CREATE A CHATBOT IN PYTHON

aut104302-M.SIVAVETRIVEL

Development part -1

Introduction

1. **Natural Language Processing (NLP)**: This is a critical component of chatbots. NLP enables the bot to understand and interpret user messages, allowing it to extract meaning and context from text or speech inputs.
2. **Machine Learning and AI Algorithms**: Chatbots often utilize machine learning algorithms to improve their performance over time. They learn from the interactions they have with users, allowing them to provide more accurate and relevant responses.
3. **User Intent Recognition**: Chatbots are trained to recognize the intention behind a user's message. For example, if a user asks about store hours, the bot needs to identify the intent of the question and respond appropriately.
4. **Response Generation**: Based on the user's input and the identified intent, the chatbot generates a relevant response. This can range from simple pre-defined answers to dynamically generated responses based on the context.
5. **Dialog Management**: Chatbots need to maintain context within a conversation. They should remember previous interactions and use that information to generate appropriate responses.
6. **Integration with Systems and APIs**: Depending on the use case, chatbots may need to interact with external systems or APIs to retrieve or update information. For example, a chatbot in an e-commerce setting might need to check product availability.
7. **User Experience Design**: The design of the chatbot's interface (e.g., chat window, voice interaction) and the flow of conversation are crucial for a seamless user experience.

8. **Testing and Evaluation**: Before deployment, chatbots go through extensive testing to ensure they understand a wide range of user inputs and provide accurate responses. User feedback is also important for ongoing improvement.
9. **Ethical Considerations**: Ensuring that chatbots operate ethically is important. This includes considerations about data privacy, transparency, and the potential biases that may arise in AI models.
10. **Deployment Platforms**: Chatbots can be deployed on websites, messaging platforms like Facebook Messenger or WhatsApp, mobile apps, and other channels.
11. **Maintenance and Updates**: Once deployed, chatbots require ongoing maintenance to keep them up-to-date with changes in user behavior, business processes, and technology.

Chatbots have become increasingly sophisticated in recent years, thanks to advancements in AI and NLP technologies. They play a significant role in automating tasks, improving customer service, and enhancing user engagement in various industries.

Table Content

These three methods can greatly improve the NLU (Natural Language Understanding) classification training process in your chatbot development project and aid the preprocessing in text mining. Below we demonstrate how they can increase intent detection accuracy.

```
!git clone https://github.com/interds/3-methods-of-nlu-data-pre-processing.git
```

```
%cd ./3-methods-of-nlu-data-pre-processing
```

```
!apt-get install python3-venv
```

```
!python -m venv --system-site-packages ./venv
```

```
!source ./venv/bin/activate
```

```
!pip install rasa[transformers]
```

```
!pip install -U ipython # fix create_prompt_application
```

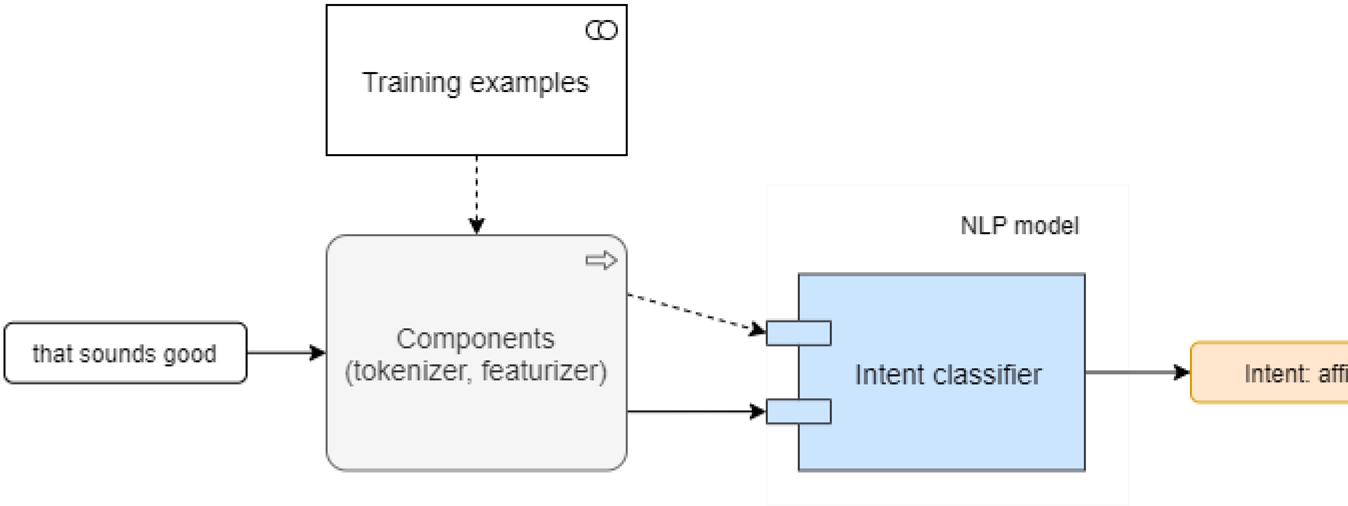
```
!pip install pandas
```

```
!pip install chatette
```

```
!pip install transformers
```

```
!pip install tensorflow_datasets
```

Initial model



Rasa's boilerplate generated by '*rasa init*' is enough to demonstrate the initial model in our chatbot development effort.

We train and evaluate the model with the following config:

```
language: en
```

```
pipeline:
```

```
  - name: WhitespaceTokenizer
```

```
  - name: CountVectorsFeaturizer
```

```
  - name: CountVectorsFeaturizer
```

```
    analyzer: "char_wb"
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 100
```

```
!rasa train -c config-simple.yml --fixed-model-name simple --quiet
```

```
Training Core model...
2020-06-22 20:46:54.811928: E tensorflow/stream_executor/cuda/cuda_dr
iver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-cap
able device is detected
Core model training completed.
Training NLU model...
```

```
/usr/local/lib/python3.6/dist-packages/rasa/utils/common.py:363: User
Warning: You specified 'DIET' to train entities, but no entities are
present in the training data. Skip training of entities.
```

```
NLU model training completed.
```

```
Your Rasa model is trained and saved at '/content/models/simple.tar.g
z'.
```

```
!rasa test nlu -c config-simple.yml -u test_data.md -m models/simple.
tar.gz --out results/simple --quiet
```

```
report = pd.read_json("results/simple/intent_report.json", orient="va
lues")
```

```
simple_f1 = report["weighted avg"]["f1-score"]
```

```
data = [["simple", simple_f1]]
```

```
pd.DataFrame(data, columns=["Model", "F1-sore"])
```

```
!rasa test nlu -c config-simple.yml -u test_data.md -m models/simple.
tar.gz --out results/simple --quiet
```

```
report = pd.read_json("results/simple/intent_report.json", orient="va
lues")
```

```
simple_f1 = report["weighted avg"]["f1-score"]
```

```
data = [["simple", simple_f1]]
```

```
pd.DataFrame(data, columns=["Model", "F1-sore"])
```

```
2020-06-22 21:00:29.891699: E tensorflow/stream_executor/cuda/cuda_dr
iver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-cap
able device is detected
```

```
100% 14/14 [00:00<00:00, 108.22it/s]
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classificatio
n.py:1272: UndefinedMetricWarning: Precision and F-score are ill-defi
ned and being set to 0.0 in labels with no predicted samples. Use `ze
ro_division` parameter to control this behavior.
```

```
  _warn_prf(average, modifier, msg_start, len(result))
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classificatio
n.py:1272: UndefinedMetricWarning: Precision is ill-defined and being
set to 0.0 in labels with no predicted samples. Use `zero_division` p
arameter to control this behavior.
```

```
  _warn_prf(average, modifier, msg_start, len(result))
```

```
   Model      F1-sore
```

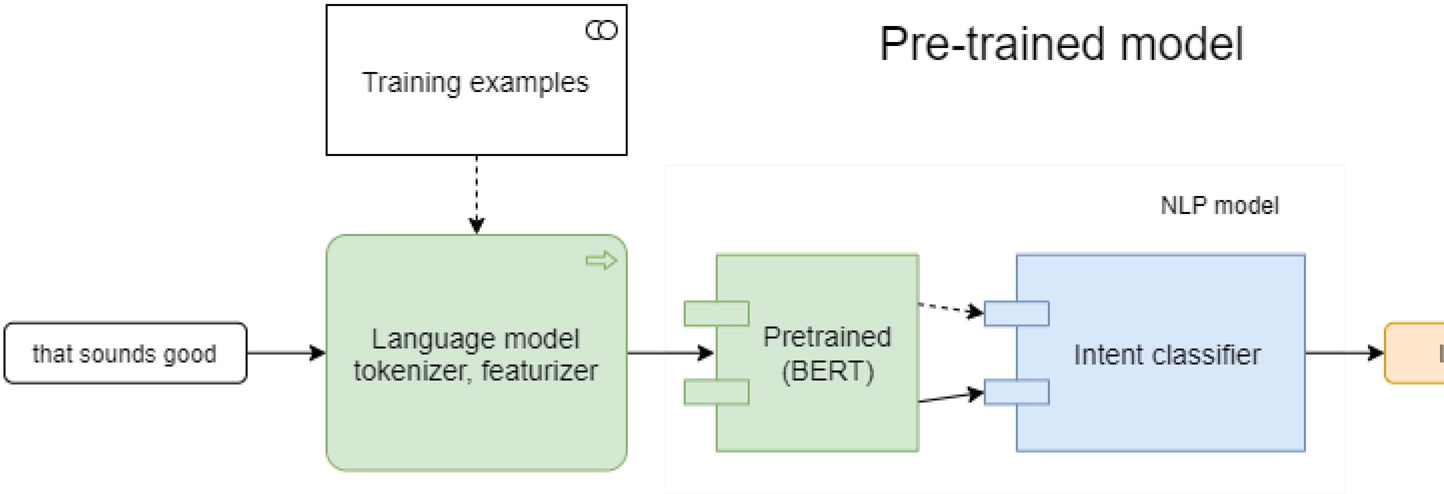
```
0  simple  0.614286
```

Expected F1-score = 0.752381

In test data we have lexically different examples from the ones in training data, so it is expected that our simple pipeline doesn't recognize them properly:

```
intent:affirm- alright- sure- ok
```

1. Pre-trained model



The pre-trained language model can be used for NLU tasks without any task-specific change to the model architecture. Pre-trained models have an ability to continue pre-training on custom data, starting from some checkpoint.

Hire talented developers from LATAM, Canada, and Europe

```
language: en

pipeline:

  - name: HFTransformersNLP

    model_weights: "bert-base-uncased"

    model_name: "bert"

  - name: LanguageModelTokenizer

  - name: LanguageModelFeaturizer

  - name: DIETClassifier

    epochs: 100
```

```
!rasa train -c config-bert.yml --fixed-model-name bert --quiet
```

```
Core stories/configuration did not change. No need to retrain Core model.

Training NLU model...

Downloading: 100% 232k/232k [00:00<00:00, 1.93MB/s]

Downloading: 100% 433/433 [00:00<00:00, 299kB/s]

Downloading: 100% 536M/536M [00:08<00:00, 63.4MB/s]

2020-06-22 20:48:18.155538: E tensorflow/stream_executor/cuda/cuda_driver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected

/usr/local/lib/python3.6/dist-packages/rasa/utils/common.py:363: UserWarning: You specified 'DIET' to train entities, but no entities are present in the training data. Skip training of entities.

NLU model training completed.
```

```
Your Rasa model is trained and saved at '/content/models/bert.tar.gz'.
```

```
!rasa test nlu -c config-bert.yml -u test_data.md -m models/bert.tar.gz --out results/bert --quiet
```

```
report = pd.read_json("results/bert/intent_report.json", orient="values")
bert_f1 = report["weighted avg"]["f1-score"]
data = [["simple", simple_f1], ["bert", bert_f1]]
pd.DataFrame(data, columns=["Model", "F1-sore"])
```

```
2020-06-22 20:49:03.856455: E tensorflow/stream_executor/cuda/cuda_driver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
```

```
100% 14/14 [00:03<00:00,  4.04it/s]
```

```
   Model    F1-sore
```

```
0  simple   0.614286
```

```
1  bert     0.930612
```

Expected F1-score = 0.930612

As we see, without modification of training data, usage of the pre-trained BERT model improves the accuracy of intent detection. This happens because the model already has knowledge about word's synonyms, which helped to recognize matches.

Fine-tuning your AI chatbot

To perform Fine-tuning of the chatbot development model, follow the instructions on Sentence (and sentence-pair) classification tasks from Google's BERT repository. In general, you need to download some text corpus or to convert your text data to BERT's input format, then run Fine-tuning command. You can prepare a new model with the following script:

```
from transformers import TFBertModel, BertTokenizer
```

```
model = TBertModel.from_pretrained("bert-base-uncased")
```

```
model.save_pretrained("./model-fine-tuned-1/")
```

```
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
```

```
tokenizer.save_pretrained("./model-fine-tuned-1/")
```

Follow the text preprocessing steps for fine-tuning. An example of Fine-tuning Bert model on the MRPC classification task is given below:

```
export BERT_BASE_DIR=/path/to/bert/uncased_L-12_H-768_A-12
```

```
export GLUE_DIR=/path/to/glue
```

```
python run_classifier.py \
```

```
  --task_name=MRPC \
```

```
  --do_train=true \
```

```
  --do_eval=true \
```

```
  --data_dir=$GLUE_DIR/MRPC \
--vocab_file=$BERT_BASE_DIR/vocab.txt \
```

```
--bert_config_file=$BERT_BASE_DIR/bert_config.json \
```

```
--init_checkpoint=$BERT_BASE_DIR/bert_model.ckpt \
```

```
--max_seq_length=128 \
```

```
--train_batch_size=32 \
```

```
--learning_rate=2e-5 \
```

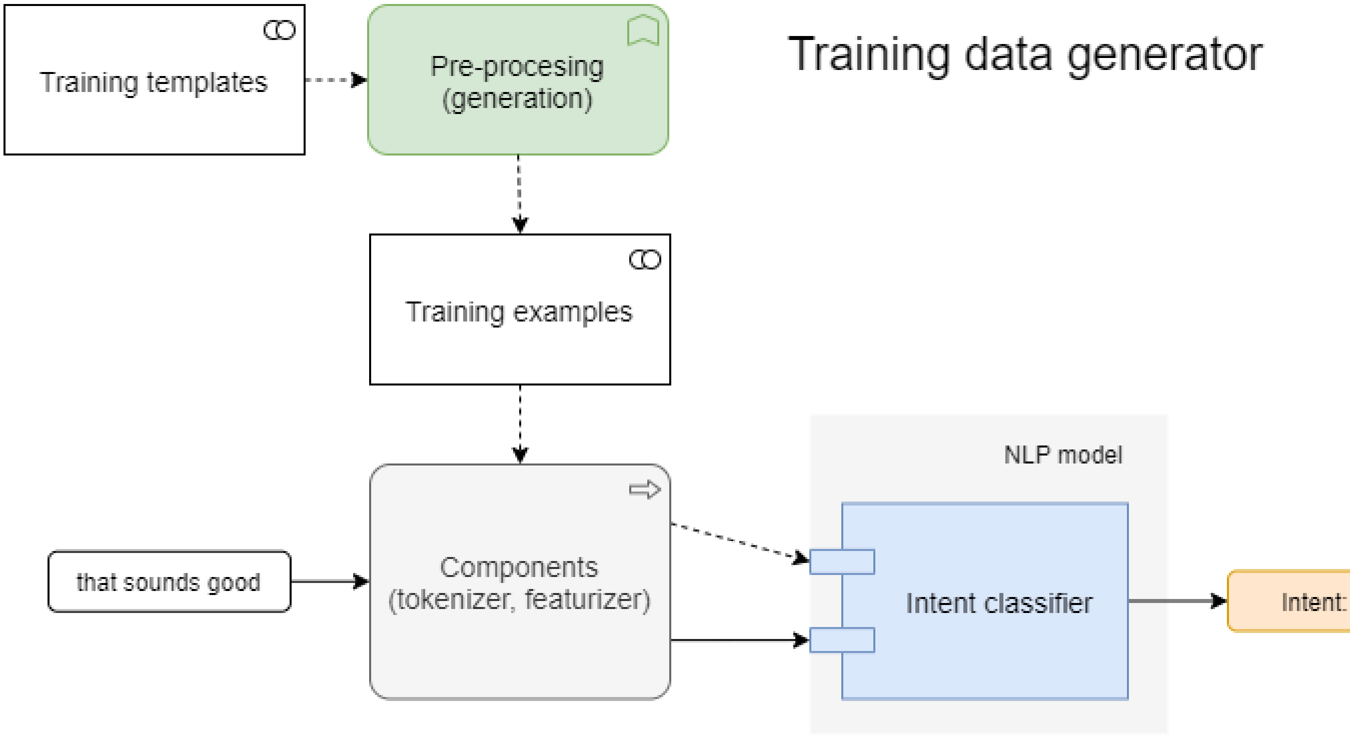
```
--num_train_epochs=3.0 \
```

```
--output_dir=/tmp/mrpc_output/
```

When ready, the model from resulting folder can be used in your pipeline and it should have higher F1-score than original one.

Here is another tuning example f

2. Training data generator



To quickly get more training data for NLU task, you can use a training data generator. It is a program which takes templates and generates a lot of examples for model training. Well-known generators are Chatito (in Node.js) and Chatette (in Python). A Python programming language preprocessor is a common tool.

Instead of writing text preprocessing examples directly, you write one or several template files in a specific format, then run the generator which parses templates and outputs ready-to-use examples. This is one of the essential preprocessing steps in text mining. To train a powerful model, you need to use the generator not just for intent detection, but more for named entity recognition.

Let's generate samples for our easy intent with the NLP text preprocessing Python Chatette. Write a template file "affirm.chatette" with the following content:

~~~
```
~~~

```
~[yes]
```

```
yes
```

```
indeed
```

```
of course
```

```
that sounds good
```

```
correct
```

```
got it
```

```
alright
```

```
sure
```

```
ok
```

```
~[thanks]
    thanks
    thank you
```

```
%[greet](training:100)
    ~[yes][,?] ~[thanks?]
```

~~~~
```
~~~~

Next, generate an nlu file.

```
!python -m chatette -f -s YZkEus -a rasa-md -o results data-generate/
affirm.chatette
```

```
!cp results/train/output.md data-generate/nlu/affirm.md
```

```
Executing Chatette with seed 'YZkEus'.
```

```
[DBG] Parsing file: /content/data-generate/affirm.chatette
[DBG] Generating training examples...
[DBG] Generating testing examples...
[DBG] Generation over
```

Next, we train and evaluate our simple model with just generated data.

```
!rasa train -c config-simple.yml --data data-generate --fixed-model-name generated --quiet
```

```
Core stories/configuration did not change. No need to retrain Core model.
Training NLU model...
```

```
2020-06-22 20:49:26.699066: E tensorflow/stream_executor/cuda/cuda_driver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
```

```
/usr/local/lib/python3.6/dist-packages/rasa/utils/common.py:363: UserWarning: You specified 'DIET' to train entities, but no entities are present in the training data. Skip training of entities.
```

```
NLU model training completed.
```

```
Your Rasa model is trained and saved at '/content/models/generated.tar.gz'.
```

```
!rasa test nlu -c config-simple.yml -u test_data.md -m models/generated.tar.gz --out results/generated --quiet
```

```
report = pd.read_json("results/generated/intent_report.json", orient="values")
```

```
generated_f1 = report["weighted avg"]["f1-score"]

data = [["simple", simple_f1], ["generated", generated_f1]]

pd.DataFrame(data, columns=["Model", "F1-sore"])
```

```
2020-06-22 21:06:57.539434: E tensorflow/stream_executor/cuda/cuda_dr
iver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-cap
able device is detected

100% 14/14 [00:00<00:00, 110.23it/s]
```

```
     Model   F1-sore

0    simple  0.614286

1    generated      1.000000
```

Expected F1-score = 1.000000

As we see in this case, we improve the accuracy by generating examples, without modifying the model config itself. Now, imagine that having 3 aliases with 10 examples each, like

Hire talented developers from LATAM, Canada, and Europe

3. Crowdsource in AI chatbot development

To get more training data of high quality, you can outsource jobs to distributed workers, using Amazon Mechanical Turk, Microworkers, Clickworker platforms.

MTurk is one of the text preprocessing tools to take simple and repetitive tasks that need to be handled manually. In a short time and at a reduced cost, you can get human-written training data for your initial model to augment training data collection and ultimately accelerate Python-based chatbot development using data preprocessing for text classification.

Key takeways

In this article, we have considered three ways to improve language model intent detection accuracy.

The first one, the Pre-trained model, is the most computation-heavy yet allows us to have fewer examples to perform training, without accuracy loss.

The second one, Training data generator, is the simplest method that still requires to write some templates in a specific language. It reduces the amount of text input and grows the number of available samples to train the model.

The third one, Crowdsource, gives the most accurate output written by humans, but typically involves an extra cost to put in place.

challenge involved loading and pre-processing in chatbots

1. **Data Quality and Quantity**:

   - **Insufficient Data**: Inadequate training data can lead to a chatbot that performs poorly, struggles with understanding user inputs, or provides inaccurate responses.
   - **Noisy Data**: Data may contain errors, inconsistencies, or irrelevant information. Cleaning and filtering this data can be time-consuming.
   - **Bias in Data**: If the training data is biased towards certain demographics or viewpoints, the chatbot may exhibit biased behavior.
2. **Data Annotation**:
   - **Manual Annotation**: Depending on the task, data may need to be annotated, which can be a time-consuming process. For example, in intent recognition, each user input needs to be labeled with the corresponding intent.
3. **Text Pre-processing**:
   - **Tokenization**: Breaking text into words or tokens can be challenging for languages with no clear word boundaries or complex punctuation.
   - **Stopword Removal and Stemming/Lemmatization**: Deciding which words to remove and how to reduce words to their base form can affect the model's performance.
4. **Handling Special Characters and Symbols**:
   - **Emojis, URLs, and Special Characters**: These can be challenging to handle, as they may not have clear semantic meanings or could cause issues during tokenization.
5. **Dealing with Multilingual Data**:
   - If the chatbot is intended to support multiple languages, handling different character sets, grammar rules, and linguistic nuances can be complex.
6. **Domain-Specific Language**:
   - If the chatbot is designed for a specific domain (e.g., medical, legal), understanding and processing the specialized language of that domain can be challenging.
7. **Entity Recognition**:
   - Identifying and extracting entities (e.g., names, dates, locations) from user inputs can be complex, especially when dealing with ambiguous references or non-standard terminology.
8. **Intent Recognition**:
   - Training a model to accurately recognize user intents can be challenging, particularly when intents are closely related or when there is a wide range of possible user inputs.
9. **Context Management**:

- Maintaining context during a conversation, especially in long or complex interactions, requires careful design to ensure the chatbot understands and responds appropriately.

10. **Dynamic Responses**:

- If the chatbot needs to generate dynamic responses (e.g., providing real-time information), integration with external systems and services may be required.

11. **Security and Privacy**:

- Ensuring that sensitive information is handled securely and that user privacy is maintained is crucial, especially when dealing with personal data.

12. **Scalability and Performance**:

- Efficiently handling a large volume of user inputs and generating responses in real-time, especially in production environments, can be a significant technical challenge.

13. **Testing and Validation**:

- Ensuring that the pre-processed data is of high quality and representative of actual user inputs is essential for building a reliable chatbot.

.

How to challenge involving loading and pre-processing in chatbots

1. **Data Collection**:
    - Gather a diverse and representative dataset for training your chatbot. This dataset should include a wide range of topics and language variations.
2. **Data Cleaning**:
    - Inspect and clean the dataset to remove any noise, inconsistencies, or irrelevant information. This can involve tasks like removing duplicates, correcting spelling mistakes, and handling special characters.
3. **Data Formatting**:
    - Format the data in a way that is suitable for your chatbot's training process. For example, if you're using a neural network-based model, you'll typically want to represent the data in a format like sequences of tokens.
4. **Tokenization**:
    - Break the text into smaller units, usually words or subwords. This is important for models like neural networks that operate on fixed-size input.
5. **Vocabulary Selection**:

- Decide on the size of your vocabulary. A larger vocabulary allows the model to represent more words but may require more computational resources.

6. **Embedding and Vectorization**:
    - Convert the tokenized text into numerical vectors. This process involves assigning a unique vector to each word or subword in your vocabulary. Techniques like Word2Vec, GloVe, or embeddings layers in neural networks can be used for this purpose.
7. **Handling Out-of-Vocabulary Words**:
    - Implement a mechanism to handle words that are not in your vocabulary. This can be done through techniques like subword tokenization, which can break down unknown words into smaller units.
8. **Padding and Truncation**:
    - Ensure that all input sequences have the same length by either padding shorter sequences or truncating longer ones.
9. **Data Splitting**:
    - Divide your dataset into training, validation, and testing sets. The training set is used to train the model, the validation set is used to fine-tune hyperparameters, and the testing set is used to evaluate the model's performance.
10. **Batching**:
    - Organize the data into batches for efficient training. This allows the model to process multiple examples simultaneously, which can significantly speed up the training process.
11. **Loading and Handling Large Datasets**:
    - If you're dealing with a large dataset that doesn't fit into memory, consider using techniques like data generators or streaming to load and process data in chunks.
12. **Pre-processing for Specific Use Cases**:
    - Depending on your use case, you may need additional pre-processing steps. For example, if your chatbot needs to understand multiple languages, you might need to implement a language detection and translation step.