



UNIVERSITY OF
LEICESTER

**Department of Informatics
University of Leicester
CO7201 Individual Project**

Outfit.ai

Kavin Arasu

ka370@student.le.ac.uk

199048731

Project Supervisor: [Dr. Yi Hong]

Second Marker: [Dr. Shuihua Wang]

Word Count : 10128

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Kavın Arasu

Date: 20th May 2022

Signature: Kavın Arasu

Table of Contents

Introduction	5
Aim.....	5
Challenges	5
Background	6
Machine Learning	6
Computer Vision	7
Recommender Systems	9
Data Gathering and Pre-processing	10
Kaggle	10
Project Requirements	12
Detailed Requirements	12
Essential Requirements	12
Recommended Requirements	12
Optional Requirements	12
System Design	13
Architecture	13
Database Design	14
Implementation	16
Technical Details	16
Fashion Classifier Model	17
Recommender Model	21
Project Outcome.....	24
Project Structure.....	33
Build and run	34
Testing and Quality Assurance	35
Evaluation	38
Limitations and Future Work	41
Conclusion	42
References	43

Abstract

In today's world, time is money. Imagine yourselves looking at the mirror thinking what would look good on you today, walking towards the closet to find yourselves in a state of dilemma not knowing what to choose – this is where a virtual companion can help you. With the abundance of technology and data that is available right now, the idea is to develop a recommender system to help you choose outfits at the comfort of your fingertips.

In this report, we discuss the concepts that helped build this application. We also discuss on the approaches like Content-based and Collaborative filtering techniques, about VGG and ResNet models, the outcomes of the project, and metrics that were used to evaluate the application.

Introduction

Aim

This project aims to develop an application that will help users escape the dilemma of choosing what outfits to wear on different occasions. The motivation is that users are not always sure of what to wear where or spend too much time choosing an outfit or end up repeating the same outfits. Hence a recommender system that can suggest a set of outfits from their closet is the motive of the project. In this project, I will be building a web application that will allow users to upload photos of clothing items they own and a recommender system that will show recommendations of outfits from their closets.

Challenges

Every Machine Learning project demands the need for the right dataset for it to become a successful project. Getting the right dataset for this project was quite a challenging task, even though datasets such as DeepFashion [1] were available, the results were not so promising; mainly because they consisted of low-res images. In this project, I have used a Fashion dataset that I have obtained from Kaggle [2] – more about this dataset and data cleaning will be explained in Section 3 (Data Gathering and Pre-processing). Another challenging aspect of this project was training and inferring from the trained models on which the author had no prior expertise.

Background

To successfully build an application that is capable of analysing users' closets and recommending outfits, an understanding of Machine Learning (ML), and Computer Vision (CV) is deemed necessary to achieve the essential requirements outlined in the project. This section intends to give the reader a brief overview of ML and CV.

Machine Learning

Machine learning is a field of artificial intelligence which enables the system to be able to predict or decide without manually programming the system to do it. This is enabled by using 'training data' which is nothing but a set of sample data that is used to build and train a model. Machine learning algorithms have now started to take more importance in the field of speech recognition, medicine, computer vision, email filtering, automated vehicles etc. Now that more companies have started to adopt machine learning algorithms, the study of handling many data sets and the use of cloud computing has also directly increased its popularity.

Machine learning helps the system to perform tasks that are usually tough to do if it was programmed with thousands of lines of codes or done without the use of a computing device. The algorithm uses a large set of data which is then applied to build a model. This model will later use these data to train itself and be able to provide outcomes in a very paced manner which would usually take years and years to compute if it was done manually. This will also be able to tackle problems that are mostly repetitive in nature and machine learning algorithms will help the system handle these repetitive tasks without any manual interventions.

One such example where we can demonstrate the use of machine learning to help humans with simple tasks is in the field of drug discovery and development. This field has been known to involve a lot of studies on medical data and clinical outcomes and before the machine learning concepts were discovered, the scientist had to rely on manual computations or rather the use of the limited amount of supercomputers. This made the whole process of drug discovery and development very slow. Once the machine learning and artificial intelligence concepts were adopted by the medical industry, these tasks were assigned to these heavily trained models which help to perform target validation, analysis of digital pathology data in clinical trials and identification of prognostic biomarkers. The processing power of these models is very high which enabled the scientists to get relevant data for the drug discovery and development.

In the fashion industry, machine learning helps to build a prediction model like the cross-scenario retrieval model, interactive retrieval model, scenario-oriented recommendation model, explainable recommendation model etc. These models use datasets that involve preferences and rating data provided by the users. They can be also trained by experts in fashion technology with key in-depth knowledge of specific styling, clothing material, colour contrasting etc. At the end of the day, these models will help people at the business end to be able to use the potential predictive outcomes that the models can generate and use them wherever they intend to apply.

Machine learning uses concepts like statistics, linear algebra, calculus, and probability. The two main objectives of using machine learning in the modern world are to classify data based on the models and to make predictions for future outcomes based on these models.

The major aspect of machine learning is to train the models and the approaches to do it are broadly categorised into three main parts with the help of the nature of “feedback” and “signal” available to the learning system.

- Supervised Learning: A “teacher” is employed who provides example data of inputs and the relevant outputs to the system. The main goal of the system is to adapt to these data and learn the rule of mapping inputs to outputs.
- Unsupervised Learning: No label data is provided to the systems and allows the system to figure out on its own to develop the structure. The goal of such a learning system is to work in an unsupervised manner and discover a way to tackle the future processing of inputs to deliver the corresponding outputs.
- Reinforcement learning: In this type of learning the system is made to work in a dynamic environment and is asked to perform a certain goal, for example, the system is made to play a game and the goal is to defeat the opponent. As the system progresses through the problem space, the program will be able to input feedback which will reciprocate rewards for the system. The system will try to maximize these rewards and in the course of it, the system will try to learn all possible ways of attaining the goal.

Computer Vision

Computer Vision helps a computer extract information about data in images, videos, etc... Artificial Intelligence helps computers to think, whereas Computer Vision enables them to see, observe and understand. In terms of engineering, computer vision can be termed as a process of understanding and automating tasks which the normal human visual system tries to perform. The set of tasks that the computer vision engines perform are acquiring, processing and analysing images. During this, the models try to extract high-dimensional data from the real world which will be used to generate symbolic or numerical information. The models will also try to deliver descriptive data for the images according to the real world which will be then used to analyse the given imagery. This is done with the help of models that are designed with the help of geometry, statistics, physics and learning theories. There are several subdomains of computer vision, which include object detection, scene reconstruction, video tracking, event detection, 3D pose estimation, object recognition, indexing, visual serving, image restoration, 3D modelling and motion estimation. Computer Vision is now applied to a variety of fields given the improved processing and handling of data through technologies like cloud, 5G networks etc. Predominantly Computer Vision is applied to fields like medicine, machine vision, military, tactical feedback, and autonomous vehicles.

The tasks that the computer vision tends to perform are discussed in detail below,

- **Recognition:** One of the wide uses of computer vision is to perform tasks that will help recognize a particular object, feature or activity from a given image or video file. This is also considered to be a classical problem of computer vision due to the lack of training data available to the system. Three main recognition problems are tackled by the computer vision, they are (1) Object recognition, which is done with the help of learned objects obtained from 2D positions in an image or the 3D poses from a scene, and (2) Identification, which is a process of determining an individual instance of an object in the given specimen data and (3) Detection, which is the process of scanning the given data for a specific condition. Computer vision also performs certain specialised tasks based on recognition like content-based image retrieval, pose estimation, optical character recognition, 2D code reading, facial recognition, and shape recognition.
- **Motion Analysis:** In this type of task the computer vision tries to capture the motion-related estimations like velocity, which is processed from various points of an image or a 3D scene. The tasks that are performed during the motion analysis are, (1) Ego motion which is the process of obtaining the rotation and translation of a camera that was used to produce the image, and (2) Tracking is the process of following a specific content of an image or a video and trying to follow its movement in the given data and (3) Optical flow which determines the movement of a point in the image relative to the image plane.
- **Scene reconstruction:** This process allows the computer vision to develop a 3D model of a scene with the help of one or more images of a scene, a video or any other data related to the scene. In simple terms, computer vision will be enabled to produce 3D points of a scene which can be later used to analyse or deconstruct specific data of the scene. These 3D points can also be used to create a 3D model of a whole scene with the help of algorithms that can stitch the produced 3D points. This will help engineers to visualise and produce changes to the smaller version of the original scene to check if the changes that were incorporated would produce any significant outcomes as desired.
- **Image restoration:** This specific task is widely adopted by archaeologists who attempt to uncover ancient secrets through gathered historical data. These data include images which are mostly damaged or tampered with due to their age and require restoration to be studied. Computer vision helps to analyse such images with limited information and tries to restore them through processes that involve reducing or removing image noises. This is done with the help of low-pass filters and median filters.

The functions that the computer vision engines carry out are, image acquisition, pre-processing, feature extraction, detection & segmentation, high-level processing, and decision making. Computer Vision needs lots of data to work, once fed with enough data, it learns to find patterns in the data to recognize images.

We can train a Computer Vision model using either of the two technologies –

1. Deep Learning
2. Convolutional Neural Networks (CNN)

Deep learning – Deep Learning is an extension of neural networks. A neural network that is stacked several layers deep is said to be a Deep Learning model. A deep learning model tries to mimic how our human brains work, that is by learning very simple patterns in the initial layers to learning complex concepts and patterns. The process that is followed in the deep learning mechanism is that in each layer of learning the algorithm tries to transform the given data into a more abstract and composite version. For example, in image processing for facial recognition consisting of a matrix of pixels, the first layer may concentrate on encoding the edges and abstracting the pixels, the next layer will compose the arrangements of the edges, and the third layer may identify a feature of a human-like the nose or the eyes, the fourth layer will finally abstract a face from the image. The level-by-level process of analysing and representing the data helps deep learning to deliver the advanced outputs.

A convolution neural network is a specific type of artificial neural network which adopts mathematical operations termed convolution instead of the matrix multiplication in either of the layers. The main use of this neural network is to compute and process the pixel data of an image or a video. It is mainly used in image recognition and processing for detailed processing of the image to identify an object based on its pixel data. The neural network is built on three main layers, the input, hidden and output layers. The hidden layer consists of the layers that are responsible to perform the convolution. It performs a dot product of the convolution kernel with the help of the layer's input matrix. These are then followed by other types of layers such as the pooling layer, fully connected layer, and the normalisation layer each performing a specific set of processing tasks which will later be used to produce the required data for the intended processing.

Recommender Systems

Recommender Systems are aimed at suggesting relevant items to the user. The relevance could be inferred from either the user or from the item itself. The two main paradigms of Recommender Systems are collaborative and content-based methods. In the Collaborative filtering method, as the name states, the system would try to find if a user would like/dislike the item based on the reaction the system received from a similar user. However, in the Content-based methods, the system would find relevant items to the user only based on the attributes of the item.

Feedback can be obtained from users using explicit feedback or implicit feedback, where explicit feedback can be captured if they either like, dislike or report, etc... Whereas, implicit feedback is captured when the user clicks on an item, etc... This feedback can be used to improvise the recommendations a system provides to its users, and the simplest way of achieving it would be to add or subtract a score based on the feedback obtained.

Data Gathering and Pre-processing

Kaggle

As mentioned above, the Fashion dataset [3] was obtained from Kaggle. The purpose of choosing this dataset versus the others is because this dataset had high-resolution images with good annotations for each image. The original dataset consisted of an images folder and multiple CSV files. The CSV file of interest to us is the 'articles.csv' file which contained information about each image (Referred to as an article in the dataset) such as the *Product Code*, *Product Group*, *Product Type*, *Graphical Appearance*, *Colour*, *Perceived Colour*, *Section*, *Detailed Description*, etc...

The dataset went into some pre-processing steps before it reached its final form, and they were –

1. Removing unwanted columns such as *product_code*, *prod_name*, *product_type_no*, *product_type_name*, *graphical_appearance_no*, *colour_group_code*, *perceived_colour_value_id*, *perceived_colour_value_name*, *perceived_colour_master_id*, *perceived_colour_master_name*, *department_no*, *department_name*, *index_code*, *index_name*, *index_group_no*, *index_group_name*, *section_no*, *section_name*, *garment_group_no*, *garment_group_name*.
2. Removing rows in which the *article_id* does not have a corresponding image in the images folder.
3. The dataset consisted of about 105,542 articles with 131 product types. For our project, we are only considering 14 product types, as shown below in figure 1. After removing rows in which the *product_type_name* is not one of those 14 product types, we end up with a dataset consisting of about 62,621 articles.

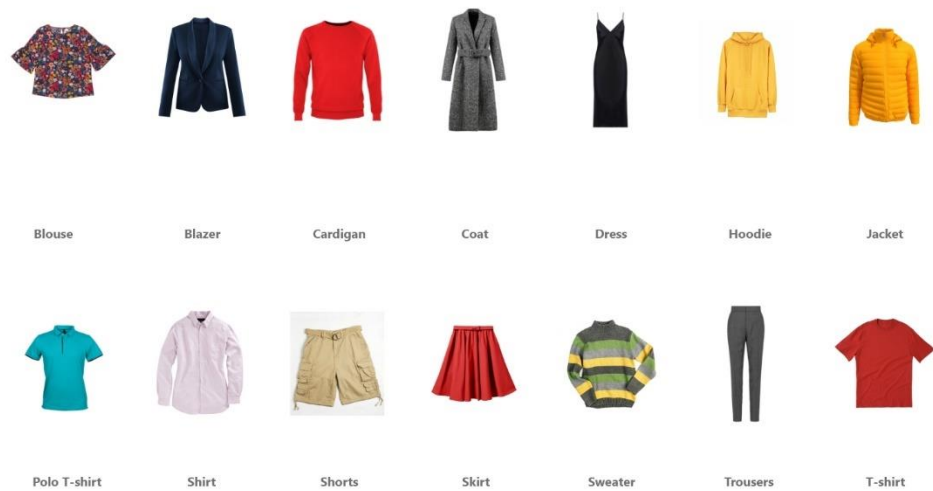


Figure 1 – The 14 product types

- The column *detail_desc* gives a brief description of the article. To process this description to retrieve meaningful tags, we first start by removing stop words from the text, then extracting nouns and adjectives and only keeping the top 25 most occurring tags across the dataset while keeping only the top 10 most important tags for each article. Let us look at an example below -

Before processing - 'Short dress in airy, patterned chiffon with a flounce-trimmed V-neck and long sleeves with elasticated cuffs. Smocked seam at the waist and a skirt with different lengths of tiered flounces. Lined.'

After processing - ['waist', 'sleeves', 'cuffs', 'seam', 'skirt', 'elasticated', 'long', 'short']

Project Requirements

Detailed Requirements

Essential Requirements

- Develop a Fashion web application with features like sign in, sign out, outfit recommender, etc.
- To train an event detection model to predict the social event from an image.
- To train a clothing detection model to segment and predict what type of clothes are worn in an image.
- To train a model to learn the correlation between the type of the event and the type of clothes worn during each event.
- To use a pre-trained model to categorize the pieces of clothing that the user uploads which will be useful in recommending the outfit. This is done by learning a local appearance model of all the items that the user uploads and then using the nearest neighbour approach to recommend the outfit.

Recommended Requirements

- To integrate a chatbot for the user to converse with the system to communicate further preferences (such as the colours they like/dislike to wear to get a better recommendation suited for them)
- To avoid recommending repetitive outfits for the same type of occasion.
- User feedback mechanism to tell the system if the recommendation was appropriate or not.

Optional Requirements

- Recommending matching accessories for the outfit (complete-the-look).
- Recommending missing items in their wardrobe to complete the look (shop-the-look).
- To show how the user would look in the recommended outfit in a 2D model

System Design Architecture

The architecture for our project follows a Model-View-Controller pattern because it was developed using the Flask framework which inherits the MVC pattern in its structuring. Here, we will discuss the architecture in detail with a design diagram for reference. The structuring of our Flask application will be discussed in the later sections of the report. A Model-View-Controller (MVC) pattern turns a complex application development into a much more manageable process. A *model*, in this context, is what we would call the backend of an application, which handles where and how the data should be stored. A *view* is what we would call the front end of an application, which is what the user sees. A *controller* is the brain of our application, where all the logical processing is done. The reader is advised to read the section *Project Structure* before proceeding with the rest of the content here to understand the following text better.

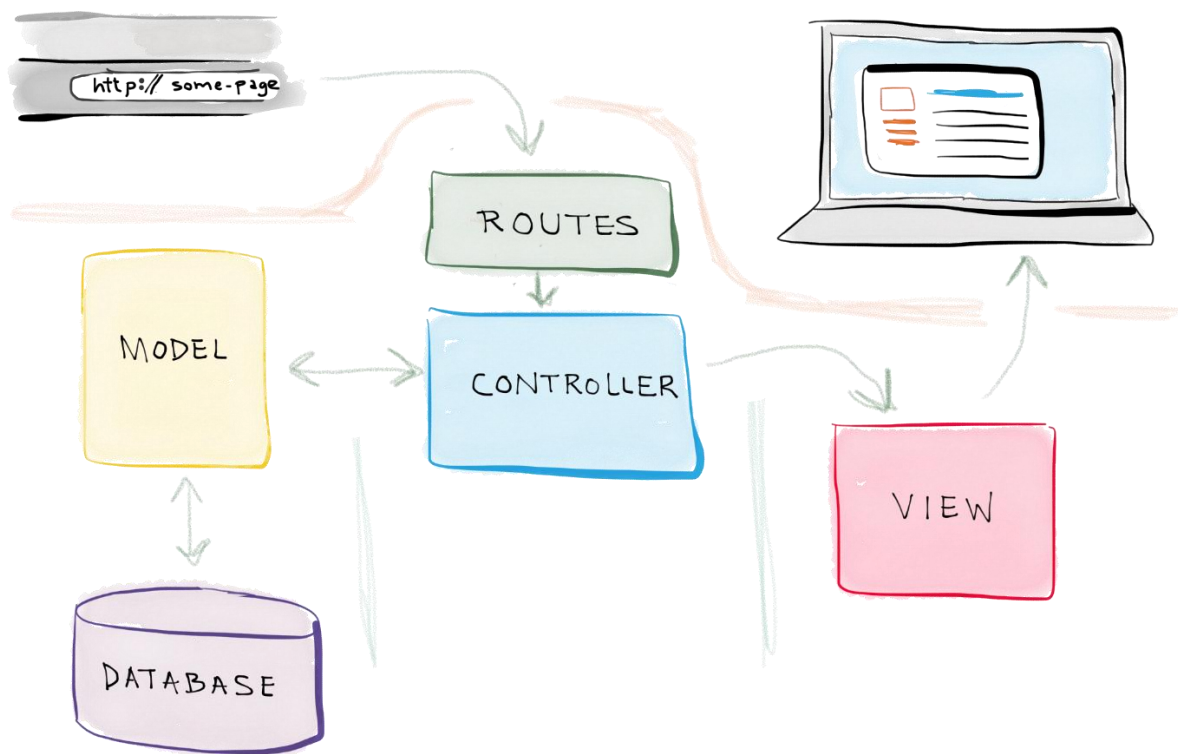


Figure 2 – Model-View-Controller

In our application, the `models.py` file which contains the schema of the database such as the tables, the datatypes of the columns and the constraints on these columns can be called the Model of our application. The `views.py` file can be called the Controller of our application as it contains the code for logical processing, however, the name '`views.py`' can be confusing as it is a Controller rather than a View, but it has been named '`views.py`' because it returns a view, just like `models.py` which returns a database model. The templates and the static folder are the Views of our application as they contain the HTML, CSS, Jinja and images that are required to

be rendered on the front end of the application.

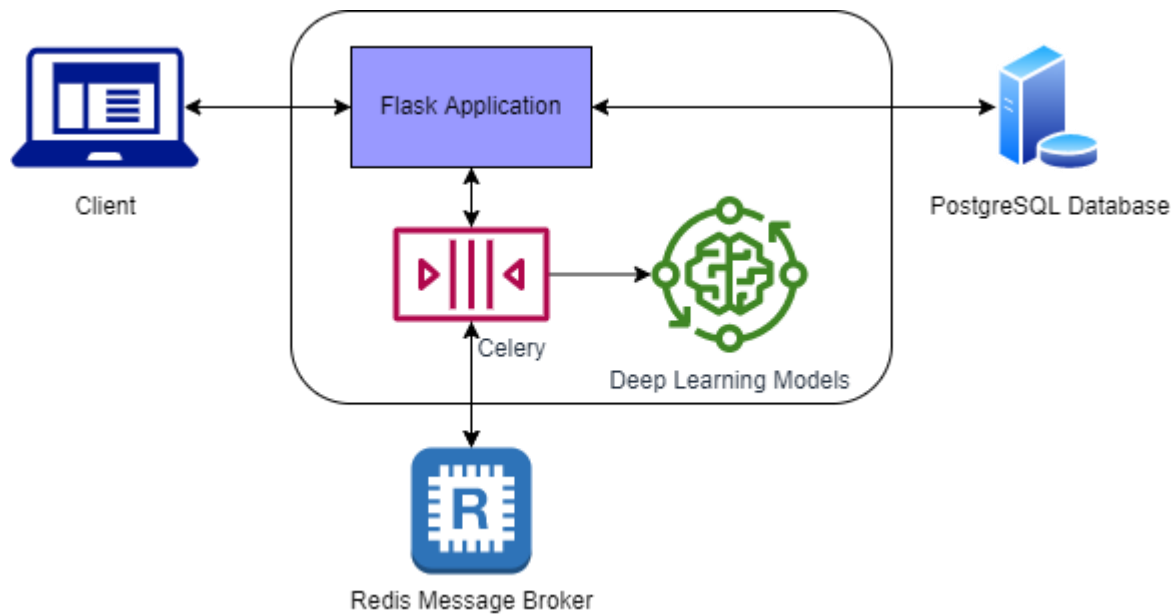


Figure 3 – System Architecture

Database Design

Our application being a data-dependent application requires most of the data to be persisted, and to persist the data, we use a database to query and store the information. For this project, the author felt that PostgreSQL was a good fit with our architecture and framework, the reason being that Flask has a good Object-Relational Mapping (ORM) support with PostgreSQL and with good documentation. In this section, we will discuss the tables that exist inside the database, the mapping between those tables, the datatypes of the columns and the constraints applied on few of the columns.

Our application allows users to register and log in to the application. To store the details of the User and to authenticate them into the application, we have a table 'Users' that stores basic details in the columns – id, Username, Email, Password, ProfilePhoto, FirstName, LastName, Gender.

Our application allows users to upload images of their clothing items into our system. To store the details of the item being stored associated with each user, we have a table 'UsersCloset' which has a many-to-one relationship with the 'Users' table. This table stores details in the columns – id, ImageFile, PredictedCategory, Embeddings, UserId.

Our application allows users to submit their preferences for the clothing items they would wear. To store the details of the preferences associated with each user, we

have a table 'UsersPreferences' which has a one-to-one relationship with the 'Users' table. This table stored details in the columns – id, choice, UserId.

Our application stores the recommendation details each user got for each run. This table called the 'UsersRecommendations' has a many-to-one relationship with the 'Users' table and stores details in the columns – id, img_index, UserId.

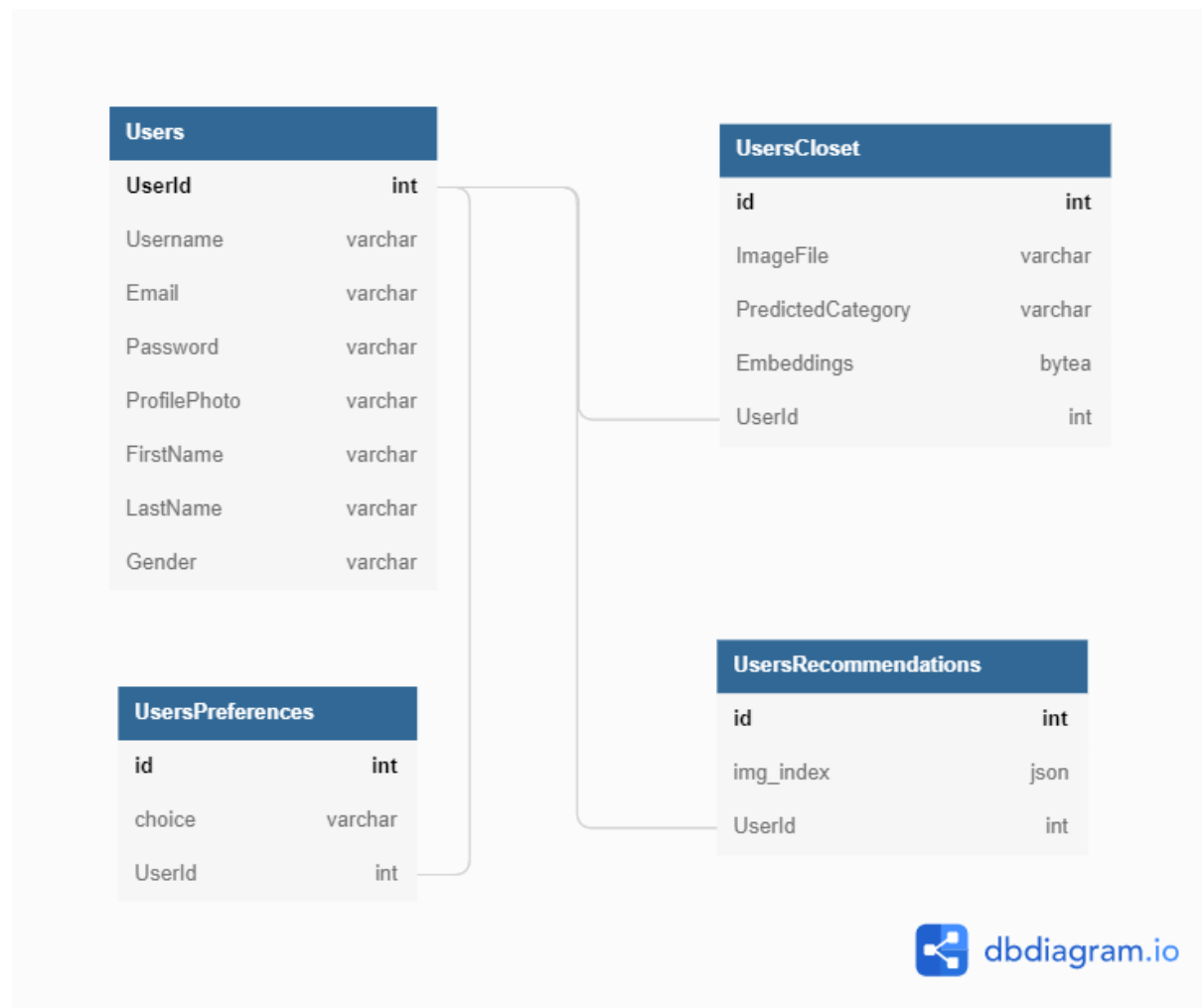


Figure 4 – Database Diagram

Implementation

Technical Details

This project involved a careful design of the application stack, wherein a lot of thought processes went in before choosing the framework, libraries, DB, etc... In the previous section, we saw an overview of the components that were used, here we will discuss in detail the technical stack –

Type	Name	Version	Summary
Programming Language	Python	3.7	This programming language is used to code the application and build the ML models.
	HTML	5.0	This markup language is used to create the UI for web pages.
	JavaScript	ES6	The language used to make the web pages interactive.
Framework	Flask	2.0.3	A micro-framework used to quickly build web applications
	PyTorch		A Deep Learning framework to build Machine Learning models.
Operating System	Windows	11	The operating system on which the application was developed.
	Ubuntu on Windows	18.04.5	The required operating system to run Redis, and hence the need for a Windows Subsystem for Linux.
Database	PostgreSQL	14.2	The database to store and query information.
IDE	Jupyter notebook	6.4.5	A web-based interactive computing platform, used for developing models and checking intermediate outputs.
	Google Colab		A web-based online interactive computing platform which is used for training purposes, due to the availability of GPUs
Other technology and software used	Redis	4.2.2	A message broker service for the task scheduling process for asynchronous processing.

Table 1 - Technical stack of the project

Fashion Classifier Model

Dataset Walkthrough

The dataset that we will be using to build the Fashion Classifier Model is the data that we got from Kaggle as mentioned in section 3. To build the fashion classifier model, we will only be using the *image* as a feature, and *product_type_name* as the target. The structure of our directory is as follows, as seen in the figure below.



Figure 5 – Directory structure of the dataset

The images are of variable widths and heights, and we do not resize them specifically in the pre-processing step because we will be applying a transform function on our dataset which we will discuss in detail in the model building section. To evaluate the performance of our model, we will use a Train-Valid-Test split [4] – and to split the folders into proportions of 80,10,10 we use a python library called ‘*split-folders*’ [5] which does this for us.

The images are of variable widths and heights, and we do not resize them specifically in the pre-processing step because we will be applying a transform function on our dataset which we will discuss in detail in the model building section.

Model Building

The deep learning framework we will be using to build the model is PyTorch [6]. PyTorch is an open-source Machine Learning framework used in building neural networks with support for GPU to do faster tensor computations. The author chose PyTorch because of their experience with the same versus TensorFlow which they weren’t that comfortable with.

Before training the classifier, we apply a few transformations [7] to our dataset to make the model generalize better. The transformations we used on this dataset are – Rotation, Resized Crop, Horizontal Flip, and Normalization. We normalized the dataset using the mean [0.485, 0.456, 0.406] and using the standard deviation [0.229, 0.224, 0.225] because we will be using a pre-trained network that was trained on the ImageNet dataset, and the mean and standard deviation was calculated on the ImageNet dataset.

To build our model, we will employ a technique called Transfer Learning [8] in which we will take an existing model and then use our dataset to re-train the model specific to the task at hand. The intuition behind Transfer Learning is that if a model has

been trained on a large dataset from scratch, that model would have learned the feature maps and we could use those models without having to start from scratch. There are many pre-trained networks available such as – VGG, ResNet, DenseNet, EfficientNet, MobileNet, etc... To evaluate which model performs the best on this task of classifying the clothing item amongst the 14 categories, we tried a few pre-trained networks. The basic idea is that we freeze the pre-trained model parameters to avoid backpropagating through them and build a custom classifier and then start the training process.

1. Resnet-18

Resnet-18 is a convolutional neural network with residual connections that is 18 layers deep, hence the name. The author used a pre-trained model of the Resnet-18 model which was made available on *torchvision* [9]. This pre-trained model was trained on ImageNet [10] which is a large dataset of about 14 million images of about more than 20,000 categories. The author then applied the concept of Transfer Learning to train the model on the dataset that was prepared above. The model was trained for 10 epochs and was stopped after the 10th iteration because the loss wasn't decreasing significantly. This process is called *Early Stopping* [11] and is done to avoid Overfitting [12]. Overfitting causes the model to perform well on the training set but fails to perform as well on the test set. This model achieved an accuracy of about 66% on the test dataset. The figure below shows the training performance of Resnet-18 and how the loss decreases for each iteration and how it flattens out at the end of the 10th iteration.

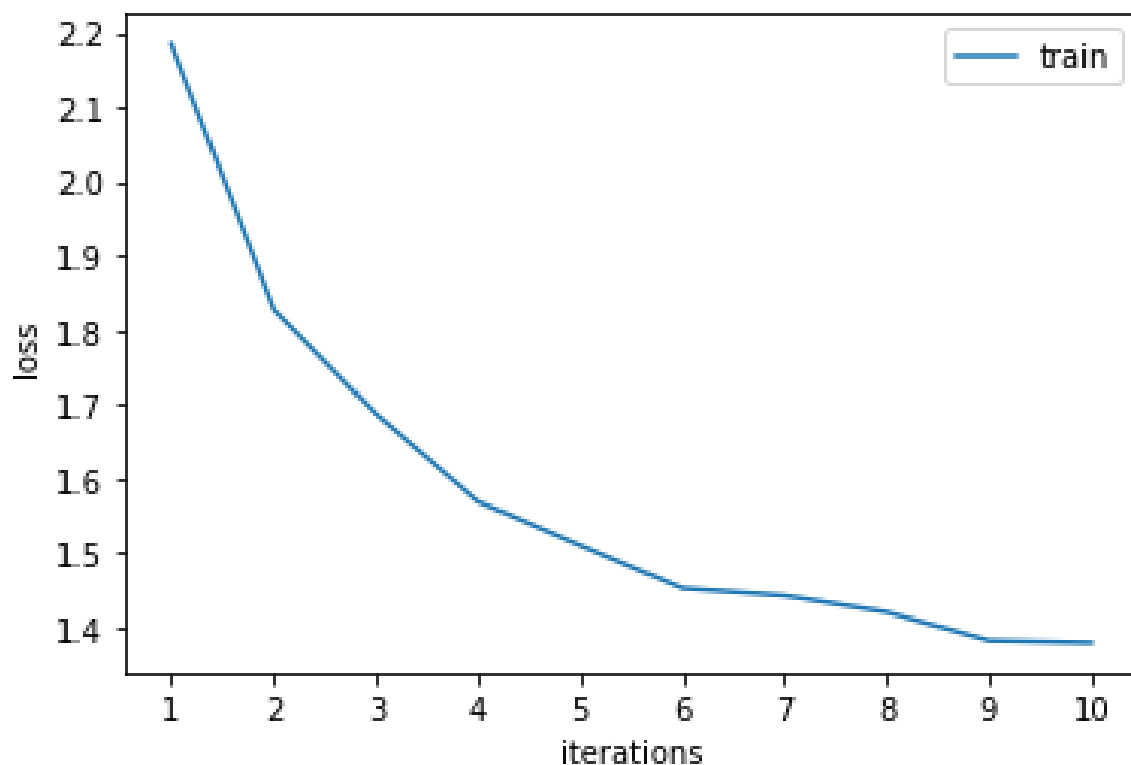


Figure 6 - Training performance of Resnet-18

2. Resnet-50

Resnet-50 is a convolutional neural network with residual connections that is 50 layers deep, hence the name. The author tried a deeper neural network this time to see how well the accuracy would improve. The model was trained on the same dataset without any changes in the data pre-processing and was allowed to run for 18 epochs. The model achieved an accuracy score of 67% on the test dataset. This meant that the Resnet architecture wasn't the best fit for this type of problem – one of the possible reasons could be that most of the categories looked like in a lot of ways. For example, a Polo shirt and T-shirt look alike, the main differentiation there would be the material used. The figure below shows the training performance of Resnet-50. As seen in the graph, the model has some spikes and one of the reasons could be that we are using a log-likelihood loss known as the Negative log-likelihood loss function due to which it might evaluate to $\log(0)$ causing exploding gradients, and hence causing bad predictions in the dataset. However, the model was stopped after 18 epochs because of the fear of overfitting the dataset. Resnet-50 model has far fewer model parameters compared to other models and has faster inference times. However, this model could have been chosen if size and speed were of concern, but the author was more focussed on accuracy (spoiler alert!) and thus chose the VGG-16 model.

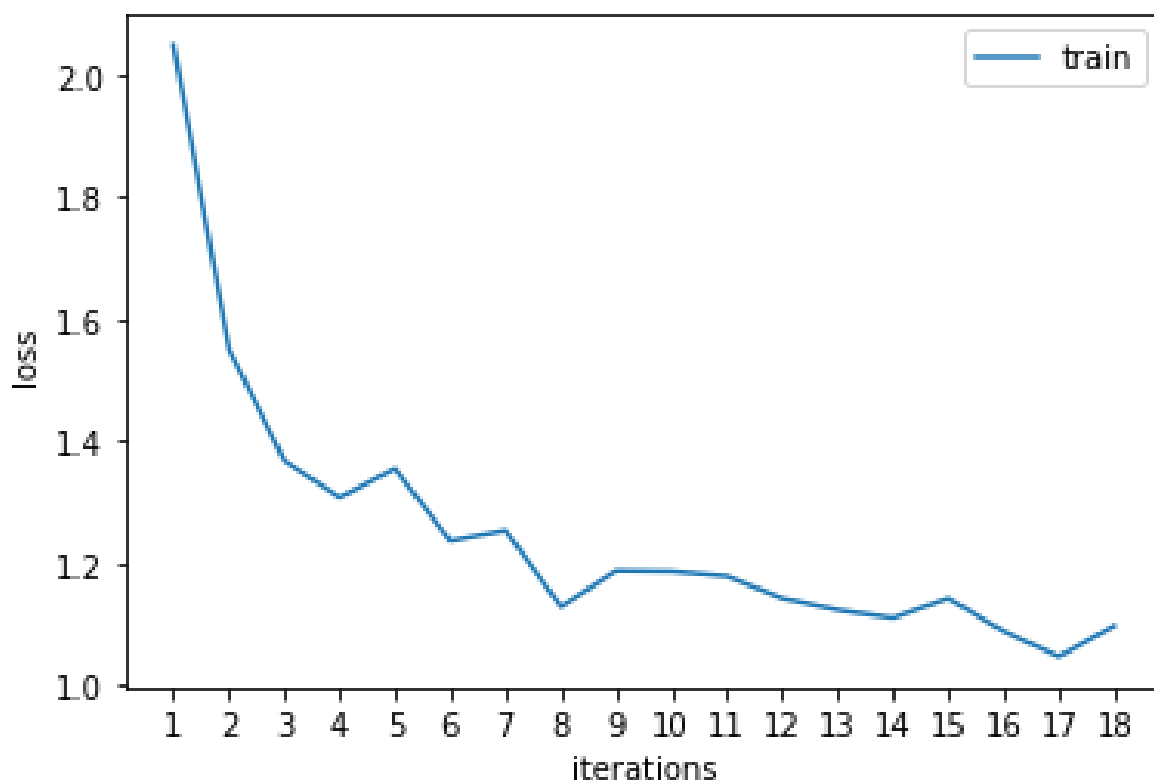


Figure 7 - Training performance of Resnet-50

3. VGG-16

VGG-16 is also a convolutional neural network with an architecture of 16 layers deep. This time the author tried the task of fashion item classification using the VGG-16 model. The model was trained on the same dataset and was also allowed to run for 10 epochs and was stopped as the loss wasn't decreasing. The model achieved an accuracy score of around 76% and is the best performing model among the rest. The figure below shows the training performance graph of VGG-16 and as seen in the graph, a detail to notice is how the loss has significantly decreased that too in just 1 iteration – this instantly shows how good this model might perform in comparison with others even without validating this model.

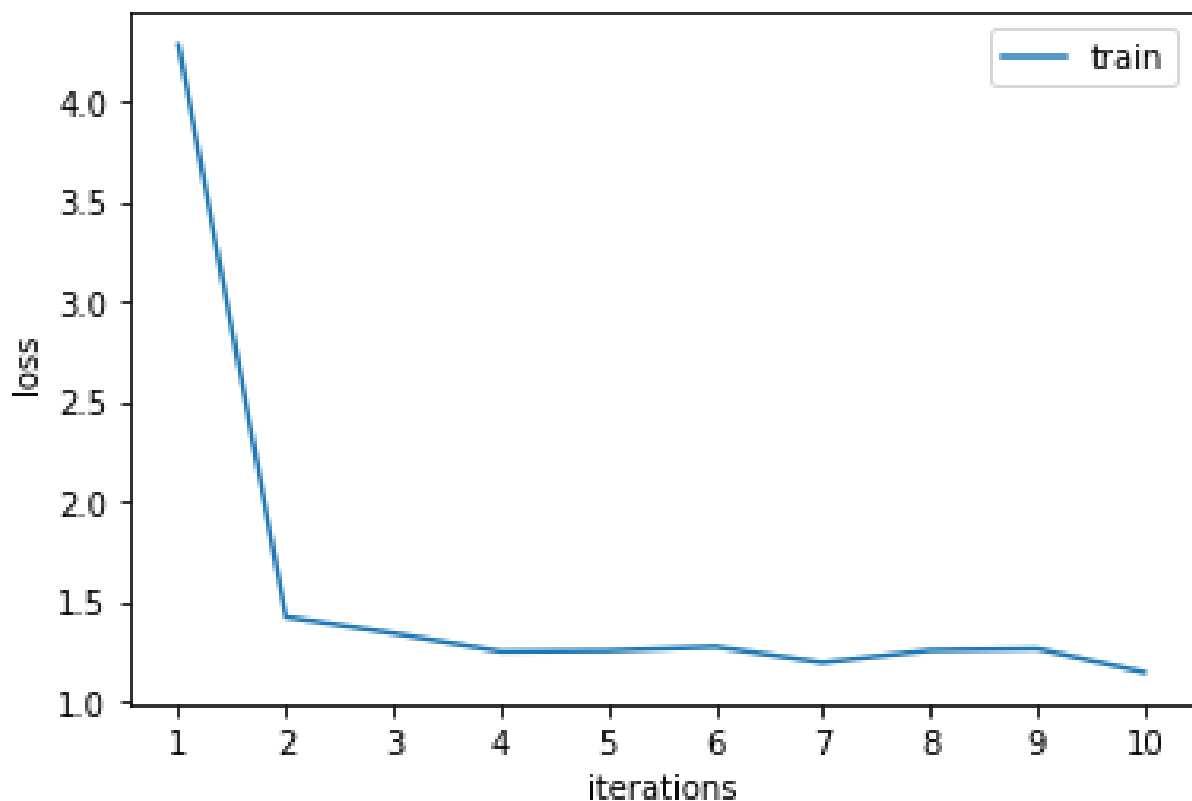


Figure 8 - Training performance of Resnet-50

Model Integration

This section discusses how the author integrated the model with the Flask application. For the model to be available during the time of the inference, the model needs to be saved as a checkpoint – this means that the model is serialized using the pickling process [13]. The pickled object is then loaded and can be used for inference. The model was trained on Google Colab [14] using a GPU on CUDA [15] for faster training purposes, however, for this application to work irrespective of GPU on any system, the tensors, and the model must be converted to *CPU* for it to be compatible. PyTorch provides an API to save and load the checkpoints and specify which device (CUDA or CPU) to use for model inference.

The purpose of this model is to categorise users' closet items into appropriate categories to present themselves as a digital closet for the user. For instance, the user can use the interface to upload the clothing items they own, and upon submitting, this model is triggered in the background asynchronously using *Celery* (as discussed in the above section) to categorise each item and is made available for the users to see their items in the application. For optimisation purposes, the model is only loaded once, and at the first time which takes around *5 seconds*, for every image uploaded after the model loading process, the inference time is around *0.1 seconds*. Whenever image(s) were uploaded using the interface, this image is run through the model to get the predictions and is then saved to the database.

Recommender Model

Dataset Walkthrough

The dataset used to build the Recommender Model is the dataset described in Section 3. Here, the task at hand was to train the model using *images* as features and *product_group_name*, *product_type_name*, *graphical_appearance*, *detail_desc* as targets, and *article_id* to link the row in the CSV file with the image filename. The column *detail_desc* was pre-processed as mentioned in Section 3 to output the corresponding relevant *tags*. Table 2 shows a sample representation of the dataset (with *article_id* excluded).

product_group_name	graphical_appearance_name	Product_type_name	Tags
Garment Upper body	Stripe	Shirt	['hem', 'waist', 'sleeves', 'cuffs', 'shoulders', 'adjustable', 'long', 'narrow', 'rounded']
Garment Full body	Glittering/Metallic	Dress	['sleeves', 'jersey', 'top']
Garment Lower body	Solid	Trousers	['waist', 'jersey', 'legs', 'cotton', 'elasticated', 'soft', 'ribbed']

Table 2 - Sample representation of the dataset

The column *product_group_name* consists of 3 categories – Garment Upper body, Garment Full body, and Garment Lower body. The column *graphical_appearance_name* consists of 29 categories – All over pattern, Melange, Solid, Stripe, Denim, etc... The column *product_type_name* consists of 14 categories as shown in Figure 1. Other columns such as *Colour* were disregarded from the training because the author felt the need for the recommender system to be diverse while recommending outfits.

Model Building

The model was trained on this dataset using a Resnet-34 CNN pre-trained model. The same transforms which were applied for the Fashion Classifier model are used

to transform each of the images which come in as batches through the DataLoader. The model training is different from what was done for the training of the Fashion Classifier model because of the presence of multiple targets for this task versus one target in the Fashion Classifier model. The task of outputting multiple outputs is known as Multi-label Classification [16]. This required the model to have 3 different classifiers – one for each of the targets (Excluding *tags*). The classifier is a layer with *in_features* as 512 and *out_features* as the number of classes for each of the targets. For example, the classifier for the *graphical_appearance_name* would have a layer taking 512 features as input and outputting 29 tensors. The model was trained using a Cross-Entropy Loss function and calculated loss for each of the classifier heads separately which can then later be used for evaluation purposes. The model underwent training for a total of 10 epochs which was then stopped. Once the model was trained, we took the layer preceding the 3 classifier heads to get the *feature vector* of our images. A feature vector, simply put, is a tensor that has encoded information about the image. For example, a feature vector of a t-shirt image is closer to a feature vector of another t-shirt than it is to a feature vector of a trouser. These feature vectors are interchangeably used with the word ‘Embeddings’, and we will be using the word *Embedding* to represent a feature vector in this project. In PyTorch, we will be registering a Forward Hook [17] on the layer from which we want to extract the Embeddings and when an image is passed through the network, the forward function runs our image across the network and when it reaches the registered layer, we copy and output the embeddings. Attached is a code snippet (Refer to Code Snippet 1) implementing the text discussed above.

Model Integration

The model was saved and loaded the same way, as explained in the above section. The purpose of this model is to recommend the outfits to the users based on their preferences, likes, and dislikes. To tackle the Cold Start problem [18], we employ a technique where we collect the preferences of the users and then compare them with the items in their closet and output the items suitable to their preferences. This model is triggered asynchronously in the background whenever the user types the occasion, they are planning to go to in the search box. Spotify Annoy [19] (Approximate Nearest Neighbours Oh Yeah) is a Python library that searches for points that are close to a given query point; we use this library to compute for similar images in the closet, given the users' preferences. Whenever an image is uploaded to the closet, we capture the embeddings, pickle it, and store it in the database. We also build an index of the embeddings and their corresponding indices using AnnoyIndex using a hyperparameter of the number of trees required, which is a trade-off between speed and accuracy, where more the trees mean better the prediction. When this model is triggered, we compute the top 5 closest vectors to the user preferences vector and output them as recommendations. The recommendations are sorted in the order of the closest distances first. For optimization purposes, this model is only loaded once and for the first time and has quick inference times after the model loading activity.

```

class MultiOutputModel(nn.Module):
    def __init__(self, n_product_group_classes, n_graphic_classes, n_product_type_classes):
        super().__init__()
        self.resnet = models.resnet34(pretrained=True)
        self.model_wo_fc = nn.Sequential(*(list(self.resnet.children())[:-1]))

        # create separate classifiers for our outputs
        self.productGroup = nn.Sequential(
            nn.Dropout(p=0.2),
            nn.Linear(in_features=512, out_features=n_product_group_classes)
        )
        self.graphic = nn.Sequential(
            nn.Dropout(p=0.2),
            nn.Linear(in_features=512, out_features=n_graphic_classes)
        )
        self.productType = nn.Sequential(
            nn.Dropout(p=0.2),
            nn.Linear(in_features=512, out_features=n_product_type_classes)
        )

    def forward(self, x):
        x = self.model_wo_fc(x)
        x = torch.flatten(x, 1)

        return {
            'productGroup': self.productGroup(x),
            'graphic': self.graphic(x),
            'productType': self.productType(x)
        }

model = MultiOutputModel(n_product_group_classes=len(df['product_group_name'].unique()),
                        n_graphic_classes=len(df['graphical_appearance_name'].unique()),
                        n_product_type_classes=len(df['product_type_name'].unique())).to(device)

def criterion(loss_func, outputs, pictures):
    losses = 0
    for i, key in enumerate(outputs):
        losses += loss_func(outputs[key], pictures['labels'][f'label_{key}']).to(device)
    return losses

def training(model, device, lr_rate, epochs, train_loader):
    num_epochs = epochs
    losses = []
    checkpoint_losses = []

    optimizer = torch.optim.Adam(model.parameters(), lr=lr_rate)
    n_total_steps = len(train_loader)

    loss_func = nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        for i, pictures in enumerate(train_loader):
            images = pictures['img'].to(device)
            pictures = pictures

            outputs = model(images)

            loss = criterion(loss_func, outputs, pictures)
            losses.append(loss.item())

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if (i+1) % (int(n_total_steps/1)) == 0:
                checkpoint_loss = torch.tensor(losses).mean().item()
                checkpoint_losses.append(checkpoint_loss)
                print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_total_steps}], Loss: {checkpoint_loss:.4f}')
        return checkpoint_losses

checkpoint_losses = training(model, device, 0.0001, 10, train_loader)

```

Code Snippet 1 – Training a multi-output classifier

Project Outcome

The project is a web application developed using Flask to allow the users to sign up, log in, upload pictures of their closet, edit their profile, submit their preferences, etc...

1. Registration

The Flask application presents a *registration page* for the users to register themselves with the application. Basic information such as name, email, and password are captured and stored in the database. Storing passwords in the database as plaintext is a bad practice and hence, we encrypt the password before storing it in the database. We generate a password hash for the plaintext password using the werkzeug security [20] library. Basic validations are done when a user tries to register with the application and a user will be registered successfully following these validations. The API route '/signup' is called with a *POST* method which accepts the values from those fields and performs a set of validations -

- Check if a user already exists with the given username
- Check if a user already exists with the given email

Post these validations, a message is displayed to the user whether they were successfully registered, if an email already exists or if the username is already taken. In Flask, displaying these messages is called Message Flashing [21].

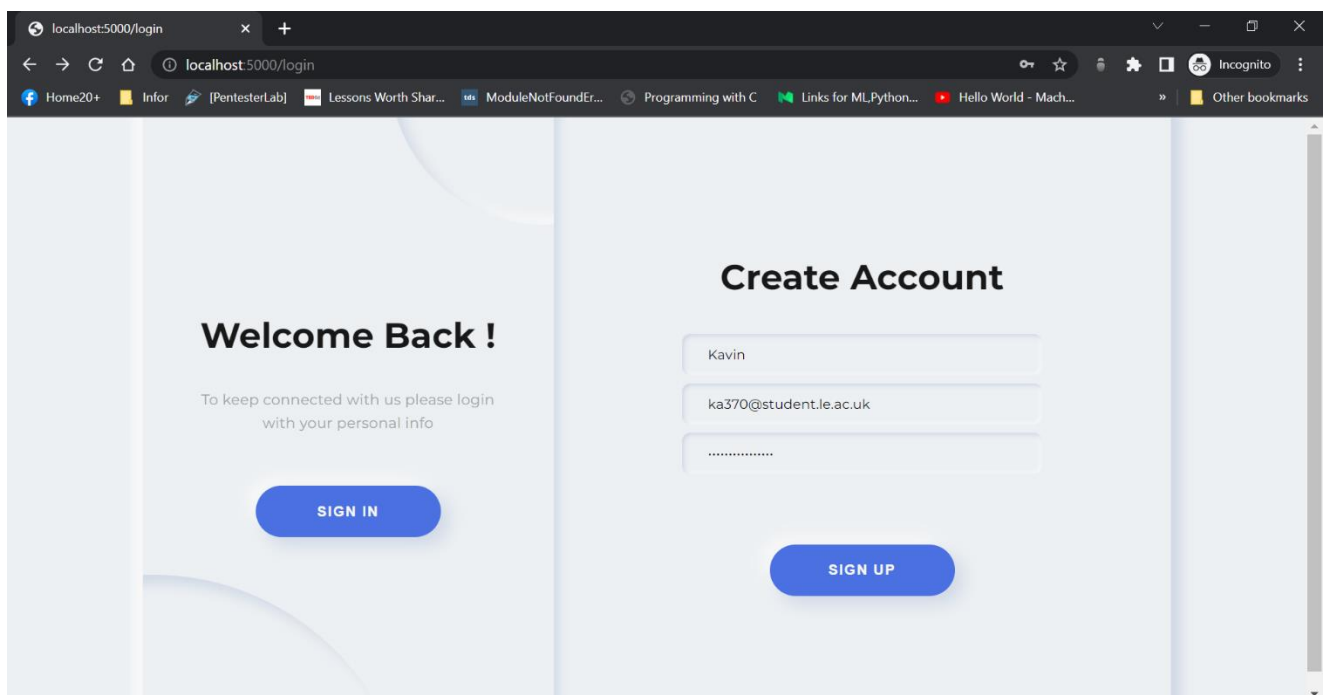


Figure 9 - Registration Page

Message Flashing is done to show the users the state they are in, and in our application, we show the success messages in green colour, whereas the error messages in red colour.

2. Login

The Flask application presents a login page for the users to log in and use the functionality of the application. A user can enter their details such as the email id and the password they registered with to log in, and upon successful validation of the credentials, they will be logged into the application. The API route `‘/signin’` is called with a *POST* method where the entered email id and password are sent for validation. The password entered is hashed and checked with the hashed password that is stored in the database using the werkzeug security library. Flask-Login [22] has been used for user management purposes for common tasks such as logging in, logging out, and remembering the user’s session over extended periods. It also allows to stop users from viewing pages that they do not have access to, for example, only logged-in users are allowed to view the *home page*.

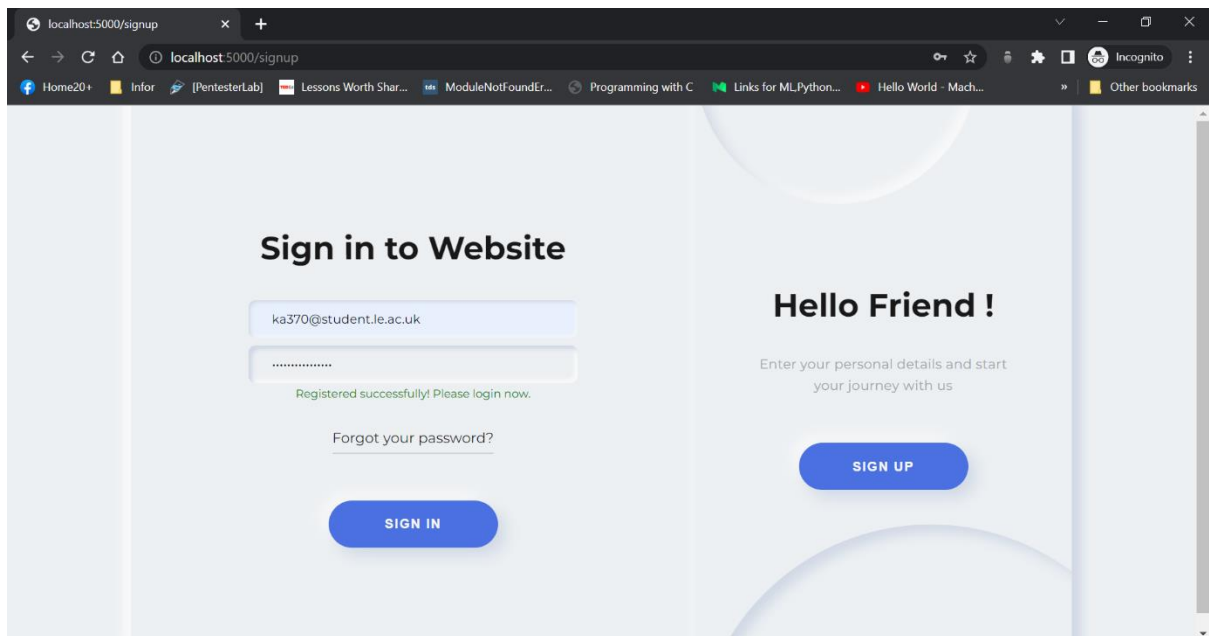


Figure 10 – Login Page

3. Home

The Flask application presents a home page for the users to use the functionality of the application.

The home page consists of a section called '*My Closet*' where a user can view their existing pieces (if applicable) of clothing categorized by their product types and has an interface where a user can upload the items of their closet. The API route – '*/home*' with the method GET is used to render the HTML to the user, in which, the table *UsersCloset* is queried to get the existing images in the user's closet to be displayed in the appropriate categories. The API route – '*/uploadClosetImages*' is called with the method POST upon the user clicking the '*upload images*' button on the interface – this triggers the model asynchronously for it to categorise the piece of clothing and to calculate the embeddings of the uploaded image(s). Here, we will discuss in detail how we made this application work asynchronously to give the user a seamless experience.

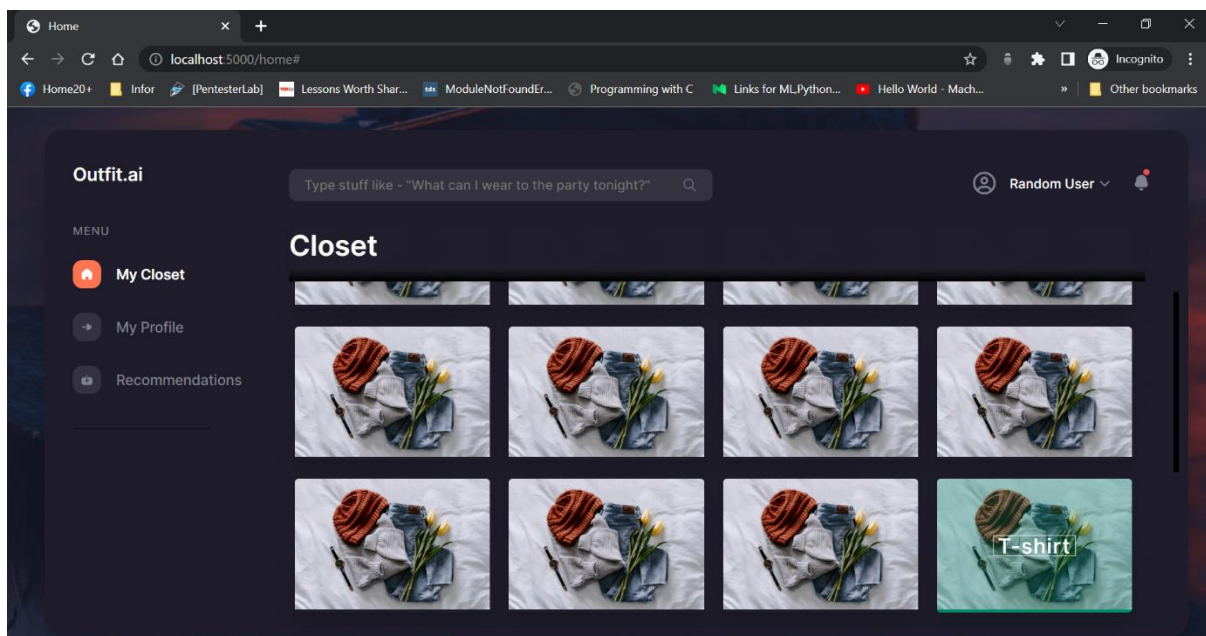


Figure 11 – Home Page showing the Closet section

To achieve asynchronous processing, a python library called *Celery* was used. Celery is an asynchronous task queue based on distributed message passing. In our project, we have used Celery to run the long-running tasks in the background and to allow the user to use the application without having the users wait for the long-running processes to complete. We are using Redis [23] as the Message Broker to allow for the tasks to communicate with each other. In this celery task, we load the Fashion Classifier Model if the model has not already been loaded and try to get the predictions for the unprocessed images and store them in the *UsersCloset* Table under the *PredictedCategory* column. In the same task, we also load the model from which we get the feature vectors by attaching a forward hook on the layer which precedes the classifier heads (as explained in the section explaining our recommender model) and use the obtained feature vectors to be stored in the *UsersCloset* Table under the Embeddings column, after pickling.

Attached is a code snippet (Refer to Code Snippet 2) implementing the text discussed above.

```
def classifyClosetImagesAsync(self,userid):

    global fashion_classifier_model

    if fashion_classifier_model is None:
        fashion_classifier_model = load_checkpoint(os.path.join(app.root_path, 'ml',
'models',clothingClassifierCheckpoint))

    global rec_model
    if rec_model is None:
        rec_model = MultiOutputModel(n_product_group_classes=n_product_group_classes,
                                    n_graphic_classes=n_product_group_classes,
                                    n_product_type_classes=n_product_group_classes).to(device)

        rec_model.load_state_dict(torch.load(os.path.join(app.root_path, 'ml',
'models',recommenderCheckpoint)))
        rec_model.cpu()
        rec_model.eval()

    layer = rec_model._modules['model_wo_fc']._modules['8']

    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                              [0.229, 0.224, 0.225])])

    def get_vector(image):
        t_img = transform(image)
        my_embedding = torch.zeros(512)
        def copy_data(m, i, o):
            my_embedding.copy_(o.flatten())
        h = layer.register_forward_hook(copy_data)
        with torch.no_grad():
            rec_model(t_img.unsqueeze(0))
        h.remove()
        return my_embedding

    closetImagesFiles = os.listdir(os.path.join(app.root_path, 'static',
'upload',str(userid),'closet'))

    processedFiles = UsersCloset.query.filter(UsersCloset.UserId ==
userid).with_entities(UsersCloset.ImageFile).all()
    processedFiles = [imageFile for imageFile, in processedFiles]

    unprocessedFiles = [item for item in closetImagesFiles if item not in processedFiles]

    for file in unprocessedFiles:
        img_path = os.path.join(app.root_path, 'static', 'upload',str(userid),'closet',file)
        probs, classes = predict(img_path, fashion_classifier_model, 1)

        predictedCategory = classes[0]

        img = Image.open(img_path)
        embeddings = get_vector(img)

        pickle_string = pickle.dumps(embeddings)

        closet_object = UsersCloset(ImageFile = file,PredictedCategory = predictedCategory,
                                    Embeddings = embeddings, UserId = userid)
        db.session.add(closet_object)

    db.session.commit()
```

Code Snippet 2 – Classifying User’s Closet using Celery

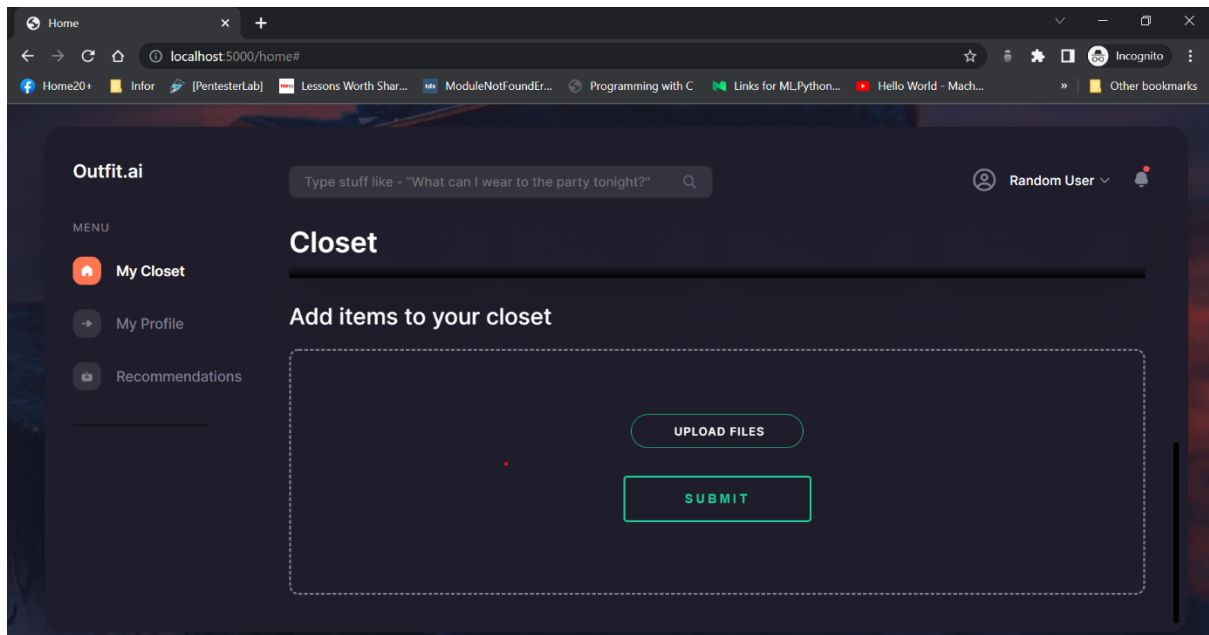


Figure 12 – Home Page showing the upload interface

The home page consists of a second section called 'My Profile' wherein a user can update their *First Name*, *Last Name*, *Email Address*, *Profile Photo*, and *Gender*. This section is particularly useful in capturing this information to understand the type of users using this application and display the recommendations accordingly. For instance, the Gender information is useful in displaying the categories accordingly and the profile photo could be used to create a 2D or 3D model representing the user's physical appearance to show how the outfits would look on them. However, the 2D or 3D model character development is out of scope for this project, given the time constraints, but can be implemented given enough time for the same. Figure 12 shows the profile section as displayed in our application.

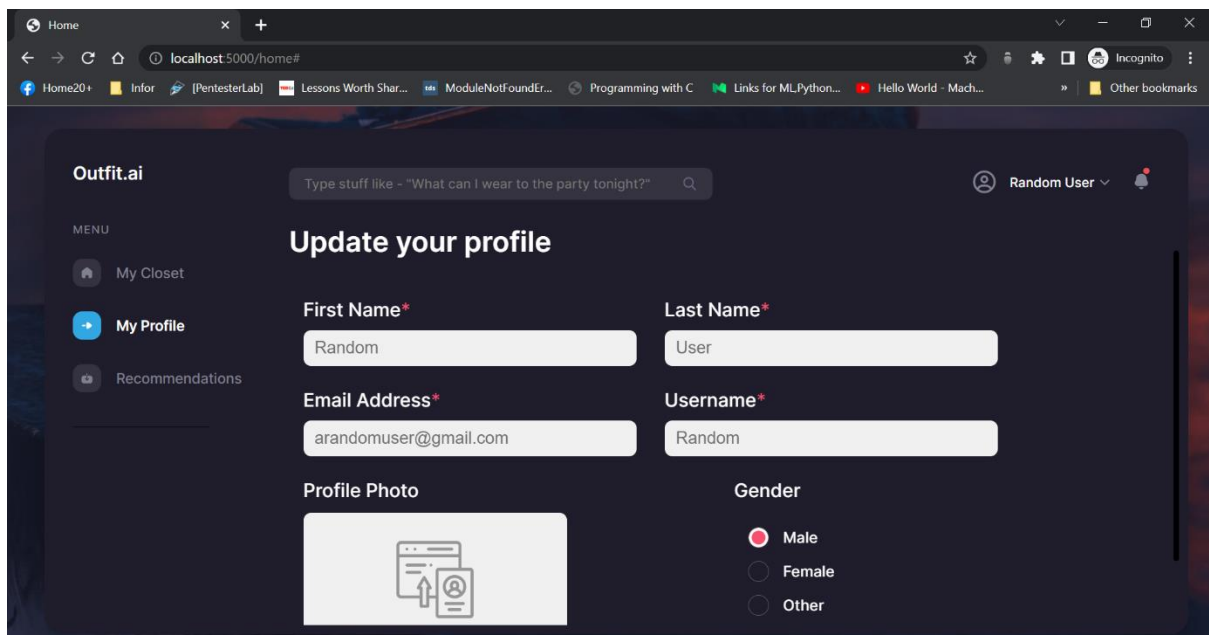


Figure 13 – Home Page showing the profile section

The third section on the home page is the recommendations section, where the user will be able to see the outfit recommendations tailored for them. Users who log in the first time and have uploaded some of their items in the digital closet would not be able to see the recommendations straight away; this is because the recommender model would not be able to build a user vector for the user without knowing their preferences, and hence, they are presented with a questionnaire showing a set of questions and images to understand the user better. A few questions that the application might ask are – “What do you prefer wearing to a beach?”, “What do you prefer wearing to a dinner?”, etc...

This method of asking for users’ preferences is to remedy the Cold Start problem that usually occurs in Recommender Systems. The Cold Start problem is a problem in cars, where if the engine is too cold, the car wouldn’t start, however, after running for a bit, it runs smoothly. The same is the problem in Recommender Systems, where, we wouldn’t have much information about the user at the beginning, however, along the way, we would have collected as much information to give them real good recommendations.

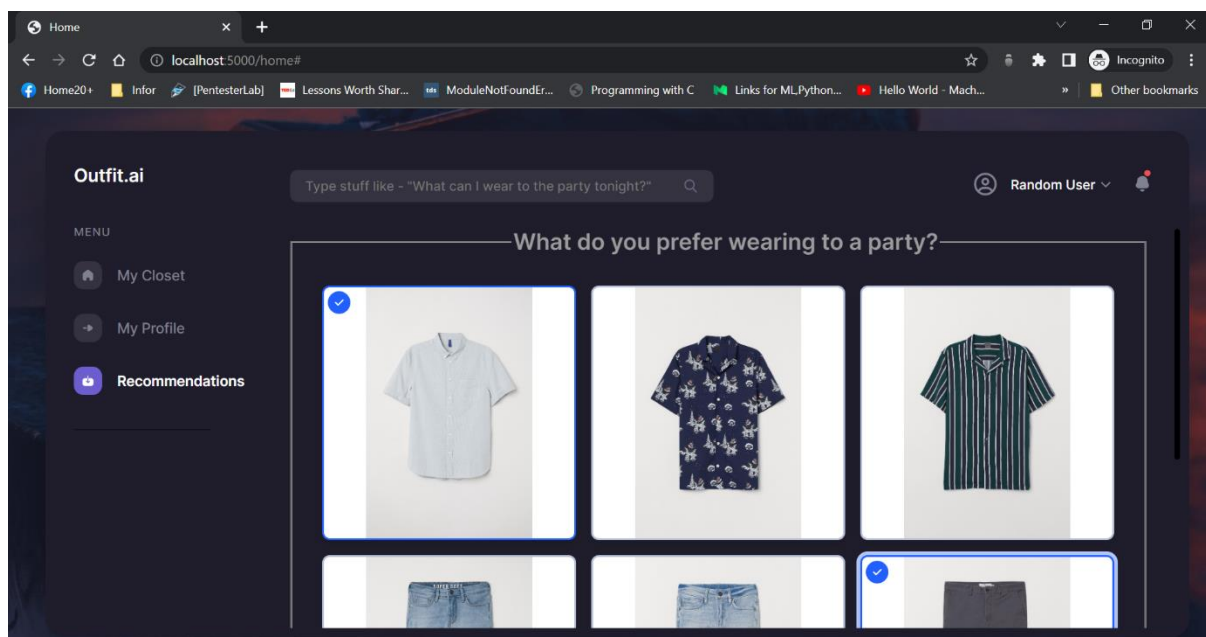


Figure 14 – Home Page showing the recommendations section with preferences

Once the preferences are stored in the *UserPreferences* table, the application is now ready to recommend outfits to the users. To trigger the recommender model, one must type into the search box (displayed on the top of the page) where they plan to go. The API route – ‘/search’ is called with the method POST and triggers the recommender model in the background with the help of Celery which computes a vector using the user preferences, likes, and dislikes and then compares the resulting vector with the embeddings stored in the database for each closet item and recommends the top 5 ranked items for each of the wear type (Upper, Lower). This

Recommender model is a Content-based model, meaning that it produces recommendations solely based on the product attributes available, whereas Collaborative filtering generates recommendations based on user behaviour. However, Collaborative Filtering couldn't be applied to an application such as ours, since Collaborative Filtering is a filtering technique that is used to filter items that a user might like based on reactions by similar users; in our application, the items are user-owned, and not specific to the platform and hence we will not be able to filter out the items based on similar users. Coming back to the discussion of Content-based models, in the context of our application, the product attributes available to us are the dense representation of the item (Embeddings) from which we do a cosine similarity matching to find the closest items from their closet. Once the user uploads the images in their closet and submits their preferences, they are now ready to get recommendations given that they enter the occasion they're planning to go. The figure below shows the recommendations generated for a user when they type in the words "What should I wear to the upcoming party?"

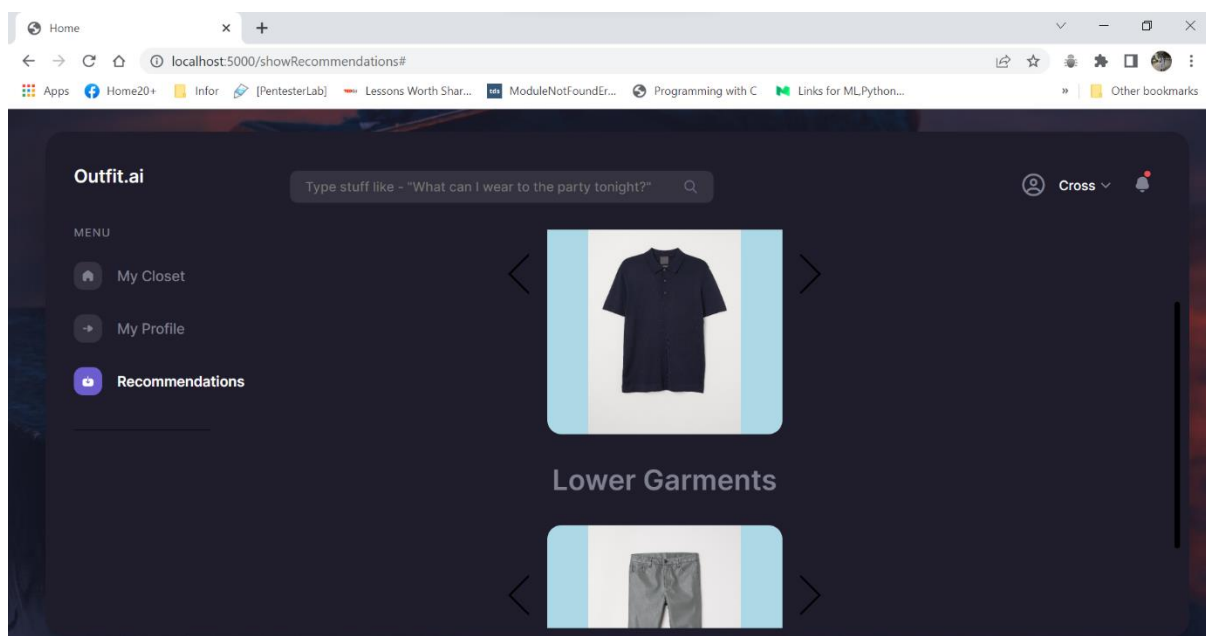


Figure 15 – A home page showing the recommendations in the recommendations section

The celery task '*triggerRecommenderModel*' is called whenever a user types something relevant to the occasion they're planning to go. The user id and the event are passed as parameters to the task. The user id is needed to query the *UsersCloset* table to get the closet images and their corresponding embeddings from the database to build an index using the AnnoyIndex (as discussed in the above sections). The recommender model is initialized and loaded from the checkpoint, and a layer is selected on which a forward hook is attached to get the feature vector of the user preferences – the event is used to get the relevant images from the list of all the user preferences. Each of the selected images is then run through a function which gets the nearest neighbors by its vector, and these images are then displayed to the user as recommendations in the sorted order of closest first. Attached is a code snippet that contains the code implementation for the text above.


```

@celery.task(bind=True)
def triggerRecommenderModel(self,userid, userEvent):

    closet = UsersCloset.query.filter(UsersCloset.UserId ==
userid).with_entities(UsersCloset.ImageFile,UsersCloset.Embeddings).all()

    df = pd.DataFrame(closet, columns=['ImageFile', 'Embeddings'])

    f = len(df['Embeddings'][0])
    t = AnnoyIndex(f, metric='euclidean')

    ntree = 1000 # hyper-parameter, the more the number of trees better the prediction
    for i, vector in enumerate(df['Embeddings']):
        t.add_item(i, vector)
    _ = t.build(ntree)

    # Initialize the recommender model and get the embeddings of users preferences
    global recommender_model
    if recommender_model is None:
        print("Loading for the first time")

        recommender_model = MultiOutputModel(n_product_group_classes=n_product_group_classes,
n_graphic_classes=n_product_group_classes,
n_product_type_classes=n_product_group_classes).to(device)

        recommender_model.load_state_dict(torch.load(os.path.join(app.root_path, 'ml',
'models',recommenderCheckpoint)))
        recommender_model.cpu()
        recommender_model.eval()

    # Use the model object to select the desired layer
    layer = recommender_model._modules['model_wo_fc']._modules['8']

    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
[0.229, 0.224, 0.225])])

    def get_vector(image):
        t_img = transform(image)
        my_embedding = torch.zeros(512)
        def copy_data(m, i, o):
            my_embedding.copy_(o.flatten())
        h = layer.register_forward_hook(copy_data)
        with torch.no_grad():
            recommender_model(t_img.unsqueeze(0))
        h.remove()
        return my_embedding

    closetImagesFiles = os.listdir(os.path.join(app.root_path, 'static',
'upload',str(userid),'closet'))

    usersPref = UsersPreferences.query.filter(UsersPreferences.UserId ==
userid).with_entities(UsersPreferences.choice).first()

    userPreferences = list(usersPref)
    userPreferences = list(np.concatenate(userPreferences).flat)

    recommended_items = []

    allEvents = db.session.query(Events.event).all()
    EventsList = []
    for event in allEvents:
        EventsList.append(event[0].lower())

    userPreferencesIndex = EventsList.index(userEvent)

    # From user preferences, choose 1 of upper body garment and choose 1 of lower body garment or
    choose 1 of full body garment

    for file in userPreferences[userPreferencesIndex]:
        img_path = os.path.join(app.root_path, 'static', 'upload',str(userid),'preferences',file)
        img = Image.open(img_path)
        embeddings = get_vector(img)
        similar_images = t.get_nns_by_vector(embeddings, n=5, include_distances=True)
        recommended_items.append(similar_images[0])

    recommendations = UsersRecommendations(img_index = recommended_items, UserId = userid)
    db.session.add(recommendations)

    db.session.commit()

```

Code Snippet 3

The API route to show recommendations is – ‘/showRecommendations’ which captures the recommendations from the database. Recommendations can be in the form of ‘full type garments’ or ‘an upper garment and a lower garment’, for the scope of this project, we are now only considering the recommendations of the latter category in which we will show the recommendations for the upper garments and lower garments separately. We get the `img_index` from the recommendations table and query the UsersCloset to get the corresponding images to then show them as the output in our application.

```
@app.route("/showRecommendations")
@login_required
def showRecommendations():
    recommendations = UsersRecommendations.query.filter(UsersRecommendations.UserId ==
current_user.UserId).order_by(UsersRecommendations.id.desc()).with_entities(UsersRecommendations.img_in
dex).first()

    recs = recommendations[0]

    if len(recs) == 1:
        # Full body garment
        pass
    else:
        # 1 top garment, 1 lower garment
        top_garment_items = recs[0]
        lower_garment_items = recs[1]

        print("206",top_garment_items)
        print("207", lower_garment_items)

        top_garments = UsersCloset.query.filter(UsersCloset.UserId ==
current_user.UserId).with_entities(UsersCloset.ImageFile).all()
        lower_garments = UsersCloset.query.filter(UsersCloset.UserId ==
current_user.UserId).with_entities(UsersCloset.ImageFile).all()

        top_garments_list = []
        lower_garments_list = []

        for item in top_garment_items:
            top_garments_list.append(top_garments[item])

        topGarments = [i[0] for i in top_garments_list]

        for item in lower_garment_items:
            lower_garments_list.append(lower_garments[item])

        lowerGarments = [i[0] for i in lower_garments_list]

        garments_dict['lowerGarments'] = lowerGarments
        garments_dict['topGarments'] = topGarments

        UserPreferencesExists = UsersPreferences.query.filter(UsersPreferences.UserId ==
current_user.UserId).first()

        return render_template("home.html", clothes_data = dic, user = str(current_user.UserId), garments =
garments_dict, UserPreferencesExists = UserPreferencesExists)
```

Code Snippet 4 – Code to show recommendations

Project Structure

The author has chosen to carefully architect and structure the system for code maintainability purposes. This being an application with a lot of components, things can get quite messy real quick, and hence, a good structure always helps for debugging and usability purposes.

```
Root Folder/
├── /__pycache__
├── /env
├── /outfitai
│   ├── ml/
│   │   ├── models/
│   │   │   ├── fashion-classifier.pth
│   │   │   └── recommender-model.pth
│   ├── /static
│   │   ├── /css
│   │   ├── /js
│   │   ├── /img
│   │   └── /upload
│   ├── /templates
│   ├── __init__.py
│   ├── models.py
│   └── views.py
├── tests/
│   └── test_outfitai.py
├── celery_utils.py
├── run.py
└── requirements.txt
```

Figure 16 – Directory structure of the application

1. The *env* folder contains the information about the virtual environment the application was developed in. A virtual environment is helpful when developing applications because it isolates the working environment from colliding with other libraries installed in the system.
2. The *outfitai* folder contains a few other folders and files –
 - `__init__.py` – This file is the secondary point of the application where the flask, celery, and database instances are created.
 - `models.py` – This file contains the schema of the database. The tables, the columns, and the constraints for those are created through this file.
 - `views.py` – This file contains all the API routings, the Celery tasks, and a few miscellaneous functions.
 - `Templates` – This folder contains the HTML and Jinja templates that the application uses for rendering purposes.

- Static – This folder contains the CSS, JS, and images that are used by the HTML, also for rendering purposes. It also contains a folder named 'upload' which contains all the content that the user uploads from the application, each user has a separate sub-folder in this folder.
 - ml – This folder contains sub-folder named *models* which contains the checkpoints of the trained models.
3. The tests folder contains a python file containing the unit tests to test the application.
 4. The *celery_utils.py* contains the configurations for the broker, and backend and has a method to initialize the celery instance, which is called by the *__init__.py* file.
 5. The *run.py* file is the entry point to the whole application, which imports the outfitai as a package and runs it.
 6. The *requirements.txt* file contains the library requirements needed to run this application, particularly helpful during deployment stages.

Build and run

This section assumes that the reader has downloaded the code from **/ka370/code/trunk/app**. This application requires Redis to serve as a message broker, and this demands the need for a Subsystem for Linux.

For windows users, WSL (Windows Subsystem for Linux) must be enabled and installed.


For macOS users, Lima (Linux-on-mac) must be enabled and installed.

1. Open a terminal and navigate to the root folder of the application.
2. Change your current directory to **/env/Scripts** and run **activate**
3. Change your current directory to the root folder of the application and run **flask run**
4. Open Ubuntu and run **sudo apt install redis-server**
5. In the same Ubuntu terminal, run **sudo service redis-server start**
6. Open a second terminal and navigate to the root folder of the application.
7. Change your current directory to **/env/Scripts** and run **activate**
8. Change your current directory to the root folder of the application and run **celery -A outfitai.celery worker --loglevel=INFO --concurrency 1 -P solo**
9. Go to **localhost:3000** in your browser

Testing and Quality Assurance

For Quality Assurance purposes, the author decided to test the application using a library called PyTest. PyTest allows us to write Unit Tests in Python and is mainly used for API testing. However, we used it in our application to see if everything is working as intended.

In the attached code snippet (Code Snippet 5), the first unit test tests whether we are fetching the right landing page when we query the endpoint ('/'). In the same code snippet, the second unit test tests whether we are fetching the login page when we query the endpoint ('/login').

A code snippet displayed in a dark-themed editor window with three colored window control buttons (red, yellow, green) in the top-left corner. The code is a Python file for unit testing using PyTest. It imports pytest, json, flask's session, and the application's app and models. It defines a pytest fixture 'client' that returns app.test_client(). Two test functions are defined: 'test_index' which checks the status code is 200 and the response contains 'Let me in!', and 'test_login' which checks the status code is 200 and the response contains 'Sign in'.

```
import pytest
import json
from flask import session
from outfitai import app
from outfitai.models import *

@pytest.fixture
def client():
    return app.test_client()

def test_index(client):
    response = client.get('/')
    assert response.status_code == 200
    assert b'Let me in!' in response.data

def test_login(client):
    response = client.get('/login')
    assert response.status_code == 200
    assert b'Sign in' in response.data
```

Code Snippet 5 – Unit Tests

In the attached code snippet (Code Snippet 6), the first test checks this - given a user model, when a new user is created, then check if the email and the password hash are created correctly.

In the same code snippet, the second test checks this – given a user who isn't logged in, when the user tries to access a page that they do not have access to (home page), then check if the user is being redirected to the sign-in page instead.

In the same code snippet, the third test checks this – given a user who is logged in, when the user tries to access a page that they have access to, then check if the user is being allowed to access that page.

```
def test_new_user(client):
    """
    GIVEN a user model
    WHEN a new user is created
    THEN check the email and password hash are defined correctly
    """
    new_user = Users('kavin','kavinarasu22@gmail.com','1234')
    assert new_user.Username == 'kavin'
    assert new_user.Email == 'kavinarasu22@gmail.com'
    assert new_user.Password != '1234'

def test_home_page_not_logged_in(client):
    """
    GIVEN a user who isn't logged in
    WHEN the user tries to access a page that they do not have access to
    THEN check if the user is being redirected to the sign in page instead
    """
    response = client.get('/home', follow_redirects = True)
    assert response.status_code == 200
    assert b'Sign in' in response.data

def test_home_page_logged_in(client):
    """
    GIVEN a user who isn logged in
    WHEN the user tries to access a page that they have access to
    THEN check if the user is being allowed to access that page
    """
    with client:
        client.post("/signin",data=dict(email="kavinarasu22@gmail.com",password="1234"))
        response = client.get('/home', follow_redirects = True)
        assert response.status_code == 200
        assert b'My Closet' in response.data
}
```

Code Snippet 6 – Unit Tests

In the code snippet (Code Snippet – 7), the first test checks whether the fashion recommender model checkpoint and recommender model checkpoint is available in the path - /ml/models

In the same code snippet, the second test asserts an error message if they have not uploaded anything in their closet and tries to get recommendations by typing in the search box.

In the same code snippet, the third test asserts an error message if they have not selected any of their preferences and tries to get recommendations by typing in the search box.

```
def test_model_checkpoints_exists():
    clothingClassifierCheckpoint = 'fashion-classifier.pth'
    assert os.path.exists(os.path.join(app.root_path, 'ml', 'models', clothingClassifierCheckpoint))

    recommenderCheckpoint = 'recommender-model.pth'
    assert os.path.exists(os.path.join(app.root_path, 'ml', 'models', recommenderCheckpoint))

def test_with_no_closet_items(client):

    with client:
        client.post("/signin", data=dict(email="kavinarasu22@gmail.com", password="1234"))
        client.post("/search", data=dict(searchbox="What can I wear to the dinner tomorrow?"))
        response = client.get('/home')
        assert b'You either do not have anything in your closet or have not given your preferences' in
        response.data

def test_with_no_preferences(client):

    with client:
        client.post("/signin", data=dict(email="kavinarasu22@gmail.com", password="1234"))
        client.post("/search", data=dict(searchbox="What can I wear to the dinner tomorrow?"))
        response = client.get('/home')
        assert b'You either do not have anything in your closet or have not given your preferences' in
        response.data
```

Code Snippet 7 – Unit Tests

Evaluation

Recommender Model

To evaluate the recommender model in terms of accuracy, we would need to validate how well the model can classify an image. Our recommender model was trained to predict *product_group_name*, *graphical_appearance*, and *product_type_name* given an image and out of the several categories, it needs to pick the correct labels for each image. The figures below show how our model performed on our test dataset.

Overall class performance: 94.8 %
Accuracy of Garment Full body: 92.0 %
Accuracy of Garment Lower body: 95.6 %
Accuracy of Garment Upper body: 95.1 %

Figure 17 – Accuracy scores of product_group_name by categories

Overall class performance: 76.7 %
Accuracy of All over pattern: 86.2 %
Accuracy of Application/3D: 17.2 %
Accuracy of Argyle: 0.0 %
Accuracy of Chambray: 23.5 %
Accuracy of Check: 88.4 %
Accuracy of Colour blocking: 51.9 %
Accuracy of Contrast: 20.0 %
Accuracy of Denim: 80.0 %
Accuracy of Dot: 36.4 %
Accuracy of Embroidery: 32.9 %
Accuracy of Front print: 66.7 %
Accuracy of Glittering/Metallic: 22.2 %
Accuracy of Jacquard: 30.4 %
Accuracy of Lace: 62.7 %
Accuracy of Melange: 63.7 %
Accuracy of Mesh: 0.0 %
Accuracy of Metallic: 0.0 %
Accuracy of Mixed solid/pattern: 6.5 %
Accuracy of Neps: 50.0 %
Accuracy of Other pattern: 0.0 %
Accuracy of Other structure: 16.8 %
Accuracy of Placement print: 37.6 %
Accuracy of Sequin: 55.6 %
Accuracy of Slub: 0.0 %
Accuracy of Solid: 89.6 %
Accuracy of Stripe: 75.2 %
Accuracy of Treatment: 18.2 %
Accuracy of Unknown: 0.0 %

Figure 18 – Accuracy scores of graphical_appearance by categories

Overall class performance: 84.5 %
Accuracy of Blazer: 88.3 %
Accuracy of Blouse: 72.6 %
Accuracy of Cardigan: 60.3 %
Accuracy of Coat: 61.2 %
Accuracy of Dress: 93.3 %
Accuracy of Hoodie: 67.5 %
Accuracy of Jacket: 65.1 %
Accuracy of Polo shirt: 78.0 %
Accuracy of Shirt: 80.1 %
Accuracy of Shorts: 90.0 %
Accuracy of Skirt: 80.7 %
Accuracy of Sweater: 89.9 %
Accuracy of T-shirt: 79.6 %
Accuracy of Trousers: 93.5 %

Figure 19 – Accuracy scores of product_type_name

The accuracy scores for some of the categories look quite good, however, there are a few categories where the model got an accuracy score of 0% and this might be because of the following reasons –

- Some of the categories do not have enough data and it could be that the test dataset, unfortunately, did not include these categories for them to be classified.
- The model might have got confused some patterns with others and misclassified those pieces.



Figure 20 - Similar images to a given query image

Next, let us evaluate the embeddings that the model gives us. We do this by selecting an image randomly from the dataset, querying for the closest 4 images – that is, by using the embedding we got for our selected image, we calculate the Euclidean distance with each of the image embeddings in our dataset.

In the figure above, we can see that our query image is a patterned shirt, and the closest 5 images are all patterned shirts as well. However, what is more interesting is that the images with article_id – 747133001, 817295005 also have patterns very similar to that of the query image, the only difference is that the latter is of a different colour. Interestingly, the results which we obtained are out of a huge dataset of almost 64,000 images and the accuracy to output all shirts for a query shirt, with patterns closer to that of the query. This shows that our embeddings were able to capture the information about the images.

Now that we have evaluated the recommender model in terms of accuracy, let us now evaluate the recommender model in terms of speed. To make the recommender system work in real-time, the model must be fast while giving recommendations – and thus, we used Spotify Annoy (as discussed in the model integration section of the recommender model) which is a trade-off between speed and accuracy. However, only little accuracy is sacrificed while giving extremely fast results.

Metric Considered	Metric Value	Comments
Time taken for the model to load	5 seconds	Only loaded for the first time
Time taken to produce recommendations for a given user and event	0.2 – 0.3 seconds	

Table 3 - Metrics for recommender model

Fashion Classifier Model

To evaluate our fashion classifier model, we need to calculate the accuracy, which shows how well our model was able to classify images on the test dataset or real-world data.

Metric Considered	Metric Value	Comments
Accuracy on the test dataset	76%	
Time taken to load the model	5 - 6 seconds	
Time taken for inference on an image	Less than 0.1 seconds	

Table 4 - Metrics for fashion classifier model

Limitations and Future work

Given time, there are a few things I would do to improve the application. These improvements are roughly divided into 3 categories – additional features, code refactoring, and frontend styling.

Additional Features

1. *Complete-the-Look* – This feature would recommend the matching accessories for the outfit that was recommended.
2. *Shop-the-Look* – This feature would suggest to users any missing items in their closet for them to buy to make them look their best when outside.
3. *Outfit visualiser* – This feature would show the recommended outfits on the users' 2D or 3D models to give them a picture glance of how they would look in that outfit.
4. *Gender categorisation* – This feature would only try to show the relevant categories according to their gender.
5. *Hybrid Recommender Model* – This feature would try to incorporate the preferences of the users more than what the content-based model could do.

Code refactoring

The code could be refactored in many places, optimising the application for the better. The HTML page which shows the home page needs a refactor – code duplication could have been reduced using the jinja2 templating style. The coding style for Python could be improved using a style guide [24].

Frontend styling

The frontend displaying the flash messages is now just a text with colours changing with success or error messages, however, it could have been improved with icons, bumper animations for invalid inputs, etc... The frontend could have been developed in React, as it could have been easier to reuse components, avoid duplicating code, etc...

Conclusion

The project aimed to create an outfit recommender system for users who do not wish to spend much time selecting their outfits. To ensure accuracy and speed, an approach to using embeddings to find similar outfits from their preferences was introduced. All the essential requirements have been implemented, except for the requirement in which we state to use a dataset to train a model which would detect social events from images – this wasn't implemented as it didn't give satisfiable results. This requirement was then substituted with another requirement which captures the users' outfit preferences for social events. In the implementation and evaluation section, explanations regarding each of the models and their metrics were discussed. Explanations regarding why a content-based model is suitable for our project versus a collaborative filtering method were also discussed. The report also explains some key code snippets and figures to understand the project better.

References

- [1]"DeepFashion Database", Mmlab.ie.cuhk.edu.hk, 2022. [Online]. Available: <https://mmlab.ie.cuhk.edu.hk/projects/DeepFashion.html>.
- [2]"What is Kaggle, Why I Participate, What is the Impact? | Data Science and Machine Learning", Kaggle.com, 2022. [Online]. Available: <https://www.kaggle.com/getting-started/44916>.
- [3]"H&M Personalized Fashion Recommendations | Kaggle", Kaggle.com, 2022. [Online]. Available: <https://www.kaggle.com/c/h-and-m-personalized-fashion-recommendations>.
- [4]"Splitting into train, dev and test sets", Cs230.stanford.edu, 2022. [Online]. Available: <https://cs230.stanford.edu/blog/split/>.
- [5]"split-folders", PyPI, 2022. [Online]. Available: <https://pypi.org/project/split-folders/>.
- [6]"PyTorch", Pytorch.org, 2022. [Online]. Available: <https://pytorch.org/>.
- [7]2019.ieeeicip.org, 2022. [Online]. Available: <https://www.2019.ieeeicip.org/2019.ieeeicip.org/upload/files/201901181502263536.pdf>.
- [8]J. Brownlee, "A Gentle Introduction to Transfer Learning for Deep Learning", Machine Learning Mastery, 2022. [Online]. Available: <https://machinelearningmastery.com/transfer-learning-for-deep-learning/>.
- [9]"torchvision — Torchvision 0.12 documentation", Pytorch.org, 2022. [Online]. Available: <https://pytorch.org/vision/stable/index.html>.
- [10]"ImageNet - Wikipedia", En.wikipedia.org, 2022. [Online]. Available: <https://en.wikipedia.org/wiki/ImageNet>.
- [11]"What is early stopping?", Educative: Interactive Courses for Software Developers, 2022. [Online]. Available: <https://www.educative.io/edpresso/what-is-early-stopping>.
- [12]I. Education, "What is Overfitting?", Ibm.com, 2022. [Online]. Available: <https://www.ibm.com/cloud/learn/overfitting#:~:text=Overfitting%20is%20a%20concept%20in,unseen%20data%2C%20defeating%20its%20purpose>.
- [13]"pickle — Python object serialization — Python 3.10.4 documentation", Docs.python.org, 2022. [Online]. Available: <https://docs.python.org/3/library/pickle.html>.
- [14]"Google Colab", Research.google.com, 2022. [Online]. Available: <https://research.google.com/colaboratory/faq.html#:~:text=Colaboratory%2C%20or%20%E2%80%9CColab%E2%80%9D%20for,learning%2C%20data%20analysis%20and%20education>.
- [15]"What Is CUDA | NVIDIA Official Blog", NVIDIA Blog, 2022. [Online]. Available: <https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/>.

- [16]J. Brownlee, "Multi-Label Classification with Deep Learning", Machine Learning Mastery, 2022. [Online]. Available: <https://machinelearningmastery.com/multi-label-classification-with-deep-learning/>.
- [17]"Debugging and Visualisation in PyTorch using Hooks", Paperspace Blog, 2022. [Online]. Available: <https://blog.paperspace.com/pytorch-hooks-gradient-clipping-debugging/>.
- [18]"The Cold Start Problem for Recommender Systems", Medium, 2022. [Online]. Available: <https://medium.com/@markmilankovich/the-cold-start-problem-for-recommender-systems-89a76505a7>.
- [19]"GitHub - spotify/annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk", GitHub, 2022. [Online]. Available: <https://github.com/spotify/annoy>.
- [20]"Utilities — Werkzeug Documentation (2.1.x)", Werkzeug.palletsprojects.com, 2022. [Online]. Available: <https://werkzeug.palletsprojects.com/en/2.1.x/utils/>.
- [21]"Message Flashing — Flask Documentation (1.1.x)", Flask.palletsprojects.com, 2022. [Online]. Available: <https://flask.palletsprojects.com/en/1.1.x/patterns/flashing/>.
- [22]"Flask-Login — Flask-Login 0.6.1 documentation", Flask-login.readthedocs.io, 2022. [Online]. Available: <https://flask-login.readthedocs.io/en/latest/>.
- [23]"Redis Message Broker | Redis Enterprise", Redis, 2022. [Online]. Available: <https://redis.com/solutions/use-cases/messaging/>.
- [24]"styleguide", styleguide, 2022. [Online]. Available: <https://google.github.io/styleguide/pyguide.html>.