1 Machine Learning Design Methodology

1.1 Data Preprocessing

- Data loading: We load the raw and true color satellite images using a library called `rasterio`.
- Data extraction: We extract the individual RGB channels from both images.
- Data normalization: We normalize the pixel values to the range [0, 1] to ensure consistent scaling.

1.2 Resampling Method

- We've used the `train_test_split` function from scikit-learn to split our dataset into training and testing sets.
- This method randomly splits the data into two subsets (80% training, 20% testing) while ensuring that the class distribution remains consistent.

1.3 Model Selection

- In our code, we've chosen to implement two types of models:
- **Linear Regression:** A simple baseline model for regression tasks.
- **Neural Network (Keras with TensorFlow):** A more complex model capable of learning non-linear relationships.
- Model selection is based on the problem's complexity and requirements. Linear regression is straightforward and interpretable, while neural networks can capture complex patterns.

1.4 Model Performance Metrics

- **Mean Squared Error (MSE):** We've chosen MSE as the primary performance metric.
- MSE measures the average squared difference between predicted and actual values. Lower MSE indicates better model performance.
- MSE is a common choice for regression tasks, but we could also consider other metrics like Mean Absolute Error (MAE) or R-squared (R2) depending on our specific goals.

1.5 Neural Network Parameters (Keras with TensorFlow)

- **Model Architecture:** We've chosen a fully-connected feedforward neural network with:
- An input layer with 3 neurons (for the RGB channels).
- First hidden layer with 64 neurons followed by two hidden layers with 32 neurons each and ReLU activation functions.
- An output layer with 3 neurons (for RGB values) and a linear activation function.
- **Optimizer:** We've used the Adam optimizer, a popular choice for gradient-based optimization.
- **Loss Function:** We've used Mean Squared Error (MSE) as the loss function, which is appropriate for regression tasks.
- **Training Parameters:**

- We've trained for 10 epochs (we may need more epochs depending on the convergence behavior).
- A batch size of 32 indicates that the model's weights are updated after processing 32 samples.
- A validation split of 0.2 means that 20% of the training data is used for validation during training.

1.6 Saving and Model Evaluation

- We save the trained model to a pickle file for future use.
- After training, we evaluate the model's performance on the test dataset using the chosen metric (MSE).

1.7 Further Considerations

- Depending on the complexity of our data and the desired level of accuracy, we may need to experiment with different neural network architectures, hyperparameters (e.g., learning rate, number of hidden layers, neurons per layer), and regularization techniques.
- We may also want to consider cross-validation for more robust model evaluation, especially if our dataset is limited.

2 Regularization in Linear Regression

2.1 How Regularization Affects Linear Regression

Regularization modifies the behavior of a linear regression model by adding a penalty term to the standard least squares loss function. The primary goal of regularization is to control model complexity and prevent overfitting. Here's how it affects the model:

- **Magnitude of Coefficients:** Regularization reduces the magnitude of the coefficients. In Ridge (L2) regularization, it does so by adding the sum of squared coefficients to the loss function, while Lasso (L1) regularization uses the sum of absolute values of coefficients. This means that the coefficients become smaller, and some may become exactly zero in Lasso.
- **Bias-Variance Trade-Off:** Regularization introduces a bias into the model by shrinking the coefficients. While this bias may slightly reduce the model's ability to fit the training data perfectly, it often leads to a better ability to generalize to unseen data, reducing the model's variance and overfitting tendencies.
- **Feature Selection (Lasso):** Lasso regularization can perform feature selection by driving some coefficients to exactly zero. This means that Lasso can identify and exclude irrelevant or less important features from the model, leading to a simpler and potentially more interpretable model.

2.2 Reasons to Use Regularization

- **Preventing Overfitting:** Regularization is crucial when the linear regression model shows signs of overfitting, which occurs when the model fits the training data too

closely and struggles to generalize to new data. Regularization helps combat overfitting by reducing the complexity of the model.

- **Handling Multicollinearity:** Regularization techniques like Ridge can handle multicollinearity, where predictor variables are highly correlated. By shrinking the coefficients, Ridge can stabilize the model in the presence of multicollinearity.

- **Feature Selection (Lasso):** If you suspect that many of the features in your dataset are irrelevant or redundant, Lasso can automatically select a subset of the most informative features, leading to a more parsimonious model.

2.3 Reasons Not to Use Regularization

- **Simplicity:** In some cases, a simple linear regression model without regularization may be sufficient, especially when the dataset is small, and there's no evidence of overfitting.

- **Interpretability:** Regularized models, particularly Lasso, can make interpretation more challenging because they may set some coefficients to zero. If interpretability is a top priority, a non-regularized linear regression model might be preferred.

- **Complex Models:** For very large datasets with many features, regularization may not be necessary if the risk of overfitting is low. In such cases, a simple linear regression model might suffice.

3 Linear Regression vs. Neural Networks

3.1 Differences

Linearity vs. Non-Linearity:
- *Linear Regression:* Assumes a linear relationship between features and the output.
- *Neural Network:* Can model both linear and non-linear relationships.

Model Complexity:
- *Linear Regression:* Simple and interpretable with limited capacity.
- *Neural Network:* Complex and flexible with the ability to capture intricate patterns.

Feature Engineering:
- *Linear Regression:* Requires manual feature engineering for meaningful input features.
- *Neural Network:* Can learn feature representations from raw data.

Model Training:
- *Linear Regression:* Can have closed-form solutions (simple cases) or use gradient descent.
- *Neural Network:* Trained using gradient-based optimization, requiring hyperparameter tuning.

Interpretability:
- *Linear Regression:* Provides interpretable coefficients for feature impacts.
- *Neural Network:* Often considered a "black-box" model, less interpretable.

3.2 Tradeoff

The choice between linear regression and neural networks involves a tradeoff:

Linear Regression:

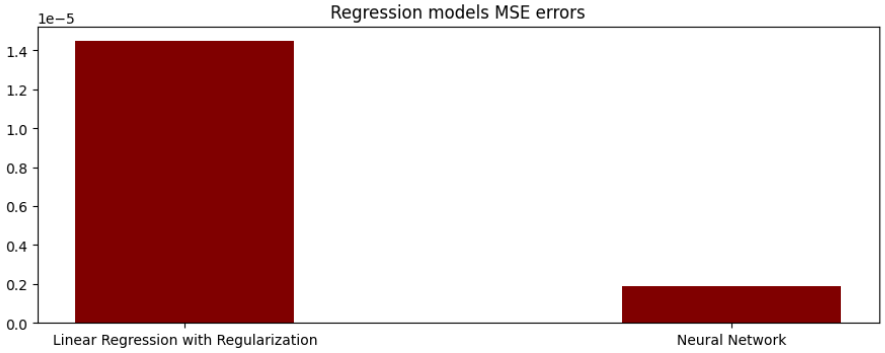
- *Advantages:* Interpretability, simplicity, and efficiency.
- *Limitations:* Limited modeling capacity for complex relationships.

Neural Networks:

- *Advantages:* High complexity, feature learning, versatility.
- *Limitations:* Reduced interpretability and increased computational requirements.

Tradeoff: The tradeoff is primarily between model complexity and interpretability. Linear regression offers simplicity and interpretability but may not capture complex patterns. Neural networks provide high modeling capacity but are less interpretable.

The choice depends on the problem, dataset size, computational resources, and the need for interpretability. A balance between complexity and interpretability is often sought.



Exploring Raster Data

We explore various attributes of the raster data using the `rasterio` library.

[ ]:
```
%pip install rasterio
```

[2]:
```
import os
import rasterio
from rasterio.plot import show
from rasterio.plot import show_hist
import numpy as np
import matplotlib.pyplot as plt
```

[5]:
```
raw_raster = rasterio.open("raw_RGB_image.tif")
true_raster = rasterio.open("true_color_RGB_image.tif")
```

Check type of the variable 'raster'

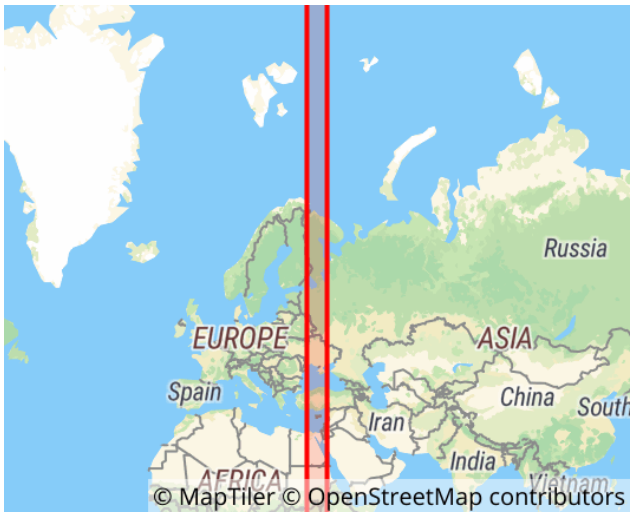
[6]:
```
type(raw_raster)
```

[6]:
```
rasterio.io.DatasetReader
```

Projection

[7]:
```
raw_raster.crs  # outs EPSG:32636 which includes Turkey
```

[7]:
```
CRS.from_epsg(32636)
```



Affine transform (how raster is scaled, rotated, skewed, and/or translated)

[8]:
```
raw_raster.transform
```

[8]:
```
Affine(10.0, 0.0, 739440.0,
       0.0, -10.0, 4615020.0)
```

Raster width and height

[9]:
```
raw_raster.width
```

[9]:
```
804
```

[10]:
```
raw_raster.height
```

[10]:
```
693
```

Number of bands

[11]:
```
raw_raster.count
```

[11]:
```
3
```

Bounds of the file

[12]:
```
raw_raster.bounds
```

[12]:
```
BoundingBox(left=739440.0, bottom=4608090.0, right=747480.0, top=4615020.0)
```

[13]:
```
raw_raster.bounds.right - raw_raster.bounds.left # width*10
```

[13]:
```
8040.0
```

[14]:
```
raw_raster.bounds.top - raw_raster.bounds.bottom # height*10
```

[14]:
```
6930.0
```

Driver (data format)

[15]:
```
raw_raster.driver
```

[15]:
```
'GTiff'
```

No data values for all channels

[16]:
```
raw_raster.nodatavals
```

[16]:
```
(None, None, None)
```

All Metadata for the raster

[17]:
```
raw_raster.meta
```

[17]:
```
{'driver': 'GTiff',
 'dtype': 'uint16',
 'nodata': None,
 'width': 804,
 'height': 693,
 'count': 3,
 'crs': CRS.from_epsg(32636),
```

```
 'transform': Affine(10.0, 0.0, 739440.0,
        0.0, -10.0, 4615020.0)}
```

RGB bands of the satellite image is stacked together in one raster dataset.

Read the raster band as separate variable

[18]:
```
band1 = raw_raster.read(1)
# Check type of the variable 'band'
type(band1) # <class 'numpy.ndarray'>
```

[18]:
```
numpy.ndarray
```

Data type of the values

[19]:
```
band1.dtype # uint16
```

[19]:
```
dtype('uint16')
```

An unsigned 16-bit integer can store 2^16 (=65,536) values ranging from 0 to 65,535.

Satellite optic channels read between 0 and ~7000.

Read all bands

[20]:
```
array = raw_raster.read()
```

Calculate statistics for each band

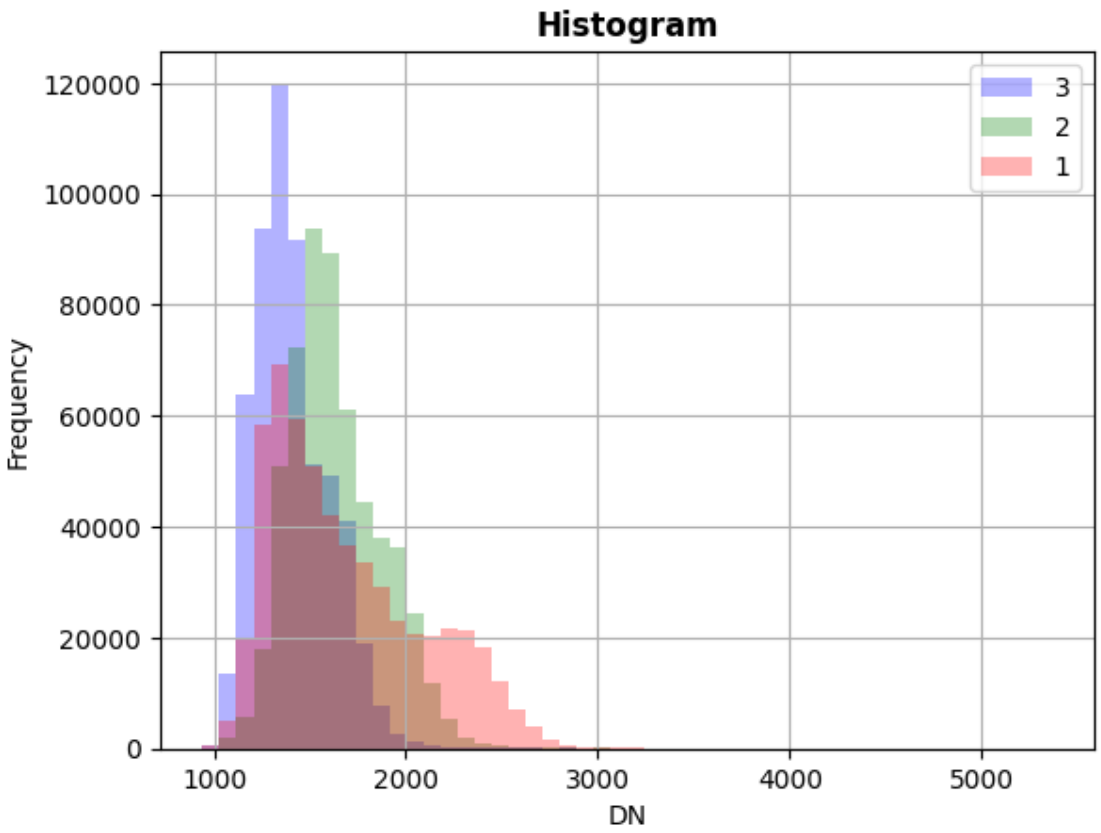
[21]:
```
stats = []
for band in array:
    stats.append({
        'min': band.min(),
        'mean': band.mean(),
        'median': np.median(band),
        'max': band.max()})

stats
```

[21]:
```
[{'min': 994, 'mean': 1690.3487486808383, 'median': 1588.0, 'max': 5364},
 {'min': 997, 'mean': 1632.349568894345, 'median': 1593.0, 'max': 5176},
 {'min': 940, 'mean': 1408.0579659422942, 'median': 1374.0, 'max': 5104}]
```

Channels and corresponding wavelengths in nanometers Red (620 - 750) Green (495 - 570) Blue (450 - 495)

Let's see the band distributions in the raw image.

[22]:
```
show_hist(raw_raster, bins=50, lw=0.0, stacked=False, alpha=0.3,␣
↪histtype='stepfilled', title="Histogram", label = ("1","2","3"))
```


Histogram
120000
100000
80000
60000
40000
20000
0
Frequency
1000
2000
3000
4000
5000
DN
3
2
1