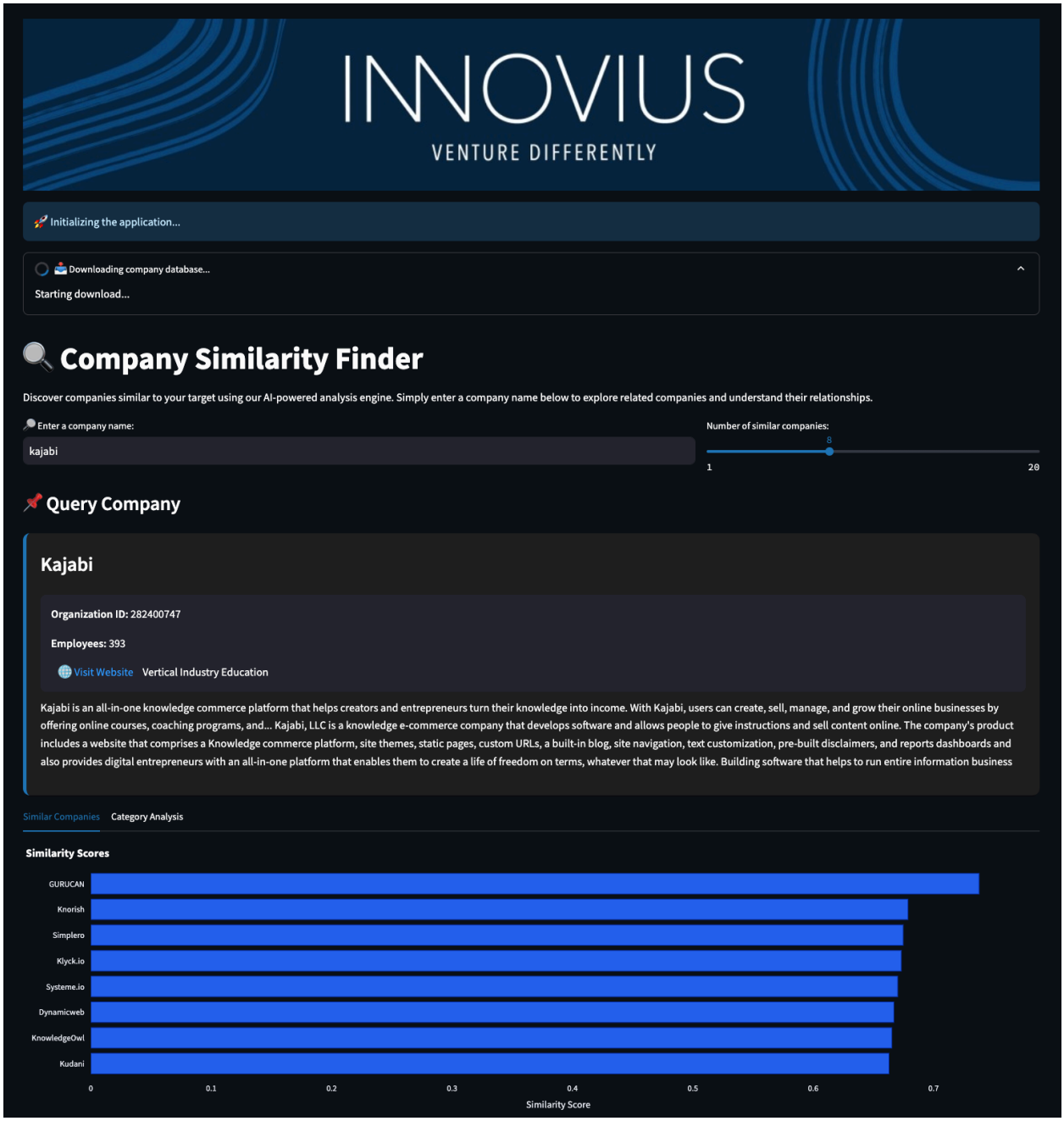
Mohan Bhosale

Technical Case Study Report - Innovious Capital

Web application: Try it out here!



Author: **Mohan Bhosale**

[Likedin] x [Email]

Repo: Link

Submission Folder: Link

(Submission folder includes Python notebook and pickle file)

Overview of the Solution:

This project involves developing a **Company Similarity Finder** web application that enables users to find and explore companies similar to a specified company based on their descriptions. The core components and processes of the project are as follows:

- **Embedding Generation using Python Notebook**: We utilized a Python notebook to preprocess company descriptions and generate embeddings. Advanced Natural Language Processing (NLP) techniques, specifically using **Sentence-BERT (SBERT)**, were employed to create meaningful vector representations of the textual data.
- **Handling Large Data Files on the Cloud**: Given the substantial size of the dataset (~55,000 companies), the generated embeddings and company data were stored in a large pickle file. This file was uploaded to **Google Drive**, a cloud storage service, to facilitate easy access and scalability without the need for local storage of large files.
- **Accessing Data in the Streamlit App**: The web application, built using **Streamlit**, accesses the large data file stored on the cloud. Upon initialization, the app downloads the data file, loads the company data and embeddings, and uses them to perform efficient similarity searches. This approach allows the application to handle large datasets effectively within the constraints of web application environments.

By combining these components, the application provides a scalable and efficient solution for users to discover and analyze company similarities based on textual descriptions.

Python Notebook Work:

Introduction

The objective of this work is to develop a solution that computes and visualizes similarity scores between companies based on their descriptions. This analysis is valuable for understanding market trends, competition, and identifying potential investment opportunities. The dataset consists of approximately 55,000 company descriptions, including attributes such as company IDs, names, websites, industry tags, and employee counts.

This report documents the entire process undertaken in a Python notebook environment (Google Colab), detailing each step from data preprocessing to similarity score calculation, addressing challenges encountered, and the solutions implemented.

Data Preprocessing

Data preprocessing is a critical step in any Natural Language Processing (NLP) project. It involves cleaning and transforming raw text data into a format suitable for analysis and modeling.

Loading the Dataset

Objective: Load the dataset from an Excel file into a pandas DataFrame.

- **Function Used:** `pandas.read_excel()`

Explanation:

- Used `pandas` to read data from an Excel file, specifying the correct sheet name.
- Displayed initial rows to verify data loading.

Exploring the Data

Objective: Understand the structure and content of the dataset.

- **Functions Used:**
    - `DataFrame.info()`
    - `DataFrame.isnull().sum()`
    - `DataFrame.head()`

Explanation:

- Reviewed data types and non-null counts.
- Checked for missing values in each column.
- Viewed sample data to understand the text content.

Handling Missing Values

Objective: Remove rows with missing essential data (descriptions).

- **Function Used:** `DataFrame.dropna()`

Explanation:

- Removed rows where the 'Description' column is null.
- Reset the DataFrame index after dropping rows.

Combining Description Columns

Objective: Merge multiple description columns to create a comprehensive text for each company.

- **Function Used:** `DataFrame.apply()`

Explanation:

- Combined 'Description' and 'Sourcscrub Description' columns.
- Used a lambda function to concatenate descriptions, ensuring only non-null strings are joined.

Cleaning Text Data

Objective: Normalize text by removing noise and ensuring consistency.

- **Techniques Used:**

- Converting text to lowercase.
- Removing URLs using regular expressions (`re.sub()`).
- Eliminating punctuation and special characters.
- Removing extra whitespace.

Explanation:

- Standardized text to improve consistency.
- Reduced noise in the data by removing irrelevant characters.

Removing Stop Words and Lemmatization

Objective: Reduce words to their base form and remove common words that add little value.

- **Libraries and Functions Used:**
  - NLTK (`nltk.download()`)
  - Stop words removal (`stopwords.words('english')`)
  - Lemmatization (`WordNetLemmatizer`)

Explanation:

- Removed common English stop words that do not contribute significant meaning.
- Lemmatized words to reduce them to their base form, aiding in grouping similar words.

Feature Extraction

Feature extraction transforms text data into numerical representations suitable for machine learning models.

Choosing the Right Technique

Selected Method: Sentence-BERT (SBERT)

Rationale:

- Captures semantic meaning of sentences.

- Efficient and provides high-quality embeddings.
- Utilizes transformer models for superior performance.

Installing Required Libraries

- **Library Installed:** `sentence-transformers`

Explanation:

- Required to access pre-trained SBERT models.

Loading Preprocessed Data

- Ensured that the DataFrame from the preprocessing step is available for feature extraction.

Loading the Pre-trained SBERT Model

- **Function Used:** `SentenceTransformer()`

Explanation:

- Loaded the `'all-MiniLM-L6-v2'` model, which balances performance and computational efficiency.

Generating Embeddings

- **Function Used:** `model.encode()`

Explanation:

- Generated vector representations (embeddings) for the processed descriptions.
- Stored embeddings back into the DataFrame for easy access.

Similarity Score Calculation

Calculating similarity scores involves measuring how close company descriptions are in the embedding space.

Challenges with Large Datasets

- **Issue:** Computing a full similarity matrix for ~55,000 companies requires excessive memory (~24 GB).
- **Constraint:** Google Colab typically offers around 12 GB of RAM, leading to crashes when attempting to compute large matrices.

Using FAISS for Efficient Similarity Search

FAISS (Facebook AI Similarity Search) is a library designed for efficient similarity search on large datasets.

Installing FAISS

- **Library Installed:** `faiss-cpu`

Normalizing Embeddings

- **Techniques Used:**
  - Normalization of embeddings using NumPy (`np.linalg.norm()`).
  - Conversion to `float32` data type (required by FAISS).

Building the FAISS Index

- **Functions Used:**
  - `faiss.IndexFlatIP()`
  - `index.add()`

Explanation:

- Initialized an inner product index for cosine similarity.
- Added normalized embeddings to the FAISS index.

Function to Retrieve Similar Companies

- **Function Created:** `get_similar_companies_faiss()`

Explanation:

- Retrieves top N similar companies using FAISS.
- Excludes the query company from the results.
- Returns company information along with similarity scores.

Example Usage

- Used the function to find companies similar to a specified company name.
- Handled cases where the company might not be found in the dataset.

Alternative: Computing Similarities On-the-Fly

For situations where FAISS cannot be used, similarities can be computed when needed.

Function to Compute Similarities for One Company

- **Function Created:** `get_similar_companies_on_the_fly()`

Techniques Used:

- Computed similarities in batches to reduce memory usage.
- Used `sklearn.metrics.pairwise.cosine_similarity()` for computing cosine similarities.

Explanation:

- Calculated similarities only for the selected company.
- Managed memory usage by processing in smaller batches.

Conclusion

Throughout this project, we have:

- **Preprocessed the Data:**
  - Cleaned and normalized text data.
  - Handled missing values and combined multiple description fields.
  - Removed stop words and lemmatized text for better semantic representation.
- **Extracted Features:**
  - Used Sentence-BERT to generate high-quality embeddings of company descriptions.
  - Leveraged state-of-the-art NLP techniques to capture semantic meaning.
- **Calculated Similarity Scores:**
  - Addressed memory limitations when computing full similarity matrices.
  - Implemented FAISS for efficient similarity search on large datasets.
  - Provided an alternative on-the-fly computation method for scenarios where FAISS isn't suitable.

Key Takeaways:

- **Efficient Processing:** By using FAISS and on-the-fly computations, we effectively managed computational resources.
- **Scalability:** The methods implemented are scalable and can handle the addition of new data.
- **Practical Application:** The similarity analysis can aid in market research, competition analysis, and investment decision-making.

Company Similarity Finder Web Application:

Introduction

The **Company Similarity Finder** is a web application developed using **Streamlit** that allows users to find and explore companies similar to a specified company based on their descriptions. It leverages advanced Natural Language Processing (NLP) techniques and efficient similarity search algorithms to provide quick and accurate results.

This report details the implementation of the web application, focusing on how large datasets are accessed and processed, the techniques used for efficient computation, and how the application is designed to be scalable with the addition of new companies.

Web Application Overview

The application performs the following key functions:

- **Data Loading**: Downloads and loads a large dataset containing company information and precomputed embeddings.
- **Data Processing**: Cleans and preprocesses company descriptions to ensure consistency.
- **Similarity Search**: Uses **FAISS** (Facebook AI Similarity Search) for efficient similarity computation.
- **User Interaction**: Provides an intuitive user interface for inputting company names and viewing results.
- **Visualization**: Displays similarity scores and category distributions using interactive charts.

Key Components and Techniques

Data Loading and Caching

Objective: Efficiently load a large dataset and reduce redundant computations.

Techniques Used:

- **Google Drive Integration**: The dataset (a pickle file containing company data and embeddings) is stored on Google Drive and downloaded when the application runs.
- **Caching with `@st.cache_data`**: Streamlit's caching mechanism is used to store the data in memory, avoiding re-downloading and re-processing on every interaction.
- **Conditional Downloading**: The application checks if the data file already exists locally to prevent unnecessary downloads.

Functions:

- `download_data()`: Downloads the data file from Google Drive if not already present.
- `load_data()`: Loads the data into memory, processes embeddings, and initializes the FAISS index.

Explanation:

- **Data Path Verification**: Before downloading, the application checks if the data file exists locally.
- **Download with Progress Feedback**: Uses `gdown` to download the file, providing user feedback through Streamlit's status messages.
- **Data Loading**: Reads the pickle file into a pandas DataFrame and extracts embeddings.
- **Memory Management**: Employs garbage collection (`gc.collect()`) and clears GPU memory (if available) to optimize resource usage.

Data Processing and Cleaning

Objective: Ensure company descriptions are clean and formatted for display and further processing.

Techniques Used:

- **Regular Expressions (`re`)**: Used to clean HTML tags and unwanted elements from company descriptions.
- **String Manipulation**: Formatting website URLs and handling missing values.

Functions:

- `clean_description(text)`: Cleans and formats the company description text.
- `format_website_link(website)`: Ensures website URLs are properly formatted as hyperlinks.
- `display_company_details(company)`: Renders company information in a formatted card layout.

Explanation:

- **HTML Tag Removal**: Strips out incomplete and complete HTML tags to prevent rendering issues.
- **Whitespace Normalization**: Ensures consistent spacing in the descriptions.
- **Safe Handling of Missing Data**: Checks for `NaN` values to prevent errors during string operations.

Embedding Retrieval and FAISS Indexing

Objective: Perform efficient similarity searches on high-dimensional data.

Techniques Used:

- **Normalized Embeddings**: Embeddings are normalized to unit length to facilitate cosine similarity computation via inner product.
- **FAISS Indexing**: Utilizes FAISS for fast nearest neighbor searches in high-dimensional space.

Functions:

- `load_data()`: Processes embeddings and builds the FAISS index.
- `get_similar_companies()`: Retrieves similar companies using the FAISS index.

Explanation:

- **Embedding Normalization**: Converts embeddings to `float32` and normalizes them to unit vectors.
- **FAISS Index Creation**: An inner product index (`IndexFlatIP`) is created, and embeddings are added to it.
- **Similarity Search**: Given a query company's embedding, the index returns the most similar companies based on cosine similarity.

User Interface and Interactivity

Objective: Provide an intuitive and responsive interface for user interaction.

Techniques Used:

- **Streamlit Widgets**: Text inputs, sliders, buttons, and tabs are used for user inputs and navigation.
- **Custom CSS Styling**: Enhances the visual appeal with a dark theme and custom styles for components.
- **Session State Management**: Maintains application state across interactions to improve performance.

Functions:

- `display_banner()`: Displays the application header image or a fallback text.
- **Streamlit Widgets in `main()`**: Collects user inputs for the company name and number of similar companies to display.

Explanation:

- **Input Validation**: Checks if the entered company name exists in the dataset and handles errors gracefully.
- **Dynamic Content Rendering**: Updates the displayed results based on user inputs without reloading the entire page.

- **Session State Variables**: Variables like `data_loaded` and `download_attempted` prevent redundant data loading and downloading.

Visualization and Output

Objective: Present the results in an informative and visually appealing manner.

Techniques Used:

- **Plotly Charts**: Interactive bar and pie charts for visualizing similarity scores and category distributions.
- **Markdown and HTML Rendering**: Displays company details in a structured format using HTML within Streamlit.

Functions:

- `create_similarity_chart(similar_companies)`: Generates a horizontal bar chart of similarity scores.
- `create_category_distribution(similar_companies)`: Creates a pie chart showing the distribution of top-level categories.
- `display_company_details(company)`: Shows individual company details in a card format.

Explanation:

- **Interactive Charts**: Users can hover over chart elements to get more information.
- **Export Functionality**: Allows users to download the results as a CSV file for further analysis.

Data Engineering & Scalability

Efficient Data Pipeline Design

Objective: Design a pipeline that efficiently processes and updates similarity scores as new companies are added.

Approach:

- **Incremental Data Updates**: When new companies are added, their descriptions are preprocessed, and embeddings are computed using the same SBERT model.
- **Updating the FAISS Index**: The new embeddings are normalized and added to the existing FAISS index without rebuilding it from scratch.
- **Data Persistence**: Updated embeddings and the DataFrame are saved back to the pickle file for future use.

Explanation:

- **Scalability**: The ability to add new data without reprocessing the entire dataset ensures the application scales with growing data.
- **Efficiency**: Incremental updates are computationally less intensive than full re-computation, making the pipeline efficient.

Scalability and Performance Considerations

Challenges:

- **Large Dataset Handling**: Loading and processing large datasets can be memory-intensive.
- **Real-Time Performance**: Ensuring the application responds quickly to user inputs.

Solutions Implemented:

- **Data Caching**: Uses Streamlit's `@st.cache_data` decorator to cache expensive operations.
- **Memory Management**: Employs garbage collection and clears GPU cache to free up resources.

- **Efficient Libraries**: Utilizes FAISS, which is optimized for high-dimensional similarity search.

Future Considerations:

- **Distributed Computing**: For extremely large datasets, consider distributed processing frameworks like Apache Spark.
- **Asynchronous Loading**: Load data asynchronously to improve application startup time.

Updating Similarity Scores with New Data

Process:

1. **Data Preprocessing**:
    - Clean and preprocess the new company descriptions using the same methods.
2. **Embedding Generation**:
    - Compute embeddings for the new companies using the SBERT model.
3. **Normalization**:
    - Normalize the new embeddings to unit vectors.
4. **Index Update**:
    - Add the new embeddings to the FAISS index using `index.add()`.
5. **DataFrame Update**:
    - Append the new company data and embeddings to the existing DataFrame.
6. **Data Persistence**:
    - Save the updated DataFrame and embeddings to the pickle file.

Benefits:

- **No Reprocessing Required**: Existing data remains untouched, saving computational resources.
- **Seamless Integration**: New data becomes immediately available for similarity searches.
- **Scalable Solution**: The pipeline accommodates data growth without significant performance degradation.

Conclusion

The Company Similarity Finder web application effectively demonstrates how large-scale NLP tasks can be implemented efficiently and scalably using modern technologies. By leveraging advanced techniques like SBERT for embeddings and FAISS for similarity search, the application provides quick and accurate results even with large datasets.

Key achievements include:

- **Efficient Data Handling**: Through caching and optimized libraries, the application manages large datasets without overwhelming system resources.
- **Scalable Design**: The data pipeline accommodates the addition of new companies with minimal overhead.
- **User-Friendly Interface**: Provides an intuitive platform for users to explore company similarities and gain valuable insights.