Questions Auto-Complete System

Maymona Obaid
Department of computer engineering
Birzeit university
Ramallah,Palestine
maymonaobaid2001@gmail.com

Rama Abdalrahman
Department of computer engineering
Birzeit university
Ramallah,Palestine
ramaabdalrahman@gmail.com

Arwa Doha
Department of computer engineering
Birzeit university
Ramallah,Palestine
arwadoha@gmail.com

***Abstract*— This paper presents a project aimed at building a search bar autocomplete feature using deep learning and recurrent neural networks. The objective is to generate auto-complete predictions as the user types into the search bar. The project is divided into three parts: building and training the character-based recurrent neural network (RNN) model, constructing the frontend and backend components, and establishing communication through WebSockets. The model is trained on a dataset of questions obtained from Kaggle, which is preprocessed by converting text to lowercase, removing punctuation, null values, tabs, newlines, and multiple whitespaces. The dataset statistics are analyzed to determine an optimum question length. The questions are encoded using a custom dataset class in PyTorch, allowing for batching, shuffling, and sampling of data. The charRNN model architecture is implemented, and training is performed using the Adam optimizer and Cross Entropy loss. The trained model is evaluated based on the training and validation loss curves. Finally, the model is utilized for generating predictions by employing an inference process that involves encoding input characters, obtaining logits, and applying softmax to generate the probability distribution of the next character.**

Keywords—model, dataset, training, socket

I. INTRODUCTION

Nowadays digital age, autocomplete systems have been a crucial component in several applications, assisting users in finding relevant information quickly and efficiently. This project focuses on developing an AI-based autocomplete system using recurrent neural networks (RNN) with the goal of enhancing the user experience during search queries. The system utilizes Vue.js, FastAPI, and WebSockets to create a seamless and responsive interface between the user and the backend prediction model[1].

The project encompasses three key components. First, a character-level RNN model is trained using a carefully curated dataset of questions. By leveraging the power of deep learning, the model learns the patterns and relationships within the dataset to predict the next character in a given sequence[2]. The training process involves preprocessing the dataset, creating a custom PyTorch dataset class, and utilizing dataloaders to efficiently handle the training data.

Secondly, the project involves building a robust frontend and backend infrastructure. Vue.js, a versatile JavaScript framework, is employed to develop an intuitive and interactive user interface. The frontend comprises a search bar component that captures user input in real-time. This input is then transmitted to the backend, implemented using FastAPI, a modern and high-performance Python web framework. The backend receives the user input and feeds it to the trained RNN model for autocomplete prediction[3].

Lastly, WebSockets are employed to establish a bidirectional communication channel between the frontend and backend. Leveraging the real-time capabilities of WebSockets, the system ensures instantaneous updates and dynamic autocomplete suggestions as the user types into the search bar. This seamless communication allows for a responsive user interface, providing relevant suggestions as the user formulates their search query[4].

By integrating Vue.js, FastAPI, and WebSockets, this project demonstrates the implementation of an AI-based autocomplete system, enhancing the search experience for users. The trained RNN model, coupled with real-time updates, empowers users to find information efficiently and effortlessly. The project showcases the intersection of machine learning, web development, and real-time communication technologies, contributing to the advancement of intelligent search systems[5].

II. METHODOLOGY

Our research comprises three key components. Firstly, we undertake the construction and training of a character-based recurrent neural network model. Secondly, we engage in the development of both the frontend and backend components. Lastly, we investigate the establishment of communication between these components using WebSockets. This communication mechanism allows us to generate auto-complete predictions by inputting the user's text character by character into the RNN model as they begin typing in the search bar.

A. Dataset

In order to train our model for predicting user questions, we require a dataset consisting of questions. We have discovered a suitable dataset on Kaggle that is tailored for question-answering tasks. Although this dataset contains various attributes, we can easily extract and focus solely on the questions, which is the crucial element for our project.The questions within this dataset are relatively short and bear resemblance to the type of questions users often ask on search engines like Google. However, we intend to enhance the dataset in the near future by filtering it specifically for interview questions related to machine learning job roles. This targeted approach will further improve the relevance and applicability of our training data.

Content:

There are three question files, one for each year of students: S08, S09, and S10, as well as 690,000 words worth of cleaned text from Wikipedia that was used to generate the questions.

The "question_answer_pairs.txt" files contain both the questions and answers. The columns in this file are as follows:

- **ArticleTitle** is the name of the Wikipedia article from which questions and answers initially came.
- **Question** is the question.
- **Answer** is the answer.
- **DifficultyFromQuestioner** is the prescribed difficulty rating for the question as given to the question-writer.
- **DifficultyFromAnswerer** is a difficulty rating assigned by the individual who evaluated and answered the question, which may differ from the difficulty in field 4.
- **ArticleFile** is the name of the file with the relevant article

Figure 1: Related to our dataset

	ArticleTitle	Question	Answer	DifficultyFromQuestioner	DifficultyFromAnswerer	ArticleFile
0	Abraham_Lincoln	Was Abraham Lincoln the sixteenth President of...	yes	easy	easy	S08_set3_a4
1	Abraham_Lincoln	Was Abraham Lincoln the sixteenth President of...	Yes.	easy	easy	S08_set3_a4
2	Abraham_Lincoln	Did Lincoln sign the National Banking Act of 1...	yes	easy	medium	S08_set3_a4
3	Abraham_Lincoln	Did Lincoln sign the National Banking Act of 1...	Yes.	easy	easy	S08_set3_a4
4	Abraham_Lincoln	Did his mother die of pneumonia?	no	easy	medium	S08_set3_a4

Figure 2: Showing the head of the dataset

B. Processing the data

Preparing raw data for analysis and model training is known as data preprocessing, and it is a crucial stage in machine learning. Several preprocessing processes are carried out in this project. The dataset is first cleaned by eliminating null values, punctuation, tabs, newlines, and numerous whitespaces. To maintain uniformity, the text data is changed to lowercase.

```
questions_all = [
    "S08_What is the capital of France?",
    "S08_[Important] Please provide detailed instructions.",
    "S08_This question is not found.",
    "S08_What's your favorite color?",
    "S08_Why did the chicken cross the road?",
]
```

```
what is the capital of france
please provide detailed instructions
what's your favorite color
why did the chicken cross the road
```

Figure 3: Example of preprocessing in our case

Continuing with the data preprocessing stage, it is important to analyze the distribution of the questions' lengths. This provides insights into the characteristics of the dataset. The distribution plot of the questions' length reveals valuable information. On average, a question consists of approximately 50 characters. However, it is noteworthy that there are questions exceeding 200 characters in length. Considering typical user behavior, it is rare for users to search for such lengthy questions. Therefore, it is prudent to remove them from the dataset. This action is justifiable since the average question length is only 50 characters.To do this, we use a technique to determine the ideal length by computing the distribution's mean and standard deviation. We calculate the length that is one standard deviation away from the mean by adding the mean and standard deviation. Questions that are longer than this ideal length are then removed. By using this strategy, we are able to keep the majority of the dataset—questions that are within one standard deviation of the mean—while removing outliers that might not reflect usual user behavior, most of the questions have lengths between roughly 20 and 80.

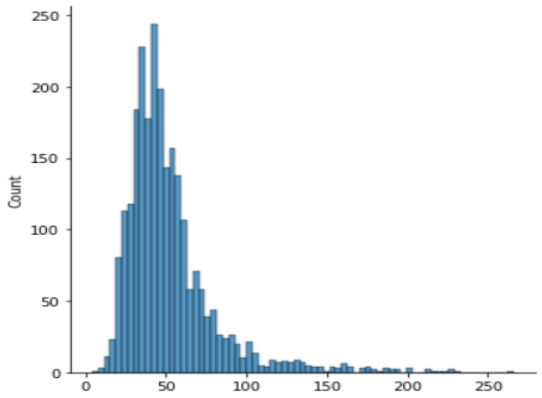

Figure 4: Questions Length distribution

```
mean = np.mean(q_lengths)
std = np.std(q_lengths)
print(f"mean: {mean}, std: {std}")
```

```
mean: 52.3442088091354, std: 29.053359020555256
```

Figure 5: Mean and standard deviation

```
optimum_length = int(mean + std)
print(f"optimum length: {optimum_length}")
```

```
optimum length: 81
```

Figure 6: optimum length

C. Dataset preparing

Shuffling the data is an essential initial step. Its purpose is to counteract any inherent biases or patterns that may exist in the original order of the questions, thereby minimizing their influence on subsequent analysis or modeling tasks. The process involves arranging the questions in a random order, effectively disrupting any underlying patterns or

dependencies that may be present in the dataset. This step holds particular significance when employing machine learning algorithms, as they have the potential to learn and rely on sequential patterns within the data. By shuffling the data, we ensure a more unbiased and robust foundation for further analysis and modeling tasks.

We implemented a code to create a dataset and dataloader for our question-based task. The purpose of this code was to prepare the data for training and evaluation. To begin with, we created a QuestionsDataset class, which inherits from the Dataset class provided by PyTorch, this approach offers greater data control and code modularity, while also enabling effortless creation of a PyTorch Dataloader for batching, shuffling, and data sampling.

First, we load the questions and the vocabulary. We also initialized several parameters including the initiation of sentences token (sos_token), and the ending of sentences token (eos_token), and whether the batch dimension should be the first dimension (batch_first), so the model could know when to start and end the sentence.

We also created two dictionaries, int2char and char2int, to map characters to indices and vice versa. The int2char dictionary was initialized with the start of sentence and end of sentence tokens.

We then updated this dictionary by assigning indices to each character in the given vocabulary. The char2int dictionary was created by reversing the key-value pairs of the int2char dictionary.

In order to encode and pad the questions, we applied the encode_question method. This method takes a question as input and encodes it as character indices using the char2int dictionary. Each character in the question is appended first, then the start of the sentence token, and finally the end of the sentence token. A one-hot encoded tensor is created from the index sequence that results. Then, this tensor is given back.

```
tensor([[[1., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [1., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.]],

        [[1., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [1., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.],
         [1., 0., 0.,  ..., 0., 0., 0.]]])
```

Figure 7: one-hot encoded tensor

```
Input: [h, e, l, l, o], here is the step-by-step transformation:
[h, e, l, l, o] is transformed to
['<start>', h, e, l, l, o, '<end>']
by adding the start and end tokens.
['<start>', h, e, l, l, o, '<end>'] is transformed to
[0, 2, 3, 4, 4, 5, 1]
by replacing each character with its corresponding index in the vocabulary.

[0, 2, 3, 4, 4, 5, 1] is transformed to
[[1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0],
[0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 1],
[0, 1, 0, 0, 0, 0]]
by performing one-hot encoding.
```

Figure 8: Example of how encoding will be performed

The questions are encoded and padded using the pad_sequence function from PyTorch, which ensures that all sequences in the dataset have the same length. The pad_sequence function takes a list of encoded questions as input and pads them with zeros to match the length of the longest question. The QuestionsDataset class also returns the length of the questions dataset, and retrieves a specific encoded and padded question based on its index.

In order to create the vocabulary, we extracted all unique characters from the questions_short variable, which presumably contains a collection of questions. These characters were then sorted and assigned to the vocab variable, using some methods.

Finally, we specified the batch-first setting (BATCH_FIRST = True) and set the batch size to 64 (BATCH_SIZE = 64). We also determined the device to be used for computation based on the availability of a CUDA-enabled GPU (torch.device("cuda")) or the CPU (torch.device("cpu")).

D.Train validation tests

We do a train-validation split to separate our data into sets for training and validation while also making it easier to load the data. This is achieved by building a language out of the special characters present in the queries. Additionally, it specifies variables like the batch size, start and end tokens (which must be different from the vocabulary), and the computing device to be used (GPU if available, CPU otherwise).

A portion of the data is allocated for validation, while the remaining portion is used for training. The validation set makes up 10% of the total data in this instance, and the training set makes up the remaining 90%. For validation, a printed count of the questions assigned to each set is provided.

```
val_percent = 0.1
n_val = int(val_percent * len(questions_short))
n_train = len(questions_short) - n_val
print(f"n_train: {n_train}, n_val: {n_val}")
```

```
n_train: 1977, n_val: 219
```

Figure 9: training and validation sets

Subsequently, we create datasets for training and validation using the QuestionsDataset class defined earlier. These datasets are then used to construct dataloaders using PyTorch's DataLoader constructor. The dataloaders enable efficient batching, sampling, and shuffling of the data during training and validation.

With the data prepared and the dataloaders in place, we are now ready to proceed with building the model.

E. RNN-Model

The next step to achieve the process is to define the model. The model used in our project operates by accepting an input character at each time step and generating a probability distribution for the subsequent appropriate character. The diagram below illustrates this concept:

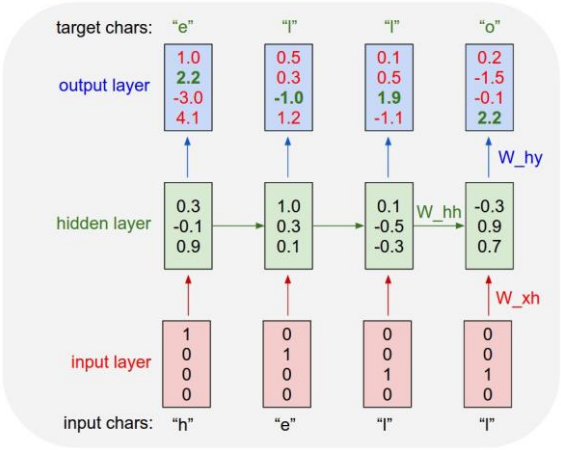

Figure 10: RNN concept

The idea behind the model is to feed an encoded question into a recurrent neural network (RNN), which produces a hidden state at each time step. These hidden states are then passed through a fully connected network to obtain logits. Subsequently, we calculate the Cross Entropy Loss by comparing the logits with the target values. To achieve this, we compute the loss for each time step and aggregate them across all time steps to obtain the total loss.

This total loss is subsequently used for backpropagation throughout the entire network.This model takes several parameters: `VOCAB_SIZE`, `HIDDEN_SIZE`, `N_LAYERS`, `P_DROPOUT`, and `batch_first`. The `VOCAB_SIZE` represents the vocabulary size while `HIDDEN-SIZE` determines rhe hidden state size. Additionally, `N_LAYERS` specifies the number of layers in the LSTM, `P_DROPOUT` controls the dropout probability, and `batch_first` determines whether the batch dimension is the first dimension.

The 'charRNN' model has an LSTM layer, a dropout layer, and a fully connected (linear) layer after that. The LSTM layer generates output sequences and updated hidden states from input sequences and hidden states. In order to avoid overfitting, the output of the LSTM layer is subsequently flattened and applied. The final output is obtained by passing the flattened output through the linear layer. The 'BATCH_SIZE' and 'device' parameters are inputs to the 'init_hidden' method, which initializes the model's hidden state. It then moves the initial hidden state tensor, which is made up entirely of zeros, to the designated device.The parameters 'VOCAB_SIZE', 'HIDDEN_SIZE', 'N_LAYERS', 'P_DROPOUT', and 'BATCH_FIRST' are used to instantiate the model. The 'charRNN' model was developed by. Consequently, we flatten the output along the batch dimension to combine all the batches.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 128)               95232
_________________________________________________________________
dense_2 (Dense)              (None, 57)                7353
_________________________________________________________________
activation_2 (Activation)    (None, 57)                0
=================================================================
Total params: 102,585
Trainable params: 102,585
Non-trainable params: 0
```

Figure 11: Network Architecture

By integrating this information with the previously paraphrased content, we have provided a comprehensive explanation of the charRNN model, its architecture, and its functioning. This paves the way for the subsequent steps in the training process.

III. TRAINING THE MODEL

we have defined a three-layer LSTM network that is arranged vertically, with each layer having a hidden state size of 512 and a dropout probability of 0.4.

We used the Adam optimizer with a preset learning rate of 1e-3, and Cross Entropy was used as the loss function. The training and validation process is executed for a total of 100 epochs, and the model is saved after every 10 epochs. These hyperparameters can be adjusted based on the performance of the model. Additionally, there are a few important points to consider:

1. The hidden states are passed between batches, meaning the final state of one batch becomes the initial state of the next batch.
2. To address the issue of exploding gradients that arise in RNNs during backpropagation through time, we apply gradient clipping. This involves limiting the gradients with large magnitudes to a specific threshold.
3. Once training is complete, we can generate a graph showing the training and validation curves.

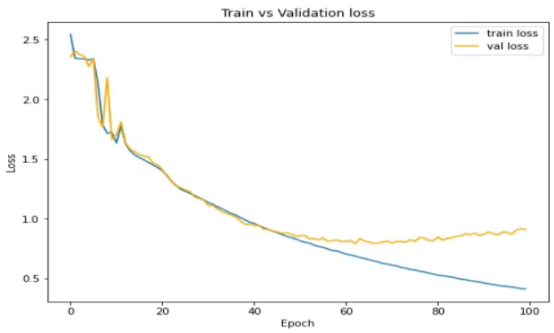

Figure 12: training and validation curve

Initially, the plot appears promising, but upon closer observation, we notice that the train and validation loss values are in close proximity. This situation is typically considered unfavorable as it suggests that the model struggles to grasp the underlying patterns and representations within the data. This outcome is expected since we are utilizing a relatively straightforward charRNN model. Although employing transformer-based models might enhance the performance, we will proceed with the current model for the time being.

IV. Generating Questions

After completing the training of our model, it is necessary to comprehend the process of inference, which entails predicting the question based on the user's input. The approach involves encoding an input character and passing it through the network to obtain logits as the output. These logits are then subjected to the softmax function, resulting in a probability distribution for the next character. Although we could simply select the highest probability character from this distribution, doing so would lead to overfitting. Therefore, we select the top k characters from the distribution and randomly choose one of them as the next character.

The aforementioned process is demonstrated in the predict_next_char method within the GenerateText class. The predict_next_char method accepts an input character and the hidden state as inputs, enabling the prediction of the subsequent appropriate character.

Our objective is to predict the question based on the user's context. The context, also referred to as the prime, consists of a series of initial characters. The approach involves feeding these initial characters to the model one by one, thereby building the hidden state. Subsequently, using the hidden state and the last character in the context, the predict_next_char method is employed to predict the next character. This predicted character is then used as input to predict the subsequent character. This iterative process continues until the end token is encountered, indicating the completion of the question. The generate_text method exemplifies this process.

The following examples provide further illustration:

```
text_generator.generate_text('when di')
# when did the election of the lowate of the korean lengthened

text_generator.generate_text('who i')
# who is the modern made of chinese

text_generator.generate_text('is it')
# is it true that indonesia has a political citizen
```

Figure 13: charRNN-testing

V. Preparing the Frontend

As shown in the following graphic, our frontend will be simple and consist of merely a search bar with the user input presented on the front and the auto-complete displayed behind it with a reduced opacity:
The aim is to stack two span elements one on top of the other within the same div container, which can be accomplished by specifying the relative position of the div container and the absolute position of the span components. To set the span elements at the same position as the div container, we are essentially positioning them relative to the div container.

VI. Communication through WebSockets

The idea is that the text should be provided as input to our charRNN model at the backend as soon as the user begins typing in the browser in order to produce predictions. The predictions will then be transmitted back to the frontend and shown in the auto-complete placeholder behind the search bar's user input. WebSockets will be perfect because we want to create a real-time, two-way connection between the frontend and the backend. Let's create the frontend and backend and set up WebSockets for communication.

VII. Building the Backend

To serve the requests on the backend, we simply need to build an API wrapper around our charRNN model. The WebSocket API is defined here using FastAPI.

First, we define, load, and switch to evaluation mode the charRNN model that has already been trained. In the following section, we construct a route that employs the WebSocket protocol and a function called predict_question that is run each time a request encounters this route.

The predict_question function is an asynchronous function, which simply implies that it does additional tasks before returning to the work at hand rather than waiting for one to complete. The await keyword instructs not to wait for the completion of a task, but the async keyword defines an asynchronous function.

VIII. Building the Frontend

In order to send the text to the backend as soon as the user begins typing, we need a technique to detect user input on the frontend. For this, we can make use of event handlers and Vue directives. The handler function is called by the v-on directive (also denoted by the @ symbol) in response to certain events occurring on an element.

The WebSocket object is first created in the mounted lifecycle hook. In essence, lifecycle hooks are functions that are called at various points during the construction of a component. Once the component is rendered into the DOM, the mounted lifecycle hook is called, which is an ideal time to create the WebSocket object and establish a connection to the backend.

IX. Results

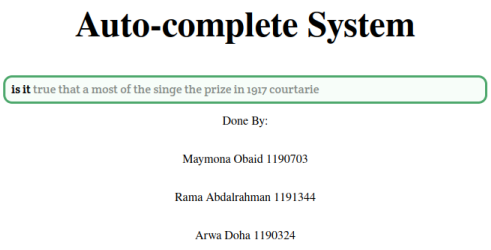

Figure 14: testing the system on browser

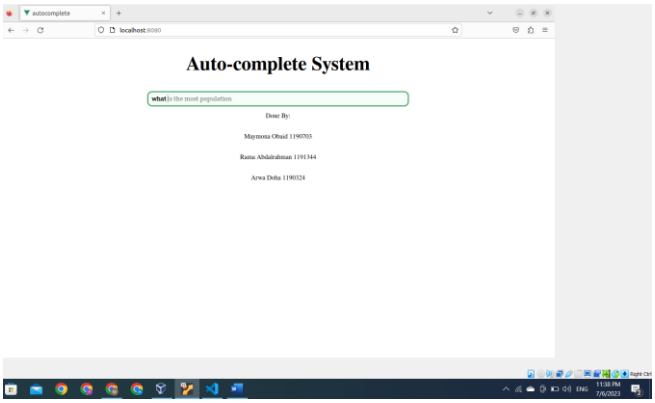

Figure 15: testing the system on browser

X. Conclusion and Possible Improvement

In conclusion, this paper presented a project that aimed to build an autocomplete system using deep learning. The project was divided into three main parts: building and training the character-based recurrent neural network (RNN) model, constructing the frontend and backend components, and establishing communication through WebSockets.

The dataset used for training the model consisted of questions obtained from Kaggle, which were preprocessed by converting text to lowercase, removing punctuation, null values, and multiple whitespaces. The dataset statistics were analyzed to determine an optimum question length, and the questions were encoded using a custom dataset class in PyTorch.

The charRNN model architecture was implemented, and training was performed using the Adam optimizer and Cross Entropy loss. The trained model was evaluated based on the training and validation loss curves.
WebSockets were utilized to establish real-time bidirectional communication between the frontend and backend, enabling instantaneous updates and dynamic autocomplete suggestions as the user typed into the search bar.

The project successfully demonstrated the integration of deep learning, web development, and real-time communication technologies to create an AI-based autocomplete system. The trained RNN model, coupled with the responsive user interface, enhanced the search experience for users by providing relevant suggestions as they formulated their search queries.

Further improvements could be made by exploring more advanced architectures, such as transformer-based models. Additionally, expanding the dataset with more targeted and relevant questions could enhance the performance and relevance of the autocomplete predictions.

Overall, this project showcased the potential of deep learning and web technologies in creating intelligent search systems, and it provided valuable insights into the implementation and integration of various components to achieve the desired functionality.

References

[1] Burgueño, L. (2021, March 16). An NLP-based architecture for the autocompletion of partial domain models. Modeling Languages. https://modeling-languages.com/nlp-architecture-model-autocompletion-domain/ (Accessed: June 18, 2023).

[2] Introduction to Recurrent Neural Network - GeeksforGeeks. (2018, October 3). GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/ (Accessed: June 22, 2023).

[3] Next Word Prediction with Deep Learning in NLP - GeeksforGeeks. (2023, June 30). GeeksforGeeks. https://www.geeksforgeeks.org/next-word-prediction-with-deep-learning-in-nlp/ (Accessed: June 27, 2023).

[4] M, S. (2021, August 17). Predict the next word of your text using Long Short Term Memory (LSTM). Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/08/predict-the-next-word-of-your-text-using-long-short-term-memory-lstm/ (Accessed: July 1, 2023).

[5] Kumar, A. (2023, May 30). NLP Pre-trained Models: Concepts, Examples - Data Analytics. Data Analytics. https://vitalflux.com/nlp-pre-trained-models-explained-with-examples/ (Accessed: July 1, 2023).