Content-Based Image Retrieval

Ebrahim Golriz

Supervisor: Dr. Asadollah Shahbahrami

Spring 2024

Content-Based Image Retrieval (CBIR) is a rapidly evolving field that aims to bridge the gap between human perception and machine understanding of digital images. With the advent of deep learning techniques, such as Convolutional Neural Network (CNN) computer vision has witnessed a shift in its ability to extract meaningful features and representations from visual data. This project explores the synergistic integration of CBIR and deep learning methodologies for various computer vision tasks.

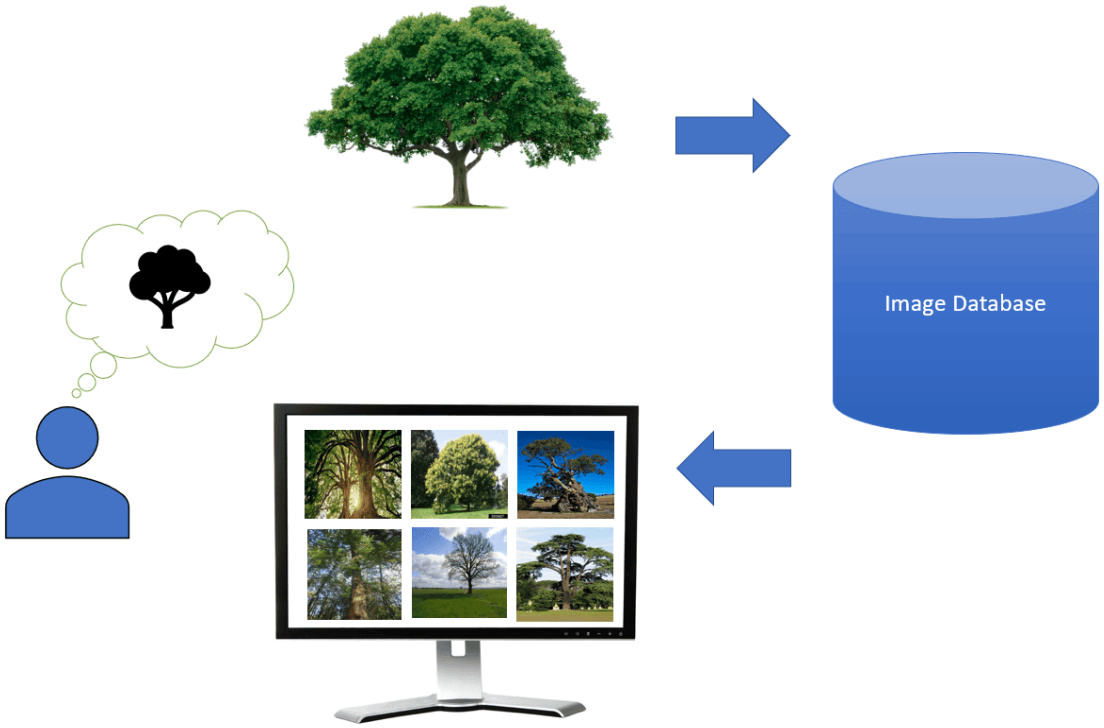
Traditional text-based image retrieval methods often fall short in capturing the rich semantic information contained within visual data, necessitating the development of more sophisticated techniques [17]. CBIR systems aim to improve computer vision by analyzing the natural visual features of images, such as color, texture, and shape, to facilitate effective image retrieval and management.

However, the complexity and diversity of visual content pose significant challenges for CBIR systems, limiting their performance and scalability. This is where machine learning, particularly deep learning, has revolutionized the field of computer vision. Deep learning architectures, with their ability to automatically learn hierarchical representations from raw data, have demonstrated remarkable success in various computer vision tasks, including image classification, object detection, and semantic segmentation [7].

In this project, we explore the integration of CBIR and deep learning techniques to develop a robust and efficient system for recognizing and retrieving images of multiple celebrities. Leveraging the power of deep convolutional neural networks (CNNs), our model is trained on a dataset of celebrity faces, enabling it to learn discriminative features and representations that capture the unique characteristics of each individual.

The project also highlights the challenges and limitations encountered, such as the need for large-scale datasets and the computational complexity associated with training deep neural networks.

Contents



What is CBIR?

Content-Based Image Retrieval (CBIR) is an approach that focuses on analyzing the visual content of images rather than relying on metadata or textual annotations. The term "content-based" signifies that the search process examines the natural features of the image itself, such as colors, shapes, textures, or any other information that can be derived from the image data. In contrast, Text-Based Image Retrieval (TBIR) systems retrieve images based on user-specified text or descriptions associated with the images, matching them against the assigned keywords, tags, or descriptions [1].

The overall structure of TBIR is shown in Figure 1.

While TBIR has been widely used, it has several limitations. Firstly, it relies on manual annotation, where humans assign textual descriptions, keywords, or tags to images. This process is time-consuming, subjective, and prone to inconsistencies, especially for large-scale image databases. Additionally, textual annotations can only capture a limited amount of information about an image, failing to adequately represent visual information such as color, texture, shape, and spatial relationships. Furthermore, TBIR systems are language-dependent, which can be a barrier when dealing with multilingual or cross-lingual image databases, as translating annotations across languages can introduce additional challenges and potential inaccuracies. As image databases continue to grow in size, manually annotating each image becomes increasingly impractical and time-consuming, limiting the scalability of TBIR systems [17] [1].

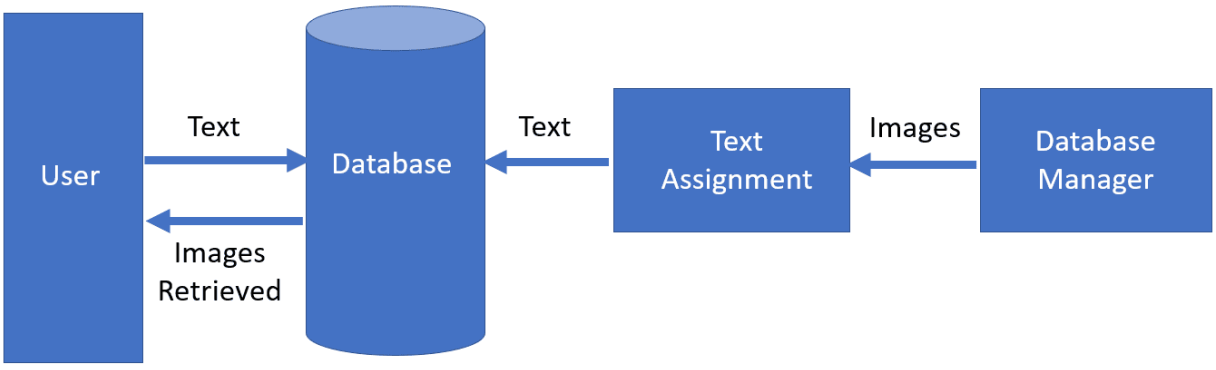


Figure 1: Text-Based Image Retrieval [17]

In contrast, CBIR offers several advantages over traditional TBIR approaches. One of the primary benefits is automated annotation, where CBIR systems can automatically extract visual features from images, eliminating the need for manual annotation. This process is objective and consistent, ensuring unbiased retrieval results. Additionally, CBIR systems can handle complex queries that may be difficult to express in text, such as finding images with specific shapes, textures, or color combinations.

CBIR techniques can also be tailored to specific domains, such as medical image analysis, remote sensing, and biometric recognition, enabling more accurate and efficient retrieval in specialized applications. Furthermore, CBIR opens up new possibilities for applications that rely heavily on visual content, such as image search engines, visual recommender systems, and content-based video analysis.

Content-Based Image Retrieval (CBIR) has a wide range of applications across various domains:

Digital Libraries and Multimedia Databases: CBIR is widely used in digital libraries, art galleries, and multimedia databases to organize and search for images based on their visual content. Users can query the system by providing an example image or specifying desired features, and the system retrieves similar images from the database. This enables efficient browsing, exploration, and retrieval of relevant visual information from large collections.

Medical Image Analysis: CBIR plays a crucial role in medical imaging applications, assisting in diagnosis and treatment planning. Physicians can search for similar medical images (e.g., X-rays, MRI scans, CT scans) based on visual features, which can help in identifying patterns and anomalies. By comparing a patient's images with similar cases, CBIR can aid in disease detection, treatment planning, and decision-making processes.

Social Media and Image Sharing Platforms: CBIR can be integrated into social media and image sharing platforms to enable users to search for similar images based on visual features, facilitating image discovery and organization. This can enhance user experience by providing relevant and visually similar content recommendations, improving content creation, and enabling more effective image management.

Face Detection and Recognition: Face detection and recognition are prominent use cases of CBIR techniques. Face detection involves identifying and locating human faces within digital images or video streams, while face recognition aims to match detected faces with known identities. These applications are widely used in biometrics, security systems, and various multimedia applications.

One of the key challenges in CBIR algorithms is addressing the **semantic gap** between the high-level meaning of an image and its low-level visual features. As depicted in Figure 2 CBIR algorithms typically start with simple low-level features, such as

color, texture, and shape, but bridging the gap to understand the semantic content of an image is a complex task [2]. Researchers have dedicated significant efforts to developing techniques that can effectively bridge this semantic gap and improve the retrieval accuracy of CBIR systems.

Generally, methods for reducing the semantic gap fall into two categories: those that combine low-level features, and those that employ machine learning techniques such as object recognition [3].

Relying solely on low-level features is insufficient to fully capture the visual and perceptual content of an image. However, these methods do offer the advantage of lower computational complexity.

To further narrow the semantic gap, some researchers have developed hybrid approaches that combine both low-level features and machine learning techniques. These methods often incorporate relevance feedback, creating an interactive interface between the user and the system. In this process, the system retrieves images, and the user identifies the correct matches. Machine learning algorithms then use this feedback to improve subsequent retrieval results [3].

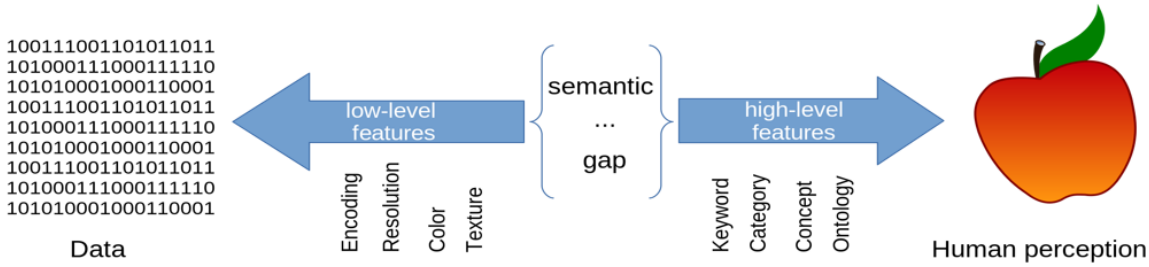


Figure 2: Semantic Gap [2]

Content-Based Image Retrieval systems, as shown in Figure 3 typically consist of three main components: feature extraction, indexing, and similarity measurement. Feature extraction involves identifying and extracting relevant visual features from the images, such as color histograms, texture descriptors, or shape representations. Indexing organizes the extracted features in a structured manner, enabling efficient storage and retrieval. Similarity measurement compares the features of the query image with those of the images in the database, using appropriate distance or similarity metrics, to rank and retrieve the most relevant images [1].

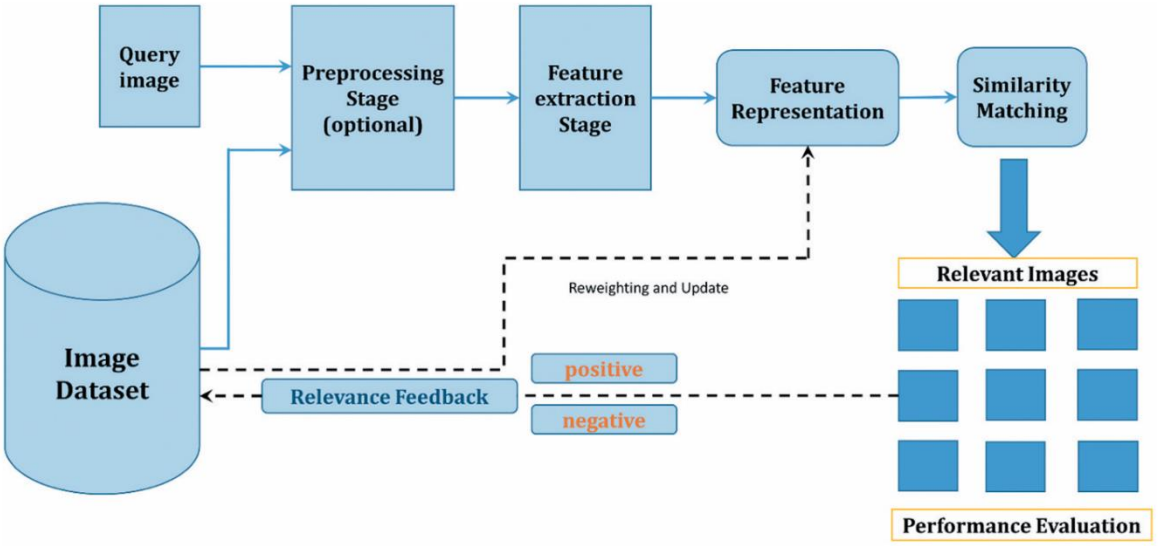


Figure 3: Architecture of a CBIR system [1]

Feature Extraction is the main component of Content-Based Image Retrieval (CBIR) systems, as it involves identifying and extracting relevant visual features from the images. There are two main types of features: global features and local features [1].

Global Features:

Figure 4 shows some examples of global features that describe the entire image and contain information about the overall visual characteristics. Several descriptors characterize color spaces, such as color moments and color histograms. Other global features are concerned with visual elements like shapes and textures. While global features have the advantage of being

computationally efficient and providing a general representation of the image, they may be sensitive to scaling, rotation, and other transformations [3].

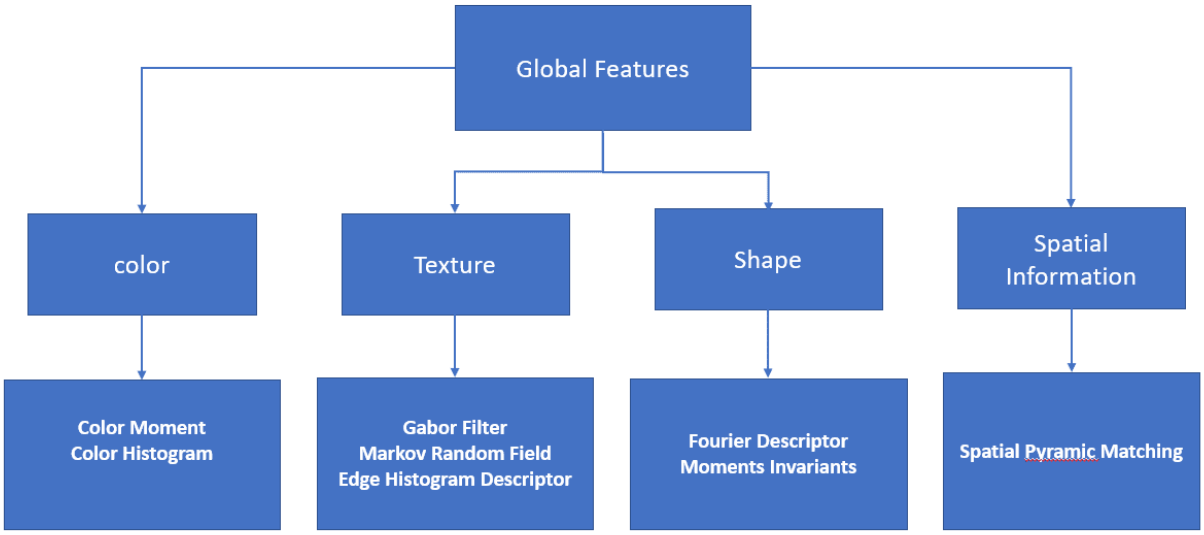


Figure 4: Global Features [17]

Local Features:

In contrast, local features describe visual patterns or structures identifiable in small groups of pixels, such as edges, points, and various image patches. Examples of local feature descriptors include Scale-Invariant Feature Transform (SIFT) and Local Binary Patterns (LBP) [3], which is depicted in Figure 5. Local features are more robust to scaling, rotation, and other transformations, making them more reliable in various conditions. However, they often require more computational

resources and may not capture the overall context of the image as effectively as global features.

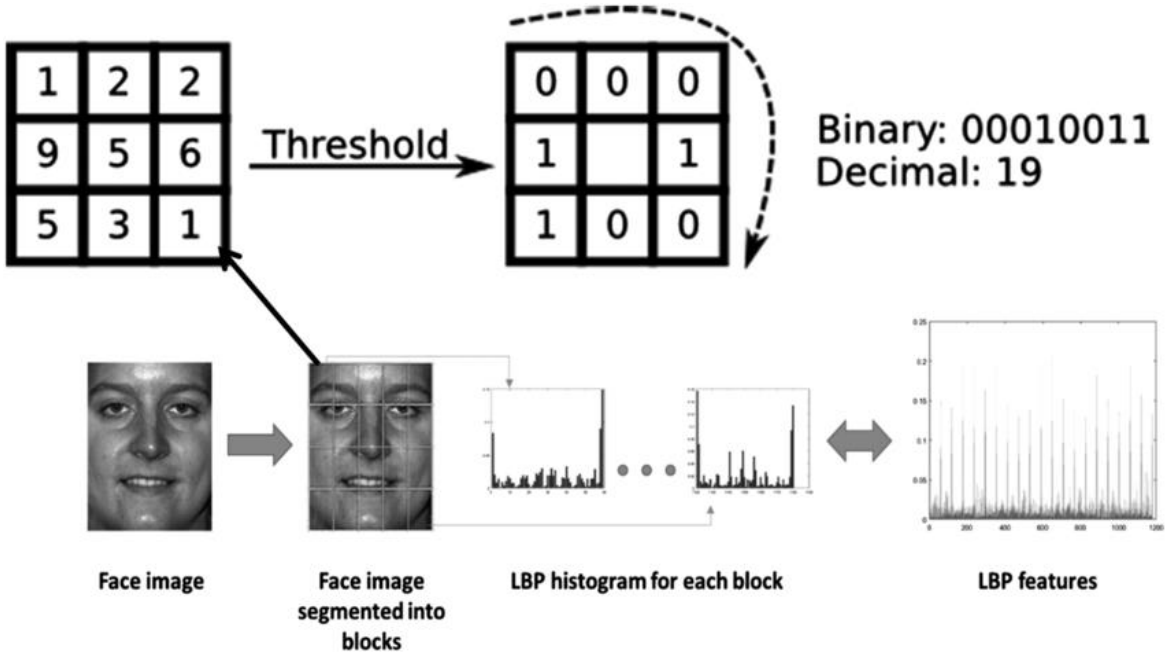


Figure 5: Local Binary Patterns (LBP), an example of local features [14]

Feature Vectors:

Once the relevant features are extracted, each image is represented as a point in an n-dimensional feature space, where n is the number of features extracted. Figure 6 shows an example of a three-dimensional feature space. The coordinates of this point are the values of the n features. The goal of CBIR systems is to find images in the database that are "closest" or

most similar to the query image in this high-dimensional feature space [7].

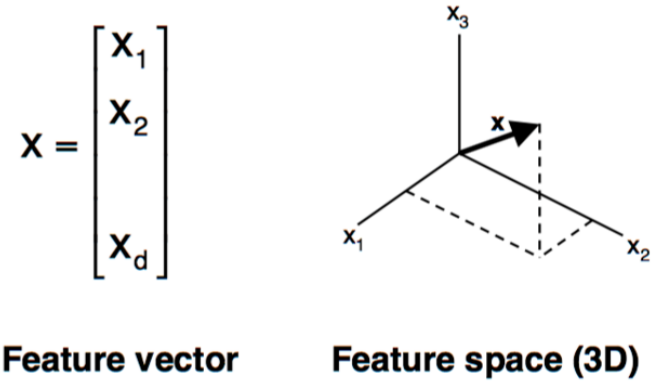


Figure 6: Feature Vector [18]

Similarity Measures:

Similarity measures play a crucial role in quantifying how similar a database image is to the query image. The selection of the appropriate similarity measure is a challenging task, as it can significantly impact the retrieval accuracy. There are two types of similarity measures: distance measures and similarity metrics [1].

Distance Measures:

Distance measures quantify the dissimilarity between two feature vectors by calculating the distance between them in a

metric space. The most similar images to the input image are those with the smallest distances. Several distance functions, as shown in Figure 7, can be used, such as Manhattan distance, Euclidean distance, and Mahalanobis distance [1].

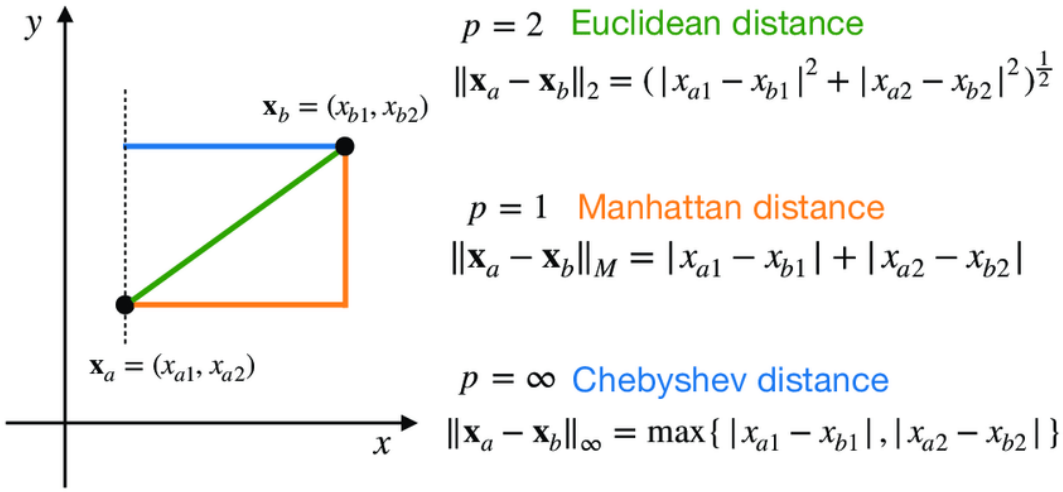


Figure 7: Distance Measures [15]

Similarity Metrics:

Similarity metrics, on the other hand, quantify the similarity between two feature vectors directly. The higher the value of the similarity metric, the more similar the images are. Examples of similarity metrics include cosine similarity, which is shown in Figure 8 and correlation coefficients [1].

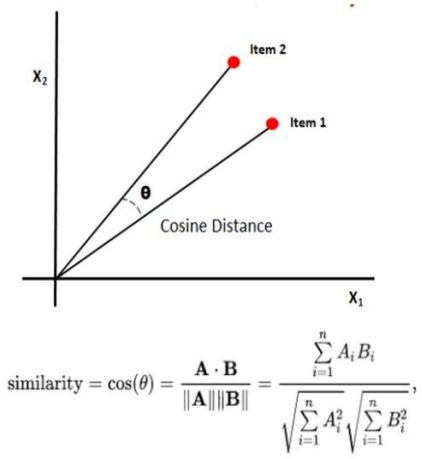


Figure 8: Cosine Similarity [16]

It is crucial to choose the appropriate feature extraction techniques and similarity measures based on the specific application domain and the characteristics of the image dataset. Additionally, advanced techniques such as relevance feedback and machine learning algorithms can be employed to improve the retrieval accuracy and adapt to user preferences over time.

Machine learning

Recently, CBIR systems have been shifted toward using machine learning algorithms to obtain a model that can deal with new input data and give correct prediction, which will improve the image search. This shift has led to the utilization of various machine learning techniques, including unsupervised learning, supervised learning, and deep learning [7].

Unsupervised Learning (Clustering)

Unsupervised learning, specifically clustering, is used in CBIR systems to group similar images together based on their visual features, without relying on predefined labels or categories. Clustering algorithms analyze the inherent patterns and similarities within the data to form clusters of related images. A common clustering algorithm used in CBIR is k-means clustering. These algorithms help organize and navigate large image databases by grouping visually similar images together, enabling efficient browsing and retrieval.

Clustering is a data mining technique for grouping unlabeled data based on their similarities or differences [5]. For example, K-means clustering algorithms assign similar data points into groups, where the K value represents the size of the grouping [19]. This technique is helpful for market segmentation, image compression, and so on. Figure 9 shows a general depiction of the k-means clustering algorithm.

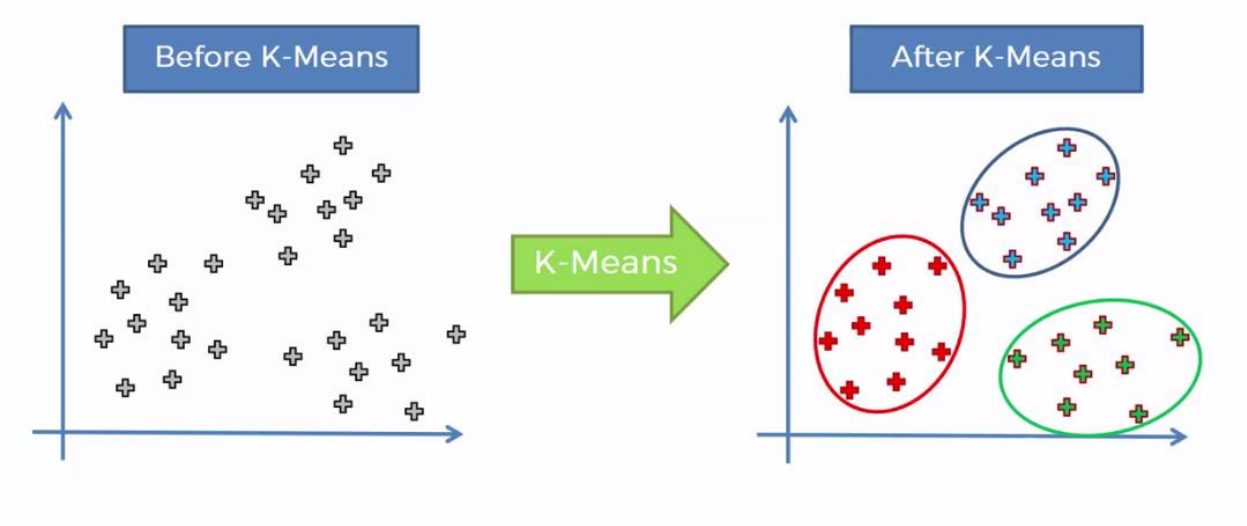


Figure 9: k-means clustering [19]

Supervised Learning (Classification)

Supervised learning techniques are employed in CBIR systems for image classification tasks. A labeled dataset of images is required, where each image is associated with a specific class or category (e.g., landscapes, animals, buildings, etc.). Supervised learning algorithms, such as Support Vector Machines (SVMs), shown in Figure 10, Convolutional Neural Networks (CNNs), or decision trees, are trained on this labeled dataset to learn the mapping between image features (e.g., color, texture, shape) and the corresponding labels. Once trained, these models can classify new, unseen images into the appropriate categories, enabling accurate and efficient image retrieval based on semantic labels [1].

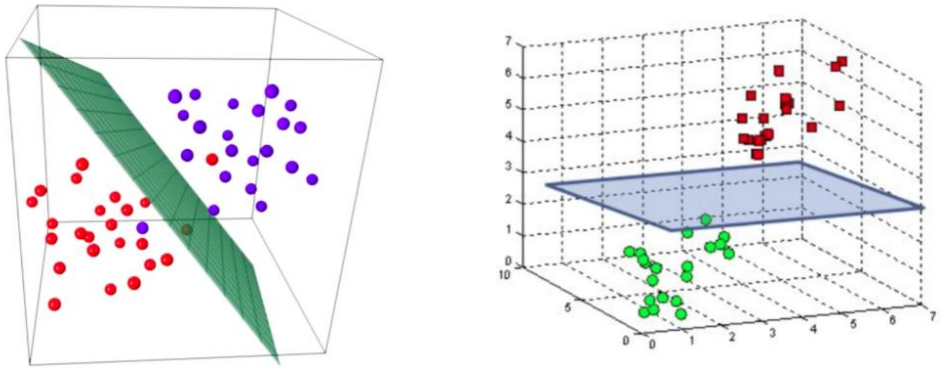


Figure 10: Support Vector Machine (SVM) [20]

Maximizing the margin between classes is a fundamental principle in supervised machine learning, particularly in the context of Support Vector Machines (SVMs). The margin is a measure of the separation between different classes in a classification problem, and maximizing it is crucial for achieving good generalization performance and robustness to noise or outliers [6].

In a binary classification problem, the goal is to find a decision boundary (hyperplane) that separates the two classes as cleanly as possible. The margin is defined as the distance between the decision boundary and the closest data points from each class, known as support vectors. Figure 11 shows Margin between two classes. This process can be observed in [21].

As discussed in [6], the idea behind maximizing the margin is to find the decision boundary that provides the maximum separation

between the classes. This is achieved by maximizing the distance between the decision boundary and the support vectors from each class.

A larger margin implies that the decision boundary is more robust to noise or outliers, as it is less likely to be influenced by individual data points that may be mislabeled or lie close to the boundary. A larger margin leads to better generalization performance, meaning that the classifier is more likely to perform well on unseen data. This is because the decision boundary is less influenced by individual data points and captures the underlying structure of the data more effectively.

If there is an outlier data point within a class that lies closer to the decision boundary or even in between the two classes, the SVM algorithm will essentially ignore or give less importance to that outlier in order to maximize the margin between the classes.

SVMs employ a technique called soft-margin optimization. In this approach, the SVM algorithm allows for some data points to be misclassified or to lie within the margin region, but it penalizes these misclassifications or margin violations. The goal is to find the decision boundary that maximizes the margin while minimizing the number of misclassifications or margin violations [6].

By allowing for some margin violations and misclassifications, the SVM can effectively ignore or give less importance to outliers or noisy data points that lie close to or within the region between the two classes. This way, the decision boundary is primarily

determined by the majority of the data points, resulting in a larger margin and better generalization performance.

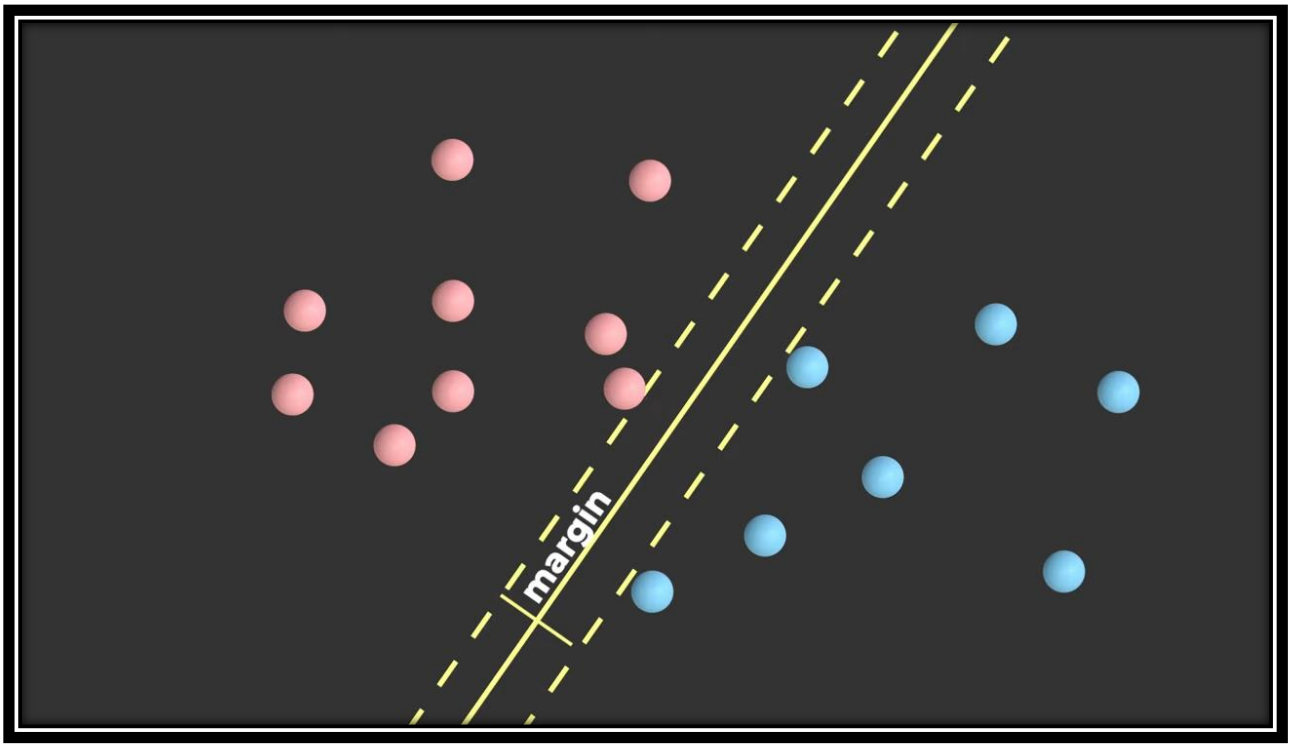


Figure 11: Margin between two classes [21]

The main distinction between the two approaches is the use of labeled data sets. Supervised learning uses labeled input and output data, while an unsupervised learning algorithm does not.

In supervised learning, the algorithm learns from a training dataset by iteratively making predictions on the labeled data and adjusting its internal parameters to minimize the discrepancy between its predictions and the correct labels. Supervised learning models tend to achieve higher accuracy compared to

unsupervised learning models, but they require upfront human effort to label the data appropriately.

On the other hand, unsupervised learning models work on unlabeled data to discover inherent patterns and structures without relying on predefined labels. However, some human intervention is still required to validate and interpret the output variables or clusters identified by the model [5].

Figure 12 Provides the overall comparison between supervised and unsupervised learning

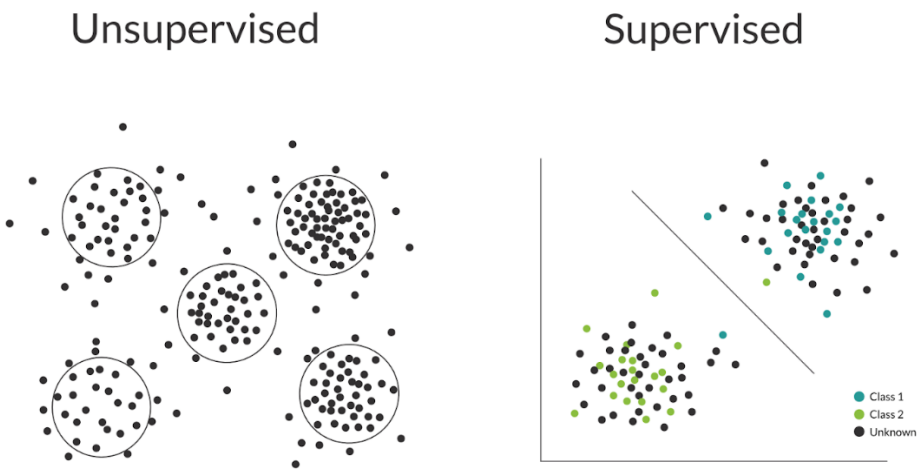


Figure 12: Supervised and Unsupervised learning [22]

Deep learning

Deep learning, a subset of machine learning, has gained significant attention in the past decade for solving real-world problems. Inspired by the human brain, deep learning architecture consists of algorithms that process information through multiple stages of representation and transformation. Its success in various fields, including object recognition, has led to its application in Content-Based Image Retrieval (CBIR) to address the semantic gap. Deep learning can map input data to output data without relying on human-provided features. The main types of deep learning algorithms include Convolutional Neural Networks (CNN), Deep Neural Networks (DNN), deep belief networks, and Boltzmann machines. Among these, CNNs have shown exceptional performance in computer vision applications and specifically in CBIR [1].

Since the 1950s, AI researchers have tried to develop systems that can understand visual data, leading to the establishment of the field known as Computer Vision. A pivotal moment in this journey occurred in 2012, when researchers from the University of Toronto developed AlexNet, an AI model that significantly outperformed previous image recognition algorithms. AlexNet, created by Alex Krizhevsky, won the 2012 ImageNet contest with 85% accuracy, significantly outperforming the runner-up's 74%. This success was driven by CNNs [29].

Throughout recent years, Convolutional Neural Networks (CNNs) have emerged as essential tools in various computer vision applications, including image classification, object detection, and image segmentation. Contemporary CNNs are typically implemented using popular programming languages such as Python and employ sophisticated techniques to extract and learn features from visual data. The effective training of these models relies heavily on careful selection of hyperparameters, optimization strategies, and regularization methods [29].

Following the breakthrough of AlexNet, numerous advancements and innovative architectures like VGG, ResNet, and EfficientNet have been introduced, expanding the capabilities of CNNs. In the present day, CNNs play a crucial role in a wide range of applications, spanning from autonomous vehicle systems to medical imaging analysis [29]. Some common applications of this computer vision can be seen in [9]:

- **Marketing:** Social media platforms provide suggestions on who might be in photograph that has been posted on a profile, making it easier to tag friends in photo albums.
- **Healthcare:** Computer vision has been incorporated into radiology technology, enabling doctors to better identify cancerous tumors in healthy anatomy.

- **Retail:** Visual search has been incorporated into some e-commerce platforms, allowing brands to recommend items that would complement an existing wardrobe.
- **Automotive**: While the age of driverless cars hasn't quite emerged, the underlying technology has started to make its way into automobiles, improving driver and passenger safety through features like lane line detection.

Artificial Neural Networks (ANNs) are computational systems inspired by the way biological nervous systems, like the human brain, operate. ANNs consist of numerous interconnected computational nodes, or neurons, which work together in a distributed manner to learn from input data and optimize the final output. These networks mimic the brain's learning process by adjusting connections between neurons based on the input they receive, enabling them to recognize patterns and make decisions.

The basic structure of an ANN, which is depicted in Figure 18, involves loading input data, typically a multidimensional vector, into the input layer, which then distributes the data to the hidden layers. The hidden layers process the data, making decisions and adjustments to improve the final output—a process known as learning. When multiple hidden layers are stacked, this configuration is referred to as deep learning, allowing the network to model complex patterns and representations. This structure enables ANNs to excel in various tasks, such as image and speech

recognition, by continuously refining their performance through iterative learning [8].

Convolutional Neural Networks (CNNs)

Deep learning, particularly **Convolutional Neural Networks (CNNs)**, has emerged as a powerful technique for CBIR systems. CNNs are a type of deep neural network architecture specifically designed for processing image data. They automatically learn hierarchical representations of visual features, from low-level features like edges and shapes to high-level semantic concepts. Pre-trained CNNs, such as VGGNet, ResNet, or Inception, can be fine-tuned on specific image datasets or used as feature extractors for CBIR tasks. Deep learning models have shown remarkable performance in image classification, object detection, and retrieval tasks, outperforming traditional hand-crafted feature extraction methods [1].

CNN architecture is inspired by the human visual system and does not require manual feature extraction. In this model, artificial neurons correspond to biological neurons, while CNN kernels act as various receptors that respond to different features. The activation functions in CNN mimic the biological process where only neural signals exceeding a certain threshold are transmitted to the next neuron. This design allows CNN to automatically learn and process visual information in a manner similar to human perception [9].

Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. The general structure of CNNs is depicted in Figure 13. They have three main types of layers, which are [8]:

- Convolutional layer
- Pooling layer
- Fully-connected (FC) layer

The convolutional layer serves as the initial layer in a convolutional network. While subsequent layers can include additional convolutional or pooling layers, the network typically concludes with a fully-connected layer. As the CNN progresses through its layers, it becomes increasingly complex, allowing it to identify larger parts of the image. The earlier layers concentrate on basic features like colors and edges, while deeper layers gradually recognize more complex elements and shapes. This hierarchical processing continues until the network can ultimately identify the target object in its entirety.

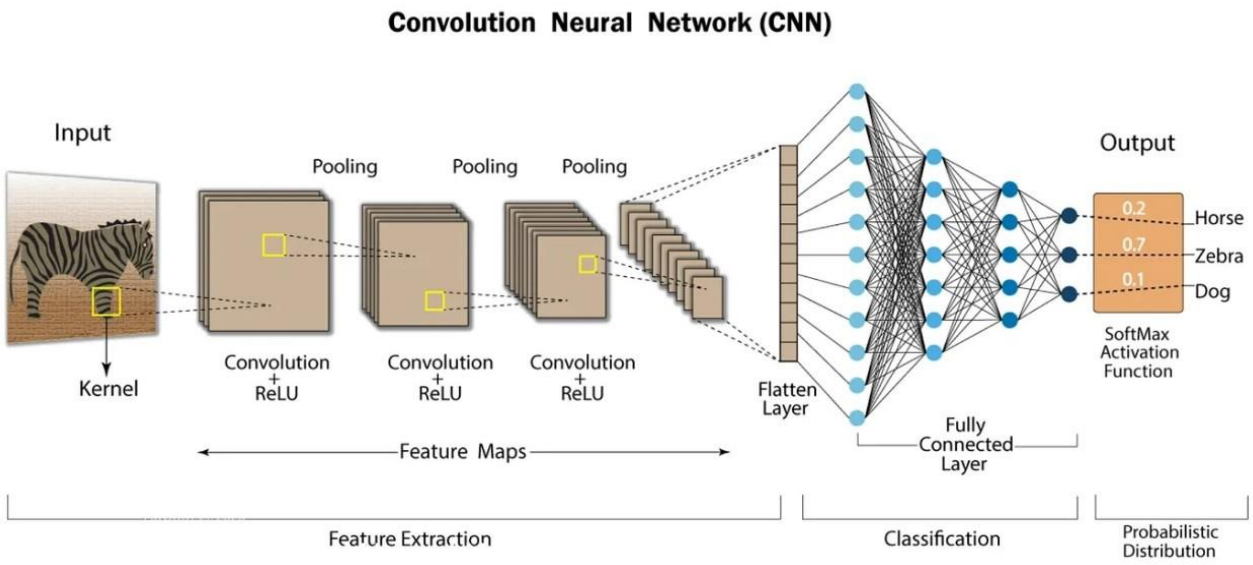


Figure 13: Convolutional Neural Networks (CNNs) [23]

Convolutional layer

The convolutional layer is where most of the heavy computations occurs in a CNN. It requires three main components: input data, a filter (or kernel), and a feature map. Let's assume the input is a color image, which is a 3D matrix of pixels representing the height, width, and depth (RGB channels). The filter, or feature detector, is a 2D array of weights that represents a specific pattern or feature in the image. This filter moves across the receptive fields of the input image, checking for the presence of that feature through a process called convolution [8].

As it can be seen in Figure 14 , The filter is typically a 3x3 matrix, which represents part of the image. While they can vary in size, the filter size is typically a 3x3 matrix; It is applied to an area of the input image, and a dot product is calculated between the input pixels and the filter weights. This dot product is then stored in an

output array, known as a feature map. The filter then shifts by a stride (the number of pixels it moves) and repeats the process until it has swept across the entire image.

the weights in the feature detector remain fixed as it moves across the image, which is also known as parameter sharing. Some parameters, like the weight values, adjust during training through the process of backpropagation and gradient descent [33].

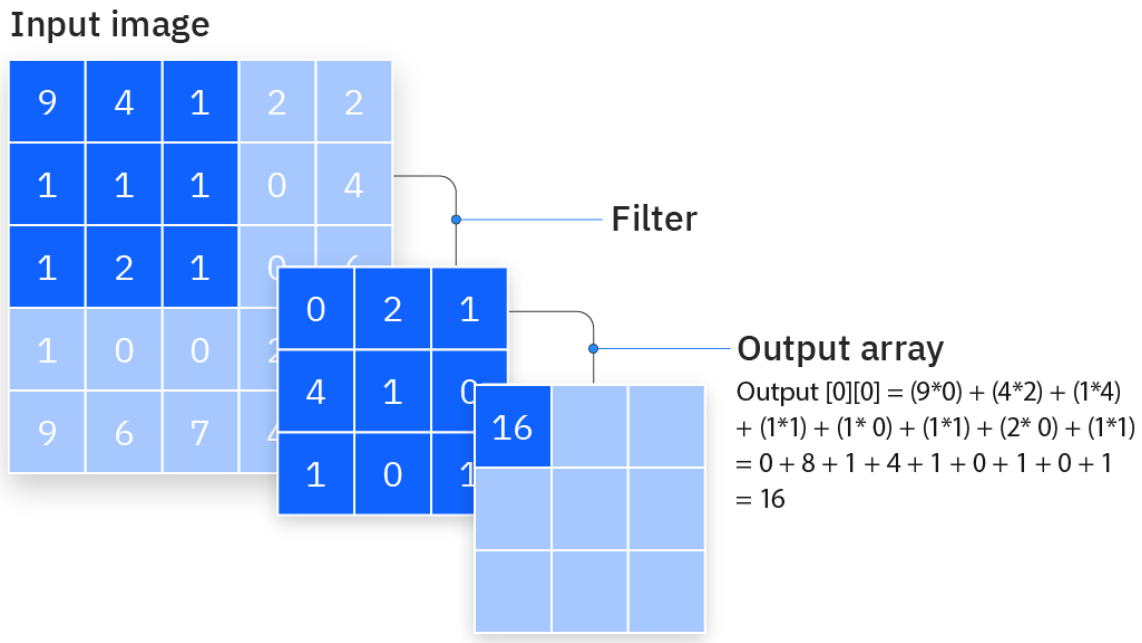


Figure 14: Convolution [24]

The **number of filters** affects the depth of the output. For example, three distinct filters would yield three different feature maps, creating a depth of three.

Stride is the distance, or number of pixels, that the kernel moves over the input matrix. While stride values of two or greater is rare, a larger stride yields a smaller output.

Zero-padding is usually used when the filters do not fit the input image. This sets all elements that fall outside of the input matrix to zero, producing a larger or equally sized output.

After the convolution operation between the input image and the filter, the resulting feature map contains the dot product values for each receptive field. However, these values can be negative or positive, and they are essentially linear transformations of the input data.

In deep learning models, especially in CNNs, it is crucial to introduce non-linearity to the network. This is because linear functions, when composed together, still result in a linear function, which can limit the model's ability to learn complex patterns and representations from the data [9].

The Rectified Linear Unit (ReLU) is a non-linear activation function that is applied to the feature map after each convolution operation. The ReLU function will output the value x if it is positive, and 0 if it is negative.

By applying the ReLU function, the linear outputs from the convolution operation are transformed into non-linear representations. This non-linearity allows the CNN to learn more complex patterns and representations from the data.

Additional convolutional layers

Additional convolutional layers serve several purposes, enhancing the model's ability to understand and represent complex features within the data. Figure 15 shows that each convolutional layer in a CNN learns to detect features from the input data. As the network gets deeper, with more convolutional layers, the features learned become progressively more complex and abstract [24].

The receptive field of a neuron in a convolutional layer refers to the region of the input data that affects the neuron's output. As we stack more convolutional layers, the receptive field increases, allowing the network to consider a larger context of the input data.

In traditional ANNs, each neuron is fully connected to all inputs, which becomes problematic for large inputs like images, resulting in models that are too large and inefficient to train. CNNs address this issue by using convolutional layers, where each neuron connects only to a small, localized region of the input, known as its receptive field. This approach significantly reduces the number of connections and parameters, making the model more manageable and effective for image processing tasks. The depth of these connections typically matches the depth of the input, allowing the network to process all channels of the image data [8].

"*For example, if the input to the network is an image of size 64 ×64 ×3(a RGB-colored image with a dimensionality of 64 ×64)*

and we set the receptive field size as 6×6, we would have a total of 108 weights on each neuron within the convolutional layer. (6×6×3where 3is the magnitude of connectivity across the depth of the volume) To put this into perspective, a standard neuron seen in other forms of ANN would contain 12,288 weights each" [8]

Additional layers help in abstracting the input data into more meaningful representations. This abstraction is crucial for tasks such as image classification, where the goal is to recognize high-level concepts rather than low-level details.

- **Low-Level Features:** Edges, colors, textures.
- **Mid-Level Features:** Shapes, parts of objects.
- **High-Level Features:** Objects, scenes, specific categories (like faces).

Figure 15: Convolution Layers [24]

Pooling layer

Pooling layers play a crucial role in CNNs by performing a form of downsampling. Their primary function is to progressively reduce the spatial dimensions (width and height) of the input representation. This reduction in size leads to fewer parameters and computations in the network, which in turn helps to control overfitting and improve the efficiency of the model [8].

One of the key benefits of pooling layers is their ability to make the network's output more robust to small translations or rotations in the input image. This is achieved by summarizing the features present in a region of the feature map. For instance, if a particular feature is detected in a slightly different location in two similar images, the pooling operation can help ensure that this feature is represented similarly in both cases.

There are two main types of pooling [24] [25]:

- **Max pooling:** As the filter moves across the input, it selects the pixel with the maximum value to send to the output array. As an aside, this approach tends to be used more often compared to average pooling.

  Figure 16 illustrates an example of max pooling.

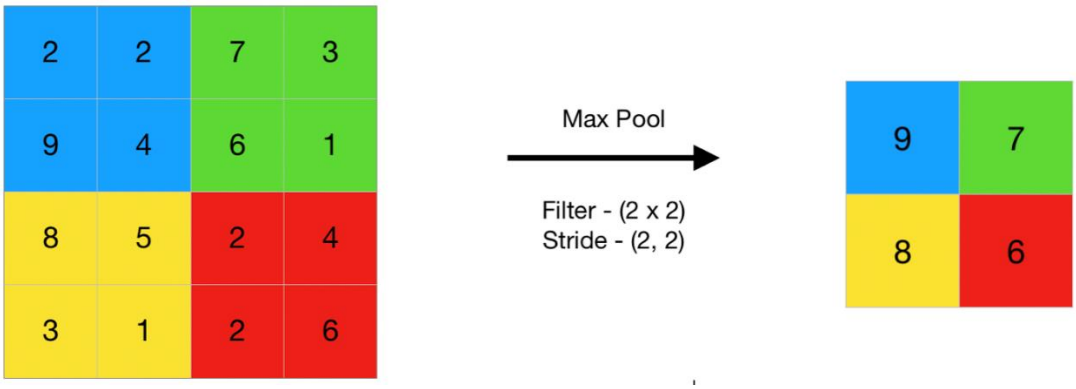


Figure 16: Max Pooling [25]

- **Average pooling:** As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

  Figure 17 illustrates an example of average pooling.

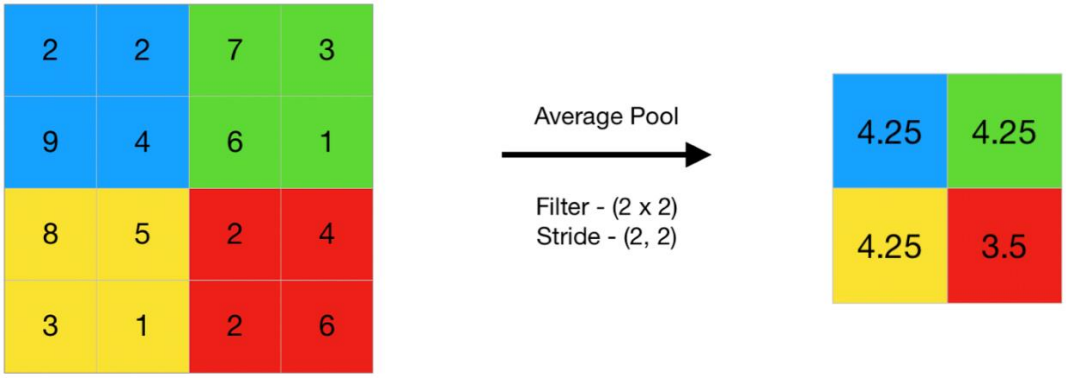


Figure 17: Average Pooling [25]

Fully-connected layer

As its name suggests and can be observed from Figure 18, in this layer, each neuron is connected to every neuron in the previous layer, much like traditional forms of ANN .

The primary purpose of the Fully-connected layer is to take the high-level features learned by the convolutional layers and their downsampled versions from pooling layers, and combine them to create a final set of features. These features are then used to classify the input into various categories or to perform other complex tasks, depending on the network's objective [24].

For example, in a CNN used for image classification, the convolutional layers might learn to detect edges, shapes, and more complex patterns, while the Fully-connected layers learn how to combine these features to classify the entire image. The FC layer essentially acts as a classifier on top of these extracted features.

The Fully-connected layer flattens the 2D or 3D output from the previous layers into a 1D vector. This allows the network to operate on a simple vector of features, rather than on the complex structures output by convolutional layers. Each neuron in the FC layer computes a weighted sum of all inputs from the previous layer, applies a non-linear activation function, and passes the result to the next layer or as the final output.

One important characteristic of Fully-connected layers is that they contain the majority of the parameters in a CNN. While this gives them significant learning capacity, it also makes them prone to

overfitting [8], which we will discuss later in this report, especially on smaller datasets. To combat this, techniques like dropout are often applied to FC layers during training.

The final Fully-connected layer in a classification network typically has as many neurons as there are classes, with each neuron representing the network's confidence that the input belongs to that particular class. A SoftMax activation function is often applied to this layer to convert the outputs into probabilities that sum to one.

The loss function, is a vital component in CNN and other machine learning models. It measures the difference between the model's predictions and the actual values, serving as a guide for the learning process. The primary goal is to minimize this function, thereby improving the model's accuracy. Loss functions are versatile, applicable to both regression and classification problems in CNNs. Common types include Mean Absolute Error (MAE), Mean Square Error (MSE), and cross-entropy, each with its own specific use cases. By continually working to minimize the chosen loss function during training, the CNN learns to make increasingly accurate predictions or classifications, ultimately enhancing its overall performance [9].

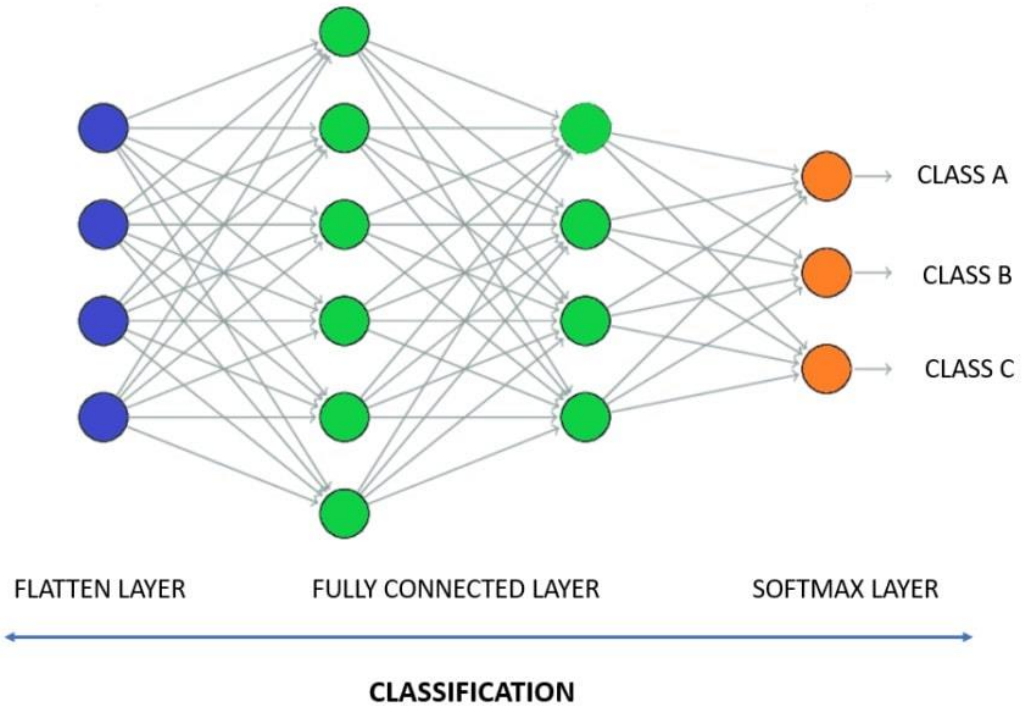


Figure 18: Fully-connected layers [26]

A great visualization of a CNN model can be seen in the following link:

https://adamharley.com/nn_vis/

"with the intent of showing the actual behavior of the network given user-provided input. The user can interact with the network through a drawing pad, and watch the activation patterns of the network respond in real time."

Implementations

The following projects focuses on the implementation of a face recognition system using a dataset of celebrity faces. The dataset comprises a collection of images featuring various well-known personalities, organized into named folders corresponding to each celebrity [31].

The core of our face recognition system is built upon a dataset of celebrity images. This dataset consists of:

1. Structure: Multiple folders, the original dataset includes 31 folders, each corresponding to one of the 31 individuals.
2. Content: Each folder contains numerous images of the corresponding celebrity, captured under various conditions including different angles, lighting, and expressions. The number of images in each folder is not the same, making the dataset unbalanced. Additionally, the images are not limited to face-only shots; they include various parts of the celebrities' appearances.

Figure 19 illustrates the folder structure of the dataset we use for this project and Figure 20 shows the Inside of a folder of the dataset.

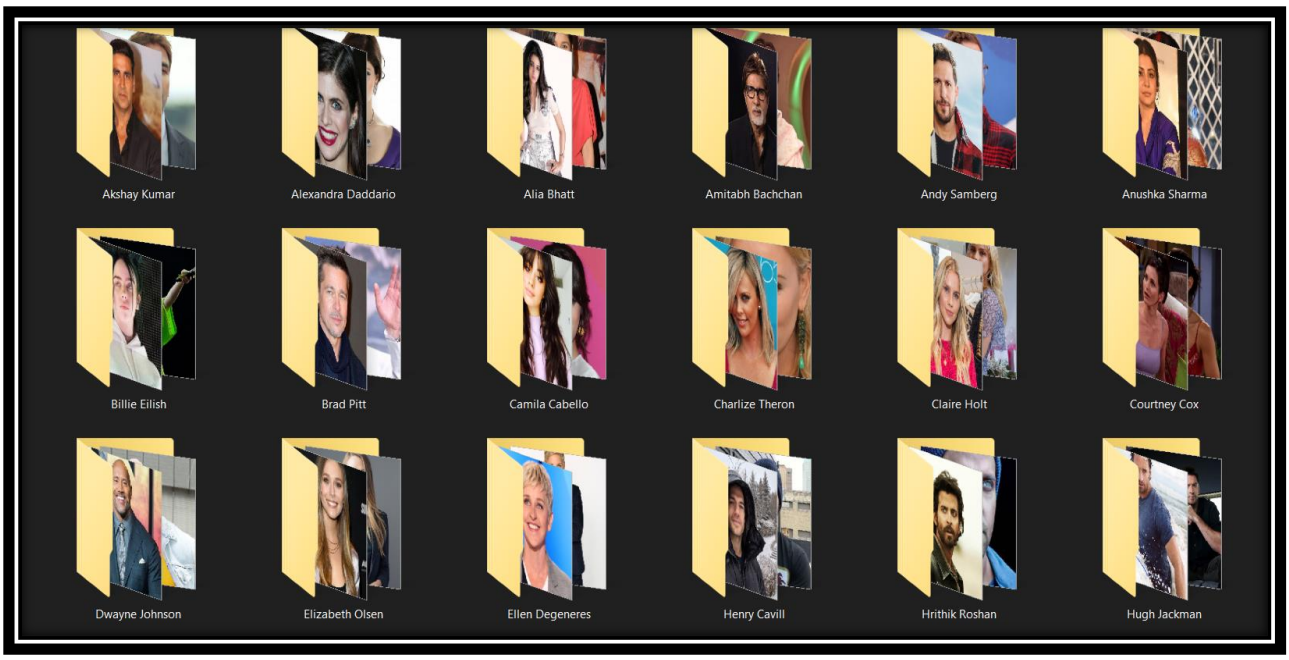


Figure 19: Folder structure of the dataset

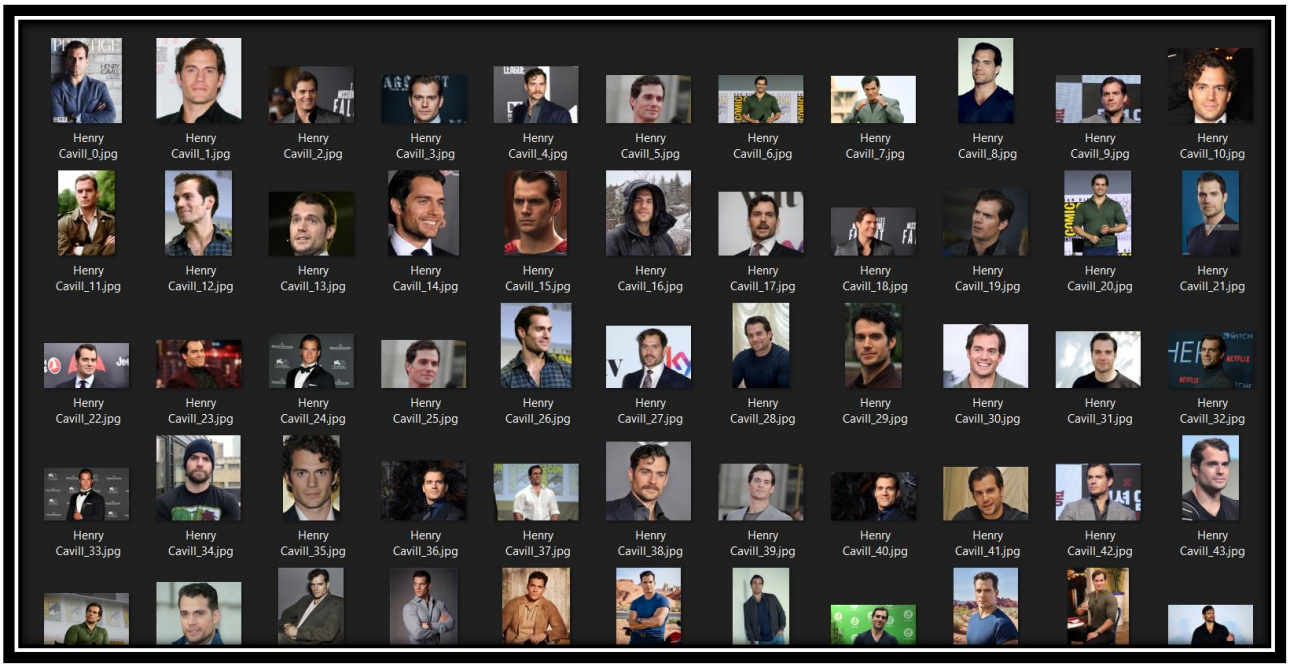


Figure 20: Inside a folder of the dataset, this folder contains images of Henry Cavill

Custom Convolutional Neural Network

In the first implementation project, we use a small Convolutional Neural Network (compared to larger networks like VGG16 and ResNet) and train it on the dataset. We will not be using the entire dataset with all its 31 folders. Instead, we will use only three classes: Henry Cavill with 96 images, Natalie Portman with 95 images, and Robert Downey Jr. with 102 images.

In order to create test data, we remove 9 images from the classes mentioned above and move it to our test directory, we then use the images in this directory to assess the model's performance.

Before We use this dataset, we crop the faces from images to keep the model focused on facial details, this is discussed later in this report.

The network is built using TensorFlow and Keras libraries and includes model creation, training and evaluation.

TensorFlow is an open-source deep learning framework developed by Google. It is designed for high-performance numerical computation and is widely used for developing machine learning applications.

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. It simplifies the process of building deep learning models by providing user-friendly APIs.

```
train_dir = "Cropprd_customCNN/"
test_dir = "Cropped_CustomCNN_Test/"

# Using ImageDataGenerator for validation split
generator = ImageDataGenerator(
    validation_split=0.1          # 10% of the data will be used for validation
)


# Load and split the data into training and validation sets
train_ds = generator.flow_from_directory(
    train_dir,
    target_size=(224, 224),
    batch_size=32,
    subset="training"  # This is for training data
)

val_ds = generator.flow_from_directory(
    train_dir,
    target_size=(224, 224),
    batch_size=32,
    subset="validation"  # This is for validation data
)

# Load the test data from the test directory
test_generator = ImageDataGenerator()
test_ds = test_generator.flow_from_directory(
    test_dir,
    target_size=(224, 224),
    batch_size=32,
    shuffle=False
)
```

Figure 21: Defining training, validation and test data

Figure 21 contains the ImageDataGenerator which is used to generate batches of tensor image. This step helps in enhancing the diversity of the training data without actually increasing the size of the dataset.

The training dataset (train_ds) is used to train the model, while the validation dataset (val_ds) is used to evaluate the model's performance during training. Both datasets are resized to 224x224 pixels, and the batch size is set to 32.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(224, 224, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(96, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(len(classes), activation='softmax'))
```

Figure 22: Adding CNN layers

As depicted in Figure 22, we add the CNN layers to our model.

Convolutional Layers: The model includes multiple convolutional layers with ReLU activation functions. The number

of filters increases progressively to capture more complex features. The number of filters in each convolutional layer is relatively small, ranging from 32 to 96 filters. Larger networks often use hundreds of filters per layer.

Pooling Layers: MaxPooling layers reduce the spatial dimensions of the feature maps, thus reducing the number of parameters and computation in the network.

Batch Normalization: Applied after each pooling layer to normalize the activations and improve the training speed and stability.

Dropout Layer: A dropout layer with a rate of 0.2 is used to prevent overfitting by randomly setting a fraction of input units to 0 during training.

Fully Connected Layers: The network ends with two dense layers, with the final layer using a softmax activation function to output the class probabilities.

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=["accuracy"])
model.summary()

# Train and save the model with validation
history = model.fit(train_ds, epochs=30, batch_size=32, validation_data=val_ds)
model.save('face_recognition_model.keras')
```

Figure 23: Training the model

Figure 23 Shows the training process of our model.

Optimizer: Adam optimizer is chosen for its efficiency and adaptability.

Loss Function: Categorical cross-entropy is suitable for multi-class classification tasks.

Metrics: Accuracy is used as the evaluation metric.

Training: The model is trained for 30 epochs, with the batch size set to 32. The training history is stored for later visualization.

- **Batches**: A batch is a subset of the training data. Instead of feeding the entire dataset into the model at once, which would be computationally expensive and memory-intensive, the data is divided into smaller batches. Each batch is used to update the model's parameters. In this code, the batch size is set to 32, meaning 32 samples are processed before updating the model's weights.

- **Epochs**: An epoch is one complete pass through the entire training dataset. During training, the model goes through multiple epochs, each time learning from the data and updating its parameters. In this code, the model is trained for 30 epochs.

Backpropagation is the process by which the model learns. This process is comprehensively discussed in chapter 3 and 4 of [33]. In general, it involves the following steps:

1. **Forward Pass**: The input data is passed through the network, and the output is computed.
2. **Loss Calculation**: The loss (error) is calculated by comparing the model's output to the true labels using the loss function.
3. **Backward Pass**: Gradients of the loss with respect to each parameter are computed using the chain rule. This is done by backpropagating the error from the output layer to the input layer.
4. **Parameter Update**: The model's parameters (weights and biases) are updated using an optimization algorithm (Adam optimizer in this code), which adjusts the parameters in the direction that reduces the loss.

A written example of Backpropagation(with numbers) can be found in [34].

During each epoch, for each batch of training data, the model performs a forward pass, calculates the loss, computes gradients via backpropagation, and updates the parameters.

Model Saving: The trained model is saved for future use.

```
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_ds)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_accuracy}')

# Predict on the test set
test_ds.reset()
predictions = model.predict(test_ds)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = test_ds.classes
class_labels = list(test_ds.class_indices.keys())


# Compute and print classification report
report = classification_report(true_classes, predicted_classes, target_names=class_labels)
print(report)

# Compute and plot confusion matrix
conf_matrix = confusion_matrix(true_classes, predicted_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

Figure 24: Model Evaluation

Figure 24 displays the process required for evaluating this model.

Evaluation: This step provides a measure of the model's performance on unseen data, which is crucial for assessing its generalization capability.

Classification Report: Provides detailed metrics such as precision, recall, and F1-score for each class.

Confusion Matrix: Visualizes the performance of the classification model, indicating how often predictions are correct and where errors occur.

To keep the model focused on the facial structures, instead of the background objects, we use a version of our dataset which has cropped faces.

A great tool for detecting faces in images is MTCNN [32].

The MTCNN (Multi-Task Cascaded Convolutional Networks) algorithm is a deep learning-based face detection and alignment method that uses a cascading series of convolutional neural networks (CNNs) to detect and localize faces in digital images or videos.

Figure 25 depicts an implementation of a program used for detecting and cropping faces from an image. It receives as input an image, and it places the cropped images of the faces contained in the input image, to the given directory.

Figure 26 is a folder of our dataset which contains only images from cropped faces.

```python
def detect_and_crop_faces(image_path, output_dir):
    # Load the image
    image = cv2.imread(image_path)
    if image is None:
        print(f"Could not read image {image_path}")
        return

    # Convert the image to RGB (OpenCV loads images in BGR format)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Initialize MTCNN detector
    detector = MTCNN()

    # Detect faces in the image
    results = detector.detect_faces(image_rgb)

    # Ensure the output directory exists
    os.makedirs(output_dir, exist_ok=True)

    # Plot the image with detected faces
    plt.imshow(image_rgb)
    ax = plt.gca()

    # Loop through all detected faces
    for i, result in enumerate(results):
        # Get the bounding box coordinates
        x, y, width, height = result['box']
        x, y = abs(x), abs(y)

        # Draw rectangle around the face
        rect = plt.Rectangle((x, y), width, height, edgecolor='red', facecolor='none')
        ax.add_patch(rect)

        # Crop the face from the image
        cropped_face = image_rgb[y:y+height, x:x+width]

        # Convert the cropped face to an Image object
        face_image = Image.fromarray(cropped_face)

        # Save the cropped face image to the output directory
        face_image_path = os.path.join(output_dir, f"face_{i+1}.jpg")
        face_image.save(face_image_path)

        print(f"Saved cropped face {i+1} to {face_image_path}")

    # Display the plot
    plt.axis('off')
    plt.show()

# Example usage
image_path = 'duo.jfif'
output_dir = 'E:\Dars_14022\Multimedia\Project\cropped_faces'
detect_and_crop_faces(image_path, output_dir)
```

Figure 25: Program for cropping faces in an image

Figure 26: Inside of Henry Cavill's folder, now containing cropped faces

Results

```
Test Loss: 0.6549947261810303
Test Accuracy: 0.8148148059844971
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 276ms/step
                    precision    recall  f1-score   support

       Henry Cavill       0.78      0.78      0.78         9
    Natalie Portman       1.00      0.67      0.80         9
Robert Daweny Junior      0.75      1.00      0.86         9

           accuracy                           0.81        27
          macro avg       0.84      0.81      0.81        27
       weighted avg       0.84      0.81      0.81        27
```

Figure 27: Evaluation results for custom CNN

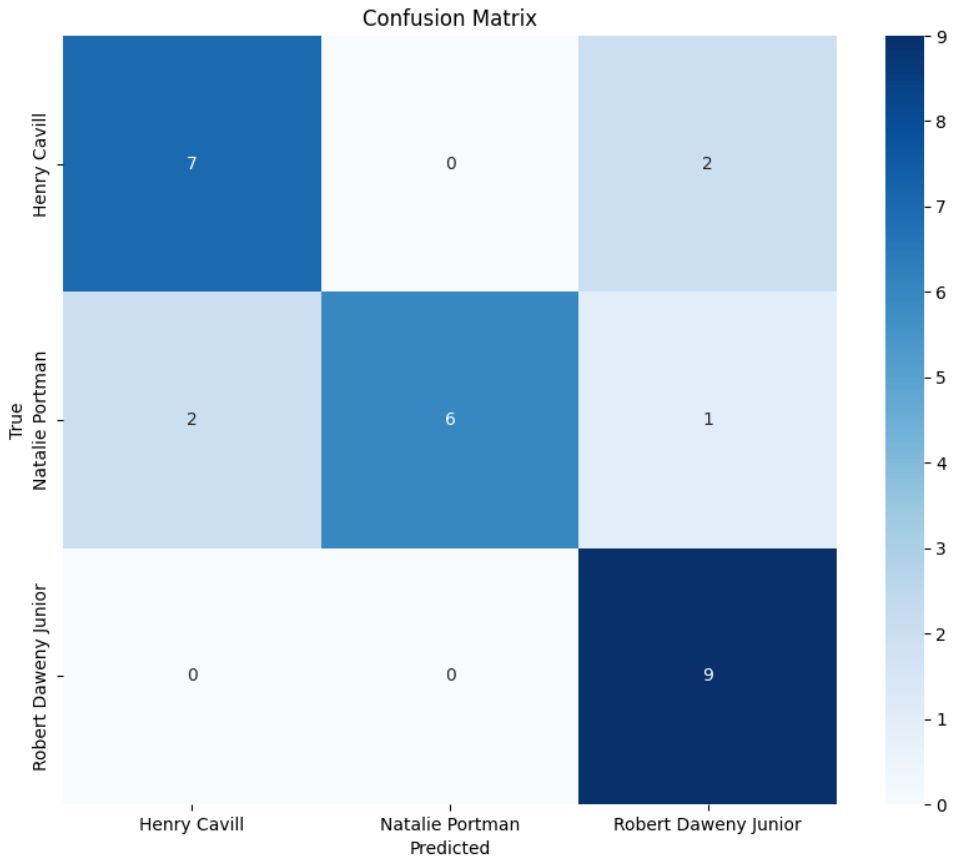


Figure 28: Confusion Matrix for custom CNN

In the discussion of model evaluation metrics, the following definitions are provided [30]:

Accuracy: This metric evaluates the overall correctness of a model's predictions across the entire dataset. It is computed by dividing the sum of true positives (TP) and true negatives (TN) by the total number of samples. Accuracy provides a general overview of the model's performance.

Precision: This measure focuses on the accuracy of positive predictions. It is calculated by dividing the number of true positives (TP) by the total number of positive predictions (TP + FP). Precision is particularly useful when the cost of false positives is high.

Recall: Also known as sensitivity or true positive rate, recall assesses the model's ability to identify all positive instances. It is determined by dividing the number of true positives (TP) by the total number of actual positive instances (TP + FN). Recall is crucial when the cost of false negatives is high.

F1 Score: This metric provides a single score that balances precision and recall. It is calculated as the harmonic mean of precision and recall, giving equal weight to both. The F1 score is particularly valuable when you need to find an optimal balance between precision and recall, as it is sensitive to imbalances between these two metrics.

As it can be observed from Figure 27 and Figure 28 The model shows good overall performance with an accuracy of around 81%.

Henry Cavill:

Precision: 0.78 - Out of all predictions made for Henry Cavill, 78% were correct.

Recall: 0.78 - Out of all actual instances of Henry Cavill, 78% were correctly identified by the model.

F1-Score: 0.78 - A balance between precision and recall.

Natalie Portman:

Precision: 1.00 - All predictions made for Natalie Portman were correct.

Recall: 0.67 - Out of all actual instances of Natalie Portman, 67% were correctly identified.

F1-Score: 0.80 - Indicates a slight imbalance between precision and recall, with precision being perfect.

Robert Downey Junior:

Precision: 0.75 - Out of all predictions made for Robert Downey Junior, 75% were correct.

Recall: 1.00 - All actual instances of Robert Downey Junior were correctly identified.

F1-Score: 0.86 - Shows a good balance, with perfect recall.

Small Dataset

This custom CNN may not perform well with the small dataset that we used due to several reasons:

- **Overfitting:** With a limited amount of data, the model is likely to overfit, capturing noise rather than learning general patterns.
- **Insufficient Diversity:** A small dataset may not represent the variability of real-world data, leading to poor generalization.
- **Data Augmentation Limits:** While data augmentation helps, it cannot fully replace the need for a large, diverse dataset.

Another reason for that may cause poor performance is that we are implementing a custom CNN, which has a few layers of convolution and a low level of complexity, for complex tasks like distinguishing between different individuals, the model might need more layers and neurons to capture the subtle differences.

A small network has fewer parameters (weights) to learn from the data. Faces in particular have complex features that require deeper networks to differentiate effectively. A small CNN might miss these subtle differences.

Therefore, in order to have a better performance in our face recognition task, we need to use large pre-trained models that has

been trained on very large datasets of images and has more complex layers than this custom CNN.

VGG16

Pre-trained CNN models have become essential tools in computer vision tasks, offering a range of architectures with different strengths and trade-offs. The VGG16 and VGG19 models are known for their simple architecture with consecutive convolutional layers. They offer good feature extraction capabilities but come with a large number of parameters, resulting in slower inference times. Despite this, they still perform well on image classification tasks.

VGG16 is a convolutional neural network trained on a subset of the ImageNet dataset, a collection of over 14 million images belonging to 1,000 categories, a small example of which is shown in Figure 29. It is characterized by its depth, consisting of 16 layers, including 13 convolutional layers and 3 fully connected layers. VGG-16 is known for its simplicity and effectiveness, as well as its ability to achieve strong performance on various computer vision tasks, including image classification and object recognition [11].

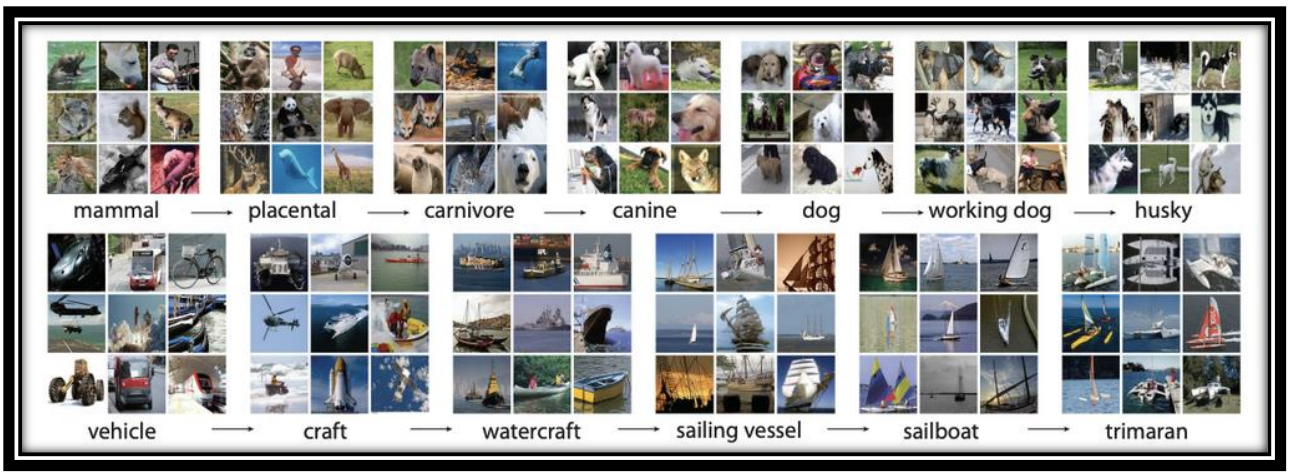


Figure 29: Example ImageNet images [27]

The model's architecture features a stack of convolutional layers followed by max-pooling layers, with progressively increasing depth. This design enables the model to learn intricate hierarchical representations of visual features, leading to robust and accurate predictions. Despite its simplicity compared to more recent architectures, VGG-16 remains a popular choice for many deep learning applications due to its versatility and excellent performance.

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual competition in computer vision where teams tackle tasks including object localization and image classification. VGG16, proposed by Karen Simonyan and Andrew Zisserman in 2014, achieved top ranks in both tasks, detecting objects from 200 classes and classifying images into 1000 categories [10].

Figure 30 shows the general structure of VGG-16.

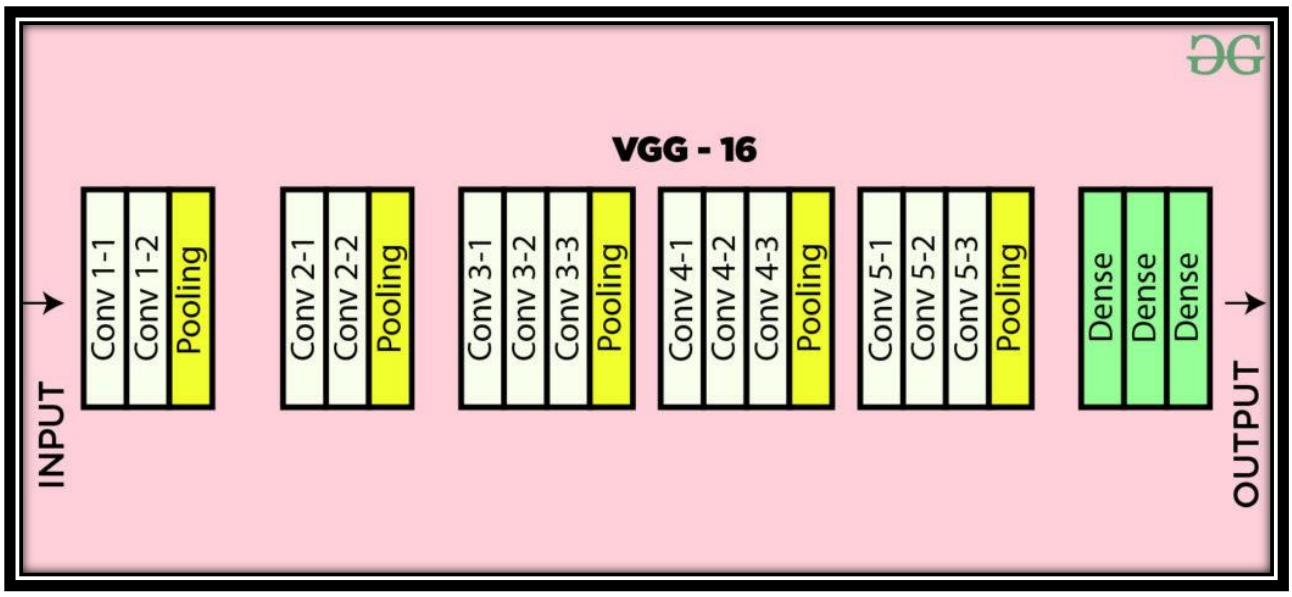


Figure 30: VGG-16 architecture [10]

Limitations of VGG 16 [11]:

- It is very slow to train.
- The size of VGG-16 trained imageNet weights is *528* MB. So, it takes a lot of disk space and bandwidth which makes it inefficient.

Now, we will train the VGG16 model on our own dataset to see how it performs. We use the same classes that we used earlier in the custom implementation.

```python
# Using ImageDataGenerator for data augmentation
generator = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,             # Shear transformation
    zoom_range=0.2,               # Random zoom
    rotation_range=20,            # Random rotation in the range [-20, 20] degrees
    width_shift_range=0.1,        # Random horizontal shift
    height_shift_range=0.1,       # Random vertical shift
    horizontal_flip=True,        # Random horizontal flip
    vertical_flip=True,          # Random vertical flip
    brightness_range=[0.8, 1.2],  # Random brightness adjustment
    validation_split=0.1          # 10% of the data will be used for validation
)
```

Figure 31: Data Augmentation

As shown in Figure 31, the ImageDataGenerator class from Keras is used to augment the training images. Various augmentation techniques such as rescaling, shear transformation, zoom, rotation, width and height shifts, horizontal and vertical flips, and brightness adjustments are applied to make the model more robust to variations in the input data. Additionally, 10% of the data is reserved for validation.

Augmentation effectively expands the size of our training dataset without actually collecting new data. This is especially beneficial when working with limited datasets

By introducing variations of the original images, augmentation helps the model learn to recognize objects or features under different conditions. This improves the model's ability to generalize to new, unseen data. For example, if the model only

sees perfectly centered images during training, it might struggle with slightly off-center images in real-world applications.

Figure 32 shows some output images from the augmentation process.

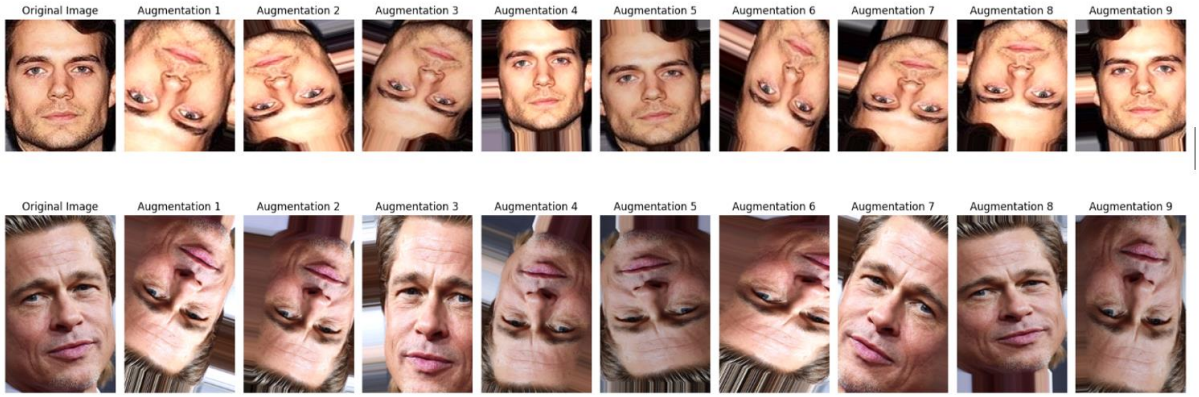


Figure 32: Examples of data augmentation

```
# Load and split the data into training and validation sets
train_ds = generator.flow_from_directory(
    train_dir,
    target_size=(224, 224),
    batch_size=32,
    subset="training"  # This is for training data
)

val_ds = generator.flow_from_directory(
    train_dir,
    target_size=(224, 224),
    batch_size=32,
    subset="validation"  # This is for validation data
)

# Load the test data from the test directory
test_generator = ImageDataGenerator(rescale=1./255)
test_ds = test_generator.flow_from_directory(
    test_dir,
    target_size=(224, 224),
    batch_size=32,
    shuffle=False
)
```

Figure 33: Defining training, validation and test data

Figure 33 shows that the data is loaded from the specified directory and split into training and validation sets using the flow_from_directory method. The images are resized to 224x224 pixels, and a batch size of 32 is used.

```
# Get the list of classes
classes = list(train_ds.class_indices.keys())
print(classes)

# Load VGG16 model with pre-trained ImageNet weights
vgg = VGG16(input_shape=(224, 224, 3), weights='imagenet', include_top=False)

# Freeze the layers
for layer in vgg.layers:
    layer.trainable = False

# Add custom layers
x = Flatten()(vgg.output)
x = Dense(4096, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(1023, activation='relu')(x)
x = Dropout(0.25)(x)
x = Dense(4, activation='softmax')(x)
model = Model(inputs = vgg.input, outputs = x)

# Create the model
model = Model(inputs=vgg.input, outputs=x)

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Figure 34: Loading VGG16 and creating our model

As it can be seen in Figure 34, the VGG16 model pre-trained on ImageNet is loaded without the top classification layer. This base model is used to leverage the rich feature representations learned from the vast ImageNet dataset.

All the layers of the VGG16 model are frozen to retain the pre-trained weights. This prevents the weights from being updated during training, preserving the learned features.

Custom layers are added on top of the frozen VGG16 base. This includes a Flatten layer to convert the 2D feature maps to 1D, followed by Dense layers with ReLU activation and Dropout for regularization. The final Dense layer uses a softmax activation function for classification.

The model is compiled with the Adam optimizer and categorical cross-entropy loss function. Accuracy is used as the performance metric.

```
# Train the model
history = model.fit(train_ds, epochs=30, validation_data=val_ds)
```

Figure 35: Training the model

Figure 35 shows that the model is trained for 30 epochs using the training and validation datasets. The history of training and validation accuracy and loss is stored for later visualization.

```python
# Evaluate the model on the training set
train_loss, train_accuracy = model.evaluate(train_ds)
print(f"Training Accuracy: {train_accuracy*100: .2f}")

# Evaluate the model on the validation set
val_loss, val_accuracy = model.evaluate(val_ds)
print(f'Validation Loss: {val_loss}')
print(f'Validation Accuracy: {val_accuracy}')


# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_ds)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_accuracy}')


# Predict on the test set
test_ds.reset()
predictions = model.predict(test_ds)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = test_ds.classes
class_labels = list(test_ds.class_indices.keys())

# Compute and print classification report
report = classification_report(true_classes, predicted_classes, target_names=class_labels)
print(report)

# Compute and plot confusion matrix
conf_matrix = confusion_matrix(true_classes, predicted_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

Figure 36: Model Evaluation

It can be seen in Figure 36 that the model is evaluated on the training, validation and test sets to obtain accuracy and loss metrics. These metrics provide insight into how well the model has learned the training data and how well it generalizes to the validation data.

The model's predictions on the test set are used to generate a classification report and confusion matrix, which provide detailed insights into the model's performance for each class.

Results

```
Training Accuracy:  96.27
Training Accuracy:  96.27
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - accuracy: 0.9600 - loss: 0.1062
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - accuracy: 0.9600 - loss: 0.1062
Validation Loss: 0.1062135249376297
Validation Accuracy: 0.9599999785423279
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - accuracy: 0.9259 - loss: 0.2474
Test Loss: 0.24737977981567383
Test Accuracy: 0.9259259104728699
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
                    precision    recall  f1-score   support

       Henry Cavill       0.89      0.89      0.89         9
    Natalie Portman       1.00      1.00      1.00         9
Robert Daweny Junior      0.89      0.89      0.89         9

           accuracy                           0.93        27
          macro avg       0.93      0.93      0.93        27
       weighted avg       0.93      0.93      0.93        27
```

Figure 37: Evaluation Results for VGG16 model

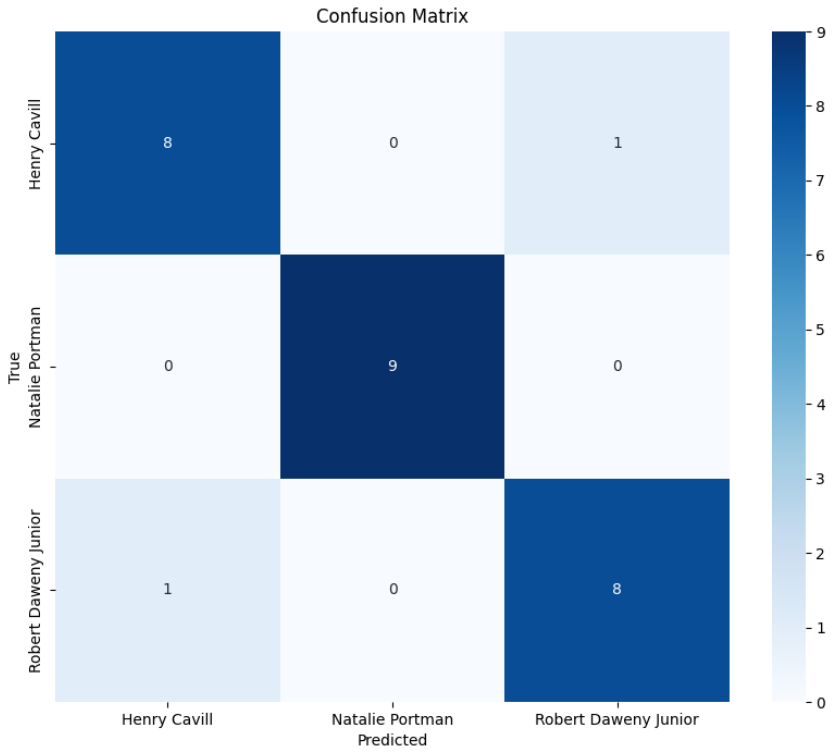


Figure 38: Confusion Matrix for VGG16 model

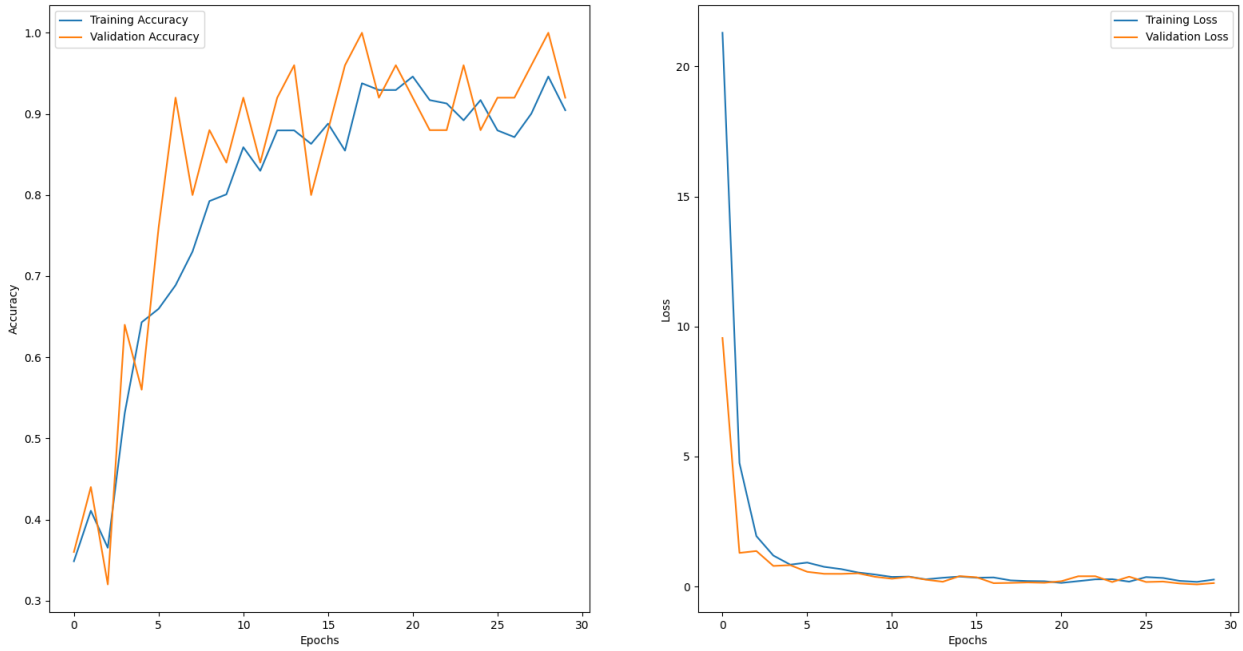


Figure 39: VGG16 performance in the progression of epochs

Figure 37, Figure 38 and Figure 39, illustrate the results of the VGG16-based model that show a test accuracy of approximately 92.59%, which indicates that the model generalizes well to unseen data, maintaining high accuracy.

High Precision and Recall: Both precision and recall are high across all classes, indicating that the model is correctly identifying instances of each class and not missing many instances.

High F1-Score: The high F1-scores across all classes indicate a good balance between precision and recall.

Consistent Performance: The model performs consistently well across the training, validation, and test datasets.

An issue that we have faced is that despite the good performance the model has on input images that belong to the classes it has been trained on, it exhibits high confidence in its predictions when presented with irrelevant images that are not included in any of the classes the model is trained on, an example of which can be seen in Figure 40. This behavior indicates that the model might be overfitting to the training data. As discussed earlier, overfitting occurs when a model learns the training data too well, including its noise and outliers, rather than learning the underlying patterns. In essence, the model starts to "memorize" the specific examples in the training set rather than learning general patterns that can be applied to new, unseen data.

Why it happens:

1. Model Complexity: If a model is too complex (has too many parameters like VGG16) relative to the amount of training data, it can start to fit the noise in the data rather than the underlying pattern.
2. Limited Data: When there's not enough diverse training data, the model might learn specific features of the training examples that don't generalize well to new data.
3. Training Too Long: If a model is trained for too many epochs, it can start to learn the peculiarities of the training data, including noise and outliers.
4. Class Imbalance: If some classes are underrepresented or if there is significant overlap in the features of different classes, the model might struggle to differentiate between them. It may thus be more confident in assigning an input to a familiar class, even if it's not accurate.

Due to the very high confidence our model has, we are unable to set a confidence threshold for the model's predictions.

Adding "Unknown" class to our dataset

One solution to reduce the confidence on our model when facing irrelevant images, is to add an "Unknown" class to our training dataset. This class is populated with a large number of diverse

images that don't belong to any of our main classes of interest. When the model encounters an image that doesn't strongly match any of the main classes, it should classify it as "unknown". This is not a very efficient solution to this problem and it has many implementation flaws and limitations.

Limitations and Challenges:

1. Class Imbalance: The "unknown" class often ends up being much larger than other classes, which can lead to bias in the model.
2. Coherent learning: The "unknown" class is inherently very diverse, which can make it difficult for the model to learn a coherent representation.
3. Reduced Specificity: The model might become too eager to classify borderline cases as "unknown," reducing its ability to correctly identify instances of the main classes.
4. Data Collection: Gathering a truly representative set of "unknown" images that cover all possible irrelevant inputs is challenging and potentially endless.
5. Computational Cost: Adding a large number of additional images increases training time and computational requirements.

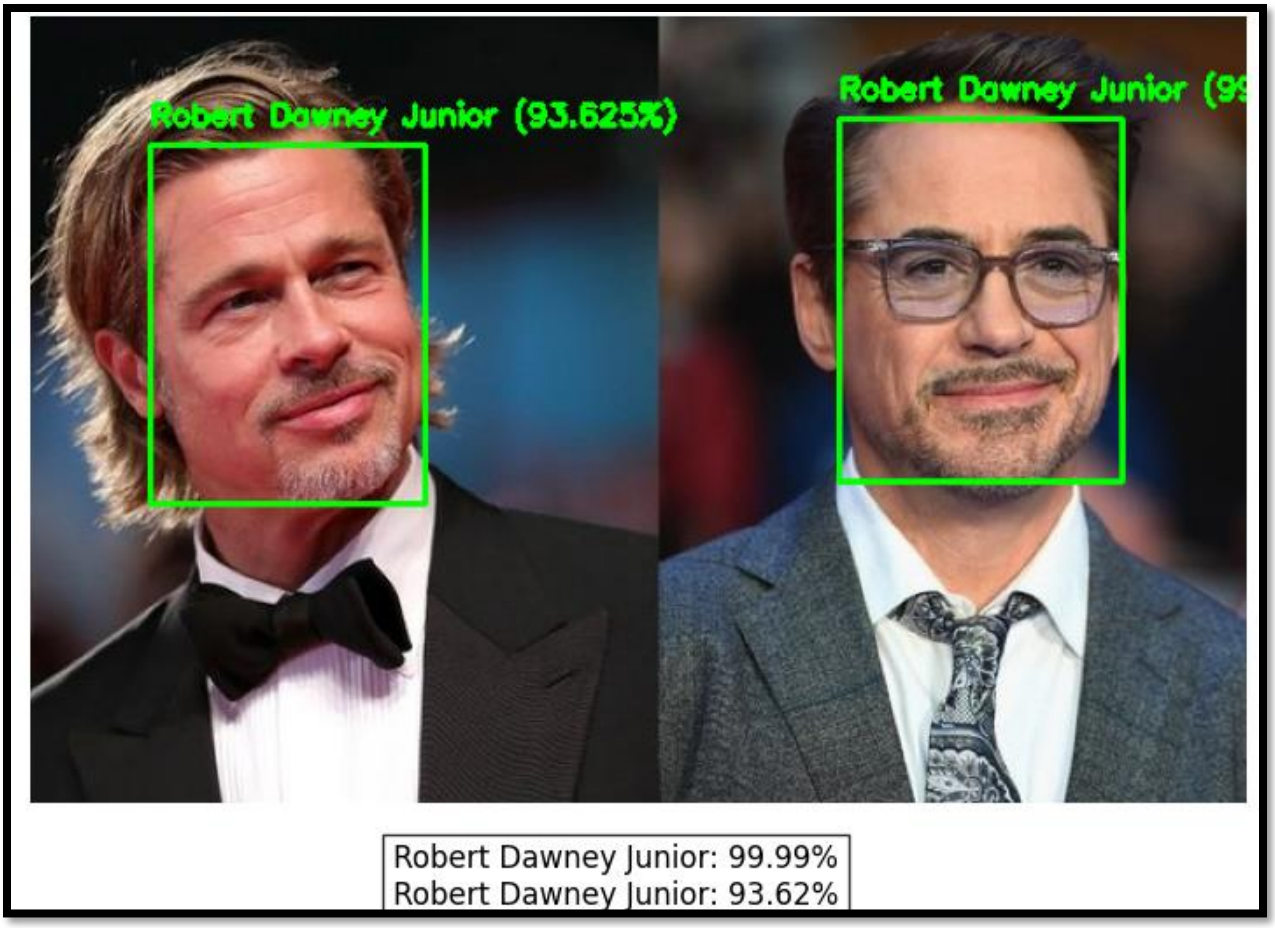


Figure 40: Model predicting Robert Dawney Junior, while the face belongs to Brad Pitt

While there are many methods that can be used to detect unknown images and to correctly label them [35], in the next step we focus on finding a model that is specefically designed for face recognition.

Given the specific requirements of face recognition and the constraints of the small dataset, it would be advisable to explore models specifically designed for face recognition that are more relevant to face recognition tasks, and are generally more efficient and can perform well with smaller datasets.

FaceNet and Support vector machine [13]

FaceNet is a facial recognition system developed by Florian Schroff, Dmitry Kalenichenko and James Philbina, a group of researchers affiliated to Google. The system was first presented in the IEEE Conference on Computer Vision and Pattern Recognition held in 2015. The system uses a deep convolutional neural network to learn a mapping (also called an embedding) from a set of face images to the 128-dimensional Euclidean space and the similarity between two face images is assessed based on the square of the Euclidean distance between the corresponding normalized vectors in the 128-dimensional Euclidean space. The system used the triplet loss function as the cost function and introduced a new online triplet mining method. The system achieved an accuracy of 99.63% which is the highest score on Labeled Faces in the Wild dataset in the unrestricted with labeled outside data protocol.

"*Our method uses a deep convolutional network trained to directly optimize the embedding itself, rather than an in termediate bottleneck layer as in previous deep learning approaches. To train, we use triplets of roughly aligned matching / non-matching face patches generated using a novel online triplet mining method. The benefit of our approach is much greater representational efficiency: we*

achieve state-of-the-art face recognition performance using only 128-bytes per face." [13]

Figure 41 shows the Overall structure of the FaceNet face recognition system.

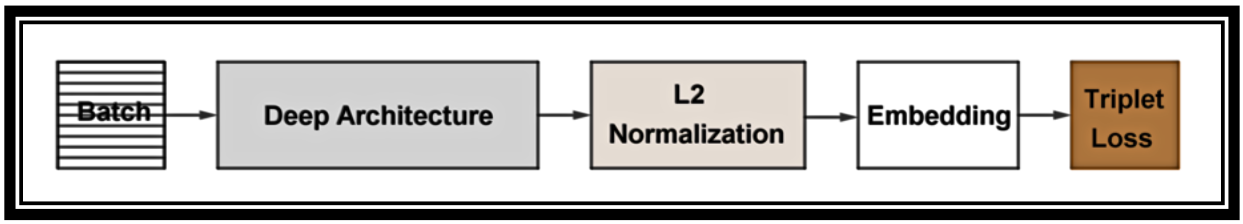


Figure 41: Overall structure of the FaceNet face recognition system [28]

For training, the researchers used as input batches of about 1800 images in which for each identity there were about 40 similar images and several randomly sampled images relating to other identities. These batches were fed to a deep convolutional neural network.

The CNN layers used in FaceNet is given in Figure 43.

As FaceNet has achieved state-of-the-art results in the many benchmark face recognition dataset such as Labeled Faces in the

Wild (LFW) and Youtube Face Database and is specifically tailored for face recognition and classification task, using a combination of a pre-trained FaceNet model to extract face embeddings and a Support Vector Machine (SVM) classifier to recognize individuals, We are going to train this model on our own datasets to see how it compares to the previous implementations. Figure 42 display the variability that FaceNet model can handle.

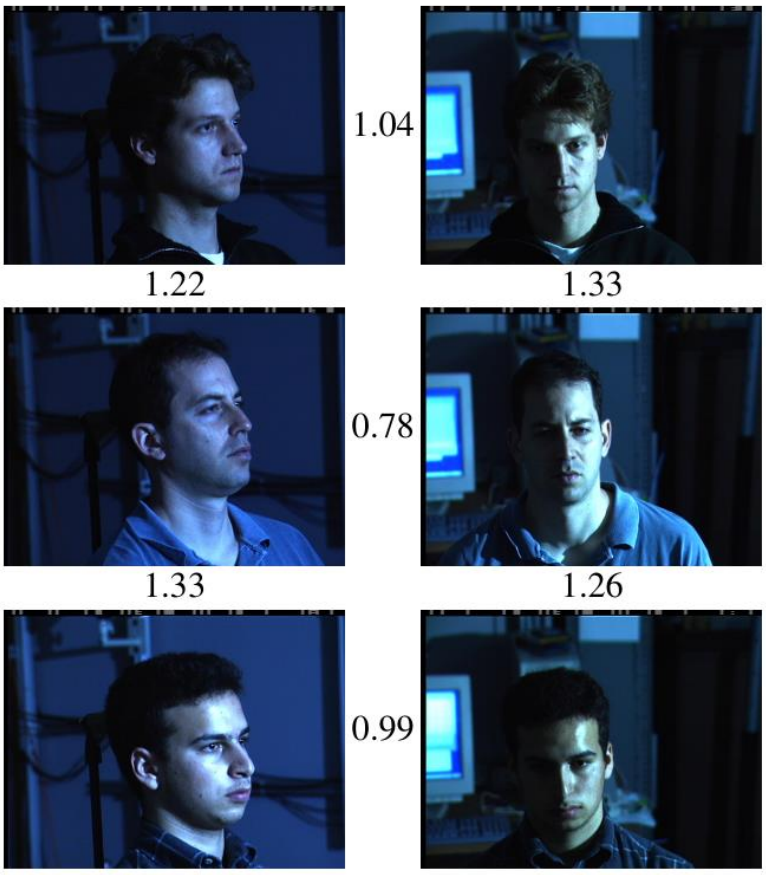


Figure 42: Illumination and Pose invariance. Pose and illumination have been a long-standing problem in face recognition. This figure shows the output distances of FaceNet between pairs of faces of the same and a different person in different pose and illumination combinations. A distance of 0.0 means the faces are identical, 4.0 corresponds to the opposite spectrum, two different identities. You can see that a threshold of 1.1 would classify every pair correctly. [13]

Structure of the CNN used in the model NN1 in the FaceNet face recognition system

Layer	Size-in (rows × cols × #filters)	Size-out (rows × cols × #filters)	Kernel (rows × cols, stride)	Parameters	FLOPS
conv1	220×220×3	110×110×64	7×7×3, 2	9K	115M
pool1	110×110×64	55×55×64	3×3×64, 2	0	—
rnorm1	55×55×64	55×55×64		0	
conv2a	55×55×64	55×55×64	1×1×64, 1	4K	13M
conv2	55×55×64	55×55×192	3×3×64, 1	111K	335M
rnorm2	55×55×192	55×55×192		0	
pool2	55×55×192	28×28×192	3×3×192, 2	0	
conv3a	28×28×192	28×28×192	1×1×192, 1	37K	29M
conv3	28×28×192	28×28×384	3×3×192, 1	664K	521M
pool3	28×28×384	14×14×384	3×3×384, 2	0	
conv4a	14×14×384	14×14×384	1×1×384, 1	148K	29M
conv4	14×14×384	14×14×256	3×3×384, 1	885K	173M
conv5a	14×14×256	14×14×256	1×1×256, 1	66K	13M
conv5	14×14×256	14×14×256	3×3×256, 1	590K	116M
conv6a	14×14×256	14×14×256	1×1×256, 1	66K	13M
conv6	14×14×256	14×14×256	3×3×256, 1	590K	116M
pool4	14×14×256	3×3×256, 2	7×7×256	0	
concat	7×7×256	7×7×256		0	
fc1	7×7×256	1×32×128	maxout p=2	103M	103M
fc2	1×32×128	1×32×128	maxout p=2	34M	34M
fc7128	1×32×128	1×1×128		524K	0.5M
L2	1×1×128	1×1×128		0	
Total				**140M**	**1.6B**

Figure 43: CNN layers used in FaceNet [28]

For this implementation, we use 13 classes of our original dataset to train our model, the details of those are as follows:

Alexandra Daddario, 83 images

Andy Samberg, 83 images

Billie Eilish, 88 images

Brad Pitt, 108 images

Charlize Theron, 71 images

Elizabeth Olsen, 64 images

Henry Cavill, 96 images

Hugh Jackman, 101 images

Margot Robbie, 65 images

Natalie Portman, 95 images

Robert Daweny Junior, 102 images

Tom Cruise, 53 images

Zac Efron, 81 images

Multiple images(ranging from 5 to 12) belonging to each of these classes, are used as test dataset.

```python
import numpy as np
import os
from keras_facenet import FaceNet
from sklearn.preprocessing import LabelEncoder, Normalizer
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from random import choice
from matplotlib import pyplot as plt
import joblib

# Function to load faces and their labels
def load_dataset(dir):
    X, y = [], []
    for subdir in os.listdir(dir):
        subdir_path = os.path.join(dir, subdir)
        for filename in os.listdir(subdir_path):
            img_path = os.path.join(subdir_path, filename)
            X.append(img_path)
            y.append(subdir)
    return np.array(X), np.array(y)

# Paths to dataset directories
train_dir = 'E:/Dars_14022/Multimedia/Project/Final Project/TrainData_Faces_Cropped'
test_dir = 'E:/Dars_14022/Multimedia/Project/Final Project/TrainData_Faces_Cropped_Test'

# Load the dataset
trainX_paths, trainy = load_dataset(train_dir)
testX_paths, testy = load_dataset(test_dir)

print(f"Found {len(trainX_paths)} training images and {len(testX_paths)} testing images.")
```

Figure 44: Loading train and test images with their labels

As shown in Figure 44, the load_dataset function is designed to load images from a given directory and their corresponding labels. It recursively goes through subdirectories, each representing a different person, and collects image paths and labels.

train_dir and test_dir: Paths to the training and testing datasets, respectively.

trainX_paths and trainy, testX_paths and testy: Arrays holding the paths to the images and their corresponding labels for both training and testing datasets.

```python
# Initialize FaceNet model
embedder = FaceNet()

# Function to extract embeddings from images
def get_embeddings(paths, labels):
    embeddings = []
    valid_labels = []
    for i, (path, label) in enumerate(zip(paths, labels)):
        print(f"Processing image {i+1}/{len(paths)}")
        image = plt.imread(path)  # Load image using matplotlib
        detections = embedder.extract(image, threshold=0.95)
        if detections:
            embedding = detections[0]['embedding']
            embeddings.append(embedding)
            valid_labels.append(label)
        else:
            print(f"No face detected in image: {path}")
    return np.array(embeddings), np.array(valid_labels)
```

Figure 45: Extracting embeddings form images

The FaceNet model is initialized to generate embeddings for the face images, the process of which is shown in Figure 45. Embeddings are vector representations of faces that capture the distinctive features necessary for recognition.

embedder: Initialized FaceNet model.

get_embeddings: Function to process images and extract embeddings. It filters out images where no face is detected, ensuring only valid embeddings are used.

- The function takes two parameters: paths (list of image file paths) and labels (corresponding labels).
- It initializes empty lists for embeddings and valid_labels.
- It iterates through each image path and label:
    - Loads the image using plt.imread().
    - Uses the FaceNet embedder.extract() method to detect faces and extract embeddings.
    - If a face is detected (with a confidence threshold of 0.95):
        - It appends the embedding and label to their respective lists.
    - If no face is detected, it prints a message.
- Finally, it returns numpy arrays of embeddings and valid labels.

```
# Extract embeddings for train and test sets
emdTrainX, trainy_valid = get_embeddings(trainX_paths, trainy)
emdTestX, testy_valid = get_embeddings(testX_paths, testy)

print(f"Extracted embeddings: Train set: {emdTrainX.shape}, Test set: {emdTestX.shape}")

# Save embeddings
np.savez_compressed('celebrity-faces-embeddings.npz', emdTrainX, trainy_valid, emdTestX, testy_valid)

# Train SVM classifier
in_encoder = Normalizer()
emdTrainX_norm = in_encoder.transform(emdTrainX)
emdTestX_norm = in_encoder.transform(emdTestX)
```

Figure 46: Training the SVM classifier

Figure 46 shows that the embeddings for both training and testing datasets are extracted and normalized. Normalization ensures that the embeddings have a standard scale, improving the performance of the SVM classifier.

emdTrainX, emdTestX: Extracted embeddings.

in_encoder: Normalizer instance to standardize embeddings.

emdTrainX_norm, emdTestX_norm: Normalized embeddings.

When we later use the model to classify new images, we need to apply the same normalization to the embeddings of these new images to ensure consistency. If we don't normalize new embeddings in the same way, the SVM might not perform well because the input data distribution would be different from what it was trained on.

```
out_encoder = LabelEncoder()
out_encoder.fit(trainy_valid)
trainy_enc = out_encoder.transform(trainy_valid)
testy_enc = out_encoder.transform(testy_valid)

model = SVC(kernel='linear', probability=True)
model.fit(emdTrainX_norm, trainy_enc)
```

Figure 47: Training the SVM classifier

The labels are encoded to numerical values suitable for the SVM classifier, as shown in Figure 47. (e.g., "Brad_Pitt") to numerical values (e.g., 0, 1). The SVM classifier is then trained using the normalized embeddings.

When we make predictions with the trained model, the output will be numerical labels. We need the out_encoder to convert these numerical labels back to the original class labels (names). Without saving the out_encoder, we would not be able to interpret the numerical predictions made by the SVM.

out_encoder: Label encoder to transform categorical labels to numerical values.

trainy_enc, testy_enc: Encoded training and testing labels.

model: SVM classifier with a linear kernel, configured to output probability estimates.

Summary

- **Extract Embedding**: For each detected face, we extract the embedding using FaceNet.
- **Normalize Embedding**: Normalize the embedding using in_encoder.
- **Classify with SVM**: Use the trained SVM model to classify the normalized embedding (model.predict).
- **Predict Name and Probability**: Convert the numerical class index to the corresponding name and determine the prediction probability.

Using a separate program, the model's performance is evaluated on the testing datasets, measuring accuracy to understand how well the model generalizes. The embeddings get extracted from the test dataset images and the same Normalization and encoding is applied to the test dataset. Using the saved model, we can predict the class of each image.

Results:

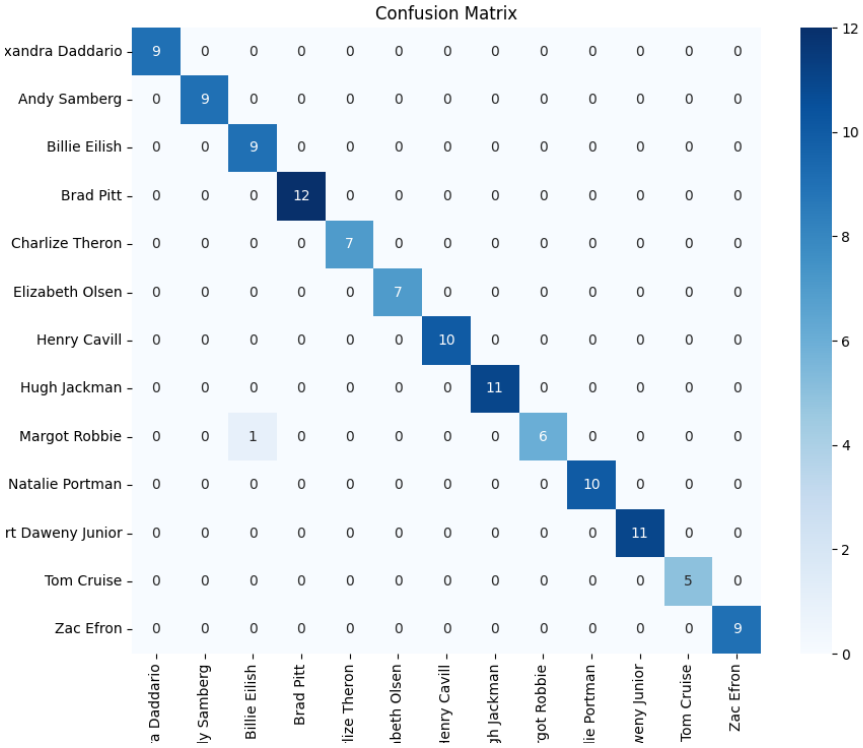


Figure 48: Confusion Matrix for FaceNet model, True classes(vertical) and Predicted classes(horizontal)

	precision	recall	f1-score	support
Alexandra Daddario	1.00	1.00	1.00	9
Andy Samberg	1.00	1.00	1.00	9
Billie Eilish	0.90	1.00	0.95	9
Brad Pitt	1.00	1.00	1.00	12
Charlize Theron	1.00	1.00	1.00	7
Elizabeth Olsen	1.00	1.00	1.00	7
Henry Cavill	1.00	1.00	1.00	10
Hugh Jackman	1.00	1.00	1.00	11
Margot Robbie	1.00	0.86	0.92	7
Natalie Portman	1.00	1.00	1.00	10
Robert Daweny Junior	1.00	1.00	1.00	11
Tom Cruise	1.00	1.00	1.00	5
Zac Efron	1.00	1.00	1.00	9
accuracy			0.99	116
macro avg	0.99	0.99	0.99	116
weighted avg	0.99	0.99	0.99	116

Figure 49: Evaluation of the FaceNet model

By observing the results shown in Figure 48 and Figure 49, we can see that our FaceNet model has performed exceptionally well on the dataset, with nearly perfect precision, recall, and F1-scores for almost all classes.

This performance suggests that the model is very effective at distinguishing between the different individuals in the dataset.

Now, let’s create a **face recognition application** to see how this model performs.

In general, the application’s functionality is as follows:

1. It loads the pre-trained face classification model and necessary encoders that we have saved.
2. The GUI allows users to upload an image.
3. When an image is uploaded, it's displayed in the GUI.
4. The program then uses FaceNet to detect faces in the uploaded image.
5. For each detected face:
    - It extracts facial embeddings.
    - Uses the pre-trained model to predict the identity of the face.

- Calculates the probability of the prediction.

6. The program draws rectangles around detected faces in the image:
   - Green rectangles for known faces (probability > 85%)
   - Red rectangles for unknown faces (probability <= 85%)
7. It displays the predicted name and probability for each face on the image.
8. The processed image with face detections and predictions is shown in the GUI.
9. A list of all predictions (names and probabilities) is displayed below the image.
10. The GUI is updated with each new image upload, replacing the previous results.

```
# Load the trained model and encoders
model = joblib.load('face_classifier.pkl')
in_encoder = joblib.load('in_encoder.pkl')
out_encoder = joblib.load('out_encoder.pkl')

# Initialize FaceNet for embedding extraction
embedder = FaceNet()

# Keep a reference to the current canvas
current_canvas = None
```

Figure 50: Loading the saved model and encoders

The pre-trained SVM model, input encoder, and output encoder are loaded using joblib.load, as shown in Figure 50. The FaceNet model is initialized to extract face embeddings.

```
def upload_image():
    filename = filedialog.askopenfilename(filetypes=[("Image files", "*.jpg *.jpeg *.png")])
    if filename:
        img = Image.open(filename)
        img.thumbnail((400, 400))
        img = ImageTk.PhotoImage(img)
        image_label.config(image=img)
        image_label.image = img
        predict_image(filename)
```

Figure 51: Uploading Images

As it can be seen in Figure 51, a button allows users to upload images, which are then processed for face detection and recognition. The upload_image function uses filedialog to open an image file. The selected image is displayed in the GUI using ImageTk

```
def predict_image(filename):
    global current_canvas

    # Destroy the previous canvas if it exists
    if current_canvas:
        current_canvas.get_tk_widget().destroy()

    # Use keras_facenet to extract detections
    detections = embedder.extract(filename, threshold=0.95)

    if not detections:
        print('No face detected')
        messagebox.showerror("Error", "No face detected.")
        return

    # Load the image using OpenCV to draw rectangles
    image = cv2.imread(filename)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert BGR to RGB

    # Initialize figure for plotting
    fig, ax = plt.subplots(figsize=(10, 8))
    ax.imshow(image_rgb)
    ax.axis('off')
```

Figure 52: Predicting the classes for the image

```python
# Initialize a list to store the predicted names and probabilities
predictions = []

# Iterate over each detected face
for detection in detections:
    # Extract the embedding
    embedding = detection['embedding']
    embedding_norm = in_encoder.transform([embedding])

    # Predict the class and probability
    yhat_class = model.predict(embedding_norm)
    yhat_prob = model.predict_proba(embedding_norm)

    class_index = yhat_class[0]
    class_probability = yhat_prob[0, class_index] * 100
    predict_name = out_encoder.inverse_transform([class_index])[0]

    # Determine color for rectangle based on class
    color = (0, 255, 0)  # Green color for known faces
    # here We define our threshold for face reconition.
    if class_probability <= 85:
        predict_name = 'Unknown'
        color = (255, 0, 0)  # Red color for unknown faces
        class_probability = 0.0  # Set probability to 0 for unknown faces

    # Draw rectangle around the face
    x1, y1, width, height = detection['box']
    x2, y2 = x1 + width, y1 + height
    cv2.rectangle(image_rgb, (x1, y1), (x2, y2), color, 2)

    # Display predicted name
    text = f'{predict_name} ({class_probability:.3f}%)' if predict_name != 'Unknown' else f'{predict_name}'
    # Adjust text position to be visible even on small faces
    text_x = x1
    text_y = y1 - 10 if y1 - 10 > 10 else y1 + 20

    cv2.putText(image_rgb, text, (text_x, text_y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

    # Append the predicted name and probability to the list
    predictions.append(f'{predict_name}: {class_probability:.2f}%')
```

Figure 53: Predicting the classes for the image

```python
# Display the image with rectangles and predictions
ax.imshow(image_rgb)
ax.set_title('Face Recognition')

# Display the predicted names and probabilities as text below the image
fig.text(0.5, 0.01, '\n'.join(predictions), ha='center', fontsize=12, bbox=dict(facecolor='white', alpha=0.8))

# Embed the plot in the Tkinter window
current_canvas = FigureCanvasTkAgg(fig, master=root)
current_canvas.draw()
current_canvas.get_tk_widget().pack(pady=20)
```

Figure 54: Displaying the image with detected faces

As shown in Figure 52, Figure 53 and Figure 54, The predict_image function processes the uploaded image:

It extracts face embeddings using the FaceNet model.

If no face is detected, an error message is displayed.

For each detected face, embeddings are normalized and fed into the pre-trained SVM model to predict the class and probability.

If the predicted probability is below a threshold (85%), the face is labeled as "Unknown".

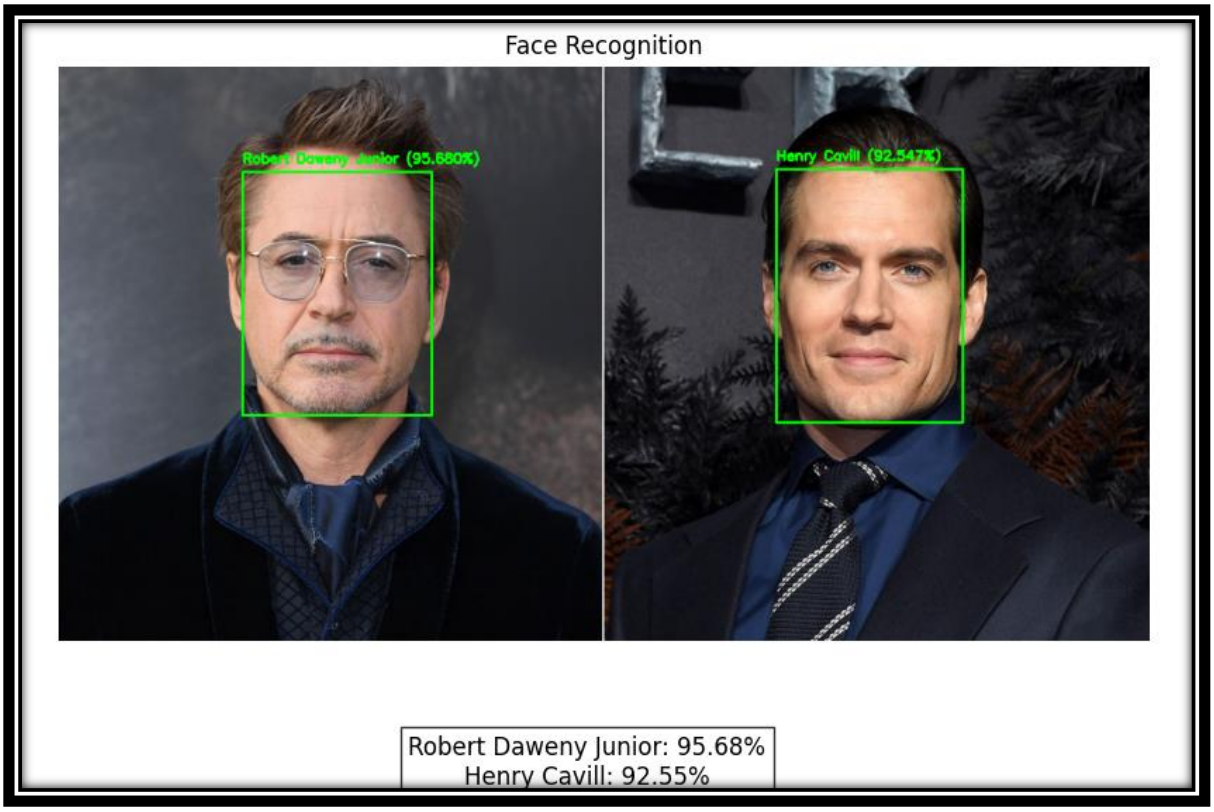
The results, including the annotated image with detected faces and prediction probabilities, are displayed using matplotlib within the Tkinter application.

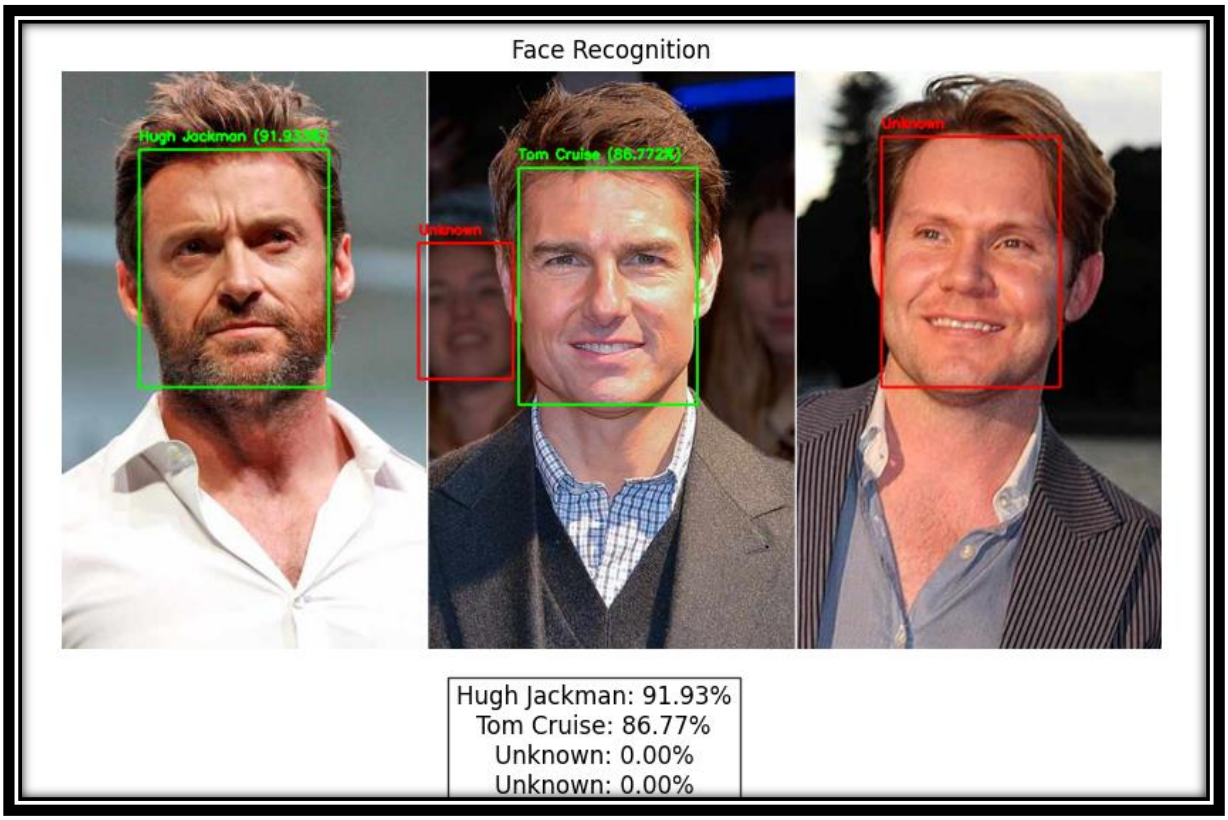
```
# Create the main window
root = tk.Tk()
root.title("Face Recognition")
root.configure(bg="black")

# Create a frame to hold the upload button and label
frame = tk.Frame(root, bg="black")
frame.pack(expand=True)

# Create widgets
upload_button = tk.Button(frame, text="Upload Image", command=upload_image, font=("Helvetica", 14), bg="green", fg="white")
upload_label = tk.Label(frame, text="Click the button to upload", bg="black", fg="white", font=("Helvetica", 12))
image_label = tk.Label(root, bg="black")  # Display the uploaded image

# Arrange widgets in the frame
upload_label.pack(pady=10)
upload_button.pack(pady=5)
image_label.pack(pady=20)

root.mainloop()
```

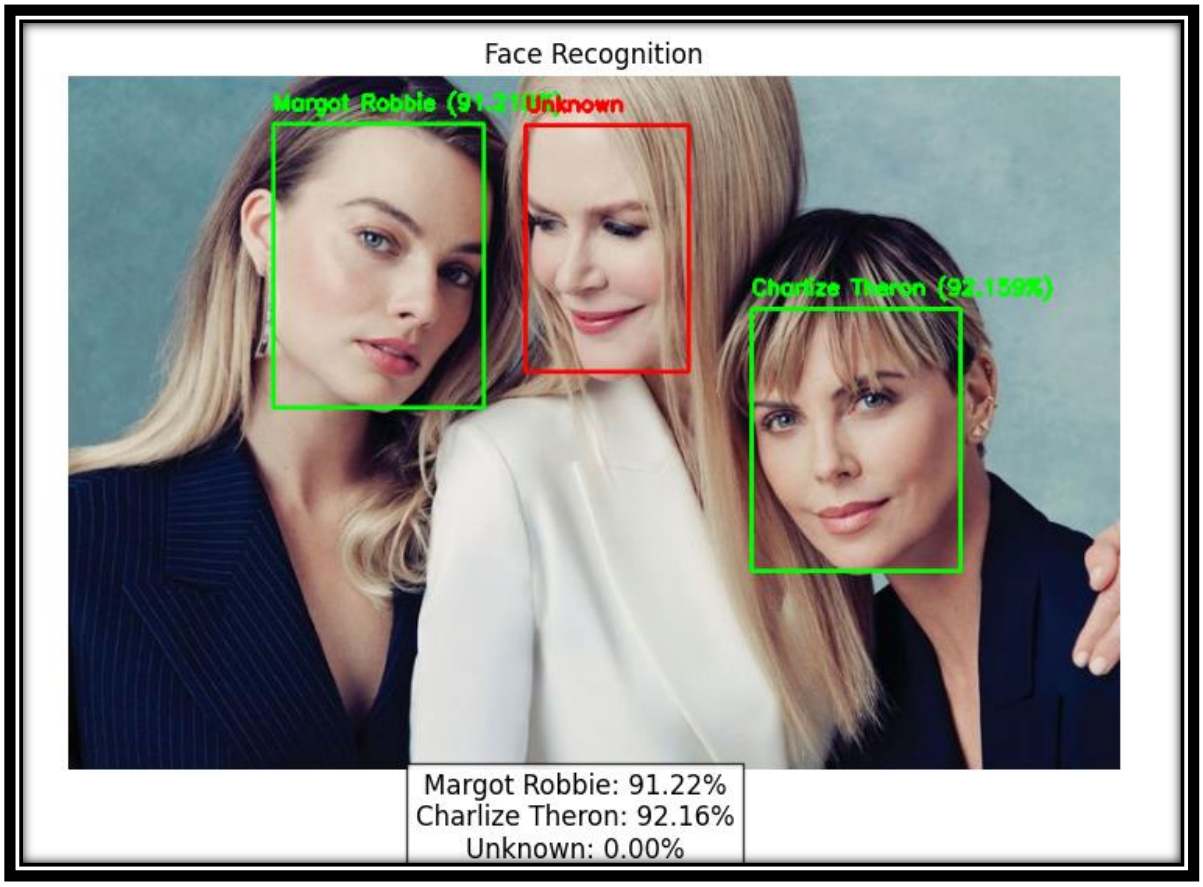
Figure 55: User Interface Layout

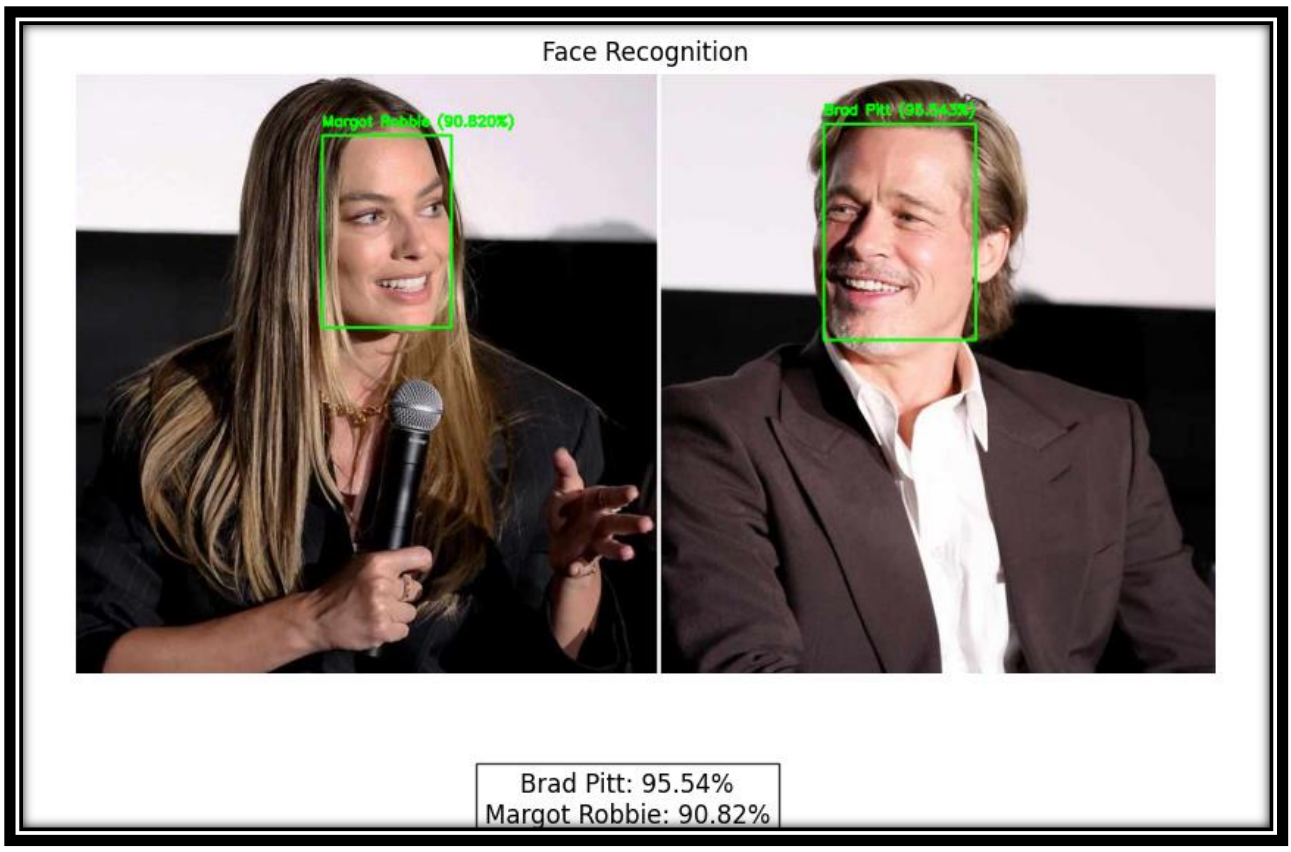
The main window of the Tkinter application is created with an upload button and a label for displaying images, this process in shown in Figure 55.

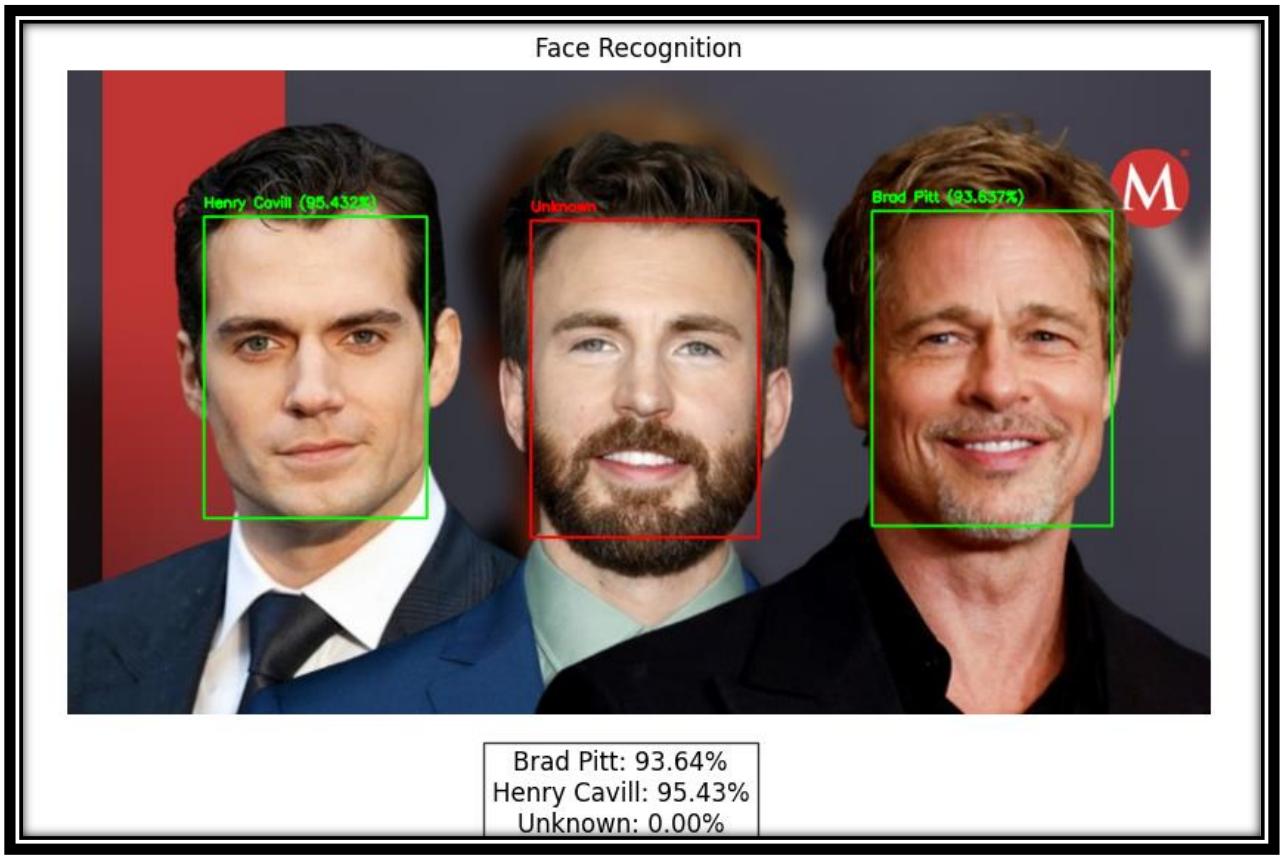
Some test images:


Face Recognition
Robert Daweny Junior (95.680%)
Henry Cavill (92.547%)
Robert Daweny Junior: 95.68%
Henry Cavill: 92.55%


Face Recognition
Hugh Jackman (91.93%)
Unknown
Tom Cruise (86.772%)
Unknown
Hugh Jackman: 91.93%
Tom Cruise: 86.77%
Unknown: 0.00%
Unknown: 0.00%


Face Recognition
Unknown
Margot Robbie: 91.22%
Charlize Theron: 92.16%
Unknown: 0.00%


Face Recognition
Margot Robbie (90.820%)
Brad Pitt (95.543%)
Brad Pitt: 95.54%
Margot Robbie: 90.82%


Face Recognition
Henry Cavill (95.432%)
Unknown
Brad Pitt (93.637%)
Brad Pitt: 93.64%
Henry Cavill: 95.43%
Unknown: 0.00%

This model of face recognition, performs very well in almost all kinds of scenarios. It works well by recognizing known persons on different scenarios, lighting conditions, and poses. It returns a high confidence score for correct classifications in most of the cases. Unlike simpler systems, it guards its integrity when presented with images of no faces or other unrelated content.

What really differentiates this model from others is the capability of actually telling apart known individuals who share similar features, with very high probabilities for the correct identity. Other highpoints for this model, in comparison to the VGG16-based implementation, are the unknown face identification capabilities. This model drops its confidence gracefully in difficult scenarios, rather than making incorrect high-confidence predictions. Therefore, doesn't identify unknown faces as known persons but marks them "Unknown" instead, with low confidence scores.

In images with multiple faces, it can detect and classify each face independently. This implementation has greatly reduced false positives to classes, especially the "Unknown" category, enhancing the general reliability of the system.

These improvements make the current implementation more robust, reliable, and practical for real-world face recognition tasks, thereby avoiding many limitations associated with previous approaches like the VGG16-based model.

Comparison and Conclusion

The improvement in performance observed with the FaceNet-based approach compared to the VGG16 implementation can be attributed to several key factors:

- FaceNet is specifically designed and optimized for face recognition tasks. It creates embeddings (high-dimensional vector representations) that are particularly effective at distinguishing between different faces.
- FaceNet uses a triplet loss during training, which ensures that faces of the same person are closer together in the embedding space while faces of different people are further apart. This directly optimizes the model for face verification and recognition tasks [13].
- VGG16 is a general-purpose convolutional neural network pre-trained on ImageNet, which contains images from a wide variety of categories (not specifically faces). While it captures useful features, it is not specialized for face recognition out of the box.
- The FaceNet model extracts embeddings that are immediately usable for classification tasks. These embeddings effectively represent facial features, making them highly suitable for subsequent classification with a simple model like SVM.
- When using VGG16, we extract features from the last convolutional layers (we freeze all the trained layers and add

a custom layer on top to train it on our dataset), which may not be as optimized for distinguishing between faces as FaceNet embeddings. This requires additional layers and fine-tuning to improve performance.

- VGG16 is a deep network with many layers and parameters. Fine-tuning such a model on a small dataset can lead to overfitting, as the network may memorize the training data without learning to generalize.
- Fine-tuning the VGG16 model typically requires more computational resources and careful hyperparameter tuning to avoid overfitting and achieve good performance.
- The FaceNet model, through its training process, becomes robust to various face transformations (e.g., different angles, lighting conditions), which helps in achieving better generalization on new faces.
- The embeddings generated by FaceNet are highly discriminative, leading to more confident and accurate predictions even on unseen data. The SVM classifier further ensures that the decision boundaries are well-defined.
- The higher complexity of VGG16 and potential overfitting on a small dataset can lead to high-confidence misclassifications, where the model is overly confident in its wrong predictions.

In summary, the remarkable performance of the FaceNet-based approach compared to the VGG16 implementation stems from FaceNet's specialized architecture for face recognition, efficient use of embeddings, and the simpler, more effective classification with an SVM. The general-purpose nature of VGG16, coupled with the challenges of training on a small dataset, contributes to its relatively poorer performance in this specific task.

Practical Application

With our newly developed face recognition model demonstrating high efficiency and precision in identifying trained individuals, we are now poised to create a range of practical applications that leverage this technology. One particularly useful implementation we are developing is an intelligent image sorting application.

This application is designed for the automation of organizing large sets of image files according to the presence of individuals who are being targeted. It goes through each image in a given dataset and recognizes individual faces with our face recognition model. When it finds a face that corresponds to one of our trained profiles, it makes a copy of the image within the folder that corresponds to the identified person.

This application is also flexible enough to take on searches for images that have multiple people. It can therefore be set up to include searches of images of many people of interest. In this case, the program will be seeking images for combinations of specified individuals in the same image. If there is a match to such an image, then it will copy the image to folders relating to each of the identified persons.

This tool has many practical applications, from organizing personal photos to professional archives. It saves much time and effort taken in going through large collections of images, thus being very useful not only for photographers and researchers but

for anyone dealing with large picture libraries. By way of automatization, our application turns something once tiring into a fast and efficient operation: namely, identifying and classifying images based on facial recognition. This gives real value to our face recognition model.

Program Summary

1. User Interface: It creates a GUI using tkinter, allowing users to interact with the facial recognition system easily.
2. Face Recognition Model: The program loads the pre-trained face recognition model from the previous section and also necessary encoders.
3. People Selection: It displays a list of recognizable people (based on the trained model) and allows users to select one or more individuals they want to find in images.
4. Folder Selection: Users can choose a folder containing images to be processed.
5. Image Processing:
    - The program goes through each image in the selected folder.
    - It uses FaceNet to detect faces in each image.
    - For each detected face, it uses the loaded model to predict who the person is.

- It checks if the recognized faces match the user's selected individuals.

6. Image Filtering and Copying:
    - If an image contains all the selected individuals, it's copied to a new folder.
    - The new folder is named after the selected individuals.
7. Completion Notification: Once all images are processed, it shows a message with the results.

```
# Load the model and encoders
model = joblib.load('face_classifier.pkl')
in_encoder = joblib.load('in_encoder.pkl')
out_encoder = joblib.load('out_encoder.pkl')
embedder = FaceNet()

# Get the list of recognizable people
recognizable_people = list(out_encoder.classes_)
```

Figure 56: Loading the saved model and encoders

Figure 56 shows that the pre-trained SVM model, input encoder, and output encoder are loaded using joblib.load. The FaceNet model is initialized for extracting face embeddings.

The list of recognizable people is obtained from the output encoder's classes.

```
# Create the main window
root = tk.Tk()
root.title("Select People for Image Filtering")
root.configure(bg="#2e2e2e")

frame = tk.Frame(root, bg="#2e2e2e")
frame.place(relx=0.5, rely=0.5, anchor=tk.CENTER)

label = tk.Label(frame, text="Select the people you want to filter images for:", bg="#2e2e2e", fg="white", font=("Helvetica", 16))
label.pack(pady=10)

listbox_frame = tk.Frame(frame, bg="#2e2e2e")
listbox_frame.pack(pady=5)
listbox_scrollbar = tk.Scrollbar(listbox_frame, orient=tk.VERTICAL)
listbox = tk.Listbox(listbox_frame, selectmode=tk.MULTIPLE, height=10, bg="#404040", fg="white", font=("Helvetica", 14), yscrollcommand=listbox_scrollbar.set)
listbox_scrollbar.config(command=listbox.yview)
listbox_scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
listbox.pack(side=tk.LEFT, fill=tk.BOTH)

for person in recognizable_people:
    listbox.insert(tk.END, person)

button = tk.Button(frame, text="Select Folder and Process Images", command=process_images, font=("Helvetica", 14), bg="#007acc", fg="white", width=30)
button.pack(pady=20)



status_label = tk.Label(frame, text="Status: Waiting for user input...", fg="white", bg="#2e2e2e", font=("Helvetica", 14))
status_label.pack(pady=10)

root.mainloop()
```

Figure 57: User Interface

The main window of the Tkinter application is created with a listbox for selecting people, a button to start the image processing, and a status label to display progress, this process can be seen in Figure 57.

```python
def process_images():
    selected_people = [listbox.get(i) for i in listbox.curselection()]
    if not selected_people:
        messagebox.showerror("Error", "No people selected.")
        return

    folder_path = filedialog.askdirectory()

    if not folder_path:
        messagebox.showerror("Error", "No folder selected.")
        return

    # Create a new folder named after the selected people if it doesn't already exist
    save_folder_name = '-'.join(selected_people)
    save_folder_path = os.path.join('filtered_images', save_folder_name)
    if not os.path.exists(save_folder_path):
        os.makedirs(save_folder_path)
```

Figure 58: Processing Images

```python
for filename in os.listdir(folder_path):
    if filename.endswith(('.jpg', '.jpeg', '.png')):
        file_path = os.path.join(folder_path, filename)
        detections = embedder.extract(file_path, threshold=0.95)
        recognized_faces = []

        for detection in detections:
            embedding = detection['embedding']
            embedding_norm = in_encoder.transform([embedding])
            yhat_class = model.predict(embedding_norm)
            yhat_prob = model.predict_proba(embedding_norm)
            class_index = yhat_class[0]
            class_probability = yhat_prob[0, class_index] * 100
            predict_name = out_encoder.inverse_transform([class_index])[0]

            if class_probability > 85:
                recognized_faces.append(predict_name)
            else:
                recognized_faces.append('Unknown')

        if set(selected_people).issubset(set(recognized_faces)):
            shutil.copy(file_path, os.path.join(save_folder_path, filename))
            matched_count += 1

        processed_count += 1
        status_label.config(text=f"Processed {processed_count} images. Matched {matched_count} images.")
        root.update_idletasks()

messagebox.showinfo("Completed", f"Processing completed. {matched_count} images matched the criteria and were copied to {save_folder_pa
```

Figure 59: Processing Images

Figure 58 and Figure 59 show how the images are processed. The process_images function is responsible for processing images based on user-selected criteria:

1. **User Selection**: It retrieves the list of selected people from the listbox, which is shown in Figure 60.
2. **Folder Selection**: It opens a file dialog for the user to select a folder containing images.
3. **Directory Creation**: It creates a new directory named after the selected people if it does not already exist.
4. **Image Processing**: It processes each image in the selected folder:
    - Face detection and embedding extraction using FaceNet.
    - Embeddings are normalized and passed through the pre-trained SVM model for classification.
    - If the predicted probability for a person exceeds 85%, the face is considered recognized.
5. **Image Copying**: If all selected people are recognized in an image, the image is copied to the new directory.
6. **Status Update**: The status label is updated with the progress, and a message box is displayed upon completion, as it can be seen in Figure 62.

Let’s say we want to find all the images with Brad Pitt and Margot Robbie in them and copy it the Brad Pitt – Margot Robbie folder.

All we need to do is to open the application and select Brad Pitt and Margot Robbie from the list of people our model is capable of recognizing and then choose the directory we want to search in.

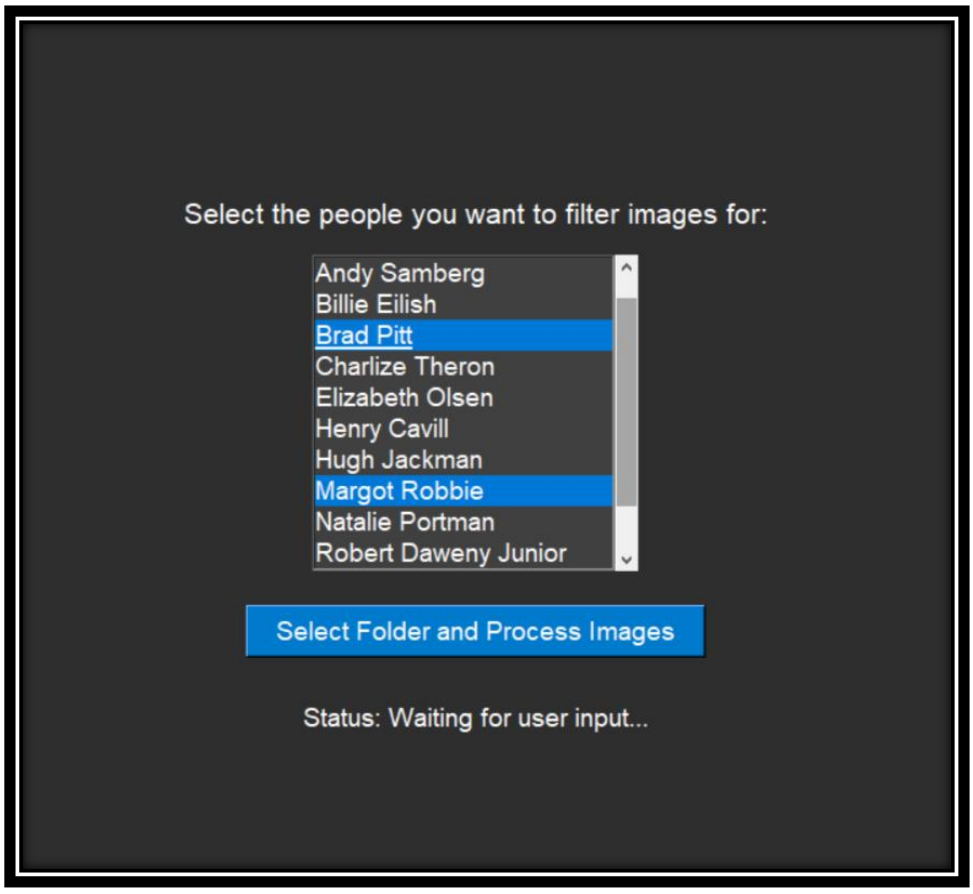


Figure 60: The program’s User Interface

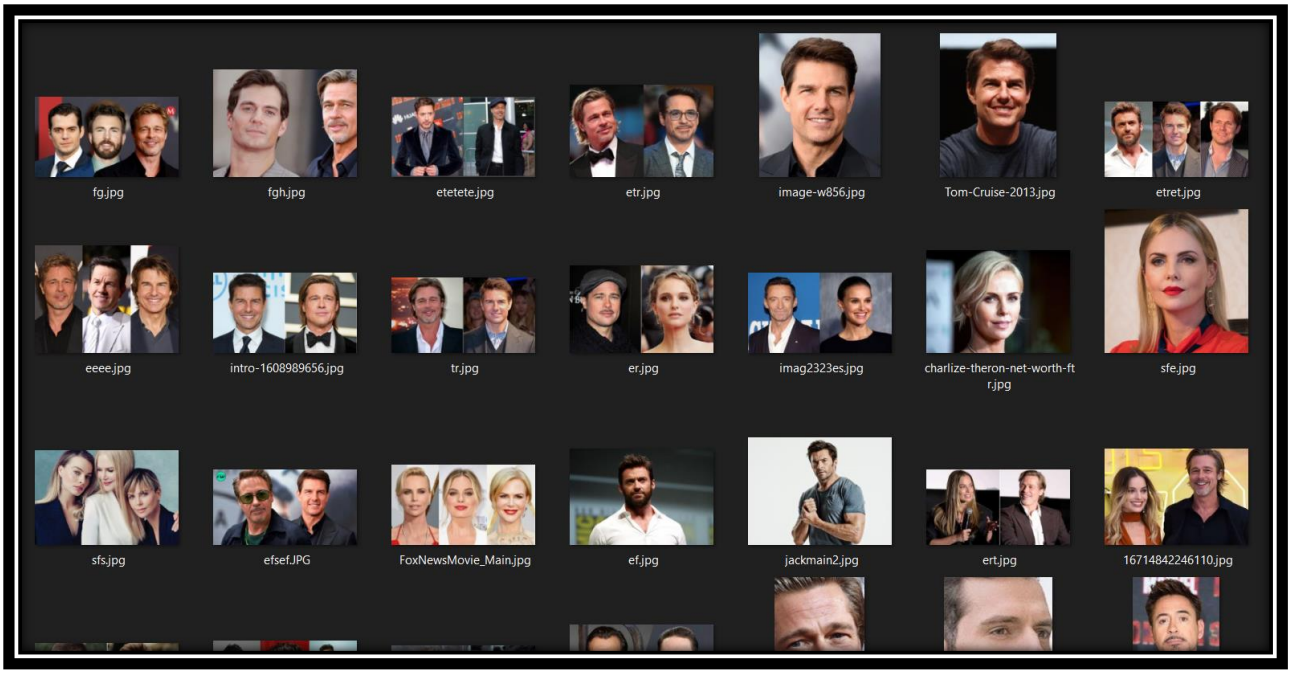


Figure 61: The directory we want to search in, containing multiple new random images from the same classes the model is trained on that includes unknown people as well

Figure 62: Completion message

The images containing Brad Pitt and Margot Robbie are copied into a folder with their names, as shown in Figure 63.

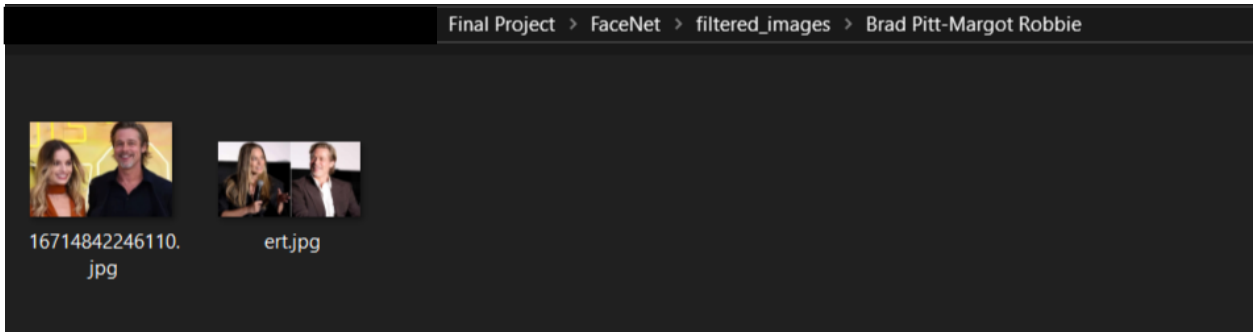


Figure 63: Brad Pitt-Margot Robbie folder

The End

References

1. **Hameed, I. M., Abdulhussain, S. H., Mahmmod, B. M., & Pham, D. T. (2021). Content-based image retrieval: A review of recent trends. Cogent Engineering, 8(1). https://doi.org/10.1080/23311916.2021.1927469**
2. **Mahajan, A., Chaudhary, S., & Degadwala, S. (2021). Comparative Analysis on Hybrid Content & Context-basedimage Retrieval System. Journal of Information Technology Management, 13(Special Issue: Big Data Analytics and Management in Internet of Things), 133-142. doi: 10.22059/jitm.2021.80765**
3. **AlyanNezhadi, M.M., Qazanfari, H., Ajam, A., & Amiri, Z. (2020). Content-based Image Retrieval Considering Colour Difference Histogram of Image Texture and Edge Orientation. International Journal of Engineering. DOI:10.5829/ije.2020.33.05b.28**
4. **Sarker, I.H. Machine Learning: Algorithms, Real-World Applications and Research Directions. SN COMPUT. SCI. 2, 160 (2021). https://doi.org/10.1007/s42979-021-00592-x**
5. **Publication, I. J. R. A. S. E. T. (2020). Study of Supervised Learning and Unsupervised Learning. International Journal for Research in Applied Science and Engineering Technology Ijraset. https://doi.org/10.22214/ijraset.2020.6095**
6. **Saraswat, P. (2022). Supervised Machine Learning Algorithm: A Review of Classification Techniques. In: García Márquez, F.P. (eds) International Conference on Intelligent Emerging Methods of Artificial Intelligence & Cloud Computing. IEMAICLOUD 2021. Smart Innovation, Systems and Technologies, vol 273. Springer, Cham. https://doi.org/10.1007/978-3-030-92905-3_58**
7. **Dubey, S.R. (2020). A Decade Survey of Content Based Image Retrieval Using Deep Learning. IEEE Transactions on Circuits and Systems for Video Technology, 32, 2687-2704.**
8. **Nash, R. (2015). An Introduction to Convolutional Neural Networks. ArXiv. /abs/1511.08458**
9. **Li, Z., Liu, F., Yang, W., Peng, S., & Zhou, J. (2020). A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. IEEE Transactions on Neural Networks and Learning Systems, 33, 6999-7019.**
10. **GeeksforGeeks. VGG-16 CNN model. https://www.geeksforgeeks.org/vgg-16-cnn-model/**
11. **Boesch, G. Very deep convolutional networks (VGG): Essential guide. Viso.ai. https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/**

12. **Akundi, P. (2020, March 20). Unknown unknowns: How to train a CNN that's ready for anything with softmax thresholding. Medium. https://medium.com/@prathyuakundi/unknown-unknowns-how-to-train-a-cnn-thats-ready-for-anything-with-softmax-thresholding-20cba0496990**
13. **Schroff, F., Kalenichenko, D., & Philbin, J. (2015). FaceNet: A unified embedding for face recognition and clustering. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 815-823.**
14. **Dagher, Issam & Azar, Fady. (2019). Improving the SVM gender classification accuracy using clustering and incremental learning. Expert Systems. 36. e12372. 10.1111/exsy.12372. https://www.researchgate.net/figure/Local-binary-pattern-LBP-feature-extraction-method_fig3_330519739**
15. **Fu, Chen & Yang, Jianhua. (2021). Granular Classification for Imbalanced Datasets: A Minkowski Distance-Based Method. Algorithms. 14. 54. 10.3390/a14020054. https://www.researchgate.net/figure/Three-typical-Minkowski-distances-ie-Euclidean-Manhattan-and-Chebyshev-distances_fig1_349155159**
16. **Prof. Pascal Tyrrell. (2021). Cosine Distance https://www.tyrrell4innovation.ca/miword-of-the-day-iscosine-distance/**
17. **Baeldung. (2024, March 18). What is content-based image retrieval? Retrieved from https://www.baeldung.com/cs/cbir-tbir**
18. **Pang, H., Moore, K., Padmanabha, A., & others. Feature vector. Retrieved from https://brilliant.org/wiki/feature-vector/#citation-2**
19. **Sharma, P. (2021, April). K means clustering in Python | Step-by-step tutorials for clustering in data analysis. Retrieved from https://www.analyticsvidhya.com/blog/2021/04/k-means-clustering-simplified-in-python/**
20. **Autonise. K-means + SVM (support vector machine) | Clustering | Unsupervised learning [Video]. YouTube. https://www.youtube.com/watch?v=Mfn0zwAt27o&ab_channel=Autonise**
21. **Visually Explained. Support vector machine (SVM) in 2 minutes [Video]. YouTube. https://www.youtube.com/watch?v=_YPScrckx28&ab_channel=VisuallyExplained**
22. **Wu, E. (2019, November 14). Supervised vs. unsupervised machine learning: Which is better for network threat detection, and why? Retrieved from https://www.extrahop.com/blog/supervised-vs-unsupervised-machine-learning-for-network-threat-detection**
23. **Haque, K. N. What is convolutional neural network — CNN (deep learning). Retrieved from https://www.linkedin.com/pulse/what-convolutional-neural-network-cnn-deep-learning-nafiz-shahriar**
24. **IBM. What are convolutional neural networks? Retrieved from https://www.ibm.com/topics/convolutional-neural-networks**

25. GeeksforGeeks. CNN | Introduction to pooling layer. Retrieved from https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/
26. Indian Tech Warrior. Fully connected layers in convolutional neural networks. Retrieved from https://indiantechwarrior.com/fully-connected-layers-in-convolutional-neural-networks/
27. Roboflow. An introduction to ImageNet. Retrieved from https://blog.roboflow.com/introduction-to-imagenet/
28. Wikipedia. FaceNet. Retrieved from https://en.wikipedia.org/wiki/FaceNet
29. Mandal, M. (2021, May). Introduction to convolutional neural networks (CNN). Retrieved from https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/
30. Walker, S. M. II. F-Score: What are accuracy, precision, recall, and F1 score? Retrieved from https://klu.ai/glossary/accuracy-precision-recall-f1
31. https://www.kaggle.com/datasets/vasukipatel/face-recognition-dataset?select=Original+Images
32. https://pypi.org/project/mtcnn/#zhang2016
33. 3Blue1Brown. Neural networks [YouTube playlist]. YouTube. Retrieved from https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
34. Mazur, M. (2015, March 17). *A step by step backpropagation example*. Matt Mazur. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
35. Yang, J., Zhou, K., Li, Y. et al. Generalized Out-of-Distribution Detection: A Survey. Int J Comput Vis (2024). https://doi.org/10.1007/s11263-024-02117-4