

Skill based LU

Contacts

Role	Name
Dev Coach	Ankit Jain, Amit Kumar Yadav
EM	Abhishek Agarwal
Dev Interns	Abhirupa Mitra, Koushiki Dasgupta Chaudhuri, Parth Shettiwar, Shivam Goel
UX Coach	Shourya Mehrotra

Abstract

Voice technology is getting more and more ubiquitous and with that, the need for it to work always. Our implementation of dictation and commanding in mobile word is dependent on good network connection to work, which may not always be available. Hence there is a need to support offline voice commanding in Word.

The aim of this project is to enable a voice command mode which takes commands in the offline mode in Word.

User Scenario

User is travelling to office in a car and she wants to utilize this time to write the spec she has to present. She launches Word app on her mobile phone and goes to dictate. She finishes the dictation and now wants to format the document. Meanwhile, the car passes through a remote area and she loses internet connection. But she finds that she can still use the commands seamlessly. She is pleasantly surprised by the productivity boost that she got by using Microsoft Word.

Goals & Measures

Enable a voice command mode which takes commands in the offline mode:

An ML based module running on client must be able to infer the right command from set of available commands.

Fundamentals:

- **Size** (<250 kb) & The model should be downloadable dynamically so that App install size has no impact. Localized and scenario specific modules must be available for client.
- **Scalable Architecture** - Addition of new commands, should not require any code change in module. Adding new training data is all that should be needed.

Extended Goals:

- End to end local scenario (including dictation)
- Inline Voice Command for easy switch between the command mode and dictation.

Non-Goals

- Automatic detection of command in free dictation
- Online learning as user fires commands
- Training on device

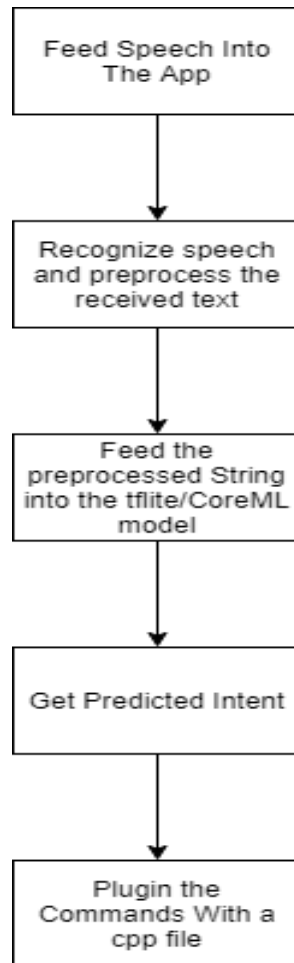
Principles:

A) Catering to offline scenario does not mean that device is always offline. We are open to exploit any opportunity when device is actually online to enable/enrich our scenario. Eg. We can download the models at runtime when device first comes online with target app open. This will help keep size small. Similarly, we can record the input-output of online luis to help us in offline mode. This gives consistency of experience as well.

B) At one point in time, the device is either online or offline. If online it uses online dictation and online luis and if offline, it uses offline dictation and offline luis. There is no combination of online stt with offline luis or vice-versa.

C) This work is independent of how offline recognizer works. For purpose of training and testing, device provided recognizer has been used and its data preprocessed.

Workflow/Flowchart:



Requirement

- Switch to move from transcription to command mode.
- An ML based module running on client must be able to infer the right command from set of available commands
- The model should be downloadable dynamically based on scenario and language
- Working with local dictation

Creating the test dataset and incorporating similar sounding words:

Twenty-six commonly used sample commands were chosen as a starting point to build the dataset. They were undo, bold, remove bold, italics, remove italics, underline, remove underline, superscript, remove superscript, subscript, remove subscript, strike through, remove strike through, center align, insert comment, left align, right align, remove formatting, insert bullets,

next bullet, end bullet, pause dictation, stop dictation, show commands, show help, delete. A dataset of around 400 sentences was created initially which had variations of these commands. Next, to compensate for shortcomings of offline speech to text API, a voice dataset created by another team was used. The voice commands were fed into the offline speech-to-text API and similar sounding words to commands were generated whenever the API was not performing optimally. These similar sounding variations like old for bold, eat Alex for italics were then added to our original dataset.

Models

1) Convolutional Neural Network (CNN) based:

Though CNNs are associated more frequently with computer vision problems, recently they have been used in Natural Language Processing with interesting results. CNNs are basically just several layers of convolutions with non-linear activation functions like ReLU or tanh or SoftMax applied to the results. Convolutions over the input layer are used to compute the output. This results in local connections, where each region of the input is connected to a neuron in the output. Each layer applies different filters, typically hundreds or thousands and combines their results. Another key aspect of CNNs are pooling layers, typically applied after the convolutional layers. Pooling layers subsample their input. Pooling is required to provide a fixed size of the output matrix by reducing the output dimensionality but keeping the most salient information.

Pre-processing the text:

Two main pre-processing methods were applied in the CNN based approach which were:

- **Tokenizing:** Keras' inbuilt tokenizer API was fit on the dataset which split the sentences into words and created a dictionary of all unique words found and their uniquely assigned integers. Each sentence was converted into an array of integers representing all the unique words present in it.
- **Sequence Padding:** The array representing each sentence in the dataset was filled with zeroes to the left to make the size of the array 10 and bring all arrays to the same length.
- The labels were converted into one-hot vector using **to_categorical** function.

Generating a train-test split in the dataset:

Random shuffling of indices was used to split the dataset into training and testing data in roughly 9:1 ratio.

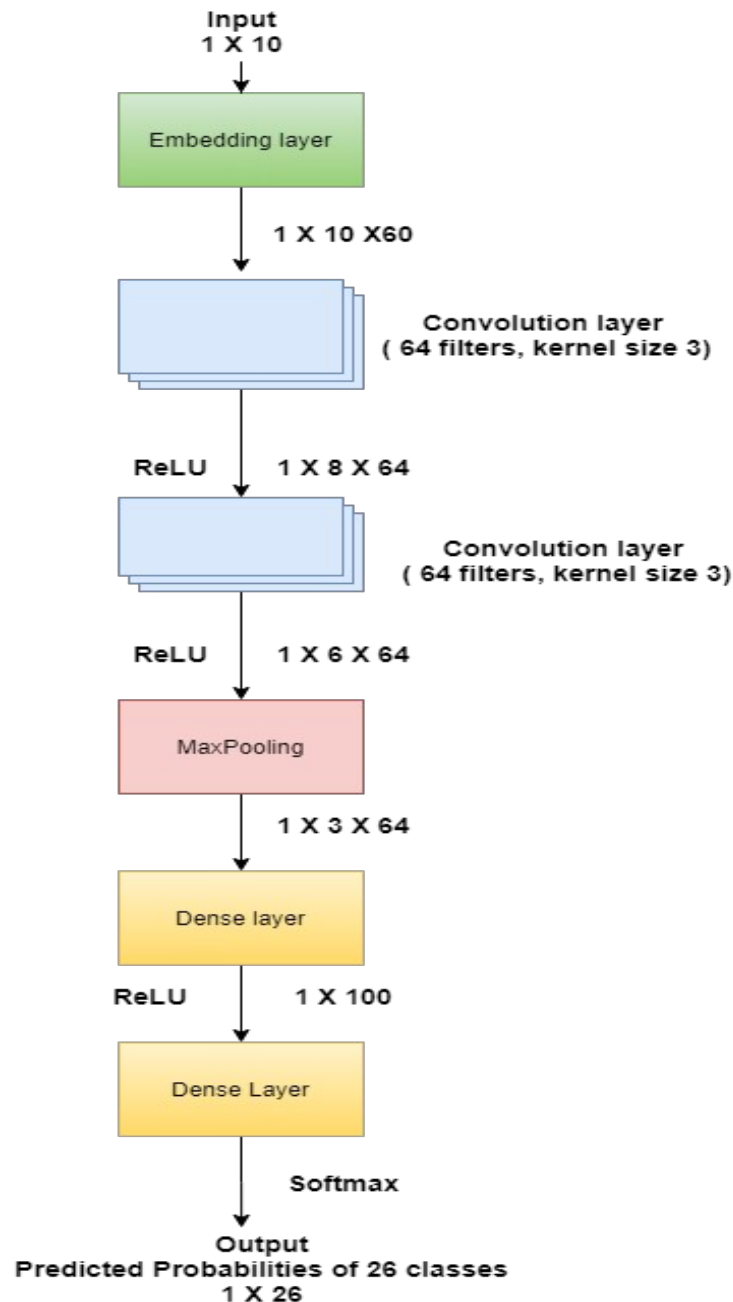
The model:

A keras functional model was implemented. It had the following layers:

- An input layer which took the array of length 10 representing a sentence.
- An embedding layer of dimension 60 whose weights could be updated during training.
- Two convolutional layers (Conv1D) with 64 filters, kernel size of 3 and relu activation.
- A max pooling layer(MaxPooling1D) with pool size 2.

- A flatten layer to flatten the input without affecting batch size.
- A dense layer of 100 units and relu activation.
- A dense layer of 26 units and softmax activation.

Model Diagram:



Compilation:

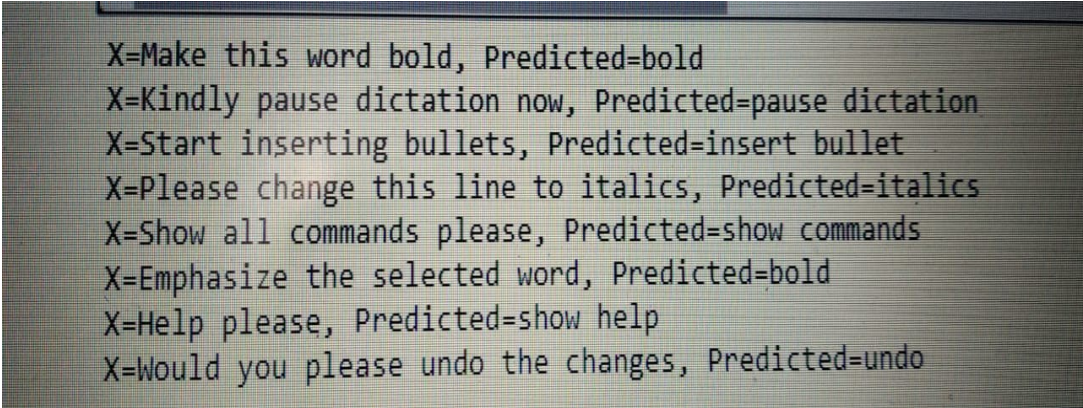
The model was compiled with categorical cross-entropy as loss and rmsprop optimizer. It was judged on the metric of accuracy.

Training:

The model was trained for 30 epochs with batch size 50.

Results:

The model yielded an accuracy between 90-95%, averaged over 50 runs of training. It also gave good predictions on new data as shown below:



```
X=Make this word bold, Predicted=bold
X=Kindly pause dictation now, Predicted=pause dictation
X=Start inserting bullets, Predicted=insert bullet
X=Please change this line to italics, Predicted=italics
X=Show all commands please, Predicted=show commands
X=Emphasize the selected word, Predicted=bold
X=Help please, Predicted=show help
X=Would you please undo the changes, Predicted=undo
```

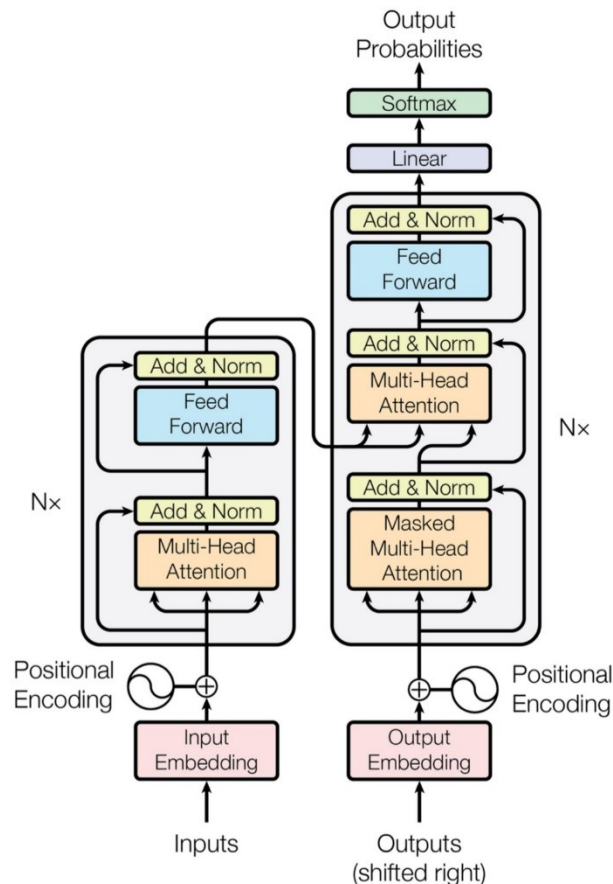
Size:

The model size in Keras was 244 kB. After converting to a coreML format for iOS, it's size was 112 kB. After converting to TFLite format for Android deployment, it's size was 60 kB.

2)Transformer based:

The transformer architecture was a breakthrough in Natural Language Processing tasks when it was published in the paper "Attention is All you need" back in 2017. The architecture has no Convolutional or Recurrent layers, just attention layers are incorporated where it is learned to put more attention on the words which are important to produce the output. The sequential nature of the previous model architectures, which prevented parallelisation, was mainly overcome by the Transformer Architecture. For ex. Architectures like RNN and LSTM have hidden states which depend on the output of past Hidden state. The following diagram would give a better understanding of the Transformer Architecture.

Model Architecture:



The Transformer - model architecture.

Model Flow:

The model is broken into Encoder(Left) and Decoder(Right) architecture.

- 1) First the input is preprocessed, where first according to a dictionary , the words of a sentence are mapped to Integer numbers. After this each vector of Integers for every sentence, are padded with 0's, to keep the Input length of vector feeded to network constant. This preprocessing is exactly similar to the Pre-processing text section described in CNN based model.
- 2) This vector generated is then fed as input to the Embedding layer of Encoder Pipeline(left). The output of this is such that each word(fed as integer) is projected to a higher dimension which is a hyperparameter. Let this parameter be called K.
- 3) The positional encoding is optional and is omitted in our implementation. This is due to the fact that the words of a sentence can be in any order, still our intent prediction should be same.
- 4) This is followed by Multi-Head attention block and normalisation. This is the crucial stage where weights are learned to put more attention on the words which are crucial in producing the output. This block mainly performs a dot product of the words(Integer array) with the weights . The block finally helps to learn a mapping where it learns on which word to put more attention.
- 5) The output of this is passed through a Dense Layer which is then given to the Decoder Pipeline.

6) Since this is a Intent Classification task, the output is a single label. Hence in the decoder pipeline, we directly start from the second Attention block where the output of Encoder pipeline is fed. The output intent label is not fed to the decoder pipeline.

7) Similar to the Encoder Pipeline, the Decoder pipeline Has a Attention block and Dense layer. Finally Softmax is performed to produce the final 1×26 vector of Probabilities for Intents. The max is taken among them to give the predicted Intent.

Generating a train-test split in the dataset:

Random shuffling of indices was used to split the dataset into training and testing data in 9:1 ratio.

Training

The backpropagation is mainly done using Cross Entropy Loss. The loss is taken between the One hot array of Ground Truth Labels and Predicted Probabilities Array. Also training is done using Adam Optimizer. The model is trained for about 300 epochs with a batch size of 28. The input sequence length is fixed to the max sentence length in the dataset.

Size vs Accuracy Tradeoff

Important point to be considered is we need a good accuracy as well size should be below 250kb. In this we face a tradeoff. The size is mainly controlled by the parameter K(the embedding Dimension). At $K = 50$, the size of the model is about 850 KB. After As K is decreased, the model size is reduced but at the same time accuracy of the model is also dropped. The sweet point is found after some trials, we find at $K = 30$, we get a model size of about 450 KB with accuracy of about 88%. The size is further reduced after converting it to CoreML model to about 350 KB.

Pros

- 1) With Proper Hyper-Parameter Tuning, this model easily achieves 100% accuracy.
- 2) The model is highly scalable and can be extended to many commands without drop in accuracy

Cons

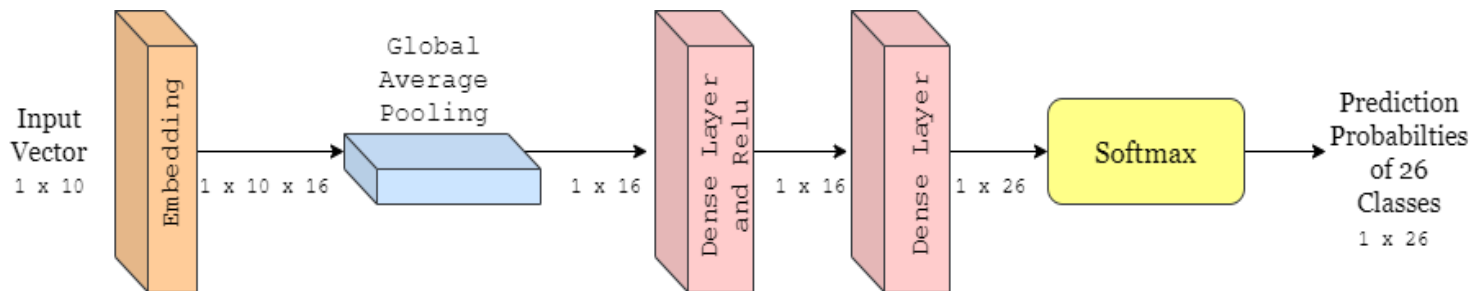
- 1) The model size is much over the Expected Constraints /Specifications. The model suffers from Size vs Accuracy tradeoff.
- 2) This model was Implemented in Pytorch framework. The pytorch model cant be converted to ".tflite" model which is mainly required for deploying on android apps.

The alternative option to counter point 2 can be, writing the whole script in Keras Framework, which can be easily converted to ".tflite" model. The other option is to convert the torch model file to something called "Torch Script" which can be easily deployed on Android Apps similar to Tflite. However Torch Scripts increase the size of the model. In any case the 1st point was a big challenge which needed a deeper investigation.

3)Average Word2Vec:

This model is purely based on Embedding of the Input vector array. As the name suggests, the input array of words are converted to vectors after which Global Average Pooling is performed on the output on the Sequence Length Dimension. This model works exceptionally well on Small Datasets. Due to simplicity and good accuracy of the model, this architecture is quite often used for Intent Classification Tasks.

Model Architecture:



Model Flow:

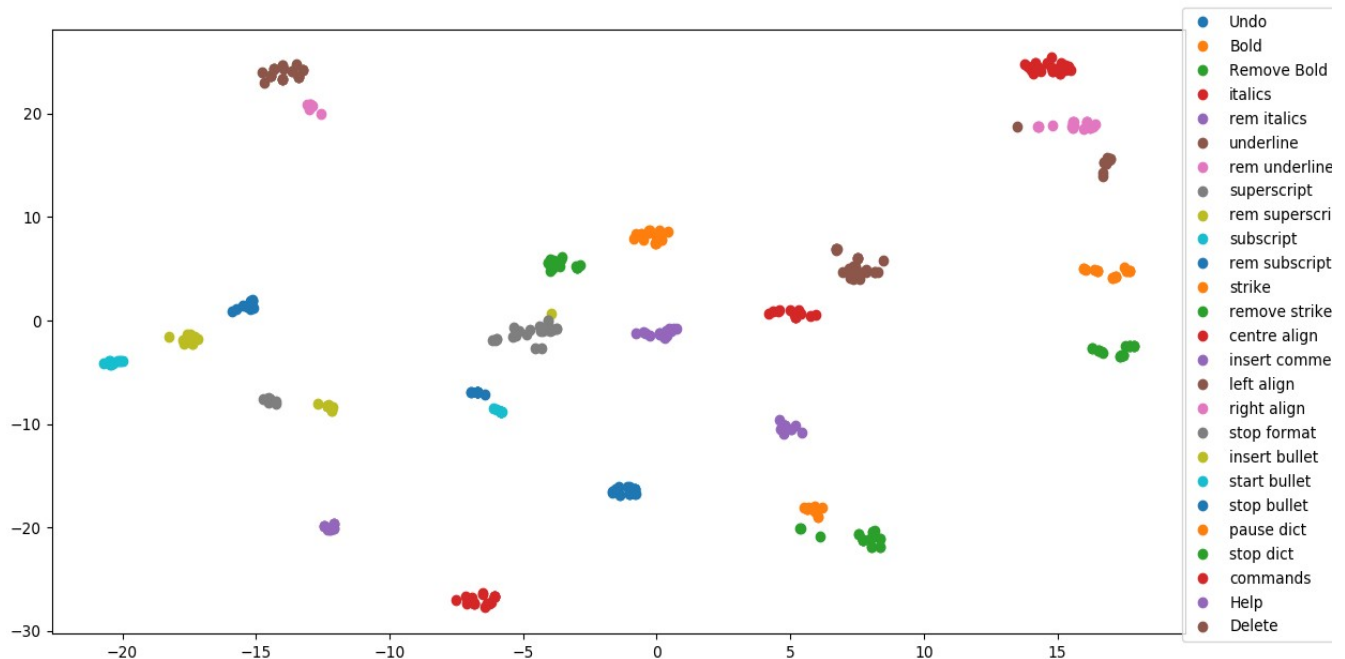
- 1) Similar to previous model's Pre-processing, we create a dictionary, convert input words to Integer Array and finally Right pad to a fix specific length with zeroes. The Input sequence length is fixed to 10.
- 2) The Input Integer array of 1×10 size, is then passed through an Embedding Layer, which learns the mapping for every word in vocabulary to a higher dimension. Hence the resultant vector size comes out to be $1 \times 10 \times 16$, where 16 is the Higher Dimension set and is a Hyper-parameter.
- 3) The output array is then passed through Global Average Pooling layer, which simply takes the average of all the vectors of words of a sentence. This gives a 1×16 array. Note, the average is taken across the sequence length dimension. The intuition behind this is we that important words required for classifying the intent, like Emphasis for Bold Intent and Remove for Negative Intent, would play a major contribution is differentiating from other Intents. The less important words like "Can", "You" wont make big difference to the final output of pool layer(That's how embeddings of each word will be learned).
- 4) The output is followed by 2 Dense layers with a Relu sandwiched in between. The Dense layers mainly learn to learn the underlying distribution of classes by looking at the 1×16 vector generated from Average Pooling Layer.
- 5) Finally a softmax layer is incorporated which gives 1×26 array of Prediction Probabilities for 26 Intents.

Training

The training is done with a batch size of 32 for 300 epochs with Adam optimizer. Cross entropy loss is taken between the one hot array of Groundtruth class labels and prediced probabilities array.

Results

This model gives staggering size of only 11 KB and accuracy of about 95-100%. To get to know, whether the model really learnt the underlying distribution, tsne plot of the output of Global Average Layer is made for all Examples of Dataset.



From the tsne plot, it can be clearly seen, all 26 clusters are distinct. The opposite intent (Like Bold and Remove Bold) clusters nearby, clearly building upon the intuition that since they have a common word Bold in them, their vector average becomes similar, the word Remove being the only differentiator. A similar case can be seen for insert bullet, start bullet and stop bullet, all three clusters are located nearby.

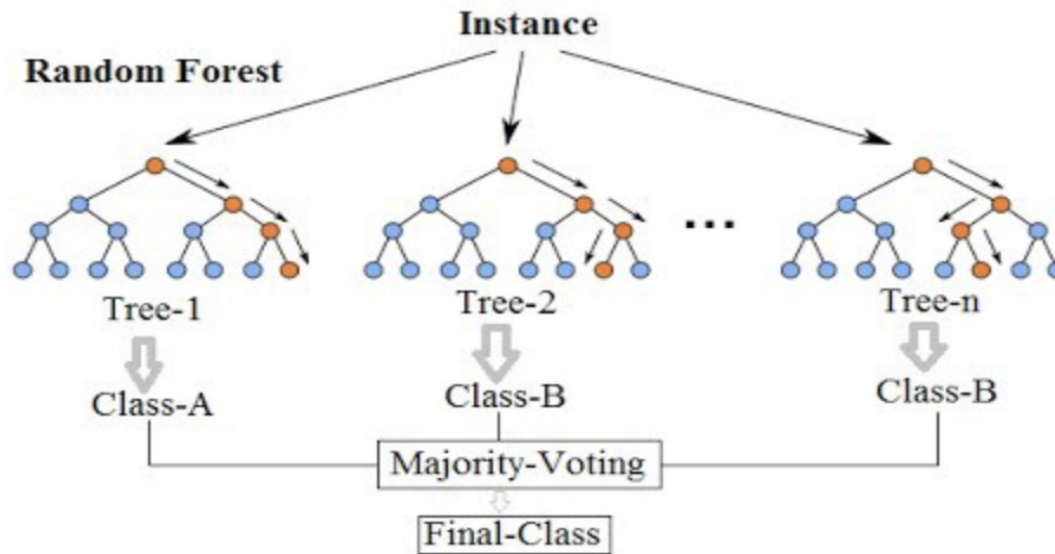
Pros

- 1) The model size s very less and much below the specifications
- 2)High accuracy is achieved.
- 3) Can be easily deployed on android devices.

4) Random Forest:

A decision tree is the building block of a random forest and is an intuitive model. We can think of a decision tree as a series of yes/no questions asked about our data eventually leading to a predicted class (or continuous value in the case of regression). This is an interpretable model because it makes classifications much like we do.

Random Forest Simplified



Pre-processing the text:

The pre-processing methods applied in the Random Forest based approach were:

- **Stemming:** Stemming was used to make the vocabulary. All words which means similar like 'walk' and 'walked' were made one.
- **Stopwords:** The words that do not convey any meaning to our intent like 'the', 'was' and 'is' were removed.
- **Tokenizing:** Count vectorization was used. Each sentence was converted into an array of numbers representing the count of each word in the sentence that are present in vocabulary.
- The labels were converted into one-hot vector using **to_categorical** function.

Generating a train-test split in the dataset:

Random shuffling of indices was used to split the dataset into training and testing data in roughly 6:1 ratio.

The model:

Data set was fit into the Random Forest model. The decision trees worked on the count of words in the input sentences and based on that were inferring the intent of the sentence.

It gave an accuracy of 90% on testing dataset. The model had a size of 595 kB.

==> The main issue which came with this model was its size. It went out of our constraint, so this model was rejected.

5) Naïve Bayes:

A Naive Bayes Classifier is a supervised machine-learning algorithm that uses the Bayes' Theorem, which assumes that features are statistically independent. The theorem relies on the

naive assumption that input variables are independent of each other, i.e. there is no way to know anything about other variables when given an additional variable.

$$P(A | B) = \frac{P(B | A)P(A)}{P(B)}$$

where A and B are events and $P(B) \neq 0$.

- $P(A | B)$ is a **conditional probability**: the likelihood of event A occurring given that B is true.
- $P(B | A)$ is also a conditional probability: the likelihood of event B occurring given that A is true.
- $P(A)$ and $P(B)$ are the probabilities of observing A and B independently of each other; this is known as the **marginal probability**.

Pre-processing the text:

The pre-processing methods applied in the Random Forest based approach were:

- **Stemming**: Stemming was used to make the vocabulary. All words which means similar like 'walk' and 'walked' were made one.
- **Stopwords**: The words that do not convey any meaning to our intent like 'the', 'was' and 'is' were removed.
- **Tokenizing**: Count vectorization was used. Each sentence was converted into an array of numbers representing the count of each word in the sentence that are present in vocabulary.
- The labels were converted into one-hot vector using **to_categorical** function.

Generating a train-test split in the dataset:

Random shuffling of indices was used to split the dataset into training and testing data in roughly 6:1 ratio.

The model:

The naïve bayes model was trained on the dataset. Using the formula above and the count of words in each sentence as other independent events, the model was determining the intent of the sentence.

It gave an accuracy of 80% on testing dataset. The model had a size of 44 kB.

==> The model was performing decently and was giving a small size which was under our constraints. The accuracy was a concern here and we looked for better alternatives.

6) Linear Regression:

Linear regression outputs continuous number values which can then be mapped to two or more discrete classes.

Pre-processing the text:

The pre-processing methods applied in the Random Forest based approach were:

- **Stemming**: Stemming was used to make the vocabulary. All words which means similar like 'walk' and 'walked' were made one.

- **Stopwords:** The words that do not convey any meaning to our intent like 'the', 'was' and 'is' were removed.
- **Tokenizing:** Count vectorization was used. Each sentence was converted into an array of numbers representing the count of each word in the sentence that are present in vocabulary.
- The labels were converted into one-hot vector using **to_categorical** function.

Generating a train-test split in the dataset:

Random shuffling of indices was used to split the dataset into training and testing data in roughly 6:1 ratio.

The model:

The model was first trained in python using tensorflow.

The accuracy was around 100% and the size was 9.8 kB when converted to tflite.

==> The major drawback was that that tflite used some dependencies which increases the overall size of the app. So, it took the app size out of our constraint and we were forced to reject this model.

Then, the mathematics was extracted out of the model and it was implemented in core java so it can be used in our mobile app directly.

Diagram illustrating the model's mathematical logic:

$$\begin{matrix}
 \text{Intent} & \xrightarrow{\text{W (vocab)}} & X & + & B & = & \text{res} \\
 \begin{bmatrix} w_{11} & w_{12} & \dots & w_{185} \\ \vdots & \vdots & & \vdots \\ w_{261} & \dots & \dots & w_{26,85} \end{bmatrix} & & \begin{bmatrix} x_1 \\ \vdots \\ x_{85} \end{bmatrix} & & \begin{bmatrix} b_1 \\ \vdots \\ b_{26} \end{bmatrix} & & \begin{bmatrix} res_1 \\ \vdots \\ res_{26} \end{bmatrix} \\
 26 \times 85 & & 85 \times 1 & & 26 \times 1 & & 26 \times 1
 \end{matrix}$$

$w_{ij} \Rightarrow$ weight of vocab j on word i

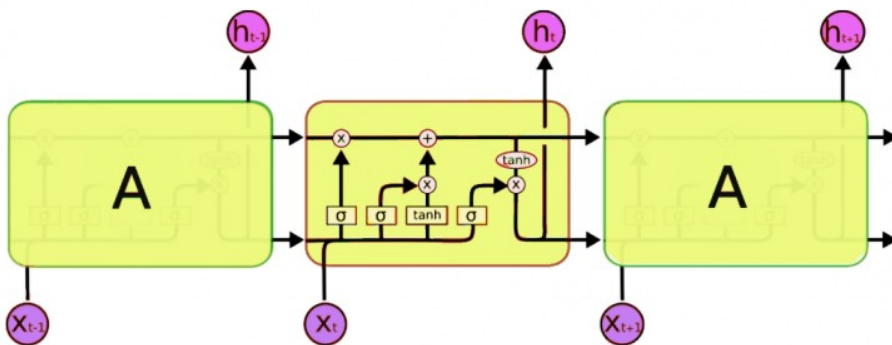
$$F(\text{str}) = \max(\text{res})$$

The model gives each word in the vocabulary a weight corresponding to each intent which signifies the importance of that word for the intent. So, a matrix is formed telling importance of each word. Also, a bias is given for each intent. When the count of each word in the input sentence is multiplied with weight matrix and bias is added, it gives an array of values that shows the confidence of the model for each intent. Now the intent having the maximum confidence value is inferred out by the model.

The above model gave an accuracy around 100%. The size was noted before and after removal of the model from the android app and it did not show any decrease in size. This showed us that the size of the model is very less even when integrated with app.

7) Long Short Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems.



LSTM network is comprised of different memory blocks called **cells (the rectangles that we see in the image)**. There are two states that are being transferred to the next cell; the **cell state** and the **hidden state**. The memory blocks are responsible for remembering things and manipulations to this memory is done through three major mechanisms, called **gates**.

Pre-processing the text:

Two main pre-processing methods that were applied are:

- **Lemmatization:** Lemmatization is the process of grouping together the different inflected forms of a word so they can be analysed as a single item. Lemmatization is similar to stemming but it brings context to the words. So it links words with similar meaning to one word.
- **Stopword Removal:** A stop word is a commonly used word (such as “the”, “a”, “an”, “in”) that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query.
- **Tokenizing:** Keras’ inbuilt tokenizer API was fit on the dataset which split the sentences into words and created a dictionary of all unique words found and their uniquely assigned

integers. Each sentence was converted into an array of integers representing all the unique words present in it.

- **Sequence Padding:** The array representing each sentence in the dataset was filled with zeroes to the right to make the size of the array 6 and bring all arrays to the same length.

Parameters For Model:

```
vocab_size = 1000  
embedding_dim = 20  
max_length = 10  
trunc_type = 'post'  
padding_type = 'post'  
oov_tok = '<OOV>'  
training_portion = 0.8
```

Generating a train-test split in the dataset:

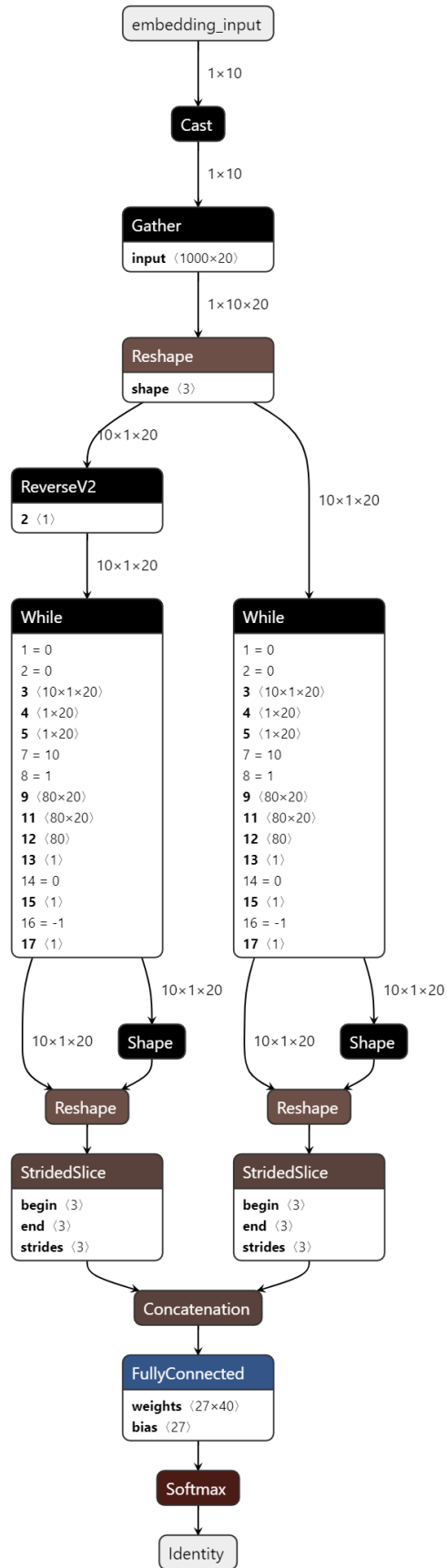
Random shuffling of indices was used to split the dataset into training and testing data in roughly 9:1 ratio.

The model:

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 10, 20)	20000
bidirectional (Bidirectional)	(None, 40)	6560
dense (Dense)	(None, 27)	1107
Total params: 27,667		
Trainable params: 27,667		
Non-trainable params: 0		

Model Diagram:



Training:

The model was trained for 100 epochs

Results:

Epoch 100/100

11/11 - 0s - loss: 2.3967e-04 - accuracy: 1.0000 - val_loss: 0.3030 - val_accuracy: 0.9419

Models Comparison

Model Type	Model Size(kB)	Model Size:After CoreML	Model Size(TFLite)	Reported Accuracy(average over 50 runs)	Reported Accuracy: after CoreML
Transformer	451.9 kB	356.6 kB	-	88.42%	
Average Word2Vec	-	-	11.58	95-100%	
LSTM					
CNN	244 kB	112 kB (after converting spe	60 kB(after post-training quantization)	89.99%	
Random Forest (stemming and count vect)	595kB			85-93%	
Naive Bayes (stemming and count vect)	44kB			75-90%	
Naive Bayes (lem and count vect)	59kB			72-86%	
Naive Bayes (stem and tf vect)	42kB			72-86%	
Naive Bayes (stem and ngram vect)	303kB			86-96%	
Linear Regression(stemming and count vect)	22kB			88-98%	
Linear Regression (lemmatize and count vec	30kB			77-95%	
Linear Regression (stem and tf vect)	22kB			77-95%	
Linear Regression (stem and ngram vect)	152kB			77-95%	
Fast RNN					
LSTM	7.2mb (Without Tflite)				
LSTM (Removed relu dense layer and broug	3.2mb (Without Tflite)			89%	
LSTM (Bidirectional Layer)	2.6mb (Without Tflite)			97%	
LSTM (Tflite)			148.7kb (With TFLITE)	95 - 98%	
LR (stem count vect removing please)	22kb			90-100%	
LR tf			9.8 kB(tflite)	100%	

The above table compares all the models implemented on Sizes and Accuracies. For size, the main Viewpoint is the Size of model after converting to tflite. . A proper analysis of the above table reveals the following 2 models are promising on both accuracy and size:

1) Average Word2Vec - Accuracy : 95 – 100% , Size : 11.58 kB

2) LR tf(Linear Regression) - Accuracy : 100%, Size : 9.8 kB

Further the above 2 models are scalable and can be easily deployed on any android device with minimum installation. Moreover, two other models namely, LSTM and CNN based, both satisfy the specifications(<250 kb size and 90%+ accuracy) and can be explored further. For other models, which still satisfy both constraints, are not feasible, since they cant be converted to tflite(Models whose tflite size is not mentioned).

Custom Tflite Library

The usual tflite library for an architecture is about 700+ kb. Following are the sizes of the “.so” file which is imported into project for prediction using tflite:

1)x86: 1.1 MB

2)arm64-v8a: 794 kb

3)armeabi-v7a: 701.8kb

4)x86_64: 1.1 MB

So even though the model size is less, we have to import these big dependencies which would increase the overall size of App . An approach was taken to reduce the size of this library by stripping it off any unrequired operations(since the model used doesn't entail all operations).

We built a custom tflite library for only armeabi-v7a architecture(covers all Android devices).

The following basic operations(ops) are supported by this Custom tflite library:

1)Relu

2)Softmax

3)Fully Connected Layer

4)Mean operation

5)Gather(For slicing)

6)Cast(for changing the variable type)

The resultant size of the tflite library is **268.9 kb**.

Dynamic Installation of models

We have explored three platforms to host our ML models and implemented the dynamic installation of them in the app during runtime.

1) FTP Server

2) Azure Blob Storage

3) One Drive

The FTP Server had the following cons associated with it :

- The IP Address of the machine has to be made public
- The machine has to be kept on for all the time

The Azure Blob Storage doesn't entail the above cons but had the following disadvantage

- An extra dependency of about 300kb has to be imported in the project, and hence adds to the size hit.

The implementation using One-Drive doesn't face any of the above cons. Hence One-Drive is used in our final implementation. When the app opens, the models begin to download asynchronously in the external directory of the app , which is app-specific and can't be accessed by other apps, and notify the user when the download is complete. Further they are automatically deleted when the app is uninstalled.