Type of the Paper: Article – Appendices for Supplementary Materials

Predicting Persistent Forest Fire Refugia Using Machine Learning Models with Topographic, Microclimate, and Surface Wind Variables

Sven Christ[1], Tineke Kraaij[2], Coert J. Geldenhuys[3], Helen M. de Klerk[1, 4] *

[1] Department of Geography and Environmental Studies, Stellenbosch University, P.Bag X1, Stellenbosch, 7602 South Africa

[2] Natural Resource Science and Management, Science Faculty, Nelson Mandela University, George Campus, South Africa; tineke.kraaij@mandela.ac.za

[3] Forest Science Program, Department of Plant and Soil Sciences, University of Pretoria. c/o Forestwood cc, 35 Grace Avenue, Murrayfield, 0184, Pretoria, South Africa; cgelden@mweb.co.za

[4] Centre for Geospatial and Computing Technologies, Faculty of Environment, Society and Design | Te Wāhaka ki te Taiao, te Hāpori Whānui me kā Mahi Hoahoa, Lincoln University, Forbes Building, Ellesmere Junction Road, Lincoln, 7608, New Zealand

* Correspondence: helen.deklerk@lincoln.ac.nz

Appendix S1 . Patterns based validation

Due to the lack of wind stations to validate wind speed and direction at specific points, a wind pattern based analysis was done. The table below indicates where samples were taken, the patterns using stream tracers and the literature that indicates these types of patterns should occur at these features.

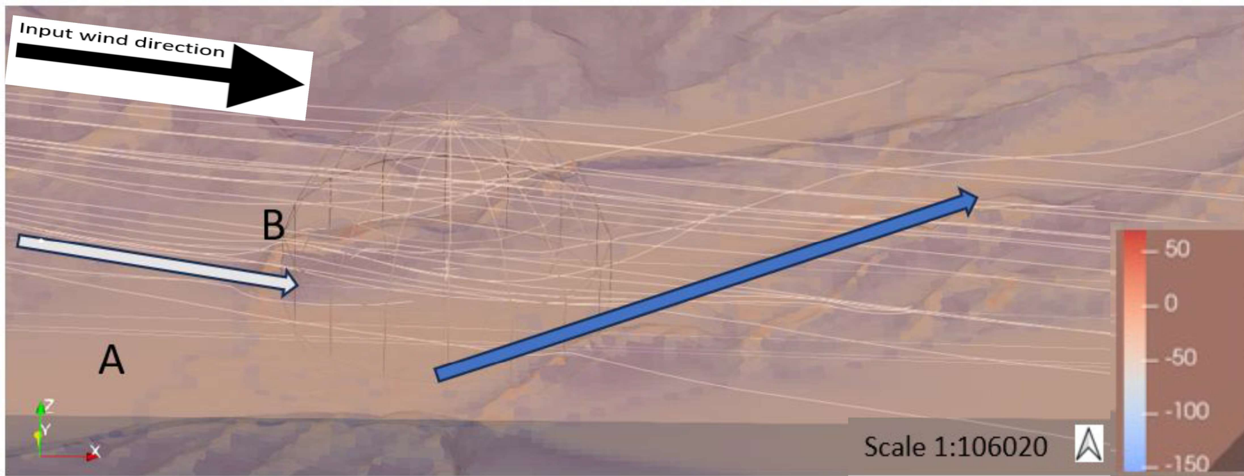

Figure 1. Flow channeling around topographic features (white arrow) and along a valley (blue arrow) starting from Swartvlei, Sedgefield (A) and Perdespruit River Mouth (B) [47].

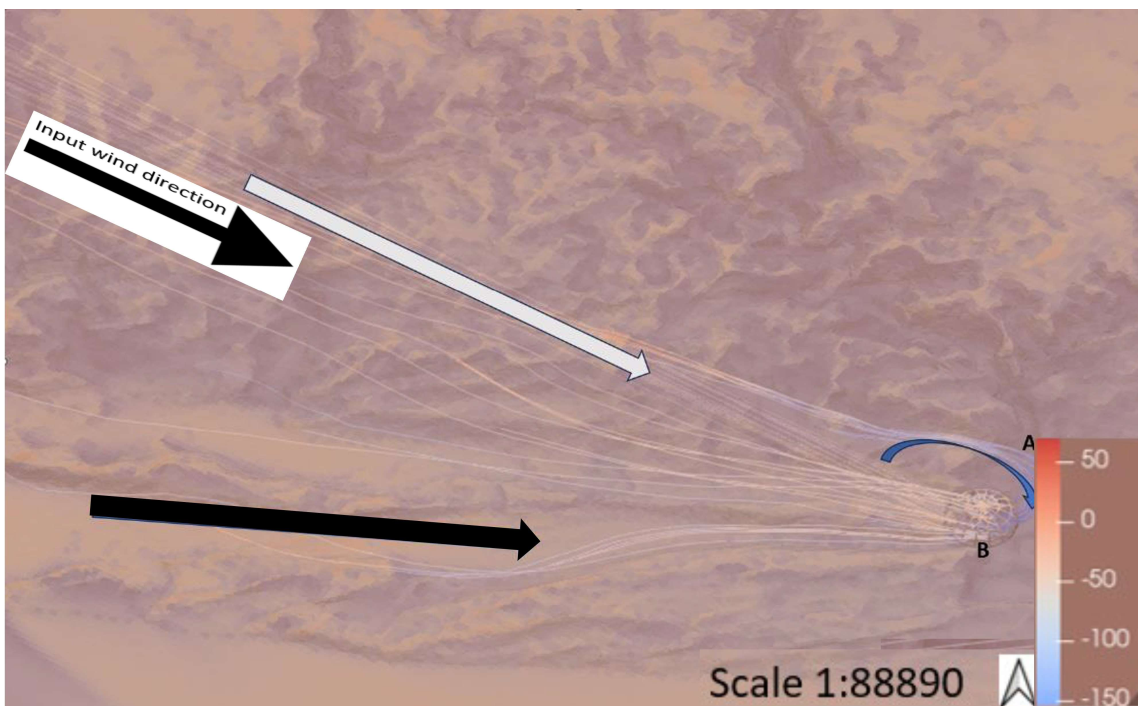

Figure 2. Flow channeling along a valley (black arrow) and around hill (blue arrow) towards the Goukamma and N2 crossing (A). The white arrow shows flow at higher altitudes over the top of the mountains to reach point B. These patterns are described in [47]

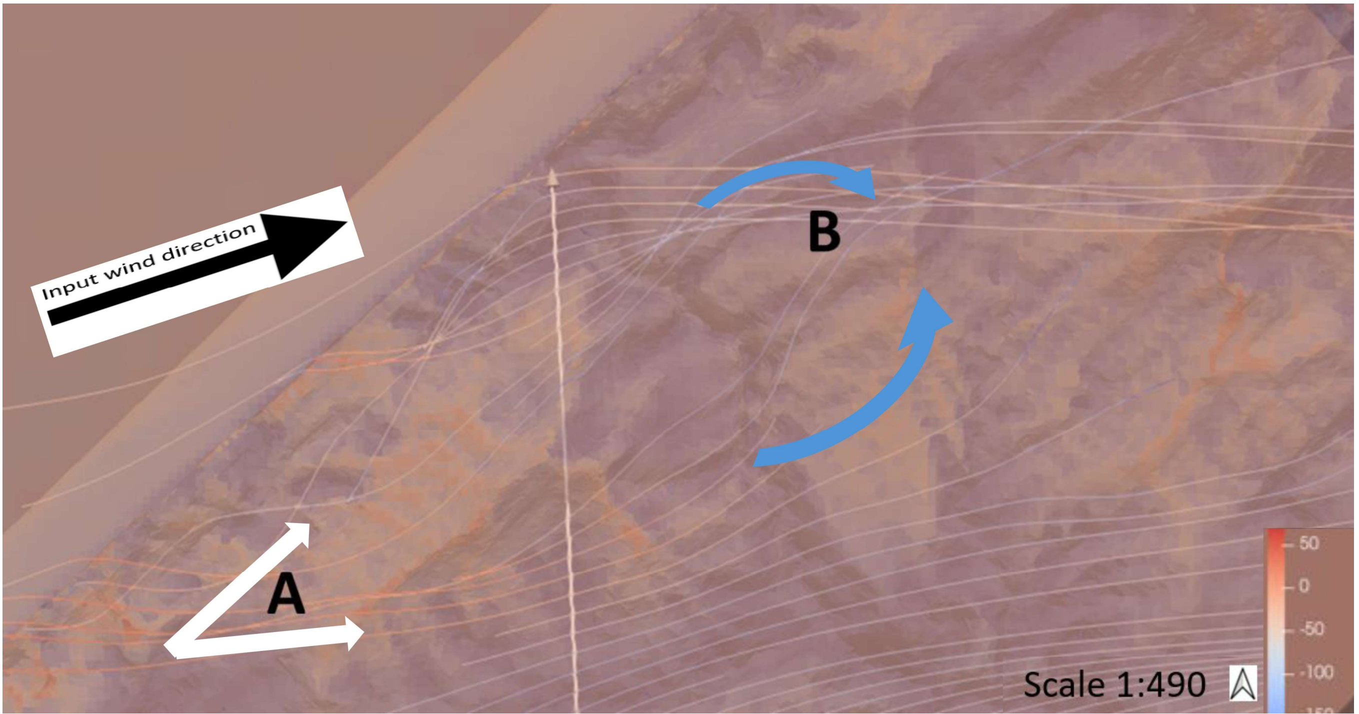

Figure 3. Aerial view of surface wind flow separation (white arrows) around a topographic feature, namely the mountain peak at Millwood hut (A) and convergences (blue arrows) a short distance after that (B) [47].

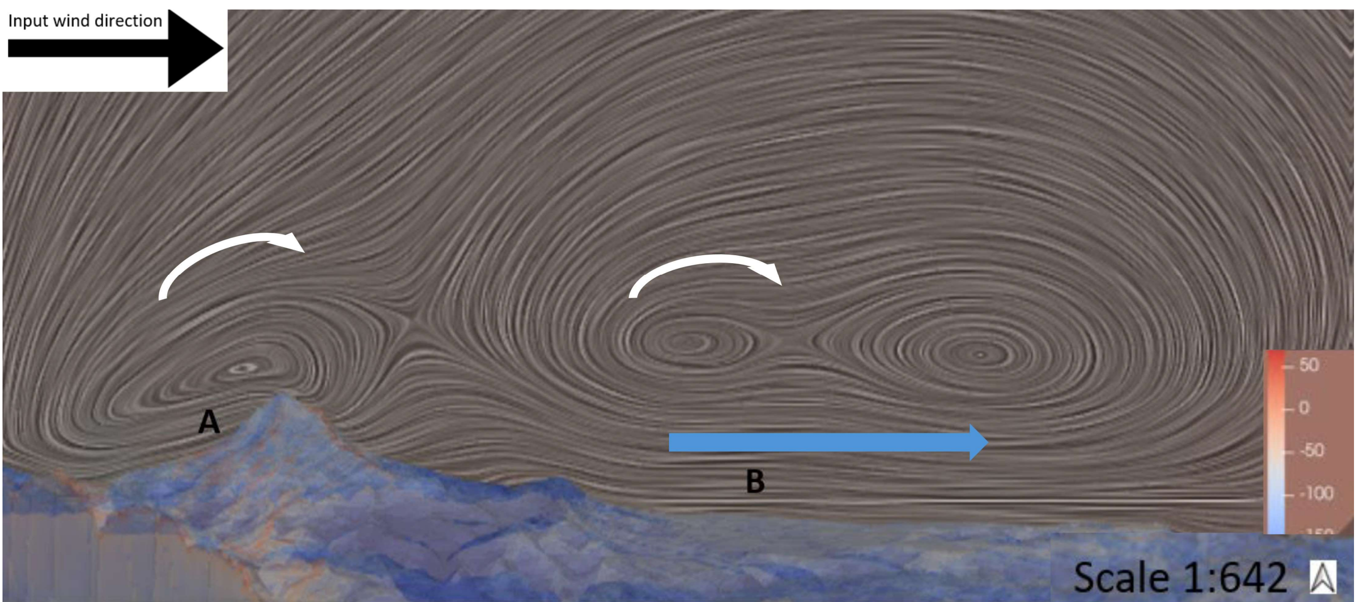

Figure 4. Profile view of flow separation as wind is approaching the mountain peak behind Millwood hut (A) with eddies on the lee side of Millwood hut (white arrow), as well as the laminar flow low to the ground (blue arrow) below the eddies (B) These patterns are described in [5]

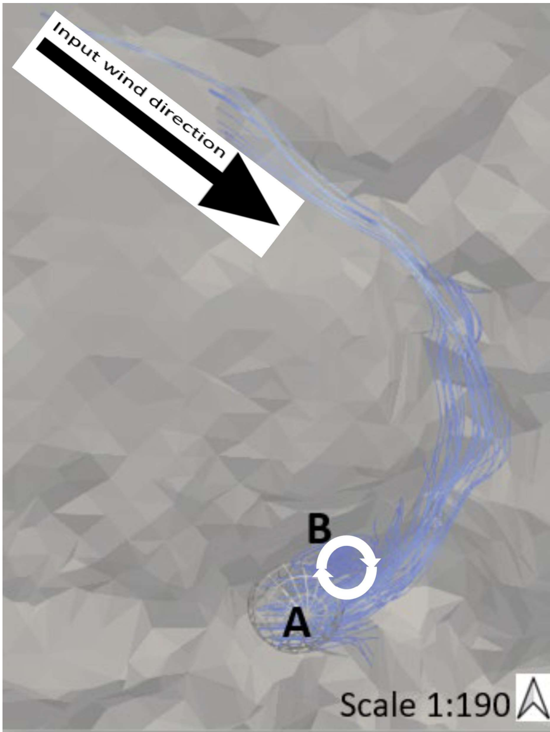

Figure 5. Surface wind flow moves around the base of the Elandsraal/Kooigoed hill (A) and the air becomes turbulent at point B forming an eddie (white circular arrows). This observation is described in [48].

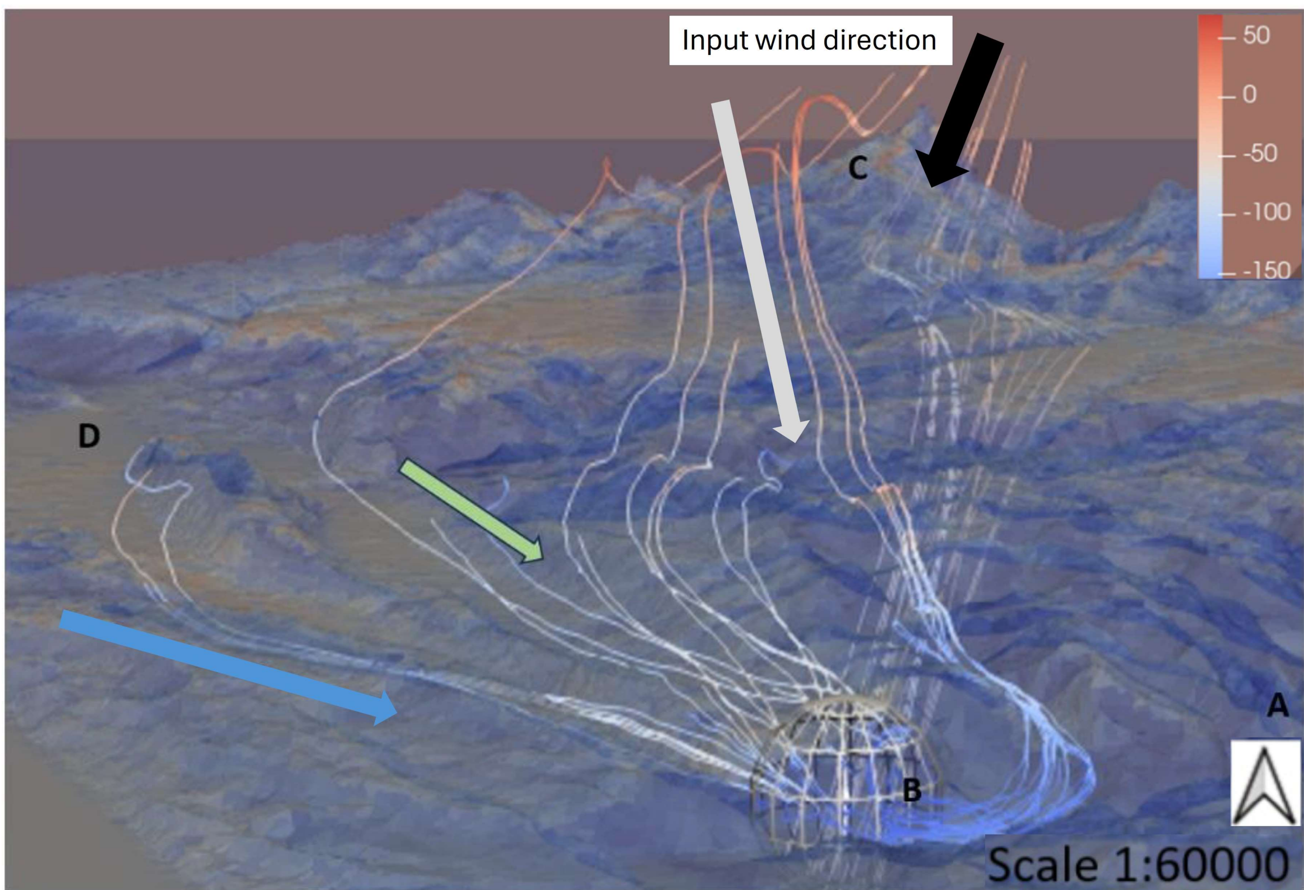

Figure 6. Oblique view of wind channeling of surface wind flow along valleys (green and blue arrow shows) from Swartvlei (D) to Sedgefield (B). The grey arrow shows how surface wind moves over mountains and hills, while skipping areas in the lee of the wind direction. These patterns are described in [5]

Appendix S2. Fire model pattern based validation

For the 2017 Knysna fire, [48] produced a detailed report that included a FARSITE fire spread simulation. This report was used as a reference to construct a comparable FARSITE simulation in the present study, replicating all model parameters described by [48]. Three model configurations were tested: (i) the standard model following [48] (termed standard model), (ii) the standard model with WindNinja activated to simulate wind flow (termed WindNinja-enabled model), and (iii) a modified version of the standard model incorporating the CFD-derived wind data layer (termed CFD-based model). To evaluate model performance, the simulated fire arrival time at the Kooigoed weather station (33° 57′ 46.1″ S, 22° 51′ 18.0″ E) was compared against the observed fire arrival time at this location. A visual comparison of the model outputs (Figure 7) shows that all simulations captured the main extent of the manually mapped fire scar (shown in black). The similarity between the standard model (Figure 7a) and the WindNinja-enabled model (Figure 7b) indicates minimal improvement from the inclusion of the WindNinja modelling of wind, consistent with findings by {Blanchard, 2015 #200}. The CFD-based model (Figure 7c) most accurately replicated the manually mapped fire scar, exhibiting a more south-easterly spread compared to the predominantly easterly trend observed in the first two models. Additionally, the CFD-based model reflected fire behavior patterns consistent with published literature with preferential movement along the hillside rather than downslope, where spread is typically slower {Sullivan, 2014 #201}.

To further assess model performance, five point locations were identified (Figure 8) and the arrival of the modelled fire arrival time at these points compared (Table 1). [48] manually determined the fire arrival time at the Kooigoed station as 45 minutes after the fire ignition, based on imagery from a motion-activated camera. Only the CFD-based model produced results within 10 % of the observed value (4 minutes later), whereas the other models substantially underestimated arrival time (Table 1). This consistency, supported by the spatial pattern analysis presented in Appendix S1 Figure 1 to Figure 6, strengthens confidence in the CFD-based model's capacity to represent real wind dynamics during wildfire events.

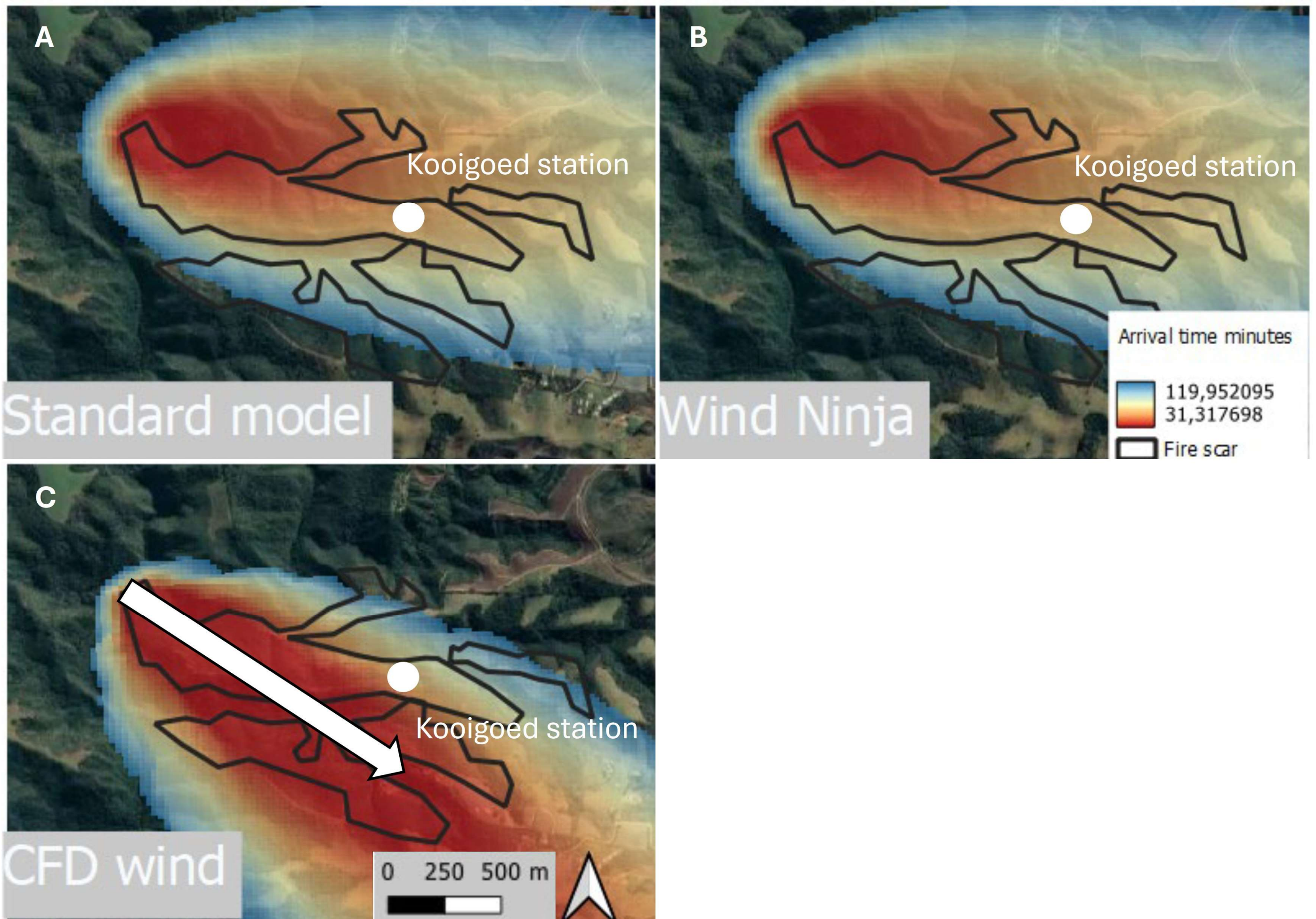

Figure 7. Comparison for standard model created in FARSITE with a single wind source (a), standard model with wind modelled by the WindNinja-enabled model (b), and CFD-based model. The manually digitized fire scar is mapped by the black line. All models are run to the Kooigoed station. The white arrow shows the south-easterly fire spread of the CFD-based model and emphasizes how this model output follows the mountain ridges.

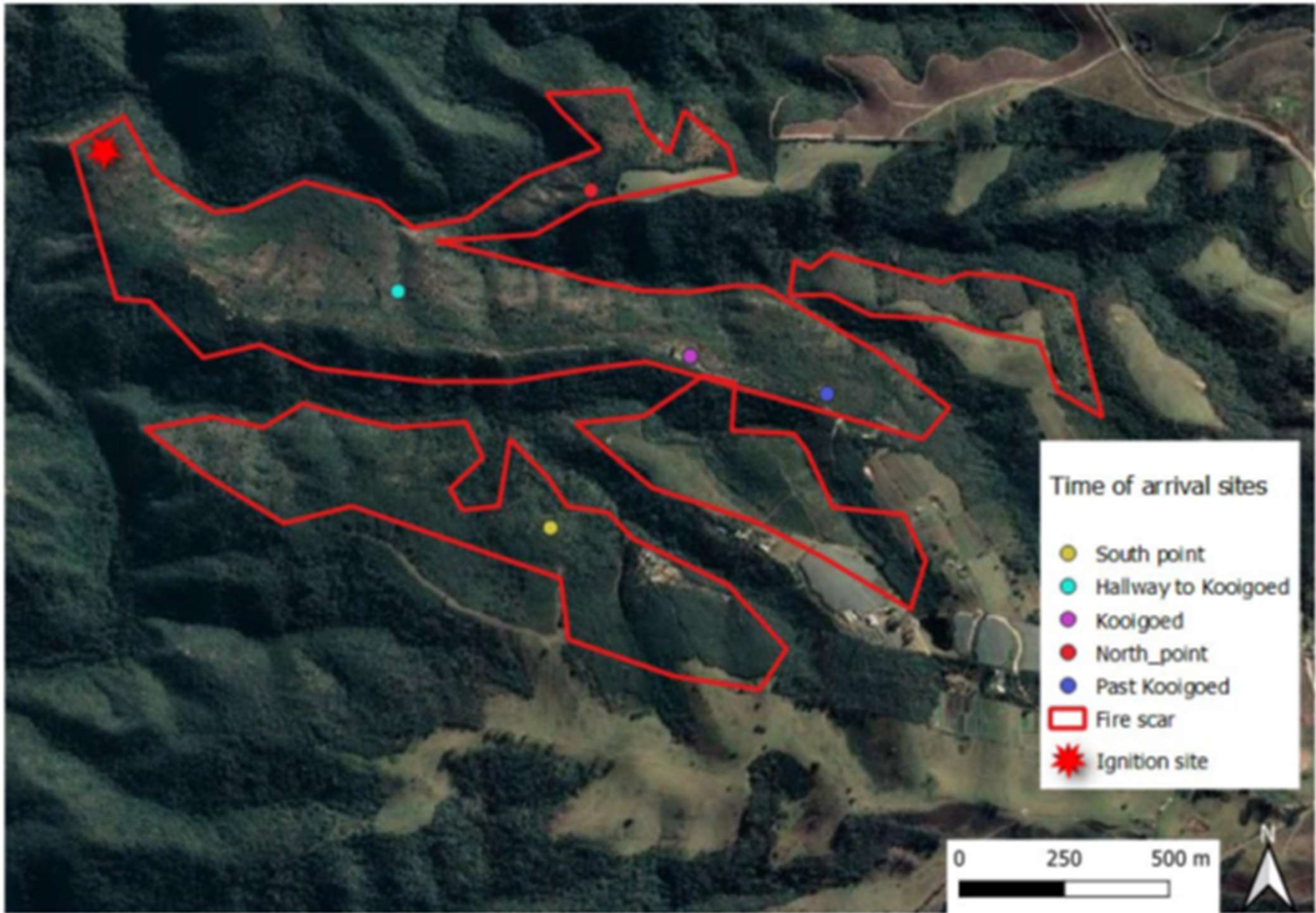

Table 1: Arrival time (in minutes from the start of the fire) of modelled fire at different locations. Note the measured time of fire arrival at Kooigoed was 45 minutes [48]

Location	Standard model (minutes)	WindNinja-enabled model (minutes)	CFD-based model(minutes)
Halfway to Kooigoed	47	49	24
Kooigoed	62	66	49
Past Kooigoed	69	73	66
North Point	48	50	83
South Point	110	114	26
Average (std deviation)	67.2 (21.0)	70.4 (21.6)	49.6 (20.8)

Appendix S3. PCA

```
import geopandas as gpd #PCA small area cleaned
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.impute import SimpleImputer

# Load the shapefiles
data = gpd.read_file(shapefile with cetroid values taken of all features for all areas identified as refugia or non refugia)
print(data.columns)

#fields = ['glohorizon', 'direct_nor', 'diffuse_ho', 'aspect_stu', 'topo_wetne', 'topo_rough', 'topo_conve', 'temp', 'Study_slop', 'merged_srt']

fields = ['glohorizon', 'direct_nor', 'diffuse_ho', 'aspect_stu', 'topo_wetne', 'topo_rough', 'topo_conve', 'temp', 'Study_slop', 'merged_srt', 'NW_wind_di',
          'NW_wind_sp']

labels = data['1_refugia_'].values
parttrain1 = data['parttrain1'].values

# Handle NaN values in the dataset
imputer = SimpleImputer(strategy='mean')
data_imputed = imputer.fit_transform(data[fields].values)

# Split data based on parttrain1
train_data = data_imputed[parttrain1 == 1]
validation_data = data_imputed[parttrain1 == 0]

train_labels = labels[parttrain1 == 1]
validation_labels = labels[parttrain1 == 0]

# Perform PCA on both datasets
pca = PCA(n_components=3)

# Training
projected_train = pca.fit_transform(train_data)

# Validation
projected_validation = pca.transform(validation_data)

# Print explained variance
print(f"PC1 explains: {pca.explained_variance_ratio_[0]*100:.2f}% of the variance")
```

```
print(f"PC2 explains: {pca.explained_variance_ratio_[1]*100:.2f}% of the variance")
print(f"PC3 explains: {pca.explained_variance_ratio_[2]*100:.2f}% of the variance")

eigenvalues = pca.explained_variance_
proportions = pca.explained_variance_ratio_
cumulative_proportions = np.cumsum(proportions)
eigenvectors = pca.components_

# Eigenanalysis of the Correlation Matrix
print("Eigenanalysis of the Correlation Matrix\n")
print("Eigenvalue    ", "    ".join([f"{val:.4f}" for val in eigenvalues]))
print("Proportion    ", "    ".join([f"{val:.3f}" for val in proportions]))
print("Cumulative    ", "    ".join([f"{val:.3f}" for val in cumulative_proportions]))
print("\nEigenvectors\n")
print("Variable", "    ".join([f"PC{i+1}" for i in range(len(eigenvalues))]))
for idx, var in enumerate(fields):
    print(var, "    ".join([f"{eigenvectors[pc][idx]:.3f}" for pc in range(len(eigenvalues))]))

colors = {
    'train_refugia': 'darkgreen',
    'train_not_refugia': 'darkred',
    'validation_refugia': 'lightgreen',
    'validation_not_refugia': 'lightcoral'
}

# Plotting
# Combined 3D plot for both Refugia and Not Refugia
fig = plt.figure(figsize=(12, 9))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(projected_train[train_labels == 1, 0], projected_train[train_labels == 1, 1], projected_train[train_labels == 1, 2],
c=colors['train_refugia'], label='Train - Refugia')
ax.scatter(projected_train[train_labels == 2, 0], projected_train[train_labels == 2, 1], projected_train[train_labels == 2, 2],
c=colors['train_not_refugia'], label='Train - Not Refugia')
ax.scatter(projected_validation[validation_labels == 1, 0], projected_validation[validation_labels == 1, 1],
projected_validation[validation_labels == 1, 2], c=colors['validation_refugia'], label='Validation - Refugia')
ax.scatter(projected_validation[validation_labels == 2, 0], projected_validation[validation_labels == 2, 1],
projected_validation[validation_labels == 2, 2], c=colors['validation_not_refugia'], label='Validation - Not Refugia')
ax.set_xlabel(f'PC1 - {pca.explained_variance_ratio_[0]*100:.2f}%')
ax.set_ylabel(f'PC2 - {pca.explained_variance_ratio_[1]*100:.2f}%')
ax.set_zlabel(f'PC3 - {pca.explained_variance_ratio_[2]*100:.2f}%')
ax.set_xlim([x_min, x_max])
ax.set_ylim([y_min, y_max])
ax.set_zlim([z_min, z_max])
ax.legend()
```

```
plt.show()

# 2D scatter plot for PC1 and PC2
plt.figure(figsize=(10, 7))
plt.scatter(projected_train[train_labels == 1, 0], projected_train[train_labels == 1, 1], c=colors['train_refugia'], label='Train - Refugia')
plt.scatter(projected_train[train_labels == 2, 0], projected_train[train_labels == 2, 1], c=colors['train_not_refugia'], label='Train - Not Refugia')
plt.scatter(projected_validation[validation_labels == 1, 0], projected_validation[validation_labels == 1, 1], c=colors['validation_refugia'], label='Validation - Refugia')
plt.scatter(projected_validation[validation_labels == 2, 0], projected_validation[validation_labels == 2, 1], c=colors['validation_not_refugia'], label='Validation - Not Refugia')
plt.xlabel(f'PC1 - {pca.explained_variance_ratio_[0]*100:.2f}%')
plt.ylabel(f'PC2 - {pca.explained_variance_ratio_[1]*100:.2f}%')
plt.legend()
plt.title('2D Scatter Plot for PC1 vs PC2')
plt.show()

# Split data based on parttrain1
train_data = data_imputed[parttrain1 == 1]
validation_data = data_imputed[parttrain1 == 0]

train_labels = labels[parttrain1 == 1]
validation_labels = labels[parttrain1 == 0]

# Perform PCA on training dataset
pca_train = PCA(n_components=3)
projected_train = pca_train.fit_transform(train_data)

# Print explained variance for training PCA
print("Explained Variance for Training Data:")
print(f"Train PC1 explains: {pca_train.explained_variance_ratio_[0]*100:.2f}% of the variance")
print(f"Train PC2 explains: {pca_train.explained_variance_ratio_[1]*100:.2f}% of the variance")
print(f"Train PC3 explains: {pca_train.explained_variance_ratio_[2]*100:.2f}% of the variance")

# Perform PCA on validation dataset
pca_validation = PCA(n_components=3)
projected_validation = pca_validation.fit_transform(validation_data)

# Print explained variance for validation PCA
print("\nExplained Variance for Validation Data:")
print(f"Validation PC1 explains: {pca_validation.explained_variance_ratio_[0]*100:.2f}% of the variance")
print(f"Validation PC2 explains: {pca_validation.explained_variance_ratio_[1]*100:.2f}% of the variance")
print(f"Validation PC3 explains: {pca_validation.explained_variance_ratio_[2]*100:.2f}% of the variance")
```

```
# Function to print explained variance and eigenanalysis
def print_pca_analysis(pca, name=""):
    print(f"\n{name} PCA Analysis")
    print(f"PC1 explains: {pca.explained_variance_ratio_[0]*100:.2f}% of the variance")
    print(f"PC2 explains: {pca.explained_variance_ratio_[1]*100:.2f}% of the variance")
    print(f"PC3 explains: {pca.explained_variance_ratio_[2]*100:.2f}% of the variance")

    eigenvalues = pca.explained_variance_
    proportions = pca.explained_variance_ratio_
    cumulative_proportions = np.cumsum(proportions)
    eigenvectors = pca.components_

    # Eigenanalysis of the Correlation Matrix
    print("\nEigenanalysis of the Correlation Matrix\n")
    print("Eigenvalue    ", "    ".join([f"{val:.4f}" for val in eigenvalues]))
    print("Proportion    ", "    ".join([f"{val:.3f}" for val in proportions]))
    print("Cumulative    ", "    ".join([f"{val:.3f}" for val in cumulative_proportions]))
    print("\nEigenvectors\n")
    print("Variable", "    ".join([f"PC{i+1}" for i in range(len(eigenvalues))]))
    for idx, var in enumerate(fields):
        print(var, "    ".join([f"{eigenvectors[pc][idx]:.3f}" for pc in range(len(eigenvalues))]))

# Split data based on parttrain1
train_data = data_imputed[parttrain1 == 1]
validation_data = data_imputed[parttrain1 == 0]

train_labels = labels[parttrain1 == 1]
validation_labels = labels[parttrain1 == 0]

# Perform PCA on training dataset
pca_train = PCA(n_components=3)
projected_train = pca_train.fit_transform(train_data)
print_pca_analysis(pca_train, name="Training Data")

# Perform PCA on validation dataset
pca_validation = PCA(n_components=3)
projected_validation = pca_validation.fit_transform(validation_data)
print_pca_analysis(pca_validation, name="Validation Data")

def plot_3d(data, labels, group_label, color):
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(data[labels == 1, 0], data[labels == 1, 1], data[labels == 1, 2], c=color, label=f'{group_label} - Refugia')
```

```
    ax.scatter(data[labels == 2, 0], data[labels == 2, 1], data[labels == 2, 2], c=color, label=f'{group_label} - Not Refugia')
    ax.set_xlabel(f'PC1 - {pca.explained_variance_ratio_[0]*100:.2f}%')
    ax.set_ylabel(f'PC2 - {pca.explained_variance_ratio_[1]*100:.2f}%')
    ax.set_zlabel(f'PC3 - {pca.explained_variance_ratio_[2]*100:.2f}%')
    ax.set_title(f'{group_label}')
    ax.legend()
    plt.show()

# Now call the function for each group:

# Train - Refugia
plot_3d(projected_train, train_labels, 'Train', colors['train_refugia'])

# Train - Not Refugia
plot_3d(projected_train, train_labels, 'Train', colors['train_not_refugia'])

# Validation - Refugia
plot_3d(projected_validation, validation_labels, 'Validation', colors['validation_refugia'])

# Validation - Not Refugia
plot_3d(projected_validation, validation_labels, 'Validation', colors['validation_not_refugia'])

def plot_combined_3d(train_data, validation_data, train_labels, validation_labels, label_value, title):
    fig = plt.figure(figsize=(10, 7))
    ax = fig.add_subplot(111, projection='3d')

    # Plotting training data
    ax.scatter(train_data[train_labels == label_value, 0],
               train_data[train_labels == label_value, 1],
               train_data[train_labels == label_value, 2],
               c=colors['train_refugia' if label_value == 1 else 'train_not_refugia'],
               label='Training Data')

    # Plotting validation data
    ax.scatter(validation_data[validation_labels == label_value, 0],
               validation_data[validation_labels == label_value, 1],
               validation_data[validation_labels == label_value, 2],
               c=colors['validation_refugia' if label_value == 1 else 'validation_not_refugia'],
               label='Validation Data')

    ax.set_xlabel(f'PC1 - {pca.explained_variance_ratio_[0]*100:.2f}%')
    ax.set_ylabel(f'PC2 - {pca.explained_variance_ratio_[1]*100:.2f}%')
    ax.set_zlabel(f'PC3 - {pca.explained_variance_ratio_[2]*100:.2f}%')
    ax.set_title(title)
```

```
    ax.legend()
    plt.show()

# All refugia plot (separated by training and validation)
plot_combined_3d(projected_train, projected_validation, train_labels, validation_labels, 1, 'Refugia (Training vs Validation)')

# All not-refugia plot (separated by training and validation)
plot_combined_3d(projected_train, projected_validation, train_labels, validation_labels, 2, 'Not-Refugia (Training vs Validation)')
```

Appendix S4. Random forest script with grid search and ADASYN sampling

```
import os # RF with ADASYN
import rasterio
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_auc_score
)
from sklearn.model_selection import GridSearchCV
from sklearn.impute import SimpleImputer
from imblearn.over_sampling import ADASYN
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import csv

# ---------------------------
# 1. Load raster with training and validation data
# ---------------------------
dataset_path = r'location to training and validation dataset'

with rasterio.open(dataset_path) as src:
    data = src.read(1)
    profile = src.profile
    height, width = src.shape

# Keep only classes 1 and 2; everything else = 0 (unlabelled)
data[(data != 1) & (data != 2)] = 0

# ---------------------------
# 2. Load feature rasters
# ---------------------------
raster_files = [
  r'location to elevation',
  r'location to aspect',
  r'location to diffuse horizntal irradiation',
  r'location to direct normal irradiation',
  r'location to global horizontal irradiation',
  r'location to slope',
  r'location to cfd wind direction',
  r'location to cfd wind speed',
  r'location to temperature',
  r'location to topographical covergence',
  r'location to topographical roughness',
  r'location to topographical roughness index
]

rasters = [rasterio.open(file).read(1) for file in raster_files]

# Stack rasters into [n_pixels, n_features]
```

```
X_full = np.column_stack([raster.ravel() for raster in rasters])

# Handle missing values in X_full
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
X_full = imputer.fit_transform(X_full)

raster_names = [os.path.splitext(os.path.basename(file))[0] for file in raster_files]

print("X_full shape (all pixels):", X_full.shape)

# --------------------------------------
# 3. Labels & spatial block setup
# --------------------------------------

# label_mask: pixels with 1 (refugia) or 2 (non-refugia)
label_mask = data != 0

# Encode labels: refugia = 1, non-refugia = 0, others = -1 to keep classifier clean and avoid classifier errors that
    occured
flat_data = data.ravel()
y_all = np.full(flat_data.shape, -1, dtype=int)
y_all[flat_data == 1] = 1      # refugia
y_all[flat_data == 2] = 0      # non-refugia

rows, cols = np.indices((height, width))

# Block size for selection
tile_h, tile_w = 64, 64

block_row = rows // tile_h
block_col = cols // tile_w

n_block_rows = int(np.ceil(height / tile_h))
n_block_cols = int(np.ceil(width   / tile_w))

block_ids = block_row * n_block_cols + block_col    # unique ID per block

# Only blocks with at least one labelled pixel
block_ids_labelled = block_ids[label_mask]
unique_blocks = np.unique(block_ids_labelled)

# ADASYN setup
adasyn = ADASYN(random_state=80)

# Feature raster profile for predictions
with rasterio.open(raster_files[0]) as src0:
    pred_profile = src0.profile
pred_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

# Output base directory
base_output_dir = r'location for output datasets'
os.makedirs(base_output_dir, exist_ok=True)
```

```
# ------------------------
# 4. Parameter grid
# ------------------------
param_grid = {
    'n_estimators': [50, 100, 200, 500],
    'max_features': ['sqrt', 'log2'],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'oob_score': [True]
}

# ------------------------
# 5. visual check confusion matrix plot
# ------------------------
def plot_confusion_matrix(cm, title_suffix=""):
    plt.figure(figsize=(8, 6))
    sns.heatmap(
        cm, annot=True, fmt='g', cmap='Blues',
        xticklabels=['Refugia', 'Not Refugia'],
        yticklabels=['Refugia', 'Not Refugia']
    )
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix {title_suffix}')
    plt.tight_layout()
    plt.show()

# ------------------------
# 6. Multi-run loop: Vary ONLY spatial split, RF seed fixed
# ------------------------
n_runs = 10
rf_seed = 80                    # fixed RF random_state for all runs
val_fraction = 0.30
max_split_tries = 50

all_run_metrics = []
best_params = None
best_params_found = False

# For refugia probability map
sum_refugia_preds = np.zeros((height, width), dtype=np.float32)

# For per-run feature importances
all_feature_importances = []

# ------------------------
# Loop over runs
# ------------------------
for i in range(n_runs):
    run_id = i + 1
    print(f"\n=== Run {run_id}/{n_runs} with RF random_state={rf_seed} (fixed) ===")
```

```
# ------------------------
# 6.1 New spatial block split for this run
# ------------------------
# Different seed per run for block selection (spatial variation)
rng_blocks = np.random.RandomState(1000 + run_id)

# Try multiple times to get a split where train has both classes (0 and 1)
for attempt in range(max_split_tries):
    n_val_blocks = max(1, int(np.round(val_fraction * len(unique_blocks))))
    val_blocks = rng_blocks.choice(unique_blocks, size=n_val_blocks, replace=False)
    train_blocks = np.setdiff1d(unique_blocks, val_blocks)

    train_mask = label_mask & np.isin(block_ids, train_blocks)
    val_mask   = label_mask & np.isin(block_ids, val_blocks)

    train_idx = np.where(train_mask.ravel())[0]
    val_idx   = np.where(val_mask.ravel())[0]

    y_train_all = y_all[train_idx]
    uniq_train_classes = np.unique(y_train_all)

    # Check that both classes {0, 1} are present in training
    if set(uniq_train_classes.tolist()) >= {0, 1}:
        print(f"   Split attempt {attempt+1}: OK (both classes in train)")
        break
    else:
        print(f"   Split attempt {attempt+1}: only classes {uniq_train_classes}, retrying...")
else:
    raise RuntimeError("Could not find a train/val split with both classes in training.")

# Build X and y for split
X = X_full
X_train = X[train_idx, :]
y_train = y_all[train_idx]

X_val   = X[val_idx, :]
y_val   = y_all[val_idx]

n_train = X_train.shape[0]
n_val   = X_val.shape[0]
print(f"   Train pixels: {n_train}, Val pixels: {n_val}, Val fraction: {n_val / (n_train + n_val):.3f}")

# Save the train/valilidation split raster
split_map = np.zeros_like(data, dtype=np.uint8)
split_map[train_mask] = 1
split_map[val_mask]   = 2

split_profile = profile.copy()
split_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

split_raster_path = os.path.join(
    base_output_dir,
    f'train_val_split_run{run_id}.tif'
```

```
)
with rasterio.open(split_raster_path, 'w', **split_profile) as dst:
    dst.write(split_map, 1)
print(f"    Train/validation split raster saved to: {split_raster_path}")

# Save training "sites" (blocks) info
train_block_ids = np.unique(block_ids[train_mask])
train_sites_rows = []
for b in train_block_ids:
    mask_b = (block_ids == b) & label_mask
    flat_idx_b = np.where(mask_b.ravel())[0]
    labels_b = y_all[flat_idx_b]
    n_refugia = np.sum(labels_b == 1)
    n_nonrefugia = np.sum(labels_b == 0)
    train_sites_rows.append({
        'block_id': int(b),
        'n_labelled_pixels': int(len(flat_idx_b)),
        'n_refugia_pixels': int(n_refugia),
        'n_nonrefugia_pixels': int(n_nonrefugia)
    })

train_sites_df = pd.DataFrame(train_sites_rows)
train_sites_csv_path = os.path.join(
    base_output_dir,
    f'train_blocks_run{run_id}.csv'
)
train_sites_df.to_csv(train_sites_csv_path, index=False)
print(f"    Training blocks info saved to: {train_sites_csv_path}")

# ------------------------
# 6.2 ADASYN
# ------------------------
X_train_resampled, y_train_resampled = adasyn.fit_resample(X_train, y_train)
print("    After ADASYN - train samples:", X_train_resampled.shape[0])

# ------------------------
# 6.3 Get model
# ------------------------
if run_id == 1:
    # First run: GridSearch
    rf = RandomForestClassifier(random_state=rf_seed, n_jobs=-1)
    grid_search = GridSearchCV(
        estimator=rf,
        param_grid=param_grid,
        cv=3,
        n_jobs=-1,
        verbose=2
    )
    grid_search.fit(X_train_resampled, y_train_resampled)
    best_params = grid_search.best_params_
    best_params_found = True

    print("    Best Parameters (from run 1):")
```

```
    for param, value in best_params.items():
        print(f"    {param}: {value}")

    # Save best params once
    best_params_path = os.path.join(
        base_output_dir,
        'Random_forest_best_parameters_run1_seed80.csv'
    )
    with open(best_params_path, 'w', newline='') as file:
        writer = csv.writer(file)
        for key, value in best_params.items():
            writer.writerow([key, value])

    rf_kwargs = best_params.copy()
    rf_kwargs['random_state'] = rf_seed
    rf_kwargs['n_jobs'] = -1
    model = RandomForestClassifier(**rf_kwargs)
else:
    if not best_params_found:
        raise RuntimeError("Best parameters not found before run > 1.")
    rf_kwargs = best_params.copy()
    rf_kwargs['random_state'] = rf_seed
    rf_kwargs['n_jobs'] = -1
    model = RandomForestClassifier(**rf_kwargs)

# ------------------------
# 6.4 Fit + evaluate
# ------------------------
model.fit(X_train_resampled, y_train_resampled)

y_pred = model.predict(X_val)
y_pred_proba = model.predict_proba(X_val)[:, 1]

if np.isnan(y_pred_proba).any():
    y_pred_proba = np.nan_to_num(y_pred_proba)

accuracy = accuracy_score(y_val, y_pred)
precision = precision_score(y_val, y_pred, average='binary')
recall = recall_score(y_val, y_pred, average='binary')
f1 = f1_score(y_val, y_pred, average='binary')
cm = confusion_matrix(y_val, y_pred)
roc_auc = roc_auc_score(y_val, y_pred_proba)
oob = model.oob_score_ if hasattr(model, "oob_score_") else np.nan

print("   Unique predictions:", np.unique(y_pred))
print('   Accuracy:', accuracy)
print('   Precision:', precision)
print('   Recall:', recall)
print('   F1 Score:', f1)
print('   OOB Score:', oob)
print('   AUC-ROC:', roc_auc)
print('   Confusion Matrix:\n', cm)
```

```
        if run_id == 1:
            plot_confusion_matrix(cm, title_suffix=f"(run 1, seed={rf_seed})")

        # Feature importances for this run
        importances = model.feature_importances_

        run_metrics = {
            'run': run_id,
            'RF_seed': rf_seed,
            'Accuracy': accuracy,
            'Precision': precision,
            'Recall': recall,
            'F1': f1,
            'OOB': oob,
            'AUC_ROC': roc_auc,
            'Train_pixels': int(n_train),
            'Val_pixels': int(n_val)
        }

        for feat_name, imp in zip(raster_names, importances):
            run_metrics[f'FI_{feat_name}'] = imp

        all_run_metrics.append(run_metrics)

        # save feature importances
        fi_df = pd.DataFrame({
            'run': run_id,
            'Feature': raster_names,
            'Importance': importances
        })
        fi_run_path = os.path.join(
            base_output_dir,
            f'Random_forest_feature_importances_run{run_id}.csv'
        )
        fi_df.to_csv(fi_run_path, index=False)
        print(f"    Feature importances saved to: {fi_run_path}")

        all_feature_importances.append(fi_df)

        # ------------------------
        # 6.5 Save per-run prediction raster
        # ------------------------
        full_predictions = model.predict(X_full)            # 0/1
        full_pred_map = full_predictions.reshape((height, width))

        run_pred_path = os.path.join(
            base_output_dir,
            f'Random_forest_experiment_run{run_id}.tif'
        )
    # Use elevation raster as base to keep raster formatting stable
        with rasterio.open(
            r'location to raster'
        ) as src0:
```

```
            pred_profile = src0.profile

        pred_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

        print(f"    Per-run prediction raster saved to: {run_pred_path}")

        sum_refugia_preds += full_pred_map.astype(np.float32)

# ------------------------
# 7. Metrics
# ------------------------
metrics_df = pd.DataFrame(all_run_metrics)

metrics_csv_path = os.path.join(
    base_output_dir,
    'Random_forest_all_runs_metrics_spatial_splits_only.csv'
)
metrics_df.to_csv(metrics_csv_path, index=False)
print("\nPer-run metrics + feature importances (wide) saved to:", metrics_csv_path)

summary_rows = []
metric_names = ['Accuracy', 'Precision', 'Recall', 'F1', 'OOB', 'AUC_ROC']
n = n_runs

for m in metric_names:
    vals = metrics_df[m].values
    mean = np.mean(vals)
    std = np.std(vals, ddof=1)
    ci_half_width = 1.96 * std / np.sqrt(n)
    ci_low = mean - ci_half_width
    ci_high = mean + ci_half_width

    summary_rows.append({
        'Metric': m,
        'Mean': mean,
        'Std': std,
        'CI_lower_95': ci_low,
        'CI_upper_95': ci_high
    })

summary_df = pd.DataFrame(summary_rows)
summary_csv_path = os.path.join(
    base_output_dir,
    'Random_forest_metrics_summary_spatial_splits_only.csv'
)
summary_df.to_csv(summary_csv_path, index=False)
print("Summary metrics (mean/std/CI) saved to:", summary_csv_path)

print("\nSummary metrics:")
print(summary_df)

# ------------------------
# 8. Feature importances summary
```

```
# ------------------------
fi_all = pd.concat(all_feature_importances, ignore_index=True)

fi_all_path = os.path.join(
    base_output_dir,
    'Random_forest_feature_importances_all_runs_spatial_splits_only.csv'
)
fi_all.to_csv(fi_all_path, index=False)
print("All per-run feature importances (long) saved to:", fi_all_path)

# Summary per feature
fi_summary = (
    fi_all
    .groupby('Feature')['Importance']
    .agg(['mean', 'std'])
    .reset_index()
    .rename(columns={'mean': 'Mean_importance', 'std': 'Std_importance'})
)

fi_summary['CI_lower_95'] = (
    fi_summary['Mean_importance']
    - 1.96 * fi_summary['Std_importance'] / np.sqrt(n_runs)
)
fi_summary['CI_upper_95'] = (
    fi_summary['Mean_importance']
    + 1.96 * fi_summary['Std_importance'] / np.sqrt(n_runs)
)

fi_summary_csv_path = os.path.join(
    base_output_dir,
    'Random_forest_feature_importances_summary_spatial_splits_only.csv'
)
fi_summary.to_csv(fi_summary_csv_path, index=False)
print("Feature importances summary (mean/std/CI) saved to:", fi_summary_csv_path)

# ------------------------
# 9. Refugia probability map
# ------------------------
refugia_prob_map = sum_refugia_preds / float(n_runs)

prob_profile = pred_profile.copy()
prob_profile.update(dtype=rasterio.float32, count=1, compress='lzw')

prob_raster_path = os.path.join(
    base_output_dir,
    'Random_forest_refugia_probability_0_1_spatial_splits_only.tif'
)

with rasterio.open(prob_raster_path, 'w', **prob_profile) as dst:
    dst.write(refugia_prob_map.astype(rasterio.float32), 1)

print("Refugia probability raster saved to:", prob_raster_path)
```

Appendix S5. XGBoost implementation in Python

```
import os    # XGBoost with ADASYN and spatial splits
import rasterio
import xgboost as xgb
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_auc_score
)
from sklearn.model_selection import GridSearchCV
from sklearn.impute import SimpleImputer
from imblearn.over_sampling import ADASYN
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import csv

# ---------------------------
# 1. Load raster with training and validation data
# ---------------------------
dataset_path = r'location to training and validation dataset'

with rasterio.open(dataset_path) as src:
    data = src.read(1)
    profile = src.profile
    height, width = src.shape

# Keep only classes 1 and 2; everything else = 0 (unlabelled)
data[(data != 1) & (data != 2)] = 0

# ---------------------------
# 2. Load feature rasters
# ---------------------------
raster_files = [
  r'location to elevation',
  r'location to aspect',
  r'location to diffuse horizntal irradiation',
  r'location to direct normal irradiation',
  r'location to global horizontal irradiation',
  r'location to slope',
  r'location to cfd wind direction',
  r'location to cfd wind speed',
  r'location to temperature',
  r'location to topographical covergence',
  r'location to topographical roughness',
  r'location to topographical roughness index
]

rasters = [rasterio.open(file).read(1) for file in raster_files]

# Stack rasters into [n_pixels, n_features]
```

```
X_full = np.column_stack([raster.ravel() for raster in rasters])

# Handle missing values in X_full
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
X_full = imputer.fit_transform(X_full)

raster_names = [os.path.splitext(os.path.basename(file))[0] for file in raster_files]

print("X_full shape (all pixels):", X_full.shape)

# ---------------------------------------
# 3. Labels & spatial block setup
# ---------------------------------------

# label_mask: pixels with 1 (refugia) or 2 (non-refugia)
label_mask = data != 0

# Encode labels: refugia = 1, non-refugia = 0, others = -1 to keep classifier clean and avoid classifier errors that
        occured
flat_data = data.ravel()
y_all = np.full(flat_data.shape, -1, dtype=int)
y_all[flat_data == 1] = 1     # refugia
y_all[flat_data == 2] = 0     # non-refugia

rows, cols = np.indices((height, width))

# Block size for selection
tile_h, tile_w = 64, 64

block_row = rows // tile_h
block_col = cols // tile_w

n_block_rows = int(np.ceil(height / tile_h))
n_block_cols = int(np.ceil(width   / tile_w))

block_ids = block_row * n_block_cols + block_col   # unique ID per block

# Only blocks with at least one labelled pixel
block_ids_labelled = block_ids[label_mask]
unique_blocks = np.unique(block_ids_labelled)

# ADASYN setup
adasyn = ADASYN(random_state=80)

# Feature raster profile for predictions to keep a stable raster
with rasterio.open(
    r'location of elevation'
) as src0:
    pred_profile = src0.profile
pred_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

# Output base directory (parallel to RF version)
base_output_dir = r'output location for output'
```

```
os.makedirs(base_output_dir, exist_ok=True)

# ------------------------
# 4. Paramaater grid
# ------------------------
param_grid = {
    'n_estimators': [50, 100, 200, 1500],
    'eta': [0.01, 0.1, 0.5],
    'max_depth': [3, 5, 7],
    'reg_alpha': [0, 0.001, 0.005, 0.01, 0.05],
    'reg_lambda': [0.5, 1, 1.5, 2],
}

# ------------------------
# 5.Confusion matrix plot
# ------------------------
def plot_confusion_matrix(cm, title_suffix=""):
    plt.figure(figsize=(8, 6))
    sns.heatmap(
        cm, annot=True, fmt='g', cmap='Blues',
        xticklabels=['Refugia', 'Not Refugia'],
        yticklabels=['Refugia', 'Not Refugia']
    )
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix {title_suffix}')
    plt.tight_layout()
    plt.show()

# ------------------------
# 6. Multi-run loop: Vary ONLY spatial split, XGB seed fixed
# ------------------------
n_runs = 10
xgb_seed = 80
val_fraction = 0.30
max_split_tries = 50

all_run_metrics = []
best_params = None
best_params_found = False

# For refugia probability map (
sum_refugia_preds = np.zeros((height, width), dtype=np.float32)

# For per-run feature importances
all_feature_importances = []

# ------------------------
# Loop over runs
# ------------------------
for i in range(n_runs):
    run_id = i + 1
    print(f"\n=== Run {run_id}/{n_runs} with XGBoost random_state={xgb_seed} (fixed) ===")
```

```
# ------------------------
    # 6.1 New spatial block split for this run
# ------------------------
# Different seed per run for block selection (spatial variation)
rng_blocks = np.random.RandomState(1000 + run_id)

# Try multiple times to get a split where train has both classes (0 and 1)
for attempt in range(max_split_tries):
    n_val_blocks = max(1, int(np.round(val_fraction * len(unique_blocks))))
    val_blocks = rng_blocks.choice(unique_blocks, size=n_val_blocks, replace=False)
    train_blocks = np.setdiff1d(unique_blocks, val_blocks)

    train_mask = label_mask & np.isin(block_ids, train_blocks)
    val_mask   = label_mask & np.isin(block_ids, val_blocks)

    train_idx = np.where(train_mask.ravel())[0]
    val_idx   = np.where(val_mask.ravel())[0]

    y_train_all = y_all[train_idx]
    uniq_train_classes = np.unique(y_train_all)

    # Check that both classes {0, 1} are present in training
    if set(uniq_train_classes.tolist()) >= {0, 1}:
        print(f"   Split attempt {attempt+1}: OK (both classes in train)")
        break
    else:
        print(f"   Split attempt {attempt+1}: only classes {uniq_train_classes}, retrying...")
else:
    raise RuntimeError("Could not find a train/val split with both classes in training.")

X = X_full
X_train = X[train_idx, :]
y_train = y_all[train_idx]

X_val   = X[val_idx, :]
y_val   = y_all[val_idx]

n_train = X_train.shape[0]
n_val   = X_val.shape[0]
print(f"   Train pixels: {n_train}, Val pixels: {n_val}, Val fraction: {n_val / (n_train + n_val):.3f}")

split_map = np.zeros_like(data, dtype=np.uint8)
split_map[train_mask] = 1
split_map[val_mask]   = 2

split_profile = profile.copy()
split_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

split_raster_path = os.path.join(
    base_output_dir,
    f'train_val_split_run{run_id}.tif'
)
```

```
with rasterio.open(split_raster_path, 'w', **split_profile) as dst:
    dst.write(split_map, 1)
print(f"    Train/validation split raster saved to: {split_raster_path}")

# Save training "sites" (blocks) info
train_block_ids = np.unique(block_ids[train_mask])
train_sites_rows = []
for b in train_block_ids:
    mask_b = (block_ids == b) & label_mask
    flat_idx_b = np.where(mask_b.ravel())[0]
    labels_b = y_all[flat_idx_b]
    n_refugia = np.sum(labels_b == 1)
    n_nonrefugia = np.sum(labels_b == 0)
    train_sites_rows.append({
        'block_id': int(b),
        'n_labelled_pixels': int(len(flat_idx_b)),
        'n_refugia_pixels': int(n_refugia),
        'n_nonrefugia_pixels': int(n_nonrefugia)
    })

train_sites_df = pd.DataFrame(train_sites_rows)
train_sites_csv_path = os.path.join(
    base_output_dir,
    f'train_blocks_run{run_id}.csv'
)
train_sites_df.to_csv(train_sites_csv_path, index=False)
print(f"    Training blocks info saved to: {train_sites_csv_path}")

# ------------------------
# 6.2 ADASYN on this run's train data
# ------------------------
X_train_resampled, y_train_resampled = adasyn.fit_resample(X_train, y_train)
print("    After ADASYN - train samples:", X_train_resampled.shape[0])

if run_id == 1:
    # First run: GridSearch
    base_model = xgb.XGBClassifier(
        random_state=xgb_seed,
        n_jobs=-1,
        use_label_encoder=False,
        eval_metric="logloss"
    )

    grid_search = GridSearchCV(
        estimator=base_model,
        param_grid=param_grid,
        cv=3,
        scoring='roc_auc',
        n_jobs=-1,
        verbose=2
    )

    grid_search.fit(X_train_resampled, y_train_resampled)
```

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

        print("    Best Parameters (from run 1):")
        for param, value in best_params.items():
            print(f"        {param}: {value}")
        print("    Best mean CV ROC AUC (run 1):", best_score)

        # Save best params once
        best_params_path = os.path.join(
            base_output_dir,
            'XGBoost_best_parameters_run1_seed80.csv'
        )
        with open(best_params_path, 'w', newline='') as file:
            writer = csv.writer(file)
            for key, value in best_params.items():
                writer.writerow([key, value])
            writer.writerow([])
            writer.writerow(['best_cv_roc_auc', best_score])

        xgb_kwargs = best_params.copy()
        xgb_kwargs['random_state'] = xgb_seed
        xgb_kwargs['n_jobs'] = -1
        xgb_kwargs['use_label_encoder'] = False
        xgb_kwargs['eval_metric'] = 'logloss'
        model = xgb.XGBClassifier(**xgb_kwargs)
    else:
        # Subsequent runs: reuse best_params, same XGB seed
        if not best_params_found:
            raise RuntimeError("Best parameters not found before run > 1.")
        xgb_kwargs = best_params.copy()
        xgb_kwargs['random_state'] = xgb_seed
        xgb_kwargs['n_jobs'] = -1
        xgb_kwargs['use_label_encoder'] = False
        xgb_kwargs['eval_metric'] = 'logloss'
        model = xgb.XGBClassifier(**xgb_kwargs)

    # ------------------------
    # 6.4 Fit + evaluate
    # ------------------------
    model.fit(X_train_resampled, y_train_resampled)

    y_pred = model.predict(X_val)
    y_pred_proba = model.predict_proba(X_val)[:, 1]

    if np.isnan(y_pred_proba).any():
        y_pred_proba = np.nan_to_num(y_pred_proba)

    accuracy = accuracy_score(y_val, y_pred)
    precision = precision_score(y_val, y_pred, average='binary')
    recall = recall_score(y_val, y_pred, average='binary')
    f1 = f1_score(y_val, y_pred, average='binary')
```

```
cm = confusion_matrix(y_val, y_pred)
roc_auc = roc_auc_score(y_val, y_pred_proba)

print("    Unique predictions:", np.unique(y_pred))
print('    Accuracy:', accuracy)
print('    Precision:', precision)
print('    Recall:', recall)
print('    F1 Score:', f1)
print('    AUC-ROC:', roc_auc)
print('    Confusion Matrix:\n', cm)

if run_id == 1:
    plot_confusion_matrix(cm, title_suffix=f"(run 1, seed={xgb_seed})")

# ------------------------
# 6.4b Feature importances (gain) for this run
# ------------------------
booster = model.get_booster()
fscores = booster.get_score(importance_type="gain")

importances = []
for i in range(len(raster_names)):
    key = f"f{i}"
    importances.append(fscores.get(key, 0.0))
importances = np.array(importances, dtype=float)

run_metrics = {
    'run': run_id,
    'XGB_seed': xgb_seed,
    'Accuracy': accuracy,
    'Precision': precision,
    'Recall': recall,
    'F1': f1,
    'AUC_ROC': roc_auc,
    'Train_pixels': int(n_train),
    'Val_pixels': int(n_val)
}
for feat_name, imp in zip(raster_names, importances):
    run_metrics[f'FI_{feat_name}'] = imp

all_run_metrics.append(run_metrics)

# Also save feature importances for this run
fi_df = pd.DataFrame({
    'run': run_id,
    'Feature': raster_names,
    'Importance': importances
})
fi_run_path = os.path.join(
    base_output_dir,
    f'XGBoost_feature_importances_run{run_id}.csv'
)
fi_df.to_csv(fi_run_path, index=False)
```

```
        print(f"    Feature importances saved to: {fi_run_path}")

        all_feature_importances.append(fi_df)

        # ------------------------
        # 6.5 Save per-run prediction raster
        # ------------------------
        full_predictions = model.predict(X_full)            # 0/1
        full_pred_map = full_predictions.reshape((height, width))

        run_pred_path = os.path.join(
            base_output_dir,
            f'XGBoost_experiment_run{run_id}.tif'
        )
        with rasterio.open(run_pred_path, 'w', **pred_profile) as dst:
            dst.write(full_pred_map.astype(rasterio.uint8), 1)

        print(f"    Per-run prediction raster saved to: {run_pred_path}")

        sum_refugia_preds += full_pred_map.astype(np.float32)

    # ------------------------
    # 7. Metrics: per-run CSV + summary stats
    # ------------------------
    metrics_df = pd.DataFrame(all_run_metrics)

    metrics_csv_path = os.path.join(
        base_output_dir,
        'XGBoost_all_runs_metrics_spatial_splits_only.csv'
    )
    metrics_df.to_csv(metrics_csv_path, index=False)
    print("\nPer-run metrics + feature importances (wide) saved to:", metrics_csv_path)

    summary_rows = []
    metric_names = ['Accuracy', 'Precision', 'Recall', 'F1', 'AUC_ROC']
    n = n_runs

    for m in metric_names:
        vals = metrics_df[m].values
        mean = np.mean(vals)
        std = np.std(vals, ddof=1)
        ci_half_width = 1.96 * std / np.sqrt(n)
        ci_low = mean - ci_half_width
        ci_high = mean + ci_half_width

        summary_rows.append({
            'Metric': m,
            'Mean': mean,
            'Std': std,
            'CI_lower_95': ci_low,
            'CI_upper_95': ci_high
        })
```

```
summary_df = pd.DataFrame(summary_rows)
summary_csv_path = os.path.join(
    base_output_dir,
    'XGBoost_metrics_summary_spatial_splits_only.csv'
)
summary_df.to_csv(summary_csv_path, index=False)
print("Summary metrics (mean/std/CI) saved to:", summary_csv_path)

print("\nSummary metrics:")
print(summary_df)

# ------------------------
# 8. Feature importances summary across runs (long + summary with CI)
# ------------------------
fi_all = pd.concat(all_feature_importances, ignore_index=True)

fi_all_path = os.path.join(
    base_output_dir,
    'XGBoost_feature_importances_all_runs_spatial_splits_only.csv'
)
fi_all.to_csv(fi_all_path, index=False)
print("All per-run feature importances (long) saved to:", fi_all_path)

# Summary per feature
fi_summary = (
    fi_all
    .groupby('Feature')['Importance']
    .agg(['mean', 'std'])
    .reset_index()
    .rename(columns={'mean': 'Mean_importance', 'std': 'Std_importance'})
)

fi_summary['CI_lower_95'] = (
    fi_summary['Mean_importance']
    - 1.96 * fi_summary['Std_importance'] / np.sqrt(n_runs)
)
fi_summary['CI_upper_95'] = (
    fi_summary['Mean_importance']
    + 1.96 * fi_summary['Std_importance'] / np.sqrt(n_runs)
)

fi_summary_csv_path = os.path.join(
    base_output_dir,
    'XGBoost_feature_importances_summary_spatial_splits_only.csv'
)
fi_summary.to_csv(fi_summary_csv_path, index=False)
print("Feature importances summary (mean/std/CI) saved to:", fi_summary_csv_path)

# ------------------------
# 9. Refugia probability map (0–1: fraction of runs predicted as refugia)
# ------------------------
refugia_prob_map = sum_refugia_preds / float(n_runs)
```

```
prob_profile = pred_profile.copy()
prob_profile.update(dtype=rasterio.float32, count=1, compress='lzw')

prob_raster_path = os.path.join(
    base_output_dir,
    'XGBoost_refugia_probability_0_1_spatial_splits_only.tif'
)

with rasterio.open(prob_raster_path, 'w', **prob_profile) as dst:
    dst.write(refugia_prob_map.astype(rasterio.float32), 1)

print("Refugia probability raster saved to:", prob_raster_path)
```

Appendix S6. The KNN implementation was performed in Python

```
import os    # KNN with ADASYN and spatial splits
import rasterio
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_auc_score
)
from sklearn.model_selection import GridSearchCV
from sklearn.impute import SimpleImputer
from imblearn.over_sampling import ADASYN
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import csv

# ---------------------------
# 1. Load raster with training and validation data
# ---------------------------
dataset_path = r'location to training and validation dataset'

with rasterio.open(dataset_path) as src:
    data = src.read(1)
    profile = src.profile
    height, width = src.shape

# Keep only classes 1 and 2; everything else = 0 (unlabelled)
data[(data != 1) & (data != 2)] = 0

# ---------------------------
# 2. Load feature rasters
# ---------------------------
raster_files = [
  r'location to elevation',
  r'location to aspect',
  r'location to diffuse horizntal irradiation',
  r'location to direct normal irradiation',
  r'location to global horizontal irradiation',
  r'location to slope',
  r'location to cfd wind direction',
  r'location to cfd wind speed',
  r'location to temperature',
  r'location to topographical covergence',
  r'location to topographical roughness',
  r'location to topographical roughness index
]

rasters = [rasterio.open(file).read(1) for file in raster_files]

# Stack rasters into [n_pixels, n_features]
X_full = np.column_stack([raster.ravel() for raster in rasters])
```

```
# Handle missing values in X_full
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
X_full = imputer.fit_transform(X_full)

raster_names = [os.path.splitext(os.path.basename(file))[0] for file in raster_files]

print("X_full shape (all pixels):", X_full.shape)

# ---------------------------------------
# 3. Labels & spatial block setup
# ---------------------------------------

# label_mask: pixels with 1 (refugia) or 2 (non-refugia)
label_mask = data != 0

# Encode labels: refugia = 1, non-refugia = 0, others = -1 to keep classifier clean and avoid classifier errors that
    occured
flat_data = data.ravel()
y_all = np.full(flat_data.shape, -1, dtype=int)
y_all[flat_data == 1] = 1      # refugia
y_all[flat_data == 2] = 0      # non-refugia

rows, cols = np.indices((height, width))

# Block size for selection
tile_h, tile_w = 64, 64

block_row = rows // tile_h
block_col = cols // tile_w

n_block_rows = int(np.ceil(height / tile_h))
n_block_cols = int(np.ceil(width   / tile_w))

block_ids = block_row * n_block_cols + block_col    # unique ID per block

# Only blocks with at least one labelled pixel
block_ids_labelled = block_ids[label_mask]
unique_blocks = np.unique(block_ids_labelled)

# ADASYN setup
adasyn = ADASYN(random_state=80)

# Feature raster profile for predictions use elevation to keep raster output stable
with rasterio.open(
    r'location to elevation'
) as src0:
    pred_profile = src0.profile
pred_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

# Output base directory
base_output_dir = r'location to save output'
os.makedirs(base_output_dir, exist_ok=True)
```

```
# ------------------------
# 4. Parameter grid
# ------------------------
param_grid = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}

# ------------------------
# 5.Confusion matrix plot
# ------------------------
def plot_confusion_matrix(cm, title_suffix=""):
    plt.figure(figsize=(8, 6))
    sns.heatmap(
        cm, annot=True, fmt='g', cmap='Blues',
        xticklabels=['Refugia', 'Not Refugia'],
        yticklabels=['Refugia', 'Not Refugia']
    )
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix {title_suffix}')
    plt.tight_layout()
    plt.show()

# ------------------------
# 6. Multi-run loop: Vary ONLY spatial split
# ------------------------
n_runs = 10
val_fraction = 0.30
max_split_tries = 50

all_run_metrics = []
best_params = None
best_params_found = False

# For refugia probability map
sum_refugia_preds = np.zeros((height, width), dtype=np.float32)

# ------------------------
# Loop over runs
# ------------------------
for i in range(n_runs):
    run_id = i + 1
    print(f"\n=== Run {run_id}/{n_runs} with KNN (deterministic) ===")

    # Different seed per run for block selection (spatial variation)
    rng_blocks = np.random.RandomState(1000 + run_id)

    for attempt in range(max_split_tries):
        n_val_blocks = max(1, int(np.round(val_fraction * len(unique_blocks))))
        val_blocks = rng_blocks.choice(unique_blocks, size=n_val_blocks, replace=False)
```

```
        train_blocks = np.setdiff1d(unique_blocks, val_blocks)

        train_mask = label_mask & np.isin(block_ids, train_blocks)
        val_mask   = label_mask & np.isin(block_ids, val_blocks)

        train_idx = np.where(train_mask.ravel())[0]
        val_idx   = np.where(val_mask.ravel())[0]

        y_train_all = y_all[train_idx]
        uniq_train_classes = np.unique(y_train_all)

        if set(uniq_train_classes.tolist()) >= {0, 1}:
            print(f"   Split attempt {attempt+1}: OK (both classes in train)")
            break
        else:
            print(f"   Split attempt {attempt+1}: only classes {uniq_train_classes}, retrying...")
    else:
        raise RuntimeError("Could not find a train/val split with both classes in training.")

    # Build X and y
    X = X_full
    X_train = X[train_idx, :]
    y_train = y_all[train_idx]

    X_val   = X[val_idx, :]
    y_val   = y_all[val_idx]

    n_train = X_train.shape[0]
    n_val   = X_val.shape[0]
    print(f"   Train pixels: {n_train}, Val pixels: {n_val}, Val fraction: {n_val / (n_train + n_val):.3f}")

    # Save the split raster for this run
    split_map = np.zeros_like(data, dtype=np.uint8)
    split_map[train_mask] = 1
    split_map[val_mask]   = 2

    split_profile = profile.copy()
    split_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

    split_raster_path = os.path.join(
        base_output_dir,
        f'train_val_split_run{run_id}.tif'
    )
    with rasterio.open(split_raster_path, 'w', **split_profile) as dst:
        dst.write(split_map, 1)
    print(f"   Train/validation split raster saved to: {split_raster_path}")

    # Save training blocks
    train_block_ids = np.unique(block_ids[train_mask])
    train_sites_rows = []
    for b in train_block_ids:
        mask_b = (block_ids == b) & label_mask
        flat_idx_b = np.where(mask_b.ravel())[0]
```

```
        labels_b = y_all[flat_idx_b]
        n_refugia = np.sum(labels_b == 1)
        n_nonrefugia = np.sum(labels_b == 0)
        train_sites_rows.append({
            'block_id': int(b),
            'n_labelled_pixels': int(len(flat_idx_b)),
            'n_refugia_pixels': int(n_refugia),
            'n_nonrefugia_pixels': int(n_nonrefugia)
        })

    train_sites_df = pd.DataFrame(train_sites_rows)
    train_sites_csv_path = os.path.join(
        base_output_dir,
        f'train_blocks_run{run_id}.csv'
    )
    train_sites_df.to_csv(train_sites_csv_path, index=False)
    print(f"    Training blocks info saved to: {train_sites_csv_path}")

    # ------------------------
    # 6.2 ADASYN
    # ------------------------
    X_train_resampled, y_train_resampled = adasyn.fit_resample(X_train, y_train)
    print("    After ADASYN - train samples:", X_train_resampled.shape[0])

    if run_id == 1:
        base_model = KNeighborsClassifier(n_jobs=-1)

        grid_search = GridSearchCV(
            estimator=base_model,
            param_grid=param_grid,
            cv=3,
            scoring='accuracy',
            n_jobs=-1,
            verbose=2
        )

        grid_search.fit(X_train_resampled, y_train_resampled)
        best_params = grid_search.best_params_
        best_score = grid_search.best_score_
        best_params_found = True

        print("    Best Parameters (from run 1):")
        for param, value in best_params.items():
            print(f"        {param}: {value}")
        print("    Best mean CV accuracy (run 1):", best_score)

        # Save best parameter
        best_params_path = os.path.join(
            base_output_dir,
            'KNN_best_parameters_run1.csv'
        )
        with open(best_params_path, 'w', newline='') as file:
            writer = csv.writer(file)
```

```
            for key, value in best_params.items():
                writer.writerow([key, value])
            writer.writerow([])
            writer.writerow(['best_cv_accuracy', best_score])

        model = KNeighborsClassifier(**best_params, n_jobs=-1)
    else:
        # Subsequent runs: reuse best_params
        if not best_params_found:
            raise RuntimeError("Best parameters not found before run > 1.")
        model = KNeighborsClassifier(**best_params, n_jobs=-1)

    # ------------------------
    # 6.4 Fit + evaluate
    # ------------------------
    model.fit(X_train_resampled, y_train_resampled)

    y_pred = model.predict(X_val)
    y_pred_proba = model.predict_proba(X_val)[:, 1]

    if np.isnan(y_pred_proba).any():
        y_pred_proba = np.nan_to_num(y_pred_proba)

    accuracy = accuracy_score(y_val, y_pred)
    precision = precision_score(y_val, y_pred, average='binary')
    recall = recall_score(y_val, y_pred, average='binary')
    f1 = f1_score(y_val, y_pred, average='binary')
    cm = confusion_matrix(y_val, y_pred)
    roc_auc = roc_auc_score(y_val, y_pred_proba)

    print("    Unique predictions:", np.unique(y_pred))
    print('    Accuracy:', accuracy)
    print('    Precision:', precision)
    print('    Recall:', recall)
    print('    F1 Score:', f1)
    print('    AUC-ROC:', roc_auc)
    print('    Confusion Matrix:\n', cm)

    if run_id == 1:
        plot_confusion_matrix(cm, title_suffix="(run 1)")

    run_metrics = {
        'run': run_id,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1': f1,
        'AUC_ROC': roc_auc,
        'Train_pixels': int(n_train),
        'Val_pixels': int(n_val)
    }

    all_run_metrics.append(run_metrics)
```

```
    # ------------------------
    # 6.5 Save per-run prediction raster
    # ------------------------
    full_predictions = model.predict(X_full)
    full_pred_map = full_predictions.reshape((height, width))

    run_pred_path = os.path.join(
        base_output_dir,
        f'KNN_experiment_run{run_id}.tif'
    )
    with rasterio.open(run_pred_path, 'w', **pred_profile) as dst:
        dst.write(full_pred_map.astype(rasterio.uint8), 1)

    print(f"    Per-run prediction raster saved to: {run_pred_path}")

    sum_refugia_preds += full_pred_map.astype(np.float32)

# ------------------------
# 7. Metrics: per-run
# ------------------------
metrics_df = pd.DataFrame(all_run_metrics)

metrics_csv_path = os.path.join(
    base_output_dir,
    'KNN_all_runs_metrics_spatial_splits_only.csv'
)
metrics_df.to_csv(metrics_csv_path, index=False)
print("\nPer-run metrics saved to:", metrics_csv_path)

summary_rows = []
metric_names = ['Accuracy', 'Precision', 'Recall', 'F1', 'AUC_ROC']
n = n_runs

for m in metric_names:
    vals = metrics_df[m].values
    mean = np.mean(vals)
    std = np.std(vals, ddof=1)
    ci_half_width = 1.96 * std / np.sqrt(n)
    ci_low = mean - ci_half_width
    ci_high = mean + ci_half_width

    summary_rows.append({
        'Metric': m,
        'Mean': mean,
        'Std': std,
        'CI_lower_95': ci_low,
        'CI_upper_95': ci_high
    })

summary_df = pd.DataFrame(summary_rows)
summary_csv_path = os.path.join(
    base_output_dir,
```

```
    'KNN_metrics_summary_spatial_splits_only.csv'
)
summary_df.to_csv(summary_csv_path, index=False)
print("Summary metrics (mean/std/CI) saved to:", summary_csv_path)

print("\nSummary metrics:")
print(summary_df)

# ------------------------
# 8. Refugia probability map (0–1: fraction of runs predicted as refugia)
# ------------------------
refugia_prob_map = sum_refugia_preds / float(n_runs)

prob_profile = pred_profile.copy()
prob_profile.update(dtype=rasterio.float32, count=1, compress='lzw')

prob_raster_path = os.path.join(
    base_output_dir,
    'KNN_refugia_probability_0_1_spatial_splits_only.tif'
)

with rasterio.open(prob_raster_path, 'w', **prob_profile) as dst:
    dst.write(refugia_prob_map.astype(rasterio.float32), 1)

print("Refugia probability raster saved to:", prob_raster_path)
```

Appendix S7. Ensemble script soft

```
import os   # soft Ensemble with ADASYN and spatial splits
import rasterio
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.neighbors import KNeighborsClassifier
import xgboost as xgb
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_auc_score
)
from sklearn.impute import SimpleImputer
from imblearn.over_sampling import ADASYN
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import csv

# ---------------------------
# 1. Load raster with training and validation data
# ---------------------------
dataset_path = r'location to training and validation dataset'

with rasterio.open(dataset_path) as src:
    data = src.read(1)
    profile = src.profile
    height, width = src.shape

# Keep only classes 1 and 2; everything else = 0 (unlabelled)
data[(data != 1) & (data != 2)] = 0

# ---------------------------
# 2. Load feature rasters
# ---------------------------
raster_files = [
  r'location to elevation',
  r'location to aspect',
  r'location to diffuse horizntal irradiation',
  r'location to direct normal irradiation',
  r'location to global horizontal irradiation',
  r'location to slope',
  r'location to cfd wind direction',
  r'location to cfd wind speed',
  r'location to temperature',
  r'location to topographical covergence',
  r'location to topographical roughness',
  r'location to topographical roughness index
]

rasters = [rasterio.open(file).read(1) for file in raster_files]

# Stack rasters into [n_pixels, n_features]
```

```
X_full = np.column_stack([raster.ravel() for raster in rasters])

# Handle missing values in X_full
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
X_full = imputer.fit_transform(X_full)

raster_names = [os.path.splitext(os.path.basename(file))[0] for file in raster_files]

print("X_full shape (all pixels):", X_full.shape)

# --------------------------------------
# 3. Labels & spatial block setup
# --------------------------------------

# label_mask: pixels with 1 (refugia) or 2 (non-refugia)
label_mask = data != 0

# Encode labels: refugia = 1, non-refugia = 0, others = -1 to keep classifier clean and avoid classifier errors that
    occured
flat_data = data.ravel()
y_all = np.full(flat_data.shape, -1, dtype=int)
y_all[flat_data == 1] = 1      # refugia
y_all[flat_data == 2] = 0      # non-refugia

rows, cols = np.indices((height, width))

# Block size for selection
tile_h, tile_w = 64, 64

block_row = rows // tile_h
block_col = cols // tile_w

n_block_rows = int(np.ceil(height / tile_h))
n_block_cols = int(np.ceil(width    / tile_w))

block_ids = block_row * n_block_cols + block_col    # unique ID per block

# Only blocks with at least one labelled pixel
block_ids_labelled = block_ids[label_mask]
unique_blocks = np.unique(block_ids_labelled)

# ADASYN setup
adasyn = ADASYN(random_state=80)

# Use elevation for profile to keep output raster stable
with rasterio.open(
    r'location to elevation'
) as src0:
    pred_profile = src0.profile
pred_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

# Output base directory
base_output_dir = r'location to where output should go'
```

```
os.makedirs(base_output_dir, exist_ok=True)

# ------------------------
# 4.Confusion matrix plot
# ------------------------
def plot_confusion_matrix(cm, title_suffix=""):
    plt.figure(figsize=(8, 6))
    sns.heatmap(
        cm, annot=True, fmt='g', cmap='Blues',
        xticklabels=['Refugia', 'Not Refugia'],
        yticklabels=['Refugia', 'Not Refugia']
    )
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix {title_suffix}')
    plt.tight_layout()
    plt.show()

# ------------------------
# 5. Base classifiers (same as the best for the feature combination)
# ------------------------
rf_clf = RandomForestClassifier(
    n_estimators=500,
    max_features='sqrt',
    max_depth=30,
    min_samples_split=2,
    min_samples_leaf=1,
    oob_score=True,
    random_state=80,
    n_jobs=-1
)

xgb_clf = xgb.XGBClassifier(
    n_estimators=1500,
    eta=0.5,
    max_depth=7,
    reg_alpha=0.001,
    reg_lambda=0.5,
    use_label_encoder=False,
    eval_metric='logloss',
    n_jobs=-1
)

knn_clf = KNeighborsClassifier(
    n_neighbors=3,
    weights='distance',
    metric='manhattan',
    n_jobs=-1
)

# ------------------------
# 6. Spatial splits only
# ------------------------
```

```
n_runs = 10
val_fraction = 0.30
max_split_tries = 50

all_run_metrics = []

sum_refugia_proba = np.zeros((height, width), dtype=np.float32)

# ------------------------
# Loop over runs
# ------------------------
for i in range(n_runs):
    run_id = i + 1
    print(f"\n=== Run {run_id}/{n_runs} - Soft Ensemble (RF+KNN+XGB) ===")

    # ------------------------
    # 6.1 New spatial block split for this run
    # ------------------------
    rng_blocks = np.random.RandomState(1000 + run_id)

    for attempt in range(max_split_tries):
        n_val_blocks = max(1, int(np.round(val_fraction * len(unique_blocks))))
        val_blocks = rng_blocks.choice(unique_blocks, size=n_val_blocks, replace=False)
        train_blocks = np.setdiff1d(unique_blocks, val_blocks)

        train_mask = label_mask & np.isin(block_ids, train_blocks)
        val_mask   = label_mask & np.isin(block_ids, val_blocks)

        train_idx = np.where(train_mask.ravel())[0]
        val_idx   = np.where(val_mask.ravel())[0]

        y_train_all = y_all[train_idx]
        uniq_train_classes = np.unique(y_train_all)

        if set(uniq_train_classes.tolist()) >= {0, 1}:
            print(f"   Split attempt {attempt+1}: OK (both classes in train)")
            break
        else:
            print(f"   Split attempt {attempt+1}: only classes {uniq_train_classes}, retrying...")
    else:
        raise RuntimeError("Could not find a train/val split with both classes in training.")

    # Build X and y
    X = X_full
    X_train = X[train_idx, :]
    y_train = y_all[train_idx]

    X_val   = X[val_idx, :]
    y_val   = y_all[val_idx]

    n_train = X_train.shape[0]
    n_val   = X_val.shape[0]
    print(f"   Train pixels: {n_train}, Val pixels: {n_val}, Val fraction: {n_val / (n_train + n_val):.3f}")
```

```
# Save the split raster for this run
split_map = np.zeros_like(data, dtype=np.uint8)
split_map[train_mask] = 1
split_map[val_mask]   = 2

split_profile = profile.copy()
split_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

split_raster_path = os.path.join(
    base_output_dir,
    f'train_val_split_run{run_id}.tif'
)
with rasterio.open(split_raster_path, 'w', **split_profile) as dst:
    dst.write(split_map, 1)
print(f"   Train/validation split raster saved to: {split_raster_path}")

# Save training block info
train_block_ids = np.unique(block_ids[train_mask])
train_sites_rows = []
for b in train_block_ids:
    mask_b = (block_ids == b) & label_mask
    flat_idx_b = np.where(mask_b.ravel())[0]
    labels_b = y_all[flat_idx_b]
    n_refugia = np.sum(labels_b == 1)
    n_nonrefugia = np.sum(labels_b == 0)
    train_sites_rows.append({
        'block_id': int(b),
        'n_labelled_pixels': int(len(flat_idx_b)),
        'n_refugia_pixels': int(n_refugia),
        'n_nonrefugia_pixels': int(n_nonrefugia)
    })

train_sites_df = pd.DataFrame(train_sites_rows)
train_sites_csv_path = os.path.join(
    base_output_dir,
    f'train_blocks_run{run_id}.csv'
)
train_sites_df.to_csv(train_sites_csv_path, index=False)
print(f"   Training blocks info saved to: {train_sites_csv_path}")

# ------------------------
# 6.2 ADASYN
# ------------------------
X_train_resampled, y_train_resampled = adasyn.fit_resample(X_train, y_train)
print("   After ADASYN - train samples:", X_train_resampled.shape[0])

# ------------------------
# 6.3 Instantiate fresh classifiers & soft-voting ensemble
# ------------------------
rf_clf_run = RandomForestClassifier(
    n_estimators=50,
    max_features='sqrt',
```

```
    max_depth=30,
    min_samples_split=2,
    min_samples_leaf=1,
    oob_score=True,
    random_state=80,
    n_jobs=-1
)
xgb_clf_run = xgb.XGBClassifier(
    n_estimators=1500,
    eta=0.1,
    max_depth=4,
    reg_alpha=0.005,
    reg_lambda=1,
    use_label_encoder=False,
    eval_metric='logloss',
    n_jobs=-1
)
knn_clf_run = KNeighborsClassifier(
    n_neighbors=3,
    weights='distance',
    metric='manhattan',
    n_jobs=-1
)

voting_clf = VotingClassifier(
    estimators=[('rf', rf_clf_run), ('knn', knn_clf_run), ('xgb', xgb_clf_run)],
    voting='soft',
    n_jobs=-1
)

# ------------------------
# 6.4 Fit + evaluate
# ------------------------
voting_clf.fit(X_train_resampled, y_train_resampled)

y_pred = voting_clf.predict(X_val)
y_pred_proba = voting_clf.predict_proba(X_val)[:, 1]

if np.isnan(y_pred_proba).any():
    y_pred_proba = np.nan_to_num(y_pred_proba)

accuracy = accuracy_score(y_val, y_pred)
precision = precision_score(y_val, y_pred, average='binary')
recall = recall_score(y_val, y_pred, average='binary')
f1 = f1_score(y_val, y_pred, average='binary')
cm = confusion_matrix(y_val, y_pred)
roc_auc = roc_auc_score(y_val, y_pred_proba)

print("   Unique predictions:", np.unique(y_pred))
print('   Accuracy:', accuracy)
print('   Precision:', precision)
print('   Recall:', recall)
print('   F1 Score:', f1)
```

```
    print('    AUC-ROC:', roc_auc)
    print('    Confusion Matrix:\n', cm)

    if run_id == 1:
        plot_confusion_matrix(cm, title_suffix="(run 1)")

    run_metrics = {
        'run': run_id,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1': f1,
        'AUC_ROC': roc_auc,
        'Train_pixels': int(n_train),
        'Val_pixels': int(n_val)
    }

    all_run_metrics.append(run_metrics)

    # ------------------------
    # 6.5 Save per-run prediction raster
    # ------------------------
    full_predictions = voting_clf.predict(X_full)              # 0/1
    full_pred_map = full_predictions.reshape((height, width))

    run_pred_path = os.path.join(
        base_output_dir,
        f'EnsembleSoft_experiment_run{run_id}.tif'
    )
    with rasterio.open(run_pred_path, 'w', **pred_profile) as dst:
        dst.write(full_pred_map.astype(rasterio.uint8), 1)

    print(f"    Per-run prediction raster saved to: {run_pred_path}")

    # ------------------------
    # 6.6 Accumulate refugia probabilities for final probability map
    # ------------------------
    full_proba = voting_clf.predict_proba(X_full)[:, 1]
    full_proba_map = full_proba.reshape((height, width))
    sum_refugia_proba += full_proba_map.astype(np.float32)

# ------------------------
# 7. Metrics
# ------------------------
metrics_df = pd.DataFrame(all_run_metrics)

metrics_csv_path = os.path.join(
    base_output_dir,
    'EnsembleSoft_all_runs_metrics_spatial_splits_only.csv'
)
metrics_df.to_csv(metrics_csv_path, index=False)
print("\nPer-run metrics saved to:", metrics_csv_path)
```

```
summary_rows = []
metric_names = ['Accuracy', 'Precision', 'Recall', 'F1', 'AUC_ROC']
n = n_runs

for m in metric_names:
    vals = metrics_df[m].values
    mean = np.mean(vals)
    std = np.std(vals, ddof=1)
    ci_half_width = 1.96 * std / np.sqrt(n)
    ci_low = mean - ci_half_width
    ci_high = mean + ci_half_width

    summary_rows.append({
        'Metric': m,
        'Mean': mean,
        'Std': std,
        'CI_lower_95': ci_low,
        'CI_upper_95': ci_high
    })

summary_df = pd.DataFrame(summary_rows)
summary_csv_path = os.path.join(
    base_output_dir,
    'EnsembleSoft_metrics_summary_spatial_splits_only.csv'
)
summary_df.to_csv(summary_csv_path, index=False)
print("Summary metrics (mean/std/CI) saved to:", summary_csv_path)

print("\nSummary metrics:")
print(summary_df)

# ------------------------
# 8. Refugia probability map (0–1:)
# ------------------------
refugia_prob_map = sum_refugia_proba / float(n_runs)

prob_profile = pred_profile.copy()
prob_profile.update(dtype=rasterio.float32, count=1, compress='lzw')

prob_raster_path = os.path.join(
    base_output_dir,
    'EnsembleSoft_refugia_probability_0_1_spatial_splits_only.tif'
)

with rasterio.open(prob_raster_path, 'w', **prob_profile) as dst:
    dst.write(refugia_prob_map.astype(rasterio.float32), 1)

print("Refugia probability raster saved to:", prob_raster_path)
```

Appendix S8 Ensemble hard script

```
import os    # hard Ensemble with ADASYN and spatial splits
import rasterio
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.neighbors import KNeighborsClassifier
import xgboost as xgb
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, roc_auc_score
)
from sklearn.impute import SimpleImputer
from imblearn.over_sampling import ADASYN
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import csv

# ---------------------------
# 1. Load raster with training and validation data
# ---------------------------
dataset_path = r'location to training and validation dataset'

with rasterio.open(dataset_path) as src:
    data = src.read(1)
    profile = src.profile
    height, width = src.shape

# Keep only classes 1 and 2; everything else = 0 (unlabelled)
data[(data != 1) & (data != 2)] = 0

# ---------------------------
# 2. Load feature rasters
# ---------------------------
raster_files = [
  r'location to elevation',
  r'location to aspect',
  r'location to diffuse horizntal irradiation',
  r'location to direct normal irradiation',
  r'location to global horizontal irradiation',
  r'location to slope',
  r'location to cfd wind direction',
  r'location to cfd wind speed',
  r'location to temperature',
  r'location to topographical covergence',
  r'location to topographical roughness',
  r'location to topographical roughness index
]

rasters = [rasterio.open(file).read(1) for file in raster_files]

# Stack rasters into [n_pixels, n_features]
```

```
X_full = np.column_stack([raster.ravel() for raster in rasters])

# Handle missing values in X_full
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
X_full = imputer.fit_transform(X_full)

raster_names = [os.path.splitext(os.path.basename(file))[0] for file in raster_files]

print("X_full shape (all pixels):", X_full.shape)

# --------------------------------------
# 3. Labels & spatial block setup
# --------------------------------------

# label_mask: pixels with 1 (refugia) or 2 (non-refugia)
label_mask = data != 0

# Encode labels: refugia = 1, non-refugia = 0, others = -1 to keep classifier clean and avoid classifier errors that oc-
        cured
flat_data = data.ravel()
y_all = np.full(flat_data.shape, -1, dtype=int)
y_all[flat_data == 1] = 1     # refugia
y_all[flat_data == 2] = 0     # non-refugia

rows, cols = np.indices((height, width))

# Block size for selection
tile_h, tile_w = 64, 64

block_row = rows // tile_h
block_col = cols // tile_w

n_block_rows = int(np.ceil(height / tile_h))
n_block_cols = int(np.ceil(width   / tile_w))

block_ids = block_row * n_block_cols + block_col   # unique ID per block

# Only blocks with at least one labelled pixel
block_ids_labelled = block_ids[label_mask]
unique_blocks = np.unique(block_ids_labelled)

# ADASYN setup
adasyn = ADASYN(random_state=80)
# Use elevation raster to keep output uniform
with rasterio.open(
    r'location to elevation'
) as src0:
    pred_profile = src0.profile
pred_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

# Output base directory
base_output_dir = r'location to store output'
os.makedirs(base_output_dir, exist_ok=True)
```

```
# ------------------------
# 4. Confusion matrix plot
# ------------------------
def plot_confusion_matrix(cm, title_suffix=""):
    plt.figure(figsize=(8, 6))
    sns.heatmap(
        cm, annot=True, fmt='g', cmap='Blues',
        xticklabels=['Refugia', 'Not Refugia'],
        yticklabels=['Refugia', 'Not Refugia']
    )
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(f'Confusion Matrix {title_suffix}')
    plt.tight_layout()
    plt.show()

# ------------------------
# 5. Base classifiers (use best parameters from individual runs)
# ------------------------
rf_clf = RandomForestClassifier(
    n_estimators=500,
    max_features='sqrt',
    max_depth=30,
    min_samples_split=2,
    min_samples_leaf=1,
    oob_score=True,
    random_state=80,
    n_jobs=-1
)

xgb_clf = xgb.XGBClassifier(
    n_estimators=1500,
    eta=0.5,
    max_depth=7,
    reg_alpha=0.001,
    reg_lambda=0.5,
    use_label_encoder=False,
    eval_metric='logloss',
    n_jobs=-1
)

knn_clf = KNeighborsClassifier(
    n_neighbors=3,
    weights='distance',
    metric='manhattan',
    n_jobs=-1
)

voting_clf_template = VotingClassifier(
    estimators=[('rf', rf_clf), ('knn', knn_clf), ('xgb', xgb_clf)],
    voting='hard',
    n_jobs=-1
```

```
)

# ------------------------
# 6. Spatial split only
# ------------------------
n_runs = 10
val_fraction = 0.30
max_split_tries = 50

all_run_metrics = []

sum_refugia_preds = np.zeros((height, width), dtype=np.float32)

# ------------------------
# Loop over runs
# ------------------------
for i in range(n_runs):
    run_id = i + 1
    print(f"\n=== Run {run_id}/{n_runs} - Ensemble (RF+KNN+XGB) ===")

    # ------------------------
    # 6.1 New spatial block split for this run
    # ------------------------
    rng_blocks = np.random.RandomState(1000 + run_id)

    for attempt in range(max_split_tries):
        n_val_blocks = max(1, int(np.round(val_fraction * len(unique_blocks))))
        val_blocks = rng_blocks.choice(unique_blocks, size=n_val_blocks, replace=False)
        train_blocks = np.setdiff1d(unique_blocks, val_blocks)

        train_mask = label_mask & np.isin(block_ids, train_blocks)
        val_mask   = label_mask & np.isin(block_ids, val_blocks)

        train_idx = np.where(train_mask.ravel())[0]
        val_idx   = np.where(val_mask.ravel())[0]

        y_train_all = y_all[train_idx]
        uniq_train_classes = np.unique(y_train_all)

        if set(uniq_train_classes.tolist()) >= {0, 1}:
            print(f"   Split attempt {attempt+1}: OK (both classes in train)")
            break
        else:
            print(f"   Split attempt {attempt+1}: only classes {uniq_train_classes}, retrying...")
    else:
        raise RuntimeError("Could not find a train/val split with both classes in training.")

    # Build X and y
    X = X_full
    X_train = X[train_idx, :]
    y_train = y_all[train_idx]

    X_val   = X[val_idx, :]
```

```
y_val     = y_all[val_idx]

n_train = X_train.shape[0]
n_val     = X_val.shape[0]
print(f"    Train pixels: {n_train}, Val pixels: {n_val}, Val fraction: {n_val / (n_train + n_val):.3f}")

# Save the split raster for this run
split_map = np.zeros_like(data, dtype=np.uint8)
split_map[train_mask] = 1
split_map[val_mask]     = 2

split_profile = profile.copy()
split_profile.update(dtype=rasterio.uint8, count=1, compress='lzw')

split_raster_path = os.path.join(
    base_output_dir,
    f'train_val_split_run{run_id}.tif'
)
with rasterio.open(split_raster_path, 'w', **split_profile) as dst:
    dst.write(split_map, 1)
print(f"    Train/validation split raster saved to: {split_raster_path}")

# Save training
train_block_ids = np.unique(block_ids[train_mask])
train_sites_rows = []
for b in train_block_ids:
    mask_b = (block_ids == b) & label_mask
    flat_idx_b = np.where(mask_b.ravel())[0]
    labels_b = y_all[flat_idx_b]
    n_refugia = np.sum(labels_b == 1)
    n_nonrefugia = np.sum(labels_b == 0)
    train_sites_rows.append({
        'block_id': int(b),
        'n_labelled_pixels': int(len(flat_idx_b)),
        'n_refugia_pixels': int(n_refugia),
        'n_nonrefugia_pixels': int(n_nonrefugia)
    })

train_sites_df = pd.DataFrame(train_sites_rows)
train_sites_csv_path = os.path.join(
    base_output_dir,
    f'train_blocks_run{run_id}.csv'
)
train_sites_df.to_csv(train_sites_csv_path, index=False)
print(f"    Training blocks info saved to: {train_sites_csv_path}")

# ------------------------
# 6.2 ADASYN
# ------------------------
X_train_resampled, y_train_resampled = adasyn.fit_resample(X_train, y_train)
print("    After ADASYN - train samples:", X_train_resampled.shape[0])

# ------------------------
```

```
# 6.3 Clone ensemble
# ------------------------
rf_clf_run = RandomForestClassifier(
    n_estimators=50,
    max_features='sqrt',
    max_depth=30,
    min_samples_split=2,
    min_samples_leaf=1,
    oob_score=True,
    random_state=80,
    n_jobs=-1
)
xgb_clf_run = xgb.XGBClassifier(
    n_estimators=1500,
    eta=0.1,
    max_depth=4,
    reg_alpha=0.005,
    reg_lambda=1,
    use_label_encoder=False,
    eval_metric='logloss',
    n_jobs=-1
)
knn_clf_run = KNeighborsClassifier(
    n_neighbors=3,
    weights='distance',
    metric='manhattan',
    n_jobs=-1
)

voting_clf = VotingClassifier(
    estimators=[('rf', rf_clf_run), ('knn', knn_clf_run), ('xgb', xgb_clf_run)],
    voting='hard',
    n_jobs=-1
)

# ------------------------
# 6.4 Fit and evaluate
# ------------------------
voting_clf.fit(X_train_resampled, y_train_resampled)

y_pred = voting_clf.predict(X_val)

# get fitted sub-estimators from the voting classifier
rf_fitted   = voting_clf.named_estimators_['rf']
xgb_fitted = voting_clf.named_estimators_['xgb']
knn_fitted = voting_clf.named_estimators_['knn']

proba_list = [
    rf_fitted.predict_proba(X_val)[:, 1],
    xgb_fitted.predict_proba(X_val)[:, 1],
    knn_fitted.predict_proba(X_val)[:, 1]
]
y_pred_proba = np.mean(proba_list, axis=0)
```

```
if np.isnan(y_pred_proba).any():
    y_pred_proba = np.nan_to_num(y_pred_proba)

accuracy = accuracy_score(y_val, y_pred)
precision = precision_score(y_val, y_pred, average='binary')
recall = recall_score(y_val, y_pred, average='binary')
f1 = f1_score(y_val, y_pred, average='binary')
cm = confusion_matrix(y_val, y_pred)
roc_auc = roc_auc_score(y_val, y_pred_proba)

print("   Unique predictions:", np.unique(y_pred))
print('   Accuracy:', accuracy)
print('   Precision:', precision)
print('   Recall:', recall)
print('   F1 Score:', f1)
print('   AUC-ROC:', roc_auc)
print('   Confusion Matrix:\n', cm)

if run_id == 1:
    plot_confusion_matrix(cm, title_suffix="(run 1)")

# Per-run metrics
run_metrics = {
    'run': run_id,
    'Accuracy': accuracy,
    'Precision': precision,
    'Recall': recall,
    'F1': f1,
    'AUC_ROC': roc_auc,
    'Train_pixels': int(n_train),
    'Val_pixels': int(n_val)
}

all_run_metrics.append(run_metrics)

# ------------------------
# 6.5 Save raster
# ------------------------
full_predictions = voting_clf.predict(X_full)
full_pred_map = full_predictions.reshape((height, width))

run_pred_path = os.path.join(
    base_output_dir,
    f'EnsembleHard_experiment_run{run_id}.tif'
)
with rasterio.open(run_pred_path, 'w', **pred_profile) as dst:
    dst.write(full_pred_map.astype(rasterio.uint8), 1)

print(f"   Per-run prediction raster saved to: {run_pred_path}")

# For probability map: accumulate refugia predictions (class 1) across runs
sum_refugia_preds += full_pred_map.astype(np.float32)
```

```
# ------------------------
# 7. Metrics
# ------------------------
metrics_df = pd.DataFrame(all_run_metrics)

metrics_csv_path = os.path.join(
    base_output_dir,
    'EnsembleHard_all_runs_metrics_spatial_splits_only.csv'
)
metrics_df.to_csv(metrics_csv_path, index=False)
print("\nPer-run metrics saved to:", metrics_csv_path)

summary_rows = []
metric_names = ['Accuracy', 'Precision', 'Recall', 'F1', 'AUC_ROC']
n = n_runs

for m in metric_names:
    vals = metrics_df[m].values
    mean = np.mean(vals)
    std = np.std(vals, ddof=1)
    ci_half_width = 1.96 * std / np.sqrt(n)
    ci_low = mean - ci_half_width
    ci_high = mean + ci_half_width

    summary_rows.append({
        'Metric': m,
        'Mean': mean,
        'Std': std,
        'CI_lower_95': ci_low,
        'CI_upper_95': ci_high
    })

summary_df = pd.DataFrame(summary_rows)
summary_csv_path = os.path.join(
    base_output_dir,
    'EnsembleHard_metrics_summary_spatial_splits_only.csv'
)
summary_df.to_csv(summary_csv_path, index=False)
print("Summary metrics (mean/std/CI) saved to:", summary_csv_path)

print("\nSummary metrics:")
print(summary_df)

# ------------------------
# 8. Refugia probability map (0–1: fraction of runs predicted as refugia)
# ------------------------
refugia_prob_map = sum_refugia_preds / float(n_runs)

prob_profile = pred_profile.copy()
prob_profile.update(dtype=rasterio.float32, count=1, compress='lzw')

prob_raster_path = os.path.join(
```

```
    base_output_dir,
    'EnsembleHard_refugia_probability_0_1_spatial_splits_only.tif'
)

with rasterio.open(prob_raster_path, 'w', **prob_profile) as dst:
    dst.write(refugia_prob_map.astype(rasterio.float32), 1)

print("Refugia probability raster saved to:", prob_raster_path)
```

Appendix S9. Experimental design showing experiment number, dataset and oversampling, X indicate dataset was used in experiments

Method	Experiment no	Elevation	Temperature	Diffuse horizontal irradiation	Direct normal irradiation	Global horizontal irradiation	CFD Speed	CFD Direction	Aspect	Slope	Topographical convergence	Topographical roughness	Topographical wetness index	ADASYN oversampling
RF	1	x	x	x	x	x	x	x	x	x	x	x	x	N
	2	x	x	x	x	x	x	x	x	x	x	x	x	Y
	3		x	x	x	x	x	x	x	x	x	x	x	Y
	4			x	x	x	x	x	x	x	x	x	x	Y
	5						x	x	x	x	x	x	x	Y
	6								x	x	x	x	x	Y
XGBoost	7	x	x	x	x	x	x	x	x	x	x	x	x	N
	8	x	x	x	x	x	x	x	x	x	x	x	x	Y
	9		x	x	x	x	x	x	x	x	x	x	x	Y
	10			x	x	x	x	x	x	x	x	x	x	Y
	11						x	x	x	x	x	x	x	Y
	12								x	x	x	x	x	Y
KNN	13	x	x	x	x	x	x	x	x	x	x	x	x	N
	14	x	x	x	x	x	x	x	x	x	x	x	x	Y
	15		x	x	x	x	x	x	x	x	x	x	x	Y
	16			x	x	x	x	x	x	x	x	x	x	Y
	17						x	x	x	x	x	x	x	Y
	18								x	x	x	x	x	Y
Ensemble (soft)	19	x	x	x	x	x	x	x	x	x	x	x	x	N
	20	x	x	x	x	x	x	x	x	x	x	x	x	Y
	21		x	x	x	x	x	x	x	x	x	x	x	Y
	22			x	x	x	x	x	x	x	x	x	x	Y
	23						x	x	x	x	x	x	x	Y
	24								x	x	x	x	x	Y
Ensemble (hard)	25	x	x	x	x	x	x	x	x	x	x	x	x	N
	26	x	x	x	x	x	x	x	x	x	x	x	x	Y
	27		x	x	x	x	x	x	x	x	x	x	x	Y
	28			x	x	x	x	x	x	x	x	x	x	Y
	29						x	x	x	x	x	x	x	Y
	30								x	x	x	x	x	Y

Appendix 10 Best parameters per experiment

Experiment no	Method	max depth	max features	min samples leaf	min samples split	n estimators	oob score
1	RF	None	sqrt	4	2	200	TRUE
2	RF	30	sqrt	1	2	50	TRUE
3	RF	None	sqrt	1	2	500	TRUE
4	RF	None	sqrt	1	2	200	TRUE
5	RF	None	sqrt	1	2	500	TRUE
6	RF	30	sqrt	1	2	500	TRUE

Experiment no	Method	n estimators	max depth	reg alpha	reg lambda
7	XGboost	50	3	0.05	2
8	XGboost	1500	4	0.005	1
9	XGboost	1500	5	0	1.5
10	XGboost	100	7	0.001	1
11	XGboost	1500	7	0.05	1.5
12	XGboost	1500	7	0.001	0.5

Experiment no	Method	n neighbors	weights
13	KNN	5	uniform
14	KNN	3	distance
15	KNN	3	distance
16	KNN	3	distance
17	KNN	3	distance
18	KNN	3	distance