AI-chatbot for Norwegian

Done by **Majdi O. Alali**

Table of Contents:

1. introduction:

In this project we are going to build an AI chatbots for Norwegian language. The main goal of this chatbot is to answer on general questions of who are learning Norwegian as a second or third language. It is meant to help them make some natural conversations. But as the data grows and varies, we can use this chatbot for other purposes like for industrial uses. To build this chatbot, we are going to use a neural network together with some Information Retrieval techniques and NLP tools. More specifically we will create a neural network which can classify the user input and return a response back to the user. As a part of the preparing of the data, we will use NLP-tools like lemmatizing. At the end of the preprocessing stage, we are going to use two approaches to numerically represent our data. The first one is Bag of Words, an IR technique, while the second one is called tf-idf, term frequency-inverse document frequency, an NLP-algorithm. We will also do some adjustment on the user input, when necessary, to optimize the model where we will make a spell checker from scratch. This spell checker is capable to detect the misspelled Norwegian words and suggest words to substitute them. This can give better perdition of the classes. Consequently, it returns qualitive responses to the user.

2. Literature:

Chatbots are an NLP-application which simulate natural conversations of human being. It allows humans to communicate with the machines. ELIZA was the first chatbot in the history. It was created by Joseph Weizenbaum in 1966 [**1**]. At the earlier time, chatbots was text-based. But later, they got developed to communicate via voice. Many businesses appreciate them so much because they could help to save money and time and give better customer service for their business. In addition, they give fast responses and work all around the clock. The top domains in using chatbots are healthcare, personalized marketing, banking, travel, and hospitality [**1**].

The text-based chatbots has three main types:

1) Rule-Based Chatbots:

This type is based on a tree-like approach and fit well for specific queries. In other words, the bot responses are mainly hardcoded depending on the expected queries. The benefit of these chatbots is that they are precis in a way they to answer on queries they do not know about.

2) IR chatbots (with or without AI)

This type depends on searching and matching techniques to give responses. These chatbots can answer on wider queries than the rule based Chatbots. But they are less precis than rule-based chatbots. However, they can still give good responses especially if they are used in combination with traditional AI, NLP, or deep learning.

3) generative AI-chatbots:

They are capable to generate new responses, on the contrary of Rule-based and IR chatbots which are limited to data sets. They combine different techniques from both supervised, unsupervised learning, and even reinforcement learning. There is little research about them and testing them seems to be a very challenged issue. However, they sound promising.

3. Data sets

We have three data sets. The first and the most important one is in a form of Json file. It is called intents, more specifically many-to-many intents [***figure 1***]. This type of data sets, intents, is structured in a way it associates between similar utterances in a particular context. An intent contains a tag name (=class), a set of patterns, and a set of responses. The patterns are what our network will train on to predict the tag, or the class, of the user input. Then it returns a random response of the related class. We did not find a chatbot for Norwegian language. So, we had to build one from scratch. Therefore, this data set is somehow limited to 15 intents. We want to add that we tried to make the data set as a balanced as possible where we limited the number of patterns to max 4 ones pr intent. At the same time, we embedded or combined more some patterns together for some intents. But carefully in this approach because if we include more than two patterns or two long patterns the model will fail to detect the right class of the user's input in many cases. Regarding the responses, they do not need to be balanced because they will not be pushed to the network. Our chatbot will just pick up one of them randomly and send it to the user after predicting the class. Another thing to mention here is that We limit our self to Bokmål variation of Norwegian because it is more common than other dialects. The second data set is a Norwegian dictionary in form a txt-file. It contains around 350k

Norwegian words. This dictionary is especially useful for our spell checker to detect whether the user input has misspelled words or not. In other words, if one or more words of the user input does not exist in the dictionary [**2**] then it or they will be considered like misspelled words. Then the checker generates strings of these words in a try to find exactly the words which the user means by filtering out the strings which are not words, and keep a set of candidates of every misspelled word. Now our spell checker/corrector needs to choose just one candidate to substitute every misspelled word. Here the third data set comes to the picture. The data set contains about 191k lines collected from about 30 films and series subtitles in Norwegian [**3**]. In the preprocessing stage, we will make frequency table from the words of the subtitles. This makes it easy for our spell checker to choose one candidate to substitute the misspelled word by knowing which of them has highest frequency in the subtitles. One might also wonder why we have chosen data from subtitles. Because they have natural conversations which are perfect for our main goal, helping to get familiar with everyday language.

```
,{
  "tag": "hilsning_bot",
  "patterns": [
     "Går det bra?","Åssen | Hvordan går det?","Hvordan har du det?", "står til?"],
  "responses": [
   "Opp og ned! Takk for at du spør!","Det går fint, takk. Hva med deg?",
      "ikke verst!", "Det var ikke best. Men nå er det bedre! :)"]
},
```

Figure 1*:* one intent

4. Methodology

4.1. Preparing the data

After installing and importing the necessary packages and dependencies, we need to read the intents data set. We made three functions. The first one is for tokenizing or splitting strings. The function receives a string as a parameter and returns a list of the tokens like words. The second one is mainly for lemmatizing a given string. Lemmatizing is a linguistical term for dealing with word inflection. It returns a word to its base form by removing morphological adds (called also affixes) like for example the base form of the verb "gikk" is "gå", the base form of the adjective "bedre" is "god", and the base form of the noun "studenter" is "student". The reason why we need lemmatizing is that we do not want our network to look at for example "går" and "gikk" as if they are two different words. Back to our function, it does some preparing of data like removing punctuation marks and digits. Before lemmatizing and returning list of lemmas (=base form of the words), our function does one special thing. It avoids lemmatizing the first and second personal pronouns (=the speaker and the listener) like "jeg", "du", "ditt", "mitt" osv because our Norwegian lemmatizer when it sees a pronoun it returns "-pron-". In other words, it looks the same at all pronouns. But we want that our network looks different at user utterances like "Du heter?" and "Jeg heter" because these two utterances have different classes where the chatbot response on the first utterances should be something like "Jeg heter Chatto. Du da?" while the response on the second utterance should be like "Hyggelig!".

4.2. Bag of words:

BOW is an IR model which allows a numerical representation of an arbitrary text based on the vocabulary (=the unique words) of a corpus to which a text belongs to. So, if a word of the text exists in the vocabulary, then it will be represented by 1 otherwise 0 for every word in the text and for every text which belongs to the corpus. This means that the model pays no attention to grammar or the order of the words. All what does matter for it is wether a word exits in the vocabulary or not. Therefore, our function receives two parameters. The first one is a string or a sentence and the second on is a list of vocabulary/unique words of all strings in our corpus which the sentence is a part of. Then it returns a list of binary numbers. The size of the returned list is equal to the size of the vocabulary.

Now we will move to talk about how we combined these functions together for preparing the training data. First, we made tree lists, *X_res,y_re*, and *uniq_words. X_res* takes care of our textual data points and *y_re* takes care of the labels of these data points. While *uniq_words* take care of our vocabulary. Then we loop over intents and every pattern of every single intent and do the following: tokenizing the pattern, extracting the unique words, and at the end filling in the three lists. When we are done with looping, we lemmatize the vocabulary and sort them alphabetically. Then we convert our textual data points to binary numbers via the bag of words function. Then we get a matrix of our data points, and a matrix of their classes(=tags) [***Figure 2***]. Both matrixes are converts to NumPy array taking in consideration the parameters of our network in the next stage. Please notice also that we have 15 classes, and every intent belongs to one and just one. Therefore, our *y_train* has 2-dimenions.

```
X_train

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

```
y_train

array([[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
```

Figure 2: training data

4.3. Tf-idf

Tf-idf is another approach of representing the data numerically. It is a simple algorithm which is used to weight the importance of words in a set document of a corpus they belong to. In other words, it weights the words on cross of all the documents in a corpus. The less frequent word is the more important word is and vice versa. It shows some understanding of the context of the words, on the contrary of Bag of Words which look at the words as if they are equally important. This algorithm is widely used in IR and NLP. It was used intensively by Google search before the discovery of the transformers. Regarding the implementation, we make a function which receive a list of strings (=the patterns in our case) and return a data frame where the documents(=patterns) are the rows and the vocabulary stands for the columns [***Figure 3***], on the contrary of the bag of the words. Therefore, the preparing of the training data a little different from that in the bag of the words. Then it is necessary to hot- hot-encoding on our data to convert them from categorical into numerical values. After that we build the network exactly as we did earlier in BOW.

```
tf_idf(["Hei", "Hei du",  "Hvor gikk du", "Hva heter du"])
```

	du	gikk	hei	heter	hva	hvor
0	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000
1	0.629228	0.000000	0.777221	0.000000	0.000000	0.000000
2	0.411378	0.644503	0.000000	0.000000	0.000000	0.644503
3	0.411378	0.000000	0.000000	0.644503	0.644503	0.000000

Figure 3: a smaller example

4.4. Network building

After preparing the data, we are ready to build a neural network. First, it is worthy to mention that some AI-experts think that the fewer layers and neurons a model has the more reliable and interpretable is so long this simplicity of the model does not lead to underfitting [**4**]. Therefore, we tried to make our network as simple as possible where our network consists of just 3 layers. Having just one hidden layer helps us to avoid overfitting, make it easier for optimization, and compute more efficiently. Consequently, the model will be able to generate to new data. We tried 4 layers, but they give almost the same accuracy with some irregularity regarding the loss function over epochs. Regarding number neurons, our input layer has neurons as many as the features of our training data have. our hidden layer has 15 neurons, as many classes as we have in the data. The activation function we use in the hidden layer is Relu, rectified linear activation unit. It is known to work best with hidden layers because it helps the model to avoid linearity and is efficient computationally. The number of neurons in the output is equal the number of the classes as long we use softMax. It is also natural in the backpropagation step to choose the categorical cross-entropy loss function and SoftMax activation function when dealing with multi-class problem. Regarding the optimizer in this stage, we go for Adam because it is simple and fast, and it combines the advantages of both AdaGrad and RMSProp. Therefore, Adam is most popular optimizer is deep learning. One more reason why we choose Adam is that it is compatible with the cross-entropy loss function. Now our network is ready for training. After some experimenting in fitting stage, we found out that 400 epochs work very good for our data. The accuracy of the model and loss function are somehow optimal [***Figure 4***]. At the end we save our model using pickle package to avoid retrain the model every time we want to use it.

```
2/2 [==============================] - 0s 5ms/step - loss: 0.1418 - accuracy: 1.0000
```

Figure 4: accuracy

4.5. Spell checker

It is natural that we, human being, make spelling mistakes. However, our network will not understand what we mean. Consequently, that will result in bad performance of our model. Therefore, we need a spell checker which detects these mistakes and correct them before to push the numeric values of our textual input to the network. So, the spell checker will help a lot in optimizing the model. Moving to explain about our implementation, we made a function which receives a file (subtitles in our case), reads, and cleans it [***Figure 5***]. The cleaning process includes removes digits and special chars like punctuation marks and removing timeline and line-break from series and films. In addition, it arises error exception when needed.

Figure 5: subtitles cleaned

After that we have to find the frequency of the subtitles' vocabulary where we need to use them to decide which of the candidate words, extracted from the generated strings, will substitute the misspelled word. The same is applied for all misspelled words of the user's input.

We made a class called spell checker, build totally from scratch. This class has 10 methods. We want to pay your attention that there are a lot of details in the implementation of this class, but we are not going to explain all of them to make this report easy to read. The class receives the user input as a list of the cleaned words (=tokenized and lemmatized words). The first one is for reading the Norwegian dictionary file and extracting the words. The second one checks if there is one or more misspelled words in the user input by matching them with the Norwegian words in the dictionary. In other words, if a word is not in the dictionary, then it will be considered as a misspelled word.

The next four functions work as one package for generating new strings from the misspelled words. Every one of them receives one string (=a misspelled word) and returns a list of the generated strings. The first one of this package inserts every Norwegian letter (one letter per time) before and after every single letter in the misspelled words. Let say for example that the misspelled word is "åg" for "gå". So, the generated strings should look like as follows:

['aåg', 'båg', 'cåg',..............., 'åag', åbg', 'åcg',…….........., 'åga', ågb', 'ågc',..................]

The second one is simpler. Given a misspelled word, it just deletes one letter from the word pr time, if the word consists of more than two letters, and returns the generated strings. The figure down is an example on how this method works. In a later stage, we are going to remove the duplicated strings.

```
delete("lyys")

 ['yys', 'lys', 'lys', 'lyy']
```

Figure 6: delete function

The third one replaces every letter of the misspelled word with every Norwegian letter, again one letter pr time. Let us suppose that the misspelled word is "sor". So, the generated strings should look like as follows:

['aor', 'bor', 'cor', ..., 'sar', 'sbr', 'scr', 'soa' 'sob', 'soc',....]

The last method in this package is used for swapping the letters of a misspelled word. In other words, change the place of a letter with its neighbor letter. The figure down is an example on how this method works.

```
swap("safmunn")

['asfmunn', 'sfamunn', 'samfunn', 'safumnn', 'safmnun', 'safmunn']
```

Figure 7: swap function

The next function in the class is called, *check one*. What it does is simply putting together the previous fourth functions. In other words, it receives a misspelled word, and generate all possible strings of this misspelled word by using the previous functions. The function which follows this is called *check all*. It builds on the previous one and all other previous functions as well. By calling the function which detects all misspelled words in the user input, and the previous one, it generates all possible strings of all misspelled words of the user input. The function which comes next extracts, from the generated string, the strings which are words and exclude the meaningless strings. This is done by checking whether a string is in our Norwegian dictionary or not. The tenth and last function in this class is called *auto_correct*. It is somehow the main function where it puts all functions of the class together. It starts by checking if there are misspelled words. If not, it returns the same input. Otherwise, it calls the function which generate all possible string of all misspelled words. Then extract the words from these strings via *wrds_frm_strs* function. If there is just one candidate word, then it is the word which will replace the misspelled. Otherwise, the method selects the candidate with highest frequency depending on the subtitles if this word is in the vocabulary of the subtitles. In a case, the word is not in the subtitles frequency table then the word is given 1 as its frequency. This process is applied for every single misspelled word. At the end the method returns a list of the chosen candidates for the misspelled words. Here the relation between misspelled words and selected/chosen candidates is one-to-one [***Figure 8***].

```
sc=spell_checker(["hvro","gammmel","være","du"])
print("misspelled words: ", sc.find_misspelled_ws())
print("selected candidates: ", sc.auto_correct())

misspelled words:  ['hvro', 'gammmel']
selected candidates:  ['hvor', 'gammel']
```

Figure 8: test spell checker

4.6. **The main function (BOW)**

The main function connects the different parts of the project together to run the chatbot. It starts by a while loop to hold the dialogue going as long as the user wants. If the user type "q" then the while loop will stop with a nice goodbye. Here there is an another if-check which puts limits on number words user can type in, less than 10 words is ok. Otherwise, it tokenizes and lemmatize the user input. Then it checks if there are misspelled words which need to be corrected before to convert the input into numerical values and feed it to the network which predicts the tag of this input and return a random response of the related responses for the intent. We want to mention that the bot keeps truck on the previous input. So long as we use SoftMax activation function, the output is probability distribution of the classes. So, we, via trial and fail, found out that the predicted class should have at least 40% of this distribution to give a good response from the related intent. Otherwise, the bot will say “Sorry I don’t understand the question, try with other words!” So the threshold is 40%.

5. Testing and the results

In general, our chatbot performs outstanding good if we take in consideration the small training data set (just 15 topics or intents) and the simple network. Regarding the testing step, we found out that splitting the intents data into ‘train’ and ‘test’ is not a good idea. The reason is that the patterns of the same intent or class can have totally different words. So, the model will not be able to detect the common features between these patterns although they belong to the same tag or class. Consequently, the is a high probability that the model will fail to classify them correctly in the training stage. For example, let us assume that a pattern like “Hei” is in training data while “Halla” in test data set when splitting data. Although both patterns belong to the same intent “hilsning”, the model will not be able to classify Halla correctly just by knowing the class of “Hei”. In other words, patterns of the same intent can be mutually exclusive /uniquely identified. So, it is not surprising to have underfitting problem as the figure down shows. Here the data set was divided intov10% test and 90% train. We can observe also that in epoch 26 the is very good fit between train and test. It seems that in test there is patterns like “tusen takk” and in “takk skal du ha” or “hjertlig takk”.

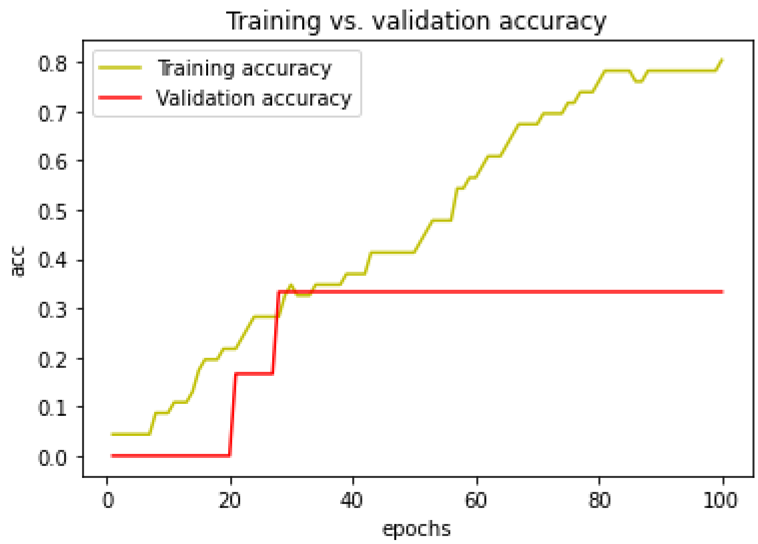


Figure 9: test vs. train accuracy

So, testing a chatbot is not an easy task because human languages like Norwegian have an amazing complex phenomenon. But anyway, we invented some standards which can help us to test a part of the performance of our chatbot:

1) **Repetitive words:** In natural conversations, some people tend to repeat some words.

This is not problometic to our chatbot. So, it passes this test.

```
You: takk takk!
   prob. of predicted class: 0.81 %
Bot:  Vær så god!
```

2) **negation**: It is also natural in daily conversations to say the opposite of a state. Our chatbot fails in this part because it uses IR matching techniques, not really on the context. To make the chatbot comes over this problem we need to include sentimental analysis.

```
You: Jeg er ikke syk
   prob. of predicted class: 0.77 %
Bot:  God bedring!
```

3) **different word order:** Here we mean changing the order of the words where it is different from the order of the matched pattern. The chatbot passes this part as well because what it does matter for it is the existence of the words not the order. In our example the matches pattern is “Går det bra?”

```
You: Det går bra nå?
   prob. of predicted class: 0.95 %
Bot:  Det var ikke best. Men nå er det bedre!
```

4) **Grammatical variation:** What we mean by that is using different tenses like the past tense instead of the present. In our example, the matched pattern is “Hva heter du?” The chatbot passes this test as well because it lemmatizes the input of the user before to send it to the network. So “var” and “er” will be lemmatized to “være”.

```
You: hva var navnet ditt igjen
   prob. of predicted class: 0.88 %
Bot:  Du kan kalle me Chatto. Du da?
```

5) **fragments**: Here we mean not typing full sentences. So long as at leat 40% of the words exist in the matched pattern then the chatbot will success to predict the class and give a good response.

```
You: språk du
   prob. of predicted class: 0.44 %
Bot:  Jeg kan norsk.
```

6) **Repeat a question:** Some tends to repeat the same question but in a different way. The chatbot passes this test if in the data there is a pattern which has some words in common with the new input.

```
You: går det bra
Bot:  Opp og ned .. Takk for at du spør!
You: Hvordan har du det
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og ditt forrige spørsmål.
```

7) **Misspelling**: After building a spell checker our chatbot was able to pass the test of misspelling when the problem of the misspelled words is related to one letter like there is an extra letter, miss one letter, swap two neighbor letters, and replace one letter with another in a word. As we saw earlier, the spell checker helps in optimizing the model. Let us take one example about misspelling. Assuming that we have a user input like: “Hvodan hra duu ”. The first misspelled word misses one letter, the second one has a swapped letter, and the third one has one extra letter. Our spell checker in this example can guess the words which the user exactly means. Please pay attention that “ha” is the infinitive word of “har” and this is exactly what we expect and what our network needs to perform well.

```
[54] sc=spell_checker(["hvodan", "hra","duu"])
     selected_ws=sc.main()
```

```
[55] print(selected_ws)

     ['hvordan', 'ha', 'du']
```

```
You: hvodan hra du

Bot: Jeg forstår ikke helt spørsmålet. Prøv gjerne igjen med andre ord.
```

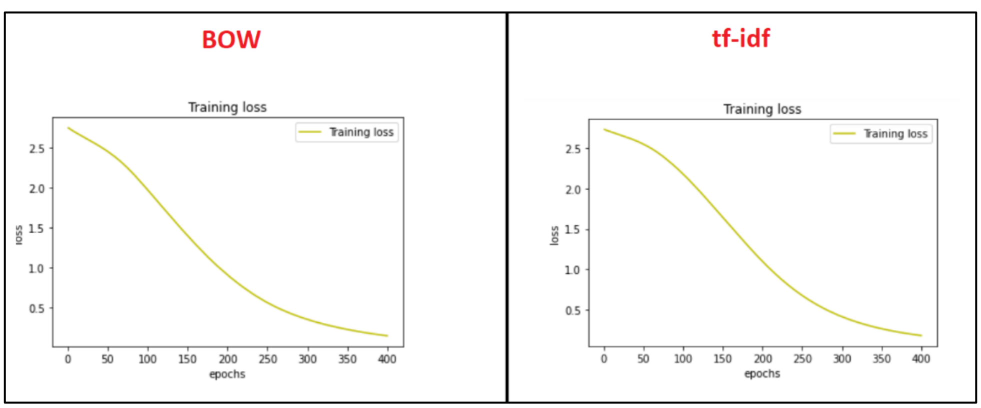
without using spell checker

```
You: hvodan hra duu
   prob. of predicted class: 0.72 %
Bot:  ikke verst!
```

with using spell checker

Bag of words vs. tf-idf

Both BOW and tf-idf seem to perform very well according to the training accuracy, and as the figure down shows they have very similar training accuracy and loss. However, in our case BOW shows to give alittle better responses than tf-idf as the figure down shows.



However, tf-idf will work much better when we have large data sets because it shows some contextual understanding, on the contrary of BOW. Bag of words work best and gives better accuracy when the number of matches between training data/patterns and user's input is high while tf-idf works best with unique and/or long utterances but the threshold needs to be around 50%. We mean by threshold is that the probability of predicted class needs to be around 50% for that our chatbot outputs good responses. While the threshold is 40% in BOW.

tf-idf

```
You: hei
prob. of predicted class:  0.66862917
Bot:  Hei, hva skjer'a?
You: din alder
prob. of predicted class:  0.4356751
Bot:  Jeg er født i april 2022. Men lærte mye i kort tid!
You: gammel er du
prob. of predicted class:  0.4991962
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og d
You: hva heter
prob. of predicted class:  0.5928392
Bot:  Hyggelig!
You: heter du
prob. of predicted class:  0.5928392
Bot: Jeg klarer ikke se forskjell på det*e spørsmålet og d
You: Hva heter du
prob. of predicted class:  0.5928392
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og d
You: ditt navn
prob. of predicted class:  0.8028593
Bot:  Du kan kalle me Chatto. Du da?
You: kalle deg
prob. of predicted class:  0.475677
Bot:  Hyggelig!
You: Navnet mitt
prob. of predicted class:  0.7235978
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og d
```

bag of words

```
You:  hei
WARNING:tensorflow:Model was constructed with shape (None, None,
prob. of predicted class:  0.8459763
Bot:  Hallo!
You: din alder
prob. of predicted class:  0.6528842
Bot:  Jeg er født i april 2022. Men lærte mye i kort tid!
You: gammel er du
prob. of predicted class:  0.54937917
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og ditt fo
You: hva heter
prob. of predicted class:  0.40189612
Bot:  Du kan kalle me Chatto. Du da?
You: : heter du
prob. of predicted class:  0.40189612
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og ditt fo
You: Hva heter du
prob. of predicted class:  0.40189612
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og ditt fo
You: ditt navn
prob. of predicted class:  0.42121238
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og ditt fo
You: kalle deg
prob. of predicted class:  0.49841255
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og ditt fo
You:  Navnet mitt
prob. of predicted class:  0.4150398
Bot:  Hyggelig!
```

```
prob. of predicted class:  0.48654082
Bot:  Nei, er singel. La meg vite hvis du er interessert
You: jobber
prob. of predicted class:  0.60118943
Bot:  Jeg jobber some en fulltids-chatbot, vettu! :)
You: faen
prob. of predicted class:  0.49472356
Bot:  Faen! Hvorfor banner du!?
You: føkk
prob. of predicted class:  0.5089824
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og
You: fy
prob. of predicted class:  0.63227177
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og
You: hore
prob. of predicted class:  0.4809553
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og
You: kuk
prob. of predicted class:  0.46850058
Bot: Jeg klarer ikke se forskjell på dette spørsmålet og
You: hvor kommer
prob. of predicted class:  0.8445916
Bot:  Jeg er norsk. Men min utvikler er syrisk.
You: kommer du
prob. of predicted class:  0.2073331
Bot: Jeg forstår ikke helt spørsmålet. Prøv gjerne igjen
You: kommer du fra
prob. of predicted class:  0.9200743
```

```
prob. of predicted class:  0.3773222
Bot: Jeg forstår ikke helt spørsmålet. Prøv gjerne igjen med andre ord.
You:  jobber
prob. of predicted class:  0.41979626
Bot:  Jeg jobber some en fulltids-chatbot, vettu! :)
You: faen
prob. of predicted class:  0.25183406
Bot: Jeg forstår ikke helt spørsmålet. Prøv gjerne igjen med andre ord.
You: fy
prob. of predicted class:  0.2874607
Bot: Jeg forstår ikke helt spørsmålet. Prøv gjerne igjen med andre ord.
You: hore
prob. of predicted class:  0.16293332
Bot: Jeg forstår ikke helt spørsmålet. Prøv gjerne igjen med andre ord.
You: kuk
prob. of predicted class:  0.25470248
Bot: Jeg forstår ikke helt spørsmålet. Prøv gjerne igjen med andre ord.
You: hvor kommer
prob. of predicted class:  0.46274632
Bot:  Jeg er norsk. Men min utvikler er syrisk.
You: kommer du
prob. of predicted class:  0.2205568
Bot: Jeg forstår ikke helt spørsmålet. Prøv gjerne igjen med andre ord.
```

6. Limitation and development

- **limitations**

- Our spell checker id built to look at words in isolation. So given a sentence like “Grå det bra?” it will not recognize the grammatical problems. A good solution can be using an nlp-algorithm called Dependency Trees or transformers for more advance use.

- Our Norwegian dictionary is just for Bokmål. So, it is nice to include Nynorsk. This task seems to be not that difficult but computationally expensive.

- The Norwegian lemmatizer, compared with the English one, is very simple and give very few grammatical classifications or the part of speech tags. For example, it looks the same on the different type pronouns -pron- while the English lemmatizer gives precis description of a pronoun. Anyway, it is ok for our case but if we want to build advanced chatbot, we need a better language model than this.

-Again, the Norwegian lemmatizer, given by SpaCy, makes some simple error. Sometimes it lemmatizes wrongly some words like “gåre” instead of “gå”. Unfortunately, we cannot do something with that.

-One problem we faced during testing the spell checker is that the checker looks at proper nouns like misspelled words. However, this seems to be manageable task because we find out a data set for Norwegian names. All what we need to do is to tell the checker if you meet a proper noun do not consider it a misspelled word.

- **Further development:**

Our priority when thinking about further development of this project is extending the size of the data set and test other types of data sets like one-to-one intents. In the second place, we would test some transformers libraries like NorBert to keep truck on whole the dialogues with uses to better understand the context of their queries. Then we would make a suitable interface and integrate the chatbot in a website. In addition, we will try to deal with the limitations mentioned earlier. We would also involve other languages. We do not need to train the model on other languages. We could do a small trick by using API-es. Let us say that a user chooses to communicate with the chatbot in polish. What we can do is sending a request to for example Google translate in order to get the text translated to Norwegian. Then the response from our model, given in Norwegian, gets sent to Google translate to be translated to polish. So, the user will get the response in Polish. Another improvement can be thought about is involving the user in evaluating the chatbot and let the chatbot stores those dialogues with users to improve its performance, taking in consideration GDPR. Another idea is to make the chatbot proactive like to take initiative by asking related questions which can be interesting for the use. Maybe we can implement that by using for example parallel programming or similar technologies. Last but not least, we would involve more NLP technique like NER, name entity recognition. This can help the chatbot to recognize places, and persons.

7. Sources

[**1**] Literature sources

https://www.sciencedirect.com/science/article/pii/S2666827020300062#:~:text=In%201950%2C%20Alan%20Turing%20wondered,name%20was%20constructed%20in%201966.

https://blog.cloudboost.io/generative-vs-retrieval-based-chatbots-a-quick-guide-8d19edb1d645

[**2**] Dictionary data

https://github.com/titoBouzout/Dictionaries/blob/master/Norwegian%20(Bokmal).txt

[**3**] Subtitles collected from

https://www.opensubtitles.org/en/search/sublanguageid-nor

[**4**] Model complexity:

https://towardsdatascience.com/simplicity-vs-complexity-in-machine-learning-finding-the-right-balance-c9000d1726fb