1. **Introduction to Django**:
    - Explain what Django is: a high-level Python web framework that enables rapid development of secure and maintainable websites.
    - Mention its key features: follows the MVC (Model-View-Controller) pattern, includes an ORM (Object-Relational Mapping) for database interaction, provides built-in authentication, etc.
    - Emphasize its popularity and widespread use in web development.
2. **Project Structure**:
    - Introduce the project's file structure: **urls.py**, **models.py**, **forms.py**, **views.py**, and **templates** directory.
    - Explain the purpose of each file and directory:
        - **urls.py**: Contains URL patterns that map to views.
        - **models.py**: Defines the database models using Django's ORM.
        - **forms.py**: Contains form definitions for user input validation.
        - **views.py**: Contains view functions that handle HTTP requests and return HTTP responses.
        - **templates**: Directory containing HTML templates for rendering dynamic content.
3. **Models**:
    - Explain the concept of models: Python classes that define the structure of database tables.
    - Walk through the **models.py** file:
        - **UploadedImage**: Model for storing uploaded images.
        - **CriminalImage**: Model for storing images of criminals, including sketched and original images.
        - **Feature** and **FeatureOption**: Models related to facial features used in the sketching interface.
4. **Forms**:
    - Discuss the purpose of forms in web applications: to handle user input and perform validation.
    - Review the forms defined in **forms.py**:
        - **ImageForm**: Form for uploading images.
        - **CanvasImageForm**, **LoginForm**, and **UserForm**: Forms for various user interactions such as sketch saving, login, and registration.
5. **Views**:

- Explain the role of views: Python functions that handle HTTP requests and generate HTTP responses.
- Go through the view functions defined in **views.py**:
    - **home**, **login**, **logout**, **register**: Views for rendering HTML pages related to user authentication and homepage.
    - **sketch_interface**, **feature_detail**: Views for rendering pages related to the sketching interface and feature details.
    - **save_image**: View for saving images uploaded via the sketching interface.
    - **upload_sketch** and **sketch_match**: Views for uploading sketches and performing sketch matching.

6. **Templates**:
    - Introduce the concept of templates: HTML files with placeholders for dynamic content.
    - Review the templates in the **templates** directory:
        - **home.html**, **navbar.html**, **login.html**, **register.html**, **sketch_interface.html**, **sketch_success.html**, **sketch_match.html**, **sketch_upload.html**: Templates for different pages of the web application.
7. **Other Concepts**:
    - Briefly discuss additional concepts used in the project, such as image processing (e.g., face detection, feature extraction), machine learning (e.g., KNN algorithm), and authentication (e.g., login/logout).
8. **Demonstration**:
    - Show how to run the Django project locally using the development server (**python manage.py runserver**) and navigate through the web application.
    - Demonstrate the functionality of key features, such as user authentication, sketching interface, and sketch matching.

Face Sketching

The Face Sketching Application is a Django-based project aimed at providing users with a platform to create digital sketches of human faces. The application allows users to select various facial features, such as eyes, nose, lips, etc., and drag them onto a canvas to create custom face sketches.

Project Components:

1. **Feature Model:**
    - The **Feature** model represents different facial features available for sketching.
    - Each feature has a name and a unique slug field.
    - Example features include eyes, nose, lips, etc.
2. **FeatureOption Model:**
    - The **FeatureOption** model stores images associated with each facial feature.
    - Each feature option belongs to a specific feature and contains an image file.
3. **Views:**
    - **sketch_interface**: Renders the main interface where users can select facial features and create sketches.
    - **feature_detail**: Displays details of a specific feature along with its available options.
    - **save_image**: Handles the saving of the created sketches. It receives the canvas image data along with the desired image name, decodes the data, and saves it as a JPEG file on the server.
4. **Templates:**
    - **sketch_interface.html**: The main template for the sketching interface. It allows users to drag and drop feature options onto a canvas.
    - **sketch_success.html**: Displays a success message when a sketch is saved successfully.

Views Functions:

1. **sketch_interface:**
    - This view is responsible for rendering the main interface of the sketching application.
    - When a user navigates to the sketching interface page, this view is called.
    - It renders the **sketch_interface.html** template, which contains the canvas and other elements necessary for creating sketches.
    - This view does not require any additional data to be passed to the template.

2. **feature_detail:**
    - This view is called when a user wants to view the details of a specific facial feature.
    - It takes a **feature_slug** parameter, which is used to identify the feature for which details are requested.
    - The view retrieves the corresponding **Feature** object from the database based on the slug.
    - It then fetches all the **FeatureOption** objects associated with that feature.
    - Finally, it renders the **sketch_interface.html** template, passing the retrieved feature options and the feature itself to the template for display.
3. **save_image:**
    - This view handles the saving of the created sketches.
    - It expects a POST request containing the canvas image data (**canvas_image_data**) and the desired image name (**image_name**).
    - Upon receiving the POST request, the view decodes the base64-encoded canvas image data.
    - It then constructs a file path where the image will be saved, using the provided image name and a predefined directory path.
    - The decoded image data is then written to the file at the specified path, saving it as a JPEG file on the server.
    - If the saving process is successful, the view renders the **sketch_success.html** template, passing a success message to be displayed.
    - If an error occurs during the saving process, the view returns an HTTP response with an error message.

Login and Registration

1. **Login Form:**
    - A **LoginForm** class is defined using Django's **forms.Form** class.
    - It contains fields for **username** and **password**, where **password** field is rendered as a password input widget.
    - The form is used to collect user credentials for logging in.
2. **User Registration Form:**
    - A **UserForm** class is defined using Django's **forms.Form** class.
    - It contains fields for **username**, **password**, and **confirm password** (cpass).
    - Both password fields are rendered as password input widgets.
    - The form is used to collect user information for registration.
3. **Login Function:**
    - A **login** view function is defined to handle user login requests.
    - It checks if the request method is POST, then validates the login form data.
    - If the form is valid, it attempts to authenticate the user using Django's **auth.authenticate()** function.
    - If authentication is successful, the user is logged in using **auth.login()** and redirected to the home page.
    - If authentication fails, an error message is displayed, and the user is redirected back to the login page.
4. **Logout Function:**
    - A **logout** view function is defined to handle user logout requests.
    - It logs out the user using Django's **auth.logout()** function and redirects them to the home page.
5. **User Registration Function:**
    - A **register** view function is defined to handle user registration requests.
    - It checks if the request method is POST, then validates the registration form data.
    - If the form is valid and passwords match, it creates a new user using **User.objects.create_user()** and saves it to the database.
    - If the username is already taken or passwords don't match, appropriate error messages are displayed.
    - Upon successful registration, the user is redirected to the login page.

Uploading the image

The view function checks if the request method is POST. If it is, it means that the form has been submitted with data.

It creates a form instance (ImageForm) using the data from the POST request (if any) and the files uploaded with it.

It checks if the form is valid. If the form data passes all validation checks defined in the form's class, the sketch image is saved to the database using the form.save() method. The saved object instance is then passed to the template for rendering.

If the form is not valid, or if the request method is not POST, the view function renders the 'sketch_upload.html' template with the form instance to allow the user to upload a sketch image.

Extracting features of input image

UploadedImage.objects.last(): This retrieves the most recent UploadedImage object from the database. UploadedImage is a Django model representing images uploaded by users.

input_image.image.path: This accesses the image field of the retrieved UploadedImage object and retrieves the filesystem path to the associated image file.

cv2.imread(image_path): This function from the OpenCV library reads the image file located at image_path and returns a NumPy array representing the image. The array contains the pixel values of the image.

Face Detection

Face detection on the provided sketch image using the MTCNN (Multi-task Cascaded Convolutional Networks) model.

MTCNN (Multi-Task Cascaded Convolutional Neural Network) is a deep learning model used for face detection and alignment. It is designed to efficiently detect faces in images and accurately locate facial landmarks such as the eyes, nose, and mouth.

cv2.cvtColor(sketch_image, cv2.COLOR_BGR2RGB): Converts the BGR (Blue-Green-Red) image format to RGB (Red-Green-Blue) format, which is required by the MTCNN model.

(The BGR format is the default color format used by OpenCV when reading and processing images. However, in many other contexts, such as in graphics software and web development, the more common RGB (Red-Green-Blue) format is used.)

detector.detect_faces(image_rgb): Detects faces in the RGB image using the MTCNN model. It returns a list of dictionaries, where each dictionary contains information about a detected face, such as its bounding box coordinates.

Face Cropping: Iterates over the detected faces and extracts each face region from the original image. It calculates the bounding box coordinates (x, y, width, height) of each face, ensuring that the bounding box stays within the image boundaries. Then, it crops the face region from the original image.

Returns the cropped face images as a list.

Feature Extraction

Pre-trained VGG16 model is loaded using Keras. VGG16 is a convolutional neural network architecture that is commonly used for image classification tasks.

1. **Loading VGG16 Model**: The **VGG16** function is used to load the VGG16 model. The **weights** parameter specifies the path to the pre-trained weights file, which is typically provided by the Keras library. These weights are learned during the training process on large datasets such as ImageNet. The **include_top** parameter is set to **True**, indicating that the fully connected layers (top layers) of the model will be included. This allows the model to be used for feature extraction.
2. Extracting Features: After loading the model, the code prepares the input image for feature extraction. The image is loaded using image.load_img and resized to the target size of (224, 224) pixels, which is the input size expected by the VGG16 model. Then, the image is converted to a NumPy array using image.img_to_array. The array is expanded to include an additional dimension using np.expand_dims to match the input shape expected by the VGG16 model. Finally, the pixel values are preprocessed using preprocess_input function to ensure compatibility with the pre-trained weights.
3. The preprocessed image is passed through the VGG16 model using the **predict** function, which returns the extracted features.

Feature extraction of whole dataset

- The function **extract_features_dataset** takes two parameters: **criminal_images**, which is a queryset containing instances of the **CriminalImage** model, and **features_file**, which is the path to the file where the extracted features will be saved.
- Inside the function, an empty list **features_array** is initialized to store the extracted features of each sketch image.
- For each image instance in **criminal_images**, the path to the sketch image file is retrieved using **image_instance.sketched_image.path**.
- The sketch image is read using OpenCV's **cv2.imread** function, and then face detection is performed on the image using the **face_detect** function.
- The cropped face image is saved to a temporary file using **cv2.imwrite**.
- Features are extracted from the cropped face image using the **extract_features** function.
- The extracted features are appended to the **features_array**.
- After processing all images in the dataset, the **features_array** is converted to a NumPy array and saved to a file using **np.save**.
- The function returns the features_array, which contains the extracted features of all sketch images in the dataset.

Finding the Match

1. Initializing the KNN model: A KNN model is created with k neighbors and the Euclidean distance metric.
2. Flattening the dataset features: The features of the dataset images are flattened to be compatible with the KNN model.
3. Fitting the model: The KNN model is trained on the flattened dataset features.
4. Finding nearest neighbors: The KNN model is used to find the k nearest neighbors (closest images) to the features of the input image.
5. Extracting the best match index: The index of the best matching image is extracted from the indices of the nearest neighbors.
6. Fetching the matching image from the database: The best_match_index is used to retrieve the path or URL of the best matching image from the CriminalImage model.

Calculating Match Score

1. Reshaping the features: The features of the input image (image_features) and the features of the best matching image from the dataset (dataset_features[best_match_index]) are reshaped into 2D arrays with one row and multiple columns. This is done to ensure that they have a compatible shape for computing cosine similarity.
2. Calculating cosine similarity: The cosine_similarity function from scikit-learn is used to compute the cosine similarity between the two feature vectors. Cosine similarity measures the cosine of the angle between two vectors and gives a value between -1 and 1, where 1 indicates perfect similarity.
3. Converting similarity to percentage: The similarity score obtained from cosine similarity is multiplied by 100 to convert it into a percentage value. This represents the percentage similarity between the input image and the best matching image.

Templates

1. home.html: This is the landing page of your application. It likely contains some introductory content and a button labelled "Identify Sketch". When the user clicks this button, it redirects to the sketch_upload.html page.
2. sketch_upload.html: This page allows users to upload an image. It contains a form with a file input field for uploading the image. Once the user selects an image and submits the form (by clicking the "Find the Match" button), it redirects to the sketch_match.html page.
3. sketch_match.html: This page displays the input image, the matched image, and the similarity score. It receives this information from the backend (Django views) after processing the uploaded image. The similarity score represents how closely the uploaded sketch matches the identified image.