**Models To generate Solutions from Queries in Jira Ticket**

**Approach 1: Creating our own Model Using RNN:**

* M-1: LSTM model :

LSTM helps in generating sequence models for our data, and has long-term memory context which offers more accurate and flexible model in generating text solutions to our query.

We will first, preprocess the text by tokenizing, removing stopwords, and converting it into numerical representations using embeddings. Next, design an LSTM architecture that includes an embedding layer to handle the input sequences, followed by one or more LSTM layers to capture sequential dependencies. Train the model using labeled data where Jira issues are paired with their solutions, optimizing it to minimize loss during training. Evaluate the model to ensure it generalizes well, and utilize it for predicting the best-matching solution for new Jira issues based on their descriptions.

Example code:

model = Sequential()

model.add(LSTM(units=100, input\_shape=(embeddings\_proxy.shape[1], embeddings\_proxy.shape[2])))

model.add(Dense(units=len(label\_dict), activation='softmax'))

# Compile model

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Print model summary

print(model.summary())

# Train model

history = model.fit(embeddings\_proxy, encoded\_answers, epochs=10, batch\_size=32, validation\_split=0.2)

* **STRENGTHS:**
* **Sequence Modeling**: Particularly advantageous for tasks where understanding the context and flow of information (such as Jira issue descriptions) is crucial for accurate matching to solutions.
* **Long-term Dependencies**: This capability allows them to accurately handle Jira issue descriptions of varying lengths and complexities, ensuring comprehensive understanding and accurate matching.
* **High Contextual Understanding and Flexibilty**: Helps in detailed queries and adapts to different datasets and query sets.
* **Scalable and have high generalization**
* **Weakness:**
* **Complex in Training**: while powerful, these can be complex to train and fine-tune effectively This complexity can sometimes lead to longer development cycles and higher computational costs compared to simpler models.
* **Limited Contextual Understanding**: While LSTMs are adept at capturing long-term dependencies within sequences, they may struggle with capturing broader contextual information or domain-specific knowledge that could impact the matching of Jira issues to solutions.
* **Fixed Length Input Limitations**
* **Might give errors in unstructured data**
* **Resource Intensive**
* **Opportunity for our Task:**
* **Very good at handling sequence data for** our jira ticket issue description. This helps with dependencies and patterns within the text, which is crucial for accurately mapping issues to corresponding solutions.
* Can be customized according to domain specific question sets which will help accuracy and relevance.
* Has long term memory so helps so has the ability to capture long context in jira issues.
* **Limitations for our Task:**
* Complex in Training and utilizes huge amount of resouces as well. LSTM models require careful parameter tuning and sufficient training data to perform effectively.
* cases where Jira issue descriptions are brief or lack sufficient contextual information, LSTM models may struggle to extract meaningful patterns or dependencies, potentially affecting the accuracy of matching solutions.
* Needs handling of preprocessing data as it has Fixed Input limit
* M-2: GRU model :

Using a GRU (Gated Recurrent Unit) model for our project involves preprocessing Jira issue descriptions, designing a GRU-based neural network with embedding and GRU layers to capture dependencies, training it on labeled data, evaluating its performance, and deploying it to predict solutions for new issues based on descriptions. GRU's efficiency in handling sequential data makes it ideal for accurately matching Jira issues with appropriate solutions, optimizing support processes and user satisfaction.

It optimizes our approach from LSTM and then helps us save resources. It utilizes update and reset box which enables to have short term as well as long term memory and saving resources.

Example code:

# Define the GRU model

model\_gru = Sequential()

# Add GRU layer

model\_gru.add(GRU(units=100, input\_shape=(embeddings\_proxy.shape[1], embeddings\_proxy.shape[2])))

# Add Dense output layer

model\_gru.add(Dense(units=len(label\_dict), activation='softmax'))

# Compile the model

model\_gru.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Print model summary

print(model\_gru.summary())

# Train the model

history\_gru = model\_gru.fit(embeddings\_proxy, encoded\_answers, epochs=10, batch\_size=32, validation\_split=0.2)

* **STRENGTHS:**
* **Sequence Modeling**: Particularly advantageous for tasks where understanding the context and flow of information (such as Jira issue descriptions) is crucial for accurate matching to solutions.
* **Simpler Architecture**: GRUs have a simpler architecture compared to LSTMs, which makes them easier to understand, implement, and train. This simplicity often results in faster training times and reduced computational overhead.
* **Efficient Training and Less Time to Train**: Due to their fewer parameters and operations (e.g., fewer gates), GRUs can be quicker to train compared to LSTMs
* **Less Susceptible to Overfitting**: With fewer parameters, GRUs are less prone to overfitting compared to LSTMs, especially when the training dataset is not extensive. This attribute can lead to better generalization on validation and test data.
* **Weakness:**
* **Limited Long-Term Dependency**: GRUs may struggle with capturing long-term dependencies in sequential data as effectively as LSTMs. This limitation can impact their ability to model relationships that span across longer sequences or require retaining information over extended periods.
* **Less Control over Memory**: Unlike LSTMs, which have separate mechanisms to control the input, forget, and output gates, GRUs combine these functionalities into a single update gate and reset gate
* **Less Efficient on Complex Task**
* **Learning Contextual Information**: In some applications, especially those requiring comprehensive context understanding and precise sequence modeling (e.g., language translation, text generation), LSTMs' ability to explicitly manage long-term dependencies can be advantageous over GRUs.
* **Opportunity for our Task:**
* **Enhanced Efficiency in Training**: reducing training time and computational resources needed for processing Jira issue descriptions.
* **Effective Handling of Short-Term Dependencies**: beneficial for tasks involving quick contextual analysis of Jira issues to map them to solutions efficiently.
* easier handling of variable-length Jira issue descriptions without the need for extensive preprocessing, making them more flexible in real-world applications.
* **Limitations for our Task:**
* P**otential Limitations in Capturing Long-Term Dependencies**: Compared to LSTMs, GRUs may not perform as well in capturing long-term dependencies in Jira issue descriptions, which could affect their ability to understand complex relationships over extended sequences.
* **Complexity in Contextual Understanding**: In cases where Jira issue descriptions require deep contextual analysis spanning multiple interactions or stages, the simplified gating mechanism of GRUs may not be as effective as LSTMs in retaining and utilizing long-term context.
* T**rade-off in Resource Efficiency**: While GRUs are generally more efficient in terms of training speed and computational resources compared to LSTMs, they may sacrifice some long-term memory capabilities, which could impact accuracy in certain complex Jira issue resolution tasks.

**Approach 2: Using Pretrained Models:**

* M-1: Openai API:

We can use any openai api’s features such as function calling and parameters to extract parameters from input preprocessed data based on a custom function which can then be passed to a fetch function to generate a json solution for our query.

Example Code:

import openai

def extract\_parameters(preprocessed\_description):

# Custom function to extract parameters from the description

# This can include regex or NLP techniques to identify key components

parameters = {

"issue\_type": "login\_issue", # Example parameter

"error\_message": "system fails to authenticate" # Example parameter

}

return parameters

openai.api\_key = 'your-api-key-here'

def fetch\_solution\_from\_api(parameters):

prompt = f"""

Given the following parameters, provide a detailed solution to resolve the issue:

Issue Type: {parameters['issue\_type']}

Error Message: {parameters['error\_message']}

Solution:

"""

response = openai.Completion.create(

engine="text-davinci-003",

prompt=prompt,

max\_tokens=150,

n=1,

stop=None,

temperature=0.7

)

solution = response.choices[0].text.strip()

return {"solution": solution}

* **STRENGTHS:**
* **Advance Training**: As it is a GPT model it is trained on immense amount of data and can accurately identify complex data in Jira Tickets increasing accuracy and relevant solutions.
* **Flexible**: We can customize it as much we want using prompt training and parameters and even making changes to our functions to get proper parameters.
* **Scalable and efficient for large enterprises where we need accurate results.**
* OpenAI API is designed to be easily integrated into existing systems via simple API calls. This makes it straightforward to incorporate into your Jira workflow, reducing development time and effort. It saves resources that would be used in training and development.
* **Weakness:**
* **Dependent on a 3rd Party Service:** Relying on an external API means your solution is dependent on the availability and performance of the OpenAI service. Any downtime or latency issues with the API can affect your system's performance.
* **Data Security Concerns: Similar to above as we rely on a 3rd party agent we might face data security concerns.**
* **Limited Control over Model**
* **Opportunity for our Task:**
* **Will be more accurate for our tasks and as api is trained on large amount of dataset we get accurate results with prompts and don’t train our model again and again.**
* **Model can easily be fine trained and tuned .**
* **It can even find parameters from large contexts like**
* **Limitations for our Task:**
* Api usage needs to be constantly monitored to not overuse and exceed the limit.
* **Errors if not enough data is provided or small context: Can lead to hallucinations for our solutions.**
* Dependence on 3rd party API.
* M-2: LangChain API:

We can use LangChain’s features such as agents,dynamic chaining and customizable logic to preprocess and extract parameters from input data. By creating a custom function within LangChain, we can efficiently process Jira ticket descriptions, extract key information, and then pass these parameters to a chain of calls. This chain can generate a JSON solution for our query, leveraging LangChain's capabilities to handle complex workflows and produce accurate, context-aware solutions.

Example Code:

from langchain import LangChain

from langchain.chains import SimpleChain

import requests

def extract\_parameters(preprocessed\_description):

# Custom function to extract parameters from the description

# This can include regex or NLP techniques to identify key components

parameters = {

"issue\_type": "login\_issue", # Example parameter

"error\_message": "system fails to authenticate" # Example parameter

}

return parameters

return parameters

# Initialize LangChain

lang\_chain = LangChain()

# Custom function to call the API

def fetch\_solution\_from\_api(parameters):

url = "https://api.openai.com/v1/completions"

headers = {

"Authorization": f"Bearer YOUR\_API\_KEY",

"Content-Type": "application/json"

}

prompt = f"""

Given the following parameters, provide a detailed solution to resolve the issue:

Issue Type: {parameters['issue\_type']}

Error Message: {parameters['error\_message']}

Solution:

"""

payload = {

"model": "text-davinci-003",

"prompt": prompt,

"max\_tokens": 150,

"n": 1,

"stop": None,

"temperature": 0.7

}

response = requests.post(url, headers=headers, json=payload)

solution = response.json()["choices"][0]["text"].strip()

return {"solution": solution}

# Define the chain of steps

class PreprocessChain(SimpleChain):

def \_call(self, inputs):

description = inputs["description"]

preprocessed\_description = preprocess\_ticket(description)

return {"preprocessed\_description": preprocessed\_description}

class ExtractParametersChain(SimpleChain):

def \_call(self, inputs):

preprocessed\_description = inputs["preprocessed\_description"]

parameters = extract\_parameters(preprocessed\_description)

return {"parameters": parameters}

class FetchSolutionChain(SimpleChain):

def \_call(self, inputs):

parameters = inputs["parameters"]

solution = fetch\_solution\_from\_api(parameters)

return solution

# Combine chains

chain = PreprocessChain() | ExtractParametersChain() | FetchSolutionChain()

# Example input

ticket = {

"description": "……"

}

# Run the chain

result = chain({"description": ticket["description"]})

print(result)

* **STRENGTHS:**
* **Modular and Extensible**: easily integrate various preprocessing, parameter extraction, and solution generation steps We can easily tailor the pipeline for specific needs.
* LangChain's chaining capability allows for creating complex workflows that can handle intricate data dependencies and processing steps. This is particularly useful for dealing with complex Jira ticket descriptions and ensuring accurate solutions.
* **Scalable and efficient for large enterprises where we need accurate results.**
* Can be easily integrated with 3rd party api’s in one location.
* **Weakness:**
* **Complexity in Setup:** Setting up and managing the chain of processes can be complex and might require more initial development effort compared to using a single API like OpenAI.
* **Dependency Management:** LangChain's extensive use of various components and external services can lead to dependency management challenges and potential integration issues.
* **Resource Intensive:** While it offers flexibility and modularity, LangChain's approach can be resource-intensive, especially when dealing with large-scale data and complex workflows.
* **Opportunity for our Task:**
* **Highly customizable , we can customize the model to our will by integrating any 3rd party api easily into our model. Can be easily integrated with diverse tools.**
* **Model can easily be fine trained and tuned .**
* **It can even find parameters from large contexts**
* Robust parameter extraction capabilities can enhance the accuracy of identifying relevant details from Jira ticket descriptions, leading to more precise solutions.
* **Limitations for our Task:**
* Api usage needs to be constantly monitored to not overuse and exceed the limit.
* **Dependence on external services.**
* Can lead to slow outputs in chaining.
* M-3: Transfer Learning Using BERT:

Using BERT's pretrained models within our custom solution pipeline, we can effectively process Jira ticket descriptions, capture intricate details, and feed this information into subsequent stages. This approach ensures that our system accurately maps Jira issues to their corresponding solutions, benefiting from BERT's ability to handle nuanced text data and produce context-aware outputs.

Example Code:

import tensorflow as tf

from transformers import BertTokenizer, TFBertModel

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, GlobalAveragePooling1D

# Load BERT tokenizer and model

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

bert\_model = TFBertModel.from\_pretrained('bert-base-uncased')

# Function to preprocess text and get BERT embeddings

def get\_bert\_embeddings(texts):

inputs = tokenizer(texts, return\_tensors='tf', padding=True, truncation=True, max\_length=512)

outputs = bert\_model(inputs)

return outputs.last\_hidden\_state

# Example data

jira\_issues = [

"Users are reporting issues with the application's login process. When attempting to log in with their credentials, the system fails to authenticate and displays an error message."

]

# Get embeddings

embeddings = get\_bert\_embeddings(jira\_issues)

# Define a simple model using BERT embeddings

input\_layer = Input(shape=(512, 768)) # BERT base model output shape

x = GlobalAveragePooling1D()(input\_layer)

x = Dense(100, activation='relu')(x)

output\_layer = Dense(len(label\_dict), activation='softmax')(x)

model = Model(inputs=input\_layer, outputs=output\_layer)

# Compile model

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Print model summary

print(model.summary())

# Example placeholder data

embeddings\_proxy = tf.random.normal([len(jira\_issues), 512, 768])

encoded\_answers = tf.random.uniform([len(jira\_issues)], maxval=len(label\_dict), dtype=tf.int32)

# Train model

history = model.fit(embeddings\_proxy, encoded\_answers, epochs=10, batch\_size=32, validation\_split=0.2)

* **STRENGTHS:**
* **Deep Contextual Understanding:** BERT (Bidirectional Encoder Representations from Transformers) provides deep contextual understanding of text by considering the context from both directions (left-to-right and right-to-left), which is crucial for accurately capturing the meaning of Jira ticket descriptions. .
* **Scalable and efficient for large enterprises where we need accurate results.**
* **Less Resources needed that other models and free to use.**
* **Weakness:**
* **Complexity in Fine Tuning :** requires careful consideration of hyperparameters and training procedures, which can be complex and time-consuming
* **It is harder for 3rd party integration that other models.**
* **Fixed Input Size:** BERT has a fixed input size limit (typically 512 tokens), which may require truncating or segmenting longer Jira ticket descriptions, potentially leading to loss of critical information.
* **Opportunity for our Task:**
* accurate mapping of Jira ticket descriptions to their corresponding solutions, improving the relevance and precision of the suggested solutions.
* **Transfer Learning:** By leveraging BERT’s pretrained models, we can significantly reduce the amount of domain-specific training data required, making it feasible to implement effective solutions with limited labeled data.
* **has enhanced feature extraction.**
* **Limitations for our Task:**
* High Computation so takes a lot of resources.
* **Complex Fine Tuning.**
* Can lead to slow outputs in long contexts.
* M-4: Using other models like T5(text to text) or other api’s from HUGGING FACE like llama ,Alberta etc :

Using other models such as T5 (text-to-text) or APIs from Hugging Face like LLAMA, ALBERT, etc., while generally less reliable and potentially less accurate, can offer significant resource savings compared to more robust alternatives. However, their effectiveness may vary and typically requires experimentation and fine-tuning to achieve optimal results in specific use cases. Experimentation and iterative testing are often necessary to fine-tune these models and achieve satisfactory results in tasks such as predicting solutions from Jira tickets.