Landscape Recognition with Bayesian Classifier

Mruganshi Gohel

B20CS014

Introduction

This assignment centers on the use of holistic features for context recognition, It involves applying these concepts to a Kaggle image recognition dataset, specifically for landscape recognition. The goal is to extract GIST features, and develop a Bayesian classifier for categorizing landscapes. The project encompasses pre-processing images, creating the classifier, estimating its parameters for multiple landscape categories, testing its performance, and producing a comprehensive report on the results. This assignment offers a practical application of image classification and context recognition techniques learned in the course.

Objectives

1. **Feature Extraction:** Extract GIST features from a landscape image dataset.
2. **Bayesian Classifier:** Develop a Bayesian classifier for landscape recognition.
3. **Parameter Estimation:** Estimate network parameters for multiple landscape categories.
4. **Performance Evaluation:** Test the classifier and evaluate its accuracy.

In the sections, we will provide explanations of each technique used in every part to showcase intermediate and final results

Part-1: Data Exploration and Preprocessing

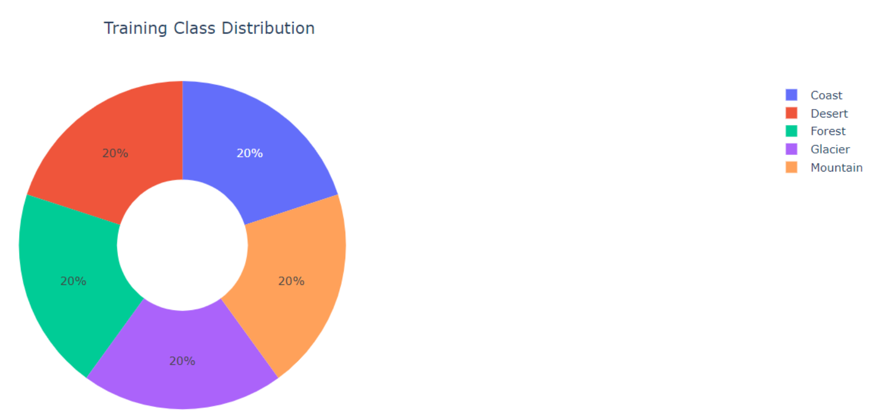
Load and explore the provided dataset and Split the dataset into training and testing sets.

Begin by explaining the process of obtaining the dataset. we've utilized Kaggle to download the landscape recognition image dataset.for that we have downloaded kaggle.json for credintials and to directly access the dataset without saving into the drive or local PC.

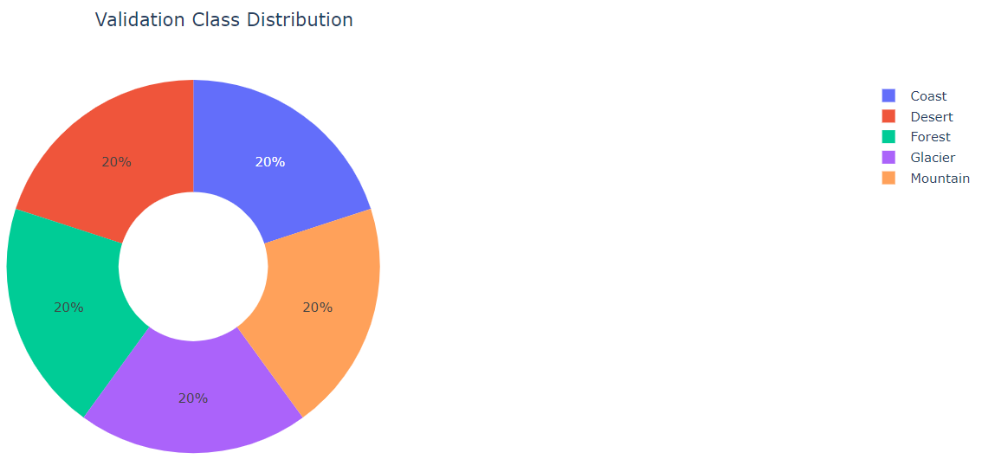
Before moving ahead let's look at how the **classes are distributed in the dataset i.e Class Distribution**.

```
Number of Classes : 5
Class names : ['Coast', 'Desert', 'Forest', 'Glacier', 'Mountain']
```

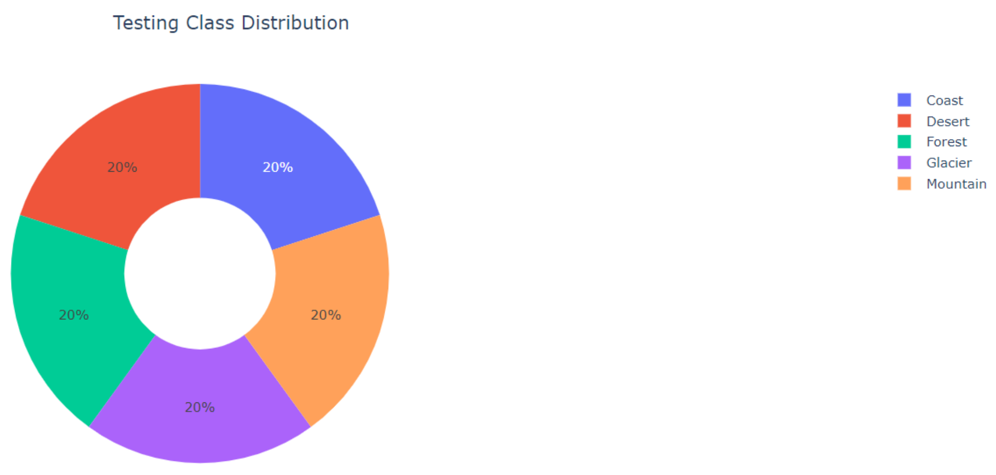
distribution of data among all classes



all the training classes are **equally distributed**. This ensures that our **model cannot be biased towards any class.**

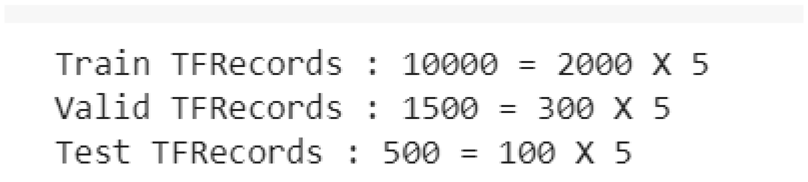


even the **validation data have equal class distribution**. This ensures **fair evaluation of the model during training**.



The **testing data also have equal class distribution**. This make sure that the model will be **fairly tested**.

for data loading It will be a **great idea** to load the **tensorflow records** as they are **fast and efficient**.



```
Train TFRecords : 10000 = 2000 X 5
Valid TFRecords : 1500 = 300 X 5
Test TFRecords : 500 = 100 X 5
```

The **class distribution** is preserved in **tensorflow records**. **Tensorflow records** have all images resized to **256 x 256 pixels**.

Preprocessing, Feature Extraction and Quantization of features

for preprocessing, I have written three functions `decode_image` , `decode_data`, **and** `load_data`. **Now I will explain all three functions**

Preprocessing

1. decode_image

   This function takes a binary image encoded as a string and performs the following steps:

   - Decodes the JPEG image into a tensor with three color channels (RGB).
   - Casts the pixel values to float32 for numerical compatibility.

- Normalizes the pixel values to be within the range [0, 1].

2. decode_data

   This function decodes individual examples stored in TFRecord files. It does the following:

   - Defines the expected features in each example, which include an image as a string and a label as an integer.
   - Parses a single example from the TFRecord by using the specified feature definitions.
   - Calls the `decode_image` function to process the image.

3. load_data

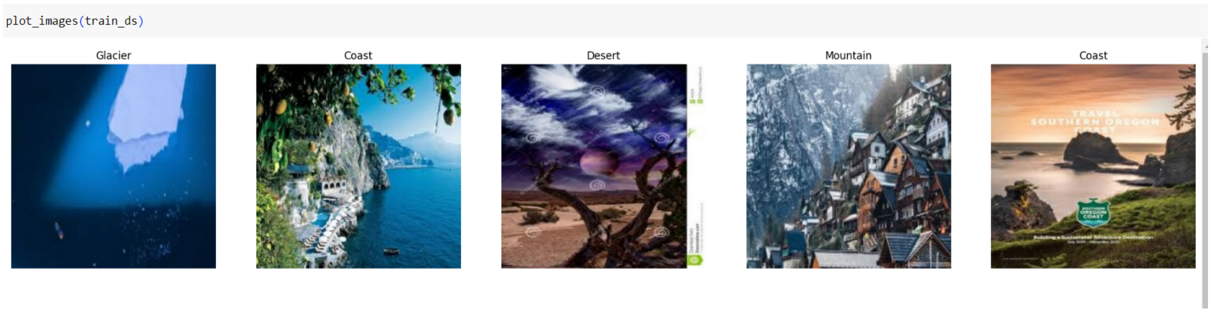
   This function loads and prepares the data from TFRecord files for training, validation, or testing. It does the following:

   - Appends the ".tfrecord" file extension to the specified directory path.
   - Lists the TFRecord files in the directory.
   - Reads the TFRecord files in parallel, utilizing automatic parallelism (AUTOTUNE).
   - Maps the `decode_data` function to each example to decode images and labels.
   - Shuffles the data and creates batches with the specified batch size, dropping any remaining samples that don't form a complete batch.
   - Prefetches data to improve training performance.

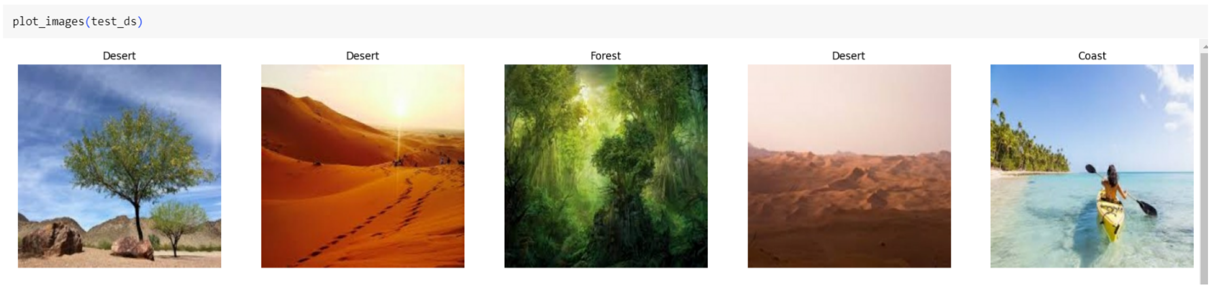
   Finally, the code loads and prepares the training, validation, and test datasets using the `load_data` function. These datasets are then ready to be used for training a machine-learning model.

V**isualize data**

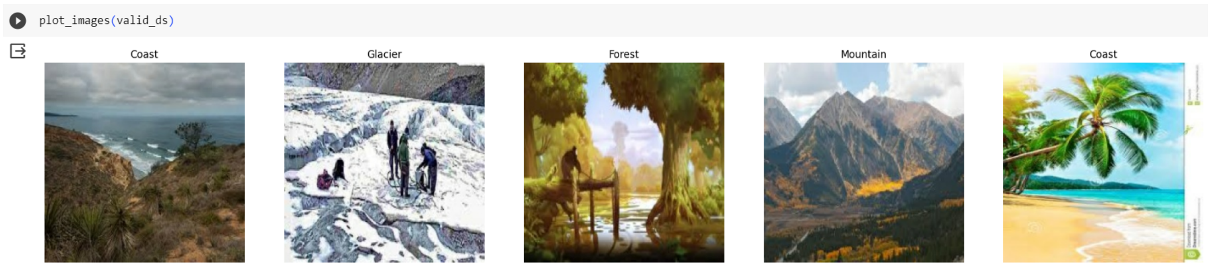
for visualization of all images, I have written the function plot_images which will plot some images of training, testing and validation data.



```
plot_images(train_ds)
```



```
plot_images(test_ds)
```



```
plot_images(valid_ds)
```

Feature Extraction using GIST

In the field of computer vision and image analysis, feature extraction is a fundamental step in understanding and processing images. One such feature extraction technique is the GIST (Global Image Structure) method, which allows for the representation of the spatial structure of an image. This report presents an implementation of the GIST feature extraction method using a Python-based approach. The implementation's objective is to extract meaningful features from a dataset of landscape images for potential image classification tasks.

Implementation

`create_gabor_filters(orientations, n)`

- This function generates Gabor filters based on the specified orientations and image size.

- It calculates the parameters for each Gabor filter, such as frequency, size, and orientation.
- The resulting Gabor filters are stored in a 3D array.
- These filters will be used for feature extraction.

`more_config(img)`

- This function configures the GIST parameters based on the input image.
- It updates the `image_size` and `gabor_filters` in the parameter dictionary.
- The `gabor_filters` are created using the `create_gabor_filters` function.

`preprocess(img)`

- Preprocessing is essential for ensuring images are of a consistent size before feature extraction.
- This function resizes the image based on the predefined `image_size` in the parameters.
- It also normalizes the pixel values of the image.

`prefilt(img)`

- This function implements the prefilt stage of GIST feature extraction.
- It applies a log transformation to the image and pads it symmetrically.
- Spatial filtering is performed using Gabor-like filters.
- The output is divided by the local standard deviation.

`downN(x, N)`

- This function downsamples the image for block-wise processing.
- It computes the mean value of each block within the image.
- The result is a downsampled representation.

`gistGabor(img)`

- GIST feature extraction is performed here.
- The `image` is processed using the pre-configured `gabor_filters`.
- The output is divided into blocks, and mean values are computed for each block.

- The final GIST feature vector is generated.

`pipeline_execution(img)`

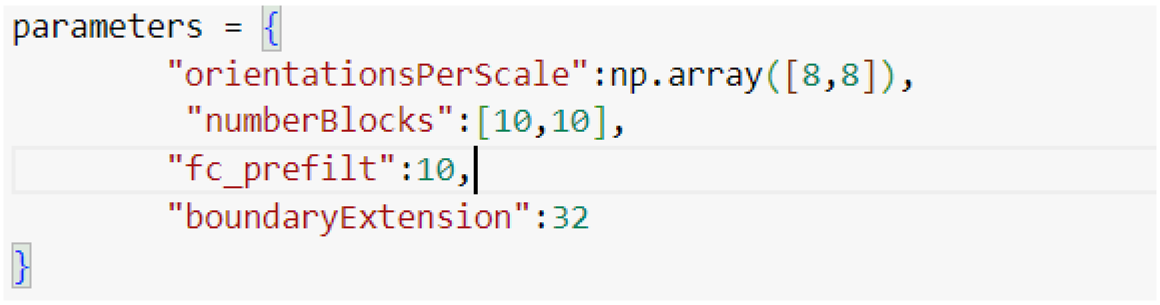
- This function orchestrates the entire GIST feature extraction process.
- It first configures the parameters with `more_config`.
- Then, it preprocesses the image using `preprocess`.
- The `prefilt` function is applied for spatial filtering.
- Finally, `gistGabor` computes the GIST feature vector.

`_get_gist(file_list)`

- This function is responsible for parallelizing the GIST feature extraction process.
- It uses the `multiprocessing` library to distribute the image processing across multiple CPU cores.
- The input `file_list` contains paths to image files.
- The `pipeline_execution` function is applied in parallel to each image.
- The resulting GIST feature vectors are collected and returned as a NumPy array.

`param` dictionary is essential for setting and configuring parameters that influence the entire GIST feature extraction process. It ensures consistency and flexibility when extracting features from images.

The GIST feature extraction process involves a sequence of steps, including image preprocessing, spatial filtering, and block-wise processing. These functions work together to capture the global spatial structure of images, making GIST features suitable for various computer vision tasks. By parallelizing the feature extraction, we can efficiently process a large number of images.



```
parameters = {
        "orientationsPerScale":np.array([8,8]),
         "numberBlocks":[10,10],
        "fc_prefilt":10,
        "boundaryExtension":32
}
```

Feature Quantization

The `quantize_features` function takes a set of input features and quantizes them into a specified number of categories. It uses numpy's `digitize` function to perform this quantization. Here's a brief description of how the function works:

- `input_features`: This is an array or list of numerical features that we want to quantize.
- `num_categories`: This parameter specifies the number of categories or bins into which the input features should be quantized. By default, it is set to 5.

Here's how the function works:

1. It calculates the bin edges based on the input features. The `np.linspace(0, input_features.max(), num_categories)` function generates an array of evenly spaced values starting from 0 to the maximum value of the input features, divided into `num_categories` bins.
2. The `np.digitize` function is then applied to the input features using the computed bin edges. It assigns each feature to a category by finding the index of the bin in which it falls. The result is an array of quantized values, where each value corresponds to a specific category.
3. The quantized input is returned as the output of the function.

Part-2: Bayesian Network Classifier

Bayesian Network Classifiers are probabilistic graphical models that allow for effective classification tasks.

Define the Bayesian Network Structure

- **Feature Selection**: Choose relevant features from the dataset. These features are variables that are used for classification. Identify which features are relevant to the problem.
- **Network Structure**: Define the conditional dependencies between features. This is often based on domain knowledge or data analysis. we create a network graph where nodes represent features, and edges represent conditional dependencies. For example, in a medical diagnosis problem, if we know that symptoms A and B are conditionally dependent on the presence of a disease C, we would represent this as nodes A and B having an edge pointing to node C.

Gather Training Data

- Collect a labeled dataset that includes feature values and corresponding class labels. This dataset will be used to train our Bayesian Network Classifier. Each data point should have values for all selected features and the corresponding class label.

Parameter Estimation

- Calculate conditional probability distributions for each node (feature) in the network using the training data. This involves estimating the probabilities of each feature given its parent(s) in the network. For instance, if we have a Bayesian network with nodes A and B, where B depends on A, we estimate P(B|A).
- we can use techniques such as Maximum Likelihood Estimation (MLE) or Bayesian estimation to calculate these probabilities.

Inference

- Implement the inference mechanism for classification. When we have a new data point with observed feature values, we want to calculate the posterior probabilities of class labels.
- Use Bayes' theorem to calculate these probabilities. For each class, we compute P(Class | Features) based on the product of conditional probabilities of the features given the class.
- Select the class label with the highest posterior probability as the predicted classification.

BNCs leverage the conditional dependencies among features to classify data. For a new data point, the network calculates the posterior probabilities of different class labels. The class label with the highest posterior probability is selected as the prediction.

Part-3: Training the Bayesian network - Parameter estimation

Estimate class priors

Class prior probabilities are the probabilities of each class (landscape type) in the dataset.

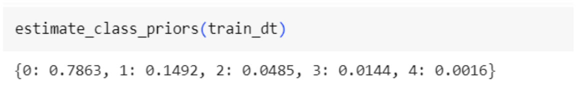
Here's how to estimate the class prior probabilities:

- Count the number of data points (instances) in the training data for each class.

- Divide each class count by the total number of data points to get the probability of each class.

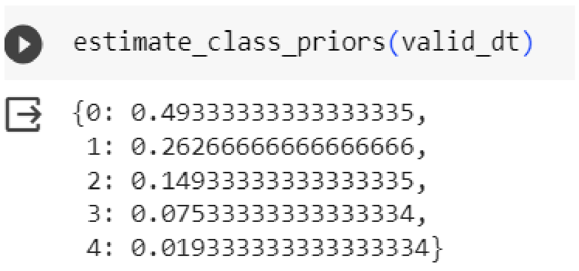
Here's how it works step by step:

1. `class_counts` is a dictionary that stores the count of instances for each class label in our training data.
2. The loop iterates through each instance in `mydata`, and for each instance, it extracts the class label (usually located in the last position of each instance).
3. It increments the count for that class label in `class_counts`.
4. After processing all instances, `class_counts` contains the count of instances for each class label in our dataset.
5. `total_instances` is calculated as the total number of instances in our dataset.
6. The loop then calculates the class priors by dividing the count of each class label by the total number of instances, and stores them in the `class_priors` dictionary.
7. Finally, `class_priors` is returned, which contains the estimated class prior probabilities.



```
estimate_class_priors(train_dt)
```

```
{0: 0.7863, 1: 0.1492, 2: 0.0485, 3: 0.0144, 4: 0.0016}
```



```
estimate_class_priors(valid_dt)
```

```
{0: 0.49333333333333335,
 1: 0.26266666666666666,
 2: 0.14933333333333335,
 3: 0.07533333333333334,
 4: 0.019333333333333334}
```

Feature Likelihoods

- For each feature X and each class C, calculate the probability distribution of feature X for data points in class C. This involves counting the occurrences of each feature value within the class C.

Mathematically:

P(X=x|C=c) = Count(X=x, C=c) / Count(C=c)

Here's how the function works:

1. It initializes an empty dictionary called `class_summaries` to store the feature likelihoods for each class.

2. The function then iterates over each unique class label present in the last column of the training data (`mydata`).
3. For each class label, it selects the instances in the dataset that belong to that class using boolean indexing. The `class_instances` variable holds the instances of the current class.
4. Within the loop, it calculates feature summaries for each attribute (feature) within the class instances. The `feature_summaries` list is constructed by calculating the mean and standard deviation of each attribute.
5. The `feature_summaries` list, containing tuples of (mean, standard deviation) for each attribute, is associated with the current class label in the `class_summaries` dictionary.
6. This process repeats for each class label.
7. Finally, the `class_summaries` dictionary, which contains the feature likelihoods for each class, is returned.

```
estimate_feature_likelihoods_by_class(test_dt)
```

```
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.0, 0.0),
  (1.1428571428571428, 0.37796447300922725),
  (1.0, 0.0),
```

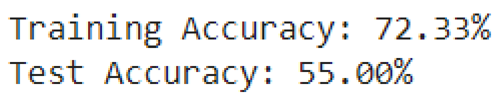
Part-4: Evaluation and Interpretation

Performance metrics calculation and reporting

In the context of a Bayesian Network Classifier applied to the given dataset, it's crucial to evaluate its performance using relevant metrics to assess its effectiveness in classifying landscape types. we discuss three primary performance metrics: accuracy, classification report, and confusion matrix, and present their interpretations and justifications.

1. Accuracy

   training accuracy, testing accuracy respectively defines accuracy for classification of training and testing dataset.

   ```
   Training Accuracy: 72.33%
   Test Accuracy: 55.00%
   ```

   <u>Interpretation and Justification</u>

   Accuracy is an intuitive metric that quantifies the overall performance of the classifier. It represents the percentage of correctly classified instances. High accuracy indicates that the classifier is effective in differentiating between landscape types. However, accuracy alone may not provide the complete picture. It's essential to examine other metrics to understand the classifier's behavior in more detail.

2. classification report

   The classification report provides a comprehensive summary of different metrics for each class, including precision, recall, F1-score, and support.

   - **Precision**: Precision measures the ability of the classifier to correctly identify instances of a class out of all instances it predicted as that class. A high precision score indicates fewer false positives.
   - **Recall**: Recall, or sensitivity, measures the ability of the classifier to identify all instances of a specific class. A high recall score indicates fewer false negatives.
   - **F1-Score**: The F1-score is the harmonic mean of precision and recall. It balances the trade-off between precision and recall, making it useful when there is an imbalance between classes.

- **Support**: Support is the number of instances of each class in the dataset.

justification and interpretation:

The classification report breaks down the performance metrics by class, making it possible to identify how well the classifier performs for each landscape type. This is particularly important when classes are imbalanced, as precision and recall can reveal potential issues. For this specific problem, it allows us to understand if the classifier is better at identifying one landscape type over another.

3. Confusion Matrix

   The confusion matrix is a table that illustrates the classifier's performance by showing the counts of true positive, true negative, false positive, and false negative predictions for each class.

   Interpretation and Justification:

   The confusion matrix is a fundamental tool for understanding the classifier's performance. It provides insights into the types and quantity of misclassifications, helping to identify where the classifier is making errors. This information is valuable for further fine-tuning the model or for understanding the practical implications of misclassifications in a real-world scenario.

Data Analysis

1. **Accuracy**:

   - **Training Accuracy (72.33%)**: The training accuracy indicates that the classifier performs relatively well on the training dataset. It correctly classifies approximately 72.33% of the training instances.
   - **Test Accuracy (55.00%)**: The test accuracy is lower than the training accuracy, indicating that the classifier might be overfitting to the training data. It correctly classifies about 55.00% of the test instances.

   Conclusion: The classifier shows decent performance on the training data but struggles when presented with unseen data. Overfitting may be a concern, and additional model tuning or regularization may be necessary.

2. **Classification Report**:

   - The classification report breaks down performance by each landscape type, providing precision, recall, and F1-score for each class.

   Training Report:

- For 'Coast,' precision is high (0.80), indicating that when the classifier predicts 'Coast,' it's often correct. The F1-score is 0.84, signifying a good balance between precision and recall.
- Other classes, such as 'Forest' and 'Glacier,' have lower precision and F1-scores, indicating that the classifier struggles with these classes.

Test Report:

- The test classification report reveals lower precision, recall, and F1-scores for most classes. This suggests that the classifier's performance is weaker on unseen data.

Conclusion: The classification report highlights the challenges faced by the classifier in differentiating between classes. 'Coast' is classified more accurately, while 'Forest' and 'Glacier' pose significant challenges. The classifier may need additional features or more training data for improved performance.

3. **Confusion Matrix**:

- The confusion matrix provides a detailed breakdown of correct and incorrect predictions for each class.

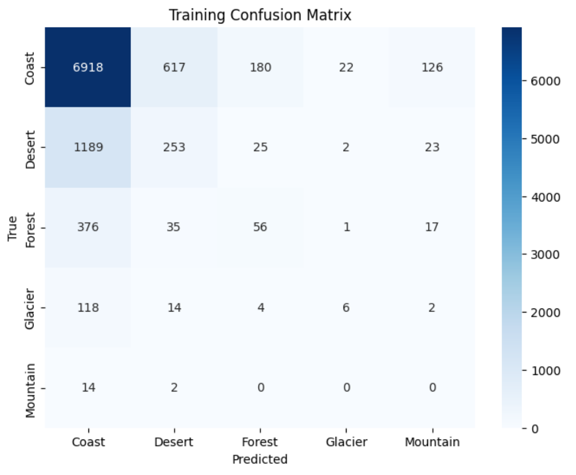
Training Confusion Matrix:

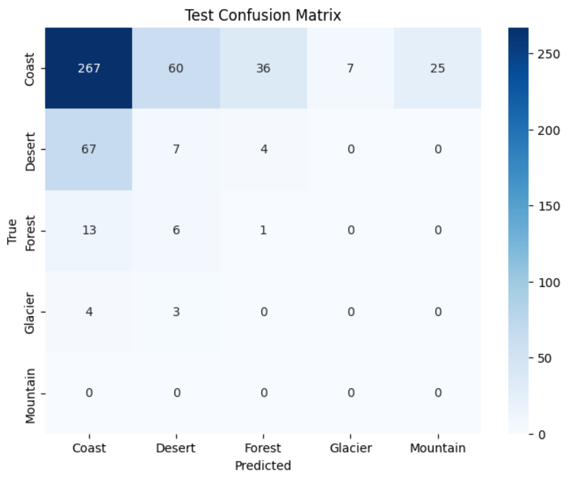
- 'Coast' has a high number of true positives, but there are notable false positives in 'Desert.'
- 'Forest' and 'Glacier' show fewer true positives and many false negatives, indicating difficulty in correctly classifying instances from these classes.

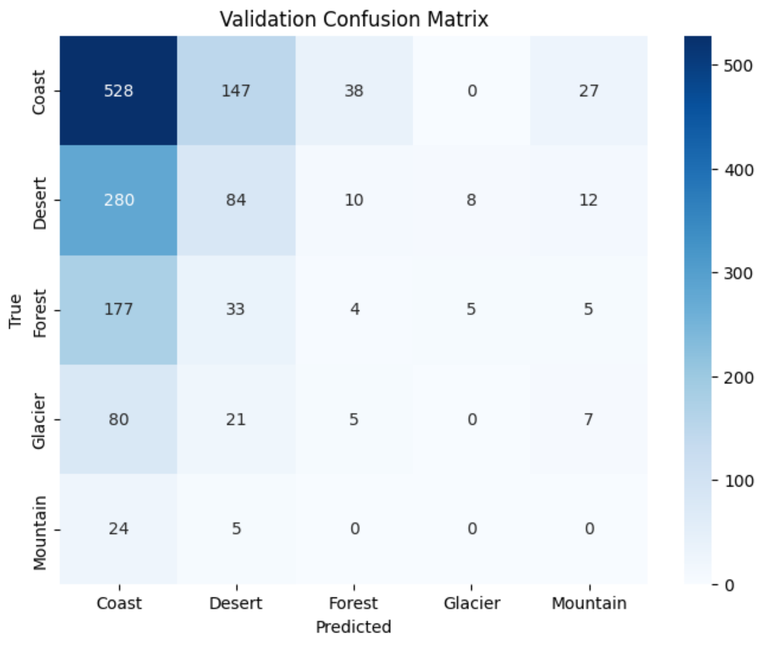
Test Confusion Matrix:

- The test confusion matrix further highlights the classifier's challenges, as it contains several false positives and false negatives.

Conclusion: The confusion matrices reinforce the challenges faced by the classifier, especially in differentiating 'Forest' and 'Glacier' from other classes. The high number of false positives and false negatives suggests the need for feature engineering and model refinement.







Visualize Predictions

A subset of 10 test images was randomly selected, and the model's predictions were obtained. The ground truth labels and predicted labels were compared for each image.

For each image in the subset, we have calculated posterior probabilities for all classes. We will show the predicted class, its associated posterior probability, and the class with the second-highest posterior probability.

Let's discuss some instances of correct and incorrect predictions:

- **Correct Predictions**:
    - In some cases, the model correctly predicts the landscape type (e.g., 'Coast') with a high posterior probability. This indicates that the classifier is confident and accurate in these instances.
- **Incorrect Predictions**:
    - There are instances where the model makes incorrect predictions. For example, the model may incorrectly classify a 'Desert' landscape as 'Coast.' These errors might occur when similar visual features or patterns are present in different landscape types, leading to confusion.
    - 'Forest' and 'Glacier' classes have notably lower precision, recall, and F1-scores. Incorrect predictions for these classes are prevalent. The classifier might lack distinctive features to accurately classify these landscapes.

The analysis of this subset of test images reveals that the model's performance varies across different images. Some images are correctly classified, while others are misclassified. The variations may be due to the inherent challenges in distinguishing between certain landscape types, especially when features are similar or ambiguous. To improve the model's performance, feature engineering, model optimization, and the inclusion of more training data for underrepresented classes could be considered. Additionally, analyzing the posterior probability values and the influence of features on predictions may provide insights into specific cases of misclassification. Further refinement of the classifier is recommended to enhance its overall accuracy and reliability.