Αναγνώριση Προτύπων 2024-2025

Εκπονήθηκε από τους μαθητές:
Λάμπρο Αλεξανδρή(Π22007)

Νικόλαο Πηλιχό(Π22144)

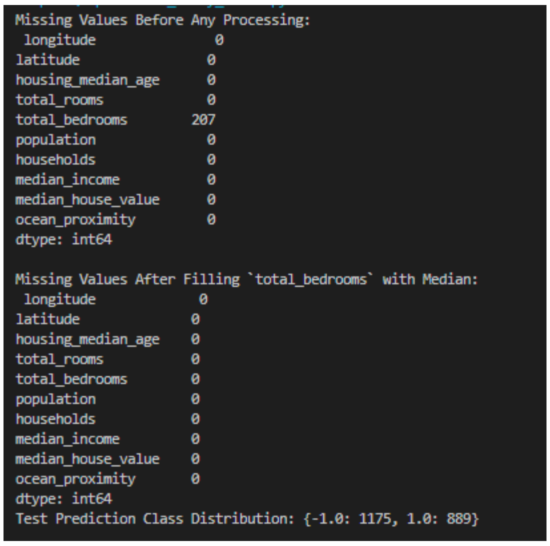
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

Data pre processing :

In this stage of the implementation the data from the given csv are processed in a way that helps the model training process.

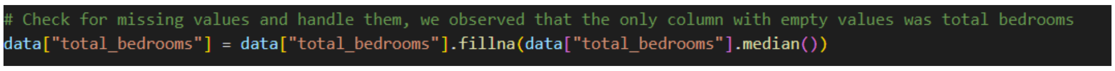
1. Missing values :

At this point of the pre processing we check the csv for missing values (after examining the csv we came to the conclusion that only the column named " total bedrooms " has some of its values missing so we narrowed the search down to that particular column only.) if a missing value is encounter then it is replaced with the median of the column. We confirm the change by printing the missing values before and after to confirm their replacement .



```
Missing Values Before Any Processing:
 longitude             0
latitude              0
housing_median_age    0
total_rooms           0
total_bedrooms      207
population            0
households            0
median_income         0
median_house_value    0
ocean_proximity       0
dtype: int64

Missing Values After Filling `total_bedrooms` with Median:
 longitude           0
latitude            0
housing_median_age  0
total_rooms         0
total_bedrooms      0
population          0
households          0
median_income       0
median_house_value  0
ocean_proximity     0
dtype: int64
Test Prediction Class Distribution: {-1.0: 1175, 1.0: 889}
```

Code :

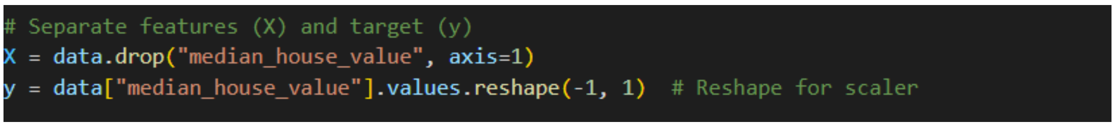


```python
# Check for missing values and handle them, we observed that the only column with empty values was total bedrooms
data["total_bedrooms"] = data["total_bedrooms"].fillna(data["total_bedrooms"].median())
```

2. Separation:

After filling the missing values we split the columns into Features and values, values being only the "median house value ".

Code :

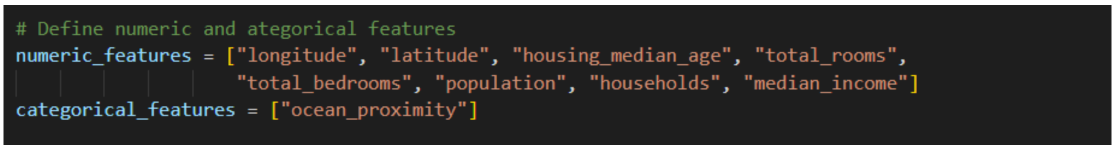


```python
# Separate features (X) and target (y)
X = data.drop("median_house_value", axis=1)
y = data["median_house_value"].values.reshape(-1, 1)  # Reshape for scaler
```

3. Feature categorization:

Then the features are split into numeric and categorical based on the data type they are represented. All features but “ ocean proximity are classed as numeric “ .

Code :

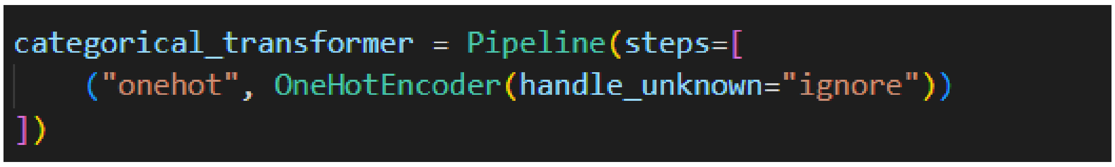


```python
# Define numeric and ategorical features
numeric_features = ["longitude", "latitude", "housing_median_age", "total_rooms",
                    "total_bedrooms", "population", "households", "median_income"]
categorical_features = ["ocean_proximity"]
```

4. Encoding:

In the case of ocean proximity the feature must be encoded so it can be processed by the algorithm. The encoding selected is one hot as the values assigned in this column can be represented by a 3X1 vector.

Code :

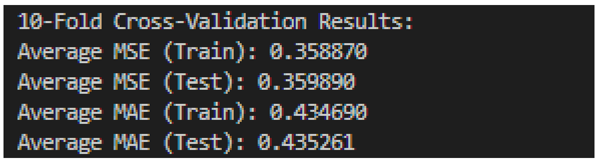


```python
categorical_transformer = Pipeline(steps=[
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])
```

5. Scaling:

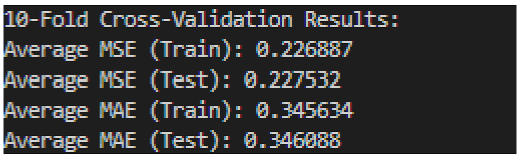
For the scaling of the data we experimented with three scalers, Min – Max scaler, Robust and Standard. After comparing the results of every algorithm, we came to the conclusion that min max scaling is the better option between the three based on MSE and MAE metrics. For the remainder of the tasks the scaler used is the Min Max one.

Standard Results:



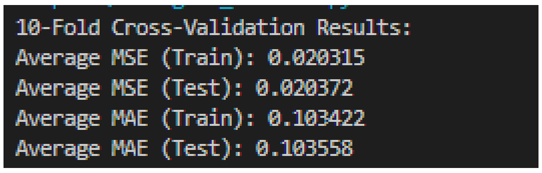
```
10-Fold Cross-Validation Results:
Average MSE (Train): 0.358870
Average MSE (Test): 0.359890
Average MAE (Train): 0.434690
Average MAE (Test): 0.435261
```

Robust Results :



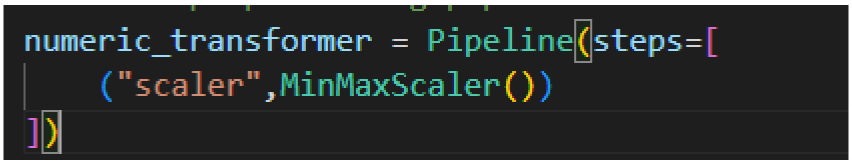
```
10-Fold Cross-Validation Results:
Average MSE (Train): 0.226887
Average MSE (Test): 0.227532
Average MAE (Train): 0.345634
Average MAE (Test): 0.346088
```

Min – Max Results:



```
10-Fold Cross-Validation Results:
Average MSE (Train): 0.020315
Average MSE (Test): 0.020372
Average MAE (Train): 0.103422
Average MAE (Test): 0.103558
```

Code :

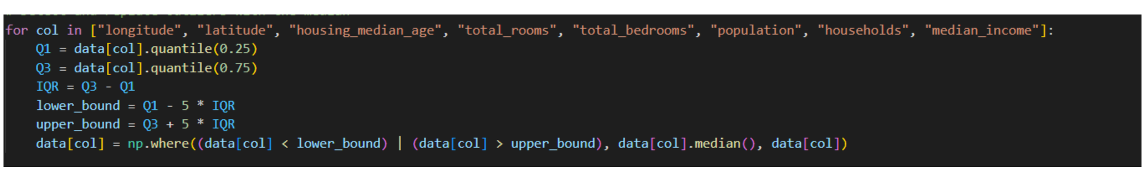


```
numeric_transformer = Pipeline(steps=[
    ("scaler",MinMaxScaler())
])
```

6. Outlier Handling:

In the case of values that are abnormally different from the median we tried to use a handling mechanism in which the value is replaced with the median.

Code :



```
for col in ["longitude", "latitude", "housing_median_age", "total_rooms", "total_bedrooms", "population", "households", "median_income"]:
    Q1 = data[col].quantile(0.25)
    Q3 = data[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 5 * IQR
    upper_bound = Q3 + 5 * IQR
    data[col] = np.where((data[col] < lower_bound) | (data[col] > upper_bound), data[col].median(), data[col])
```

Although we expected an improved performance, the metrics observed were slightly worse after adding the code above so in the final version outlier handling was removed

Data Visualization:

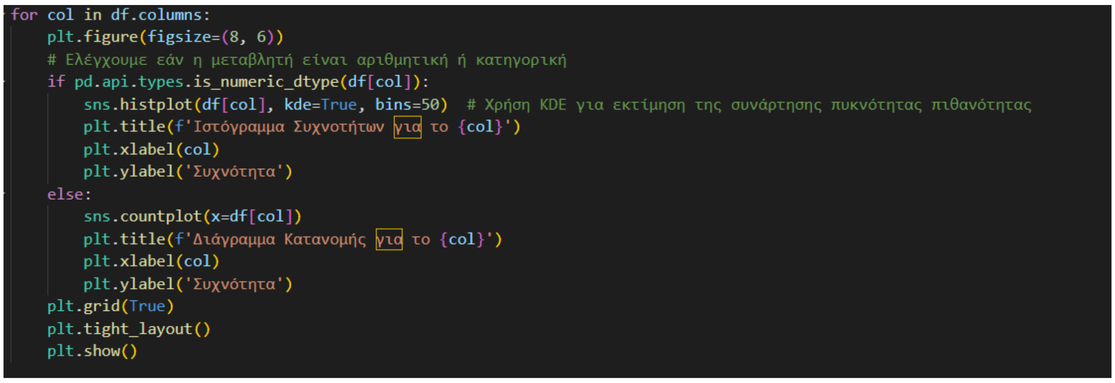
For the data visualization we implemented four separate code parts each responsible for representing features from one at the time to four together.

1.Frequency Histograms :

Explanation :

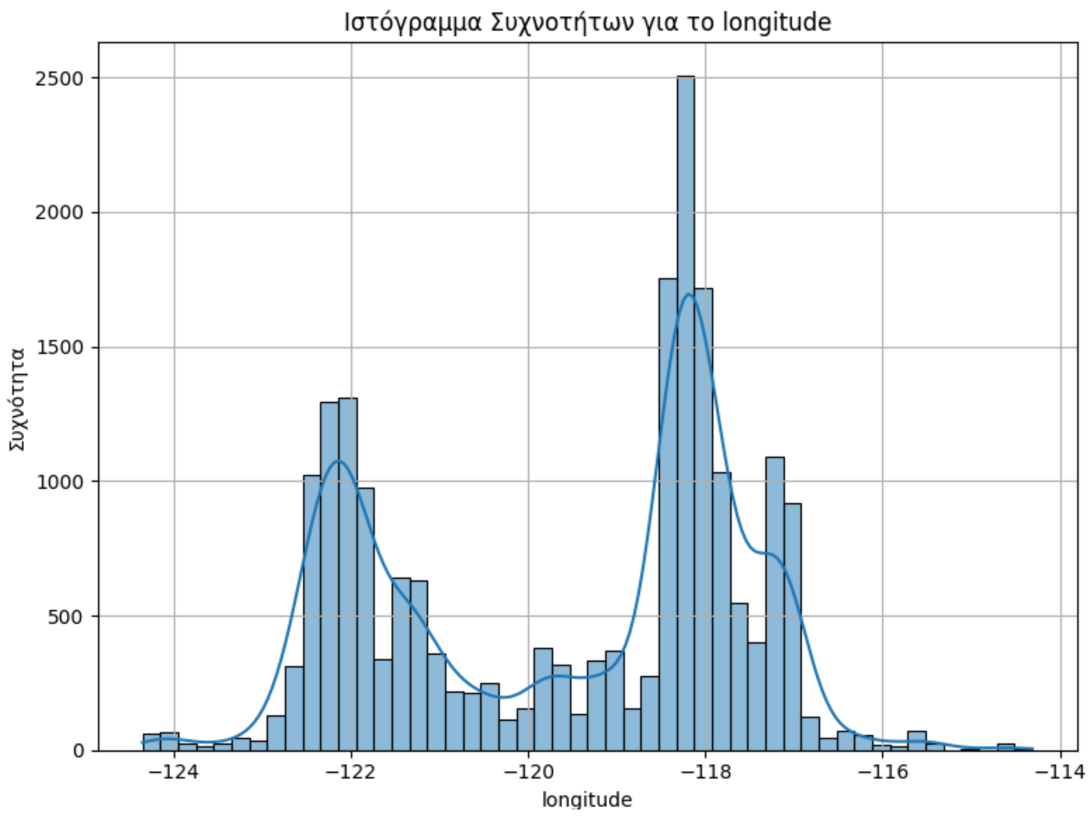
For this part we created a loop for each column in the data frame. Each variable goes through a check, if it is numeric sns.histplot is used to create the histogram, with kde set to true and bins set to 50, if the variable is categorical (in our case the “ ocean proximity”) sns.countplot is used to display the frequency of each category

Code :

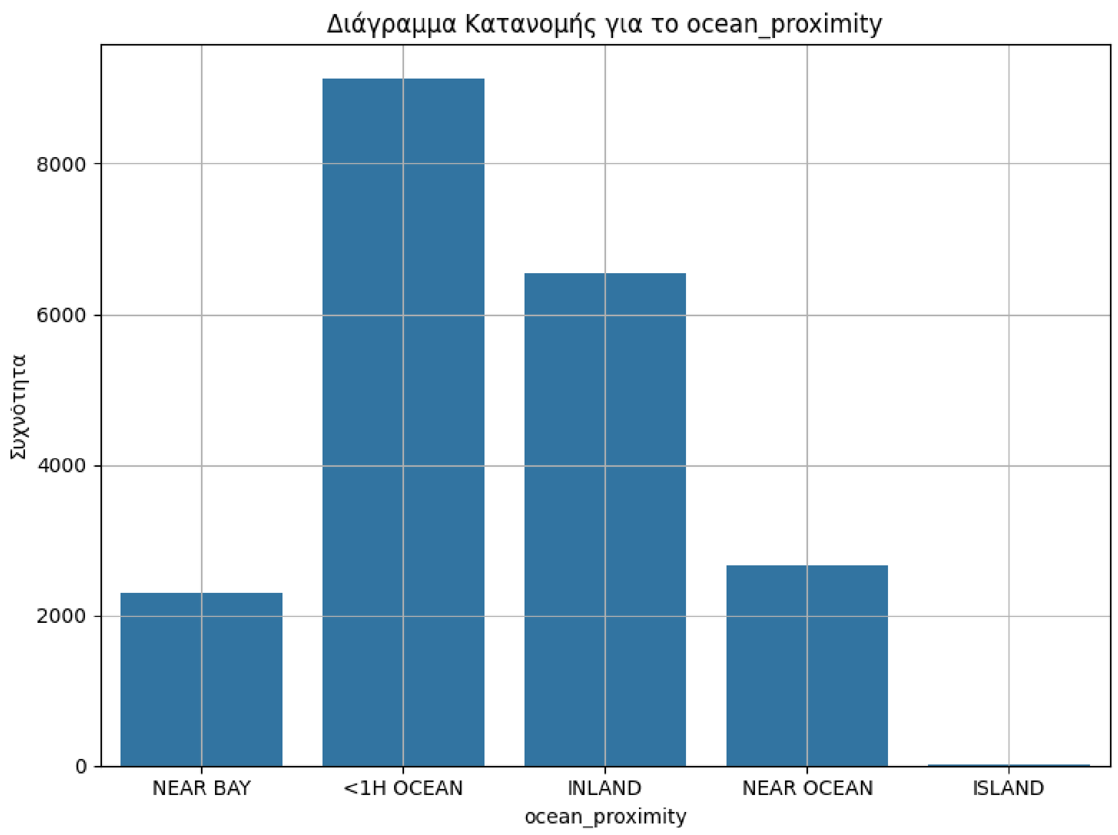


```python
for col in df.columns:
    plt.figure(figsize=(8, 6))
    # Ελέγχουμε εάν η μεταβλητή είναι αριθμητική ή κατηγορική
    if pd.api.types.is_numeric_dtype(df[col]):
        sns.histplot(df[col], kde=True, bins=50)  # Χρήση KDE για εκτίμηση της συνάρτησης πυκνότητας πιθανότητας
        plt.title(f'Ιστόγραμμα Συχνοτήτων για το {col}')
        plt.xlabel(col)
        plt.ylabel('Συχνότητα')
    else:
        sns.countplot(x=df[col])
        plt.title(f'Διάγραμμα Κατανομής για το {col}')
        plt.xlabel(col)
        plt.ylabel('Συχνότητα')
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

Result example1 (numeric):



Result example2 (categorical):



2. Two Feature Diagrams :

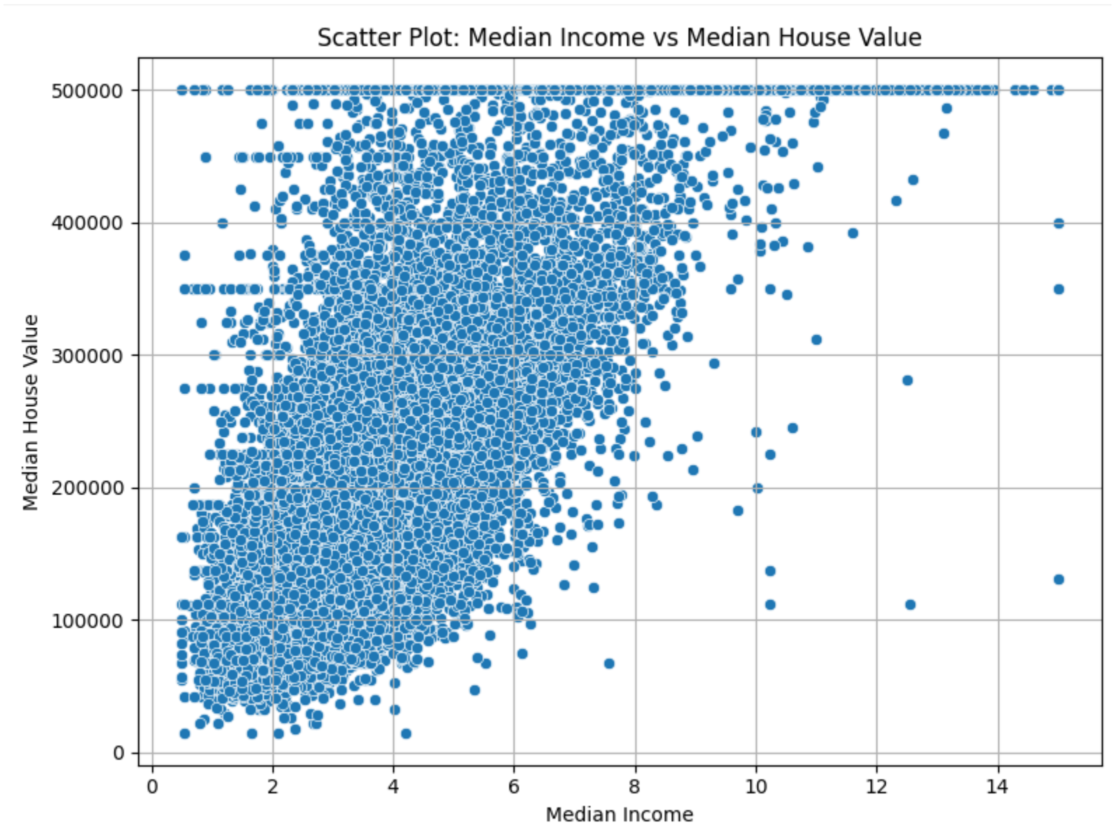
Explanation :

It represents one feature according to another

Code example:

```
plt.figure(figsize=(8, 6))
sns.scatterplot(data=df, x='longitude', y='latitude')
plt.title('Scatter Plot: Longitude vs Latitude')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Result example :

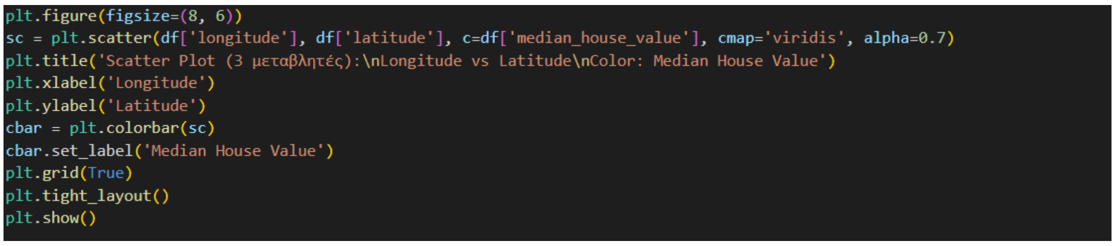


3. Three Feature Diagrams :

Explanation :

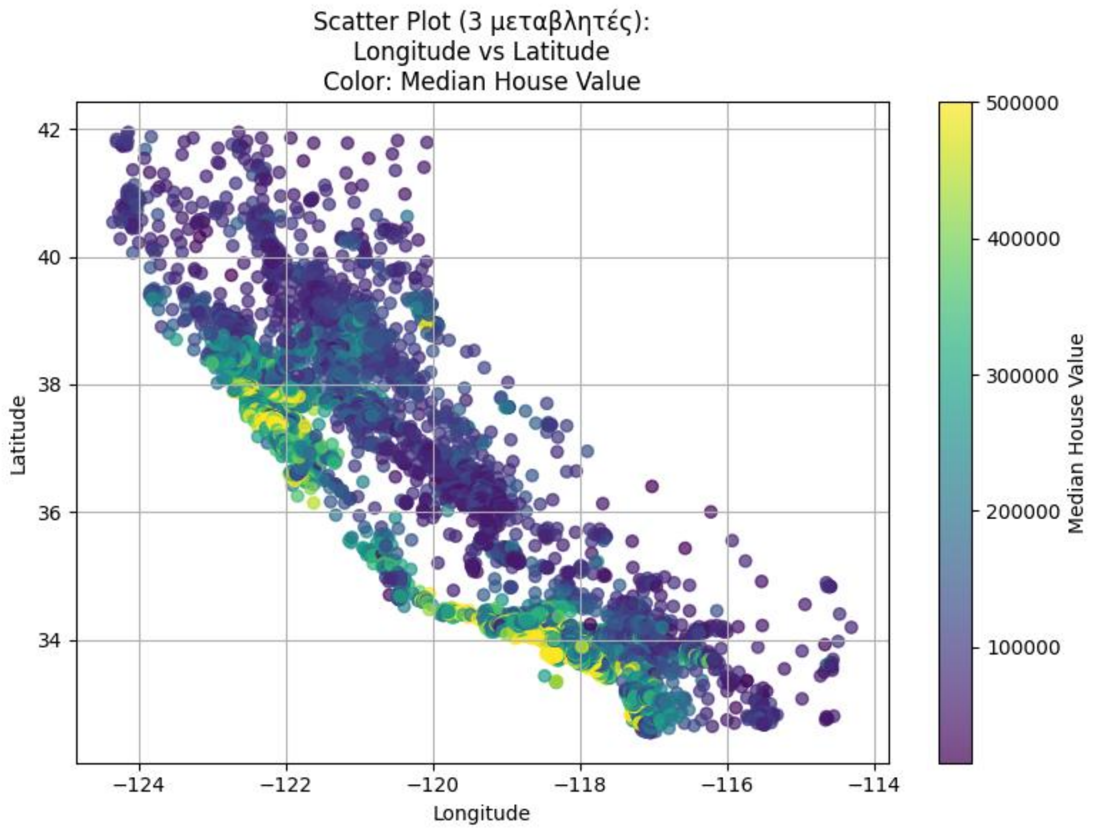
for the three features to be visible we used a color representation for the third.

Code example:



```
plt.figure(figsize=(8, 6))
sc = plt.scatter(df['longitude'], df['latitude'], c=df['median_house_value'], cmap='viridis', alpha=0.7)
plt.title('Scatter Plot (3 μεταβλητές):\nLongitude vs Latitude\nColor: Median House Value')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
cbar = plt.colorbar(sc)
cbar.set_label('Median House Value')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Result example :

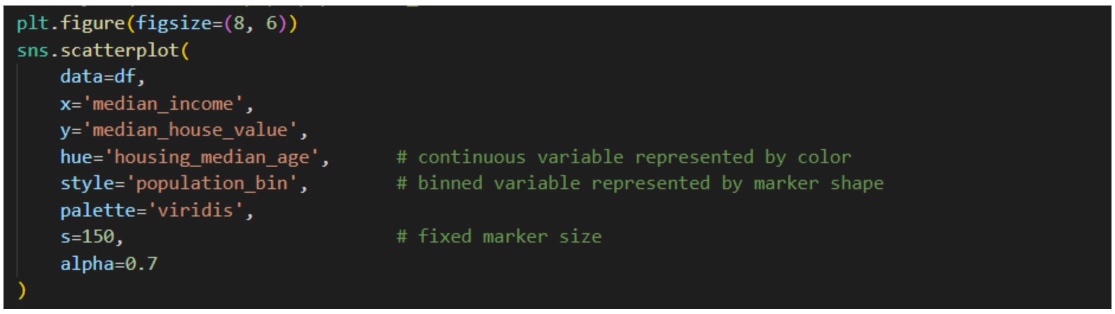


4. four Feature Diagrams :

Explanation :

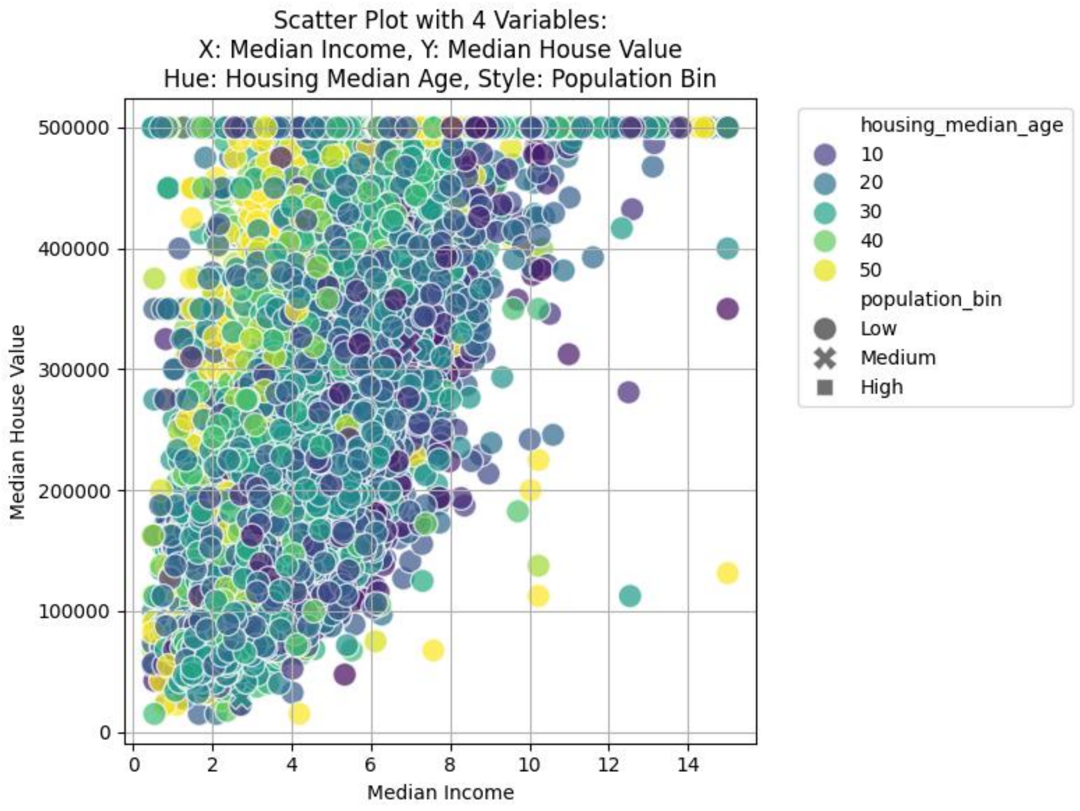
In this case symbols where used to represent the third variable as well as colors to represent the fourth .

Code example:



```
plt.figure(figsize=(8, 6))
sns.scatterplot(
    data=df,
    x='median_income',
    y='median_house_value',
    hue='housing_median_age',      # continuous variable represented by color
    style='population_bin',        # binned variable represented by marker shape
    palette='viridis',
    s=150,                         # fixed marker size
    alpha=0.7
)
```

Result example :



Single layer perceptron :

1. Threshold setting:

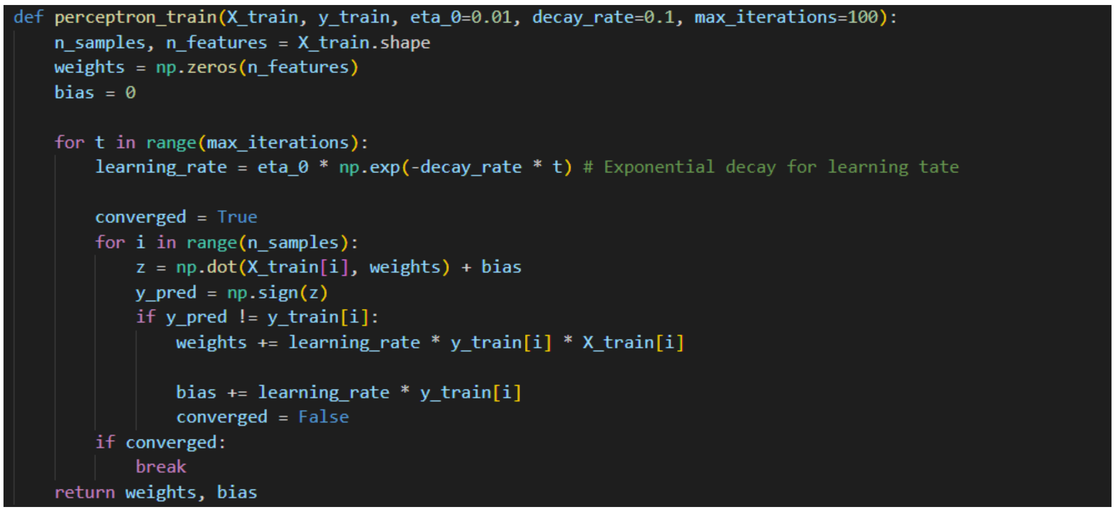
For the implementation of the single layer perceptron we need to transform the problem into a binary classification form. To achieve this we need to define a threshold in order to divide the dataset into separable classes, in our case the threshold is set to the scaled median of the target value “ median house value.

Code :

```
# Convert target to binary (+1/-1) based on the scaled median
threshold = y_scaler.transform([[np.median(y)]])[0, 0]
y_binary = np.where(y_scaled >= threshold, 1, -1)
```

2. Training :

For the training of the perceptron we implemented the following Function :



```python
def perceptron_train(X_train, y_train, eta_0=0.01, decay_rate=0.1, max_iterations=100):
    n_samples, n_features = X_train.shape
    weights = np.zeros(n_features)
    bias = 0

    for t in range(max_iterations):
        learning_rate = eta_0 * np.exp(-decay_rate * t) # Exponential decay for learning tate

        converged = True
        for i in range(n_samples):
            z = np.dot(X_train[i], weights) + bias
            y_pred = np.sign(z)
            if y_pred != y_train[i]:
                weights += learning_rate * y_train[i] * X_train[i]

                bias += learning_rate * y_train[i]
                converged = False
        if converged:
            break
    return weights, bias
```

The function initializes weights and bias to zero, It iterates up to ` max_iterations` times, updating the weights when a misclassification occurs.

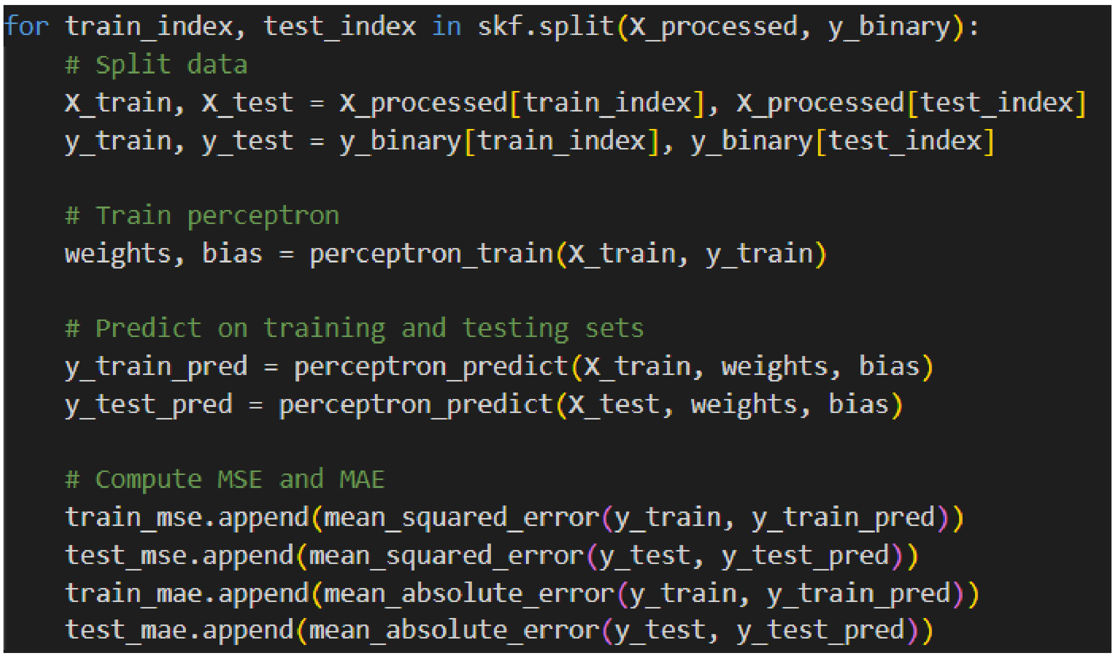
For the shake of performance we used the exponential decay technique in which we exponentially decrease the learning rate of the perceptron according to a specified decay rate. The hyperparameters decay_rate and eta_0 (initial learning rate) where set to the values seen above after experimenting and observing the output metrics.If the algorithm converges ,it stops even if the number of iterations hasn't reached ` max_iterations` .

Note : Because of limited computing power and for time saving purposes we set the max iteration to 100 although optimal performance is achieved at higher iteration number.

3. 10 fold validation:

As described in the exercise requirement the technique of 10 fold cross validation was used in the evaluation of all algorithms implemented (the code mostly remains the same throughout the algorithms so there will be no further explanation in the following parts).In the initialization we obviously used 10 folds as the exercise demands. We also set shuffle=true to randomly shuffle the data before splitting them and we set a fixed random_state so that the data is shuffled the same way every time. The 90%-10% split in each iteration is automatically determined by setting n_splits=10 in StratifiedKFold, which divides the dataset into 10 equal parts, using 9 parts (90%) for training and 1 part (10%) for testing in each fold.

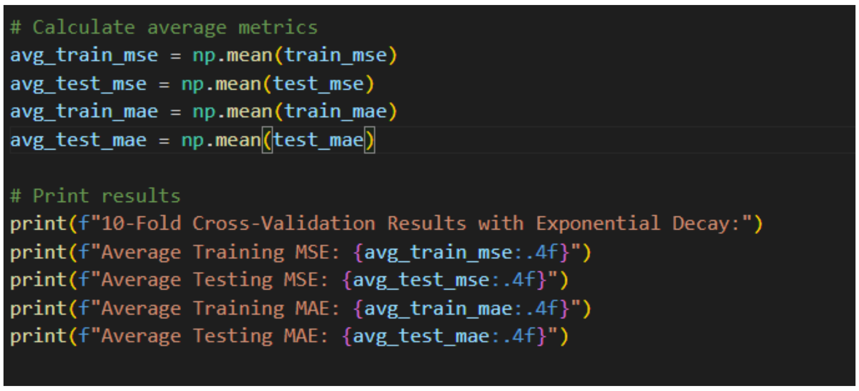
Code :



```python
for train_index, test_index in skf.split(X_processed, y_binary):
    # Split data
    X_train, X_test = X_processed[train_index], X_processed[test_index]
    y_train, y_test = y_binary[train_index], y_binary[test_index]

    # Train perceptron
    weights, bias = perceptron_train(X_train, y_train)

    # Predict on training and testing sets
    y_train_pred = perceptron_predict(X_train, weights, bias)
    y_test_pred = perceptron_predict(X_test, weights, bias)

    # Compute MSE and MAE
    train_mse.append(mean_squared_error(y_train, y_train_pred))
    test_mse.append(mean_squared_error(y_test, y_test_pred))
    train_mae.append(mean_absolute_error(y_train, y_train_pred))
    test_mae.append(mean_absolute_error(y_test, y_test_pred))
```

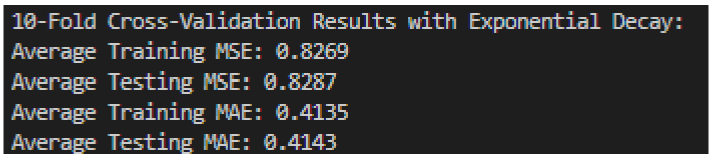
After the execution of the algorithm average testing and training MAE, MSE are printed .

Code :



```python
# Calculate average metrics
avg_train_mse = np.mean(train_mse)
avg_test_mse = np.mean(test_mse)
avg_train_mae = np.mean(train_mae)
avg_test_mae = np.mean(test_mae)

# Print results
print(f"10-Fold Cross-Validation Results with Exponential Decay:")
print(f"Average Training MSE: {avg_train_mse:.4f}")
print(f"Average Testing MSE: {avg_test_mse:.4f}")
print(f"Average Training MAE: {avg_train_mae:.4f}")
print(f"Average Testing MAE: {avg_test_mae:.4f}")
```

The best performance metrics achieved :



```
10-Fold Cross-Validation Results with Exponential Decay:
Average Training MSE: 0.8269
Average Testing MSE: 0.8287
Average Training MAE: 0.4135
Average Testing MAE: 0.4143
```

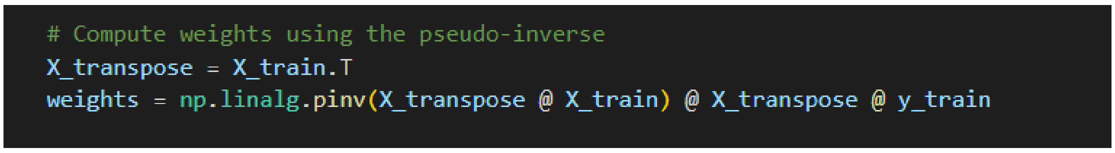
Least squares :

For the least squares algorithm the following were created :

1.Train:

Optimal weight are compute via the Normal Equation W =((X^t *X)^-1)*X^t*Y.To compute the weights we use np.linalg.pinv because it can handle matrixes that are not fully invertible.

Code :



```
# Compute weights using the pseudo-inverse
X_transpose = X_train.T
weights = np.linalg.pinv(X_transpose @ X_train) @ X_transpose @ y_train
```

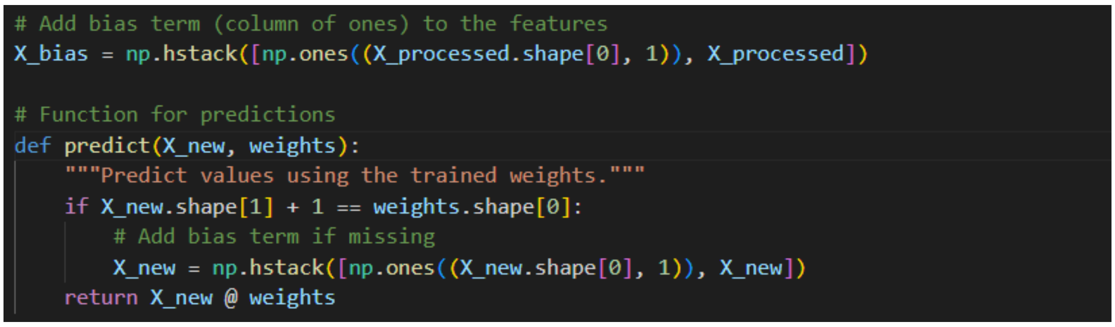
The code provided above is used inside the cross validation.

Note : In this(and in the multi-layer neural network) implementation of the 10 fold validation we used Kfold instead of StratifiedKFold which we used in single layer perceptron because Kfold is better suited for regression problems, whereas StratifiedKFold works best for classification ones.

2.predictions:

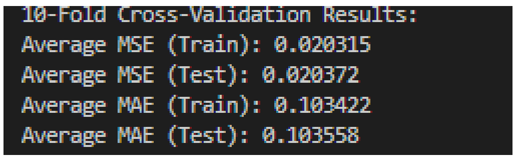
For the prediction process if the input features don't have a bias term we add column of ones and then we perform matrix multiplication with the weight to obtain the predicted value.

Code :



```
# Add bias term (column of ones) to the features
X_bias = np.hstack([np.ones((X_processed.shape[0], 1)), X_processed])

# Function for predictions
def predict(X_new, weights):
    """Predict values using the trained weights."""
    if X_new.shape[1] + 1 == weights.shape[0]:
        # Add bias term if missing
        X_new = np.hstack([np.ones((X_new.shape[0], 1)), X_new])
    return X_new @ weights
```

The best performance metrics achieved :



```
10-Fold Cross-Validation Results:
Average MSE (Train): 0.020315
Average MSE (Test): 0.020372
Average MAE (Train): 0.103422
Average MAE (Test): 0.103558
```

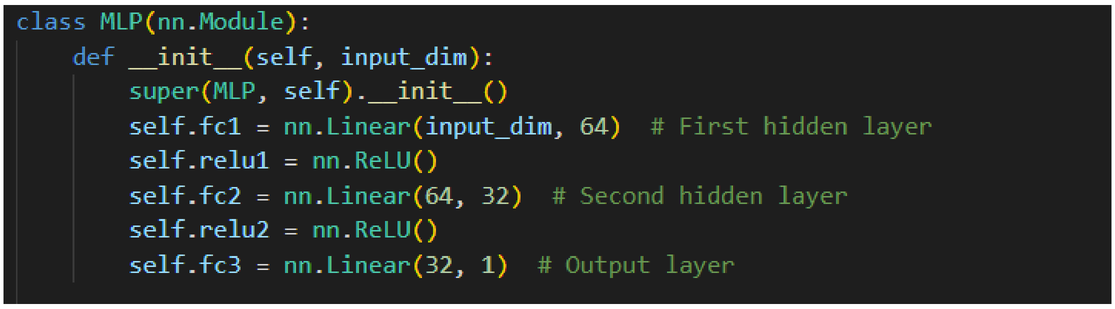
Multi layer neural network :

For this part of the exercise we constructed a multilayer neural network using the perceptron algorithm implemented in the first part as well as the same pre processing code used in the entirety of this exercise. This implementation differentiates from the previous one as I this case we used the pytorch library.

1.Layers :

Our MLP consists of two hidden layers as described in the code bellow

Code :



```
class MLP(nn.Module):
    def __init__(self, input_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, 64)  # First hidden layer
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(64, 32)  # Second hidden layer
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(32, 1)  # Output layer
```

More specifically **the First hidden layer consists of 64 neurons**, **the Second layer consists of 32** and **the output layer of 1.** The ReLU (Rectified Linear Unit) activation function helps the network handle non linearity. It replaces negative values with zero, preventing neurons from being inactive due to large negative inputs.

2.Training :

For each fold in the 10 fold validation process :

I. Data are converted to PyTorch tensors (instead of arrays as in previous parts)

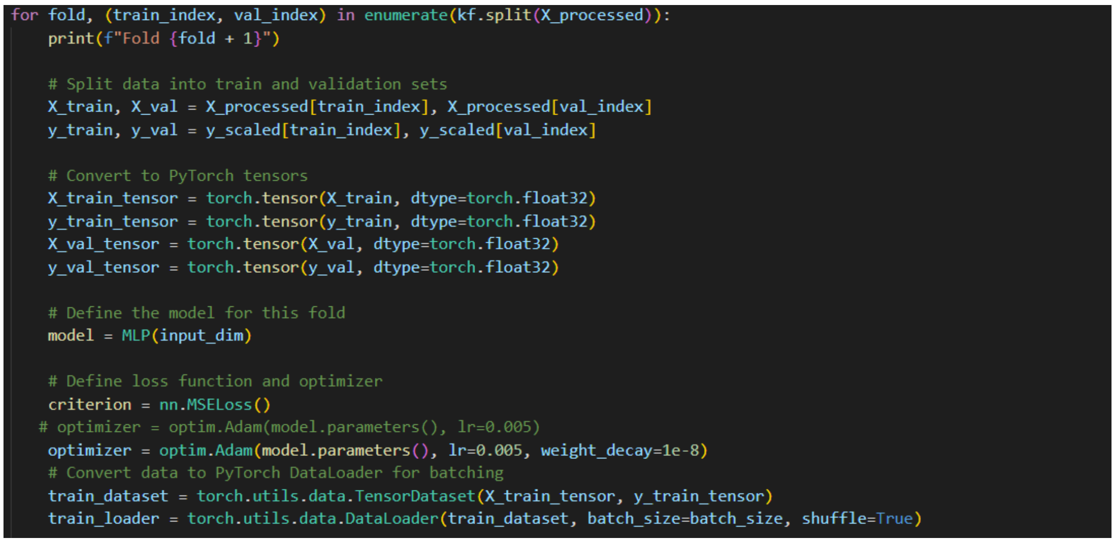
II. The adam optimizer is initialized with the following Hyperparameters :
   - Lr = 0.005
   - Weight Decay = 1e -8

   The values selected where the outcome of testing.

III. The dataloader splits the dataset into smaller batches the number of which is defined after experimenting and observing results.

IV. The loss function is initialized.

Code :



```
for fold, (train_index, val_index) in enumerate(kf.split(X_processed)):
    print(f"Fold {fold + 1}")

    # Split data into train and validation sets
    X_train, X_val = X_processed[train_index], X_processed[val_index]
    y_train, y_val = y_scaled[train_index], y_scaled[val_index]

    # Convert to PyTorch tensors
    X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
    y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
    X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
    y_val_tensor = torch.tensor(y_val, dtype=torch.float32)

    # Define the model for this fold
    model = MLP(input_dim)

    # Define loss function and optimizer
    criterion = nn.MSELoss()
   # optimizer = optim.Adam(model.parameters(), lr=0.005)
    optimizer = optim.Adam(model.parameters(), lr=0.005, weight_decay=1e-8)
    # Convert data to PyTorch DataLoader for batching
    train_dataset = torch.utils.data.TensorDataset(X_train_tensor, y_train_tensor)
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

Training Loop :

I. **model.train()**: Puts the model in training mode.

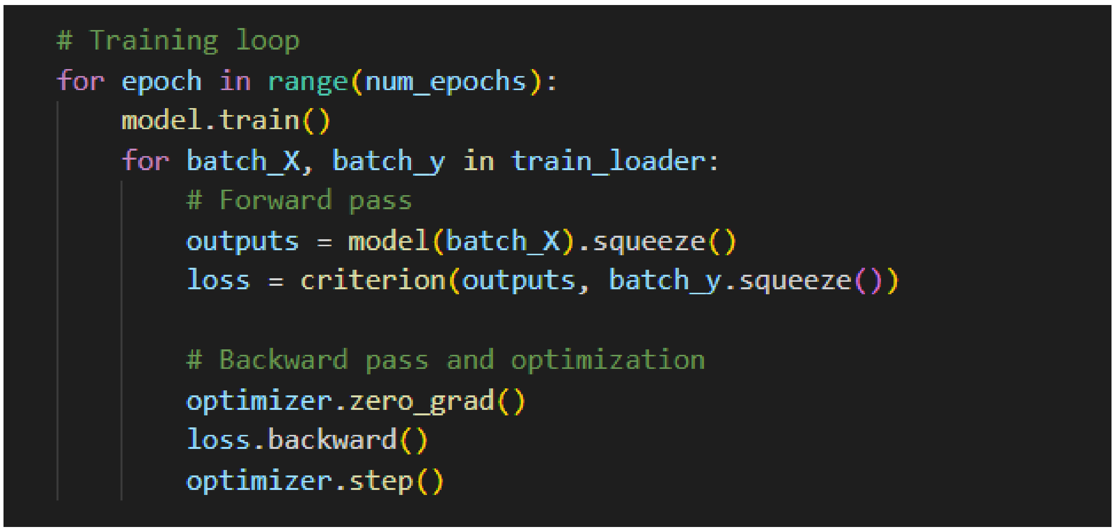
II. **Forward Pass**: Predictions are made with outputs = model(batch_X).squeeze().

III. **Loss Calculation**: Measures how wrong the predictions are.

IV. **Backward Pass (loss.backward())**: Computes gradients via the defined loss Function

V. **Optimizer Step (optimizer.step())**: Updates weights.

Code :

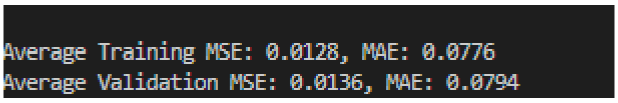


```
# Training loop
for epoch in range(num_epochs):
    model.train()
    for batch_X, batch_y in train_loader:
        # Forward pass
        outputs = model(batch_X).squeeze()
        loss = criterion(outputs, batch_y.squeeze())

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

3.Evaluation :

For the models performance evaluation we compute and print the training and testing MAE and MSE

Optimal performance achieved :



```
Average Training MSE: 0.0128, MAE: 0.0776
Average Validation MSE: 0.0136, MAE: 0.0794
```