

Leveraging Generative AI for Data-Driven Insights and Visualization in Python

Chitresh Goyal, Neelima Ratra, Swati Singh, Neha Kaushik

Abstract:

In this work, we introduce a novel solution that integrates Python, Streamlit, OpenAI, and embeddings to extract insights and generate visualizations from various data sources. Our approach empowers users to interact with data in plain text, enabling a more intuitive and efficient data analysis process. By leveraging Generative AI, our system not only simplifies data exploration but also addresses the challenges of identifying and generating charts, making data visualization more accessible. The solution accommodates diverse data sources, including PostgreSQL and CSV files, while ensuring a secure and user-friendly experience. This paper aims to demonstrate the effectiveness of combining unsupervised and supervised learning algorithms in real-time text analysis, providing a smart, quick, and scalable method for data-driven decision-making.

Introduction:

The advent of Generative AI has revolutionized the way we interact with data. By combining the power of OpenAI's language models with Python's versatility and Streamlit's interactivity, we have developed a solution that simplifies data exploration and visualization. Our system supports multiple data sources, including PostgreSQL and CSV files, and provides a secure and user-friendly interface for data analysis. In this paper, we delve into the details of our solution, its development journey, and the impact it has on data analysis and visualization processes.

Background and Rationale

In today's fast-paced business environment, data-driven insights are crucial for making informed decisions. Traditional data analysis tools often require specialized knowledge and can be time-consuming. There is a growing need for solutions that can provide quick and intuitive access to data insights. Our solution addresses this need by leveraging Generative AI to simplify the data analysis process and make it more accessible to users.

Problem Statement:

Despite the wealth of information contained in data, extracting meaningful insights can be challenging due to its unstructured nature. For instance, a business may need to categorize products based on descriptions, a task that traditionally requires manual effort and is prone to errors. Automating this process using unsupervised learning algorithms can save time and resources, but conventional methods like Topic Modelling lack the ability to assign understandable labels to the grouped data, making it difficult for non-technical users to interpret the results. Moreover, the addition of new data often necessitates re-running the entire analysis, further complicating the process.

Proposed Solution:

Our solution addresses these challenges by combining unsupervised and supervised learning algorithms in a two-stage process:

- Stage 1: Using historical text data, we group similar texts based on content and assign a relevant label to each group automatically, thus categorizing the texts.
- Stage 2: We build a supervised learning classification model using the labeled data from Stage 1. This model can classify new texts into the predefined categories without the need to re-run the initial grouping process.

Technical Architecture

Our solution is built on a robust technical architecture that integrates Python for backend processing, Streamlit for frontend interactivity, OpenAI's language models for natural language understanding, and embeddings for efficient data representation. This architecture allows for seamless integration of various data sources and provides a scalable framework for data analysis and visualization.

- **Efficient Data Representation with Embeddings:** A key challenge in our solution was managing the large number of tokens required for processing extensive data sources. To address this, we leveraged embeddings to represent data in a high-dimensional space. This approach significantly reduced the token count required for OpenAI queries, enhancing the efficiency of our system. By measuring similarity between data points using cosine similarity, we were able to generate more accurate and relevant insights with fewer tokens, thus optimizing our solution for scalability and performance.

```
1 text,embedding
2 ""public.tp_individual": [{"cid": "S2371", "sampleData1": "S2371", "sampleData2": "S2373"},
3 {"columnName": "cid", "columnDataType": "VARCHAR", "sampleData1": "S2371", "sampleData2": "S2373"},
4 [{"age_group": "50-54", "sampleData1": "50-54", "sampleData2": "50-54"},
5 {"columnName": "age_group", "columnDataType": "VARCHAR", "sampleData1": "50-54", "sampleData2": "50-54"},
6 [{"education_level": "HIGH SCHOOL", "sampleData1": "HIGH SCHOOL", "sampleData2": "COLLEGE"},
7 {"columnName": "education_level", "columnDataType": "VARCHAR", "sampleData1": "HIGH SCHOOL", "sampleData2": "COLLEGE"},
8 [{"gender": "MALE", "sampleData1": "MALE", "sampleData2": "MALE"}]
```

Methodology:

Our solution operates in several stages:

1. **Data Processing:** Upon receiving a data source, the system generates a schema, tables, statistics, and correlations. This structured representation of data serves as the foundation for further analysis.

```

9
10 db_config = {
11     "user": "airflow",
12     "password": "airflow",
13     "driver": "org.postgresql.Driver",
14     "host": "172.18.0.2",
15     "port": "5432",
16     "dbname": "udm_short"
17 }
18
19 sqlalchemy_url = f"postgresql://
20 {config.db_config['user']}:{config.db_config['password']}@
21 {config.db_config['host']}:
22 {config.db_config['port']}/{config.db_config['dbname']}"
23
24 def fetch_sample_data(engine, table_name, column_name, sample_size=2):
25     with engine.connect() as conn:
26         query = text(f"SELECT \"{column_name}\" FROM \"{table_name}\" WHERE \"{column_name}\" IS NOT NULL LIMIT {sample_size};")
27         result = conn.execute(query)
28         samples = [str(sample[0]) for sample in result.fetchall()]
29     return ', '.join(samples[:sample_size // 2]), ', '.join(samples[sample_size // 2:])
30
31 def execute_query(sql, sqlalchemy_url):
32     engine = create_engine(sqlalchemy_url)
33
34     df = pd.read_sql_query(sql, engine)
35     result = df.to_json(orient='records')
36     return result

```

2. **Embedding Generation:** We employ embeddings to represent the data in a high-dimensional space, enabling the measurement of similarity between different data points using cosine similarity.

```

63
64 def generate_embeddings(inputText, outputFile):
65     # Initialize total token usage counter
66     total_token_usage = 0
67     print(f"Total tokens {MAX_TOKENS}")
68     # Generate embeddings and token counts, updating total token usage
69     embeddings = []
70     accurate_token_counts = []
71     token_usages = [] # Store individual token usages
72     for text in inputText:
73         truncated_text = truncate_text(text, MAX_TOKENS)
74         embedding, token_usage = get_embedding_and_token_usage(truncated_text)
75
76         if embedding:
77             embeddings.append(embedding)
78             accurate_token_counts.append(token_count(truncated_text))
79             token_usages.append(token_usage)
80
81             if token_usage != 'Error' and token_usage != 'Unknown':
82                 total_token_usage += token_usage # Update total token usage
83         else:
84             embeddings.append(None)
85             accurate_token_counts.append(0)
86             token_usages.append('Error')
87     print(f"Chunk | {text[:20]}... | used: {token_usage}")
88
89     # Print total token usage
90     print(f"Total tokens consumed : {total_token_usage}")
91
92     # Update DataFrame to include individual token usages
93     df = pd.DataFrame({
94         "text": inputText,
95         "embedding": embeddings
96     })
97
98     # Save DataFrame to CSV
99     df.to_csv(outputFile, index=False)
100     return "data/"
101

```

3. **Contextual Query Generation:** The embeddings, along with the schema and user-defined instructions, are used to create a context for the OpenAI chat completion model. This context, combined with a user prompt and a system prompt, guides the model to generate relevant insights and tags related to the data source.

Token usage -> Prompt: 2911 | Completion: 335 | Total: 3246

GPT generated Insights and Tags

****Insights:****

1. **Demographics:** The 'tp_individual' table contains a wealth of demographic information, including age group, education level, gender, home status, income, net worth, occupation, marital status, and urbanicity. This data can be used to create detailed profiles of individuals.
2. **Household Composition:** The 'tp_individual' table also provides information about the number of adults and children in each household. This could be useful for understanding family dynamics and targeting specific household types.
3. **Contact Information:** Each individual's first name, last name, phone number, and email address are stored in the 'tp_individual' table. This information could be used for direct marketing efforts.
4. **Web Behavior:** The 'monthly_visits' and 'visits' columns in the 'tp_individual' table indicate how often each individual visits certain websites. This could provide insights into online behavior and preferences.
5. **App Usage:** The 'no_of_apps' column in the 'tp_individual' table shows how many apps each individual has in various categories. This could help identify popular app categories among different demographic groups.
6. **Segmentation:** The 'tp_segments' table contains information about different segments, including their names and descriptions. This could be useful for understanding how individuals are grouped based on their characteristics and behaviors.
7. **Web Content Categories:** The 'tp_web_categories_meta' table categorizes web content into tiers. This could be useful for understanding what types of content users are consuming.

Broader Tags:

Demographics, Household Composition, Contact Information, Web Behavior, App Usage, Segmentation, Web Content Categories

4. **Query Execution:** The system is configured to generate two PostgreSQL queries: one for creating a relevant view and another for selecting data from that view. The output is displayed as a Streamlit dataframe.

```
23 def query_views_prompt(query: str, embeddings_csv_path: str, token_budget: int = 4096-500) -> str:
24     message = """\n
25     When generating the PSQL query, adhere to these criteria:
26
27     - Apply explicit type casts to all columns for consistency.
28     - Utilize only the explicitly listed tables and columns from the provided schema and tables list.
29     - Exclude references to any tables or columns not mentioned.
30     - Craft the query to reflect the insights and tags accurately, using appropriate data.
31     - Ensure the query's syntax is correct for PostgreSQL execution without further modifications.
32     - Encase any table names with numeric digits in double quotes to comply with PostgreSQL naming conventions.
33     - The query should dynamically handle the extraction and comparison of relevant parts of the fields based on the data type and sample data provided.
34     for example if column
35     - ColumnXYZ(VARCHAR) have data "30-39" and user asks find records greater than 20.
36     it will do like : CAST(SPLIT_PART(xyz, '-', 1) AS INTEGER) > 20
37     - ColumnX(VARCHAR) have data like "[value1', 'value2', 'value3',...]"
38     - ColumnY(VARCHAR) have data like {"# Visits to Delhi": 0, "# Visits to Gurugram": 15, "# Visits to Target": 2, "# Visits to Ulta Beauty": 0}
39     - ColumnZ(VARCHAR) have data like {"Monthly_Visit_Freq_Jaipur": 0.033333333, "Monthly_Visit_Freq_Udaipur": 0.0,
40     "Monthly_Visit_Freq_Target": 0.0, "Monthly_Visit_Freq_Ulta_Beauty": 0.133333333}
41
42     Generate two PSQL queries and wrap them in code block markdown with SQL highlighting
43     - The first query should create or REPLACE PostgreSQL view.
44     - The second query should simply "select * from <view_name>".
45     """
46     question = f"\n User Prompt: {query}"
47
48     df = pd.read_csv(embeddings_csv_path)
49     df['embedding'] = df['embedding'].apply(ast.literal_eval)
50
51     embedding_lengths = df['embedding'].apply(len)
52     if len(set(embedding_lengths)) == 1:
53         print("All embeddings have consistent dimensions.")
54     else:
55         print("Warning: Inconsistent embedding dimensions found.")
56
57     strings, relatednesses = strings_ranked_by_relatedness(query, df)
58
59     relevance = ""
60     for string, relatedness in zip(strings, relatednesses):
61         # Optionally use relatedness to filter or prioritize strings here
62         next_string = f'\n\nRelevance: {relatedness}\n\n{string}'
63         if token_count(next_string + question) > token_budget:
64             break
65         relevance += next_string
66     print(f"token count: {token_count(relevance + message + question)}")
67     return "", relevance + message + question
```

- 5. Visualization:** The dataframe is used as input for generating graphs. The initial user prompt is included in the context to ensure that the generated Python code for plotting (using Streamlit and Altair) is relevant to the original query.

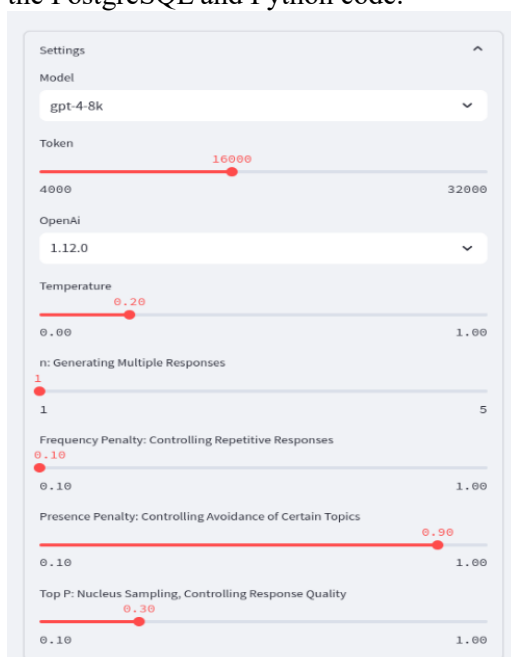
```

16 def charts_on_views_prompt(query: str, df: pd.DataFrame, csv_file_path: str) -> str:
17     # Extract DataFrame columns and a sample row
18     headers = df.columns.tolist()
19     first_row = df.iloc[0].tolist()
20
21     # Construct the prompt
22     introduction = f"""
23     Based on the insights and broader tags provided, generate Python code to:
24     1. Load the data from the CSV file at '{csv_file_path}' into a pandas DataFrame.
25     2. Extract data relevant to the provided user_query: "{query}".
26     3. Extract the name of the graph from the user_query: "{query}" and plot a graph visualizing the extracted data, highlighting key observations.
27     4. Ensure that the query handles different data formats appropriately, such as ranges, specific symbols, and text.
28
29     Ensure the Python code:
30     - Uses pandas for data manipulation and loads the data from the CSV file.
31     - Uses Altair for plotting graphs, with the final plot displayed using Streamlit.
32     For example, "st.altair_chart(altair_chart, use_container_width=True, theme="streamlit")".
33     - Is structured to be executed as a standalone script for analysis within a web application context.
34     - Handles any necessary data preprocessing or transformation as implied by the insights and the user request "{query}".
35     - use different colors in graph if suitable
36
37     Strictly follow:
38     - Do not hard code values provided in the example; those values are just for reference.
39     - Provide only working code enclosed between ``python <yourCodeGoesHere>``
40     """
41
42     question = f"Question: {query}"
43     prompt = f"DataFrame columns: {headers}\nSample row: {first_row}\n{introduction}\n{question}"
44     return prompt
45

```

Key Features:

- 1. Security:** No data is sent to OpenAI, ensuring the privacy and security of the data.
- 2. Embeddings:** Utilization of embeddings for efficient similarity measurement and context generation.
- 3. Clear Schema Context:** The detailed schema context allows users to write complex queries with ease.
- 4. Configurable OpenAI Chat Completion:** The system allows for customization of parameters such as `n`, `top_n`, `temperature`, `frequency_penalty`, and `presence_penalty` to tailor the output to specific needs.
- 5. Robust Executors:** The solution includes a Python executor and a PostgreSQL executor with proper handlers to execute code and queries safely.
- 6. Restricted Operations:** To ensure data integrity, the system restricts delete and update operations in both the PostgreSQL and Python code.



The screenshot shows a settings panel with the following controls:

- Model:** A dropdown menu set to "gpt-4-8k".
- Token:** A slider ranging from 4000 to 32000, currently set at 16000.
- OpenAI:** A dropdown menu set to "1.12.0".
- Temperature:** A slider ranging from 0.00 to 1.00, currently set at 0.20.
- n: Generating Multiple Responses:** A slider ranging from 1 to 5, currently set at 1.
- Frequency Penalty: Controlling Repetitive Responses:** A slider ranging from 0.10 to 1.00, currently set at 0.10.
- Presence Penalty: Controlling Avoidance of Certain Topics:** A slider ranging from 0.10 to 1.00, currently set at 0.90.
- Top P: Nucleus Sampling, Controlling Response Quality:** A slider ranging from 0.10 to 1.00, currently set at 0.30.

Uniqueness of the Solution:

Our solution stands out by establishing an algorithmic process that not only groups text documents efficiently but also assigns meaningful labels to each group, a feature not offered by traditional Topic Modelling algorithms. Furthermore, the predictive model built in Stage 2 enables automatic classification of future texts, eliminating the need for repetitive analysis.

Validation and Experiments:

To validate our proposed solution, we followed these steps for each stage:

- Stage 1:

Data Pre-processing and Cleaning: We cleaned the text data of special characters, unnecessary numbers, and stop words, and used tokenization to break the text into smaller chunks for analysis.

Optimal Number of Topics: Using coherence scores, we determined the optimal number of topics for Topic Modelling.

Topic Model Building: We trained an unsupervised machine learning topic model on the data, assigning a topic number to each document/text.

Naming Each Topic: We used Large Language Models like GPT-3.5 Turbo for automatic generation of names for each topic.

- Stage 2:

Data Division: We divided the labeled data into training and testing datasets.

Classification Model Building: We built a Naïve Bayes Text Classification Model using the training data.

Model Validation and Deployment: We validated the model using AUC and deployed it for predicting labels of unknown texts.

Results and Analysis:

Our solution successfully categorized and grouped product descriptions based on their content, with the model accurately grouping similar descriptions and assigning appropriate labels. The predictive model built in Stage 2 demonstrated high accuracy in classifying new texts into the identified categories, streamlining the text analysis process and enabling real-time, efficient data categorization.

Your message

Create a chart displaying the distribution of the number of males and females for the whole population.

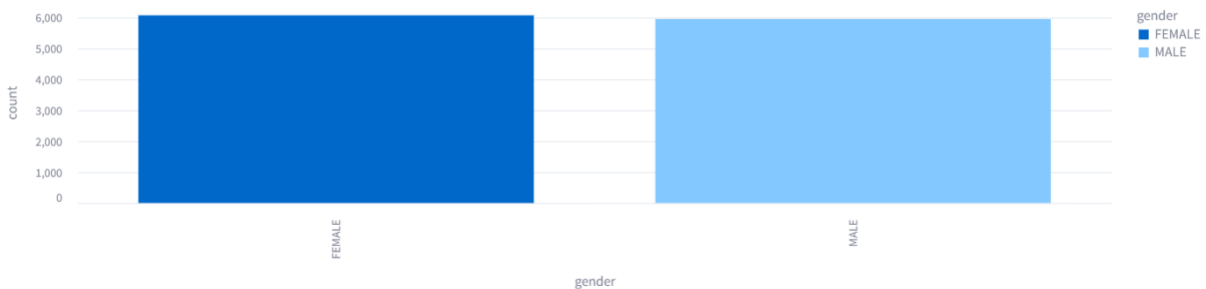
Submit

Token usage -> Prompt: 3228 | Completion: 52 | Total: 3280

| | gender | count |
|---|--------|-------|
| 0 | MALE | 5,941 |
| 1 | FEMALE | 6,059 |

Token usage -> Prompt: 521 | Completion: 285 | Total: 806

Distribution of Males and Females in the Population



AI:

```
CREATE OR REPLACE VIEW gender_distribution AS
SELECT
  gender::VARCHAR AS Gender,
  COUNT(*) AS Count
FROM
  public.tp_individual
GROUP BY
  gender;

SELECT * FROM gender_distribution;
```

My Journey in Developing the Solution:

1. Initial Challenges with CSV Data:

- Encountered issues with graph plotting accuracy when using the entire dataframe.
- Resolved by sharing sample data instead of the full dataset to maintain graph accuracy without exposing sensitive information.
- Faced a major setback when the entire POC stopped working due to updates in the pandas library and azureopenai, leading to the abandonment of the langchain, azureopenai, and pandas combination.

2. Overcoming Challenges and Redesigning the Solution:

- Switched to using OpenAI versions 0.28 and 1.12 for chat completion and completion APIs.
- Adopted smart dataframes for better data handling and implemented regex-based handlers for script security.
- Redesigned the entire approach by sending a clear context of schema, correlation, statistics, user prompt, system prompt, and instructions to OpenAI, along with using embeddings to reduce the number of tokens.

3. Implementing the Final Solution:

- Successfully executed Python scripts generated by GPT for direct CSV data analysis and graph plotting.
- Addressed security concerns by ensuring that the scripts do not alter data and by creating a temporary CSV path for the uploaded file to be used in the script.
- Utilized Altair and Bokeh libraries for beautiful chart plotting, with only these two libraries installed in the environment to prevent errors from other libraries.

4. Expanding to Database Data Analysis:

- Leveraged the CSV POC to handle database sources, initially connecting to databases using SQLAlchemy and formatting the schema similarly to CSV data.
- Faced challenges with large data leading to heavy token usage in OpenAI queries, which was mitigated by sharing a sample of scrubbed data.
- Adopted OpenAI's GPT-4 model for improved query generation and fine-tuned the solution with embeddings, descriptive schema, and configurations like `n`, `top_n`, and `temperature` for better accuracy.
- **Continuous Improvement and Future Directions:**
- Continuously improved the solution based on feedback and new insights, such as using embeddings over fine-tuning for better accuracy.
- Explored the potential of integrating a vector database to enhance accuracy further by storing feedback and maintaining an accuracy level rating for responses.
- Planned to seek feedback from senior management and explore additional applications of Generative AI in data analysis and query generation.

User Query Samples:

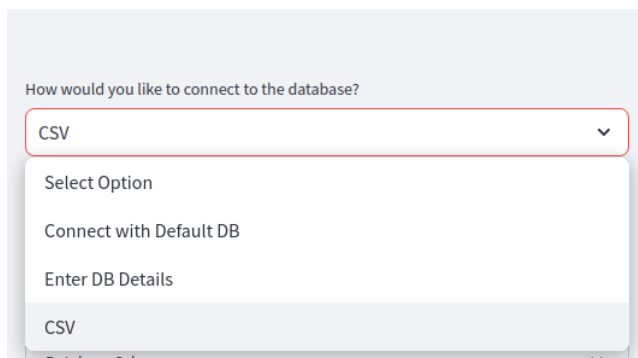
- CSV: "Create a chart displaying the distribution of the number of males and females for the whole population."
- DB: "Give me a clear view of the count of married and unmarried house owners."

Use Cases and Applications

Our solution has broad applications across industries such as healthcare, finance, and marketing. For example, in healthcare, it can be used to analyze patient data and identify trends, while in finance, it can be used to visualize market data and make investment decisions.

User Experience and Interface

The user interface is designed to be intuitive and user-friendly, allowing users to interact with data using natural language. The workflow is streamlined to simplify data analysis and visualization.



How would you like to connect to the database?

CSV

Select Option

Connect with Default DB

Enter DB Details

CSV

Database Schema

Future Enhancements

Future enhancements include integrating AI and machine learning for advanced data analysis, adding more data sources, and improving the user interface for an even more seamless experience.

Community and Collaboration

The development of our solution has been a collaborative effort, with contributions from the open-source community, partnerships with other organizations, and valuable user feedback. We encourage further collaboration to enhance the solution.

Conclusion:

Our solution demonstrates the potential of Generative AI in enhancing data analysis and visualization. By providing a user-friendly interface and leveraging the capabilities of OpenAI's language models, we enable users to gain deeper insights into their data and make informed decisions.

References:

- OpenAI API Documentation: <https://platform.openai.com/docs/api-reference/introduction>
- Azure OpenAI Service QuickStart: <https://learn.microsoft.com/en-us/azure/ai-services/openai/chatgpt-quickstart?tabs=command-line%2Cpython&pivots=programming-language-python>
- Chat Completion Parameters & Embeddings: <https://learn.microsoft.com/en-us/azure/ai-services/openai/reference>
- GPT 3.5 Fine Tuning: <https://learn.microsoft.com/en-us/azure/ai-services/openai/tutorials/fine-tune?tabs=python%2Cpowershell>
- Customize a Model with Fine Tune (from UI): <https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/fine-tuning?tabs=turbo%2Cpython&pivots=programming-language-studio>
- Embeddings Tutorial: <https://learn.microsoft.com/en-us/azure/ai-services/openai/tutorials/embeddings?tabs=python%2Ccommand-line&pivots=programming-language-python>
- LAMA: <https://github.com/facebookresearch/LAMA>
- Embedding Wikipedia Articles for Search: https://cookbook.openai.com/examples/embedding_wikipedia_articles_for_search
- Question Answering Using Embeddings-Based Search: https://cookbook.openai.com/examples/question_answering_using_embeddings
- Understanding ChatGPT Embedding: <https://medium.com/@iamamellstephen/understanding-chatgpt-embedding-unveiling-the-core-of-conversational-ai-13b792ea0f92>
- Build a Chatbot on Your CSV Data with Langchain and OpenAI: <https://betterprogramming.pub/build-a-chatbot-on-your-csv-data-with-langchain-and-openai-ed121f85f0cd>
- Model Parameters in OpenAI API: <https://medium.com/nerd-for-tech/model-parameters-in-openai-api-161a5b1f8129>
- Voyage: Embeddings in Langchain and Chat Langchain: <https://blog.langchain.dev/voyage-embeddings-in-langchain-and-chat-langchain/>
- Retrieval: <https://blog.langchain.dev/retrieval/>