Predictive Text Model for Recipes

CSCI 49900 - 03 (Advanced Applications: A Capstone for Majors)

Group Four: Naima Mamataz, Panchita Lopez-Li, Orione Brown, Sol Cruz

June 03 - August 17, 2022

GITHUB LINK

https://github.com/panchitalopez/predictive-recipes

INTRODUCTION

Our group set out to write a predictive text model for recipes that would combine our knowledge of computer science principles into an application that would predict the instructions of a typed recipe. The overarching topics that we focused on for our model were Machine Learning and Artificial Intelligence as well as some focus on Data Science. Focusing on ML/AI, we particularly honed into Natural Language Processing (NLP), where we looked into Recurrent Neural Networks (RNNs). Through the exploration of different RNN architectures, we came across the Long Short-Term Memory (LSTM) architecture that performs the predictive text using a sequence of words as inputs and processing the output as a matrix of probability from each word. In this paper, we will discuss our research, website application, data configuration, and our two predictive text models- including the LSTM model and NodeJS model.

RESEARCH

Focusing on the research we did, we provided a general overview of the overarching topics we focused on within Machine Learning and Artificial Intelligence.

Natural Language Processing (NLP) is a component of AI, where it has the ability of a computer program to understand human language as it is written or spoken. Recurrent Neural Networks (RNNs) are neural network architectures that are heavily used within NLP. They are upgraded neural networks where connections between nodes are treated as sequential signals. It heavily relies on sequence modeling. With RNNs, it is the same concept as when a person speaks, words take meaning based on previous words, and sentences take meaning based on previous sentences. A specific RNN architecture we used throughout our project is called LSTM which is an acronym for Long Short-Term Memory. LSTMs can handle long-term dependencies and retain long-term information in the network. This neural network is used within text prediction as it can take a sequence of words as the input and output and convert it as a matrix probability of each word from the dictionary to be the next of the given sequence. This concept will be unfolded on a more technical level throughout this paper.

Since Panchita was dedicated to the web design layout for the first month, she used the next couple of weeks to catch up on the backend implementation and did research on Natural Language Processing and forecasting which is a technique used in data science, as well as the Tensorflow and Keras library. The Tensorflow and Keras documentation were very dense and it was challenging to distinguish which aspects would be relevant to our project since there were many different options, most being a case to determine if they were applicable via trial and error. Relevant examples of this were the choice of optimizers, models, activation layers, a consideration of what to monitor such as loss or gain, and several loss functions. She also found several APIs that we could use to scrap recipe data to simplify the pre-processing, since most recipes are quite short if taken from an official recipe website. Some of the APIs being:

Spoonacular, Tasty, My Cookbook.io, Recipe Search, and Diet. We had contemplated using the Kaggle dataset we originally found, however, it contained over two million values - most being cooking jargon, which was irrelevant to our project - even after it had gone through preprocessing and filtering, it was still at 1.5+ million values. Panchita also considered using Google Cloud APIs to train our model but realized how potentially large our dataset could be since it was still undecided at the time. Another factor to consider was how quickly we could exhaust the free $300 credit upon signup, thus she decided it was not a viable resource to move forward with.

While the group was still figuring out our dataset situation, Naima decided to focus on how we can tokenize our future dataset to have an idea of how we can go about configuring it once we did have our dataset confirmed. Through research, she learned that with tokenization although it can be fulfilled through the Keras library, there are other components to it that needed to be considered such as saving the updated tokenized file using pickle, how each unique word needs a numeric expression, how we have to account for each token ID being incrementally passed, and how we have to append the input and output words to prepare for the actual predictive algorithm. Naima did more research on the LSTM architecture beyond its documentation scope where she obtained an understanding of how the LSTM architecture actually works from the logical gates it uses.

From research, we learned that the LSTM architecture has three main gates: the forget gate, the input gate, and the output gate. The forget gate is one of the main properties where it is responsible for memorizing and recognizing the information coming inside the network and also discarding the information which is not required for the network to learn the data and predictions. It helps in deciding whether the information can pass through the layers of the network. There are also two types of input it expects from the network, one of them being the information from the previous layers and the other being the information from the presentation layer. The input gate helps in deciding the importance of the information by updating the cell state and helps the forget function in eliminating the unimportant information and instead take in other layers to learn the information from that are actually considered important for making predictions. The output gate is basically the last gate of the circuit that helps in deciding the next hidden state of the network.

WEBSITE APPLICATION

Since we decided to create a website application that would showcase our predictive model, Naima initiated creating the basic layout of the React app. Although she had previous experience in React, she wanted to familiarize herself again with the basic documentation to recollect how the structure is created within React and learned about specifics like props, components, and hooks. She then passed on the front-end configuration to Panchita.

Initially, Panchita used the first month to visualize and research what libraries we could use throughout the course of the project. She also created the web application of the predictive text model. The languages used were React, HTML, and CSS. Ideally, the website would contain three aspects: a header, a section to put the list of ingredients, and a section where the recipe steps from the predictive text model could be inputted. She started a basic React App from

scratch and implemented the header and the ingredient list form with a button that added new ingredients to the submitted ingredient list. There are various ways to do this, the most common being a class or a useState hook. The useState hook allows the use of certain features of a class instead of writing classes, thus being the more efficient option and it was also created to simplify the process of classes. She included a sample JSON file with mock data values to ensure the list was able to handle new submissions and compute them to the list. After trying to implement the recipe step box in a text editor format and using classes to configure it differently, she decided to leave it while she caught up to work on the backend and later passed it off to Orion to add to it and connect it to the backend.

PRE-PROCESSING DATA

After our first week of research, our group found it necessary to focus on data, specifically the data being fed into the model. When Orion joined this group it was already decided that we would be using the "RecipeNLG cooking recipe dataset". After doing some research, he found documentation on a similar predictive text model, detailing the steps and precautions taken for a successful and accurate model. Many of the findings were related to data manipulation which he took an interest in. From the documentation, he found that before anything was done to the data it needed to be "pre-processed" in order for it to flow into the model smoothly. This included removing unwanted non-ascii characters, punctuation, and converting abbreviations in the text to the actual text. For example, many recipes that had fractional measurements which are fed into the model would gain their own tokenized value and associations within the model. This would confuse the model so from this research he took precautions in our own dataset. First, he removed all non-ascii characters and converted fractions back into plain text form. An obstacle unique to our dataset was that many of the data entries were pulled from a variety of websites which resulted in a large range of data. After manually going through some entries Orion identified some words or phrases that might be problematic for the model, much like the documentation he read. Instructions such as "mix-all ingredients together" and "follow box instructions" he deemed problematic because (1) the sheer number of these instructions would skew the model to almost always choose this instruction over a better fit, due to the high coefficient of relation (2) instructions like this were vague and would confuse our model's "understanding" of the English language. After removing these phrases, he found that many recipes also had characters such as asterisks to denote alternative ingredients. He removed these too since our computing power was relatively low compared to a predictive text model like Google's so it was in our best interest to skew the data as much as possible to give a favorable result. After identifying possible problems with the dataset he began manually picking out a subset of the data to feed into our model.

While the group initially decided to create our own dataset using TensorFlow, we realized that it wasn't the right step to go about it because of how overwhelming it was to account for each word that our model would require. During this time, our group had a mindset that the dataset was the most important thing to get us started on building our model, which we, later on, realized that it was not necessary. Prior to that realization, Naima researched specific pre-existing datasets and came across a specific recipe-based dataset from Kaggle. It seemed to be a well-rounded dataset at the time, as it included all the ingredient jargon that we thought was required for our predictive model. However, the dataset had over two million entries that Orion initiated to condense and

Naima tried to preprocess a bit further but it was still too vast for the model that we wanted to create.

At this point, our model was still in the testing stages so Orion reviewed the data and tried to find any improvements he could make. Towards the middle of the summer session, however, with the acceptance of API usage, our team pivoted to use more APIs to lighten the load. Although this was a setback, he was still able to utilize the knowledge he gained from processing the first dataset. The API we decided to use to replace our dataset was the “Spoonacular” API which had several useful endpoints and many optional parameters for API calls. He focused on the complex search endpoint which had many useful parameters such as a plain English search query and an option for included ingredients as well as the analyzed instructions endpoint which took in a unique ID related to a recipe and returned an array of steps. This API combined many of the features that he wanted to implement and combined them into a single API. After calling the complex search API endpoint, he extracted the ID related to each recipe in the response and passed it to the analyzed instructions endpoint, looping through the response to get all the steps for that recipe. Orion made it so the number of recipes returned was variable so that we could test the models’ performance with different sizes as well as varying datasets. He then incorporated these calls into our front-end because he figured it would be more organized to get user input, convert it into usable data, and feed it into our model.

———————— LSTM MODEL ————————

After deciding that none of the datasets including the APIs that deploy dataset values were suitable, Panchita and Naima decided to build the predictive model using a single recipe online with just the steps and made it into a text file so that we could just use it temporarily for our first trial run and to get a sense if any results are deliverable. We briefly pre-processed the small dataset where we used the nltk library that is used to tokenize the sentences into separate words. After that, the built-in maketrans, split(), lower(), and string.punctuation from Python made quick work of the preprocessing. The separated words were mapped to strings, split by whitespace, punctuation was removed, and all words were lowercase.

TOKENIZING

Initially, we wanted to form an algorithm without the use of any libraries; however, we were running into too many errors to debug. Without the use of a library, we nearly had to write all of the functions on our own which took too much time. So using what Naima learned from her previous research, she utilized the same logic with this dataset using Keras libraries. The specific methodology she used was implementing the Keras Tokenizer function to tokenize each word character, which basically separates each word individually. Then we implemented the fit_on_text tokenizer class function to update the vocabulary index where we had to use the pickle library, to save the updated tokenized file. We also had to include the text_to_sequences and word_index from the Keras Library to convert the text to a sequence of integers and incrementally account for the token IDs. We, later on, classified our independent and dependent values, appended them, then converted them into array and class vectors to construct the binary class matrix.

BUILDING LSTM MODEL LAYERS

There are three types of models in Tensorflow: the sequential model, the functional model, and a customizable model. For our purposes, we decided on the sequential model because the layers in our data set did not contain multiple inputs and outputs but rather a single input and output tensor. As for the layers, there are different categories of layers that can be built upon each other: the dense layer, the LSTM layer, and the embedding layer. Each layer has a different function -- the dense layer is the most common layer and it is a deeply connected neural network. Preceding the dense layer is the LSTM layer which learns long-term dependencies between the time steps in the time series and sequence data. Lastly, the embedding layer helps to convert each word into a fixed vector of defined size. This helps to represent the words in a better way with reduced dimensions. Altogether, these layers will aid the information through flow states, where it can selectively remember and forget certain words and aid in improving our future predictions.

TRAINING THE MODEL

After building the layers of our model, we then focused on training it. The methodology we used to train our data was saving the model file using the Keras library for the model checkpoint. We also compiled the model by passing it through the loss and optimizer function where our choice of optimizer was important to determine as it would affect the loss reduction rate and also determined how quickly we would get results. We decided to go with the Adam optimizer because it updated the data iteratively based on the training data. For fitting the model, we passed over the independent and dependent variables over an epoch size and batch size determined by the user. The results we got from the epochs reflected the full pass over our training dataset. The epochs indicated the number of iterations over the input and output values of the text. When one epoch ends or finishes, our model runs the training data through all the nodes in the network and updates the optimal loss value. The loss value is the mean squared error for regression and the log loss for classification, which we intended to minimize during the course of training our model because the lower our loss was, the more accurate our predictions would be. If there was a case within the course of running through our epochs where it resulted in a higher loss value than the previous epoch result then it will call back the previous best model (seen in our script as “predicting_next.h5”) and forget the unimproved one and start from there again to continue training our model towards more accuracy. We also implemented how long it takes for each epoch to iterate over the course of completion.

PREDICTION FOR THE MODEL

After the model and tokenizer are loaded in and opened, the model is ready for the prediction stage. However, we must first obtain the words the user wants to be for the basis of the next word prediction. The user is asked to input after how many words they would want the prediction to occur. This is crucial to the predictive algorithm because it determines the number of words the function should take into consideration. For example, if the user wants the sixth word to be the prediction and enters 15 words as the cooking step input, it will only take into account the last five words of the input. After the step is entered, it is split by whitespace which allows the phrase to become individual words. The predictive model is then called on with the three parameters:

the model (the “predicting_next.h5” file), the tokenizer, and the words that would be inputted by the user to set up the prediction. Starting from the first position of the words, the word is tokenized and transformed from text to a series of sequences. Then, the beginning of the sequence is padded to guarantee that all sequences are of equivalent length, along with a maximum size being set to the number of words that the user initially had inputted into the program. We then used the model from earlier that had gone through the process of being fitted and trained for this moment: to make a prediction. The indices from the maximum value of the specified axis are found and then returned. The reasoning for this is that the highest matching value of the axis is the highest likelihood to be the next predicted word. The index position from the dictionary that correlates with the word of that index from the dataset is then found and tokenized. This is the next predicted word. The words that were initially input by the user are then printed out with the predicted word.

———————— NODE JS MODEL ————————

BUILDING THE ML MODEL

Processing data: The Data file should not have any punctuations. During the scrapping stage the data is removed of all punctuations and redundant white spaces. So the primary data is just a sequence of words.

We read the sequence of words from the data file and convert it into a list of words in python. The tokenizer class helps us convert the words into number. It maps each distinct words to a distinct number. We will later feed these numbers to the AI model and use it predict the next word. That’s why we need to save the mapping (the tokenizer object) locally.

Buiding the model: Next we need to build the training data set. We take two consecutive word as input and the third word as the result. We will first use the tokenizer to convert our whole list of words (“data”) into numbers that we mapped earlier. Then from this list of numbers we take every 2 number to the list X and the third number to list Y.

The list Y , which represent the output for our input X has to be converted into a binary matrix to work with our model. so if the Y was [1,3] and we had only 4 distinct word in our data set then it would be converted into- [[0,1,0,0],[0,0,0,1]] matrix.

The function to_categorical() performs this task. The input and output data building process is illustrated below:-

DATA:	this	is	a	cat	and	this	is	a	dog
converted to number:	0	1	2	3	4	0	1	2	5

X = [[0,1], [1,2], [2,3], [3,4], [4,0], [0,1], [1,2]]
Y =[2 , 3 , 4, , 0 , 1 , 2 , 5]

Then Y would be converted into categorical matrix.

Then we build a sequential model in keras. We add multiple layers one by one in our model. The first layer is the Embedding layer.This layer is used to do the task of word embedding. In an embedding, words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space.The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used.

We added a LSTM layer to our architecture. It has 1000 units, of dimensionality of the output space and we made sure that it return the sequences as true. This is to ensure that we can pass it through another LSTM layer. For the next LSTM layer, we also pass it through another 1000 units but we don't need to specify return sequence as it is false by default. We will pass this through a hidden layer with 1000 node units using the dense layer function with relu set as the activation. The rectified linear activation ("relu")function is a simple calculation that returns the value provided as input directly, or the value 0.0 if the input is 0.0 or less.

Finally, we pass it through an output layer with the specified vocab size and a softmax activation. The softmax activation ensures that we receive a bunch of probabilities for the outputs equal to the vocabulary size. So it will output a vector of floating point number for all the distinct words in our dataset. That number represents the probability of a word as our prediction. Next we build and save the model.

Predicting the next word: We take 2 words as input and remove all redundant whitespaces. Then using the saved tokenizer from the building phase we convert the words into number (sequences). If we feed this input to our trained model it will produce a vector of length *vocabulary_size* of the tokenizer. From this vector of probabilities we take the highest 5 probabilities and check their index number. The index number is actually the number representation of the words. So using the index and tokenizer we get the actual words. The model can not produce a word that it has not seen before.

BUILDING THE APIs

We used node-express to build our APIs. The APIs use nodeJs' built in "child_process" module to create a new process and run the python predictor program . The spawn() method runs the prediction python script and communicates with it via stdout and stdin. When the predictor.py first loads the trained model from disk. This takes approximately 1 hour minimum. The API function only creates the predictor.py process once in its lifetime. So when the first call to the prediction API is made it takes a while to get the prediction because the scripts takes time to load the model.

Once the model is loaded the next prediction APIs can use the already created process to get the prediction. We also have a queue system of the requests. When the requests comes we queue them in an array because the communication between the predictor process and nodeJS is asynchronous.

Using this API we can create a frontend that predicts the next words as we type. Also, we have a scrapper API that adds data to our data bank from a given URL.

RESULTS

We were successful in producing two predictive text models with accurate results, one in Javascript and Python and the other as a Python script. The Python script allows the user to input the number of words the prediction should be, the size of the LSTM layer, the number of epochs, and batch size. The user can experiment to see what inputs perform the strongest due to the accuracy of the prediction returned. This is also largely dependent on the parameters that are inputted by the user and the size of the dataset. When a larger dataset of around 608 words with ten recipes from different websites was used, the prediction was less accurate due to the wider variety of vocabulary and words. When a small dataset of one recipe with around 30 words was put through, the predictive word returned was almost completely accurate. Thus, with the appropriate parameters of epochs, batch size, and LSTM size, a more accurate prediction was able to be produced. If enough epochs were run with an ideal batch size, the loss reduction would be greater and result in a significantly smaller loss rate. This results in a significantly more accurate result for the word that is returned.

CONCLUSION

The subjects of Natural Language Processing and Recurrent Neural Networks were fascinating yet challenging for us to apply to our predictive text model. Overall, our scope on this project was too open-ended, as we had to consider the best options to use with our models and weigh how and where the dataset would be incorporated in our project, depending on the dimensions we wanted the dataset to be. This, along with the timespan of the course, resulted in more general research rather than specific implementations that we could have tried out and applied to our project. Throughout the twelve weeks, we encountered many hurdles, learned new information, and resulted in two successful models.

FUTURE WORK

Orion found a lot of the findings relating to data science to be really interesting as previously when researching machine learning and artificial intelligence, it was something he either overlooked or paid no attention to at all. Throughout the length of this project, he was able to pre-process and sanitize data as well as skew the data for more desirable results. Normally skewing data is frowned upon so working with data in this context was a very new and fun perspective.

For future work, Naima would still like to work on implementing the LSTM algorithm without the help of libraries to gain a better scope of how the algorithm is constructed without them. She learned an immense amount throughout the timeline of our project on Machine Learning and Artificial Intelligence and how complicated it is to configure into a functioning product. Panchita also would like to continue with the backend by allowing the user to type in the name of the file that will serve as the user's data set. She'd also like to connect the frontend and backend together using Django. The front-end could be improved as well with an image for recipes, a title, and an

ordering system to add your recipes in one place. One final aspect she wanted to improve on was to add to the preprocessing in order to return precise results, since the predictive model may take fractions or other words that pertained to a specific recipe into consideration. With the frontend and backend connected along with some tweaks made, an interactive predictive model will be fully developed.

Bibliography

Alec Radford, Jeff Wu, Rewon Child, David Luan,Dario Amodei, and Ilya Sutskever. 2019. Languagemodels are unsupervised multitask learners.

Atienza, Rowel. *Advanced Deep Learning with Keras: Apply Deep Learning Techniques, Autoencoders, GANs, Variational Autoencoders, Deep Reinforcement Learning, Policy Gradients, and More*. Packt Publishing, 2018.

Bie ́n Michał. "RecipeNLG: A Cooking Recipes Dataset for Semi-Structured Text Generation." Https://Www.kaggle.com/Datasets/Paultimothymooney/Recipenlg?Select=RecipeNLG_paper.Pdf

Faizan Khalid, Muhammad. *Text Prediction Using Machine Learning*. Linköping University, 2021, https://www.diva-portal.org/smash/get/diva2:1632760/FULLTEXT02.

"Getting Started | Create React App." *Create React App*, https://create-react-app.dev/docs/getting-started. Accessed 17 Aug. 2022.

"Keras API reference." *Keras,* https://keras.io/api/

"Keras Tokenizer Tutorial with Examples for Beginners - MLK - Machine Learning Knowledge." *MLK - Machine Learning Knowledge*, 1 Jan. 2021, https://machinelearningknowledge.ai/keras-tokenizer-tutorial-with-examples-for-fit_on_texts-texts_to_sequences-texts_to_matrix-sequences_to_matrix/.

"Keras-Preprocessing/Text.Py at Master · Keras-Team/Keras-Preprocessing · GitHub." *GitHub*, https://github.com/keras-team/keras-preprocessing/blob/master/keras_preprocessing/text.py. Accessed 17 Aug. 2022.

"Overview." *Tensorflow,* https://www.tensorflow.org/text

"Text Generation with an RNN | TensorFlow." *TensorFlow*, https://www.tensorflow.org/text/tutorials/text_generation. Accessed 17 Aug. 2022.

"Tf.Keras.Utils.To_categorical | TensorFlow Core v2.9.1." *TensorFlow*, https://www.tensorflow.org/api_docs/python/tf/keras/utils/to_categorical. Accessed 17 Aug. 2022.

"Writing Custom Datasets | TensorFlow Datasets." *TensorFlow*, https://www.tensorflow.org/datasets/add_dataset. Accessed 17 Aug. 2022.

"Using the State Hook." *Reactjs,* https://reactjs.org/docs/hooks-state.html.