Summary

This code trains and evaluates a Random Forest Classifier to predict deforestation events based on land use and tree cover data. The input data consists of a stack of raster files, including land use plans, tree cover, and historical deforestation data. The model uses these raster files to predict deforestation events for the year 2012.

The input raster data is flattened and stacked into a single 2D array, X_flat. `NoData` values are removed from the input data (X_cleaned) and the target variable (y_cleaned) before splitting them into training and testing datasets.

The Random Forest Classifier is trained using the X_train and y_train datasets, and its performance is evaluated using cross-validation. The trained model is then used to predict deforestation events for the testing dataset (X_test). The model's performance is assessed using confusion matrices and classification reports for both the training and testing datasets.

Finally, the feature importances of the input variables (e.g., land use plans, tree cover) are calculated and visualized in a bar chart to understand the relative importance of each input variable in predicting deforestation events.

Import Libraries and Constants

```
import os
import re
import sys
import numpy as np
import pandas as pd
import tempfile
import shutil
import matplotlib.pyplot as plt
import rasterio
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint as sp_randint
```

```
# Get the current working directory
current_dir = os.path.abspath('')
```

```
# Search for the 'constants.py' file starting from the current directory and moving up the
project_root = current_dir
while not os.path.isfile(os.path.join(project_root, 'constants.py')):
    project_root = os.path.dirname(project_root)

# Add the project root to the Python path
sys.path.append(project_root)
```

```
from constants import SERVER_PATH, OUTPUT_PATH, MASKED_RASTERS_DIR
```

```
#output- update this for subsequent runs
output_folder = os.path.join(OUTPUT_PATH[0], 'basic_rf_model')
```

```
# Where files for machine learning model should be located
# Directory containing the raster files
rasters_dir = MASKED_RASTERS_DIR[0]
```

Create Stack

```
# helper function to read tiff files
def read_tiff_image(file_path):
    with rasterio.open(file_path) as src:
        return src.read(1)
```

```
# List of paths to the raster files excluding 'deforestation11_20_masked.tif'
feature_files = [os.path.join(rasters_dir, file_name) for file_name in os.listdir(rasters_

# Then you can use this list of raster_files to create feature_data_arrays and raster_data
feature_data_arrays = [read_tiff_image(file_path) for file_path in feature_files]
feature_data_flat = [data_array.flatten() for data_array in feature_data_arrays]

# Path to the y_file
y_file = os.path.join(rasters_dir, 'deforestation11_20_masked.tif')
```

```
feature_files
```

```
['/Users/romero61/../../capstone/pyforest/ml_data/output/masked_rasters/treecover_2010_maske
 '/Users/romero61/../../capstone/pyforest/ml_data/output/masked_rasters/lup_10_masked.tif']
```

```
feature_files
```

```
['/Users/romero61/../../capstone/pyforest/ml_data/output/masked_rasters/treecover_2010_maske
 '/Users/romero61/../../capstone/pyforest/ml_data/output/masked_rasters/lup_10_masked.tif']
```

```
# Find the dimensions of all the raster data arrays
raster_shapes = [raster_data.shape for raster_data in feature_data_arrays]

# Check if all raster data arrays have the same dimensions
if len(set(raster_shapes)) > 1:
    print("There are mismatching dimensions:")
    for file_path, raster_shape in zip(raster_files, raster_shapes):
        print(f"File: {file_path}, Shape: {raster_shape}")
else:
    print("All raster data arrays have the same dimensions.")
    # Check the dimensions of all the raster data arrays
    for i, data_array in enumerate(feature_data_arrays):
        print(f"Raster {i}: {data_array.shape}")
```

```
All raster data arrays have the same dimensions.
Raster 0: (22512, 20381)
Raster 1: (22512, 20381)
```

Stack and Flatten Data

```
# NoData Value
no_data_value = -1

# Stack the flattened raster data
X_flat = np.column_stack(feature_data_flat)

# Use the y_file obtained from the find_deforestation_file function
```

```
y = read_tiff_image(y_file).flatten()

# Remove rows with NoData values
'''checks each row in X_flat and creates a boolean array (valid_rows_X) that has the same
as the number of rows in X_flat. Each element in valid_rows_X is True if there is no NoDat
the corresponding row of X_flat and False otherwise.'''
valid_rows_X = ~(X_flat == no_data_value).any(axis=1)

'''checks each element in the y array and creates a boolean array (valid_rows_y) that has
elements as y. Each element in valid_rows_y is True if the corresponding element in y is n
equal to the NoData value and False otherwise.'''
valid_rows_y = y != no_data_value

'''checks each element in the y array and creates a boolean array (valid_rows_y)
that has the same number of elements as y. Each element in valid_rows_y is True if the cor
in y is not equal to the NoData value and False otherwise.'''
valid_rows = valid_rows_X & valid_rows_y

'''creates a new array X_cleaned by selecting only the rows in X_flat that
correspond to the True elements in valid_rows.'''
X_cleaned = X_flat[valid_rows]

'''creates a new array y_cleaned by selecting only the elements in y that correspond
to the True elements in valid_rows.'''
y_cleaned = y[valid_rows]
```

To ensure your data cleaning steps have been applied correctly, you can check the following:

NoData values have been removed: You should confirm that there are no NoData values in your cleaned data. This can be done by asserting that there are no occurrences of no_data_value in X_cleaned and y_cleaned.

```
assert not (X_cleaned == no_data_value).any()
assert not (y_cleaned == no_data_value).any()
```

These assertions will throw an error if there is a NoData value in X_cleaned or y_cleaned

Dimensions are correct: The shapes of X_cleaned and y_cleaned should match along the row dimension (the first dimension for 2D array X_cleaned and the only dimension for 1D array y_cleaned).

```
print("Shape of X_cleaned:", X_cleaned.shape)
print("Shape of y_cleaned:", y_cleaned.shape)
```

```
Shape of X_cleaned: (48181572, 2)
Shape of y_cleaned: (48181572,)
```

Make sure the number of rows in X_cleaned equals the number of elements in y_cleaned.

Confirm that the valid rows have been correctly identified: You can do this by checking that the number of True values in valid_rows equals the number of rows in X_cleaned (or the number of elements in y_cleaned).

```
assert valid_rows.sum() == X_cleaned.shape[0]
```

Split the data into training and testing sets

```
X_train, X_test, y_train, y_test = train_test_split(X_cleaned, y_cleaned, test_size=0.2, r
```

Class Balance Check

```
# Create pandas Series from your labels
y_train_series = pd.Series(y_train)
y_test_series = pd.Series(y_test)
y_cleaned_series = pd.Series(y_cleaned)

# Print balance of classes in training, testing, and whole dataset
print("Training data balance:\n", y_train_series.value_counts(normalize=True))
print("Testing data balance:\n", y_test_series.value_counts(normalize=True))
print("Whole dataset balance:\n", y_cleaned_series.value_counts(normalize=True))
```

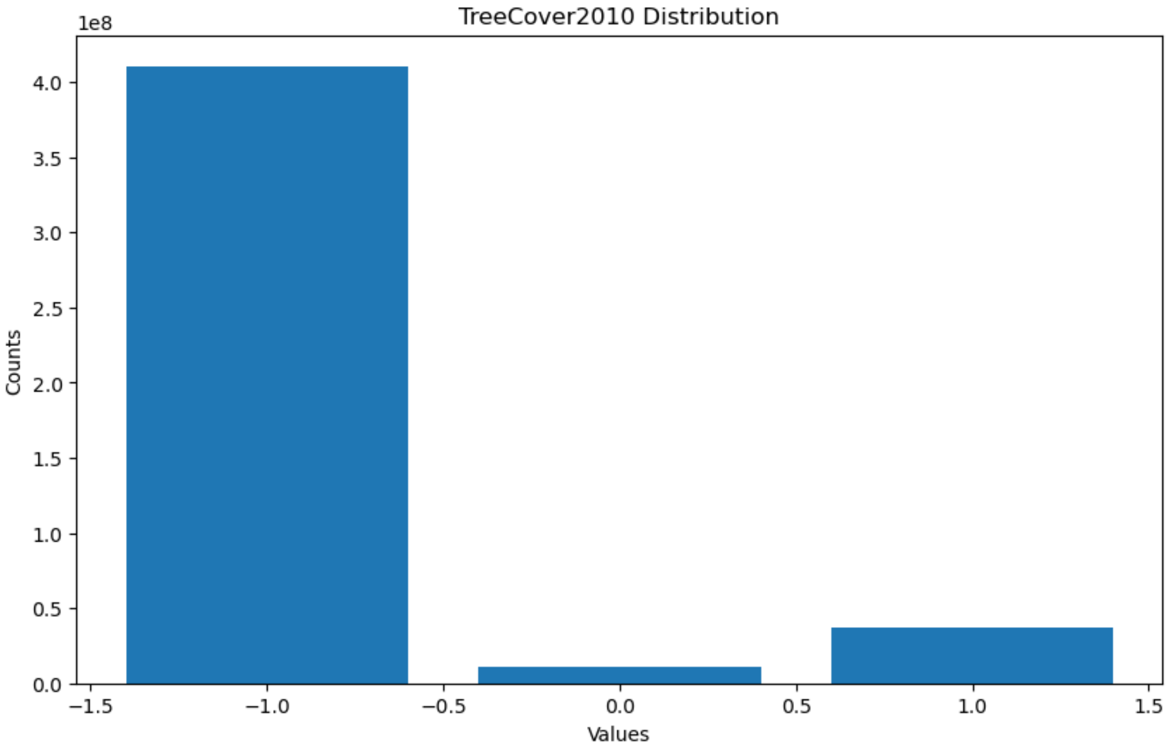
```
Training data balance:
 0    0.838179
1    0.161821
dtype: float64
Testing data balance:
 0    0.838283
```

```
1    0.161717
dtype: float64
Whole dataset balance:
 0    0.8382
1    0.1618
dtype: float64
```

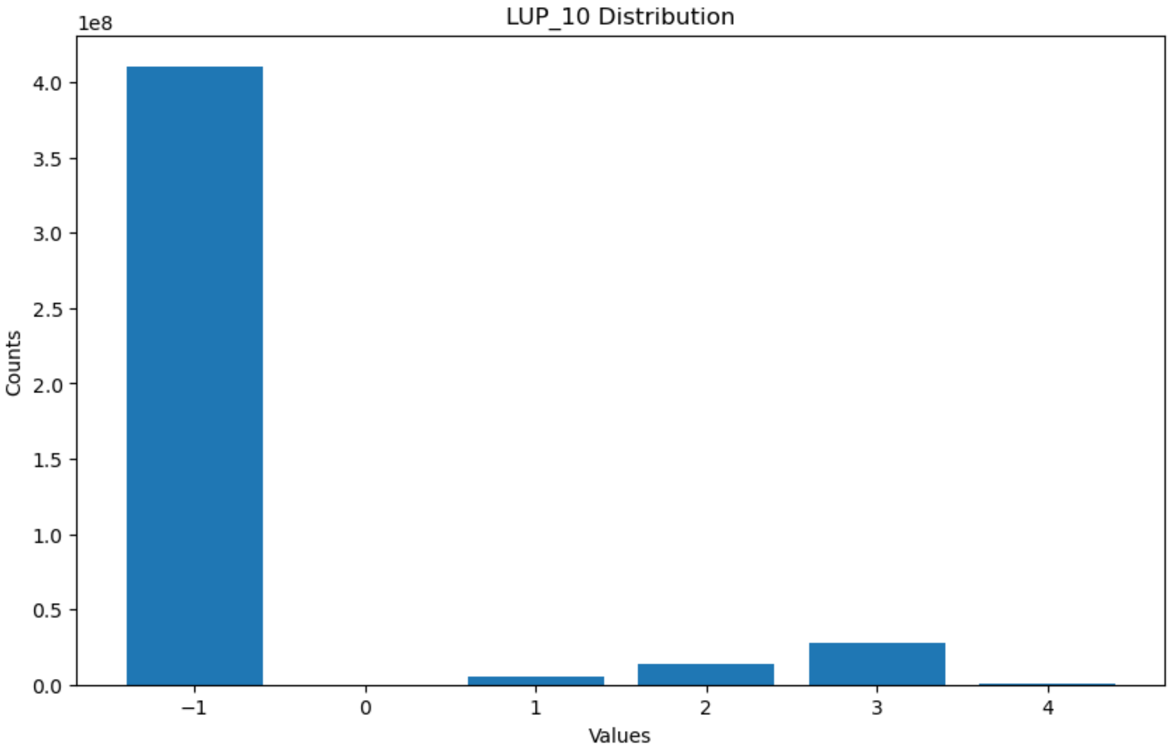
The balance of your dataset seems to be roughly the same in both the training and testing sets, with about 83.8% of the instances belonging to class 0 (no deforestation) and 16.2% to class 1 (deforestation). This shows that the classes are quite imbalanced. Machine learning algorithms, including Random Forest, may have a bias towards the majority class in such situations, which could be one of the reasons why your model is not performing well on the minority class.

```
# Create a list to hold your feature file paths


# Define the labels for your features
feature_labels = ['TreeCover2010', 'LUP_10']

# Loop through your feature files
for feature_file, label in zip(feature_files, feature_labels):
    # Open the raster file
    with rasterio.open(feature_file) as src:
        # Read the raster data into a 2D array
        feature_data = src.read(1)

    # Calculate unique values and their counts
    unique_values, counts = np.unique(feature_data, return_counts=True)

    # Print the counts for each unique value
    for value, count in zip(unique_values, counts):
        print(f'{label} Value: {value}, Count: {count}')

    print('------------------')

    # Plot histogram
    plt.figure(figsize=(10, 6))
    plt.bar(unique_values, counts)
    plt.title(f'{label} Distribution')
    plt.xlabel('Values')
    plt.ylabel('Counts')
    plt.show()


for i, feature in enumerate(feature_labels):
    unique_values, counts = np.unique(X_cleaned[:, i], return_counts=True)

    # Print the counts for each unique value
    for value, count in zip(unique_values, counts):
        print(f'{feature} Value: {value}, Count: {count}')

    print('-----------------')

    # Plot histogram
    plt.figure(figsize=(10, 6))
    plt.hist(X_cleaned[:, i], bins=20)
    plt.title(f'{feature} Distribution')
    plt.xlabel('Values')
    plt.ylabel('Counts')
    plt.show()
```

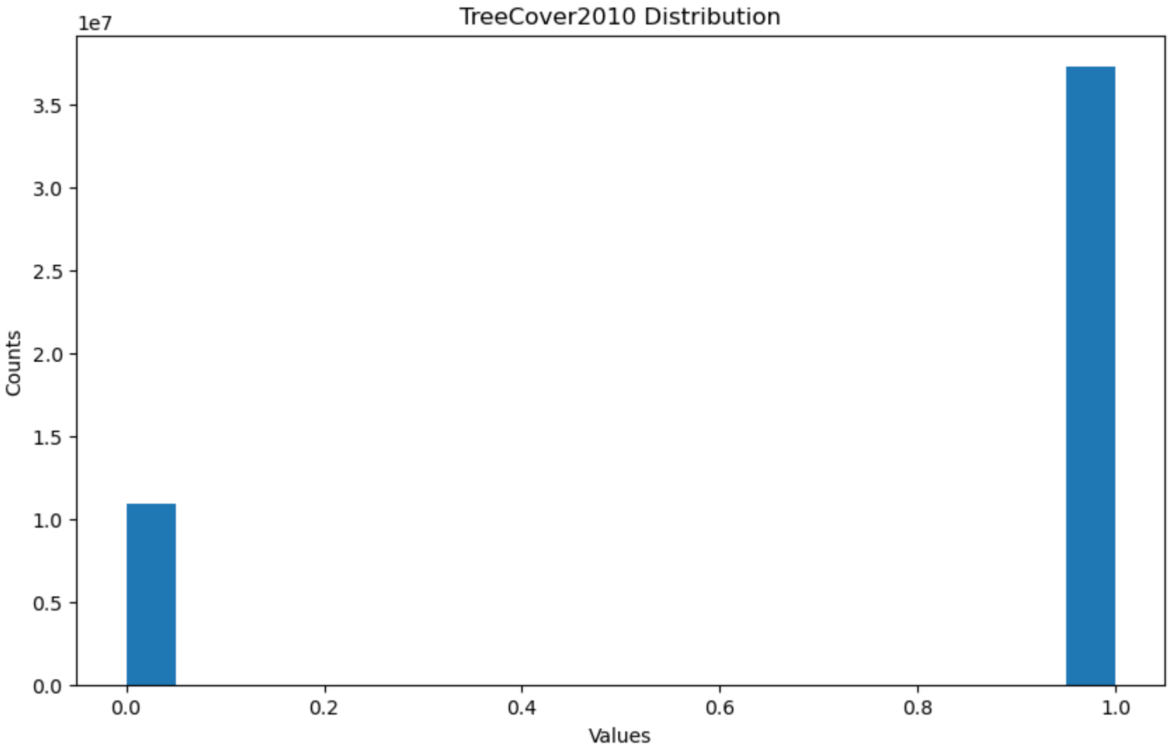
```
TreeCover2010 Value: -1, Count: 410635500
TreeCover2010 Value: 0, Count: 10878781
TreeCover2010 Value: 1, Count: 37302791
-----------------
```



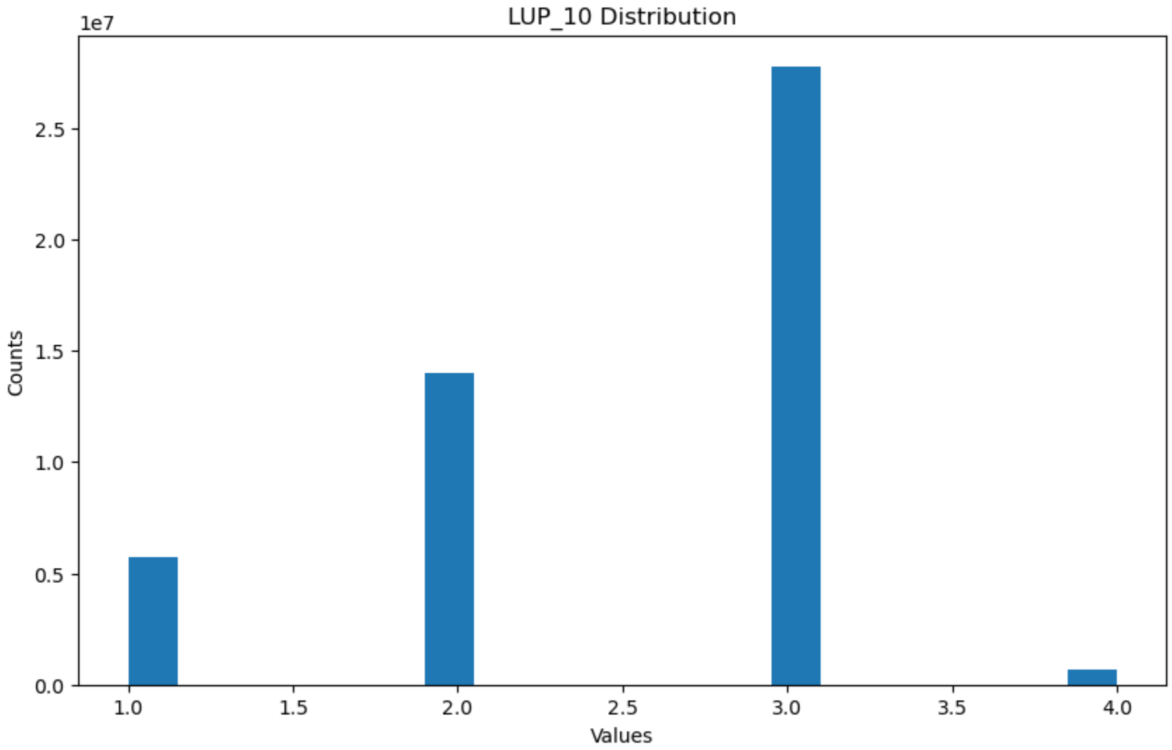
```
LUP_10 Value: -1, Count: 410635500
LUP_10 Value: 1, Count: 5754071
LUP_10 Value: 2, Count: 13967435
LUP_10 Value: 3, Count: 27775793
LUP_10 Value: 4, Count: 684273
-----------------
```



```
TreeCover2010 Value: 0, Count: 10878781
TreeCover2010 Value: 1, Count: 37302791
-----------------
```



```
LUP_10 Value: 1, Count: 5754071
LUP_10 Value: 2, Count: 13967435
LUP_10 Value: 3, Count: 27775793
LUP_10 Value: 4, Count: 684273
-----------------
```



Simple Grid Search for a Random Forest model:

```
# Create a RandomForestClassifier instance
rfc = RandomForestClassifier(random_state=42, class_weight= 'balanced')

# Define a basic parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],   # number of trees in the forest
    'max_depth': [None, 5, 10, 20]    # maximum depth of the tree
}

# Instantiate GridSearchCV
grid_search = GridSearchCV(rfc, param_grid, cv=5, n_jobs=30)
```

Info on CV, fit, predict, predict_proba

Cross-validation is a technique used to evaluate the performance of a machine learning model by training and testing it on different subsets of the dataset. It helps assess how well the model generalizes to new, unseen data and helps mitigate the risk of overfitting.

Here's how the 5-fold cross-validation works:

1. The entire dataset (X_cleaned and y_cleaned) is divided into 5 equally sized (or nearly equal) folds.
2. The model is trained and tested 5 times. In each iteration, one of the folds is used as the test set, and the remaining 4 folds are used to train the model.
3. For each iteration, the model's performance is evaluated using a chosen evaluation metric (in this case, accuracy, which is the default scoring method for cross_val_score).
4. Once all 5 iterations are completed, the performance scores are averaged to give a single cross-validation score.

By using cross-validation, you get a more reliable estimate of the model's performance because it's tested on different portions of the dataset. This helps to reduce the risk of overfitting and gives you a better understanding of how well your model generalizes to unseen data.

Cross-validation is performed before `clf.fit` to assess the performance of the model on the data without using the same data for both training and validation. It helps to understand how well the model is likely to generalize to new, unseen data before committing to training the final model.

If the cross-validation scores are satisfactory, you can proceed to train the final model using the entire dataset with clf.fit.

`clf.fit` is the method used to train the machine learning model on the provided dataset. In this case, it's training the Random Forest Classifier (denoted as `clf`) on the training dataset (`X_train` and `y_train`). The purpose of `clf.fit` is to learn the relationship between the input features (`X_train`) and the target variable (`y_train`) so that the model can make predictions on new, unseen data.

`clf.predict` is the method used to make predictions using the trained model. Once the model is trained with `clf.fit`, it can then be used to predict the target variable for new input features.

The model is predicting probabilities. The RandomForestClassifier, by default, outputs probabilities of class membership. It provides the probability of each pixel belonging to the deforested or non-deforested class. However, when you use `clf.predict()`, it returns the class with the highest probability, which is a **binary result (deforested or non-deforested).**

The **`clf.predict_proba()`** function obtains probabilities instead of the binary result returning probabilities of each class. `y_proba = clf.predict_proba(X_cleaned)[:, 1]`extracts the probabilities of deforestation events (class 1) for all pixels.

```
# Fit to the training data
grid_search.fit(X_train, y_train)
```

```
GridSearchCV(cv=5,
             estimator=RandomForestClassifier(class_weight='balanced',
                                              random_state=42),
             n_jobs=30,
             param_grid={'max_depth': [None, 5, 10, 20],
                         'n_estimators': [50, 100, 200]})
```

Examine Fit Results

grid_search.best_params_ contains the hyperparameter combination that resulted in the highest average cross-validation score across the different folds during the grid search. This is useful information as it tells you which hyperparameters worked best for your model and data.

grid_search.best_score_ is the highest mean cross-validation score achieved by the best hyperparameter combination found in the grid search. It gives you an idea of the model's performance with the optimal hyperparameters during the cross-validation process.

Best estimator: This provides the best estimator found by grid search. This is already fitted to the data and can be used for making predictions or further analysis.

CV Results: This provides a dictionary with various details about the grid search, like scores for each combination of parameters, time taken for fitting and scoring, etc. Note: cv_results_ is a dictionary and can be quite verbose, you may want to convert it into a DataFrame for easier viewing.

Scorer: This provides the scoring function used by grid search.

Refit Time: This gives the time taken to refit the best estimator with the entire dataset.

```
# Print all available attributes and methods for the random_search object
all_attributes_methods = dir(grid_search)

# Filter out attributes and methods inherited from BaseSearchCV
specific_attributes_methods = [
    attribute for attribute in all_attributes_methods
    if attribute not in dir(GridSearchCV)
]

print("Attributes and methods specific to GridSearchCV:")
for attr in specific_attributes_methods:
    print(attr)
```

```
Attributes and methods specific to GridSearchCV:
best_estimator_
best_index_
best_params_
best_score_
cv
cv_results_
error_score
estimator
multimetric_
n_jobs
n_splits_
param_grid
pre_dispatch
refit
refit_time_
return_train_score
scorer_
scoring
verbose
```

```
grid_search.score
```

```
<bound method BaseSearchCV.score of GridSearchCV(cv=5,
             estimator=RandomForestClassifier(class_weight='balanced',
                                              random_state=42),
             n_jobs=30,
             param_grid={'max_depth': [None, 5, 10, 20],
                         'n_estimators': [50, 100, 200]})>
```

```
# Get the best parameters and the corresponding score
```

```
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best parameters:", best_params)
print("Best cross-validation score:", best_score)
```

```
Best parameters: {'max_depth': None, 'n_estimators': 50}
Best cross-validation score: 0.5858522619204243
```

After fitting the GridSearchCV, you can evaluate the performance of the best model on the test data (X_test and y_test) using the best_estimator_ attribute of the grid_search object:

```
best_estimator = grid_search.best_estimator_

cv_results = grid_search.cv_results_

cv_results_df = pd.DataFrame(grid_search.cv_results_)

scorer = grid_search.scorer_

refit_time = grid_search.refit_time_

print("Best estimator:", best_estimator)
print("CV Results:",cv_results_df)
print("Scorer function:", scorer)
print("Refit time (seconds):", refit_time)
```

```
Best estimator: RandomForestClassifier(class_weight='balanced', n_estimators=50,
                       random_state=42)
CV Results:     mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0      546.787241     58.240538        21.566601        3.236626
1     1017.598757     75.864049        46.208107        9.260437
2     1918.190541    115.549766        73.645717        1.699839
3      474.310703      9.245638        21.489198        3.174777
4     1070.831887     49.192505        53.257504        7.381053
5     1934.186380     90.319932        73.316022        1.469383
6      466.367223     42.144907        22.921606        3.658064
7     1021.736046     90.103013        45.873498        6.964352
8     1686.582774     28.455681        70.935093        0.173656
```

```
9      534.640818     40.057919        24.653716        4.052579
10     904.753022     26.085620        36.163089        0.194958
11    1552.772714     32.792583        69.884495        1.043727

   param_max_depth param_n_estimators  \
0             None                 50
1             None                100
2             None                200
3                5                 50
4                5                100
5                5                200
6               10                 50
7               10                100
8               10                200
9               20                 50
10              20                100
11              20                200

                                      params  split0_test_score  \
0    {'max_depth': None, 'n_estimators': 50}           0.585995
1   {'max_depth': None, 'n_estimators': 100}           0.585995
2   {'max_depth': None, 'n_estimators': 200}           0.585995
3       {'max_depth': 5, 'n_estimators': 50}           0.585995
4      {'max_depth': 5, 'n_estimators': 100}           0.585995
5      {'max_depth': 5, 'n_estimators': 200}           0.585995
6      {'max_depth': 10, 'n_estimators': 50}           0.585995
7     {'max_depth': 10, 'n_estimators': 100}           0.585995
8     {'max_depth': 10, 'n_estimators': 200}           0.585995
9      {'max_depth': 20, 'n_estimators': 50}           0.585995
10    {'max_depth': 20, 'n_estimators': 100}           0.585995
11    {'max_depth': 20, 'n_estimators': 200}           0.585995

    split1_test_score  split2_test_score  split3_test_score  \
0            0.585956           0.586016           0.585466
1            0.585956           0.586016           0.585466
2            0.585956           0.586016           0.585466
3            0.585956           0.586016           0.585466
4            0.585956           0.586016           0.585466
5            0.585956           0.586016           0.585466
6            0.585956           0.586016           0.585466
7            0.585956           0.586016           0.585466
8            0.585956           0.586016           0.585466
9            0.585956           0.586016           0.585466
```

```
10           0.585956           0.586016           0.585466
11           0.585956           0.586016           0.585466

    split4_test_score  mean_test_score  std_test_score  rank_test_score
0            0.585829         0.585852        0.000204                1
1            0.585829         0.585852        0.000204                1
2            0.585829         0.585852        0.000204                1
3            0.585829         0.585852        0.000204                1
4            0.585829         0.585852        0.000204                1
5            0.585829         0.585852        0.000204                1
6            0.585829         0.585852        0.000204                1
7            0.585829         0.585852        0.000204                1
8            0.585829         0.585852        0.000204                1
9            0.585829         0.585852        0.000204                1
10           0.585829         0.585852        0.000204                1
11           0.585829         0.585852        0.000204                1
Scorer function: <function _passthrough_scorer at 0x7f2b78a31300>
Refit time (seconds): 459.01079297065735
```

Evaluate the model performance using your preferred metrics

e.g., confusion matrix, classification report, accuracy, F1-score, etc.

```python
best_model = grid_search.best_estimator_
```

```python
# Predictions for test data
y_pred = best_model.predict(X_test)
```

Evaluate the performance of your model by comparing the predicted labels (y_pred) with the true labels (y_test). You can use various metrics such as confusion matrix, classification report, accuracy, F1-score, etc.:

```python
from sklearn.metrics import accuracy_score, f1_score, classification_report

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Calculate F1-score (use 'weighted' or 'macro' depending on your problem)
f1 = f1_score(y_test, y_pred, average='weighted')
```

```
print("F1-score:", f1)

# Print classification report
report = classification_report(y_test, y_pred)
print("Classification report:\n", report)
```

```
Accuracy: 0.5859063345272545
F1-score: 0.6391117670833032
Classification report:
               precision    recall  f1-score   support

           0       0.93      0.55      0.69   8077959
           1       0.25      0.79      0.38   1558356

    accuracy                           0.59   9636315
   macro avg       0.59      0.67      0.53   9636315
weighted avg       0.82      0.59      0.64   9636315
```

Confusion Matrix

```
# Predictions for train data
y_pred_train = best_model.predict(X_train)
```

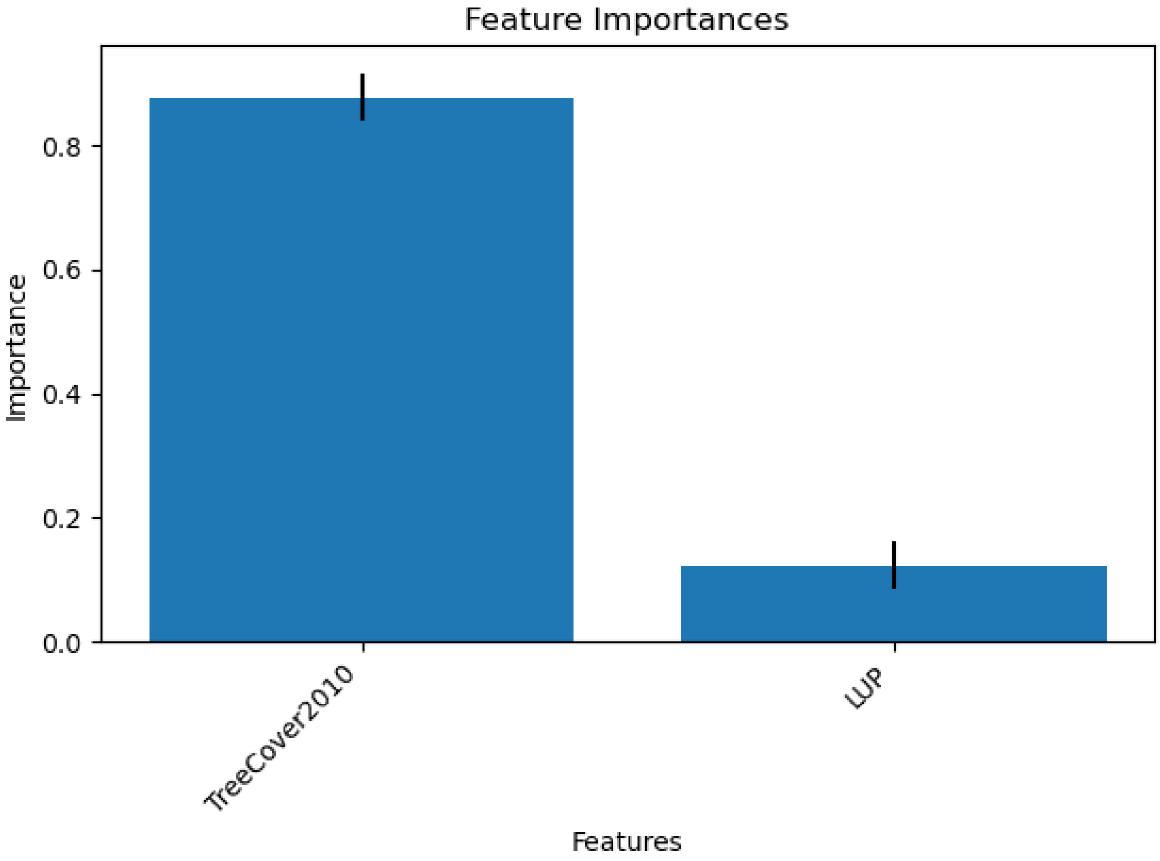
```
# Confusion matrix and classification report for train data
train_cm = confusion_matrix(y_train, y_pred_train)
train_cr = classification_report(y_train, y_pred_train)
print("Training confusion matrix:")
print(train_cm)
print("Training classification report:")
print(train_cr)
```

```
Training confusion matrix:
[[17670601 14637219]
 [ 1326212  4911225]]
Training classification report:
              precision    recall  f1-score   support

           0       0.93      0.55      0.69  32307820
           1       0.25      0.79      0.38   6237437
```

```
    accuracy                           0.59  38545257
   macro avg       0.59      0.67      0.53  38545257
weighted avg       0.82      0.59      0.64  38545257
```

```
# Calculate feature importances and the standard deviation for those importances
importances = best_model.feature_importances_
std = np.std([tree.feature_importances_ for tree in best_model.estimators_], axis=0)


 # list of feature names corresponding to the input bands of your raster stack
feature_names = ['TreeCover2010','LUP']
# Create a sorted list of tuples containing feature names and their importances:
sorted_features = sorted(zip(feature_names, importances, std), key=lambda x: x[1], reverse

# Create a bar chart
fig, ax = plt.subplots()

# Set the feature names as x-axis labels
ax.set_xticklabels([item[0] for item in sorted_features], rotation=45, ha='right')
ax.set_xticks(range(len(sorted_features)))

# Set the y-axis labels as importances
ax.bar(range(len(sorted_features)), [item[1] for item in sorted_features], yerr=[item[2] f

# Set the title and labels for the chart
ax.set_title('Feature Importances')
ax.set_xlabel('Features')
ax.set_ylabel('Importance')

# Display the chart
plt.tight_layout()
plt.show()
```

```
/tmp/ipykernel_3700312/2283730908.py:15: UserWarning: FixedFormatter should only be used toge
  ax.set_xticklabels([item[0] for item in sorted_features], rotation=45, ha='right')
```



Probabilities for deforestation

```
# Predict probabilities for deforestation events
y_proba = best_model.predict_proba(X_cleaned)[:, 1]
```

```
# Predicts the
# Create a probability raster by filling in the valid pixel values
prob_raster = np.full(y.shape, no_data_value, dtype=np.float32)
prob_raster[valid_rows] = y_proba
prob_raster = prob_raster.reshape(feature_data_arrays[0].shape)
```

```
# Save the probability raster as a GeoTIFF file
if not os.path.exists(output_folder):
    os.makedirs(output_folder)
```

```
output_file = os.path.join(output_folder, "deforestation_prob.tiff")

with rasterio.open(y_file) as src:
    profile = src.profile
    profile.update(dtype=rasterio.float32, count=1)

prob_raster_reshaped = prob_raster.reshape((1, prob_raster.shape[0], prob_raster.shape[1])

with rasterio.open(output_file, 'w', **profile) as dst:
    dst.write_band(1, prob_raster_reshaped[0])
```

Tuning Strategies

```
# Complex Grid
'''# Set the range of values for each hyperparameter
param_grid = {
    'n_estimators': [100, 200],
    'criterion': ['gini', 'entropy'],
    'max_depth': [10, 20, 30, 40],
    'min_samples_split': [ 5, 10, 20],
    'min_samples_leaf': [2, 5, 10],
    'max_features': ['sqrt', None],
    'bootstrap': [True],
    'class_weight': ['balanced']
}

# Instantiate the RandomForestClassifier
clf = RandomForestClassifier(random_state=0)

# Set up the GridSearchCV
grid_search = GridSearchCV(clf, param_grid=param_grid, cv=5, n_jobs=19)'''
```

```
# Randomized Search
# Set the range of values for each hyperparameter
'''param_dist = {
    "n_estimators": sp_randint(100, 300),
    'criterion': ['gini',],
    'max_features': ['sqrt', None],
    "max_depth": sp_randint(1, 20),
    "min_samples_split": sp_randint(2, 11),
    "min_samples_leaf": sp_randint(1, 11),
    "bootstrap": [True],
    'class_weight': ['balanced']
}

# Instantiate the RandomForestClassifier
clf = RandomForestClassifier(random_state=0)

# Set up the RandomizedSearchCV
random_search = RandomizedSearchCV(
    clf, param_distributions=param_dist, n_iter=20, cv=5, random_state=0, n_jobs=19
)'''
```