**The key NLP, LLM, and Python modules used in our application and how they are utilized:**

NLP (Natural Language Processing) Modules:

1. AWS Comprehend: AWS Comprehend is a natural language processing service provided by Amazon Web Services. In this code, it is used for two main purposes:
   * Detecting entities in the user query using the detect\_entities method. This helps in understanding the intent and extracting relevant information from the user's query.
   * Detecting the programming language of the generated code using the detect\_dominant\_language method. This allows the application to identify the language of the code snippet and provide appropriate file extensions and syntax highlighting.

LLM (Large Language Model) Modules:

1. Claude-3 Sonnet API: The code uses the Claude-3 Sonnet API, which is a large language model developed by Anthropic. It is accessed through the Bedrock Runtime, a service provided by Anthropic for interacting with their language models. The API is used for two main tasks:
   * Processing the user query and generating a response using the call\_claude3\_api function. The API takes the user query as input and returns a response that helps in understanding the intent and extracting relevant information.
   * Generating code based on the user's task description using the generate\_code function. The API takes the task description as input and generates code snippets in the identified programming language.

Python Modules:

1. Streamlit: Streamlit is a Python library used for building interactive web applications. In this code, Streamlit is used extensively to create the user interface and handle user interactions. It provides various components such as radio buttons, file uploader, text input, buttons, and containers to build the application layout.
2. Pandas: Pandas is a powerful data manipulation library in Python. It is used to load and process the data files uploaded by the user or selected from AWS S3. Pandas provides functions like read\_csv, read\_excel, read\_json, and read\_parquet to read data from different file formats. It also offers data manipulation capabilities such as grouping, filtering, and aggregating data.
3. Plotly Express: Plotly Express is a high-level Python library for creating interactive visualizations. In this code, it is used to generate various types of plots based on the user's query and the determined chart type. Plotly Express provides functions like px.bar, px.line, px.scatter, px.histogram, px.pie, etc., to create different types of charts and customize their appearance.
4. Boto3: Boto3 is the AWS SDK for Python. It is used to interact with various AWS services. In this code, Boto3 is used to initialize and access AWS services such as S3 (for listing and retrieving files), Comprehend (for entity detection and language detection), and Bedrock Runtime (for accessing the Claude-3 Sonnet API).
5. Difflib and Levenshtein: Difflib is a Python module that provides tools for comparing sequences, while Levenshtein is a library for calculating Levenshtein distance between strings. In this code, these modules are used for autocorrecting words and finding the closest match from a list of possibilities using the get\_close\_matches function from difflib and the distance function from Levenshtein.

These NLP, LLM, and Python modules work together to provide the core functionality of the application. The NLP modules (AWS Comprehend) help in understanding and processing user queries, the LLM module (Claude-3 Sonnet API) generates responses and code snippets, and the Python modules (Streamlit, Pandas, Plotly Express, Boto3, Difflib, Levenshtein) handle the user interface, data manipulation, visualization, and interaction with AWS services.

The integration of these modules enables the application to provide interactive data visualization and code generation capabilities based on user inputs and queries.

**Explaining the entire code step-by-step for you:**

python

Copy code

import streamlit as st

import pandas as pd

import plotly.express as px

import boto3

import json

from io import BytesIO

from difflib import get\_close\_matches

from Levenshtein import distance as levenshtein\_distance

from pygwalker.api.streamlit import StreamlitRenderer

import base64

import pyperclip

These lines import the necessary libraries and modules required for the application. The most important ones are:

* streamlit: The Streamlit library for building interactive web apps.
* pandas: For data manipulation and analysis.
* plotly.express: For creating interactive visualizations.
* boto3: The AWS SDK for Python, used for interacting with AWS services like S3 and Comprehend.
* json: For working with JSON data.
* io: For working with in-memory files.
* difflib: For finding the closest matching words.
* Levenshtein: For calculating the Levenshtein distance between words.
* base64: For encoding/decoding data in Base64 format.
* pyperclip: For copying text to the clipboard.

python

Copy code

*# Initialize AWS services*

if 's3' not in st.session\_state:

st.session\_state.s3 = boto3.client('s3')

if 'comprehend' not in st.session\_state:

st.session\_state.comprehend = boto3.client('comprehend')

if 'bedrock\_runtime' not in st.session\_state:

st.session\_state.bedrock\_runtime = boto3.client(service\_name='bedrock-runtime', region\_name='us-east-1')

data = pd.DataFrame()

These lines initialize the AWS services (S3, Comprehend, and Bedrock Runtime) using the boto3 library. The services are stored in the Streamlit session state, which preserves the state across multiple runs of the app. An empty pandas DataFrame is also initialized to store the loaded data.

python

Copy code

*# Set the Streamlit page to wide mode*

st.set\_page\_config(page\_title="Interactive Data Visualization", page\_icon=":chart\_with\_upwards\_trend:", layout="wide")

This line sets the configuration for the Streamlit app, including the page title, icon, and layout (wide mode).

python

Copy code

st.markdown(

"""

<style>

@import url('https://fonts.googleapis.com/css2?family=Montserrat:wght@400;600&display=swap');

body {

font-family: 'Montserrat', sans-serif;

background-color: #f5f5f5;

}

.stTextInput > div > div > input {

width: 100%;

}

footer {

text-align: center;

padding: 20px;

background-color: #f8f9fa;

font-size: 14px;

color: #6c757d;

position: fixed;

left: 0;

bottom: 0;

width: 100%;

}

.input-container {

position: fixed;

bottom: 20px;

left: 50%;

transform: translateX(-50%);

width: 50%;

display: flex;

align-items: center;

justify-content: space-between;

}

.input-container .stTextInput {

width: 80%;

}

.input-container button {

margin-left: 10px;

}

</style>

""",

unsafe\_allow\_html=True

)

This code block uses st.markdown to inject custom CSS styles into the Streamlit app. The CSS styles define the appearance and layout of various elements in the app, including:

* Importing the Montserrat font from Google Fonts.
* Setting the background color and font family for the app.
* Making the input field in stTextInput components span the full width.
* Styling a fixed footer at the bottom of the page.
* Positioning and styling a container for the user input field and submit button at the bottom center of the page.

The unsafe\_allow\_html=True parameter is required to allow Streamlit to render the HTML and CSS code.

python

Copy code

*# Function to load data based on file type*

def load\_data(file):

if file.name.endswith('.csv'):

data = pd.read\_csv(file, encoding='latin1')

elif file.name.endswith('.xlsx'):

xls = pd.ExcelFile(file)

sheet\_name = st.selectbox('Select sheet', xls.sheet\_names) if len(xls.sheet\_names) > 1 else xls.sheet\_names[0]

data = pd.read\_excel(file, sheet\_name=sheet\_name)

elif file.name.endswith('.json'):

data = pd.read\_json(file)

elif file.name.endswith('.parquet'):

data = pd.read\_parquet(file)

else:

data = pd.read\_csv(file, delimiter=st.text\_input('Enter delimiter', value=','), encoding='latin1')

data.columns = data.columns.str.lower() *# Convert column names to lowercase*

return data

This function, load\_data, is responsible for loading data from various file formats (CSV, Excel, JSON, Parquet, and others). It takes a file as input and returns a pandas DataFrame containing the loaded data. The function also handles special cases, such as selecting the sheet for Excel files with multiple sheets and prompting the user to enter a delimiter for other file formats. Additionally, it converts all column names to lowercase for consistency.

python

Copy code

*# Function to process user query using Claude-3 Sonnet*

def process\_query\_claude3(query):

prompt = f"Please inform user that the appropriate chart will be displayed below based on your input: {query}"

response = call\_claude3\_api(prompt, max\_chars=12000) *# Add max\_chars parameter*

comprehend\_output = st.session\_state.comprehend.detect\_entities(Text=response, LanguageCode='en')

entities = comprehend\_output['Entities']

return response, entities

This function, process\_query\_claude3, is responsible for processing the user's query using the Claude-3 Sonnet AI model from Anthropic. It constructs a prompt that asks the model to inform the user that an appropriate chart will be displayed based on their input. The function then calls the call\_claude3\_api function to interact with the Claude-3 Sonnet API and retrieve the model's response. Additionally, it uses the AWS Comprehend service to detect entities (e.g., quantities, dates, locations) in the model's response. The function returns the model's response and the detected entities.

python

Copy code

*# Function to call Claude-3 Sonnet API*

def call\_claude3\_api(prompt, max\_tokens=2000, max\_chars=12000):

*# Split the prompt into smaller chunks*

prompt\_chunks = [prompt[i:i+max\_chars] for i in range(0, len(prompt), max\_chars)]

inference\_results = []

for chunk in prompt\_chunks:

body = {

"messages": [

{

"role": "user",

"content": chunk

}

],

"anthropic\_version": "bedrock-2023-05-31",

"max\_tokens": max\_tokens

}

response = st.session\_state.bedrock\_runtime.invoke\_model(

modelId='anthropic.claude-3-sonnet-20240229-v1:0',

body=json.dumps(body)

)

inference\_result = json.loads(response['body'].read()).get("content")[0].get("text")

inference\_results.append(inference\_result)

return ' '.join(inference\_results)

This function, call\_claude3\_api, is responsible for interacting with the Claude-3 Sonnet API from Anthropic. It takes a prompt and optional parameters for maximum tokens and maximum characters. The function splits the prompt into smaller chunks if it exceeds the maximum character limit. For each chunk, it constructs a request body in the format expected by the Claude-3 Sonnet API and invokes the model using the Bedrock Runtime service from AWS. The responses from the model are collected and concatenated, and the final response is returned.

python

Copy code

*# Refined function to determine chart type based on query, entities, and Claude-3 Sonnet response*

def determine\_chart\_type(query, entities, response):

query\_lower = query.lower()

entity\_types = [entity['Type'] for entity in entities]

*# Check for explicit chart types in the query*

explicit\_chart\_types = ["bar", "line", "scatter", "histogram", "pie", "area", "box", "heatmap", "violin", "map"]

for chart\_type in explicit\_chart\_types:

if chart\_type in query\_lower:

return chart\_type

*# Check for explicit chart types in the Claude-3 Sonnet response*

for chart\_type in explicit\_chart\_types:

if chart\_type in response.lower():

return chart\_type

*# Check for implicit chart types based on entities and keywords*

if "QUANTITY" in entity\_types or any(keyword in query\_lower for keyword in ["quantity", "sales", "amount", "count", "total"]):

return "bar"

elif "DATE" in entity\_types or any(keyword in query\_lower for keyword in ["date", "time", "trend", "over time"]):

return "line"

elif "LOCATION" in entity\_types or any(keyword in query\_lower for keyword in ["location", "map", "geographical"]):

return "map"

elif any(keyword in query\_lower for keyword in ["distribution", "spread", "range"]):

return "histogram"

elif any(keyword in query\_lower for keyword in ["proportion", "percentage", "ratio"]):

return "pie"

elif any(keyword in query\_lower for keyword in ["compare", "difference", "variation"]):

return "bar"

*# Fallback to bar chart if no specific chart type is detected*

return "bar"

This function, determine\_chart\_type, is responsible for determining the appropriate chart type based on the user's query, detected entities, and the Claude-3 Sonnet response. It checks for explicit mentions of chart types (e.g., "bar", "line", "scatter") in the query and the model's response. If no explicit chart type is found, it looks for implicit cues based on the detected entities and keywords in the query. For example, if the query mentions "quantity" or "sales", it suggests a bar chart. If the query mentions "date" or "time", it suggests a line chart. If no specific chart type can be determined, it defaults to a bar chart.

python

Copy code

*# Function to autocorrect words using Levenshtein distance*

def autocorrect(word, possibilities, cutoff=0.6):

word = word.lower()

closest\_match = min(possibilities, key=lambda x: levenshtein\_distance(word, x.lower()))

*# Calculate the Levenshtein distance ratio*

distance = levenshtein\_distance(word, closest\_match.lower())

max\_length = max(len(word), len(closest\_match))

ratio = distance / max\_length

*# Return the closest match only if the ratio is below the cutoff*

if ratio < cutoff:

return closest\_match

else:

return word *# Return the original word if no close match is found*

This function, autocorrect, is responsible for autocorrecting words in the user's query based on the available column names in the dataset. It uses the Levenshtein distance algorithm to find the closest matching word from the list of possibilities (column names). If the Levenshtein distance ratio is below a specified cutoff (default is 0.6), it returns the closest matching word. Otherwise, it returns the original word.

python

Copy code

*# Function to combine tokens and match with columns*

def combine\_and\_match(tokens, columns):

combined\_attributes = []

for i in range(len(tokens)):

for j in range(i + 1, len(tokens) + 1):

combined = ''.join(tokens[i:j]).lower()

if any(combined == col.lower() for col in columns):

combined\_attributes.append(combined)

return combined\_attributes

This function, combine\_and\_match, is responsible for combining tokens (words) from the user's query and matching them with the column names in the dataset. It generates all possible combinations of tokens by iterating over the tokens and concatenating them. If any of the combined tokens match a column name (case-insensitive), it adds the combined token to a list of combined attributes. This helps identify column names that are composed of multiple words (e.g., "sales\_amount" or "product\_category").

python

Copy code

*# Process user query with AWS Comprehend and Claude-3 Sonnet*

def process\_query(query, columns):

response, entities = process\_query\_claude3(query)

chart\_type = determine\_chart\_type(query, entities, response)

*# Autocorrect tokens*

tokens = query.split()

corrected\_tokens = list(set([autocorrect(token, columns) for token in tokens]))

*# Extract and match attributes*

combined\_attributes = combine\_and\_match(corrected\_tokens, columns)

*# Ensure relevant and unique attributes*

relevant\_attributes = []

for attr in combined\_attributes:

matched\_col = next((col for col in columns if col.lower() == attr), None)

if matched\_col and matched\_col not in relevant\_attributes:

relevant\_attributes.append(matched\_col)

*# Further autocorrect to find the closest matching columns if needed*

if len(relevant\_attributes) < 2:

for token in tokens:

possible\_corrections = get\_close\_matches(token, [col.lower() for col in columns], n=1)

if possible\_corrections:

matched\_col = next((col for col in columns if col.lower() == possible\_corrections[0]), None)

if matched\_col and matched\_col not in relevant\_attributes:

relevant\_attributes.append(matched\_col)

*# Limit the number of attributes to 3*

if len(relevant\_attributes) > 2:

relevant\_attributes = relevant\_attributes[:2]

if len(relevant\_attributes) == 1:

numeric\_columns = data.select\_dtypes(include=['number']).columns

string\_columns = data.select\_dtypes(include=['object']).columns

if len(numeric\_columns) > 0 and len(string\_columns) > 0:

relevant\_attributes = [string\_columns[0], numeric\_columns[0]]

else:

st.error("Unable to identify suitable columns for the chart.")

*# If no attributes found, intelligently identify one string column for x-axis and one numeric column for y-axis*

if not relevant\_attributes:

numeric\_columns = data.select\_dtypes(include=['number']).columns

string\_columns = data.select\_dtypes(include=['object']).columns

if len(numeric\_columns) > 0 and len(string\_columns) > 0:

relevant\_attributes = [string\_columns[0], numeric\_columns[0]]

else:

st.error("Unable to identify suitable columns for the chart.")

*# Ensure string column is on the x-axis and numeric column is on the y-axis*

*#st.write('relevant\_attributes 1:', relevant\_attributes)*

if relevant\_attributes and data[relevant\_attributes[0]].dtype == 'object' and data[relevant\_attributes[1]].dtype != 'object':

x\_attr, y\_attr = relevant\_attributes

else:

y\_attr, x\_attr = relevant\_attributes

return chart\_type, x\_attr, y\_attr

This function, process\_query, is the core function that processes the user's query and identifies the relevant attributes (columns) for the chart. It performs the following steps:

1. Calls process\_query\_claude3 to get the Claude-3 Sonnet response and detected entities for the query.
2. Calls determine\_chart\_type to determine the appropriate chart type based on the query, entities, and the model's response.
3. Splits the query into tokens and autocorrects them using the autocorrect function and the available column names.
4. Calls combine\_and\_match to generate all possible combinations of tokens and match them with column names.
5. Filters the combined attributes to ensure they are unique and match the column names in the dataset.
6. If fewer than two attributes

------------------------------------------------------------------------------------------------------------------

**Explaining the NLP (Natural Language Processing) and LLM (Large Language Model) components, as well as other key modules used in this program:**

**NLP Components**

1. **AWS Comprehend**: AWS Comprehend is a natural language processing service provided by Amazon Web Services (AWS). It is used in the process\_query\_claude3 function to detect entities (such as quantities, dates, locations) in the response generated by the Claude-3 Sonnet model. The detected entities help in determining the appropriate chart type for the user's query.

python

Copy code

comprehend\_output = st.session\_state.comprehend.detect\_entities(Text=response, LanguageCode='en')

entities = comprehend\_output['Entities']

1. **Levenshtein Distance**: The Levenshtein distance is a metric used to measure the difference between two strings. In this program, the autocorrect function uses the Levenshtein distance to find the closest matching column name for each token in the user's query. This helps to handle misspellings or variations in the way the column names are mentioned in the query.

python

Copy code

from Levenshtein import distance as levenshtein\_distance

def autocorrect(word, possibilities, cutoff=0.6):

word = word.lower()

closest\_match = min(possibilities, key=lambda x: levenshtein\_distance(word, x.lower()))

*# Calculate the Levenshtein distance ratio*

distance = levenshtein\_distance(word, closest\_match.lower())

max\_length = max(len(word), len(closest\_match))

ratio = distance / max\_length

*# Return the closest match only if the ratio is below the cutoff*

if ratio < cutoff:

return closest\_match

else:

return word *# Return the original word if no close match is found*

1. **Token Matching and Combination**: The combine\_and\_match function is responsible for identifying column names that consist of multiple words (e.g., "sales\_amount" or "product\_category"). It generates all possible combinations of tokens from the user's query and checks if any of the combinations match the column names.

python

Copy code

def combine\_and\_match(tokens, columns):

combined\_attributes = []

for i in range(len(tokens)):

for j in range(i + 1, len(tokens) + 1):

combined = ''.join(tokens[i:j]).lower()

if any(combined == col.lower() for col in columns):

combined\_attributes.append(combined)

return combined\_attributes

**LLM Component**

1. **Claude-3 Sonnet from Anthropic**: Claude-3 Sonnet is a large language model developed by Anthropic. It is used in this program to interpret the user's query and provide context for determining the appropriate chart type. The call\_claude3\_api function interacts with the Claude-3 Sonnet API, sending the user's query as a prompt and receiving the model's response.

python

Copy code

def call\_claude3\_api(prompt, max\_tokens=2000, max\_chars=12000):

*# Split the prompt into smaller chunks*

prompt\_chunks = [prompt[i:i+max\_chars] for i in range(0, len(prompt), max\_chars)]

inference\_results = []

for chunk in prompt\_chunks:

body = {

"messages": [

{

"role": "user",

"content": chunk

}

],

"anthropic\_version": "bedrock-2023-05-31",

"max\_tokens": max\_tokens

}

response = st.session\_state.bedrock\_runtime.invoke\_model(

modelId='anthropic.claude-3-sonnet-20240229-v1:0',

body=json.dumps(body)

)

inference\_result = json.loads(response['body'].read()).get("content")[0].get("text")

inference\_results.append(inference\_result)

return ' '.join(inference\_results)

The process\_query\_claude3 function constructs the prompt for the Claude-3 Sonnet model and calls the call\_claude3\_api function to get the model's response.

python

Copy code

def process\_query\_claude3(query):

prompt = f"Please inform user that the appropriate chart will be displayed below based on your input: {query}"

response = call\_claude3\_api(prompt, max\_chars=12000) *# Add max\_chars parameter*

comprehend\_output = st.session\_state.comprehend.detect\_entities(Text=response, LanguageCode='en')

entities = comprehend\_output['Entities']

return response, entities

The model's response and the detected entities are then used by the determine\_chart\_type function to infer the appropriate chart type for the user's query.

**Other Key Modules**

1. **Pandas**: Pandas is a popular Python library for data manipulation and analysis. In this program, Pandas is used to load data from various file formats (CSV, Excel, JSON, Parquet), manipulate the data, and prepare it for visualization.

python

Copy code

import pandas as pd

def load\_data(file):

if file.name.endswith('.csv'):

data = pd.read\_csv(file, encoding='latin1')

*# ... (other file formats)*

1. **Plotly Express**: Plotly Express is a high-level data visualization library built on top of Plotly. It is used in this program to generate interactive visualizations based on the user's query and the identified attributes (columns).

python

Copy code

import plotly.express as px

def generate\_plot(result, plot\_type, x, y, title):

title = title.title()

if plot\_type == 'bar':

fig = px.bar(result, x=x, y=y, title=title, text=y)

*# ... (other chart types)*

return fig

1. **AWS S3**: Amazon Simple Storage Service (S3) is an object storage service provided by AWS. In this program, AWS S3 is used to fetch data files from an S3 bucket and load them into the application.

python

Copy code

import boto3

if 's3' not in st.session\_state:

st.session\_state.s3 = boto3.client('s3')

*# List files from S3 bucket*

bucket\_name = 'avengers-ba007'

file\_list = st.session\_state.s3.list\_objects\_v2(Bucket=bucket\_name)['Contents']

file\_names = [obj['Key'] for obj in file\_list]

1. **Streamlit**: Streamlit is a Python library used for building interactive web applications. It is the primary framework used in this program to create the user interface, handle user inputs, and display visualizations.

python

Copy code

import streamlit as st

*# Set the Streamlit page configuration*

st.set\_page\_config(page\_title="Interactive Data Visualization", page\_icon=":chart\_with\_upwards\_trend:", layout="wide")

*# Sidebar options*

user\_selection\_option = st.sidebar.radio("Choose Option:", ["Explore My Data!", "Explore data from AWS S3!", "Code Generator"])

*# User input and visualization*

with input\_container:

*# ... (user input components)*

with chart\_container:

*# ... (visualization components)*

These are the key components and modules used in the program for Natural Language Processing, Large Language Model integration, data manipulation, visualization, and user interface development.