**Implementing RAG for marketing use cases**

|  |  |
| --- | --- |
| **Summary** | Recommender with tokenization and QA engine with BERT |
| **URL** | https://github.com/prithvivasireddy/recc\_app |
| **Status** | Published |
| **Author** | Prithvi, Vasireddy |

[Introduction](#_ok7k5uux6)

[What you’ll build](#_21yzqg98x7h6)

[What you’ll need](#_319h1e1sjzdz)

[Getting set up](#_bwo0af2iwk90)

[Get a key for the Dark Sky API](#_dy319v7ol5mn)

[Verify your API key is working properly](#_z7dwjt3xqui9)

[Get the code](#_q6selyhmjlab)

[What’s next?](#_9mqifffit2ew)

[Further reading](#_lqephukhewqt)

[Reference docs](#_an95hoqxrlhg)

# Introduction

The repo contains a Streamlit app that allows users to explore a recommender system for electronics products. The app has three main features:

* Recommend from similar items: This feature allows users to get recommendations for similar products based on the title of a product. The app uses a variety of methods to calculate similarity, including cosine similarity, tf-idf similarity, and a TensorFlow Recommenders model.
* Recommendations based on customer purchase history: This feature allows users to get recommendations for products that other customers who have purchased similar products have also purchased. The app uses a collaborative filtering algorithm to calculate these recommendations.
* Product Captioning: This feature allows users to get captions for product images. The app uses a transformer-based model to generate these captions.

The app is still under development, but it is a good starting point for exploring recommender systems for electronics products.

Here are some additional details about the repo:

* The app is written in Python and uses the Streamlit library.
* The app requires the following libraries:
  + Streamlit
  + Pandas
  + NumPy
  + Scikit-learn
  + TensorFlow

## What you’ll need

* A recent version of Chrome (74 or later)  
  PWAs are just web apps, and work in all browsers, but we’ll be using a few features of the Chrome DevTools to better understand what’s happening at the browser level, and use it to test the install experience.
* Knowledge of HTML, CSS, JavaScript, and [Chrome DevTools](https://developer.chrome.com/devtools).

# **Getting set up and Pre-processing**

Vectorization can be done via the PineCone API key. In order to use it, you’ll need to request an API key. It’s easy to use, and free for non-commercial projects.

[Register for API Key](https://platform.openai.com/account/api-keys)

### \*\*Verify your API key is working properly\*\*

|  |
| --- |
| **Caution:** when you try to deploy your Application using Pinecone’s api on Streamlit:  openai.error.AuthenticationError: This app has encountered an error. The original error message is redacted to prevent data leaks. Full error details have been recorded in the logs    You would encounter such an authentication error when you expose your API key in your code. Any secret variables such as API keys or passwords should be made accessible to code through *Secrets.* Moreover when you hardcode your key value in the code on a ***public*** GitHub repository, you are allowing others to take advantage of it.  Refer this document on how to manage your secrets: [https://docs.streamlit.io/streamlit-community-cloud/get-started/deploy-an-app/connect-to-data-sources/secrets-management](https://nam12.safelinks.protection.outlook.com/?url=https%3A%2F%2Fdocs.streamlit.io%2Fstreamlit-community-cloud%2Fget-started%2Fdeploy-an-app%2Fconnect-to-data-sources%2Fsecrets-management&data=05%7C01%7Cvasireddy.p%40northeastern.edu%7Cafb157c136e3442389f708db30ab3a91%7Ca8eec281aaa34daeac9b9a398b9215e7%7C0%7C0%7C638157284194974985%7CUnknown%7CTWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D%7C3000%7C%7C%7C&sdata=L2B50yGKQ%2FeHagX%2FzmqnBihaCxv%2FJA4KXRp6ynQCrEM%3D&reserved=0)  Although it is not recommended, you can also solve this problem by making your GitHub repository **Private.** |

**Pre-Processing and Transformations applied on data (articles.csv and customers.csv):**

**1. \*\*Extracting Features from Images\*\*:** The images of the products were analyzed to extract features like color, texture, and style. These features were likely used to compute the similarity between different items.

Example: In `caption\_desc\_embeds.csv`, the `similarity` column contains similarity scores between the product descriptions and their images.

**2. \*\*Text Processing\*\*:** The text descriptions in the `articles.csv` dataset were processed and used to generate similarity scores between different items based on their textual descriptions.

Example: In `articles\_rcmnds.csv`, `text\_rcmnds`, and `text\_scores` columns contain recommendations and scores based on textual similarity.

**3. \*\*Feature Extraction\*\*:** Various features like graphical appearance, color group, perceived color, department, etc., were extracted from the `articles.csv` dataset and used to compute similarity between different items.

Example: In `articles\_rcmnds.csv`, `feature\_rcmnds`, and `feature\_scores` columns contain recommendations and scores based on extracted features.

**4. \*\*Combining Recommendations\*\*:** The recommendations from different similarity measures (image, text, features) were combined into a single set of recommendations for each item.

Example: In `articles\_rcmnds.csv`, `combined\_rcmnds`, and `combined\_scores` columns contain recommendations and scores that combine information from image, text, and features.

**5. \*\*Customer History\*\*:** The purchase history of customers was used to generate personalized recommendations for each customer based on their preferences.

Example: In `customers\_rcmnds.csv`, the `history` column contains the purchase history of each customer, and the other columns contain personalized recommendations and scores based on different similarity measures.

To reproduce these processing and transformation steps, you would need to use techniques such as image processing (e.g., CNNs for feature extraction), natural language processing (e.g., text embeddings), and similarity computation (e.g., cosine similarity) to generate recommendations based on various types of information. Once you have recommendations from different sources, you can combine them into a single set of recommendations and provide personalized recommendations to customers based on their purchase history.

**Hm\_QA**

The script sets the layout for the Streamlit page and prompts a QA interface that allows users help understand about data. This was done in DistilBert. Since this is only a Beta, little data has been fed to the engine. And because, BERT is not really augmentive in nature like other RAGS (GPT-3, 4 etc.), even with context, given the little subset we have provided for training, it can tend to produce irrelevant results.

**Knowledgebase for BErT:**

In general, a knowledge base is a structured collection of data, information, and knowledge that can be used for reference, analysis, and decision-making. It is an organized repository that stores information and knowledge that is useful for a particular domain or application.

The term "knowledge base" is often used in the context of artificial intelligence and machine learning, where it refers to a database of knowledge that is used to train and improve machine learning models. The knowledge base contains information about the domain in which the model is intended to operate, including rules, patterns, and other types of data that can be used to inform the model's predictions.

In some applications, a knowledge base may also include information about the context in which the data was collected, such as the time and location of the observations or the conditions under which the data was collected. This information can be used to help interpret the data and make more accurate predictions.

Knowledge bases are used in a variety of applications, including natural language processing, computer vision, robotics, and expert systems. They can also be used in more general applications, such as e-commerce and customer service, where they are used to provide information and assistance to customers.

Overall, a knowledge base is an essential component of many intelligent systems, providing a structured repository of information and knowledge that can be used to inform decision-making and improve performance.

\*The BerT was trained on question, answer and context based answering.

`example:

# Add question-context-answer pair for product name

qa\_pairs.append({

"question": f"What is the product name of article ID {article\_id}?",

"context": f"Article ID {article\_id} is {row['prod\_name']}.",

"answer": row['prod\_name']

})

# Add question-context-answer pair for product type name

qa\_pairs.append({

"question": f"What is the product type name of article ID {article\_id}?",

"context": f"Article ID {article\_id} has the product type {row['product\_type\_name']}.",

"answer": row['product\_type\_name']

})

# Add question-context-answer pair for product group name

qa\_pairs.append({

"question": f"What is the product group name of article ID {article\_id}?",

"context": f"Article ID {article\_id} belongs to the product group {row['product\_group\_name']}.",

"answer": row['product\_group\_name']

})

# Add question-context-answer pair for graphical appearance name

qa\_pairs.append({

"question": f"What is the graphical appearance name of article ID {article\_id}?",

"context": f"Article ID {article\_id} has a graphical appearance of {row['graphical\_appearance\_name']}.",

"answer": row['graphical\_appearance\_name']

})

# Add question-context-answer pair for color group name

qa\_pairs.append({

"question": f"What is the color group name of article ID {article\_id}?",

"context": f"Article ID {article\_id} has the color group {row['colour\_group\_name']}.",

"answer": row['colour\_group\_name']

})

# Add question-context-answer pair for department name

qa\_pairs.append({

"question": f"What is the department name of article ID {article\_id}?",

"context": f"Article ID {article\_id} is in the department {row['department\_name']}.",

"answer": row['department\_name']

})

# Add question-context-answer pair for index code

qa\_pairs.append({

"question": f"What is the index code of article ID {article\_id}?",

"context": f"Article ID {article\_id} has the index code {row['index\_code']}.",

"answer": row['index\_code']

})

# Add question-context-answer pair for index group name

qa\_pairs.append({

"question": f"What is the index group name of article ID {article\_id}?",

"context": f"Article ID {article\_id} is part of the index group {row['index\_group\_name']}.",

"answer": row['index\_group\_name']

})

# Add question-context-answer pair for section name

qa\_pairs.append({

"question": f"What is the section name of article ID {article\_id}?",

"context": f"Article ID {article\_id} is in the section {row['section\_name']}.",

"answer": row['section\_name']

})

# Add question-context-answer pair for garment group name

qa\_pairs.append({

"question": f"What is the garment group name of article ID {article\_id}?",

"context": f"Article ID {article\_id} belongs to the garment group {row['garment\_group\_name']}.",

"answer": row['garment\_group\_name']

.

.

.

.

.

})

]

THIS creates a knowledge base in the form of qa\_pairs.json for further fine-tuning BerT.

**Fine-tuning BeRT:**

Using Hugging Face Transformers would require us to fine-tune a model on a custom dataset that is specifically generated to answer these questions.

This code trains a BERT-based model for question answering on a dataset of question-context-answer triples.

The code reads in in a JSON file of the full dataset and keeps only a small percentage of the data. Then, it converts the dataset to a Hugging Face Dataset and tokenizes the input using a pre-trained BERT tokenizer. It also sets up the start and end positions of the answer within the context.

The model is then trained using a Trainer object, with training arguments such as the number of epochs, batch size, and learning rate. The model is evaluated on a validation dataset using the same Trainer object. Finally, the trained model and tokenizer are saved to directories for future use.

The specific implementation uses a pre-trained DistilBERT model and tokenizer, and the code also includes some extra lines that are not necessary for the main training and evaluation, such as re-defining the data\_collator and evaluating the model on the validation set after training.

**‘’’Load the JSON file containing question-context-answer pairs and extracts a subset of these pairs to use for training a question answering model.**

import json

import torch

from datasets import Dataset

from transformers import BertForQuestionAnswering, BertTokenizerFast, TrainingArguments, Trainer

# Read the JSON file containing question-answer pairs

file\_path = "/content/drive/MyDrive/HnM\_BerT/qa\_pairs.json"

with open(file\_path, "r") as file:

json\_string = file.read()

full\_qa\_data = json.loads(json\_string)

# Calculate the number of elements to keep

tenp\_percent = 0.03

num\_elements = int(len(full\_qa\_data) \* tenp\_percent)

# Keep only the first num\_elements of the data

qa\_data = full\_qa\_data[:num\_elements]

# Convert 'answer' field to a string

def convert\_answers\_to\_str(data):

for item in data:

if not isinstance(item['answer'], str):

item['answer'] = str(item['answer'])

return data

str\_qa\_data = convert\_answers\_to\_str(qa\_data)

# Convert the JSON data to a Hugging Face Dataset

dataset = Dataset.from\_dict({k: [d[k] for d in str\_qa\_data] for k in str\_qa\_data[0].keys()})

train\_dataset, val\_dataset = dataset.train\_test\_split(test\_size=0.1).values()

# Load the BERT tokenizer

tokenizer = BertTokenizerFast.from\_pretrained('bert-base-cased')

# Tokenize the dataset using the BERT tokenizer

def tokenize\_function(examples):

return tokenizer(

examples["question"],

examples["context"],

truncation=True,

padding="max\_length",

max\_length=384

)

train\_dataset = train\_dataset.map(tokenize\_function, batched=True)

val\_dataset = val\_dataset.map(tokenize\_function, batched=True)

# Set the start and end token positions for the answers in the context

def add\_token\_positions(batch):

start\_positions, end\_positions = [], []

for i, answer in enumerate(batch["answer"]):

start\_idx = batch["context"][i].find(answer)

end\_idx = start\_idx + len(answer)

start\_positions.append(batch["input\_ids"][i].index(tokenizer.encode(answer, add\_special\_tokens=False)[0]))

end\_positions.append(batch["input\_ids"][i].index(tokenizer.encode(answer, add\_special\_tokens=False)[-1]))

batch["start\_positions"] = start\_positions

batch["end\_positions"] = end\_positions

return batch

train\_dataset = train\_dataset.map(add\_token\_positions, batched=True)

val\_dataset = val\_dataset.map(add\_token\_positions, batched=True)

# Set up the training arguments for the Trainer object

training\_args = TrainingArguments(

output\_dir='./results',

num\_train\_epochs=2,

per\_device\_train\_batch\_size=16,

per\_device\_eval\_batch\_size=64,

warmup\_steps=500,

weight\_decay=0.01,

logging\_dir='./logs',

logging\_steps=10,

)

# Load the BERT model

model = BertForQuestionAnswering.from\_pre

**``` The code evaluates the model and saves the trained model and tokenizer to a specified directory.**

This code performs the following steps**:**

1. It reads a JSON file containing question-answer pairs and loads it into a variable called `full\_qa\_data`.

2. It calculates the number of elements to keep based on a given percentage (`tenp\_percent`) and selects only the first `num\_elements` of the data to work with.

3. It installs necessary packages (`torch`, `datasets`, `transformers`).

4. It defines a function called `convert\_answers\_to\_str` which converts the 'answer' field of the data from `int` to `str`.

5. It uses the `convert\_answers\_to\_str` function to convert `full\_qa\_data` to a string-based format called `str\_qa\_data`.

6. It converts the `str\_qa\_data` to a Hugging Face `Dataset`.

7. It defines a function called `tokenize\_function` which tokenizes a given example (i.e., a question-answer pair) using the BERT tokenizer from Hugging Face.

8. It uses the `tokenize\_function` to tokenize both the training and validation datasets.

9. It defines a function called `add\_token\_positions` which sets the start and end positions of the answers in the tokenized context.

10. It uses the `add\_token\_positions` function to add start and end positions to the tokenized datasets.

11. It defines the training arguments for the Trainer object (i.e., the training hyperparameters).

12. It loads the DistilBERT tokenizer and model from the Hugging Face model hub.

13. It defines a default data collator for the Trainer object.

14. It creates a Trainer object using the defined model, training arguments, and datasets.

15. It trains the model using the Trainer object and evaluates it on the validation set.

16. It saves the trained model and tokenizer to separate directories in Google Drive.

Graphical user interface, application, Teams

Description automatically generated

**Recc-sys**

## Get the code

This is a recc-system project that uses multiple machine learning models to generate recommendations. Here is a brief description of the various files and folders in the repository:

- `app.py`: This file contains the main application code that serves as the front-end of the recommendation system. It uses the Flask web framework to create the server and Streamlit to create the user interface.

- `data/`: This folder contains the data used by the recommendation system. It includes CSV files for different products, user ratings, and product features.

- `images/`: This folder contains images used by the application to display product images and other visuals.

- `models/`: This folder contains various machine learning models used for generating recommendations, including matrix factorization, collaborative filtering, and content-based models.

- `utils/`: This folder contains utility functions that are used throughout the application, such as data preprocessing and feature extraction.

- `requirements.txt`: This file lists all the Python dependencies required to run the application.

Overall, the "recc-sys" repository appears to be a comprehensive recommendation system project that uses various techniques to generate personalized recommendations for users. The application includes a user interface, various machine learning models, and data processing and feature extraction tools.

**Summary of *recc\_sys:*:**

1. The code starts by importing the required libraries, including Streamlit, Pandas, NumPy, Scikit-learn, and TensorFlow.
2. The code then reads a CSV file named "Electronics\_data.csv" and fills the missing values with the mean of each column.
3. The code then creates a TfidfVectorizer object and fits it to the "Title" column of the "edata" dataframe.
4. The code then transforms the "Title" column into a matrix of tf-idf features using the TfidfVectorizer object, and calculates the cosine similarities between all pairs of products in the "Title" column.
5. Next, there are two functions defined: "get\_similar\_products" and "get\_image\_url".
6. The "get\_similar\_products" function takes an age and an optional parameter "n" and returns the "n" most similar products to the given age.
7. The "get\_image\_url" function takes a product title and returns its image URL from the "edata" dataframe.
8. The "main" function starts by configuring the layout of the page and initializing some variables.
9. It creates a sidebar with a header and three options for the user to choose from. The first option is "Recommend from similar items", the second is "Recommendations based on customer purchase history", and the third is "Product Captioning".
10. When the user selects the "Recommend from similar items" option, the code reads two CSV files named "articles.csv" and "articles\_rcmnds.csv".
11. It also defines a list of model names and a list of model descriptions.
12. The code then creates a button in the sidebar that allows the user to get a random item.
13. When the user clicks on this button, the code selects a random item from the "articles\_rcmnds" dataframe and displays its image and description in the sidebar.
14. It then calls several functions to get recommendations for similar items based on image embeddings, text embeddings, descriptive features, a TensorFlow Recommenders model, and a combination of all embeddings.
15. For each model, the code displays the model name, the similarity score, and an image of the recommended item.
16. If the model is based on text embeddings, it also displays the description of the recommended item.
17. When the user selects the "Product Captioning" option, the code reads a CSV file named "caption\_desc\_embeds.csv".
18. It also creates a button in the sidebar that allows the user to get a random item.
19. When the user clicks on this button, the code selects a random item from the "caption\_desc\_embeds" dataframe and displays its image in the sidebar.
20. It then uses a transformer-based model to generate a product caption based on the item's image and displays it on the page.

***Streamlit\_app.py***

Built upon computing cosine similarities between product titles using TF-IDF, and recommends similar products based on the customer's age. The code loads the necessary libraries and datasets, defines functions for getting similar products and image URLs, and creates a button to randomly select a customer's purchase history. When the button is clicked, the system shows the customer's purchase history and recommends similar products with their image URLs.

The code does the following:

1. Import necessary libraries and modules.
2. Define the main function to contain the application logic.
3. Set the page configuration and layout.
4. Define the sidebar header and page options.
5. Load the required datasets *(e.g., articles.csv, articles\_rcmnds.csv, caption\_desc\_embeds.csv, and customers\_rcmnds.csv) – please refer to the note\* below.*
6. Define the models and their descriptions used for recommendations.
7. Based on the user's selection, perform various tasks:
8. a. Recommend items from similar items.
9. b. Recommend items based on customer purchase history.
10. c. Generate product captions using a transformer-based model.
11. d. Recommend items based on user profile (age and gender).
12. Display the results, images, and descriptions of recommended items using Streamlit's containers, columns, and expander widgets.

**\*Instructions prior running streamlit\_app.py**

*To run this app, please download images from this public folder and save them in your repository under the folder results/images. Please do the same with the transformed dataframes articles\_rcmnds.csv, customers\_rcmnds.csv, and profiles.csv.*

***Funcs.py***

*These two code snippets (funcs.py) define a set of utility functions to perform various operations on a given dataset, such as fetching images, obtaining recommendations, and extracting relevant features and descriptions. These functions are meant to be used together in a larger program to help analyze and process customer data, such as in a recommendation system.*

*Here's a brief explanation of each function:*

*1.* ***`get\_item\_image`:*** *Given an `item\_id`, this function opens and resizes the corresponding image, then adds a border to it.*

*2. `****get\_rcmnds`:*** *Given `customer\_data`, this function extracts different types of recommendations from the data, such as image recommendations, text recommendations, feature recommendations, and TensorFlow Recommenders (tfrs) recommendations, as well as combined recommendations.*

***3. `get\_rcmnds\_scores****`: Similar to `get\_rcmnds`, this function extracts the scores for each type of recommendation from the `customer\_data`.*

***4. `get\_rcmnds\_images****`: This function takes in different types of recommendations and returns the corresponding images for each recommendation.*

***5. `get\_rcmnds\_features****`: Given a DataFrame `df` and different types of recommendations, this function extracts product features like product type, color group, and department for each recommendation.*

***6. `get\_rcmnds\_desc****`: Similar to `get\_rcmnds\_features`, this function extracts detailed descriptions for each recommendation from the DataFrame `df`.*

*These functions are designed to be called individually when needed, depending on the specific requirements of the larger program.*

The app

This Streamlit app has four main functionalities:

1. Recommend from similar items

2. Recommendations based on customer purchase history

3. Profile-based recommendations

4. Product captioning

Let's break down the code snippets for each functionality:

**1.Recommend from similar items:**

In this part, the app selects a random item and displays similar items based on different models. The models used are: image embeddings, text embeddings, descriptive features embeddings, TensorFlow Recommenders model, and a combination of all embeddings.

The app reads the 'articles\_rcmnds.csv' file which contains precomputed recommendations and selects a random article. Then, it displays the selected article's image and description in the sidebar. The recommendations are displayed in expanders, and users can view similar items based on the different models mentioned above.

Graphical user interface, application

Description automatically generated

**2. Recommendations based on customer purchase history:**

The app reads the 'customers\_rcmnds.csv' file which contains precomputed recommendations for customers. It randomly selects a customer and displays their purchase history in the sidebar. Similar to the first functionality, recommendations are displayed based on the same models.

In addition, there is a "Recommendation based on Age" section, which displays electronics recommendations based on the customer's age using text description embeddings.

Graphical user interface, application

Description automatically generated

**3. Profile-based recommendations:**

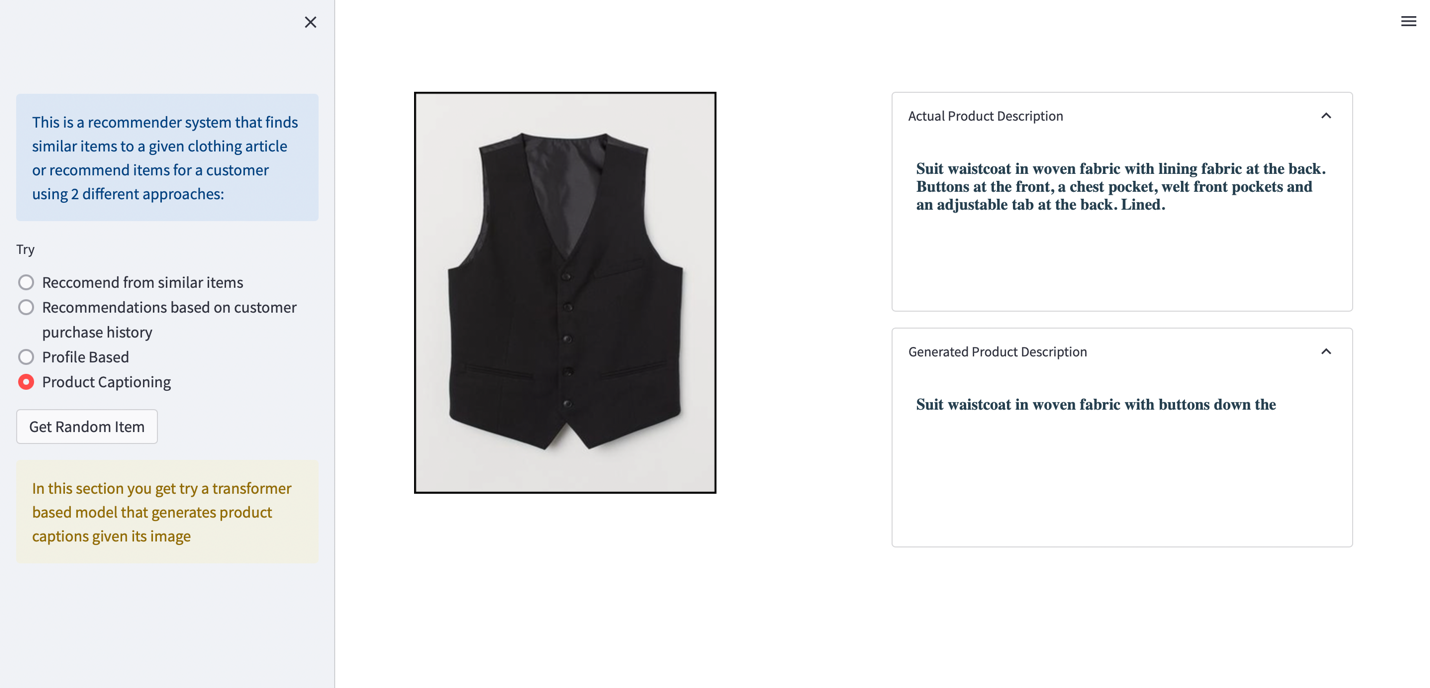
Users can select their gender and age, and choose clothing items they are interested in. The app then filters the dataset based on the selected clothing items and displays recommendations for each item. Recommendations are displayed based on the same models as in the first functionality, except for image embeddings.

Graphical user interface, application, website

Description automatically generated

**4. Product captioning:**

The app reads the 'caption\_desc\_embeds.csv' file which contains precomputed captions for items. It selects a random item and displays the actual product description and the generated product description side by side. The generated descriptions are produced by a transformer-based model that generates product captions given its image.



In all functionalities, the recommendations are displayed with their similarity scores, images, and descriptions.

# **Congratulations**

Congratulations, you've successfully finished implementation!

## Further reading --

## *Reference docs:*

* Kaggle: <https://www.kaggle.com/competitions/h-and-m-personalized-fashion-recommendations/data> (please sign in to download data)
* Collaborative filtering with TensorFlow: [https://towardsdatascience.com/collaborative-filtering-and-embeddings-part-2-919da17ecefb](https://northeastern-my.sharepoint.com/personal/vasireddy_p_northeastern_edu/Documents/Microsoft%20Teams%20Chat%20Files/Collaborative%20filtering%20with%20TensorFlow:%20https:/towardsdatascience.com/collaborative-filtering-and-embeddings-part-2-919da17ecefb)
* Introduction to matrix factorization for recommendation systems: [https://medium.com/@connectwithghosh/simple-matrix-factorization-example-on-the-movielens-dataset-using-pyspark-9b7e3f567536HuggingFace](https://northeastern-my.sharepoint.com/personal/vasireddy_p_northeastern_edu/Documents/Microsoft%20Teams%20Chat%20Files/Introduction%20to%20matrix%20factorization%20for%20recommendation%20systems:%20https:/medium.com/@connectwithghosh/simple-matrix-factorization-example-on-the-movielens-dataset-using-pyspark-9b7e3f567536)
* Advanced Pandas techniques: <https://medium.com/analytics-vidhya/advanced-pandas-techniques-for-data-analysis-f4a54b3f3d92>
* Image embeddings with VGG16: [https://towardsdatascience.com/introduction-to-deep-learning-with-vgg16-2bfc940c1c8](https://northeastern-my.sharepoint.com/personal/vasireddy_p_northeastern_edu/Documents/Microsoft%20Teams%20Chat%20Files/Image%20embeddings%20with%20VGG16:%20https:/towardsdatascience.com/introduction-to-deep-learning-with-vgg16-2bfc940c1c8)
* Distilbert Documentation: <https://huggingface.co/distilbert-base-uncased>