Lluís Guàrdia Fernàndez
4-8-2020

CAPSTONE PROJECT:

Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

Since 2003, this sentence has been circulating on the internet. However, it seems that there never was a Cambridge research about it, but the general public has been debating for some time on the reason behind why we can read that particular jumbled text.

In 2011, researchers from the University of Glasgow, conducting unrelated research, found that when something is obscured from or unclear to the eye, human minds can predict what they think they're going to see and fill in the blanks. (https://www.gla.ac.uk/news/archiveofnews/2011/april/headline_194655_en.html). This phenomenon has been given the name "Typoglycaemia" and it works because our brains don't just rely on what they see -they also rely on what we expect to see.

I was interested in this sentence since I saw it on the internet a while ago and I came across it again two weeks ago, and, since I was already searching some ideas for the project, I came up with three ideas for it: a model that tries to guess the word given the first jumbled letters only, a model that tries to guess the existing word from its jumbled version or a model that tries to reorder the word given its jumbled letters (except for the first and last, which stay in the same position). I finally decided on the third option.

The objective of this Capstone project is to apply data science techniques in the area of natural language processing and obtain the original text after the specific transformation is applied. The idea is to implement and deploy a Neural Network model and create a web to show its performance to human entered text.

The results obtained will be of arrays of integers which will represent the letters used, as will be commented in more detail later, so the problem should be considered a multi-class classification problem, being each letter a different class.

To evaluate multi-class classification problems there are some metrics that can be used to evaluate, so I selected four of them:

- Classification Accuracy: it is what we usually mean with accuracy. It is the ratio of correct predictions to the total number of input samples. In my project two accuracies will be defined:
    - the ratio of fully correct words predicted
    - the ratio of letters correctly predicted.

  The first will allow us to understand the percentage of words my model is predicting correctly, but even if that is low, as a difference of one letter would mean an incorrect prediction; the second metric will allow us to detect more generally if the letters are being correctly predicted.
- Mean Cosine Distance: the previous metric was pretty strict in terms of considering their results correct when two words are compared. So I added another metric which doesn't calculate related to the correct letters, instead, it uses the whole word distance. As the words have been expressed by non-zero vectors the cosine distance between the predictions and the ground truth can be calculated, defined by sklearn as: "the cosine similarity, or the cosine kernel, computes similarity as the normalized dot product of X and Y: K(X, Y) = <X, Y> / (||X||*||Y||). " This allows us as to compare the words predicted in values between 0 and 1, the closer to 1 the better.
- Finally, as a last measure, although more visual, a confusion matrix will be displayed. This could show the distribution the letters follow and which letters the model tends more to write.

I will use these metrics and not other Relation related metrics, as I don't want to give more value to letters that are relatively close to the right letter in the alphabet (in exception to the cosine distance). For me all the letters that are not the exact right are incorrect, and it also could make the model tend to output the letter M, as it is the one in the center of the alphabet and it will result in the minimum difference given the other letters.

I've found some studies about the recognition of words with jumbled letters (https://www.researchgate.net/publication/325665269_Recognition_of_words_with_jumbled_letters), researches with Neural Networks on reordering the words in the sentence (https://github.com/Kyubyong/word_ordering) and, as explained as the second idea, a model that tries to guess the word from a vocabulary given the jumbled word (https://arxiv.org/pdf/1608.02214.pdf) which achieves a very good result.

Apart from that, I couldn't find a model that tried to reorder the words to obtain the original sentence using a Neural Network, so I will use two benchmarks as a comparison. For the performance at the full word prediction task, I will compare the accuracy of my model with the results obtained on the last commented work previously (the one that achieved very good result). To compare the letter to letter

performance, I will use as a benchmark the probability to guess right a letter, being this one 1 out of 27 (taking into account a 'no letter' element that will be commented later).

If this suggested model obtained good results it would mean that computers can store words and rely on them to predict the words it expects to receive, achieving some kind of Typoglycaemia.

ANALYSIS:

The data: To do this project I used the Blogger Corpus (http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm).

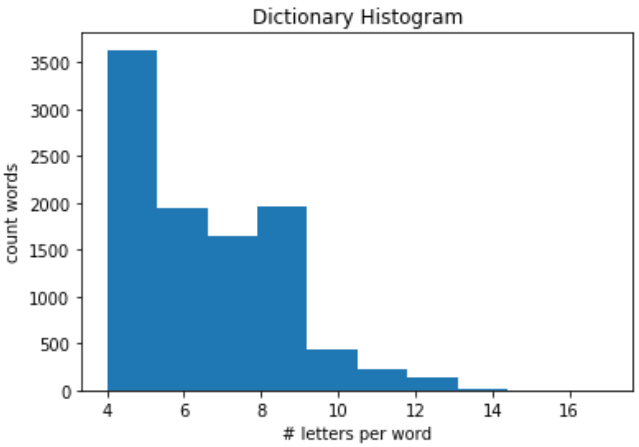
This data consists of 19,320 blogs on XML files. As it is a very considerable number of blogs, only a subset of them was be picked, being the first 1,499 blogs considering them a good representation of the complete dataset.

Each of these blogs consists of various posts with its respective date. I was only interested in the words so I extracted the posts from the blogs, resulting in 176,796 posts, making a mean of almost 120 posts per blog!

The words were extracted from the posts and resulted in 16,634,430 words, 810,981 of which are considered different. Later a count of them will be done, and only a vocabulary of the 10,000 more common words will be used.

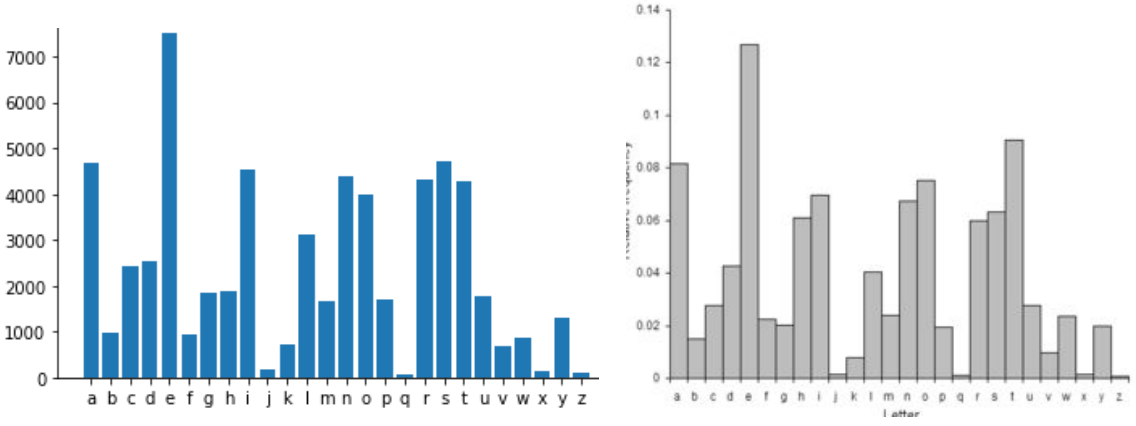
This words had a minimum length of 4 and a maximum length of 34. Having a mean length of 5,9 at first, and of 6,5 after the selection. This would mean that the majority of words were of less than 6 letters, as after the selection they were grouped and so the mean increased. This can be confirmed on the following histogram of the vocabulary:

Lluís Guàrdia Fernàndez
4-8-2020



This graph contrast with the distribution of the length on the English words, which have a mean of 8.23 characters (extracted from http://www.ravi.io/language-word-lengths). But we have to take into account that we didn't take as sample all the English dictionary, as the distribution of the lengths of the English words doesn't take into account the frequency, which is normally higher for shorter words, being more easily represented in smaller samples.

The letter distribution in our vocabulary (left) is almost identical as the letter frequencies of the English language (right, from https://www3.nd.edu/~busiforc/handouts/cryptography/letterfrequencies.html) as can be seen next:



We can also observe how the vocals are among the most used, along with *n*, *r*, *s* and *t*, being the *e* the most used one by a margin.

Lluís Guàrdia Fernàndez
4-8-2020

The model:

So our model wants to receive an entire sequence of data -in this case the letters of the words- and throw out as output another entire sequence of data, the letters resultant.

From the Deep Learning perspective, one of the models that are normally used and obtains better results are the LSTM's, in this case consisting of an encoder-decoder architecture. LSTM's are a type of artificial Recurrent Neural Network architecture, an architecture that, defined by DeepAI, is a type of neural network that contains loops, allowing information to be stored within the network. In short, Recurrent Neural Networks use their reasoning from previous experiences to inform the upcoming events. Recurrent models are valuable in their ability to sequence vectors, which opens up the API to perform more complicated tasks. In the encoder-decoder architecture, the encoder tries to learn efficient data representations (encoding) from the inputs, and the decoder tries to extract the complete information from them.

Also, Fully Connected layers in neural networks are those layers where all the inputs from one layer are connected to every activation unit of the next layer. In most popular machine learning models, the last few layers are full connected layers which compile the data extracted by previous layers to form the final output. In our model, a Fully Connected layer will be added at the output of the decoder architecture.

Lluís Guàrdia Fernàndez
4-8-2020

THE METHODOLOGY:

The structure I followed for this project was based on the structure done by Nadim Kawwa in https://github.com/NadimKawwa/rnn_sentiment_analysis/blob/master/SageMaker%20Project.ipynb

First of all, I extracted the data, which, as previously explained, consists of a subset of XML blogs, which themselves consists of different posts with its correspondent date.

To extract the post parts, at first I tried using ElementTree package, but it didn't work well so I ended up parsing the data with the help of BeatifulSoup. BeautifulSoup is a Python package for parsing HTML and XML documents. It creates a parse tree for parsed pages that can be used to extract data from HTML, which can be useful for web scraping. Despite that, at first, it didn't allow me to decode some of the files as an error arisen, so I found that I needed another encoding (I was using the predetermined UTF-8) to deal with HTML elements (like &nspb or others representing some punctuation marks for example). After dealing with that I read all the blogs available and I ended up the posts from all the subset of blogs. From now on I will consider each post a sentence.

What the model implemented wants as an input is a jumbled word, and it returns the prediction of the original word. In short, I wanted the data to consist of words, not sentences, so I applied some preprocessing to them.

The sentences processed and transformed into an array of words followed the following steps:
- remove the newlines, jumps and whitespaces that appear in the start and end of each sentence.
- remove possible HTML tags
- put all the sentence in lowercase
- some letters appear to have accents like *à*, so I normalized these to their non-accent version
- I obtained the indices of each of the word in the sentence
- from the indices earlier I obtain the word strings
- I ignore words that contain characters not considered integers, letters nor punctuation
- I also ignore words which consist of only integers or punctuation
- limited the posts to 500 words per sentence

- we ignore the words larger than *supercalifragilisticexpialidocious*, I selected this word because it's one of the largest if we ignore words more specifics to only one type of field (like chemical substances).

To do this processing, two snippets of the code used have been extracted from solutions in Stack Overflow, the first being the splitWithIndices from https://stackoverflow.com/questions/13734451/string-split-with-indices-in-python/13734815#13734815, and the second the strip_accents from https://stackoverflow.com/questions/517923/what-is-the-best-way-to-remove-accents-normalize-in-a-python-unicode-string.

I preprocessed all the sentences obtained from the blogs adding a pair of more steps:
- The words were stemmed, meaning they were reduced to the root/base word.
- All the words less than 4 letters were ignored, as once jumbled they will stay the same.
This process takes some time as the number of sentences to process is pretty large, so the result, consisting of an array of 16,634,430 words, was saved into a file for later extraction if needed.

At first, I decided to use these words as the dataset, but this resulted in a very large dataset, with some of the words repeating with the only addition of plural or feminine termination. After thinking about it, I thought that was probably better to take the most common N words from the subset after tokenizing them as it would be a better representation. So I created a dictionary with the number of appearances of each word and took the 10,000 most repeated. This resulted in the training dataset.

As the test dataset, I took 20,000 random words directly from the preprocessed file.

For the model built I constructed a feature representation which consisted in representing each letter of each word as an integer. I used the Latin alphabet as a dictionary between letters and integers.

The next step would have been to transform all the words in the dataset with the dictionary, but first, as my model is a Recurrent Neural Network, I defined the length of each word to be the same. To do this, I used the previously defined max size of a word (the length of the word *supercalifragilisticexpialidocious*) as the fixed size for the words and then pad short words with the category 'no letter' (labelled 0). As there weren't longer words (from the processing steps) there was no need to truncate any.

So then I converted the all dataset to integer arrays. The conversion of a character that wasn't in the dictionary consisted on ignoring it, this was intending to simplify

more the task and it made some of the words shorten to less than 4 letters, so I cut them out of the data. This resulted in a final count of 9,445 different words for my dataset. From now on I worked with the integer representation of the words, which I will be referring as int words. From this step, I also obtained the original length of the words, which will be of future use.

My model takes as input a jumbled word except for the first and last letter, so a function was defined to apply this transformation to an int word and it was used on all the dataset.

So now I have the training and test input, its ground truths and the original lengths of the words, so I uploaded all of them as datafiles. The dictionaries were saved also.

From this datafiles I created CSV files for both training and testing. These were saved with a format form output[34], length, input[34] where input[34] is the jumbled int word, being the sequence of 34 integers the letters in the words, and the output[34] is the integer ground truth word. Then I uploaded on the S3 these CSV files and the dictionaries, to make later use on the deployed model.

Now was the time to enter into the LSTM model commented on the analysis part, for this I used Pytorch.

My first approach was to use as 27 as the input and output elements lengths, as the integer's value of the letters will be encoded on a one-hot encoding representation, using the dictionaries defined earlier. But I later prefered to add the real length of the word as an input too with the idea that it could improve the results so the input size was changed to 34 in order to one-hot encode this length too which its maximum value is 34.

The output elements will consist of an array with the probability of each letter in its index. The letter selected from this will be the one with higher value, being the one-hot representation the ideal result.

After the model, I defined the training function. Before applying to the model which will be deployed, I defined the training function in the notebook itself in order to check it's correct functioning and then copied it on the respective file. This training function includes an early stopping method to stop the model in the case it starts to decrease its performance.

To obtain the data used for training (which will have to be in the training function as well) I had to extract the data from the CSV files, one-hot encode the input words,

transform all into torch tensors, which is the type of data that the PyTorch model expects, and join them into a dataloader, which will be used on the training function.

I defined Adam as the optimizer, as it's one of the bests, and cross-entropy as the loss, as it measures the performance of a classification model whose output is a probability value between 0 and 1. It increases as the predicted probability diverges from the actual label and I thought it was the most suitable for this job.

Once this was prepared, I created the PyTorch Estimator and trained it for some epochs to check no errors occurred. As it worked fine I copied it into the training file, created the PyTorch estimator and let it train.

Finally, it was time to see the performance of the model with its predictions. In order to deploy it correctly, the SageMaker server needs 4 functions (within the designed file), model_fn to load the model, input_fn to process the input, output_fn to process the output and prediction to perform the inference. For the input_fn and output_fn I was a little lost of what I was supposed to write on them, which resulted in copying (and understanding what I was doing) this 10 lines of code from https://github.com/NadimKawwa/rnn_sentiment_analysis/blob/master/serve/predict.py.

With the prediction, I did the same as the training and first defined the predict function inside the notebook and then copied into its respective file. But before that, I defined some functions to allow get the strings back from the int word, and join all the words within a sentence.

For the predict function, it receives the array of int jumbled words. It goes word by word, it one-hot encodes them, transforms them into torch tensors, and pass them through the model, obtaining the probabilities explained previously. From this probabilities, the letters selected are the indexes of the maximum value of the probabilities. Finally is returned the int words obtained from these letters.

I tried the predict function with both the array of int words and a sentence. For the latter, I defined a function which makes all the preprocessing steps necessary to obtain the array of int words from the sentence in order to send them to predict.

Once defined the predict function and checked its functioning, I created the PytorchModel and deployed it. In order to check it's performance, I created a function that calculates the metrics defined previously and passed the test dataset to it.

With this, the first part was done.

As the second part, I created the web app application. To do so I created an IAM Role with AmazonSageMakerFullAccess, I created a lambda function and set up the API Gateway. The notebook contains the steps on how to set up these elements. Finally, I used the index.html file to call the web app. This basic index HTML file was based on the one implemented at https://github.com/NadimKawwa/rnn_sentiment_analysis/blob/master/website/index.html

Once finished everything, the only remaining thing was to delete the endpoint and clean up everything used.

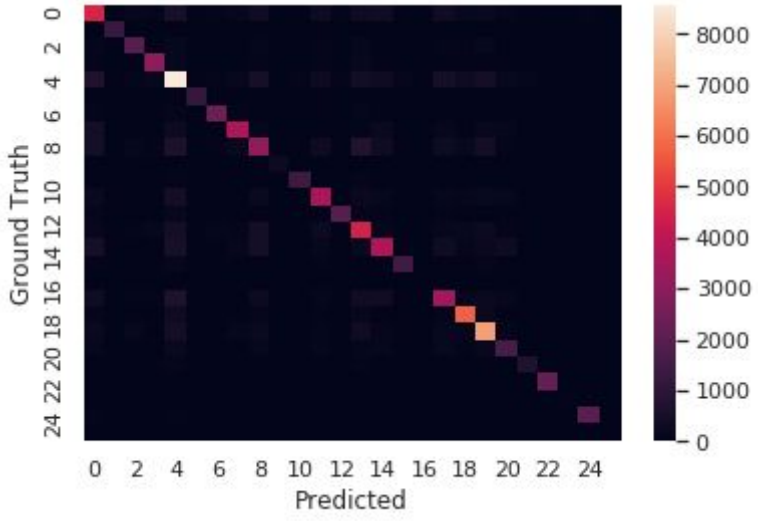
Lluís Guàrdia Fernàndez
4-8-2020

RESULTS OBTAINED:

After all the results obtained with the model defined and trained with parameters:
- input dimension = 34
- output dimension = 27
- hidden dimension = 156
- epochs = 90
- learning rate = 0.0005

were of 30% of fully correct predicted words, 63% of the letters predicted were correct and the words had a mean cosine distance of 0.51.

The resulting confusion matrix:



Examples from the web:

-this isn't an example sentence
**this inse an empplae senetene*

-Despite the appearances, she stod among the best
**deptsee the arreeaepaes she sood abyoo the bust*

-You have to be more careful and read between the lines
**you have to be mare cofferl and read breteen the lines*

-Did you know that, along with gorgeous architecture, it's home to the largest tamale?
**did you know that along with gooeeres arrtttoecure int home to the leasert timmme*

Benchmark comparison results				
full words	acc (%)		*by letters*	acc (%)
scRNN	**98.95**		random probability	3.7
our model	30.23		our model	**62.99**

Looking at the numeric results obtained one could say that the model performs poorly, only getting completely right a little more than 1 out of 4 words, even worse if compared with the scRNN model performance. But the fact that is compared to another very similar but not same task is against us, as what their model does is to select a word within a wide vocabulary, but it doesn't have to learn the structure of the word itself.

The other numeric results look a little more encouraging, as the letter accuracy surpasses by far what would be considered getting it right by luck. This result also means that some of the words have not been considered as right in the previous metric by a little margin.

From the ‘string’ results, the performance is more or less what the numbers suggested, in some words it obtains pretty good results but as the number of letters increase and the more unusual the word is, its performance decays.

At last, the fact that the diagonal line in the confusion matrix is clearly seen is a good sign, meaning that the majority of letters have been predicted correctly. Despite that, a faintly whiter column appears on the fifth position, related to the letter *e*, which means that this letter has been predicted incorrectly more usually.

I think that the difficulty of this task is the wide number of different words and different lengths it has to cover, which with even the 10,000 words used doesn't nearly cover the 171,476 words on the Oxford Dictionary. This is seen in the larger words which it doesn’t probably have seen on the training and what it seems to tend in this cases (smartly enough) is to put letters e on them, probably because is the letter most probable. This coincides with what we have seen on the confusion matrix.

Also we couldn’t train it with all the possible feminine or plural terminations of the words, as it would increase its computational time and complexity exponentially, so when one of this terminations is used in an input, it should be difficult for the model to get it completely right.

Maybe it could have work better if another more general dataset was used, maybe the blog vocabulary is too much specific for the usual sentences.

Compared to the models used on a basis in the professional Deep Learning world, this one sadly lacked memory, computational power and time. It also lacked a finer hyperparameter tuning, as I didn't have the time to perform a grid search for the parameters and only did some testing with them. Right now this model is not able to solve the problem proposed.

Lluís Guàrdia Fernàndez
4-8-2020

CONCLUSION:

As a conclusion, this project allowed me to learn about how to apply a machine learning solution to an existing problem, and all it involves.

For me what most struck me was the part processing the data. I always heard that preparing the data was really the part most time taking but I never completely believed in it, but once I had to prepare it myself I realized how true the statement was.

Also I would like to remark how infuriating was at times the Sagemaker deployment, since I had some problems on the building of the model on Sagemaker, especially on the deployment, and every time I had to wait for more than ten minutes to stop and tell me if it worked correctly, even at some times giving me error that I didn't really understand. This increased a lot the time it took to finalize the project and made me more grateful to the Cloudwatch logs.

For future work, more robust techniques could be tested such as dropout, regularization or a more complex model like adding an Embedding Layer at the input. Besides, it could be possible to implement some kind of punctuation mark holder that allows to save the punctuation of the sentence and put them back after the prediction is made.

This project, with better results, have some potential NLP applications such as normalizing social media text or correction of spelling mistakes.