# Creating a Chatbot: From Problem Statement to Innovative Development

Welcome to our presentation on creating a chatbot! Explore the fascinating journey of developing a chatbot, from problem identification to innovative techniques.

# Problem Statement

## 1 Identify the Need

Discover a real-world problem that a chatbot can solve, ensuring it adds value to users' lives or businesses.

## 2 Narrow Down the Focus

Define the specific problem and target audience to develop a solution that meets their unique requirements.

## 3 Create a Solution Strategy

Outline the goals, functionalities, and desired outcomes of the chatbot to address the identified problem effectively.

# Design Thinking Process

1

## Empathize

Understand user needs and pain points through research, surveys, and interviews.

2

## Define

Create a clear problem statement and user persona based on the collected data.

3

## Ideate

Generate creative solutions and brainstorm ideas to tackle the user's problem effectively.

4

## Prototype

Build a basic version of the chatbot to test and gather feedback for further improvement.

5

## Test

Evaluate the chatbot's performance, user experience, and functionality through user testing and feedback loops.

6

## Iterate

Make necessary refinements and enhancements based on user feedback and insights.

# Phases of Development

## Planning

Map out the project scope, timeline, and allocate the necessary resources for each development phase.

## Design

Create an intuitive and user-friendly chatbot interface, incorporating brand elements and ensuring seamless navigation.

## Development

Write clean, scalable code using relevant libraries and frameworks to bring the chatbot to life.

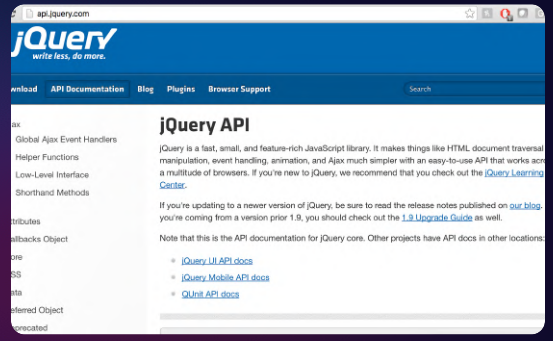
## Testing

Thoroughly test the chatbot's functionalities, interactions, and integrations with the web application.

# Libraries Used and NLP Integration

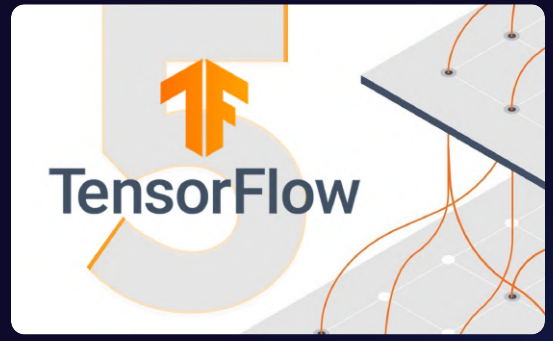

## Python

Utilize Python libraries such as NLTK and spaCy for natural language processing tasks.




## jQuery

Leverage jQuery to enhance the chatbot's interactivity and DOM manipulation.




## TensorFlow

Employ TensorFlow for developing and training advanced machine learning models in the chatbot.

# Chatbot Interaction with Users and Web Application

## Seamless Conversations

Engage users with a natural language interface capable of understanding and contextualizing their queries.

## Web Application Integration

Ensure seamless integration of the chatbot with the web application, enabling users to access its functionalities effortlessly.

# Innovative Techniques and Approaches

## 1 Data-Driven Personalization

Adopt a data-driven approach to personalize user interactions and provide tailored recommendations.

## 2 Emotion Analysis

Incorporate sentiment analysis techniques to detect and respond empathetically to users' emotions.

## 3 Multi-Platform Accessibility

Create a chatbot that can seamlessly interact across various platforms like web, mobile, and voice assistants.

# The Dataset

We will use the 'data.json' file as our dataset. It contains predefined patterns and responses.

Import and load the data file


{"intents": [

{"tag": "greeting",

"patterns": ["Hi there", "How are you", "Is anyone there?","Hey","Hola", "Hello", "Good day"],

"responses": ["Hello, thanks for asking", "Good to see you again", "Hi there, how can I help?"],

"context": [""]

},

{"tag": "goodbye",

"patterns": ["Bye", "See you later", "Goodbye", "Nice chatting to you, bye", "Till next time"],

"responses": ["See you!", "Have a nice day", "Bye! Come back again soon."], "context": [""]

},

{"tag": "thanks",

"patterns": ["Thanks", "Thank you", "That's helpful", "Awesome, thanks", "Thanks for helping me"],

"responses": ["Happy to help!", "Any time!", "My pleasure"], "context": [""]

},

{"tag": "noanswer",

"patterns": [],

"responses": ["Sorry, can't understand you", "Please give me more info", "Not sure I understand"],

"context": [""]

},

{"tag": "options",

"patterns": ["How you could help me?", "What you can do?", "What help you provide?", "How you can be helpful?", "What support is offered"],

"responses": ["I can guide you through Adverse drug reaction list, Blood pressure tracking, Hospitals and Pharmacies", "Offering support for Adverse drug reaction, Blood pressure, Hospitals and Pharmacies"],

"context": [""]

},

{"tag": "adverse_drug",

"patterns": ["How to check Adverse drug reaction?", "Open adverse drugs module", "Give me a list of drugs causing adverse behavior", "List all drugs suitable for patient with adverse reaction", "Which drugs dont have adverse reaction?" ],

"responses": ["Navigating to Adverse drug reaction module"],

"context": [""]

},

{"tag": "blood_pressure",

"patterns": ["Open blood pressure module", "Task related to blood pressure", "Blood pressure data entry", "I want to log blood pressure results", "Blood pressure data management" ],

"responses": ["Navigating to Blood Pressure module"],

"context": [""]

},

{"tag": "blood_pressure_search",

"patterns": ["I want to search for blood pressure result history", "Blood pressure for patient", "Load patient blood pressure result", "Show blood pressure results for patient", "Find blood pressure results by ID" ],

"responses": ["Please provide Patient ID", "Patient ID?"], "context": ["search_blood_pressure_by_patient_id"]

},

{"tag": "search_blood_pressure_by_patient_id", "patterns": [],

"responses": ["Loading Blood pressure result for Patient"],

"context": [""]

},

{"tag": "pharmacy_search",

"patterns": ["Find me a pharmacy", "Find pharmacy", "List of pharmacies nearby", "Locate pharmacy", "Search pharmacy" ],

"responses": ["Please provide pharmacy name"], "context": ["search_pharmacy_by_name"]

},

{"tag": "search_pharmacy_by_name", "patterns": [],

"responses": ["Loading pharmacy details"], "context": [""]

},

{"tag": "hospital_search",

"patterns": ["Lookup for hospital", "Searching for hospital to transfer patient", "I want to search hospital data", "Hospital lookup for patient", "Looking up hospital details" ],

"responses": ["Please provide hospital name or location"],

"context": ["search_hospital_by_params"]

},

{"tag": "search_hospital_by_params", "patterns": [],

"responses": ["Please provide hospital type"],

"context": ["search_hospital_by_type"]

},

{"tag": "search_hospital_by_type", "patterns": [],

"responses": ["Loading hospital details"],

"context": [""]

}

]

}

# Prerequisites

To work on this project, you should have a good understanding of Python, Keras, and Natural Language Processing (NLTK). You will also need to install some additional modules using the python-pip command.

BUILD THE MODEL:

import nltk from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer() import json import pickle

import numpy as np from keras.models import Sequential from keras.layers

import Dense, Activation, Dropout from keras.optimizers import SGD import random

words=[] ,classes = [], documents = [], ignore_words = ['?', '!'] data_file = open('data.json').read() intents = json.loads(data_file)

for intent in intents['intents']: for pattern in intent['patterns']:

lemmaztize and lower each word and remove duplicates words = [lemmatizer.lemmatize(w.lower())

for w in words if w not in ignore_words] words = sorted(list(set(words)))

sort classes

classes = sorted(list(set(classes)))

documents = combination between patterns and intents print (len(documents), "documents")

classes = intents

print (len(classes), "classes", classes) words = all words, vocabulary

print (len(words), "unique lemmatized words", words) pickle.dump(words,open('texts.pkl','wb')) pickle.dump(classes,open('labels.pkl','wb')) create our training data

training = []

create an empty array for our output output_empty = [0] * len(classes)

training set, bag of words for each sentence

for doc in documents: # initialize our bag of words bag = [] # list of tokenized words for the pattern pattern_words = doc[0] # lemmatize each word - create base word, in attempt to represent related words pattern_words = [lemmatizer.lemmatize(word.lower()) for word in pattern_words] # create our bag of words array with 1, if word match found in current pattern for w in words: bag.append(1) if w in pattern_words else bag.append(0)

shuffle our features and turn into np.array random.shuffle(training) training = np.array(training) create train and test lists. X - patterns, Y - intents train_x = list(training[:,0])

train_y = list(training[:,1]) print("Training data created")

Create model - 3 layers. First layer 128 neurons, second layer 64 neurons and 3rd output layer contains number of neurons

equal to number of intents to predict output intent with softmax model = Sequential()

model.add(Dense(128, input_shape=(len(train_x[0]),), activation='relu')) model.add(Dropout(0.5))

model.add(Dense(64, activation='relu')) model.add(Dropout(0.5)) model.add(Dense(len(train_y[0]), activation='softmax'))

Compile model. Stochastic gradient descent with Nesterov accelerated gradient gives good results for this model

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True) model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

#fitting and saving the model hist = model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_size=5, verbose=1) model.save('model.h5', hist)

print("model created")

# How l'o Make a Chatbol' in Python?

To build the chatbot, we will use the Flask framework. Let's take a look at the file structure and the files we will create:

**data.json** - Contains the predefined patterns and responses.

**training.py** - Builds the model and trains the chatbot.

**Texts.pkl** - Stores the vocabulary as a Python object using NLTK.

**Labels.pkl** - Contains the list of categories (labels). **model.h5** - The trained model with neuron weights.
**app.py** - Implements the web-based GUI for the chatbot.

# 5 Steps to Create a Chatbot in Flask

## Import and Load the Data File

**Preprocess the Data**

We need to preprocess the text data before designing the ANN model. Tokenizing is the first step, involving breaking the text into words. We use the nltk.word_tokenize() function to tokenize the patterns and create a list

of words.

## Build the Model

We build a deep neural network with 3 layers using the Keras sequential API. After training the model for 200 epochs, we achieve 100% accuracy and save the model as 'model.h5'.

In the 'training.py' file, we import the necessary packages and initialize the variables. We parse the JSON file into Python using the json package.

## Create Training and Testing Data

We create the training data by providing input patterns and their corresponding classes. Since computers understand numbers, we convert the text into numerical form.

# Fitting and Saving the Model

We fit the model using the `fit` function and save it as an h5 file using

`model.save`. The model is trained for 200 epochs with a batch size of 5.

# Predicting the Response with Flask

To predict the sentences and get a response from the user, we create a Flask web-based GUI. We need to create two folders named `static` and `templates`, and an `app.py` file.

# CSS Styling

We define the CSS styling for the chatbot interface in the `style.css` file.

:root {

--body-bg: linear-gradient( deg, #f f fa %, #c cfe %);

--msger-bg: #fff;

--border: px solid #ddd;

--left-msg-bg: #ececec;

--right-msg-bg: # ffb;

}

html {

box-sizing: border-box;

}

*,

*:before,

*:after { margin: ; padding: ;

box-sizing: inherit;

}

body { display: flex;

justify-content: center; align-items: center; height: vh;

background-image: var(--body-bg); font-family: Helvetica, sans-serif;

}

.msger { display: flex;

flex-flow: column wrap;

justify-content: space-between; width: %;

max-width: px; margin: px px; height: calc( % - px); border: var(--border); border-radius: px;

background: var(--msger-bg);

box-shadow: px px - px rgba( , , , . );

}

.msger-header {

/* display: flex; */ font-size: medium;

justify-content: space-between; padding: px;

text-align: center;

border-bottom: var(--border); background: #eee;

color: # ;

}

.msger-chat { flex: ;

overflow-y: auto; padding: px;

}

.msger-chat::-webkit-scrollbar { width: px;

}

.msger-chat::-webkit-scrollbar-track { background: #ddd;

}

.msger-chat::-webkit-scrollbar-thumb { background: #bdbdbd;

}

.msg { display: flex;

align-items: flex-end; margin-bottom: px;

}

.msg-img { width: px; height: px;

margin-right: px; background: #ddd; background-repeat: no-repeat; background-position: center; background-size: cover; border-radius: %;

}

.msg-bubble {

max-width: px; padding: px; border-radius: px;

background: var(--left-msg-bg);

}

.msg-info { display: flex;

justify-content: space-between; align-items: center;

margin-bottom: px;

}

.msg-info-name { margin-right: px; font-weight: bold;

}

.msg-info-time { font-size: . em;

}

.left-msg .msg-bubble { border-bottom-left-radius: ;

}

.right-msg {

flex-direction: row-reverse;

}

.right-msg .msg-bubble { background: var(--right-msg-bg); color: #fff;

border-bottom-right-radius: ;

}

.right-msg .msg-img { margin: px;

}

.msger-inputarea { display: flex; padding: px;

border-top: var(--border); background: #eee;

}

.msger-inputarea * { padding: px; border: none; border-radius: px; font-size: em;

}

.msger-input { flex: ;

background: #ddd;

}

.msger-send-btn { margin-left: px;

background: rgb( , , ); color: #fff;

font-weight: bold; cursor: pointer; transition: background . s;

}

.msger-send-btn:hover { background: rgb( , , );

}

.msger-chat {

background-color: #fcfcfe;

}

# HTML Template

We define the HTML template for the chatbot interface in the `index.html` file.

# Running the Flask App

We import the necessary packages and load the `texts.pkl` and `labels.pkl` pickle files that we created when we trained our model.

We implement functions to clean up the user's input, predict the class of the input, and retrieve a random response from the list of responses.

We create a Flask app and define routes for the home page and getting the bot's response. To run the app, we use `app.run()`.

import nltk nltk.download('popular')

from nltk.stem import WordNetLemmatizer lemmatizer = WordNetLemmatizer() import pickle import numpy as np

from keras.models import load_model model = load_model('model.h5')

import json import random intents = json.loads(open('data.json').read()) words = pickle.load(open('texts.pkl','rb'))

classes = pickle.load(open('labels.pkl','rb')) def clean_up_sentence(sentence):

# tokenize the pattern - split words into array sentence_words = nltk.word_tokenize(sentence)

# stem each word - create short form for word sentence_words = [lemmatizer.lemmatize(word.lower()) for word in sentence_words] return sentence_words

# return bag of words array: 0 or 1 for each word in the bag that exists in the sentence

def bow(sentence, words, show_details=True): # tokenize the pattern sentence_words = clean_up_sentence(sentence) # bag of words - matrix of N words, vocabulary matrix bag = [0]*len(words)

for s in sentence_words: for i,w in enumerate(words): if w == s: # assign 1 if current word is in the vocabulary position bag[i] = 1 if show_details: print ("found in bag: %s" % w) return(np.array(bag))

def predict_class(sentence, model): # filter out predictions below a threshold p = bow(sentence, words,show_details=False) res = model.predict(np.array([p]))[0] ERROR_THRESHOLD = 0.25 results = [[i,r] for i,r in enumerate(res) if r>ERROR_THRESHOLD] # sort by strength of probability results.sort(key=lambda x: x[1], reverse=True) return_list = [] for r in results: return_list.append({"intent": classes[r[0]], "probability": str(r[1])}) return return_list

def getResponse(ints, intents_json): tag = ints[0]['intent'] list_of_intents = intents_json['intents'] for i in list_of_intents: if(i['tag']== tag): result = random.choice(i['responses']) break return result

def chatbot_response(msg): ints = predict_class(msg, model) res = getResponse(ints, intents) return res

from flask import Flask, render_template, request app = Flask(**name**) app.static_folder = 'static'

@app.route("/") def home(): return render_template("index.html")

@app.route("/get") def get_bot_response(): userText = request.args.get('msg') return chatbot_response(userText)

if **name** == "**main**": app.run()

# Conclusion

That's it! Now we have a chatbot interface that can predict responses based on user input.