

Hair Types Classification

Developed by Team Visionaries



Project Overview



About Our Idea !

The core idea of this project is to build a computer vision model powered by deep learning to classify hair types with precision. Picture a tool that analyzes an image of your hair and instantly identifies whether it's straight, wavy, curly, or kinky. This information can be used to tailor hair care routines specifically for each texture. Additionally, it provides tailored hair routine tips, like the best shampoos, conditioners, or styling tricks suited to that specific hair type. By leveraging advanced image analysis, this project aims to make hair care more accessible and personalized, helping users understand and embrace their unique hair texture.



Dataset Description

Hair Type Dataset (Kaggle)

A high-quality image dataset for classifying four hair types: Straight, Wavy, Curly, Kinky plus Dreadlocks as a hairstyle. Useful for training ML models on hair texture recognition.

Straight: 488 | Wavy: 330 | Curly: 54 | Kinky: 217 | Dreadlocks: 443



About Our Idea !

The core idea of this project is to build a computer vision model powered by deep learning to classify hair types with precision. Picture this: a tool that analyzes an image of your hair and instantly identifies whether it's straight, wavy, curly, kinky, or styled in dreadlocks. But it doesn't stop there for each prediction, we provide tailored hair routine tips, like the best shampoos, conditioners, or styling tricks suited to that specific hair type. By leveraging advanced image analysis, this project aims to make hair care simpler and more informed, helping users understand and embrace their unique hair texture.



Dataset Description

Hair Type Dataset (Kaggle)

A high-quality image dataset for classifying four hair types: Straight, Wavy, Curly, Kinky plus Dreadlocks as a hairstyle. Useful for training ML models on hair texture recognition.

Straight: 488 | Wavy: 330 | Curly: 514 | Kinky: 217 | Dreadlocks: 443

Hair Types Classification

Developed by Team Visionaries



Dataset Pre-Processing

Our Approach!

- Exploratory Data Analysis(EDA)
- Data Preprocessing
- CNN Architecture and Training Pipeline

Exploratory Data Analysis(EDA)



Data Preprocessing

- Organize the dataset **into a DataFrame** and split it into **80% training** and **20% testing**.
- Apply **data augmentation** on images to address the issue of limited data.
- Handle the **class imbalance problem** effectively.

Data Preprocessing



Data Preprocessing

- Offline Augmentation (Preprocessing Step) - Using ImageDataGenerator.



Data Preprocessing

- Compute class weights based on class distribution using compute_class_weight to address the imbalance problem.

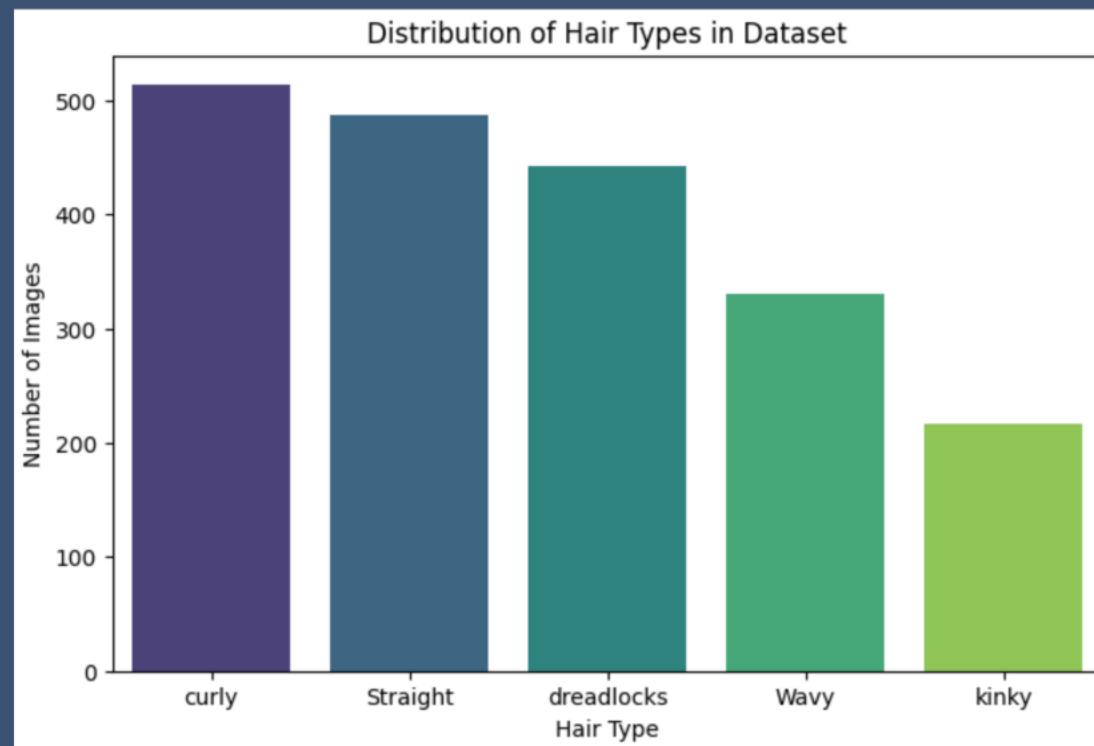


Our Approach!

- Exploratory Data Analysis(EDA)
- Data Preprocessing
- CNN Architecture and Training Pipeline

Exploratory Data Analysis(EDA)

- Imbalanced Dataset



Data Preprocessing

- Organize the dataset **into a DataFrame** and **split it** into **80% training** and **20% testing**.
- Apply **data augmentation** on images to address the issue of limited data.
- Handle the **class imbalance problem** effectively.

Data Preprocessing

- Show few of the dataframe after split

	filename	Class
187	curly/image16.jpg	curly
115	curly/curl-tips-styling-1561743013.jpg	curly
569	Straight/8-centreparted-layered-cut-for-long-h...	Straight
1176	kinky/image44.jpg	kinky
1858	Wavy/images (3).jpg	Wavy
1917	Wavy/images5.jpg	Wavy
1537	dreadlocks/image49.jpg	dreadlocks
307	curly/image31.jpg	curly
466	curly/images47.jpg	curly
1026	kinky/011b913aca7afbdd7feb6cdd4481b3bf4.jpg	kinky

Data Preprocessing

- Offline Augmentation (Preprocessing Stage) → **Using ImageDataGenerator.**

```
# Creating data generators for image preprocessing and augmentation.  
# The training data generator applies rescaling and various augmentations  
# (rotation, shifting, zooming, and flipping) to enhance model generalization.  
# The testing data generator only rescales images to normalize pixel values.  
train_datagen = ImageDataGenerator(  
    rescale=1./255,  
    rotation_range=15,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    zoom_range=0.2,  
    horizontal_flip=True  
)  
test_datagen = ImageDataGenerator(rescale=1./255)
```

Data Preprocessing

- Compute class weights based on class distribution using compute_class_weight to address the imbalance problem.

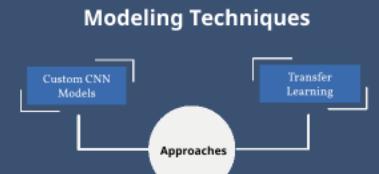
```
# Computing class weights to handle class imbalance in the dataset.  
# Extracts class labels from the training generator.  
# Retrieves class names and calculates weights using the "balanced" strategy.  
# The computed weights help the model give appropriate importance to underrepresented classes.  
  
# Extract class labels  
y_train = train_generator.classes # Get labels for training set  
class_labels = list(train_generator.class_indices.keys())  
  
# Compute class weights  
class_weights = compute_class_weight("balanced", classes=np.unique(y_train), y=y_train)  
class_weight_dict = dict(enumerate(class_weights))  
  
print("Class Indices:", train_generator.class_indices)  
print("Class Weights:", class_weight_dict)  
  
Class Indices: {'Straight': 0, 'Wavy': 1, 'curly': 2, 'dreadlocks': 3, 'kinky': 4}  
Class Weights: {0: 0.820671834625323, 1: 1.2076045627376426, 2: 0.7746341463414634, 3: 0.8971751412429378, 4: 1.825287356321839}
```

Hair Types Classification

Developed by Team Visionaries



Modeling



Custom CNN models

CNN Model 1	
Architecture Overview	
Number of Layers & Core Blocks	= Global Average Pooling + Dense
Activation Functions: ReLU (Core), Softmax (Output)	
Optimizer: Adam (LR = 0.0001)	
Loss Function: Sparse Categorical Crossentropy	
Results: Accuracy: 90%	Best: DenseNet121 (90% recall), Kdby (90% recall)
Best: DiceNet121 (90% recall), West_Mary (90% recall)	

Custom CNN models

CNN Model 2	
Architecture Overview	
Number of Layers & Core Layers	= Block + 2 Dense Layers
Activation Functions: ReLU (Core & Dense), Softmax (Output)	
Optimizer: Adam (LR = 0.0001)	
Loss Function: Sparse Categorical Crossentropy	
Results: Accuracy: 90.5%	Best: DenseNet121 (90.5% recall), straight10 (90.5% recall)
Best: DiceNet121 (90.5% recall), West_Mary (90.5% recall)	

Transfer Learning

Why VGG16?	
Strong feature extraction with pre-trained ImageNet weights, gives us for image classification.	
Layers Added:	Flatten, Dense (ReLU), Dropout (0.5), Dense (ReLU), Dropout (0.5), Output Layer (Softmax)
Optimizer:	Adam (LR = 0.0001)
Loss Function:	Sparse Categorical Crossentropy
Results:	Accuracy: 86% Best: DenseNet121 (90% recall), Kdby (90% recall)
	Hyper-CNN (90% recall)

VGG16

Architecture Overview	
Layer 1	Input
Layer 2	Conv2D (3x3, 64)
Layer 3	ReLU
Layer 4	MaxPool2D
Layer 5	Conv2D (3x3, 128)
Layer 6	ReLU
Layer 7	MaxPool2D
Layer 8	Conv2D (3x3, 256)
Layer 9	ReLU
Layer 10	MaxPool2D
Layer 11	Conv2D (3x3, 512)
Layer 12	ReLU
Layer 13	MaxPool2D
Layer 14	Conv2D (3x3, 512)
Layer 15	ReLU
Layer 16	MaxPool2D
Layer 17	Flatten
Layer 18	Dense (ReLU)
Layer 19	Dropout (0.5)
Layer 20	Dense (ReLU)
Layer 21	Dropout (0.5)
Layer 22	Output Layer (Softmax)

Best Model Result

DenseNet121



Transfer Learning

Why DenseNet?	
Pretrained on ImageNet, great for small datasets like hair type classification.	
Layers Added:	Global Average Pooling, Dense (ReLU), Dense (ReLU), Dropout (0.5) Output Layer (Softmax)
Optimizer:	Adam (LR = 0.0001)
Loss Function:	Sparse Categorical Crossentropy
Results:	Accuracy: 86% Best: DenseNet121 (90% recall), Curly (91% precision)

DenseNet121

Architecture Overview	
Layer 1	Input
Layer 2	Conv2D (3x3, 64)
Layer 3	ReLU
Layer 4	Batch Normalization
Layer 5	Group Normalization
Layer 6	Group Normalization
Layer 7	Group Normalization
Layer 8	Group Normalization
Layer 9	Group Normalization
Layer 10	Group Normalization
Layer 11	Group Normalization
Layer 12	Group Normalization
Layer 13	Group Normalization
Layer 14	Group Normalization
Layer 15	Group Normalization
Layer 16	Group Normalization
Layer 17	Group Normalization
Layer 18	Group Normalization
Layer 19	Group Normalization
Layer 20	Group Normalization
Layer 21	Group Normalization
Layer 22	Group Normalization
Layer 23	Group Normalization
Layer 24	Group Normalization
Layer 25	Group Normalization
Layer 26	Group Normalization
Layer 27	Group Normalization
Layer 28	Group Normalization
Layer 29	Group Normalization
Layer 30	Group Normalization
Layer 31	Group Normalization
Layer 32	Group Normalization
Layer 33	Group Normalization
Layer 34	Group Normalization
Layer 35	Group Normalization
Layer 36	Group Normalization
Layer 37	Group Normalization
Layer 38	Group Normalization
Layer 39	Group Normalization
Layer 40	Group Normalization
Layer 41	Group Normalization
Layer 42	Group Normalization
Layer 43	Group Normalization
Layer 44	Group Normalization
Layer 45	Group Normalization
Layer 46	Group Normalization
Layer 47	Group Normalization
Layer 48	Group Normalization
Layer 49	Group Normalization
Layer 50	Group Normalization
Layer 51	Group Normalization
Layer 52	Group Normalization
Layer 53	Group Normalization
Layer 54	Group Normalization
Layer 55	Group Normalization
Layer 56	Group Normalization
Layer 57	Group Normalization
Layer 58	Group Normalization
Layer 59	Group Normalization
Layer 60	Group Normalization
Layer 61	Group Normalization
Layer 62	Group Normalization
Layer 63	Group Normalization
Layer 64	Group Normalization
Layer 65	Group Normalization
Layer 66	Group Normalization
Layer 67	Group Normalization
Layer 68	Group Normalization
Layer 69	Group Normalization
Layer 70	Group Normalization
Layer 71	Group Normalization
Layer 72	Group Normalization
Layer 73	Group Normalization
Layer 74	Group Normalization
Layer 75	Group Normalization
Layer 76	Group Normalization
Layer 77	Group Normalization
Layer 78	Group Normalization
Layer 79	Group Normalization
Layer 80	Group Normalization
Layer 81	Group Normalization
Layer 82	Group Normalization
Layer 83	Group Normalization
Layer 84	Group Normalization
Layer 85	Group Normalization
Layer 86	Group Normalization
Layer 87	Group Normalization
Layer 88	Group Normalization
Layer 89	Group Normalization
Layer 90	Group Normalization
Layer 91	Group Normalization
Layer 92	Group Normalization
Layer 93	Group Normalization
Layer 94	Group Normalization
Layer 95	Group Normalization
Layer 96	Group Normalization
Layer 97	Group Normalization
Layer 98	Group Normalization
Layer 99	Group Normalization
Layer 100	Group Normalization
Layer 101	Group Normalization
Layer 102	Group Normalization
Layer 103	Group Normalization
Layer 104	Group Normalization
Layer 105	Group Normalization
Layer 106	Group Normalization
Layer 107	Group Normalization
Layer 108	Group Normalization
Layer 109	Group Normalization
Layer 110	Group Normalization
Layer 111	Group Normalization
Layer 112	Group Normalization
Layer 113	Group Normalization
Layer 114	Group Normalization
Layer 115	Group Normalization
Layer 116	Group Normalization
Layer 117	Group Normalization
Layer 118	Group Normalization
Layer 119	Group Normalization
Layer 120	Group Normalization
Layer 121	Group Normalization

Conclusion

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

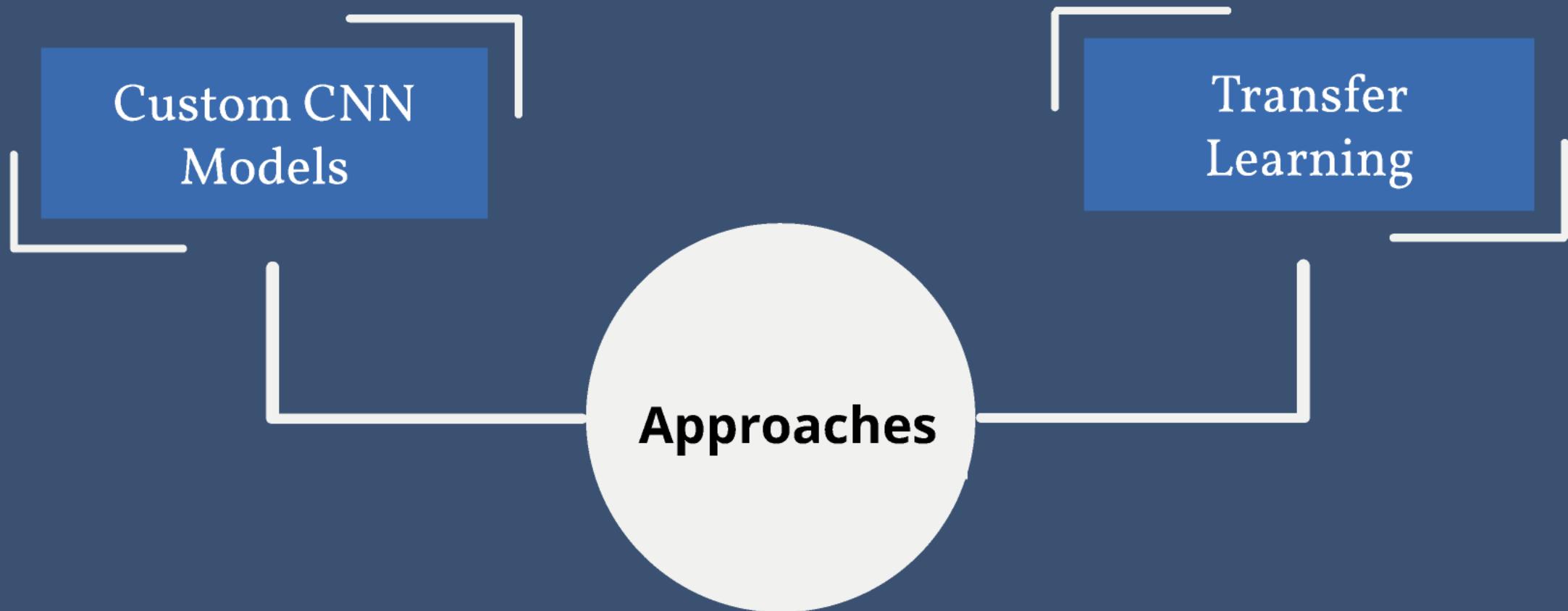
Curly and Kdby also performed well.

West_Mary and straight10 had the lowest accuracy.

Overall, DenseNet121 performed best with an accuracy of 97%.

Curly and Kdby also performed well.</p

Modeling Techniques



Custom CNN models

CNN Model 1

Number of Layers: 6 Conv Blocks + Global Average Pooling + Dense

Activation Functions: ReLU (Conv), Softmax (Output)

Optimizer: Adam (LR = 0.0003)

Loss Function: Sparse Categorical Crossentropy

Results: Accuracy: 70%
Best: Dreadlocks (86% recall), Kinky (91% recall)
Worst: Wavy (45% recall)

Architecture Overview

Layer (type)	Output Shape	Param #	Connected to
input_layer_22 (<code>InputLayer</code>)	(None, 224, 224, 3)	0	-
conv2d_77 (<code>Conv2D</code>)	(None, 224, 224, 32)	896	input_layer_22[0][0]
max_pooling2d_78 (<code>MaxPooling2D</code>)	(None, 112, 112, 32)	0	conv2d_77[0][0]
spatial_dropout2d_15 (<code>SpatialDropout2D</code>)	(None, 112, 112, 32)	0	max_pooling2d_78[0][0]
conv2d_79 (<code>Conv2D</code>)	(None, 112, 112, 64)	18,496	spatial_dropout2d_15[0][0]
conv2d_78 (<code>Conv2D</code>)	(None, 112, 112, 64)	2,112	spatial_dropout2d_15[0][0]
max_pooling2d_80 (<code>MaxPooling2D</code>)	(None, 56, 56, 64)	0	conv2d_79[0][0]
max_pooling2d_79 (<code>MaxPooling2D</code>)	(None, 56, 56, 64)	0	conv2d_78[0][0]
add_1 (<code>Add</code>)	(None, 56, 56, 64)	0	max_pooling2d_80[0][0] + max_pooling2d_79[0][0]
spatial_dropout2d_16 (<code>SpatialDropout2D</code>)	(None, 56, 56, 64)	0	add_1[0][0]
conv2d_80 (<code>Conv2D</code>)	(None, 56, 56, 128)	73,856	spatial_dropout2d_16[0][0]
max_pooling2d_81 (<code>MaxPooling2D</code>)	(None, 28, 28, 128)	0	conv2d_80[0][0]
spatial_dropout2d_17 (<code>SpatialDropout2D</code>)	(None, 28, 28, 128)	0	max_pooling2d_81[0][0]
conv2d_81 (<code>Conv2D</code>)	(None, 28, 28, 256)	295,168	spatial_dropout2d_17[0][0]
max_pooling2d_82 (<code>MaxPooling2D</code>)	(None, 14, 14, 256)	0	conv2d_81[0][0]
spatial_dropout2d_18 (<code>SpatialDropout2D</code>)	(None, 14, 14, 256)	0	max_pooling2d_82[0][0]
conv2d_82 (<code>Conv2D</code>)	(None, 14, 14, 512)	1,180,160	spatial_dropout2d_18[0][0]
max_pooling2d_83 (<code>MaxPooling2D</code>)	(None, 7, 7, 512)	0	conv2d_82[0][0]
spatial_dropout2d_19 (<code>SpatialDropout2D</code>)	(None, 7, 7, 512)	0	max_pooling2d_83[0][0]
global_average_pooling2d (<code>GlobalAveragePooling2D</code>)	(None, 512)	0	spatial_dropout2d_19[0][0]

Custom CNN models

CNN Model 2

Number of Layers: 4 Conv Blocks
+ Flatten + 2 Dense Layers

Activation Functions: ReLU
(Conv & Dense), Softmax (Output)

Optimizer: Adam (LR = 0.0005)

Loss Function: Sparse Categorical
Crossentropy

Results: Accuracy: 76.1%
Best: Dreadlocks (81% recall), Straight (77% recall)
Challenges: Wavy Hair (45% recall, misclassified often as Curly)

Architecture Overview

Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d_8 (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_9 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_9 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_10 (Conv2D)	(None, 52, 52, 128)	73,856
max_pooling2d_10 (MaxPooling2D)	(None, 26, 26, 128)	0
conv2d_11 (Conv2D)	(None, 24, 24, 256)	295,168
max_pooling2d_11 (MaxPooling2D)	(None, 12, 12, 256)	0
flatten_2 (Flatten)	(None, 36864)	0
dense_4 (Dense)	(None, 256)	9,437,440
dropout_2 (Dropout)	(None, 256)	0
dense_5 (Dense)	(None, 5)	1,285

Transfer Learning

VGG16

Why VGG16?

Strong feature extraction with pre-trained ImageNet weights, proven for image classification.

Layers Added: Flatten, Dense (ReLU) + Dropout (0.5) + Dense (ReLU) + Dropout (0.5) + Output Layer (Softmax)

Optimizer: Adam (LR = 0.0001 (initial) and 3e-6 (for fine-tuning)).

Loss Function: Sparse Categorical Crossentropy

Results: Accuracy: 86%

Best: Dreadlocks (93% recall), Kinky (91% recall)

Worst: Curly (81% recall)

Architecture Overview

Layer (type)	Output Shape	Param #
keras_tensor_145 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,000
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,000
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,188,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,000
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,000
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,000
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,000
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,000
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_4 (Flatten)	(None, 25408)	0
dense_9 (Dense)	(None, 512)	12,845,560
dropout_8 (Dropout)	(None, 512)	0
dense_10 (Dense)	(None, 256)	131,328
dropout_9 (Dropout)	(None, 256)	0
dense_11 (Dense)	(None, 5)	1,285

Transfer Learning

DenseNet121

Why DenseNet?

Pretrained on ImageNet, great for small datasets like hair type classification.

Layers Added: Global Average Pooling, Dense (ReLU) + Dense (ReLU) + Dropout (0.3) Output Layer (Softmax)

Optimizer: Adam (LR 0.0001)

Loss Function: Sparse Categorical Crossentropy

Results: 88%
Best: Dreadlocks (94% recall), Curly (91% precision)

Architecture Overview

Layer (type)	Output Shape	Param #
densenet121 (Functional)	(None, 7, 7, 1024)	7,037,504
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 1024)	0
dense_6 (Dense)	(None, 224)	229,600
dense_7 (Dense)	(None, 128)	28,800
dropout_3 (Dropout)	(None, 128)	0
dense_8 (Dense)	(None, 5)	645

Best Model Result

DenceNet121

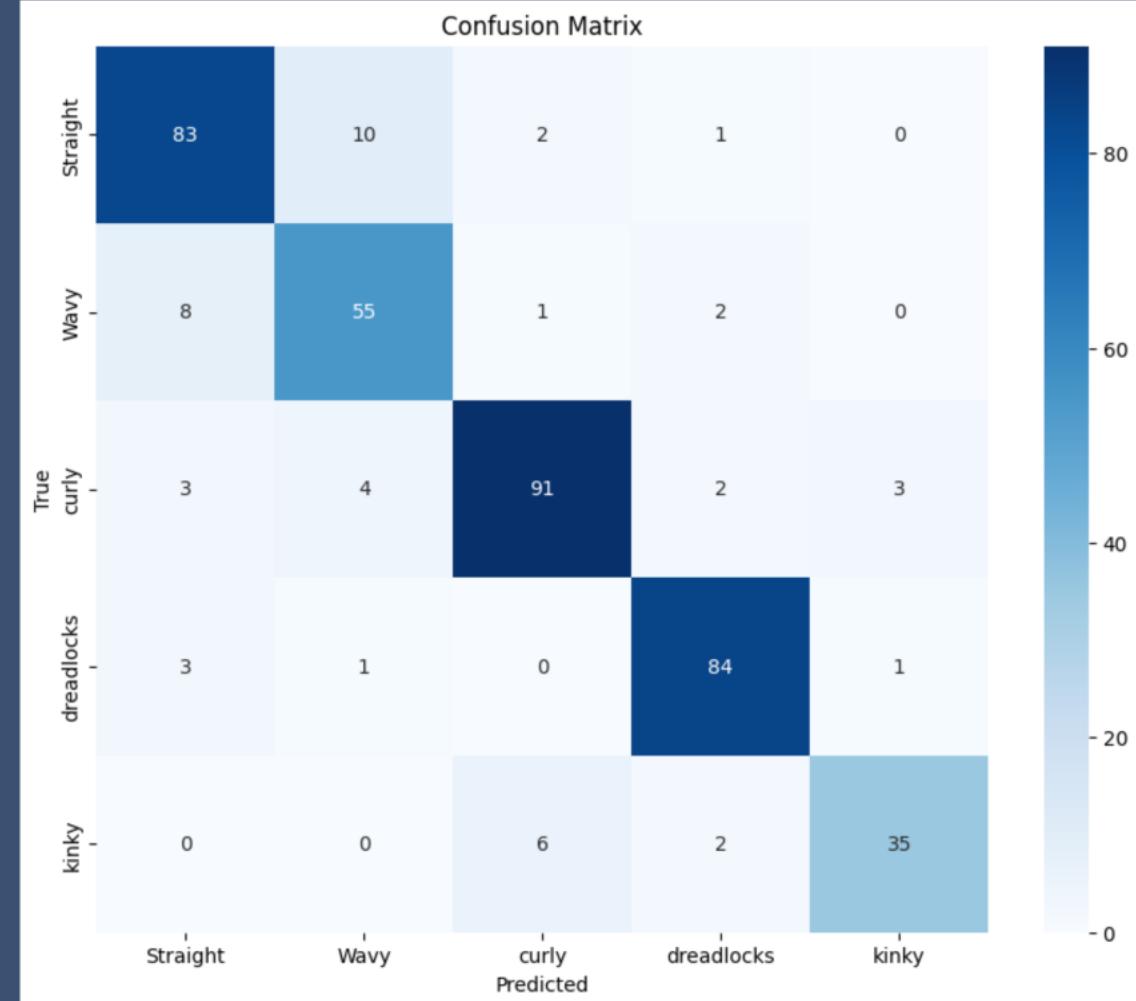
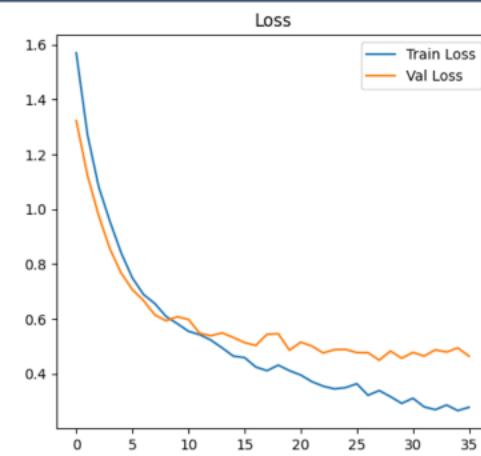
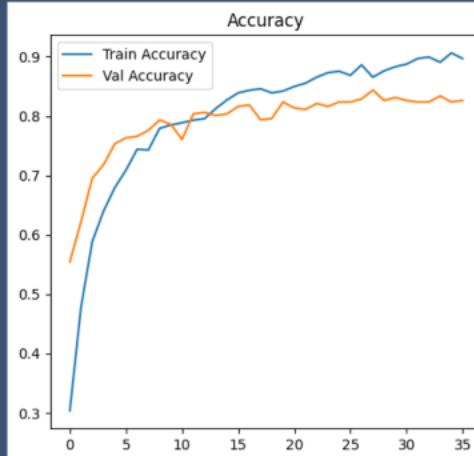
Test Accuracy:

88%

Test Loss:
33%

Accuracy:
97%

Train Loss:
8%



Hair Types Classification

Developed by Team Visionaries





Present By Mayar

Conclusion and Streamlit development

Streamlit

This Streamlit app classifies hair types from uploaded images using a TensorFlow model (advanced_densenet_model.h5). Users can upload images in [JPG, JPEG, or PNG formats, and the app preprocesses these images before making predictions. The predicted hair type and confidence level are displayed, along with fun facts and care tips related to the hair type. The layout features a sidebar for uploads and a main area for results, enhanced with custom CSS for better visuals.



Conclusion

In conclusion, our hair type classification project effectively combines AI learning and user-friendly design to empower users with insights about their hair. By using a Streamlit application, we provide real-time predictions.

This innovative approach not only enhances users' understanding of their unique hair types but also promotes better hair care practices, ultimately contributing to healthier hair.



Visionaries Group
Thank you!

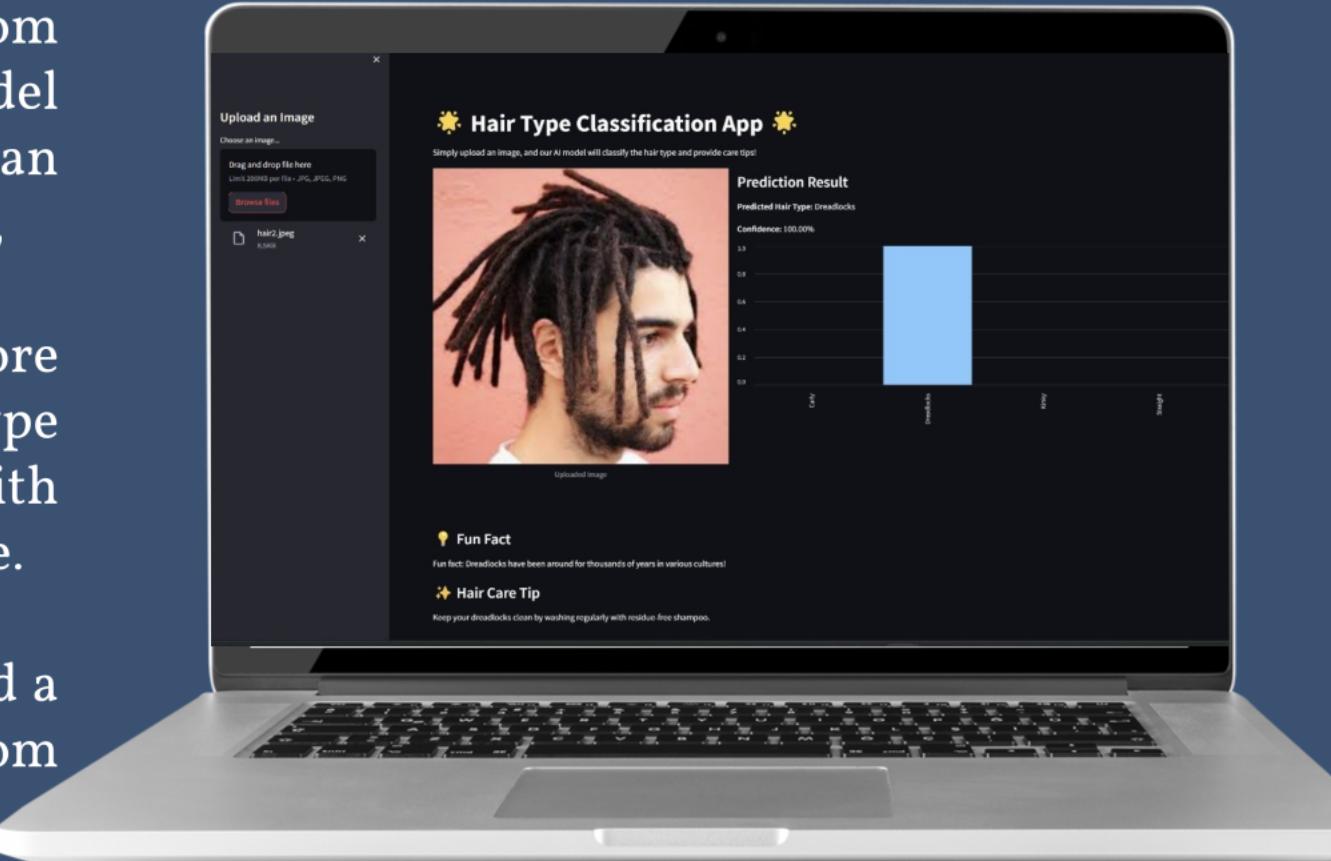
Group Names:
• Mayar Fouz Althebati
• Shatha Saleh Alqubaisi
• Raghad Khaled Almutairi
• Waleef Yousef Alqurashi

Streamlit

This Streamlit app classifies hair types from uploaded images using a TensorFlow model (`advanced_densenet_model.h5`). Users can upload images in JPG, JPEG, or PNG formats,

and the app preprocesses these images before making predictions. The predicted hair type and confidence level are displayed, along with fun facts and care tips related to the hair type.

The layout features a sidebar for uploads and a main area for results, enhanced with custom CSS for better visuals.



Conclusion

In conclusion, our hair type classification project effectively combines deep learning and user-friendly design to empower users with insights about their hair.

By using a Streamlit application, we provide real-time predictions .

This innovative approach not only enhances users' understanding of their unique hair types but also promotes better hair care practices, ultimately contributing to healthier hair.





Visionaries Group Thank you!

Group Names:

- Mayar Fawaz althebati
- Shatha Saleh Alqubaisi
- Raghad Khaled Almutairi
- Wareef Yousef Alqurashi

Hair Types Classification

Developed by Team Visionaries

