COURSE NOTES: BUILDING CHATBOTS IN PYTHON

Building Chatbots in Python

By

Hana Zrafi

Introduction

In the evolving landscape of artificial intelligence and machine learning, chatbots have become powerful tools that enhance user experiences by providing intelligent, interactive, and automated assistance. They serve as virtual assistants that can understand and respond to human language, offering a wide range of applications from customer service to personal management.

This report explores several fundamental concepts and technologies that are at the core of modern chatbot development. These include techniques for word vectorization and cosine similarity, which enable chatbots to understand and compare textual data in meaningful ways. We will also delve into entity extraction, a key component for identifying important data within text, allowing chatbots to perform tasks such as booking, recommending, or providing personalized responses.

One of the most critical aspects of chatbot functionality is Natural Language Understanding (NLU), which is achieved using advanced libraries such as Rasa NLU. The Rasa framework facilitates the development of powerful chatbots by providing a flexible pipeline for processing and understanding natural language. This report will examine Rasa pipelines in detail and how they can be customized for specific domains to ensure smooth and accurate conversation flow.

Moreover, we will explore incremental slot filling and negation, techniques that help chatbots improve user interactions by gradually collecting necessary information and handling user preferences and contradictions. The ability to maintain context and manage different states within a conversation is essential for a chatbot's effectiveness, and we will discuss how a state machine can be implemented to manage these transitions.

Finally, we will look at how chatbots can be integrated with databases to enable them to explore and retrieve data based on user inputs, thus enabling more dynamic and responsive interactions. By the end of this report, readers will

have a comprehensive understanding of the technologies and methodologies that power modern conversational agents and their applications.

Word_vectors

Entity Extraction

Rasa NLU library : Natural language understanding

Rasa pipelines

Chatbot: Virtual assistant

Exploring a DB with natural language

Incremental slot filling and negation

State Machine in Chatbot Development

Word_vectors

Describe a text using vectors : Assigning a vector to each word that desc ibes its meaning allows us to represent text in numerical form. There are high-quality word vector models available, such as the GloVe and Word2Vec algorithms. To achieve this, we use the spaCy library, which provides pre-trained vectors for multiple languages.

```
import spacy
nlp = spacy.load("en_core_web_sm") #
Default language english : spacy.load("en")
n'est plus valide depuis spacy 3.0

doc=nlp('Hello! I am Hana Zrafi, a data
scientist.')
for token in doc:
    print("{}:
{}".format(token,token.vector[:3]))
```

```
Hello:[-0.9334216  -0.01833366 -0.10102444]
!:[ 0.28879818 -0.46308324  0.8908046 ]
I:[-1.3057288   0.04350807 -0.6210372 ]
am:[-0.6158351 -0.4440972 -2.153456 ]
Hana:[ 0.03805721 -0.9171484   0.5289741 ]
Zrafi:[-0.8788994 -1.0113711 -1.0913285]
,:[ 0.41136155  0.4840718  -0.49762443]
a:[1.4263003  0.7538499  0.20291325]
data:[0.42284307 1.0774597  0.27537802]
scientist:[-0.43133447 -0.65270334
-0.8020495 ]
.:[-0.51895016 -0.37181318  0.48773926]
```

Direction of vectors matters: Distance between words = angle between the vectors

cosine similarity

```
doc1=nlp("cat")
doc2=nlp("dog")
doc3=nlp("can")
```

```
doc1.similarity(doc2)
```

```
0.6847176149951816
```

```
doc1.similarity(doc3)
```

```
0.08477805530959502
```

"cat" and "can" are spelled similarly but have low similarity. But "cat" and "dog" are much more similar. That's because the similarity mesures how closest the meaning of the words is rather then the spelling.

Now we can integrate machine learning algorithms to classify texts to recognise their inents.

There are various classification algorithms available, each with its advantages:

The k-Nearest Neighbors (k-NN) classifier is one of the simplest and most intuitive methods. It assigns a class to a new text based on the majority label of its closest neighbors in the feature space.

Support Vector Machine (SVM / SVC), on the other hand, is a more sophisticated approach that finds an optimal hyperplane to separate different classes. It is particularly effective for text classification tasks where data is high-dimensional.

By leveraging the scikit-learn library, we can easily implement these models, train them on labeled datasets, and use them for real-world applications like spam detection, sentiment analysis, and chatbot intent recognition.

#Import SVC from sklearn.svm import SVC

#Create a support vector classifier clf = SVC()

#Fit the classifier using the training data clf.fit(X_train, y_train) #An array X containing vectors each of the sentences in the ATIS dataset has been created.

#Predict the labels of the test set y_pred = clf.predict(X_test)

#Count the number of correct predictions n_correct = 0 for i in range(len(y_test)): if y_pred[i] == y_test[i]: n_correct += 1

print("Predicted {0} correctly out of {1} test examples".format(n_correct, len(y_test))) Predicted 162 correctly out of 201 test examples

Entity Extraction

To identify generic entities such as locations, dates, organizations, and more, we can leverage spaCy's built-in Named Entity Recognition (NER).

NER is a powerful tool that automatically detects and categorizes named entities within a text. With pre-trained models, spaCy can recognize a wide range of entity types, including persons, geopolitical entities, numerical values, and temporal expressions.

This capability is essential for tasks such as information extraction, automated tagging, and knowledge graph construction, making it a key component in many natural language processing (NLP) applications.

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("my friend Hana is student at
ESSAI since 2022")
for ent in doc.ents:
    print(ent.text, ent.label_)
```

```
Hana PERSON
ESSAI ORG
2022 DATE
```

```
# Create the document
doc = nlp("let's see that jacket in red and
some blue jeans")



# Iterate over parents in parse tree until
an item entity is found
def find_parent_item(word):
    # Iterate over the word's ancestors
    for parent in word.ancestors:
        # Check for an "item" entity
        if entity_type(parent) == "item":
            return parent.text
    return None

# For all color entities, find their parent
item
def assign_colors(doc):
    # Iterate over the document
    for word in doc:
        # Check for "color" entities
        if entity_type(word) == "color":
            # Find the parent
            item = find_parent_item(word)
            print("item: {0} has color :
{1}".format(item, word))

# Assign the colors
assign_colors(doc)
```

Hidden output

```
import spacy

# Charger le modèle NLP anglais
nlp = spacy.load("en_core_web_sm")

# Phrase exemple
text = "The apple is red and the sky is
blue."

# Analyser le texte avec spaCy
doc = nlp(text)

# Fonction pour détecter si un mot est une
couleur
def entity_type(word):
    # Pour ce modèle, on suppose que "red"
et "blue" sont reconnus comme adjectifs
    colors = {"red", "blue", "green",
"yellow", "black", "white"}  # Liste des
couleurs possibles
    if word.text.lower() in colors:
        return "color"
    return None

# Fonction pour retrouver l'objet lié à la
couleur (parent dans la phrase)
def find_parent_item(word):
    for child in word.head.children:
        if child.dep_ in ("nsubj", "dobj",
"attr"):  # Sujet, objet direct ou attribut
            return child.text
    return "Unknown object"

# Fonction principale pour assigner les
couleurs aux objets
def assign_colors(doc):
    for word in doc:
        if entity_type(word) == "color":
            item = find_parent_item(word)
            print("item: {0} has color :
{1}".format(item, word))

# Exécuter l'analyse
assign_colors(doc)
```

```
item: apple has color : red
item: sky has color : blue
```

Rasa NLU library : Natural language understanding

Rasa NLU provides a high-level API for intent recognition and entity extraction, making it a powerful tool for building conversational AI systems.

In Python, Rasa is used through an interpreter object, which contains the trained model for intent classification and entity extraction. This allows developers to process user inputs and extract structured information efficiently.

Steps to Use Rasa in Python:

1. Install Rasa → pip install rasa
2. Initialize a Project → rasa init --no-prompt
3. Define Intents → Add them in data/nlu.yml
4. Train the Model → rasa train
5. Test in CLI → rasa shell nlu
6. Use in Python → Interpreter.load(): Once the model is trained, this loads the interpreter object, which can process text inputs and return structured intent and entity predictions.

By following these steps, you can build an intent recognition system capable of understanding user inputs and extracting key information in a chatbot or AI assistant.

```
from rasa.nlu.model import Interpreter

# Charger un modèle entraîné (remplace
'model_path' par le chemin correct)
interpreter =
Interpreter.load("model_path")

# Exemple de message
message = "I want to book a flight to
London"
result = interpreter.parse(message)

print(result)
```

Hidden output

Rasa pipelines

A Rasa pipeline is a list of components that are used to process text.

spacy_sklearn_pipeline=

"nlp_spacy": Initializes the English language model using SpaCy.

"ner_crf": Uses an entity extraction tool.

"ner_synonyms": Maps entities with the same meaning to the same key.

"intent_featurizer_spacy": Creates vector representations of sentences.

"intent_classifier_sklearn": A vector classifier using scikit-learn.

"intent_featurized_ngrams": Examines all the words in the training data for which there are no word vectors (including spelling mistakes).

Once the model is trained and used with this pipeline, these steps are automatically performed.

To build a custom entity recognition tool for your domain, the recommended component is ner_crf (Conditional Random Fields), which uses random fields to identify entities.

CRFs are a machine learning model that works well for entity recognition when you have a small training dataset.

#Import necessary modules from rasa_nlu.converters import load_data from rasa_nlu.config import RasaNLUConfig from rasa_nlu.model import Trainer

#Create args dictionary args = {"pipeline":"spacy_sklearn"}

#Create a configuration and trainer config=RasaNLUConfig(cmdline_args={"pipeline": "spacy_sklearn"}) trainer = Trainer(config)

#Load the training data training_data = load_data("./training_data.json")

#Create an interpreter by training the model interpreter = trainer.train(training_data)

#Test the interpreter print(interpreter.parse("I'm looking for a Mexican restaurant in the North of town"))

Fitting 2 folds for each of 6 candidates, totalling 12 fits {'intent': {'name': 'restaurant_search', 'confidence': 0.6627604390878398}, 'entities': [{'start': 18, 'end': 25, 'value': 'mexican', 'entity': 'cuisine', 'extractor': 'ner_crf'}, {'start': 44, 'end': 49, 'value': 'north', 'entity': 'location', 'extractor': 'ner_crf'}], 'intent_ranking': [{'name': 'restaurant_search', 'confidence': 0.6627604390878398}, {'name': 'goodbye', 'confidence': 0.14633725788681204}, {'name': 'affirm', 'confidence': 0.09756426473688806}, {'name': 'greet', 'confidence': 0.09333803828846025}], 'text': "I'm looking for a Mexican restaurant in the North of town"}

#Import necessary modules from rasa_nlu.config import RasaNLUConfig from rasa_nlu.model import Trainer

pipeline = ["nlp_spacy", "tokenizer_spacy", "ner_crf"]

#Create a config that uses this pipeline config = RasaNLUConfig(cmdline_args={"pipeline": pipeline})

#Create a trainer that uses this config trainer = Trainer(config)

#Create an interpreter by training the model interpreter = trainer.train(training_data)

#Parse some messages
print(interpreter.parse("show me Chinese food in the centre of town")) print(interpreter.parse("I want an Indian restaurant in the west"))
print(interpreter.parse("are there any good pizza places in the center?"))

{'intent': {'name': '', 'confidence': 0.0}, 'entities': [{'start': 28, 'end': 34, 'value': 'centre', 'entity': 'location', 'extractor': 'ner_crf'}], 'text': 'show me Chinese food in the centre of town'} {'intent': {'name': '', 'confidence': 0.0}, 'entities': [{'start': 10, 'end': 16, 'value': 'indian', 'entity': 'cuisine', 'extractor': 'ner_crf'}, {'start': 35, 'end': 39, 'value': 'west', 'entity': 'location', 'extractor': 'ner_crf'}], 'text': 'I want an Indian restaurant in the west'} {'intent': {'name': '', 'confidence': 0.0}, 'entities': [{'start': 39, 'end': 45, 'value': 'center', 'entity': 'location', 'extractor': 'ner_crf'}], 'text': 'are there any good pizza places in the center?'}

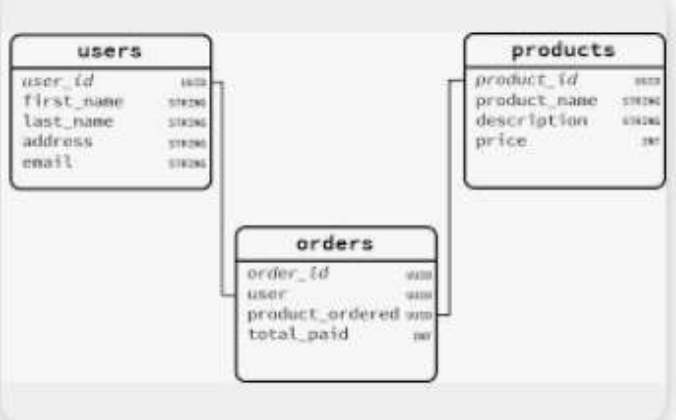
<script.py> output: {'intent': {'name': '', 'confidence': 0.0}, 'entities': [{'start': 28, 'end': 34, 'value': 'centre', 'entity': 'location', 'extractor': 'ner_crf'}], 'text': 'show me Chinese food in the centre of town'} {'intent': {'name': '', 'confidence': 0.0}, 'entities': [{'start': 10, 'end': 16, 'value': 'indian', 'entity': 'cuisine', 'extractor': 'ner_crf'}, {'start': 35, 'end': 39, 'value': 'west', 'entity': 'location', 'extractor': 'ner_crf'}], 'text': 'I want an Indian restaurant in the west'} {'intent': {'name': '', 'confidence': 0.0}, 'entities': [{'start': 39, 'end': 45, 'value': 'center', 'entity': 'location', 'extractor': 'ner_crf'}], 'text': 'are there any good pizza places in the center?'} In [1]:

Chatbot: Virtual assistant

A chatbot acts as a virtual assistant. Common use cases for chatbots include scheduling meetings, booking flights, or searching for restaurants. All of these tasks require information about the outside world, and to access it, we need to interact with databases, APIs, and other external systems.

By integrating these data sources, a chatbot can provide dynamic, real-time responses based on the most current information available. Whether it's querying a database for flight details or using an API to find restaurant reviews, chatbots can facilitate seamless interactions by tapping into these resources.

We will develop chatbots that can interact with an SQL database.



In SQL, data is stored in rows of a table.

Python has a module called sqlite3, which allows us to interact with SQLite database software, pre-installed on most operating systems.

import sqlite3

conn=sqlite.connect('base.db')

c=conn.cursor() #créer un curseur

c.execute("SELECT * FROM hotels WHERE area='south' and pricerange='hi'")

c.fetchall()

```
--> [('Grant Hotel','hi','south',5)]  the output is a list of tuple
```

Copy

This code demonstrates a basic interaction with an SQLite database using Python, which is essential for chatbots that need to retrieve data from a database to respond to user queries.

In [4]: cursor.execute("SELECT name from hotels where price = 'expensive' AND area = 'center'") Out[4]: <sqlite3.Cursor at 0×7f66693a1dc0> In [5]: cursor.fetchall() Out[5]: [] In [6]: cursor.execute("SELECT name from hotels where price = 'mid' AND area = 'north'") Out[6]: <sqlite3.Cursor at 0×7f66693a1dc0> In [7]: cursor.fetchall() Out[7]: [('Hotel California',)]

#Import sqlite3 import sqlite3

#Open connection to DB conn = sqlite3.connect('hotels.db')

#Create a cursor c = conn.cursor()

#Define area and price area, price = "south", "hi" t = (area, price)

#Execute the query c.execute('SELECT * FROM hotels WHERE area=? AND price=?', t)

#Print the results print(c.fetchall())

[('Grand Hotel', 'hi', 'south', 5)]

Exploring a DB with natural language

#Define find_hotels() def find_hotels(params): **#Create the base query** query = 'SELECT * FROM hotels' **#Add filter clauses for each of the parameters** if len(params) > 0: filters = ["{}=?".format(k) for k in params] query += " WHERE " + " and ".join(filters) **#Create the tuple of values** t = tuple(params.values())

#Open connection to DB conn = sqlite3.connect("hotels.db") **#Create a cursor** c = conn.cursor() **#Execute the query** c.execute(query,t) **#Return the results** return c.fetchall()

#Define respond() def respond(message): **#Extract the entities** entities = interpreter.parse(message)["entities"] #Initialize an empty params dictionary params = {} **#Fill the dictionary with entities** for ent in entities: params[ent["entity"]] = str(ent["value"])

#Find hotels that match the dictionary results = find_hotels(params) **#Get the names of the hotels and index of the response** names = [r[0] for r in results] n = min(len(results),3) **#Select the nth element of the responses array** return responses[n].format(*names)

#Test the respond() function print(respond("I want an expensive hotel in the south of town"))

```
-->Grand Hotel is a great hotel!
```
Copy

Incremental slot filling and negation

To improve user experience, we add memory to our bot (in the form of key-value pairs), allowing our users to gradually refine their searches. The process of collecting user preferences during a conversation is called slot filling.

Slot filling is a method used to gather necessary pieces of information (or slots) that the bot requires to complete a task. For example, if a user wants to book a flight, the bot may ask for the destination, date, and time. These are the slots that need to be filled. The process is incremental, meaning that the bot gradually collects these details over the course of the conversation, rather than asking for all information at once.