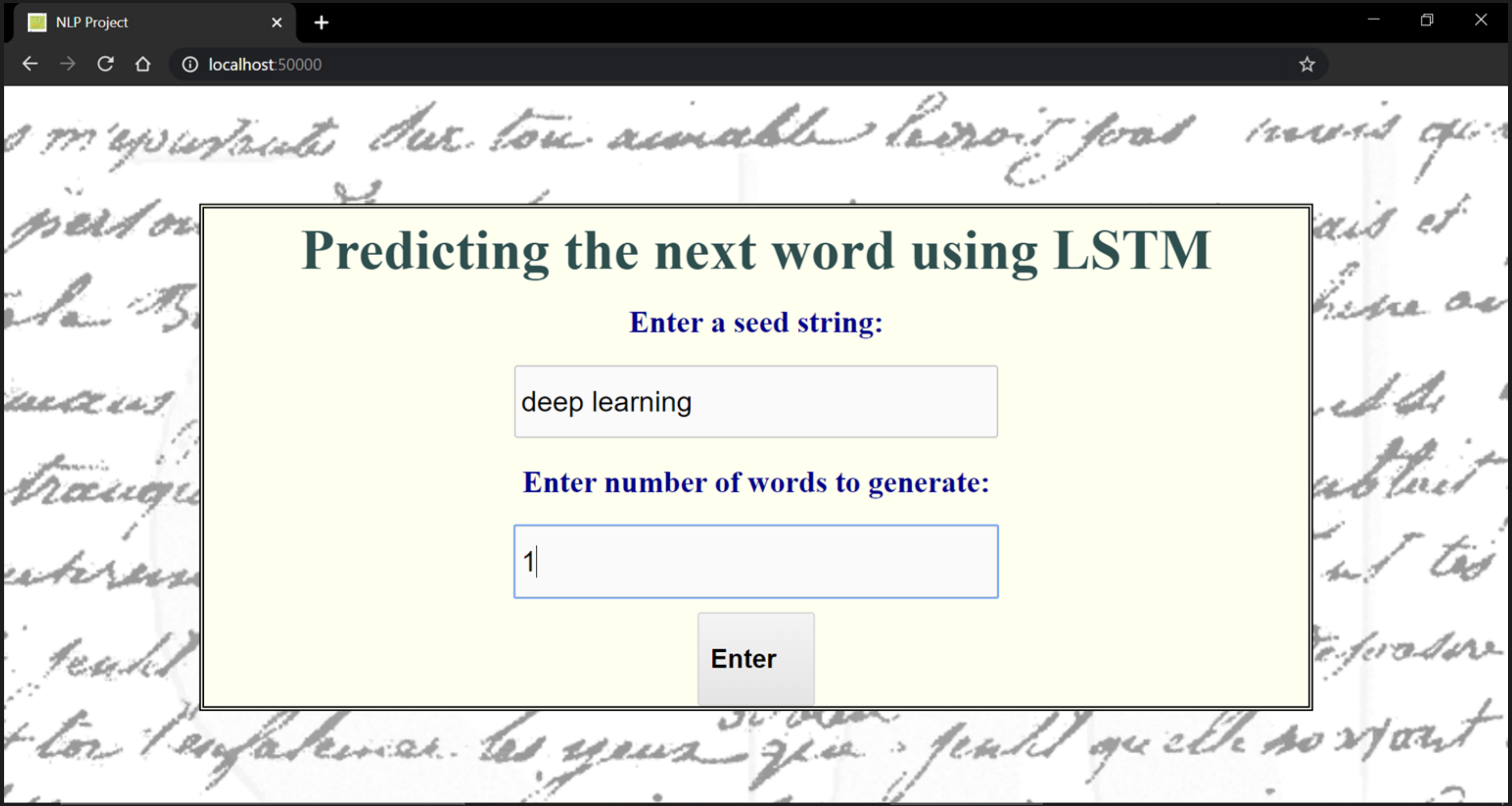
```
print(generate_text("deep learning", 1, model, 20))
print(generate_text("machine", 2, model, 20))
print(generate_text("what is your", 2, model, 20))
print(generate_text("neural", 2, model, 20))
```

```
deep learning techniques
machine learning techniques
what is your delivered examples
neural networks environments
```

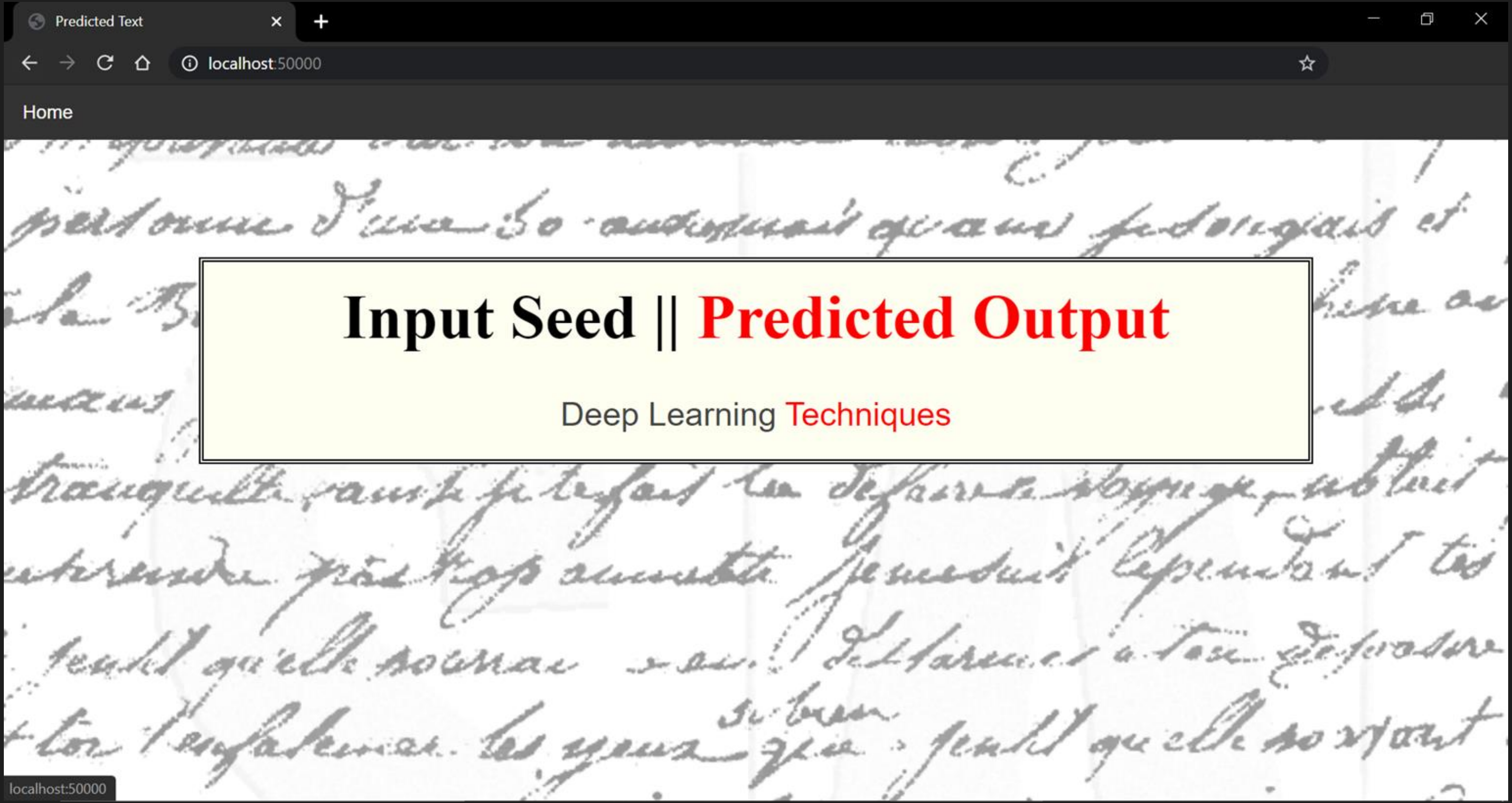
# DEPLOYMENT

We have tried to deploy our model locally, using Flask. We have created a frontend using HTML, CSS and JavaScript, and the backend of the app is being handled using Flask. As of now the model is being deployed locally, and we are attempting to deploy it on a cloud based platform.

# FRONT END - HOME PAGE



# FRONT END - PREDICTED PAGE



# CONCLUSION

- We were able to successfully generate/predict the next words of a given phrase or a word.
- The words generated by the model were proper English words, there were no misspelling and most of the sentences were realistic. This shows that the model has a good understanding of how letters are combined to form different words. Even though it is very obvious to do for a human, but for a computer model to give a reasonable performance at word formation is itself a huge feat.
- But after few words as we keep on increasing the number of next_words generation, sentences did not mean anything, as a whole. And also the grammar was not up to the mark for more number of next_words generation.
- We can probably have better results by increasing the training data, training epochs, more layers, more memory units to the layers, tuning the model to limit variance, etc. Applying these modifications could increase the capacity for the neural network to generate good phrases and less fuzzy.