**SUMMARY**

*In our project, we’re using three distinct models—PCA, ORB, and Autoencoder—to classify images as either "hate" or "nonhate." Each model independently processes the images and outputs predictions. Specifically:*

***PCA Model:*** *This model uses Principal Component Analysis for dimensionality reduction, followed by KMeans clustering to classify image features into "hate" and "nonhate" categories.*

***ORB Model:*** *The ORB (Oriented FAST and Rotated BRIEF) model extracts key image features, creating a feature vector that characterizes the image's structure. KMeans clustering is then used to categorize these features.*

***Autoencoder Model:*** *This neural network-based model learns compressed representations of images, helping to detect features relevant to classification. The output is then clustered for "hate" and "nonhate" classification.*

*Finally, a voting mechanism combines the predictions. If at least two out of the three models agree on a classification, it’s assigned to the image. This approach leverages the unique strengths of each model, increasing the reliability of the final classification output.*

*We will be briefly describing below each of the three models . We cover how the model uses its distinct features to classify the images.*

**PCA MODEL**

***from sklearn.decomposition import PCA***

*PCA, which stands for Principal Component Analysis, is a technique for dimensionality reduction that helps simplify complex data. PCA reduces the number of dimensions (features) in a dataset while retaining as much of the original information as possible. The transformed data (principal components) can be used as new features for further processing or analysis, often revealing underlying patterns in complex datasets.*

***pca = PCA(n\_components=64)***

*When we set n\_components=64 in PCA, we’re specifying that PCA should reduce our data to 64 principal components. This means that our high-dimensional data will be projected down to 64 dimensions, keeping the 64 directions of highest variance in the data. Reducing data to 64 components can greatly reduce its size while retaining most of the important information, especially in high-dimensional data like images.*

***features = pca.fit\_transform(train\_images.reshape(len(train\_images), -1))***

*applies PCA to your image data after flattening it, reducing it to a lower-dimensional representation. reshapes each image in train\_images into a 1D array. If train\_images contains images of shape (128, 128, 3), each image is flattened to a 1D array of length 128×128×3=49,152*

*After reshaping, train\_images has the shape (number\_of\_images, 49152).*

*pca.fit\_transform(...) Transforms the flattened images to a new, lower-dimensional space (64 dimensions if n\_components=64 was set in PCA).The result, features, has shape (number\_of\_images, 64), where each image is represented by a 64-dimensional feature vector.*

*Let's assume: We have 5 images with dimensions 128x128 pixels and 3 color channels (RGB). Each image has 128 × 128 × 3 = 49,152 elements in total. The train\_images array would have the shape (5, 128, 128, 3), representing 5 images with the given pixel dimensions. Step 1: Flattening the Images Before applying PCA, we flatten each image into a 1D vector (49,152 pixels per image). Using train\_images.reshape(len(train\_images), -1), we get an array of shape (5, 49152)*

*A portion of train\_images.reshape(len(train\_images), -1) might look like:*

*[ 45 , 68 , 255 , 132 , … , 76*

*50 , 120 , 240 , 200 , … , 150*

*10 , 90 , 210 , 180 , … , 35*

*75 , 130 , 180 , 220 , … , 90*

*255 , 100 , 160 , 170 , … , 180 ] ​*

*Fitting the PCA Model: PCA analyzes the pixel data across all images to identify 64 "principal components" (or directions) that capture the most variance in the data. These principal components are linear combinations of the original pixel dimensions and represent key features in the image data.*

*Transforming the Data: Each image is then projected onto these 64 principal components, effectively reducing the dimensionality from 49,152 to 64.*

*[ 2.4 , − 1.1 , 0.5 , … , − 3.2*

*1.3 , − 0.8 , 1.2 , … , 4.0*

*− 2.0 , 1.5 , − 0.3 , … , − 1.1*

*3.1 , − 0.9 , 2.4 , … , 1.8*

*− 1.5 , 2.3 , − 2.1 , … , − 0.5 ] ​*

***##############################################***

**ORB MODEL**

*The extract\_orb\_features function is designed to extract ORB (Oriented FAST and Rotated BRIEF) features from an image and transform these features into a fixed-length feature vector. This vector can then be used for machine learning tasks, such as clustering or classification. Here’s a detailed breakdown of each step in this function:*

*1. ORB Initialization*

***orb = cv2.ORB\_create()***

*This line initializes the ORB feature detector using OpenCV's cv2.ORB\_create(). ORB is an efficient algorithm for keypoint detection and descriptor extraction. It identifies distinctive points (keypoints) in the image and encodes these points into descriptors, which are vectors representing local image features.*

*2. Detecting Keypoints and Computing Descriptors*

***keypoints, descriptors = orb.detectAndCompute(image, None)***

*detectAndCompute: This function detects keypoints in the image and then computes descriptors for each detected keypoint.*

*Keypoints: These are the points in the image where the ORB algorithm has identified distinctive features (e.g., corners or edges).*

*Descriptors: These are binary vectors (32 bytes each) that represent the local visual characteristics around each keypoint.*

*If the image contains distinct textures, edges, or other patterns, ORB will detect keypoints and produce descriptors for them. However, if the image is blank or lacks texture, descriptors may be None.*

*3. Handling the Maximum Number of Features*

***if descriptors is not None:***

***if len(descriptors) > max\_features:***

***descriptors = descriptors[:max\_features]***

*The function is configured to extract up to max\_features descriptors. If more descriptors are detected, it truncates the list to the first max\_features. This ensures uniformity across images by limiting the maximum number of descriptors per image, which is essential for consistent feature vector size. If the number of descriptors is below max\_features, all available descriptors are retained.*

*4. Creating a Fixed-Length Feature Vector*

***feature\_vector = np.zeros(max\_features \* 32)***

***feature\_vector[:descriptors.shape[0] \* 32] = descriptors.flatten()***

*Fixed-Length Vector: To make this feature extraction compatible with machine learning models, the function creates a fixed-length feature vector of size max\_features \* 32, where 32 bytes represent each descriptor.*

*The descriptors array is flattened into a single one-dimensional vector. This is done because machine learning models generally expect inputs of a consistent shape. If fewer than max\_features descriptors are found, the remaining elements in feature\_vector stay as zeros. This padding ensures that the feature vector has a fixed length regardless of how many descriptors were extracted.*

*5. Handling No Descriptors Case*

***else:***

***feature\_vector = np.zeros(max\_features \* 32)***

*If no descriptors were found (e.g., in a low-texture image), the function returns a zero-filled feature vector. This uniformity ensures the model receives feature vectors of consistent shape, which is crucial for training stability.*

***###############################################***

**ENCODER MODEL**

***from tensorflow.keras import layers, models***

*We are importing key modules from TensorFlow's Keras API to build and structure neural networks:*

*layers: Provides various pre-built neural network layers (like Dense, Conv2D, Flatten, etc.), which serve as building blocks for designing models. Each layer processes data in a unique way (e.g., convolutional layers for feature extraction, dense layers for classification).*

*models: Contains tools to create and manage complete neural network models, such as the Sequential and Model classes, which help define and compile models by stacking or connecting layers in a specific order or configuration.*

**Encoding Process:**

***input\_img = layers.Input(shape=(IMG\_HEIGHT, IMG\_WIDTH, 3))***

*Defines the input layer of a neural network model using Keras, specifying the shape of the input data that the model will accept. Here, the model expects input images with dimensions corresponding to IMG\_HEIGHT and IMG\_WIDTH, along with 3 color channels (representing RGB).*

***x = layers.Flatten()(input\_img)***

*The model takes input of dimension 128 x 128 x 3 resulting in 49,152 elements. So an image is flattened. Ex. [255,65,0,....] with 49,152 elements.*

***x = layers.Dense(128, activation='relu')(x)***

*Creates a Dense layer with 128 neurons. Each of the 128 neurons will have its own set of weights and biases. The weights are randomly initialized. Each neuron will calculate its output using the formula:*

*output=activation( n∑* *i=1 ​ weight i​ × inputi ​ +bias)*

*Here, n is 49,152, which is the number of inputs for each neuron. Let's assume that for the first neuron in this layer, we have the following:*

*Weights: [0.1, -0.2, 0.3, ..., 0.001] (49,152 weights) (By having unique weights and biases, each of the 128 neurons in the layer can focus on different aspects of the input image, such as edges, textures, or colors and produce a diverse set of outputs even from the same input, which helps the model learn a variety of features.)*

*Bias: 0.5 (The bias allows the neuron to output a non-zero value even when all inputs are zero, which helps the network learn more effectively.)*

*The calculation for the first neuron would look something like this:*

*neuron\_output = (0.1 \* 255) + (-0.2 \* 0) + (0.3 \* 65) + ... + (0.001 \* last\_value)+ 0.5 (one neuron output)*

*final output = [0.23, 0.0, 0.45, 1.2, ..., 0.0] # 128 neuron values*

*If the neuron\_output is negative, the ReLU function will output 0. If it is positive, it will output the value as is.*

***encoded = layers.Dense(64, activation='relu')(x)***

*x = [0.23, 0.0, 0.45, 1.2, ..., 0.0] # 128 values*

*Since there are 128 inputs, there will be 128 random weights generated for each neuron. For one neuron in the 64-neuron layer:*

*Weights might look like: [0.05, -0.2, 0.1, 0.3, ..., 0.02] (128 values).*

*Bias might be 0.1.*

*The output of this neuron would be: output = ReLU ( ( 0.23 × 0.05 ) + ( 0.0 × − 0.2 ) + ( 0.45 × 0.1 ) + ( 1.2 × 0.3 ) + … + ( 0.0 × 0.02 ) + 0.1 ) output=ReLU((0.23×0.05)+(0.0×−0.2)+(0.45×0.1)+(1.2×0.3)+…+(0.0×0.02)+0.1)*

*encoded=[0.71,1.1,0.0,…] #64 values*

**Decoding Process:**

***x = layers.Dense(128, activation='relu')(encoded)***

*encoded=[0.71,1.1,0.0,…] #64 values*

*output=activation( n∑* *i=1 ​ weight i​ × inputi ​ +bias)*

*Each of the 128 neurons in this new layer will have its own set of 64 weights (one for each value in the encoded vector).*

*output=ReLU((encoded1​×w1​)+(encoded2​×w2​)+…+(encoded64​×w64​)+bias)*

*x = [0.25, 0.0, 1.12, 0.0, 0.78, ..., 0.0] # 128 values*

***x = layers.Dense(IMG\_HEIGHT \* IMG\_WIDTH \* 3, activation='sigmoid')(x)***

*We are recreating back the 49,152 elements which was the original number of pixels. Each of the neurons in this layer takes in a weighted sum of the 128 outputs from the previous layer (128 neurons). For each neuron 𝑖 i in this layer, the output value is calculated as follows:*

*output 𝑖 = 𝜎 ( 128 ∑ 𝑗 = 1 ( 𝑤 𝑖 𝑗 ⋅ 𝑥 𝑗 ) + 𝑏 𝑖 )*

*we apply the sigmoid activation function :*

*σ(z)= 1/(1+e −z )*

*Let’s assume we have:*

*Outputs from the previous layer, x = [0.23, 0.0, 0.45, ..., 0.0]*

*Weights for the neuron in this layer, w = [0.05, -0.2, 0.1, ..., 0.02]*

*Bias for this neuron, 𝑏 =0.1*

*For each 𝑗 from 1 to 128, multiply 𝑤 𝑖 𝑗 by 𝑥 𝑗 , Let’s assume:*

*(0.23×0.05=0.0115 + 0.0 ×−0.2=0.0 + 0.45×0.1=0.045 + ......) =0.59*

*z=0.59+0.1(bias)=0.69*

*output=σ(z) or output=σ(0.69) = 1/(1+e−0.691​)≈0.665*

*After performing this operation for each of the 49,152 neurons in the layer, we would have a list of outputs like [0.665, 0.234, 0.782, ..., 0.501], each representing a pixel intensity in the normalized range.*

***decoded = layers.Reshape((IMG\_HEIGHT, IMG\_WIDTH, 3))(x)***

*Suppose we have x as a 1D array with sample values after the final Dense layer:*

*𝑥 = [ 0.665 , 0.234 , 0.782 , 0.501 , 0.345 , . . . , 0.212 ]*

*The Reshape layer takes these values and organizes them into a 3D tensor. In this case, it will transform x into an array with shape (128, 128, 3), where each pixel is represented by three values.*

*decoded = [ [[0.665, 0.234, 0.782], [0.501, 0.345, 0.212], ...],*

*[[0.324, 0.567, 0.789], [0.123, 0.654, 0.432], ...],*

*[[0.524, 0.867, 0.989], [0.223, 0.554, 0.332], ...],*

*... ]*

***autoencoder = models.Model(input\_img, decoded)***

*The actual autoencoder model is created. The autoencoder can now process images, compress them through encoding, and attempt to reconstruct them through decoding.*

***autoencoder.compile(optimizer='adam', loss='mse')***

*The compile() method configures the model for training. The Adam optimizer is a popular optimization algorithm used in machine learning. It stands for Adaptive Moment Estimation. Adam adjusts the learning rate during training based on the first and second moments of the gradients, which helps in faster convergence. loss='mse': Here, mse stands for Mean Squared Error, which is a common loss function used for regression problems and is suitable for autoencoders, which aim to reconstruct their input. The Mean Squared Error calculates the average squared difference between the predicted values (output of the autoencoder) and the actual values (input images).*

*Mathematically, it is defined as:*

MSE = 1/ 𝑛 . 𝑛∑ 𝑖 = 1 ( 𝑦 𝑖 − 𝑦 ^ 𝑖 ) 2

*where 𝑦 𝑖  is the actual value, 𝑦 ^ 𝑖 is the predicted value, and 𝑛 is the number of samples.*

***autoencoder.fit(images, images, epochs=50, batch\_size=32, shuffle=True)***

*initiates the training process . For each image in images, the autoencoder compresses it down in the encoder, then reconstructs it in the decoder. This full encoding-decoding cycle happens within each pass of the batch. The images are processed in batches of 32, meaning 32 images are loaded and reconstructed at a time, which helps speed up training by optimizing memory and computation. The order of images is shuffled before each epoch to ensure the model doesn’t learn any sequence-specific patterns from the data order, promoting better generalization. Each image is trained for 50 epochs, or cycles. Over these cycles, the model continually refines its weights to minimize the difference (error) between the original and reconstructed images.*

***autoencoder.save('models/deep\_cluster\_autoencoder.keras')***

*saves the entire trained autoencoder model, including learned weights, architecture, and configuration, to a file. This saved model can then be reloaded later to perform encoding-decoding on new images or continue training*

***###############################################***

**KMEANS MODEL CLUSTERING**

***from sklearn.cluster import KMeans***

*We are importing the KMeans class from the sklearn.cluster module in scikit-learn. KMeans is an unsupervised machine learning algorithm used for clustering data into distinct groups or "clusters" based on feature similarity. It works by iteratively assigning data points to one of k clusters (where k is a user-defined number), minimizing the distance between each data point and the center of its assigned cluster, known as the centroid.*

***features = encoder.predict(images)***

*It takes the images array, which contains the raw pixel data of the input images. t uses the encoder model that was defined earlier, which represents the encoding layers of the trained autoencoder. The model processes the input images through its architecture. The predict() function computes the encoded representations of the input images. This means it transforms the high-dimensional input images into lower-dimensional feature vectors, capturing the essential characteristics of the images.*

*Here’s an example of a features array with a shape of (10, 64), representing 10 samples each encoded into a 64-dimensional feature vector:*

*[[0.5488135, 0.71518937, 0.60276338, ..., 0.94466892, 0.52184832, 0.41466194], [0.26455561, 0.77423369, 0.45615033, ..., 0.38344152, 0.79172504, 0.52889492], [0.56804456, 0.92559664, 0.07103606, ..., 0.87001215, 0.20218397, 0.16055113], ... [0.6818203, 0.3595079, 0.43703195, ..., 0.57019677, 0.43860151, 0.98837384]]*

*Each inner array represents a different sample, and each number is one of the 64 values in that feature vector. The values are randomly generated for this example, so our actual encoded values would depend on the specific images and the encoding process we implemented.*

***features = normalize(features)***

*By normalizing, we scale the feature vectors to have a unit length (norm of 1). This is especially important in machine learning to ensure that all features contribute equally to distance calculations, which can improve the performance of algorithms like K-Means clustering. After normalization, the relative proportions of the feature values are preserved, but the scale is adjusted so that each vector has a consistent length.*

*Suppose we have a small 2D array (matrix) called features:*

*features = np.array([ [3, 4, 0], [1, 2, 2], [0, 0, 1] ])*

*L2 norm= √(3 2 +4 2 +0 2 ​) =√ 25 ​ =5*

*√(12 +22 +22 ) = √9=3*

*√(02 +02 +12 ) = √1=1*

*Normalization would be*

*([ [3/5, 4/5, 0/5], [1/3, 2/3, 2/3], [0/1, 0/1, 1/1] ])*

*or [0.6, 0.8, 0], [0.33, 0.67, 0.67], [0, 0, 1]*

*Example Input:*

*features = [ [0.5488135, 0.71518937, 0.60276338, ..., 0.94466892, 0.52184832, 0.41466194],*

*[0.26455561, 0.77423369, 0.45615033, ..., 0.38344152, 0.79172504, 0.52889492],*

*[0.56804456, 0.92559664, 0.07103606, ..., 0.87001215, 0.20218397, 0.16055113], ... , [0.6818203, 0.3595079, 0.43703195, ..., 0.57019677, 0.43860151, 0.98837384] ]*

*Example Output:*

*normalized\_features = [ [0.26726124, 0.35617761, 0.30285549, ..., 0.44695351, 0.25047355, 0.20439306], [0.17077337, 0.51962645, 0.30801911, ..., 0.26145382, 0.53728599, 0.36931255], [0.42290544, 0.68254702, 0.05107417, ..., 0.61934066, 0.14671161, 0.11425659], ..., [0.40894638, 0.20969435, 0.24740502, ..., 0.33268714, 0.26179251, 0.57667957] ]*

***kmeans = KMeans(n\_clusters=2, random\_state=0)***

*KMeans: This is a class from the sklearn.cluster module that implements the K-Means clustering algorithm. n\_clusters=2: This parameter specifies the number of clusters (groups) we want the algorithm to identify in our data. In this case, we're asking it to find 2 clusters. random\_state=0: This is used to ensure reproducibility. Setting a random state makes sure that the results are the same each time we run the code, which is helpful for debugging and understanding the model's performance.*

***kmeans.fit(features)***

*fit: This method trains the K-Means algorithm using the input data, which in this case is the normalized features array. During this step, the algorithm iterates to assign each feature vector to one of the 2 clusters. It calculates the centroids (average points) of the clusters and assigns each data point (encoded feature vector) to the nearest centroid.*

*Let’s walk through the K-Means clustering process with specific calculations for the provided 64-dimensional feature vectors. We will use Euclidean distance to perform the clustering. Here's a detailed example that shows how the algorithm works step by step.*

*We have four feature vectors (64-dimensional):*

*Image 1: [0.26726124, 0.35617761, 0.30285549, ..., 0.44695351, 0.25047355, 0.20439306]*

*Image 2: [0.17077337, 0.51962645, 0.30801911, ..., 0.26145382, 0.53728599, 0.36931255]*

*Image 3: [0.42290544, 0.68254702, 0.05107417, ..., 0.61934066, 0.14671161, 0.11425659]*

*Image 4: [0.40894638, 0.20969435, 0.24740502, ..., 0.33268714, 0.26179251, 0.57667957]*

*For simplicity, I will only use the first three dimensions in the calculations to illustrate the process. Let's assume our vectors are:*

*Image 1: [0.267, 0.356, 0.302]*

*Image 2: [0.171, 0.520, 0.308]*

*Image 3: [0.423, 0.683, 0.051]*

*Image 4: [0.409, 0.210, 0.247]*

*Step 1: Initialization*

*Assume we set K = 2 (i.e., we want to create 2 clusters) and randomly choose two initial centroids:*

*Centroid 1 (C1): Image 1: [0.267,0.356,0.302] [0.267,0.356,0.302]*

*Centroid 2 (C2): Image 3: [0.423,0.683,0.051] [0.423,0.683,0.051]*

*Step 2: Assignment Step: Calculate the Euclidean distance from each image to each centroid. Euclidean Distance Formula. The Euclidean distance between two points 𝑝 and 𝑞 in n-dimensional space is given by:*

*Distance(p,q)=√ n∑​i=1(pi​−qi​)2​*

*Calculating Distances*

*For Image 1: Distance to C1: 𝐷 1 , 1 = √ ( 0.267 − 0.267 ) 2 + ( 0.356 − 0.356 ) 2 + ( 0.302 − 0.302 ) 2 = 0*

*Distance to C2: 𝐷 1 , 2 = √ ( 0.267 − 0.423 ) 2 + ( 0.356 − 0.683 ) 2 + ( 0.302 − 0.051 ) 2 ≈ 0.440*

*For Image 2: Distance to C1: 0.190 Distance to C2: 0.395*

*For Image 3: Distance to C1: 0.440 Distance to C2: 0*

*For Image 4: Distance to C1: 0.211 Distance to C2: 0.512*

*Summary of Distances*

*Image 1: 𝐷1,1=0 D1,1 =0 (Cluster 1), D1,2​ ≈0.440 (Cluster 2)*

*Image 2: D 2,1 ≈0.190 (Cluster 1), D 2,2 ≈0.395 (Cluster 2)*

*Image 3: D 3,1​ ≈0.440 (Cluster 1), D 3,2 =0 (Cluster 2)*

*Image 4: D 4,1 ≈0.211 (Cluster 1), D 4,2 ≈0.512 (Cluster 2)*

*Cluster Assignment*

*Image 1: Cluster 1 (nearest to C1)*

*Image 2: Cluster 1 (nearest to C1)*

*Image 3: Cluster 2 (nearest to C2)*

*Image 4: Cluster 1 (nearest to C1)*

***Step 3: Update Step***

*Now we compute the new centroids based on the assigned clusters:*

*Cluster 1 contains Images 1, 2, and 4. Compute the new centroid for*

*C1new​=((0.267+0.171+0.409)/3​, (0.356+0.520+0.210)/3​, (0.302+0.308+0.247​)/3)*

*C1new=(0.847/3,1.086/3,0.857/3)≈(0.282,0.362,0.286)*

*Cluster 2 contains only Image 3:*

*𝐶 2 new = 𝐶 2 = [ 0.423 , 0.683 , 0.051 ] C 2 new ​ =C 2=[0.423,0.683,0.051]*

***Step 4: Repeat Assignment***

*Repeat the assignment step with the new centroids. Continue this process until the centroids stabilize (i.e., no further assignments change).*

*Suppose after some steps... Centroid 1 (C1) remains unchanged:*

*𝐶1new=(0.282,0.362,0.286) (same assignment as before)*

*Centroid 2 (C2) remains unchanged:*

*𝐶2new=(0.423,0.683,0.051) (same assignment as before)*

*Conclusion: Centroids Stabilized. The centroids have stabilized since the assignments do not change from one iteration to the next. At this point, the algorithm can stop.*

*Final Centroids:*

*C1: [0.282, 0.362, 0.286]*

*C2: [0.423, 0.683, 0.051]*

***Final Clustering***

*After several iterations, K-Means will converge, and we will end up with two distinct clusters based on the input features. The clustering process allows us to group similar images together based on their encoded representations, enabling better analysis and decision-making based on image characteristics.*

***###############################################***