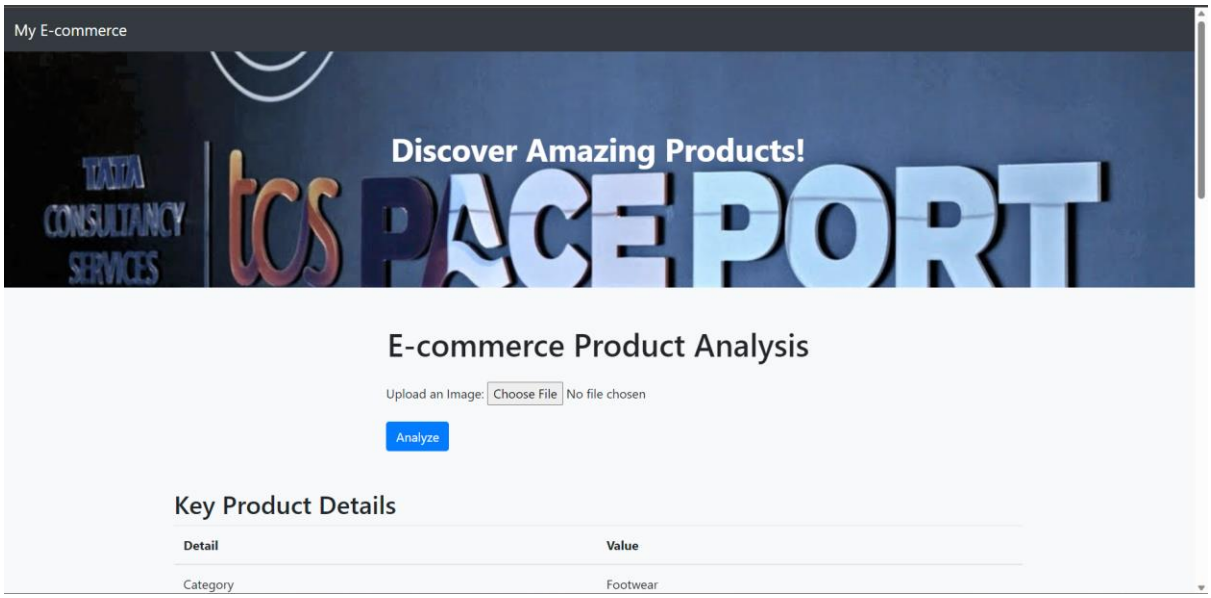
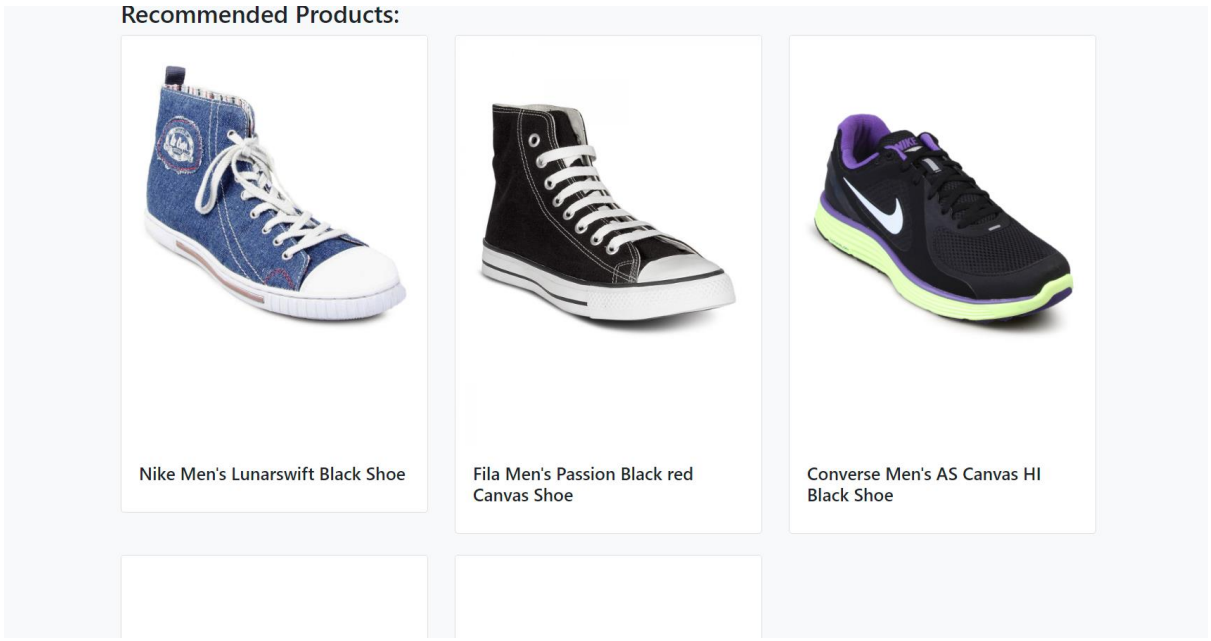
Image Similarity and Categorization Flask Application Report

1. Introduction

This report outlines the development and challenges faced in creating a Flask application for image similarity and categorization. Utilizing Resnet50 and MobileNet(feature extraction) and custom-designed models, the application can predict specific attributes of fashion items, including category, subcategory, colour, product type, and usage and similar product recommendation also.





2. Dataset Details and Preprocessing

The dataset used in the application is provided in question i.e. fashion.csv , This dataset includes images and related attributes. The data is preprocessed using common techniques such as resizing and normalization.

The dataset consists of various attributes related to fashion products:

ProductId: Unique product identifier

Gender: Gender for which the product is intended

Category: General category of the product

SubCategory: More specific classification within the category

ProductType: The type of product within the subcategory

Colour: The colour of the product

Usage: The intended use or occasion for the product

ProductTitle: The title of the product

Image: Filename of the product's image

ImageURL: URL of the product's image

Images are downloaded from URLs using the requests library and saved in a specified directory. The script tracks downloaded and skipped images, with retries for connection errors.
These models(InceptionV3, ResNet50, and MobileNet) extracted relevant features from images, transforming them into numerical representations, suitable for modeling.

3. Model Architecture

A specific model trained to predict the category, subcategory, colour etc. of fashion products. The application leverages MobileNet, a lightweight deep learning architecture optimized for mobile and embedded applications. It's used for feature extraction.

Semantic Analysis

Semantic analysis was targeted at categorizing various attributes such as colour, sub-product, etc., using deep learning models. The following approach I implemented:

As you know a Sequential model was defined with three Dense layers, two of which are followed by a Dropout layer to prevent overfitting. The architecture is as follows:

Input: The input shape is determined by the feature vector from the previous layer.

Dense Layer 1: Comprises 512 units with ReLU activation.

Dropout Layer 1: A dropout rate of 0.3 to prevent overfitting.

Dense Layer 2: Comprises 256 units with ReLU activation.

Dropout Layer 2: A dropout rate of 0.3 to prevent overfitting.

Output Layer: A softmax activation function is used for multi-class classification, with the number of units equal to the number of classes.

The model was compiled using the Adam optimizer and categorical cross-entropy loss function.

Image Similarity Model

A custom Image Similarity Model was created to determine similar products within the dataset. The similarity model is designed to extract features and use pairwise distances to identify similar products. Recommendations were generated using MobileNet for feature extraction and K-Nearest Neighbors (KNN) for finding similar products. The KNN model was trained on the extracted features to recommend similar products based on image features.

Features were extracted from images using the extract_features function, which resizes the images and utilizes a pre-trained model (MobileNet) to get feature vectors.

The find_similar_images function uses the trained KNN model to find the nearest images in feature space, providing a list of similar images.

Description Generation

Initial Experimentation with GPT-2:

Initial experiments with GPT-2 did not yield satisfactory results due to the limited size of the dataset (around 1900 images).

Model for Description Generation

The final model for description generation was designed as follows:

Encoder: A Dense layer with 256 units was applied to the image features. These processed features were repeated (using RepeatVector) to match the sequence length of the text input.

Decoder: The text input was embedded and then processed through an LSTM layer.

Concatenation: The image features and text LSTM output were concatenated, followed by another LSTM layer.

Output: A time-distributed Dense layer with softmax activation was used to predict the next word in the description.

4. Flask App Structure

The Flask application is designed with the following endpoints and functionalities:

index: Renders the main page of the application.

analyze-image: Accepts an image, predicts attributes, recommends similar products, and renders the results.

Models, label encoders, and other resources are loaded appropriately to support these functionalities.

5. Deployment and Testing Strategy

Local Deployment:

The application was designed to be deployed locally, focusing on core functionalities and usability.

Google Cloud Deployment

Deployment on Google Cloud was implemented, using scalable resources such as Kubernetes and Docker containerization.

Testing

Testing was planned with various product images to ensure accurate predictions, description generation, and recommendations.

Google Cloud Deployment

Kubernetes Service Configuration

The application is deployed as a scalable service in Kubernetes with the following specifications:

Service Name: my-ecommerce-service

Labels: app: my-ecommerce-app

Ports: TCP 80 with a target port of 8000

Type: LoadBalancer

The LoadBalancer type allows the application to balance the load among various instances.

Docker Containerization

Building the Docker Image

Base Image: python:3.8-slim-buster

Working Directory: /app

Copy Application Files: All the current directory contents are copied into the container at /app.

Install Requirements: Install packages from requirements.txt.

Expose Port: Port 80 is exposed for external access.

Environment Variable: Set Flask to run in production mode.

Launch Command: Run app.py on launch.

Keras Model Loading

Ensure that the model files being loaded are valid and not corrupted.

Reduce Concurrent Model Loading

Loading models sequentially rather than simultaneously can mitigate memory-related issues.

6. Improvement Opportunities

Code Refactoring

The code can be refactored to isolate functionalities into separate modules and functions. This separation of concerns makes it easier to maintain, test, and debug the application.

Error Handling

Implementing proper error handling and logging would make it easier to diagnose and fix problems in the future.

Resource Optimization

Optimization of memory and other resources should be done to prevent errors related to memory allocation. This includes avoiding loading large models simultaneously and utilizing appropriate data types.

Testing and Validation

Conduct thorough testing and validation of the application to uncover and address potential issues early in the development process.

Utilize Preprocessing Pipelines

Implement preprocessing pipelines for more efficient and consistent data handling. This would streamline the process of preparing images and features for predictions.

Model Optimization

Consider fine-tuning and optimizing the models used for categorization and similarity measurement. Experimenting with different architectures, hyperparameters, and training techniques can lead to improved accuracy and performance. And as I explained above need to resolve for semantic text generation for description.

Conclusion

This report has outlined the critical components, approaches, and troubleshooting methods applied throughout the project. Continued optimization and testing can further enhance the performance and robustness of the application.