REPORT

IMAGE CLASSIFICATION USING CNN, LOGISTIC REGRESSION AND DECISION TREE CLASSIFIER

GROUP B

C0886190 Ahmed Abdulrahim

C0887143 Bimsara Siman Meru Pathiranage

C0864054 Efemena Theophilus Edoja

C0885235 Simranjeet Kaur

C0885177 Himanshu

Abstract

This project focuses on developing an image classification system using advanced techniques in computer vision. The objective is to accurately categorize images into predefined classes, enabling applications such as object recognition and scene understanding. Leveraging Convolutional Neural Networks (CNNs), the project aims to extract hierarchical features from images, capturing both low-level patterns and high-level semantic information. Additionally, logistic regression provides an alternative approach for image classification, offering insights into simpler yet effective methodologies.

Implemented using Python, the project encompasses various preprocessing steps and data augmentation techniques to enhance model performance. Through the utilization of CNNs and logistic regression models, the system aims to achieve high accuracy and robustness in image classification tasks. The project's outcomes contribute to advancing the field of computer vision, providing insights into the practical implications and considerations of employing deep learning techniques for image analysis.

Table of Contents

1. Introduction

Image classification is a fundamental task in computer vision that involves categorizing images into predefined classes or categories. It has widespread applications across various domains, including object recognition, medical imaging, autonomous vehicles, and more. The ability to accurately classify images is essential for automated systems to interpret and understand visual data, enabling tasks such as image retrieval, content filtering, and decision-making.

In recent years, the advancement of deep learning techniques, particularly Convolutional Neural Networks (CNNs), has revolutionized image classification, achieving unprecedented levels of accuracy and performance. CNNs are well-suited for extracting hierarchical features from images, capturing both low-level patterns like edges and textures and high-level semantic information about objects and scenes. Additionally, logistic regression, a classic machine learning algorithm, offers a simpler yet effective approach to image classification, providing insights into alternative methodologies.

Through this report, we aim to provide a comprehensive overview of our project on image classification. We will delve into the methodology employed, including the implementation of CNNs and logistic regression models, as well as the preprocessing steps and data augmentation techniques utilized. Furthermore, we will discuss the experimental setup, including dataset selection, model training, and evaluation metrics.

By detailing our findings and analysis, we seek to elucidate the strengths and limitations of CNNs and logistic regression for image classification tasks. Additionally, we aim to provide insights into best practices and considerations for selecting appropriate models and techniques based on specific application requirements.

Ultimately, this report serves as a guide for understanding the intricacies of image classification and the practical implications of employing CNNs and logistic regression in real-world scenarios. Through our project, we aim to contribute to the advancement of computer vision and machine learning, fostering innovation and progress in this rapidly evolving field.

2. Data Collection and Preprocessing

The project utilizes a dataset of natural scene images obtained from Kaggle. The data preprocessing steps encompass transformation, and feature engineering. Transformation involves image resizing, normalization, and augmentation techniques to enhance the dataset's quality and variability. Feature engineering entails extracting relevant features from the images, such as color histograms, texture descriptors, and edge maps, to capture essential characteristics for classification.

2.1 Dataset Overview

2.1.1 Data Understanding

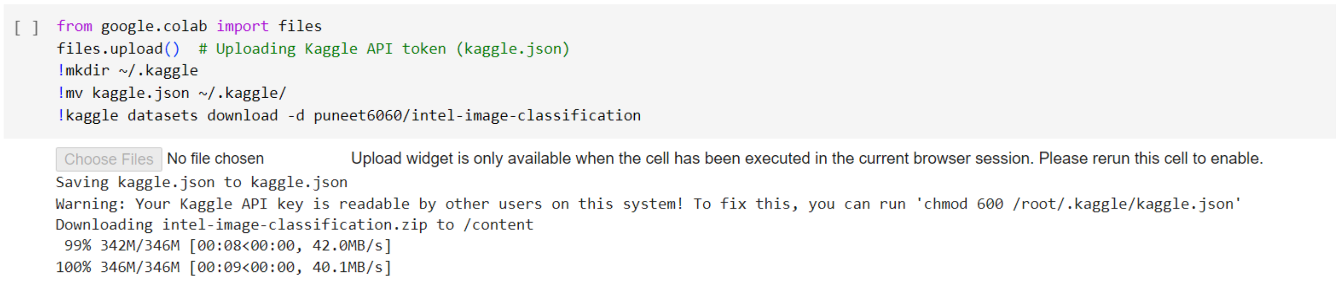
The natural scene images dataset serves as the cornerstone of our project, offering a rich source of images capturing diverse landscapes from around the world. Spanning from various locations and scenes, these images provide a comprehensive snapshot of natural environments over the dataset's timeframe. This dataset offers valuable insights into the diversity and characteristics of natural scenes, reflecting the dynamic nature of landscapes and environments.

Dataset URL : Intel Image Classification

Our primary objective in this phase is to conduct a comprehensive exploration of the natural scene images dataset, gaining insights into its structure and characteristics. By analyzing key attributes such as image dimensions, class distribution, and data quality, we aim to understand the dataset's composition and identify factors crucial for effective image classification. This exploratory phase lays the groundwork for subsequent analyses, providing valuable insights that inform model development and optimization strategies.

2.2 Dataset Information

Uploading Kaggle API token directly into google colab:



```
[ ] from google.colab import files
    files.upload()  # Uploading Kaggle API token (kaggle.json)
    !mkdir ~/.kaggle
    !mv kaggle.json ~/.kaggle/
    !kaggle datasets download -d puneet6060/intel-image-classification
```

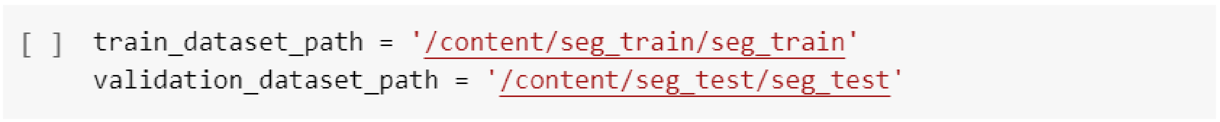
```
Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving kaggle.json to kaggle.json
Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmod 600 /root/.kaggle/kaggle.json'
Downloading intel-image-classification.zip to /content
 99% 342M/346M [00:08<00:00, 42.0MB/s]
100% 346M/346M [00:09<00:00, 40.1MB/s]
```

2.3 Data Preprocessing

Once the dataset's structure is comprehended, the focus shifts to preparing the data for thorough analysis. By executing these steps, we ensure that the dataset is streamlined, free from redundant information, and possesses consistent formatting, laying the groundwork for more sophisticated processing and analysis.

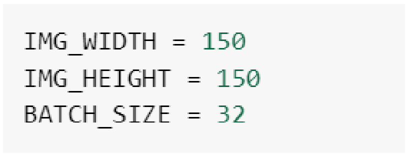
2.3.1 Dataset Augmentation for CNN Model

Getting the Image Dataset Paths:



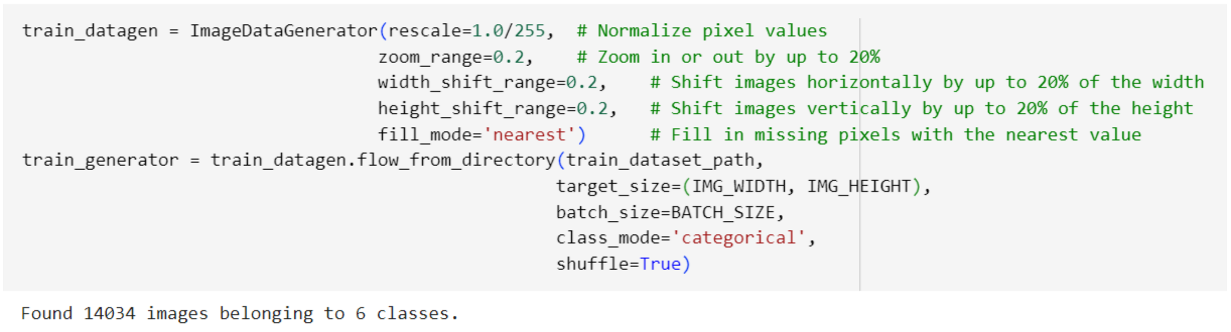
```
[ ] train_dataset_path = '/content/seg_train/seg_train'
    validation_dataset_path = '/content/seg_test/seg_test'
```

Loading Image Datasets and apply augmentations:



```
IMG_WIDTH = 150
IMG_HEIGHT = 150
BATCH_SIZE = 32
```

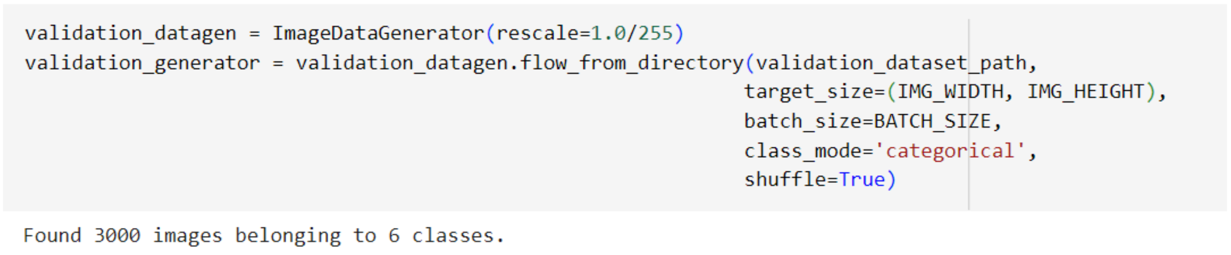
Loading training dataset and applying augmentations:



```
train_datagen = ImageDataGenerator(rescale=1.0/255,  # Normalize pixel values
                                   zoom_range=0.2,    # Zoom in or out by up to 20%
                                   width_shift_range=0.2,    # Shift images horizontally by up to 20% of the width
                                   height_shift_range=0.2,   # Shift images vertically by up to 20% of the height
                                   fill_mode='nearest')      # Fill in missing pixels with the nearest value
train_generator = train_datagen.flow_from_directory(train_dataset_path,
                                                    target_size=(IMG_WIDTH, IMG_HEIGHT),
                                                    batch_size=BATCH_SIZE,
                                                    class_mode='categorical',
                                                    shuffle=True)
```

```
Found 14034 images belonging to 6 classes.
```

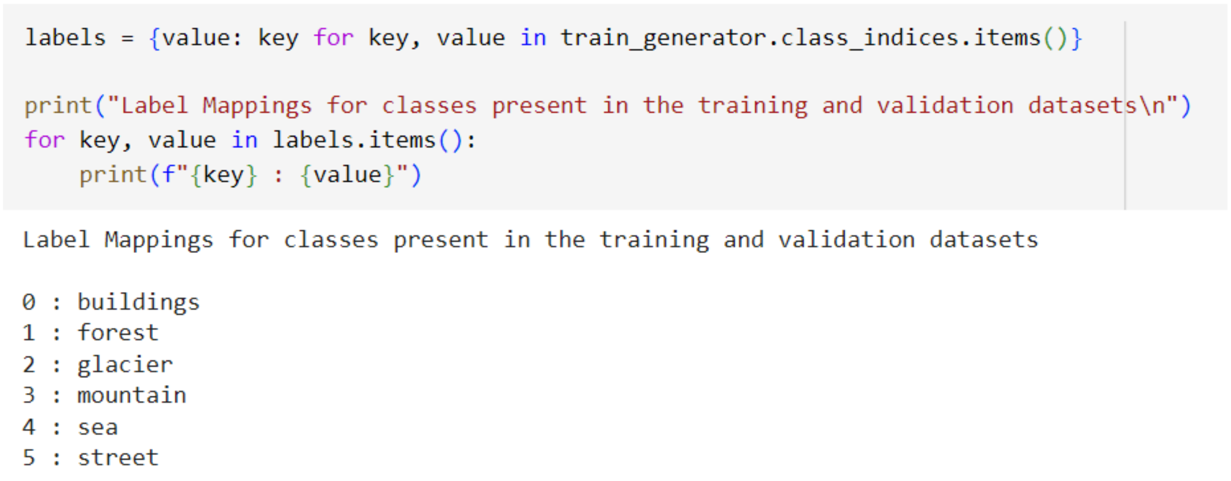
Loading the validation dataset and applying augmentations:



```
validation_datagen = ImageDataGenerator(rescale=1.0/255)
validation_generator = validation_datagen.flow_from_directory(validation_dataset_path,
                                                              target_size=(IMG_WIDTH, IMG_HEIGHT),
                                                              batch_size=BATCH_SIZE,
                                                              class_mode='categorical',
                                                              shuffle=True)
```

```
Found 3000 images belonging to 6 classes.
```

Get the label mappings:



```
labels = {value: key for key, value in train_generator.class_indices.items()}

print("Label Mappings for classes present in the training and validation datasets\n")
for key, value in labels.items():
    print(f"{key} : {value}")
```

```
Label Mappings for classes present in the training and validation datasets

0 : buildings
1 : forest
2 : glacier
3 : mountain
4 : sea
5 : street
```

2.3.2 Dataset Feature Extraction for Logistic Regression and Decision Tree Models

In this section, features are extracted from the images to prepare the data for logistic regression and decision tree models. Various image processing techniques are applied to extract meaningful features that capture texture, shape, and color information from the images.

Loading features using pickle files:

```python
# Load features and labels using Pickle
with open('/content/clf_train_features.pkl', 'rb') as f:
    features = pickle.load(f)

with open('/content/clf_train_labels.pkl', 'rb') as f:
    labels = pickle.load(f)
```

We extracted features from images using various techniques such as Local Binary Pattern (LBP) for texture, Sobel Edge Detection for edges, Histogram of Oriented Gradients (HOG) for shape and texture, and color histogram in the RGB space:

```python
def extract_features(image):
    # Convert image to grayscale
    gray_image = rgb2gray(image)

    # Apply Gaussian blur to smooth the image
    smoothed_image = gaussian(gray_image, sigma=1)

    # Local Binary Pattern for texture
    lbp = local_binary_pattern(smoothed_image, P=16, R=2, method='uniform')
    lbp_hist, _ = np.histogram(lbp, density=True, bins=np.arange(0, 18), range=(0, 17))

    # Sobel Edge Detection to capture edges
    edge_sobel = sobel(smoothed_image)
    edge_hist, _ = np.histogram(edge_sobel, density=True, bins=10)

    # Histogram of Oriented Gradients (HOG) for shape and texture
    # Note the use of `channel_axis=-1` to specify that the input is a color image
    hog_features, _ = hog(image, orientations=8, pixels_per_cell=(16, 16),
                          cells_per_block=(1, 1), visualize=True, multichannel=True, channel_axis=-1)

    # Color histogram in the RGB space
    color_hist = np.concatenate([np.histogram(image[:, :, i], bins=32, range=(0, 1), density=True)[0]
                                 for i in range(3)])  # Ensure the range is over all three channels if RGB

    # Combine features
    combined_features = np.concatenate((lbp_hist, edge_hist, hog_features, color_hist))
    return combined_features
```

These extracted features are then stored for further use in training logistic regression and decision tree models:

```
# Initialize storage for features and labels
features = []
labels = []
current_batch = 0
```

After extracting features from the images, the data is prepared for training logistic regression and decision tree models. Here's a summary of the data preparation steps:

- Conversion to Numpy Arrays: The extracted features and labels, stored as lists, are converted into Numpy arrays using np.vstack().

  ```
  # Convert list to numpy arrays
  features = np.vstack(features)
  labels = np.vstack(labels)
  ```

- Conversion of One-Hot Encoded Labels: The labels, which were originally in one-hot encoded format, are converted into integer labels using np.argmax(labels, axis=1). Each integer label now represents a class from 0 to 5.

  ```
  # Convert one-hot encoded labels to integer labels
  integer_labels = np.argmax(labels, axis=1)

  # Now integer_labels contains integer values from 0 to 5, each representing a class
  print(integer_labels.shape)
  ```

  ```
  (14034,)
  ```

2.4 Exploratory Data Analysis (EDA)

2.4.1 Analysis of train and test data

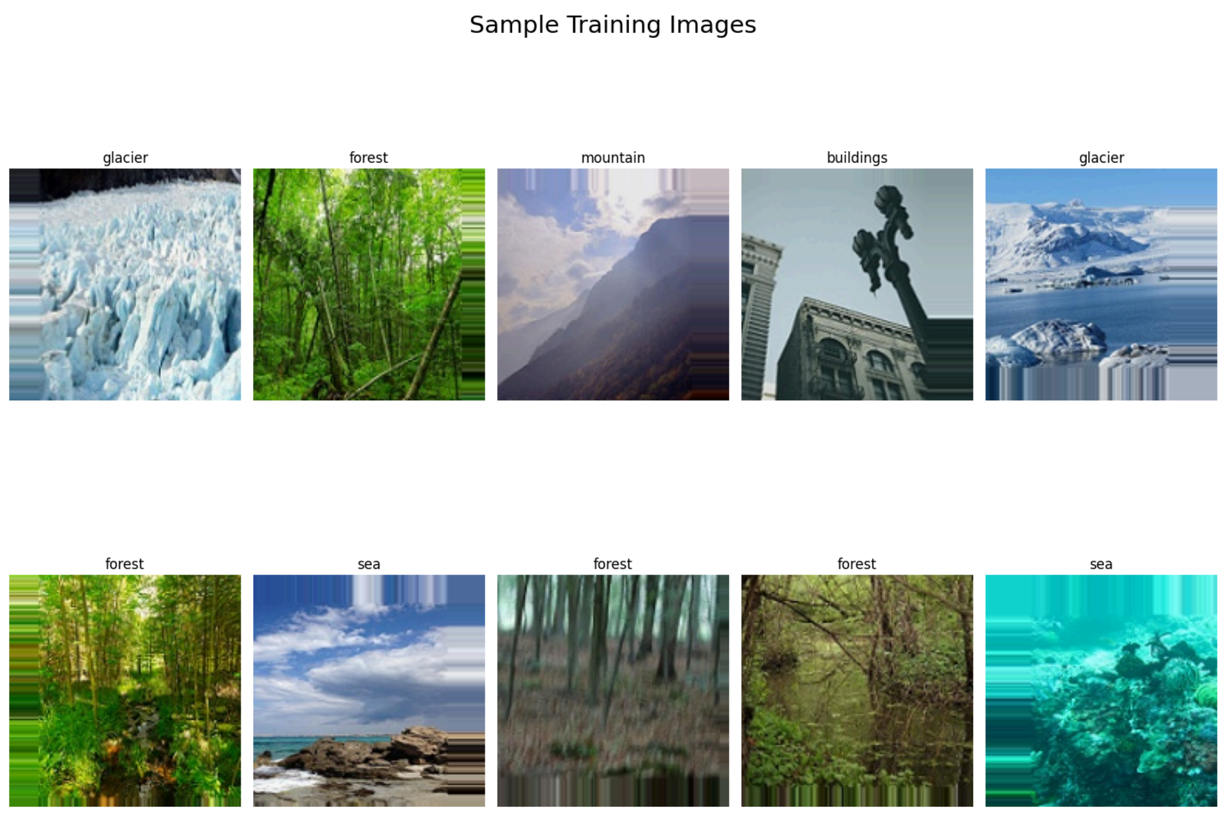
Plotting distribution bar graphs of images to see how images of each class are distributed in train and test dataset:

```
# Showing classes of images and distribution of images in train and test datasets
cat = os.listdir(train_dataset_path)
num_train = {}
num_test = {}

for c in cat:
    num_train[c] = len(os.listdir(train_dataset_path + "/" + c))
    num_test[c] = len(os.listdir(validation_dataset_path + "/" + c))


fig = plt.figure(figsize=(13, 4))
t = ('Train', 'Test')
for i, d in enumerate((num_train, num_test), start=1):
    plt.subplot(1, 2, i)
    plt.bar(tuple(d.keys()), tuple(d.values()), color='royalblue')
    plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.7)
    plt.title(t[i-1])
plt.show()
```

2.4.2 Analysis of Sample training Data

Plotting some training images to ensure the quality for further analysis:

```
fig, ax = plt.subplots(nrows=2, ncols=5, figsize=(15, 12))
idx = 0

for i in range(2):
    for j in range(5):
        label = labels[np.argmax(train_generator[0][1][idx])]
        ax[i, j].set_title(f"{label}")
        ax[i, j].imshow(train_generator[0][0][idx][:, :, :])
        ax[i, j].axis("off")
        idx += 1

plt.tight_layout()
plt.suptitle("Sample Training Images", fontsize=21)
plt.show()
```



3. Methodology

In this project, we employ Convolutional Neural Networks (CNNs) for image classification and logistic regression for comparative analysis. CNNs are chosen due to their effectiveness in extracting hierarchical features from images, capturing both low-level patterns and high-level semantic information. Logistic regression serves as an alternative method for image classification, offering insights into simpler yet effective methodologies.

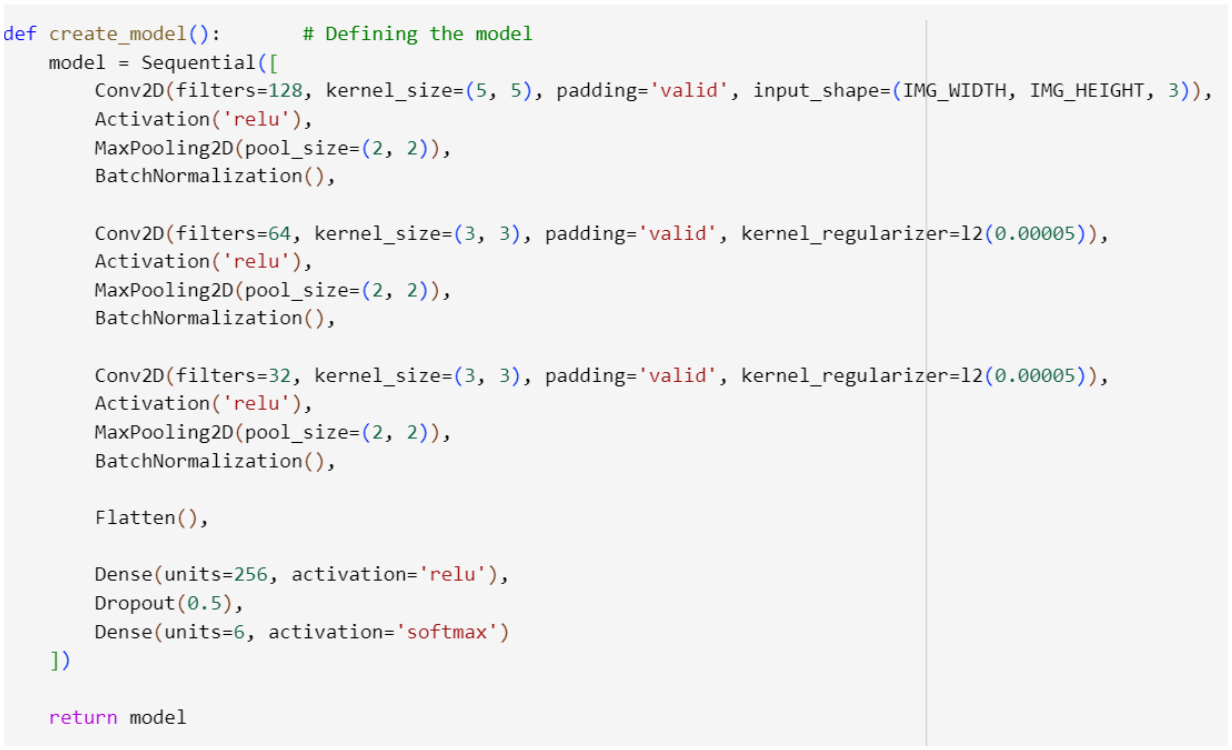
The models are implemented and trained using Python's deep learning frameworks, with preprocessing steps applied to enhance model performance. Data augmentation techniques such as rotation, scaling, and flipping are utilized to increase the diversity of the training dataset and improve model generalization.

To assess model performance, we employ cross-validation techniques, ensuring robustness and generalizability. Evaluation metrics including accuracy, precision, recall, and F1-score are utilized to measure the effectiveness of both CNNs and logistic regression in classifying images accurately across predefined categories.

3.1. Model Selection and Hyperparameter Tuning

3.1.1 Convolutional Neural Network (CNN)

For the CNN model, the chosen architecture includes convolutional layers, activation functions, max-pooling layers, batch normalization, and fully connected layers. This architecture is well-suited for capturing hierarchical features in images, making it suitable for image classification tasks.



```python
def create_model():       # Defining the model
    model = Sequential([
        Conv2D(filters=128, kernel_size=(5, 5), padding='valid', input_shape=(IMG_WIDTH, IMG_HEIGHT, 3)),
        Activation('relu'),
        MaxPooling2D(pool_size=(2, 2)),
        BatchNormalization(),

        Conv2D(filters=64, kernel_size=(3, 3), padding='valid', kernel_regularizer=l2(0.00005)),
        Activation('relu'),
        MaxPooling2D(pool_size=(2, 2)),
        BatchNormalization(),

        Conv2D(filters=32, kernel_size=(3, 3), padding='valid', kernel_regularizer=l2(0.00005)),
        Activation('relu'),
        MaxPooling2D(pool_size=(2, 2)),
        BatchNormalization(),

        Flatten(),

        Dense(units=256, activation='relu'),
        Dropout(0.5),
        Dense(units=6, activation='softmax')
    ])

    return model
```

Hyperparameter Tuning:

- Convolutional Layers: The number of filters and kernel sizes in the convolutional layers were selected empirically to balance model complexity and performance.
- Dropout: A dropout rate of 0.5 was applied to reduce overfitting by randomly dropping out units during training.
- Regularization: L2 regularization with a small coefficient of 0.00005 was applied to the convolutional layers to prevent overfitting.
- Optimizer: The Adam optimizer with a learning rate of 0.001 was chosen for its adaptive learning rate properties and efficiency in training deep neural networks.

```
optimizer = Adam(learning_rate=0.001)
```

- Learning Rate Scheduler: ReduceLROnPlateau callback was employed to reduce the learning rate when the validation loss plateaus, helping the model converge faster and achieve better performance.

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=np.sqrt(0.1), patience=5)
```

Model Compilation:

The model was compiled with the Adam optimizer, categorical cross-entropy loss function, and accuracy metric.

```
cnn_model.compile(optimizer=optimizer, loss=CategoricalCrossentropy(), metrics=['accuracy'])
```

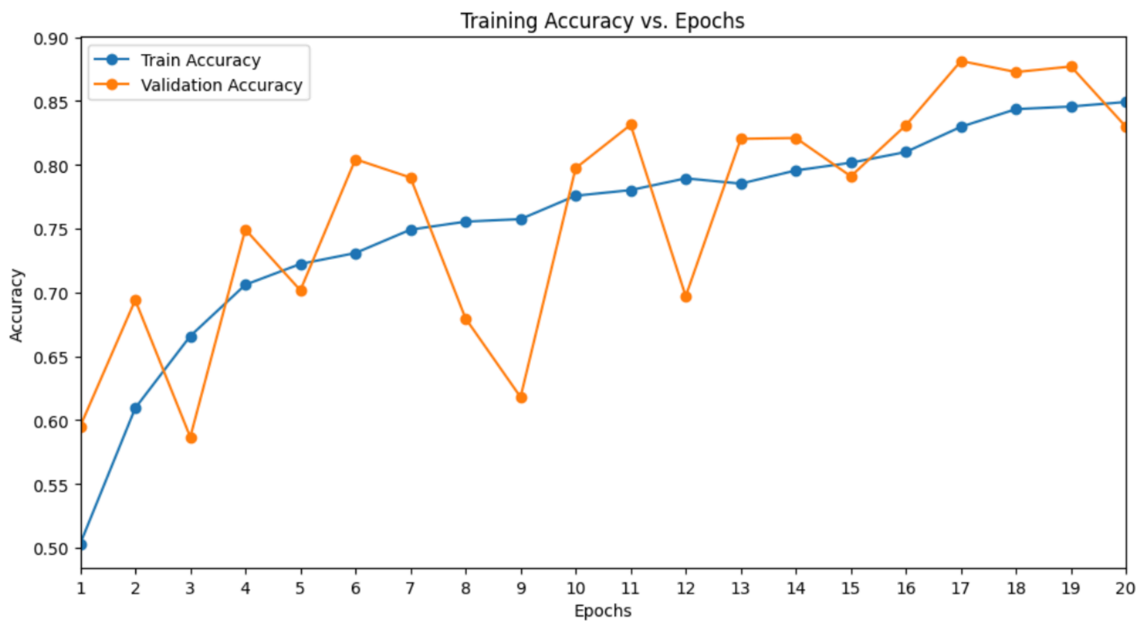
Model Training:

The CNN model was trained using the training data generated by an ImageDataGenerator. The training process involved optimizing the model parameters to minimize the categorical cross-entropy loss function. We trained the model for 20 epochs while monitoring its performance on the validation set to avoid overfitting.

```
history = cnn_model.fit(train_generator, epochs=20, validation_data=validation_generator,
                        verbose=2,
                        callbacks=[reduce_lr])
```

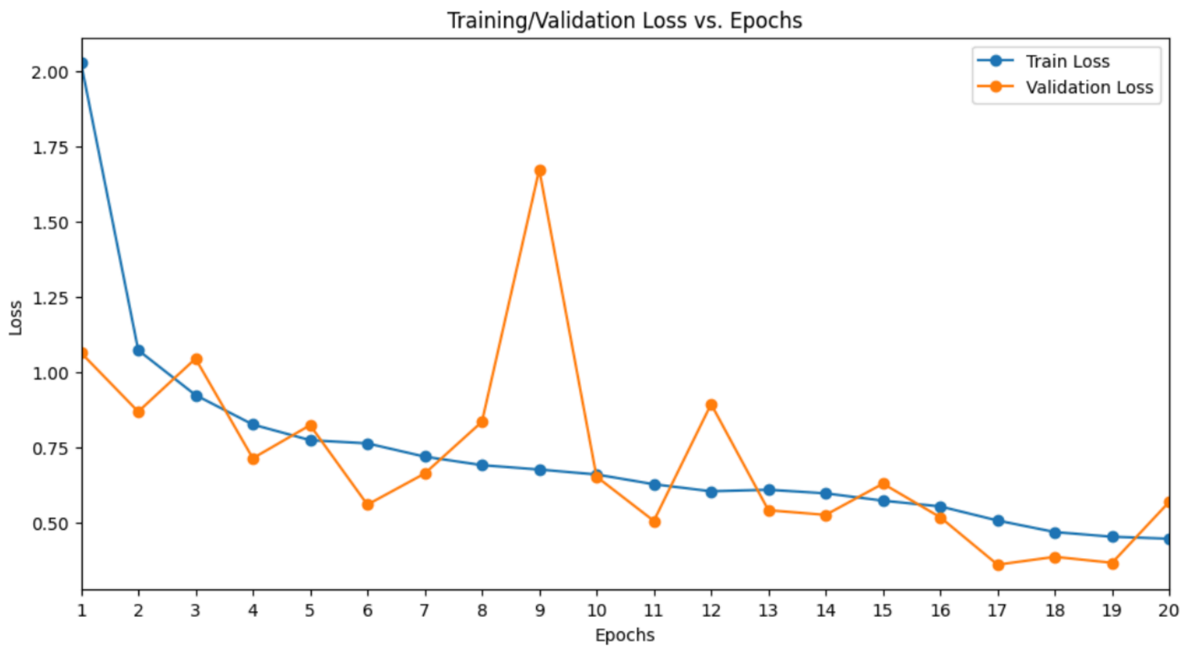
Training and Validation Accuracy vs. Epochs:

This subplot illustrates how the accuracy of the model changes over epochs during both training and validation. It helps monitor the model's ability to generalize unseen data and detect overfitting or underfitting.



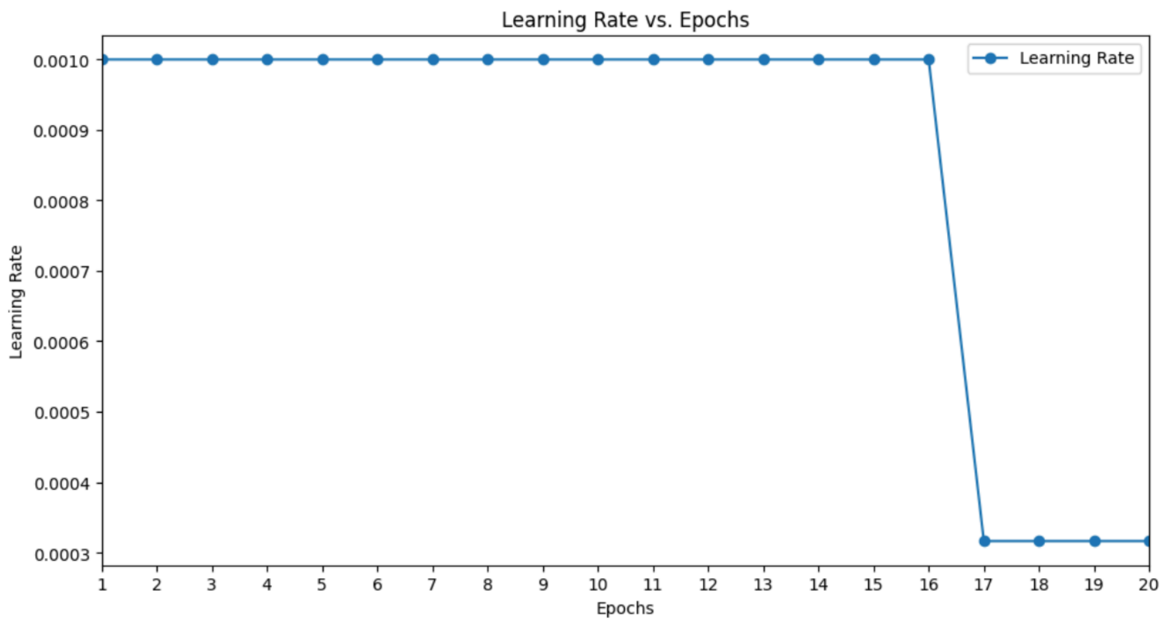
Training and Validation Loss vs. Epochs:

This subplot shows the training and validation loss over epochs. It indicates how well the model is fitting the training data and generalizing to unseen data. A decreasing trend in both training and validation loss suggests that the model is learning effectively.



Learning Rate vs. Epochs:

This subplot depicts the learning rate's variation over epochs. The learning rate influences how quickly or slowly the model learns and converges to an optimal solution. Monitoring the learning rate can help adjust the training process for better convergence and performance.

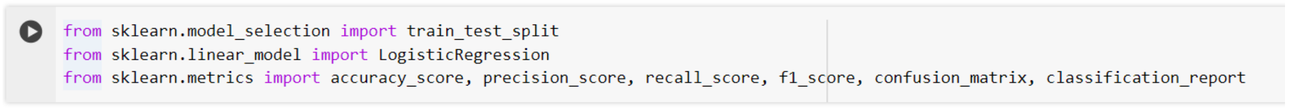


3.1.2 Logistic Regression Model

Logistic regression is suitable for binary and multiclass classification tasks, making it a viable choice for our image classification problem.

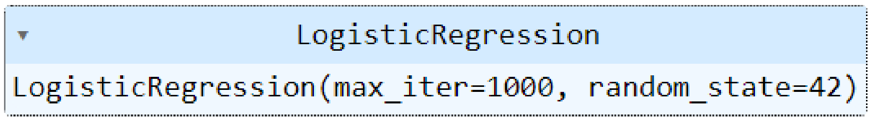
Model Training

- Data Splitting: The dataset is split into training and testing sets using a 80-20 ratio.



```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report
```

- Model Initialization: Logistic Regression model is initialized with a maximum iteration of 1000 and a random state of 42.



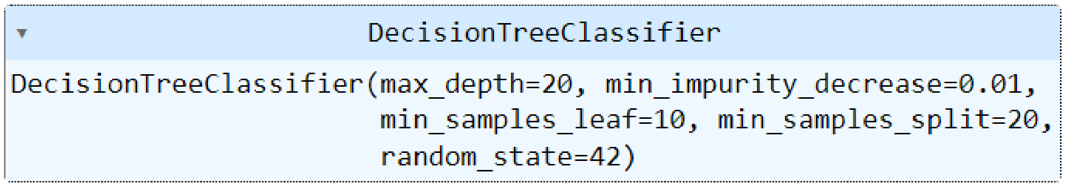
```
LogisticRegression
LogisticRegression(max_iter=1000, random_state=42)
```

- Model Fitting: The logistic regression model is fitted on the training data using the fit() function.

3.1.3 Decision Tree Classifier

Model Training

- Initialization: Decision Tree Classifier is initialized with specific parameters including criterion, splitter, max_depth, min_samples_split, min_samples_leaf, max_features, max_leaf_nodes, min_impurity_decrease, and random_state.
- Model Fitting: The classifier is fitted on the training data using the fit() function.



```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=20, min_impurity_decrease=0.01,
                       min_samples_leaf=10, min_samples_split=20,
                       random_state=42)
```

Model Tuning

PCA (Principal Component Analysis)

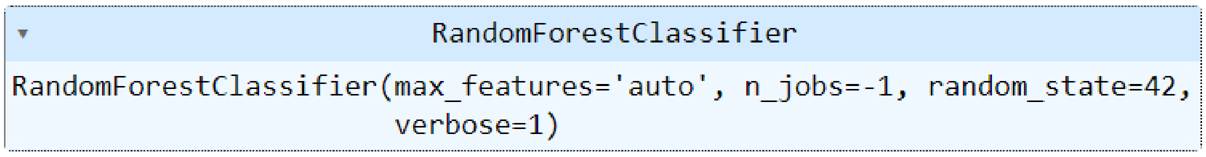
- Purpose: Principal Component Analysis is applied to reduce the dimensionality of the feature space while preserving most of the variance in the data. This helps in improving the model's performance and reducing overfitting.
- Implementation: PCA is performed using the PCA class from the sklearn.decomposition module. It is configured to retain 95% of the variance in the data.
    - n_components=0.95: Specifies that PCA should retain principal components that explain at least 95% of the variance.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.95)
features_reduced = pca.fit_transform(features)
```

3.1.4 Random Forest Classifier

Model Training and Evaluation

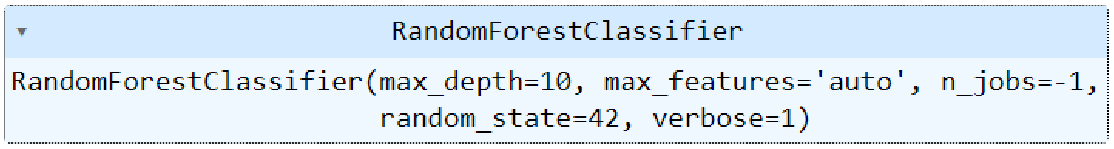
- Initialization: Random Forest Classifier is initialized with specific parameters including n_estimators, criterion, max_depth, min_samples_split, min_samples_leaf, max_features, random_state, n_jobs, and verbose.
- Model Fitting: The classifier is fitted on the training data using the fit() function.



```
RandomForestClassifier
RandomForestClassifier(max_features='auto', n_jobs=-1, random_state=42,
                       verbose=1)
```

Second Iteration of Random Forest Model

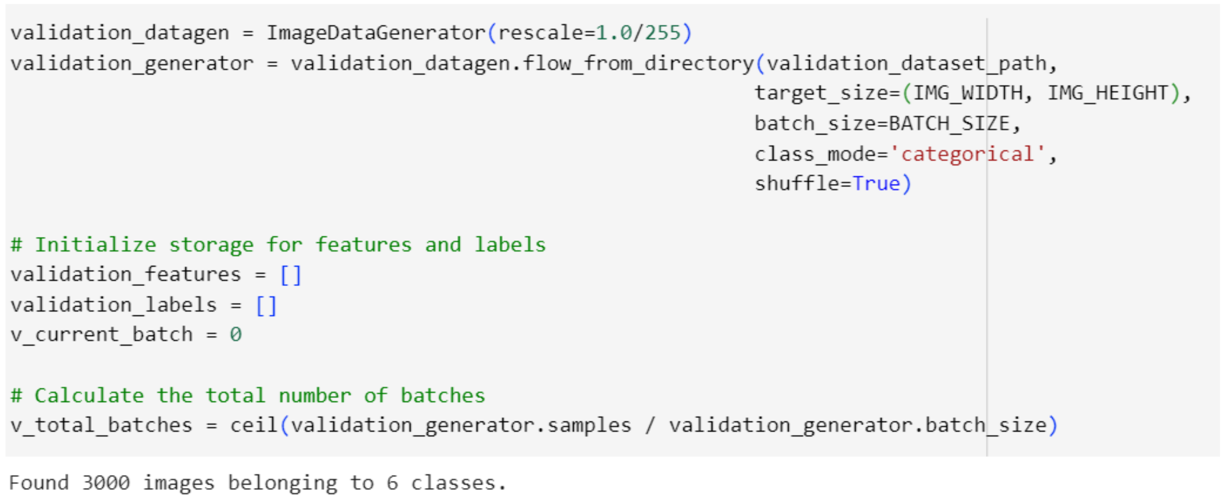
- A second Random Forest model is created with a reduced max_depth parameter to mitigate overfitting.
- Model training and evaluation are repeated for this adjusted model.



```
RandomForestClassifier
RandomForestClassifier(max_depth=10, max_features='auto', n_jobs=-1,
                       random_state=42, verbose=1)
```

Additional Validation

- Validation data is prepared using ImageDataGenerator to ensure the models' performance on unseen data.



```
validation_datagen = ImageDataGenerator(rescale=1.0/255)
validation_generator = validation_datagen.flow_from_directory(validation_dataset_path,
                                                             target_size=(IMG_WIDTH, IMG_HEIGHT),
                                                             batch_size=BATCH_SIZE,
                                                             class_mode='categorical',
                                                             shuffle=True)

# Initialize storage for features and labels
validation_features = []
validation_labels = []
v_current_batch = 0

# Calculate the total number of batches
v_total_batches = ceil(validation_generator.samples / validation_generator.batch_size)
```

```
Found 3000 images belonging to 6 classes.
```

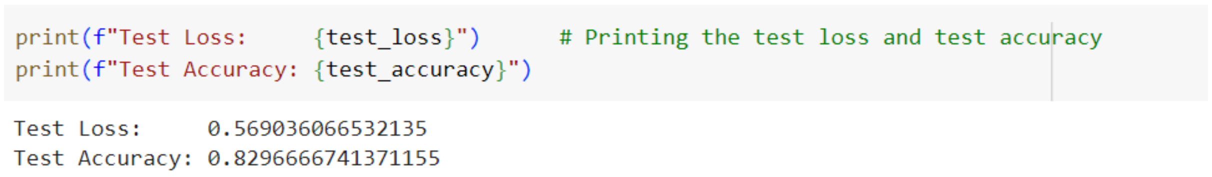
4. Results

4.1 Performance Metrics

The performance of the trained CNN model was evaluated using various metrics on the test dataset:

Convolutional Neural Network (CNN):

- Test Loss: 0.57
- Test Accuracy: 0.83



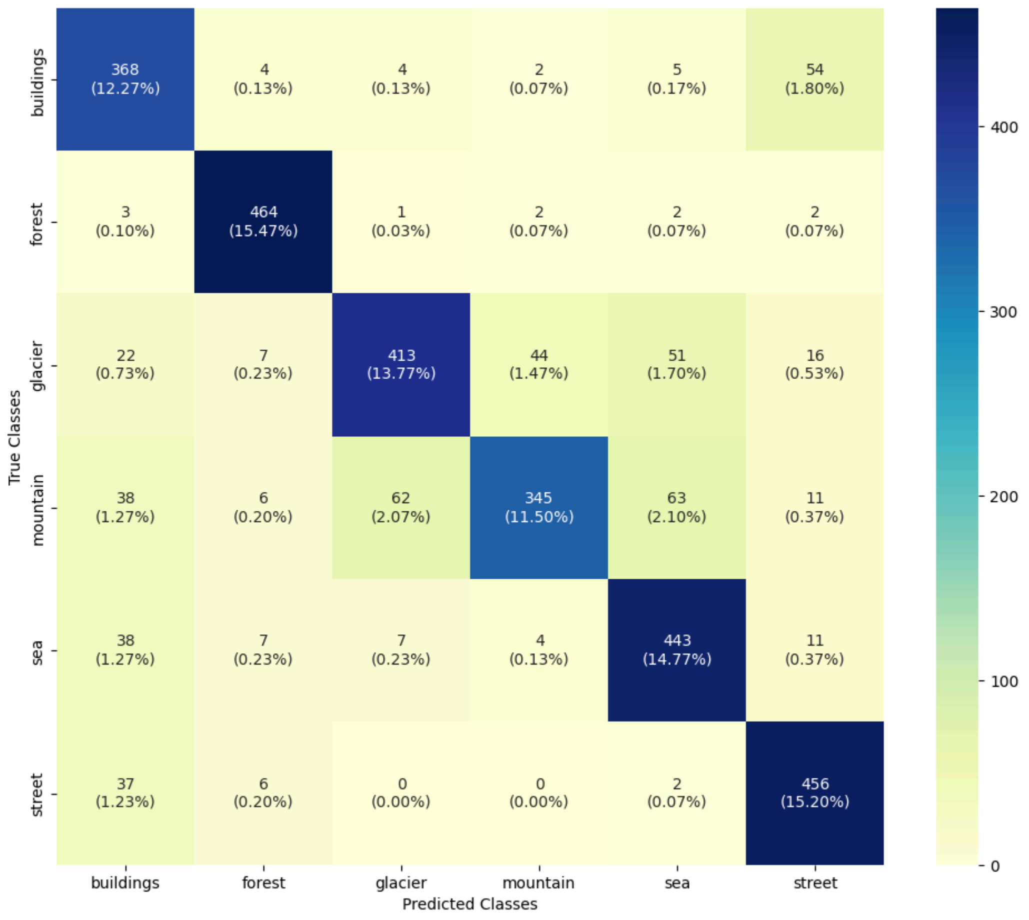
```
print(f"Test Loss:     {test_loss}")      # Printing the test loss and test accuracy
print(f"Test Accuracy: {test_accuracy}")
```

```
Test Loss:     0.569036066532135
Test Accuracy: 0.8296666741371155
```

Precision, Recall, and F1-score: These metrics were calculated for each class using the classification report. Some classes showed slightly lower precision and recall, indicating potential areas for improvement:

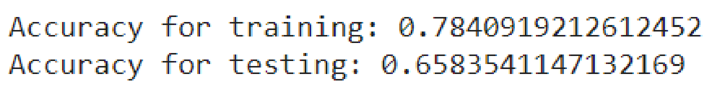
```
              precision    recall  f1-score   support

   buildings       0.73      0.84      0.78       437
      forest       0.94      0.98      0.96       474
     glacier       0.85      0.75      0.79       553
    mountain       0.87      0.66      0.75       525
         sea       0.78      0.87      0.82       510
      street       0.83      0.91      0.87       501

    accuracy                           0.83      3000
   macro avg       0.83      0.83      0.83      3000
weighted avg       0.83      0.83      0.83      3000
```

- A confusion matrix was generated to visualize the model's performance across different classes, highlighting areas of misclassification:



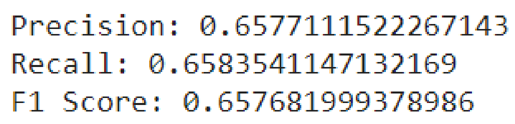
Logistic Regression Model:

- Accuracy Calculation: Accuracy is calculated for both training and testing sets using accuracy_score() function.



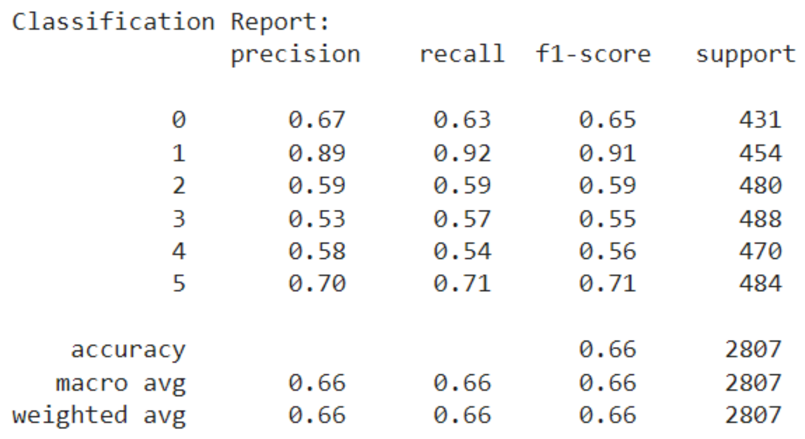
```
Accuracy for training: 0.7840919212612452
Accuracy for testing: 0.6583541147132169
```

- Other Metrics:



```
Precision: 0.6577111522267143
Recall: 0.6583541147132169
F1 Score: 0.657681999378986
```

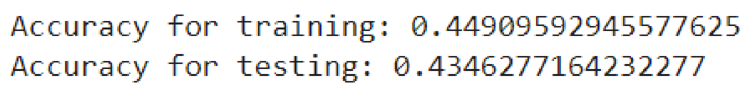
- Classification Report: The classification report provides a comprehensive summary of various metrics such as precision, recall, and F1-score for each class.



```
Classification Report:
              precision    recall  f1-score   support

           0       0.67      0.63      0.65       431
           1       0.89      0.92      0.91       454
           2       0.59      0.59      0.59       480
           3       0.53      0.57      0.55       488
           4       0.58      0.54      0.56       470
           5       0.70      0.71      0.71       484

    accuracy                           0.66      2807
   macro avg       0.66      0.66      0.66      2807
weighted avg       0.66      0.66      0.66      2807
```

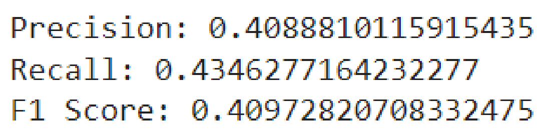
Decision Tree Classifier:

- Accuracy Calculation: Accuracy is calculated for both training and testing sets using accuracy_score().



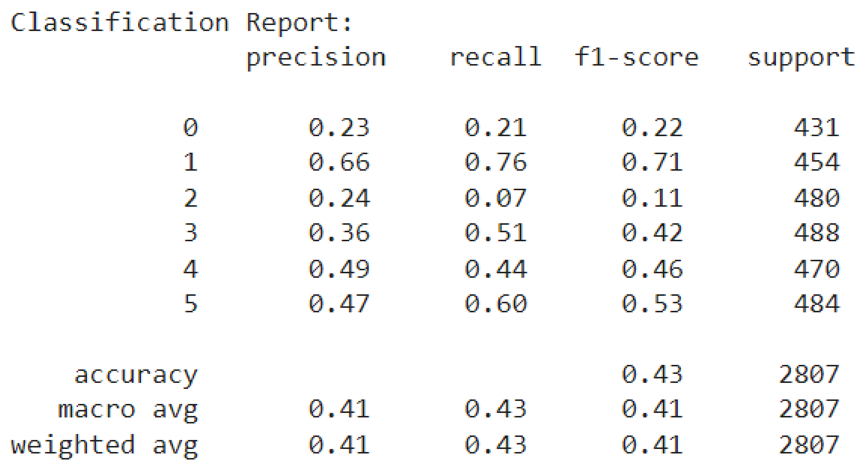
```
Accuracy for training: 0.44909592945577625
Accuracy for testing: 0.4346277164232277
```

- Other Metrics:



```
Precision: 0.4088810115915435
Recall: 0.4346277164232277
F1 Score: 0.40972820708332475
```

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

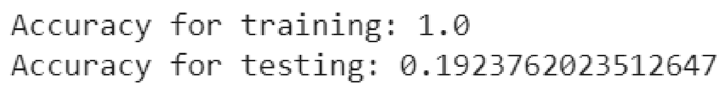


```
Classification Report:
              precision    recall  f1-score   support

           0       0.23      0.21      0.22       431
           1       0.66      0.76      0.71       454
           2       0.24      0.07      0.11       480
           3       0.36      0.51      0.42       488
           4       0.49      0.44      0.46       470
           5       0.47      0.60      0.53       484

    accuracy                           0.43      2807
   macro avg       0.41      0.43      0.41      2807
weighted avg       0.41      0.43      0.41      2807
```

Random Forest Classifier:

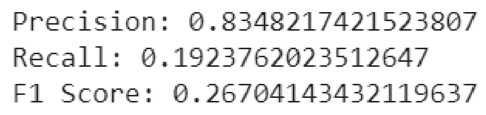
Model Evaluation

- Accuracy Calculation: Accuracy is calculated for both training and testing sets using accuracy_score().



```
Accuracy for training: 1.0
Accuracy for testing: 0.1923762023512647
```

- Other Metrics:



```
Precision: 0.8348217421523807
Recall: 0.1923762023512647
F1 Score: 0.26704143432119637
```

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

```
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.01      0.03       431
           1       0.94      0.67      0.78       454
           2       0.82      0.12      0.20       480
           3       0.72      0.05      0.09       488
           4       0.93      0.08      0.15       470
           5       0.76      0.23      0.36       484

   micro avg       0.87      0.19      0.32      2807
   macro avg       0.84      0.19      0.27      2807
weighted avg       0.83      0.19      0.27      2807
 samples avg       0.19      0.19      0.19      2807
```

Model evaluation for 2nd iteration:

```
# Predict on the test set
y_pred_test = rf_clf_2.predict(X_test)
y_pred_train = rf_clf_2.predict(X_train)
```

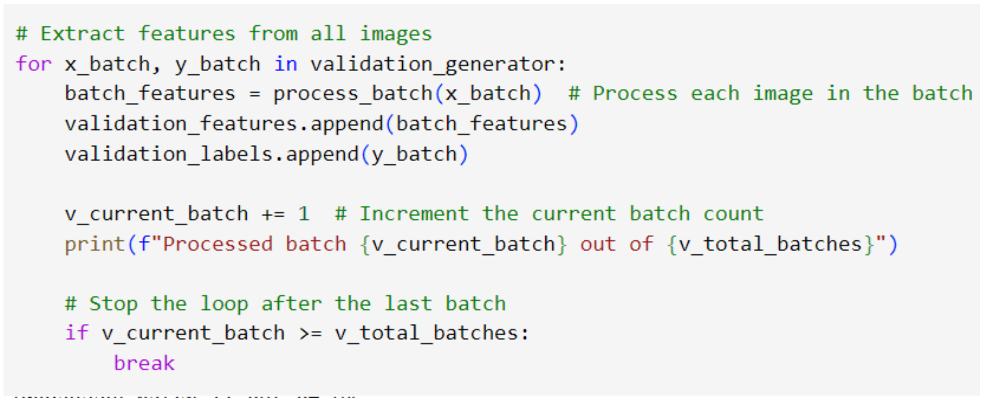
```
Accuracy for training: 0.3137970962857397
Accuracy for testing: 0.14463840399002495
Precision: 0.7211732505215358
Recall: 0.14463840399002495
F1 Score: 0.1973403710339356
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00       431
           1       0.93      0.65      0.76       454
           2       0.85      0.06      0.11       480
           3       0.76      0.03      0.05       488
           4       0.91      0.04      0.09       470
           5       0.81      0.10      0.18       484

   micro avg       0.90      0.14      0.25      2807
   macro avg       0.71      0.15      0.20      2807
weighted avg       0.72      0.14      0.20      2807
 samples avg       0.14      0.14      0.14      2807
```

Models Evaluation on Validation Set:

The validation set served as a critical component for assessing the performance of the trained models. Below outlines the methodology employed for evaluation along with the obtained results:

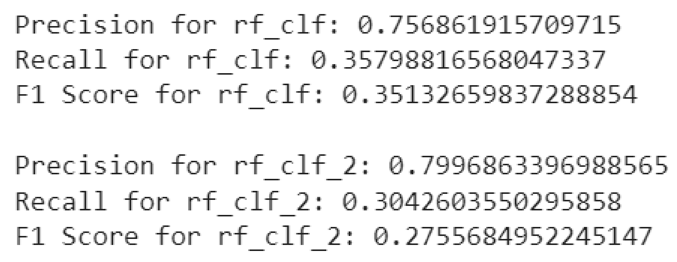
Methodology:

- Feature Extraction: Features were extracted from all images within the validation set using a predefined function process_batch. Subsequently, Principal Component Analysis (PCA) was applied to reduce the dimensionality of these features.



```python
# Extract features from all images
for x_batch, y_batch in validation_generator:
    batch_features = process_batch(x_batch)  # Process each image in the batch
    validation_features.append(batch_features)
    validation_labels.append(y_batch)

    v_current_batch += 1  # Increment the current batch count
    print(f"Processed batch {v_current_batch} out of {v_total_batches}")

    # Stop the loop after the last batch
    if v_current_batch >= v_total_batches:
        break
```

- Model Predictions: Utilizing the reduced features, predictions were made using three distinct models: the Decision Tree Classifier (clf), the Random Forest Classifier (rf_clf), and an alternative Random Forest Classifier with a maximum depth of 10 (rf_clf_2). These predictions were then converted into integer labels for evaluation purposes.
- Evaluation Metrics: The evaluation metrics employed were Precision, Recall, and F1 Score. Precision denotes the ratio of correctly predicted positive observations out of all predicted positives, while Recall measures the proportion of correctly predicted positive observations out of all actual positives. F1 Score represents the harmonic mean of precision and recall, offering a balanced assessment of a model's performance.



```
Precision for rf_clf: 0.756861915709715
Recall for rf_clf: 0.35798816568047337
F1 Score for rf_clf: 0.35132659837288854

Precision for rf_clf_2: 0.7996863396988565
Recall for rf_clf_2: 0.3042603550295858
F1 Score for rf_clf_2: 0.2755684952245147
```

4.2 Results Interpretation:

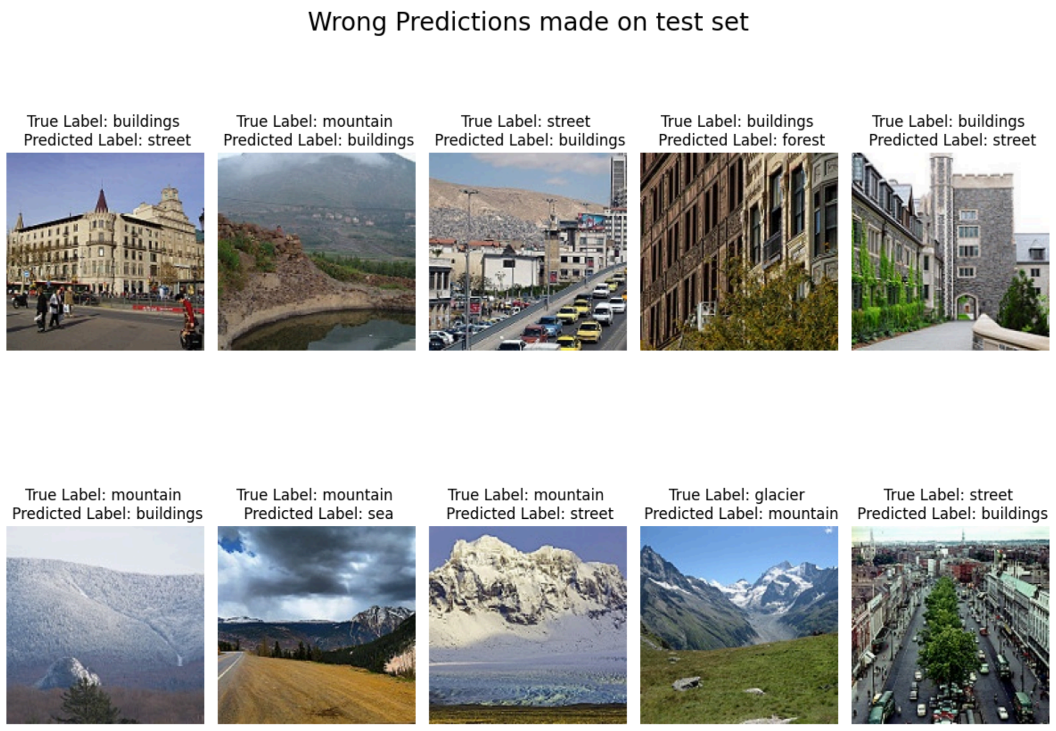
4.2.1 Exploring Misclassified Images

To gain deeper insights into the model's performance, we explore misclassified images from the test set. Misclassified images are instances where the model's predictions do not match the true labels. This analysis aids in understanding the specific challenges the model faces and potential areas for improvement.

```
errors = (y_true - y_pred != 0)
y_true_errors = y_true[errors]
y_pred_errors = y_pred[errors]
```

```
test_images = test_generator.filenames
test_img = np.asarray(test_images)[errors]
```

```
fig, ax = plt.subplots(nrows=2, ncols=5, figsize=(12, 10))
idx = 0

for i in range(2):
    for j in range(5):
        idx = np.random.randint(0, len(test_img))
        true_index = y_true_errors[idx]
        true_label = labels[true_index]
        predicted_index = y_pred_errors[idx]
        predicted_label = labels[predicted_index]
        ax[i, j].set_title(f"True Label: {true_label} \n Predicted Label: {predicted_label}")
        img_path = os.path.join(test_dataset, test_img[idx])
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        ax[i, j].imshow(img)
        ax[i, j].axis("off")

plt.tight_layout()
plt.suptitle('Wrong Predictions made on test set', fontsize=20)
plt.show()
```



4.2.2 Pickle Files

To preserve the trained CNN model and preprocessing objects for future use or deployment, we save them as pickle files:

```python
import pickle

# Save trained CNN model
cnn_model.save('cnn_model.h5')

# Save labels dictionary
with open('labels.pickle', 'wb') as handle:
    pickle.dump(labels, handle, protocol=pickle.HIGHEST_PROTOCOL)

# Save ImageDataGenerator configuration
with open('train_datagen.pickle', 'wb') as handle:
    pickle.dump(train_datagen, handle, protocol=pickle.HIGHEST_PROTOCOL)

# Save test_generator filenames
with open('test_generator_filenames.pickle', 'wb') as handle:
    pickle.dump(test_generator.filenames, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

4.3 UI Development

The API accepts image uploads and returns classification results and confidence levels

4.3.1 Technologies Used

- Flask: Web framework for handling HTTP requests.
- TensorFlow/Keras: For loading and utilizing the CNN model.
- Pillow (PIL): To process and prepare images for prediction.

4.3.2 Model and API Details

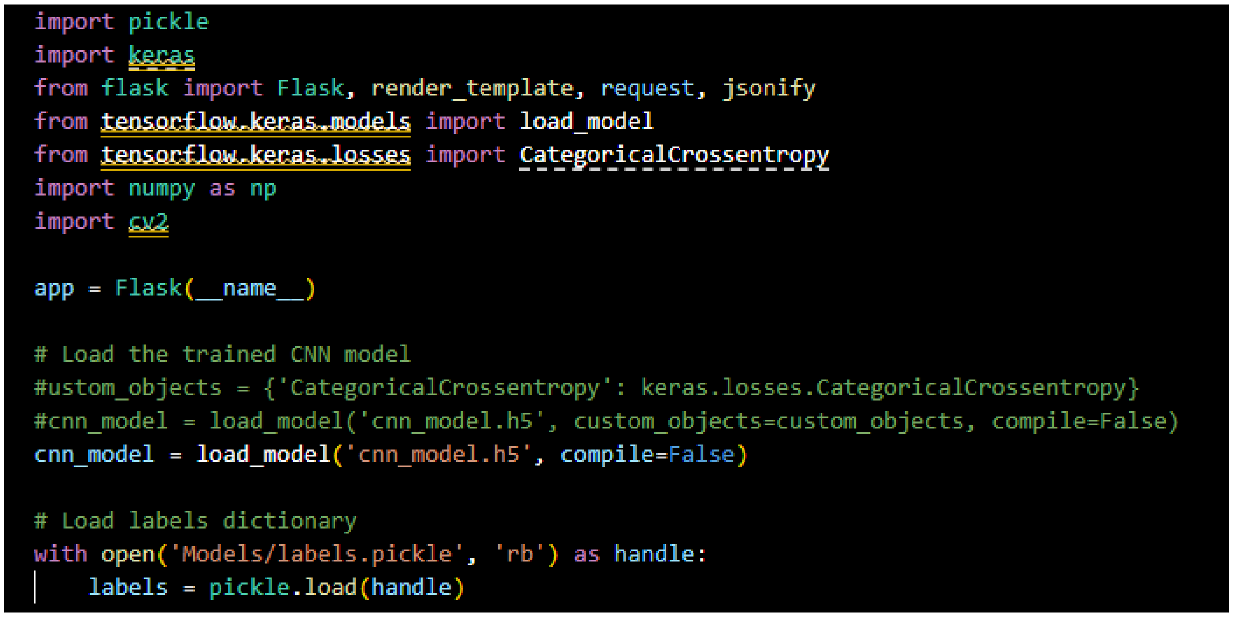
- The model (cnn_model.h5) is pretrained and expects images resized to 150x150 pixels, normalized by a factor of 255.
- The API endpoint (/predict) handles POST requests where users upload an image, and it returns the predicted class and confidence.

4.3.3 Implementation Summary

- Model Loading: The CNN model is loaded at startup.
- Endpoint Functionality:
    - Checks if the image file is included in the request.
    - Opens, resizes, and normalizes the image.
    - Uses the model to predict the class and returns the result.
- Execution: Flask app runs with debug mode enabled for development purposes.

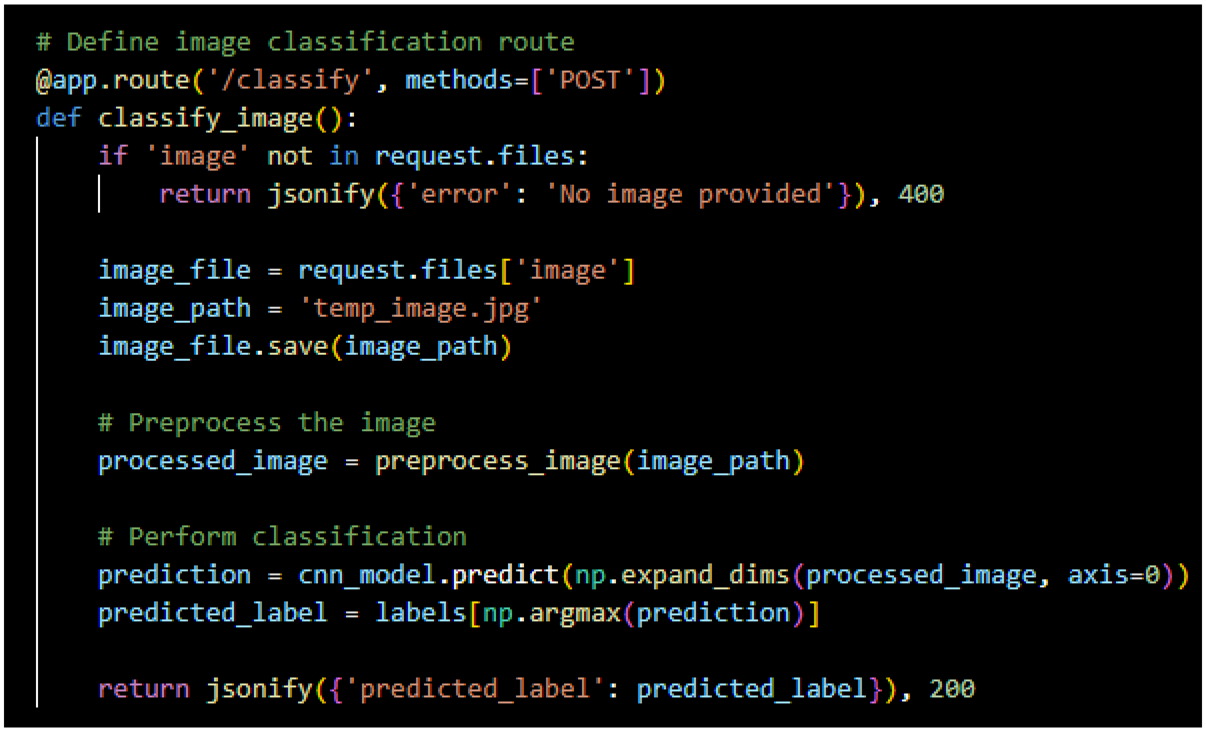
4.3.4 Code

Model and Dependencies Loading:



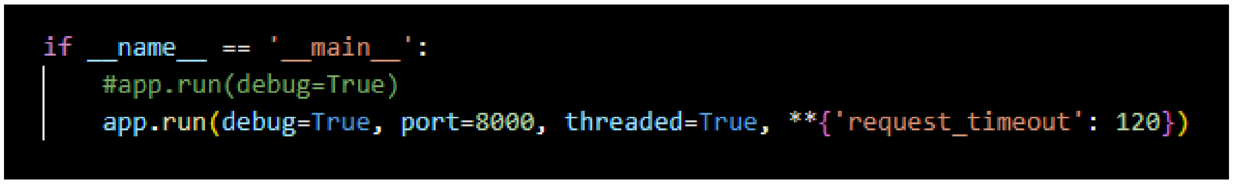
```python
import pickle
import keras
from flask import Flask, render_template, request, jsonify
from tensorflow.keras.models import load_model
from tensorflow.keras.losses import CategoricalCrossentropy
import numpy as np
import cv2

app = Flask(__name__)

# Load the trained CNN model
#ustom_objects = {'CategoricalCrossentropy': keras.losses.CategoricalCrossentropy}
#cnn_model = load_model('cnn_model.h5', custom_objects=custom_objects, compile=False)
cnn_model = load_model('cnn_model.h5', compile=False)

# Load labels dictionary
with open('Models/labels.pickle', 'rb') as handle:
    labels = pickle.load(handle)
```

Image Classification Route:



```python
# Define image classification route
@app.route('/classify', methods=['POST'])
def classify_image():
    if 'image' not in request.files:
        return jsonify({'error': 'No image provided'}), 400

    image_file = request.files['image']
    image_path = 'temp_image.jpg'
    image_file.save(image_path)

    # Preprocess the image
    processed_image = preprocess_image(image_path)

    # Perform classification
    prediction = cnn_model.predict(np.expand_dims(processed_image, axis=0))
    predicted_label = labels[np.argmax(prediction)]

    return jsonify({'predicted_label': predicted_label}), 200
```

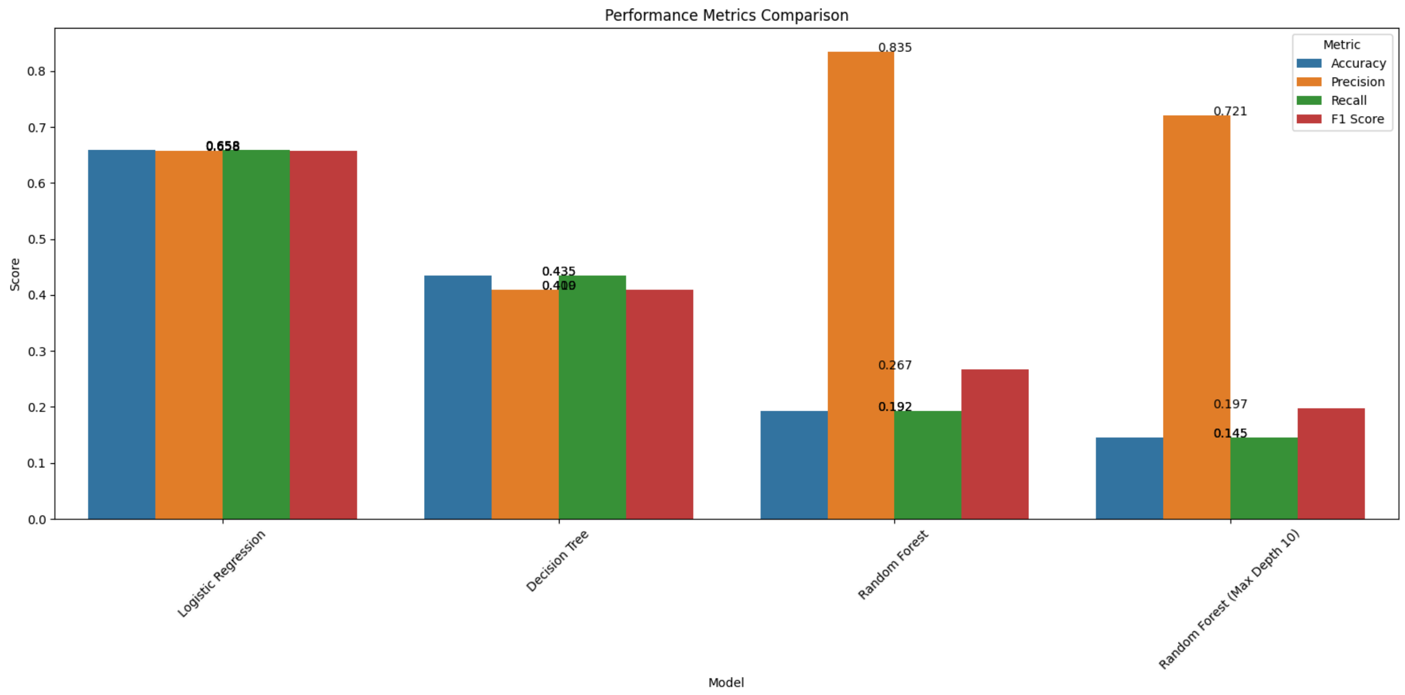
Flask Application Runner:

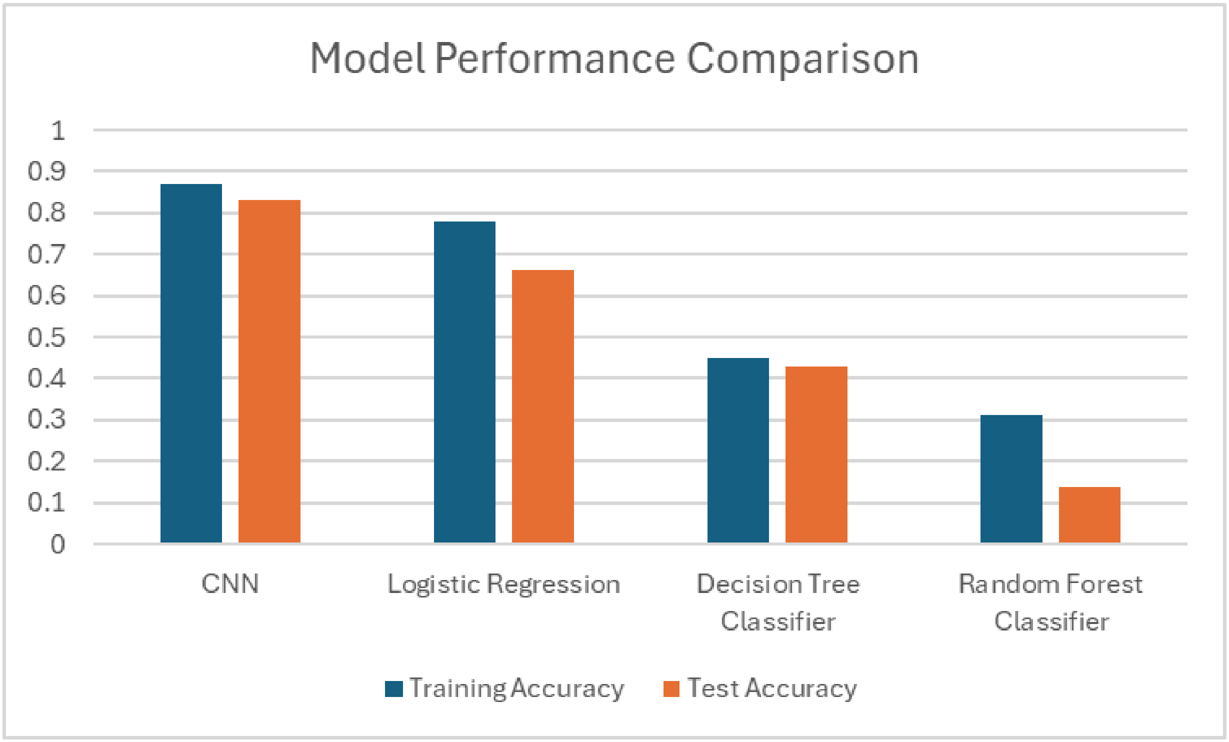


```python
if __name__ == '__main__':
    #app.run(debug=True)
    app.run(debug=True, port=8000, threaded=True, **{'request_timeout': 120})
```

6. Conclusion

6.1 Evaluation Overview:





1. Convolutional Neural Network (CNN):
   - Performance: Achieves the highest test accuracy of 83%, showcasing balanced metrics across precision, recall, and F1-scores for various classes.
   - Assessment: Demonstrates robust capabilities, although some classes (e.g., glacier and mountain) exhibit room for improvement.
2. Logistic Regression (LR):
   - Performance: Attains a moderate overall accuracy of 65.8%, with consistent yet unremarkable precision and recall across the dataset.
   - Assessment: Despite attempts at hyperparameter tuning, enhancements remain limited, suggesting logistic regression may not be optimal for this complex multi-class scenario.
3. Decision Tree Classifier (DTC):
   - Performance: Limited by a low accuracy of approximately 43%, the model struggles with precision and recall, indicating potential overfitting or inadequate complexity.

- Assessment: Underperforms in accurately classifying various classes, highlighting significant efficacy concerns.

4. Random Forest Classifier (RFC):
   - Performance: Particularly poor in the revised setup, with an accuracy of only 14.4% in testing scenarios.
   - Assessment: Despite high precision in specific classes, the extremely low recall points to a critical failure in identifying most positive instances, limiting its utility.

6.2 Optimal Model Recommendation:

Based on the gathered data, CNN is identified as the most effective model. It not only achieves the highest accuracy but also maintains consistent performance across various metrics. It is recommended for further development and deployment.

6.3 Recommendations for Future Work:

While our project has made significant strides in enhancing the usability and effectiveness of image classification there remain opportunities for further improvement:

- Data Augmentation: Focus on increasing the data variability for underperforming classes to bolster the CNN's learning capability.
- Advanced Architectures: Investigate alternative and more complex CNN architectures to explore potential gains in model accuracy and robustness.
- Feature Engineering: For logistic regression and decision trees, enhanced feature engineering could be pivotal in improving model predictions.
- Ensemble Methods: Consider deploying ensemble techniques that combine the strengths of multiple models to improve accuracy and reliability.

7. References

[1] Dataset URL: https://www.kaggle.com/datasets/puneet6060/intel-image-classification

[2] Brownlee, J. (n.d.). Logistic Regression for Image Classification using OpenCV. Machine Learning Mastery. Retrieved from https://machinelearningmastery.com/logistic-regression-for-image-classification-using-opencv/

[3] Raman, A. (2020, February). Learn Image Classification Using CNNs: Convolutional Neural Networks. Analytics Vidhya. Retrieved from https://www.analyticsvidhya.com/blog/2020/02/learn-image-classification-cnn-convolutional-neural-networks-3-datasets/

[4] Rao, A. (2022, January). Image Classification Using Machine Learning. Analytics Vidhya. Retrieved from https://www.analyticsvidhya.com/blog/2022/01/image-classification-using-machine-learning/

[5] Gonçalves, J. (September 6, 2023). Image Classification with CNN in Python [LinkedIn article]. Retrieved from https://www.linkedin.com/pulse/image-classification-cnn-python-jo%C3%A3o-gon%C3%A7alves/

[6] Prabhu, R. (Mar 4, 2018). Understanding of Convolutional Neural Network (CNN) - Deep Learning [Medium article]. Retrieved from https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148

[7] Merfarukgnaydn. (Date of publication). 90% Accuracy CNN + FeatureExtraction + FineTuning [Kaggle notebook]. Retrieved from https://www.kaggle.com/code/merfarukgnaydn/90-accuracy-cnn-featureextraction-finetuning