real-housing-hoa-prediction

September 9, 2024

[3]:
```
# Import libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

1 EDA

[4]:
```
# Reading the dataset
house_data = pd.read_csv('/content/drive/MyDrive/Tech Consulting/raw_house_data␣
↪- raw_house_data.csv')
```

[5]:
```
house_data.head()
```

[5]:
```
        MLS  sold_price  zipcode   longitude   latitude  lot_acres     taxes  \
0  21530491   5300000.0    85637 -110.378200  31.356362    2154.00   5272.00
1  21529082   4200000.0    85646 -111.045371  31.594213    1707.00  10422.36
2   3054672   4200000.0    85646 -111.040707  31.594844    1707.00  10482.00
3  21919321   4500000.0    85646 -111.035925  31.645878     636.67   8418.58
4  21306357   3411450.0    85750 -110.813768  32.285162       3.21  15393.00

   year_built  bedrooms  bathrooms  sqrt_ft  garage  \
0        1941        13       10.0  10500.0     0.0
1        1997         2        2.0   7300.0     0.0
2        1997         2        3.0      NaN     NaN
3        1930         7        5.0   9019.0     4.0
4        1995         4        6.0   6396.0     3.0

                                    kitchen_features  fireplaces  \
0           Dishwasher, Freezer, Refrigerator, Oven         6.0
1                       Dishwasher, Garbage Disposal         5.0
2         Dishwasher, Garbage Disposal, Refrigerator         5.0
3  Dishwasher, Double Sink, Pantry: Butler, Refri…        4.0
4  Dishwasher, Garbage Disposal, Refrigerator, Mi…        5.0

              floor_covering  HOA
0          Mexican Tile, Wood    0
```

```
1          Natural Stone, Other    0
2     Natural Stone, Other: Rock  NaN
3   Ceramic Tile, Laminate, Wood  NaN
4               Carpet, Concrete   55
```

[6]:

```python
# Printing the info of the dataset
house_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 16 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   MLS               5000 non-null   int64
 1   sold_price        5000 non-null   float64
 2   zipcode           5000 non-null   int64
 3   longitude         5000 non-null   float64
 4   latitude          5000 non-null   float64
 5   lot_acres         4990 non-null   float64
 6   taxes             5000 non-null   float64
 7   year_built        5000 non-null   int64
 8   bedrooms          5000 non-null   int64
 9   bathrooms         4994 non-null   float64
 10  sqrt_ft           4944 non-null   float64
 11  garage            4993 non-null   float64
 12  kitchen_features  4967 non-null   object
 13  fireplaces        4975 non-null   float64
 14  floor_covering    4999 non-null   object
 15  HOA               4438 non-null   object
dtypes: float64(9), int64(4), object(3)
memory usage: 625.1+ KB
```

[7]:

```python
house_data.describe()
```

[7]:

```
                MLS    sold_price       zipcode    longitude     latitude  \
count  5.000000e+03  5.000000e+03   5000.000000  5000.000000  5000.000000
mean   2.127070e+07  7.746262e+05  85723.025600  -110.912107    32.308512
std    2.398508e+06  3.185556e+05     38.061712     0.120629     0.178028
min    3.042851e+06  1.690000e+05  85118.000000  -112.520168    31.356362
25%    2.140718e+07  5.850000e+05  85718.000000  -110.979260    32.277484
50%    2.161469e+07  6.750000e+05  85737.000000  -110.923420    32.318517
75%    2.180480e+07  8.350000e+05  85749.000000  -110.859078    32.394334
max    2.192856e+07  5.300000e+06  86323.000000  -109.454637    34.927884

         lot_acres         taxes  year_built     bedrooms    bathrooms  \
count  4990.000000  5.000000e+03  5000.00000  5000.000000  4994.000000
mean      4.661317  9.402828e+03  1992.32800     3.933800     3.829896
```

```
std      51.685230  1.729385e+05    65.48614     1.245362     1.387063
min       0.000000  0.000000e+00     0.00000     1.000000     1.000000
25%       0.580000  4.803605e+03  1987.00000     3.000000     3.000000
50%       0.990000  6.223760e+03  1999.00000     4.000000     4.000000
75%       1.757500  8.082830e+03  2006.00000     4.000000     4.000000
max    2154.000000  1.221508e+07  2019.00000    36.000000    36.000000

             sqrt_ft       garage   fireplaces
count    4944.000000  4993.000000  4975.000000
mean     3716.366828     2.816143     1.885226
std      1120.683515     1.192946     1.136578
min      1100.000000     0.000000     0.000000
25%      3047.000000     2.000000     1.000000
50%      3512.000000     3.000000     2.000000
75%      4130.250000     3.000000     3.000000
max     22408.000000    30.000000     9.000000
```

2 Missing Values

[8]:
```python
# Converting the HOA column to float
house_data['HOA'] = house_data['HOA'].str.replace(',', '').astype(float)
```

[9]:
```python
# Checking missing values
house_data.isnull().sum()
```

[9]:
```
MLS                   0
sold_price            0
zipcode               0
longitude             0
latitude              0
lot_acres            10
taxes                 0
year_built            0
bedrooms              0
bathrooms             6
sqrt_ft              56
garage                7
kitchen_features     33
fireplaces           25
floor_covering        1
HOA                 562
dtype: int64
```

[10]:
```python
# Replace NaN values in the 'HOA' column with 0
# Impute numeric columns with the median value
# Impute categorical columns with the median value
```

```python
house_data['HOA'] = house_data['HOA'].fillna(0)
house_data['sqrt_ft'] = house_data['sqrt_ft'].fillna(house_data['sqrt_ft'].
 ↪median())
house_data['lot_acres'] = house_data['lot_acres'].
 ↪fillna(house_data['lot_acres'].median())
house_data['fireplaces'] = house_data['fireplaces'].
 ↪fillna(house_data['fireplaces'].median())
house_data['garage'] = house_data['garage'].fillna(house_data['garage'].
 ↪median())
house_data['bathrooms'] = house_data['bathrooms'].
 ↪fillna(house_data['bathrooms'].median())
house_data['kitchen_features'] = house_data['kitchen_features'].
 ↪fillna(house_data['kitchen_features'].mode()[0])
house_data['floor_covering'] = house_data['floor_covering'].
 ↪fillna(house_data['floor_covering'].mode()[0])
```

[11]:
```python
# Checking missing values
house_data.isnull().sum()
```

[11]:
```
MLS                 0
sold_price          0
zipcode             0
longitude           0
latitude            0
lot_acres           0
taxes               0
year_built          0
bedrooms            0
bathrooms           0
sqrt_ft             0
garage              0
kitchen_features    0
fireplaces          0
floor_covering      0
HOA                 0
dtype: int64
```

[12]:
```python
# Checking for duplicates
house_data.duplicated().sum()
```

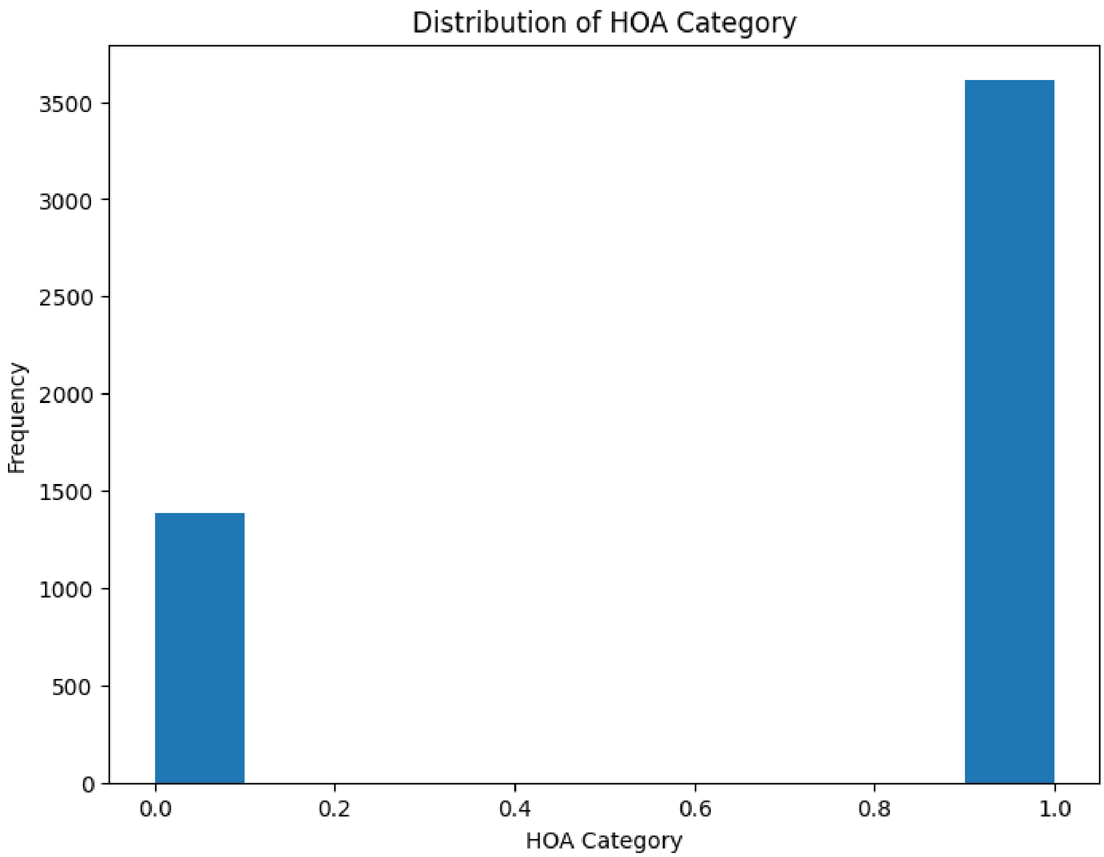
[12]:
```
0
```

3 Feature Engineering

[13]:
```
house_data['category'] = house_data['HOA'].apply(lambda x: 0 if x == 0 else 1)
house_data['category'].unique()
```

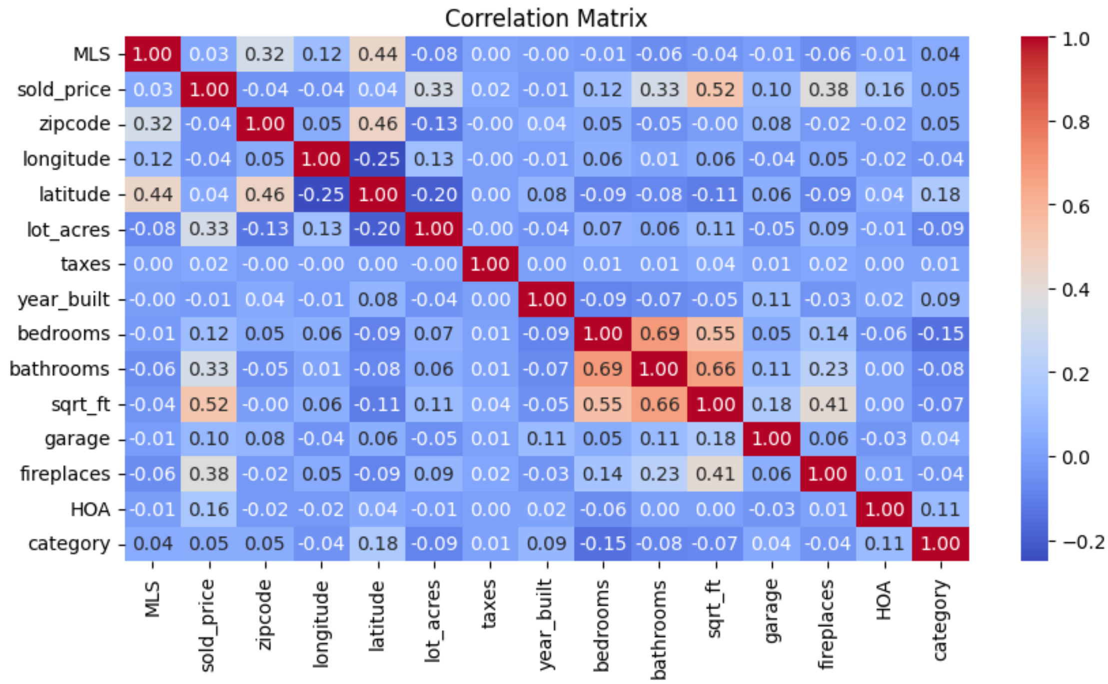
[13]:
```
array([0, 1])
```

[90]:
```
# Plot the distribution of train_df['HOA']
plt.figure(figsize=(8, 6))
plt.hist(house_data['category'])
plt.title('Distribution of HOA Category')
plt.xlabel('HOA Category')
plt.ylabel('Frequency')
plt.show()
```



[15]:
```
# Correlation matrix
plt.figure(figsize=(10,5))
corr_matrix = house_data.corr(numeric_only=True)
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix')
```

```
plt.show()
```



[16]:
```
# Label Encoding
house_data = pd.get_dummies(house_data, columns=['kitchen_features',␣
↪'floor_covering'], drop_first=True)
encoded_columns = house_data.filter(like='kitchen_features_').columns.
↪union(house_data.filter(like='floor_covering_').columns)
house_data[encoded_columns] = house_data[encoded_columns].astype(int)
```

[17]:
```
# Varibale Transformation

house_data['Lot_acres_log'] = np.log1p(house_data['lot_acres'])
house_data['Taxes_log'] = np.log1p(house_data['taxes'])
house_data['Bathrooms_log'] = np.log1p(house_data['bathrooms'])
house_data['Bedrooms_log'] = np.log1p(house_data['bedrooms'])
house_data['Garage_sqrt'] = np.sqrt(house_data['garage'])
house_data['Fireplaces_sqrt'] = np.sqrt(house_data['fireplaces'])
```

```
<ipython-input-17-f09d98be3542>:3: PerformanceWarning: DataFrame is highly
fragmented.  This is usually the result of calling `frame.insert` many times,
which has poor performance.  Consider joining all columns at once using
pd.concat(axis=1) instead. To get a de-fragmented frame, use `newframe =
frame.copy()`
  house_data['Lot_acres_log'] = np.log1p(house_data['lot_acres'])
<ipython-input-17-f09d98be3542>:4: PerformanceWarning: DataFrame is highly
```

```
fragmented.  This is usually the result of calling `frame.insert` many times,
which has poor performance.  Consider joining all columns at once using
pd.concat(axis=1) instead. To get a de-fragmented frame, use `newframe =
frame.copy()`
  house_data['Taxes_log'] = np.log1p(house_data['taxes'])
<ipython-input-17-f09d98be3542>:5: PerformanceWarning: DataFrame is highly
fragmented.  This is usually the result of calling `frame.insert` many times,
which has poor performance.  Consider joining all columns at once using
pd.concat(axis=1) instead. To get a de-fragmented frame, use `newframe =
frame.copy()`
  house_data['Bathrooms_log'] = np.log1p(house_data['bathrooms'])
<ipython-input-17-f09d98be3542>:6: PerformanceWarning: DataFrame is highly
fragmented.  This is usually the result of calling `frame.insert` many times,
which has poor performance.  Consider joining all columns at once using
pd.concat(axis=1) instead. To get a de-fragmented frame, use `newframe =
frame.copy()`
  house_data['Bedrooms_log'] = np.log1p(house_data['bedrooms'])
<ipython-input-17-f09d98be3542>:7: PerformanceWarning: DataFrame is highly
fragmented.  This is usually the result of calling `frame.insert` many times,
which has poor performance.  Consider joining all columns at once using
pd.concat(axis=1) instead. To get a de-fragmented frame, use `newframe =
frame.copy()`
  house_data['Garage_sqrt'] = np.sqrt(house_data['garage'])
<ipython-input-17-f09d98be3542>:8: PerformanceWarning: DataFrame is highly
fragmented.  This is usually the result of calling `frame.insert` many times,
which has poor performance.  Consider joining all columns at once using
pd.concat(axis=1) instead. To get a de-fragmented frame, use `newframe =
frame.copy()`
  house_data['Fireplaces_sqrt'] = np.sqrt(house_data['fireplaces'])
```

[18]:

```
# Outliers

def treat_outliers(df, features):
    df_filtered = df.copy()
    for column in features:
        Q1 = df_filtered[column].quantile(0.25)
        Q3 = df_filtered[column].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df_filtered = df_filtered[(df_filtered[column] >= lower_bound) &
 ↪(df_filtered[column] <= upper_bound)]
    return df_filtered
```

```
features= ['sold_price', 'longitude', 'latitude', 'year_built', 'sqrt_ft',␣
 ↪'HOA', 'Lot_acres_log', 'Taxes_log', 'Bathrooms_log', 'Bedrooms_log',␣
 ↪'Garage_sqrt', 'Fireplaces_sqrt']
data_cleaned = treat_outliers(house_data, features)
data_cleaned
```

[18]:

```
           MLS  sold_price  zipcode   longitude   latitude  lot_acres  \
313   21510119   1000000.0    85755 -110.992170  32.458323       3.49
359   21125701   1200000.0    85750 -110.846659  32.326433       1.05
361   21317020   1150000.0    85658 -111.085082  32.464902       0.75
371   21305294   1200000.0    85718 -110.942288  32.347119       0.99
391   21408527   1165000.0    85718 -110.942544  32.348593       1.06
…          …           …      …           …          …          …
4989  21902512    545000.0    85745 -111.061493  32.306472       1.19
4993  21908358    565000.0    85750 -110.820216  32.307646       0.83
4994  21909379    535000.0    85718 -110.922291  32.317496       0.18
4996  21908591    550000.0    85750 -110.858556  32.316373       1.42
4998  21900515    550000.0    85745 -111.055528  32.296871       1.01

         taxes  year_built  bedrooms  bathrooms  …  \
313   14400.00        2005         3        4.0  …
359    9450.00        1999         4        3.0  …
361   13534.17        2007         4        5.0  …
371   12434.42        2002         3        4.0  …
391   13129.23        2005         3        3.0  …
…          …           …        …          …  …
4989   6326.96        2007         4        3.0  …
4993   4568.71        1986         4        3.0  …
4994   4414.00        2002         3        2.0  …
4996   4822.01        1990         4        3.0  …
4998   5822.93        2009         4        4.0  …

      floor_covering_Wood, Other: Porcelain tile  \
313                                            0
359                                            0
361                                            0
371                                            0
391                                            0
…                                            …
4989                                           0
4993                                           0
4994                                           0
4996                                           0
4998                                           0

      floor_covering_Wood, Other: Travertine  \
313                                        0
```

```
359                                         0
361                                         0
371                                         0
391                                         0
…                                         …
4989                                        0
4993                                        0
4994                                        0
4996                                        0
4998                                        0

      floor_covering_Wood, Other: Travertine/Marble  \
313                                                0
359                                                0
361                                                0
371                                                0
391                                                0
…                                                …
4989                                               0
4993                                               0
4994                                               0
4996                                               0
4998                                               0

      floor_covering_Wood, Other: porcelain tile  Lot_acres_log  Taxes_log  \
313                                            0       1.501853   9.575053
359                                            0       0.717840   9.153876
361                                            0       0.559616   9.513047
371                                            0       0.688135   9.428304
391                                            0       0.722706   9.482672
…                                            …            …        …
4989                                           0       0.783902   8.752733
4993                                           0       0.604316   8.427205
4994                                           0       0.165514   8.392763
4996                                           0       0.883768   8.481153
4998                                           0       0.698135   8.669731

      Bathrooms_log  Bedrooms_log  Garage_sqrt  Fireplaces_sqrt
313        1.609438      1.386294     1.732051         1.732051
359        1.386294      1.609438     1.732051         1.414214
361        1.791759      1.609438     1.732051         1.732051
371        1.609438      1.386294     1.732051         1.414214
391        1.386294      1.386294     1.732051         1.732051
…               …           …          …              …
4989       1.386294      1.609438     2.000000         1.000000
4993       1.386294      1.609438     1.414214         1.414214
4994       1.098612      1.386294     1.414214         1.000000
```

```
4996      1.386294    1.609438    1.732051        1.000000
4998      1.609438    1.609438    1.732051        1.000000

[2984 rows x 2200 columns]
```

[19]:
```python
# Data Scaling

# def min_max_scale(df, exclude_columns):
#     df_scaled = df.copy()
#     for column in df_scaled.columns:
#         if column not in exclude_columns:
#             min_value = df_scaled[column].min()
#             max_value = df_scaled[column].max()
#             df_scaled[column] = (df_scaled[column] - min_value) / (max_value␣
↪- min_value)
#     return df_scaled
# exclude_columns = ['MLS','HOA', 'category']
# data_scaled = min_max_scale(data_cleaned, exclude_columns)
```

[94]:
```python
def min_max_scale(df, ignore_columns):
    transformed_df = df.copy()
    normalization_params = {}

    for col in transformed_df.columns:
        if col not in ignore_columns:
            min_val = transformed_df[col].min()
            max_val = transformed_df[col].max()
            transformed_df[col] = (transformed_df[col] - min_val) / (max_val -␣
↪min_val)
            normalization_params[col] = {'min_val': min_val, 'max_val': max_val}

    return transformed_df, normalization_params

# Columns to ignore from normalization
ignore_columns = ['MLS', 'HOA', 'category']

# Apply normalization and capture the parameters used for each feature
data_scaled, normalization_params = min_max_scale(data_cleaned, ignore_columns)
```

[20]:
```python
data_scaled
```

[20]:
```
          MLS  sold_price   zipcode  longitude  latitude  lot_acres  \
313  21510119    0.757576  1.000000   0.321638  0.778947   0.938172
359  21125701    1.000000  0.963235   0.645700  0.475739   0.282258
361  21317020    0.939394  0.286765   0.114716  0.794072   0.201613
371  21305294    1.000000  0.727941   0.432728  0.523295   0.266129
391  21408527    0.957576  0.727941   0.432158  0.526684   0.284946
```

```
…          …          …         …          …         …          …
4989  21902512    0.206061  0.926471   0.167251  0.429850   0.319892
4993  21908358    0.230303  0.963235   0.704591  0.432549   0.223118
4994  21909379    0.193939  0.727941   0.477263  0.455194   0.048387
4996  21908591    0.212121  0.963235   0.619205  0.452612   0.381720
4998  21900515    0.212121  0.926471   0.180535  0.407778   0.271505

         taxes  year_built  bedrooms  bathrooms  …  \
313   0.970961    0.762712  0.333333   0.666667  …
359   0.567048    0.661017  0.666667   0.333333  …
361   0.900310    0.796610  0.666667   1.000000  …
371   0.810572    0.711864  0.333333   0.666667  …
391   0.867268    0.762712  0.333333   0.333333  …
…          …          …         …          …  …
4989  0.312212    0.796610  0.666667   0.333333  …
4993  0.168742    0.440678  0.666667   0.333333  …
4994  0.156118    0.711864  0.333333   0.000000  …
4996  0.189411    0.508475  0.666667   0.333333  …
4998  0.271084    0.830508  0.666667   0.666667  …

      floor_covering_Wood, Other: Porcelain tile  \
313                                          0.0
359                                          0.0
361                                          0.0
371                                          0.0
391                                          0.0
…                                            …
4989                                         0.0
4993                                         0.0
4994                                         0.0
4996                                         0.0
4998                                         0.0

      floor_covering_Wood, Other: Travertine  \
313                                      NaN
359                                      NaN
361                                      NaN
371                                      NaN
391                                      NaN
…                                        …
4989                                     NaN
4993                                     NaN
4994                                     NaN
4996                                     NaN
4998                                     NaN

      floor_covering_Wood, Other: Travertine/Marble  \
```

```
313                                              NaN
359                                              NaN
361                                              NaN
371                                              NaN
391                                              NaN
…                                              …
4989                                             NaN
4993                                             NaN
4994                                             NaN
4996                                             NaN
4998                                             NaN

      floor_covering_Wood, Other: porcelain tile  Lot_acres_log  Taxes_log  \
313                                          0.0       0.967808   0.986245
359                                          0.0       0.462583   0.748924
361                                          0.0       0.360622   0.951306
371                                          0.0       0.443440   0.903556
391                                          0.0       0.465718   0.934191
…                                          …          …        …
4989                                         0.0       0.505153   0.522892
4993                                         0.0       0.389427   0.339466
4994                                         0.0       0.106659   0.320059
4996                                         0.0       0.569508   0.369864
4998                                         0.0       0.449884   0.476122

      Bathrooms_log  Bedrooms_log  Garage_sqrt  Fireplaces_sqrt
313        0.736966      0.415037     0.652847         0.732051
359        0.415037      0.736966     0.652847         0.414214
361        1.000000      0.736966     0.652847         0.732051
371        0.736966      0.415037     0.652847         0.414214
391        0.415037      0.415037     0.652847         0.732051
…           …           …          …             …
4989       0.415037      0.736966     0.891806         0.000000
4993       0.415037      0.736966     0.369398         0.414214
4994       0.000000      0.415037     0.369398         0.000000
4996       0.415037      0.736966     0.652847         0.000000
4998       0.736966      0.736966     0.652847         0.000000

[2984 rows x 2200 columns]
```

4 Classification Model

[21]:
```python
classification_data = data_scaled.copy()
```

[22]:
```python
import pandas as pd
```

```
# Separate majority and minority classes
majority_class = classification_data[classification_data['category'] == 1]
minority_class = classification_data[classification_data['category'] == 0]

# Oversample the minority class by repeating the minority rows until the sizes␣
↪match
oversampled_minority = minority_class.sample(len(majority_class), replace=True,␣
↪random_state=42)

# Combine the majority class with the oversampled minority class
oversampled_data = pd.concat([majority_class, oversampled_minority])

# Shuffle the data to mix the majority and oversampled minority class
oversampled_data = oversampled_data.sample(frac=1, random_state=42).
↪reset_index(drop=True)

# Display the value counts after oversampling
print(oversampled_data['category'].value_counts())
```

```
category
1    2386
0    2386
Name: count, dtype: int64
```

4.1 Splitting the data

[23]:
```
# Combine X and y_classification into one DataFrame to ensure consistent␣
↪shuffling
classification_data = oversampled_data.copy()
# data['HOA_class'] = y_classification

# Shuffling the data
data_shuffled = classification_data.sample(frac=1, random_state=42).
↪reset_index(drop=True)

# Calculating the split index
train_size = int(0.8 * len(data_shuffled))
train_df = data_shuffled[:train_size]
test_df = data_shuffled[train_size:]

X_train_classification = train_df[['zipcode', 'taxes',␣
↪'year_built','longitude', 'latitude', 'lot_acres', 'sqrt_ft', 'garage',␣
↪'fireplaces', 'bedrooms', 'bathrooms', 'sold_price']]
y_train_classification = train_df['category']
```

```
X_test_classification = test_df[['zipcode', 'taxes', 'year_built','longitude',␣
↪'latitude', 'lot_acres', 'sqrt_ft', 'garage', 'fireplaces', 'bedrooms',␣
↪'bathrooms', 'sold_price']]
y_test_classification = test_df['category']

len(X_train_classification), len(y_train_classification),␣
↪len(X_test_classification), len(y_test_classification)
```

[23]:

```
(3817, 3817, 955, 955)
```

[25]:

```
import statsmodels.api as sm
import warnings

def forward_regression(X, y,
                       threshold_in,
                       verbose=True):
    initial_list = []
    included = list(initial_list)
    model=sm.OLS(X,y)
    while True:
        changed=False
        excluded = list(set(X.columns)-set(included))
        new_pval = pd.Series(index=excluded)
        for new_column in excluded:
            model = sm.OLS(y, sm.add_constant(pd.
↪DataFrame(X[included+[new_column]]))).fit()
            new_pval[new_column] = model.pvalues[new_column]
        best_pval = new_pval.min()
        if best_pval < threshold_in:
            best_feature = new_pval.idxmin()
            included.append(best_feature)
            changed=True
            if verbose:
                print('Add  {:30} with p-value {:.6}'.format(best_feature,␣
↪best_pval))

        if not changed:
            break

    return included


model=forward_regression(X_train_classification,y_train_classification,0.05)
warnings.filterwarnings('ignore')
print(f'Useful predictors are :{model}')
```

```
Add  lot_acres                      with p-value 1.34984e-129
```

```
Add  latitude                      with p-value 2.24433e-27
Add  taxes                         with p-value 4.49862e-18
Add  zipcode                       with p-value 2.0701e-20
Add  sqrt_ft                       with p-value 2.57738e-09
Add  bedrooms                      with p-value 0.000116947
Add  longitude                     with p-value 0.0104415
Add  garage                        with p-value 0.00928583
Add  year_built                    with p-value 0.000727136
Add  sold_price                    with p-value 0.0257479
Useful predictors are :['lot_acres', 'latitude', 'taxes', 'zipcode', 'sqrt_ft',
'bedrooms', 'longitude', 'garage', 'year_built', 'sold_price']
```

[26]:
```
X_train_classification =X_train_classification[['lot_acres', 'latitude',␣
↪'taxes', 'zipcode', 'sqrt_ft', 'bedrooms', 'longitude', 'garage',␣
↪'year_built', 'sold_price']]
X_test_classification =X_test_classification[['lot_acres', 'latitude', 'taxes',␣
↪'zipcode', 'sqrt_ft', 'bedrooms', 'longitude', 'garage', 'year_built',␣
↪'sold_price']]
```

[27]:
```
X_train_classification= X_train_classification.to_numpy()
X_test_classification=X_test_classification.to_numpy()
y_train_classification=y_train_classification.to_numpy().astype('int')
y_test_classification=y_test_classification.to_numpy().astype('int')
```

4.2 Naive Bayes

[75]:
```
class GaussNB():
    def fit(self, X, y, epsilon = 1e-3):
        self.likelihoods = dict()
        self.priors = dict()
        self.K = set(y.astype(int))

        for k in self.K:
            X_k = X[y==k]
            # Naive assumption: Obseravtions are linearly independent of each␣
↪other
            self.likelihoods[k] = {"mean": X_k.mean(axis=0), "cov":X_k.
↪var(axis=0)+epsilon}
            self.priors[k] = len(X_k)/len(X)

    def predict(self, X):
        N, D = X.shape
        P_hat = np.zeros((N,len(self.K)))

        for k, l in self.likelihoods.items():
            P_hat[:,k] = mvn.logpdf(X, l["mean"], l["cov"])+np.log(self.
↪priors[k])
```

```
        return P_hat.argmax(axis=1)
```

[76]:
```
gnb = GaussNB()
gnb.fit(X_train_classification,y_train_classification)
y_hat_gnb= gnb.predict(X_test_classification)
```

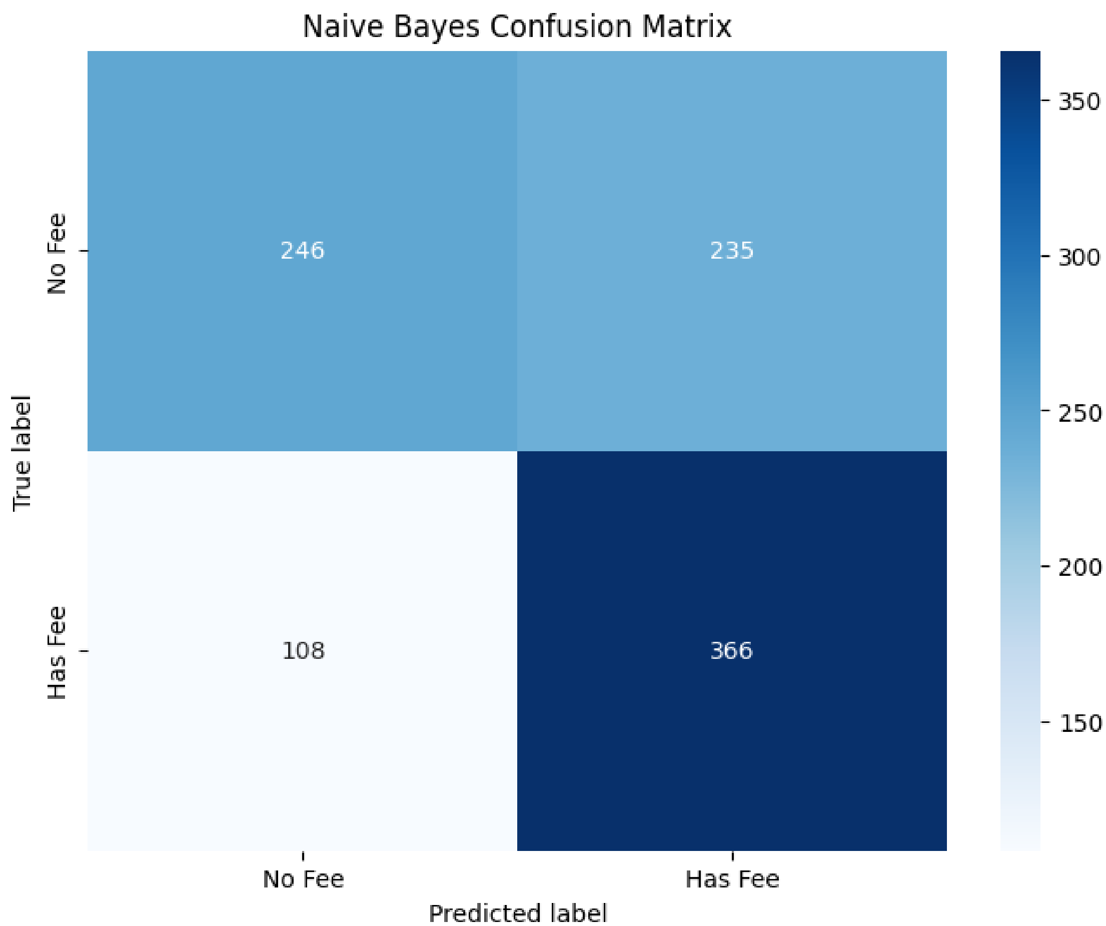
[30]:
```
def accuracy(y, y_hat):
  return np.mean(y==y_hat)
```

[77]:
```
accuracy(y_test_classification,y_hat_gnb)
```

[77]: 0.6408376963350786

[86]:
```
# 0 = "No Fee" and 1 = "Has Fee"
labels = ["No Fee", "Has Fee"]

# Plotting confusion matrix
plt.figure(figsize=(8, 6))
y_actu = pd.Series(y_test_classification, name='Actual')
y_pred = pd.Series(y_hat_gnb, name='Predicted')
cm = pd.crosstab(y_actu, y_pred)
ax = sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=labels,␣
 ↪yticklabels=labels)
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.title('Naive Bayes Confusion Matrix')
```

[86]: Text(0.5, 1.0, 'Naive Bayes Confusion Matrix')



4.3 Gaussian Bayes

[32]:
```
from scipy.stats import multivariate_normal as mvn
class GaussBayes():
    def fit(self, X, y, epsilon = 1e-3):
        self.likelihoods = dict()
        self.priors= dict()
        self.K = set(y.astype(int))

        for k in self.K:
            X_k = X[y==k, :]
            N_k, D = X_k.shape
            mu_k = X_k.mean(axis=0)

            self.likelihoods[k]= {"mean": X_k.mean(axis=0), "cov": (1/
 ↪(N_k-1))*np.matmul((X_k-mu_k).T, X_k-mu_k)+epsilon*np.identity(D)}
            self.priors[k] = len(X_k)/len(X)
```

```
    def predict(self, X):
        N, D = X.shape
        P_hat = np.zeros((N, len(self.K)))

        for k, l in self.likelihoods.items():
            P_hat[:,k]= mvn.logpdf(X, l["mean"], l["cov"])+ np.log(self.
↪priors[k])
        return P_hat.argmax(axis=1)
```

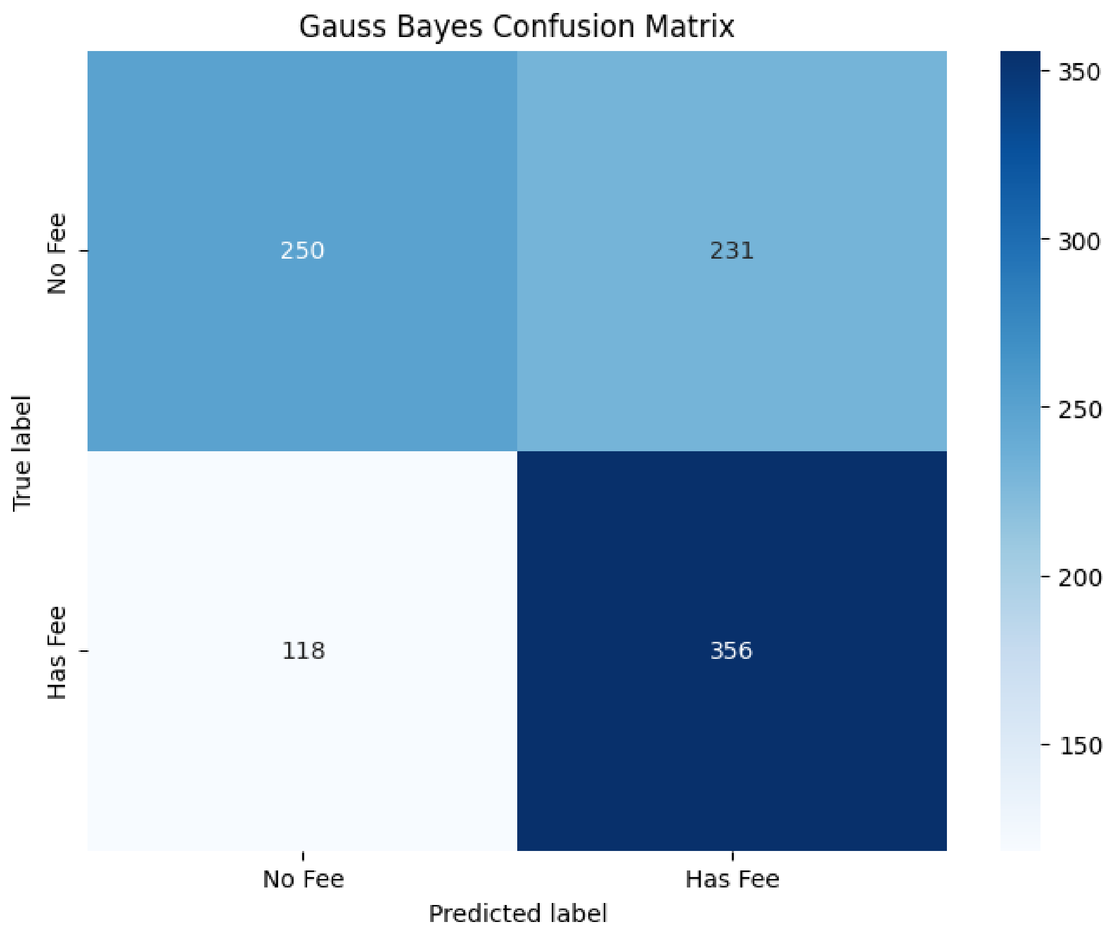
[33]:
```
gaussbayes = GaussBayes()
gaussbayes.fit(X_train_classification,y_train_classification, epsilon=1e-3)
y_hat_bayes = gaussbayes.predict(X_test_classification)
```

[34]:
```
accuracy(y_test_classification, y_hat_bayes)
```

[34]: 0.6345549738219896

[87]:
```
# 0 = "No Fee" and 1 = "Has Fee"
labels = ["No Fee", "Has Fee"]

# Plotting confusion matrix
plt.figure(figsize=(8, 6))
y_actu = pd.Series(y_test_classification, name='Actual')
y_pred = pd.Series(y_hat_bayes, name='Predicted')
cm = pd.crosstab(y_actu, y_pred)
ax = sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=labels,␣
↪yticklabels=labels)
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.title('Gauss Bayes Confusion Matrix')
```

[87]: Text(0.5, 1.0, 'Gauss Bayes Confusion Matrix')



4.4 KNN Classifier

[ ]:
```
class KNNClassifier():
  def fit(self, X, y):
    self.X = X
    self.y = y

  def predict(self, X, K, epsilon=1e-3):
    N= len(X)
    y_hat = np.zeros(N)

    for i in range(N):
      dist2 = np.sum((self.X-X[i])**2, axis=1)
      idxt = np.argsort(dist2)[:K]
      gamma_k = 1/(np.sqrt(dist2[idxt]+epsilon))

      y_hat[i] = np.bincount(self.y[idxt], weights=gamma_k).argmax()
```

```
    return y_hat
```

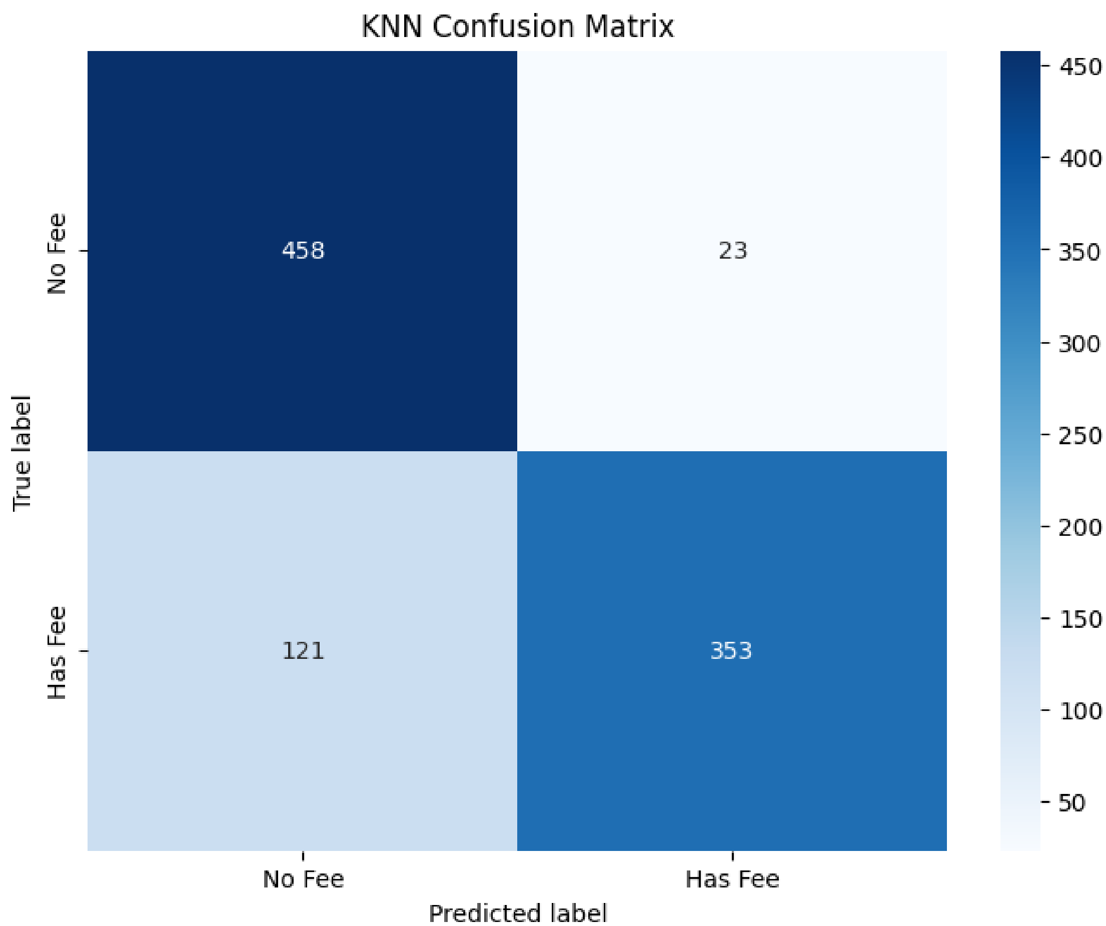
[ ]:
```
knn = KNNClassifier()
knn.fit(X_train_classification,y_train_classification)
```

[ ]:
```
best_k = 0
best_acc = 0
for i in range(1, 21):
  y_hat = knn.predict(X_test_classification, i)
  acc = accuracy(y_test_classification, y_hat)
  if acc > best_acc:
    best_acc = acc
    best_k = i
print(best_k, best_acc)
```

```
1 0.9287958115183246
```

[88]:
```
# 0 = "No Fee" and 1 = "Has Fee"
labels = ["No Fee", "Has Fee"]

# Plotting confusion matrix
plt.figure(figsize=(8, 6))
y_actu = pd.Series(y_test_classification, name='Actual')
y_pred = pd.Series(y_hat, name='Predicted')
cm = pd.crosstab(y_actu, y_pred)
ax = sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=labels,␣
 ↪yticklabels=labels)
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.title('KNN Confusion Matrix')
```

[88]:
```
Text(0.5, 1.0, 'KNN Confusion Matrix')
```



5 Regressor model

[54]:
```
X_train_reg = train_df[['zipcode', 'taxes', 'year_built','longitude',␣
 ↪'latitude', 'lot_acres', 'sqrt_ft', 'garage', 'fireplaces', 'bedrooms',␣
 ↪'bathrooms', 'sold_price']]
y_train_reg = train_df['HOA']

X_test_reg = test_df[['zipcode', 'taxes', 'year_built','longitude', 'latitude',␣
 ↪'lot_acres', 'sqrt_ft', 'garage', 'fireplaces', 'bedrooms', 'bathrooms',␣
 ↪'sold_price']]
y_test_reg = test_df['HOA']

len(X_train_reg), len(y_train_reg), len(X_test_reg), len(y_test_reg)
```

[54]: (3817, 3817, 955, 955)

[55]:

```
# Feature Selection

import statsmodels.api as sm
import warnings

def forward_regression(X, y,
                       threshold_in,
                       verbose=True):
    initial_list = []
    included = list(initial_list)
    model=sm.OLS(X,y)
    while True:
        changed=False
        excluded = list(set(X.columns)-set(included))
        new_pval = pd.Series(index=excluded)
        for new_column in excluded:
            model = sm.OLS(y, sm.add_constant(pd.
 ↪DataFrame(X[included+[new_column]]))).fit()
            new_pval[new_column] = model.pvalues[new_column]
        best_pval = new_pval.min()
        if best_pval < threshold_in:
            best_feature = new_pval.idxmin()
            included.append(best_feature)
            changed=True
            if verbose:
                print('Add  {:30} with p-value {:.6}'.format(best_feature,␣
 ↪best_pval))

        if not changed:
            break

    return included


model=forward_regression(X_train_reg,y_train_reg,0.05)
warnings.filterwarnings('ignore')
print(f'Useful predictors are :{model}')
```

```
Add  latitude                       with p-value 5.49105e-79
Add  lot_acres                      with p-value 1.34997e-48
Add  taxes                          with p-value 1.34865e-46
Add  bedrooms                       with p-value 8.05185e-40
Add  longitude                      with p-value 2.10922e-33
Add  sold_price                     with p-value 5.38334e-06
Add  sqrt_ft                        with p-value 2.68018e-07
Add  fireplaces                     with p-value 0.00194359
Useful predictors are :['latitude', 'lot_acres', 'taxes', 'bedrooms',
```

'longitude', 'sold_price', 'sqrt_ft', 'fireplaces']

[56]:
```
X_train_reg =X_train_reg[['latitude', 'lot_acres', 'taxes', 'bedrooms',␣
 ↪'longitude', 'sold_price', 'sqrt_ft', 'fireplaces']]
X_test_reg =X_test_reg[['latitude', 'lot_acres', 'taxes', 'bedrooms',␣
 ↪'longitude', 'sold_price', 'sqrt_ft', 'fireplaces']]
```

[57]:
```
X_train_reg= X_train_reg.to_numpy()
X_test_reg=X_test_reg.to_numpy()
y_train_reg= y_train_reg.to_numpy().astype('int')
y_test_reg= y_test_reg.to_numpy().astype('int')
```

5.1 KNN Regressor

[58]:
```
class KNNRegressor():

  def fit(self, X, y):
    self.X = X
    self.y = y

  def predict(self, X, K, epsilon = 1e-3):
    N = len(X)
    y_hat = np.zeros(N)

    for i in range(N):
      dist2 = np.sum((self.X-X[i])**2, axis=1)
      idxt = np.argsort(dist2)[:K]
      gamma_k = np.exp(-dist2[idxt])/(np.exp(-dist2[idxt]).sum()+epsilon)
      y_hat[i] = gamma_k.dot(self.y[idxt])
      # y_hat[i] = np.inner(gamma_k, self.y[idxt])
    return y_hat
```

[59]:
```
# Intantiate our class
knnr = KNNRegressor()
```

[60]:
```
knnr.fit(X_train_reg, y_train_reg)
```

[61]:
```
def RMSE(y_true, y_pred):
  y_true, y_pred = np.array(y_true), np.array(y_pred)
  return np.sqrt(np.mean((y_true - y_pred) ** 2))
```

[62]:
```
best_k_model3 = 3
best_err_model3 = 100
for i in range(1, 15):
  y_hat_knn = knnr.predict(X_test_reg, i, epsilon=0.1)
  err = RMSE(y_test_reg, y_hat_knn)
  if err < best_err_model3:
```

```python
        best_err_model3 = err
        best_k_model3 = i
print(best_k_model3, best_err_model3)
```

```
2 45.81614959104186
```

5.2 Multi Variate Linear Regression

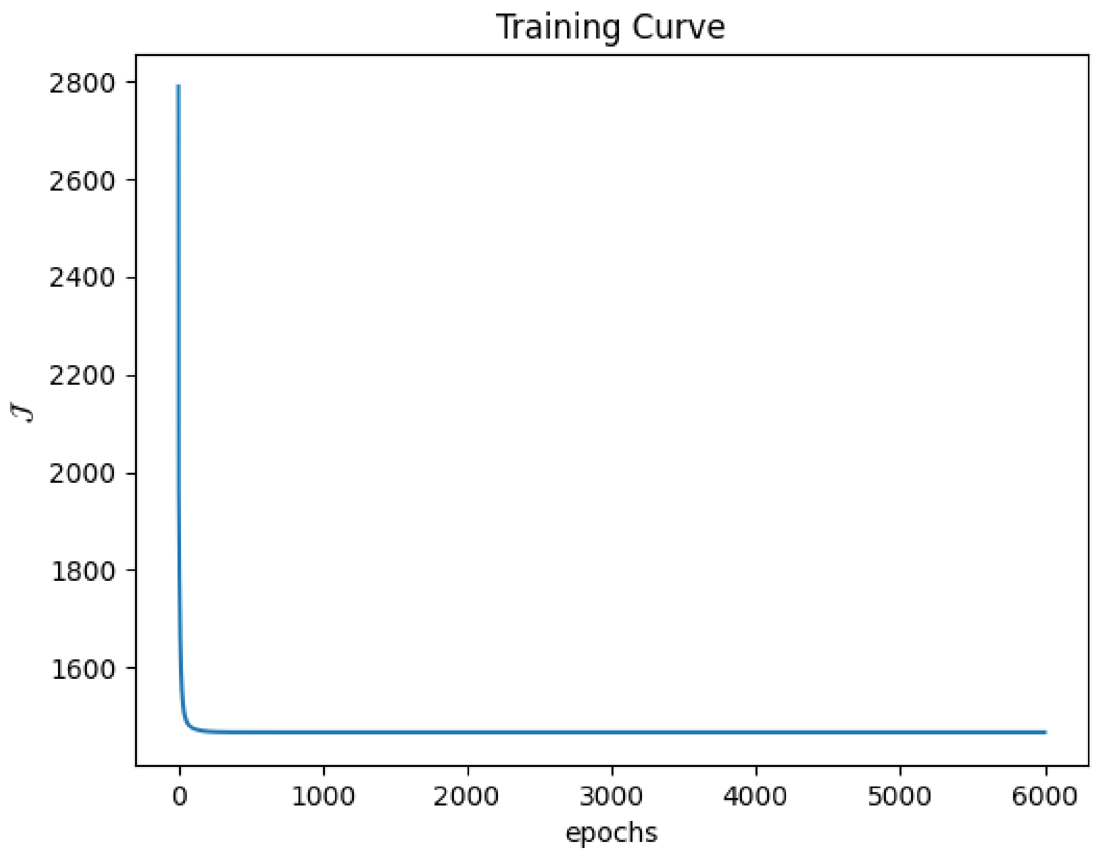
[69]:
```python
class MVLinearRegression():
  def fit(self, X, y, eta = 1e-3, epochs = 1e3, show_curve = False):
    epochs = int(epochs)
    N, D = X.shape
    Y = y

    #Begin Optimization
    self.W = np.random.randn(D)
    self.J = np.zeros(epochs)

    #Stochastic Gradient Descent
    for epoch in range(epochs):
      Y_hat = self.predict(X)
      self.J[epoch] = OLS(Y, Y_hat, N)
      #Weight Update Rule
      self.W -= eta*(1/N)*(X.T@(Y_hat-Y))

    if show_curve:
      plt.figure()
      plt.plot(self.J)
      plt.xlabel("epochs")
      plt.ylabel("$\mathcal{J}$")
      plt.title("Training Curve")

  def predict(self, X):
    return X@self.W
```

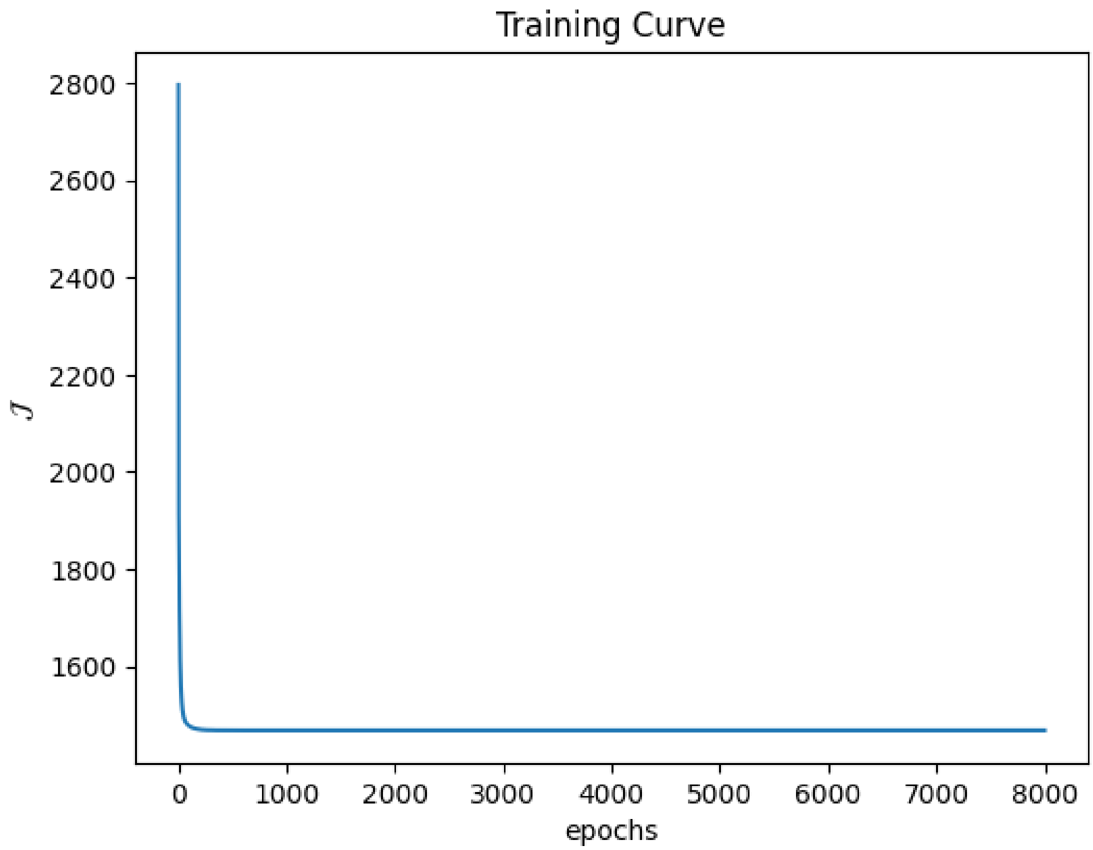
[70]:
```python
mvlin_reg = MVLinearRegression()
```

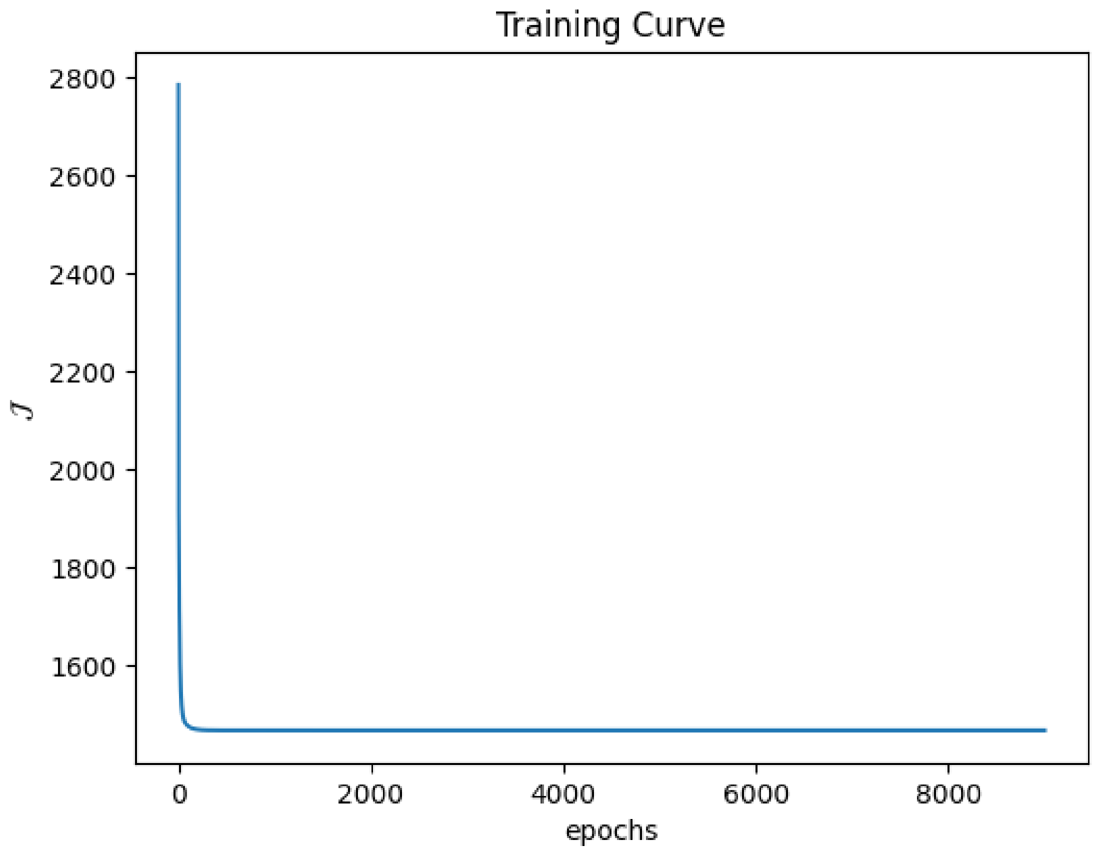
[71]:
```python
best_eta = 1e-3
best_epochs = 1e3
best_err_model2 = float('inf')
eta_values = [0.8, 1.0, 1.2, 1.5]
epoch_values = [6000, 8000, 9000, 10000]
for eta in eta_values:
    for epochs in epoch_values:
        mvlin_reg.fit(X_train_reg, y_train_reg, eta=eta, epochs=epochs,␣
↪show_curve=True)
        y_hat_reg = mvlin_reg.predict(X_test_reg)
```

```
        err = RMSE(y_test_reg, y_hat_reg)
        if err < best_err_model2:
            best_err_model2 = err
            best_eta = eta
            best_epochs = epochs
print(f"Best eta: {best_eta}, Best epochs: {best_epochs}, Best error:␣
 ↪{best_err_model2}")
```

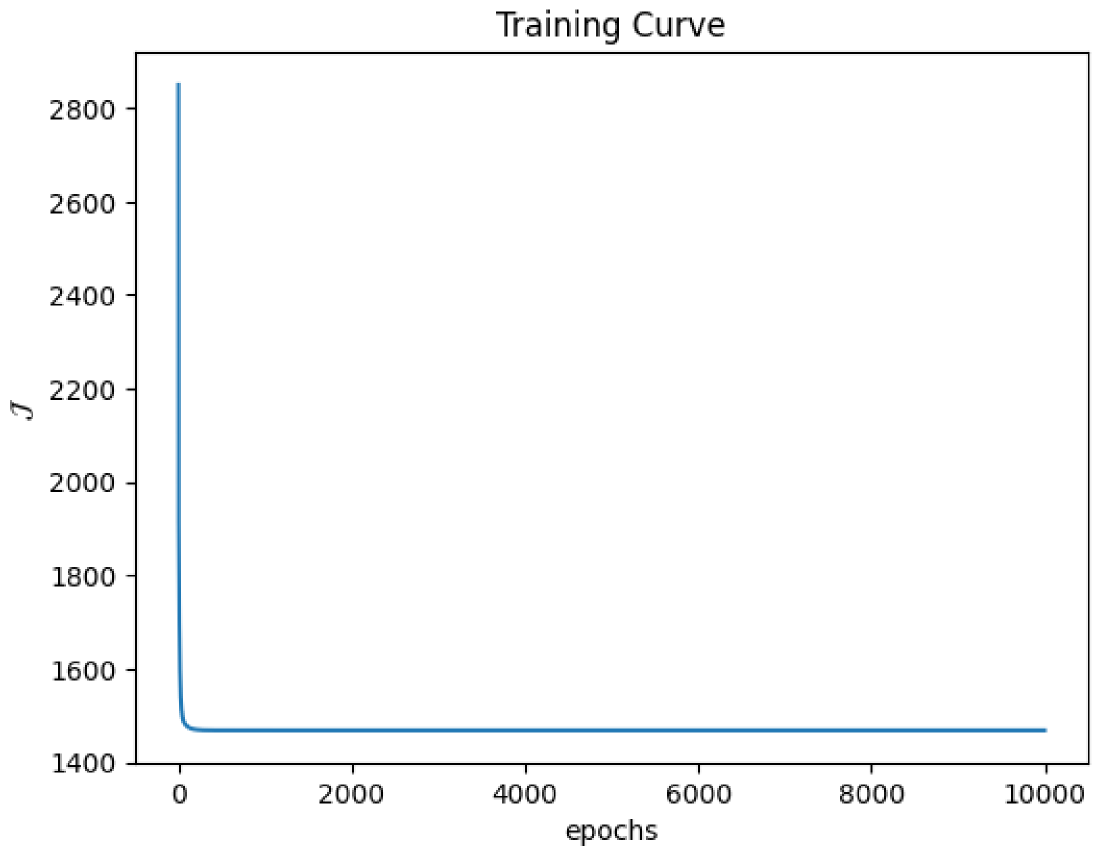
```
Best eta: 1.0, Best epochs: 6000, Best error: 53.821635088193936
```

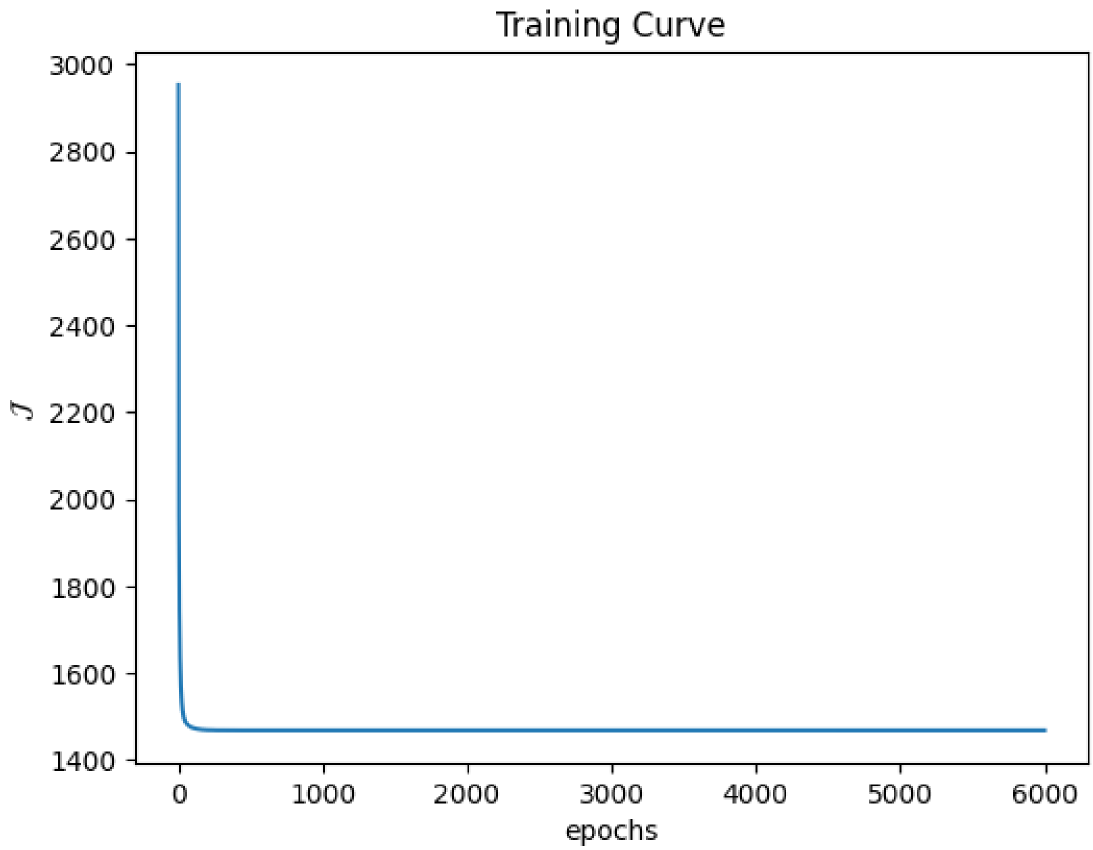



Training Curve
2800
2600
2400
2200
2000
1800
1600
$\mathcal{J}$
0
1000
2000
3000
4000
5000
6000
7000
8000
epochs

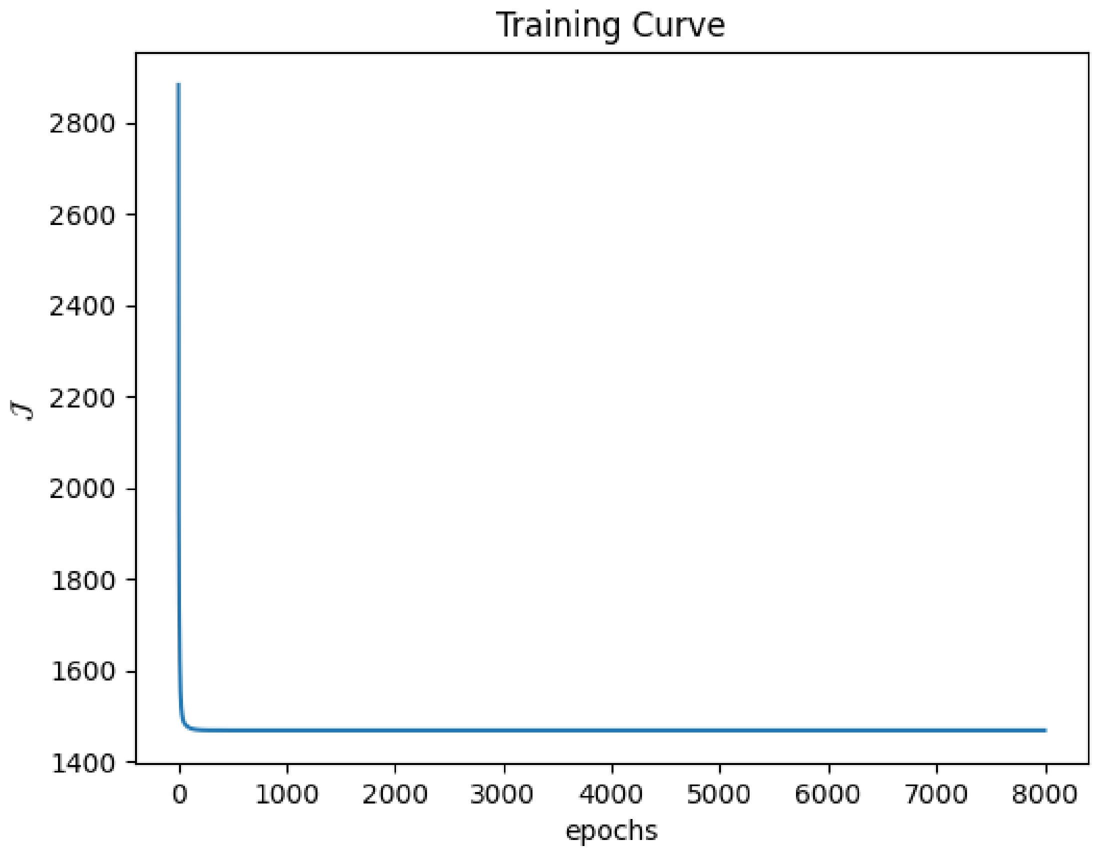


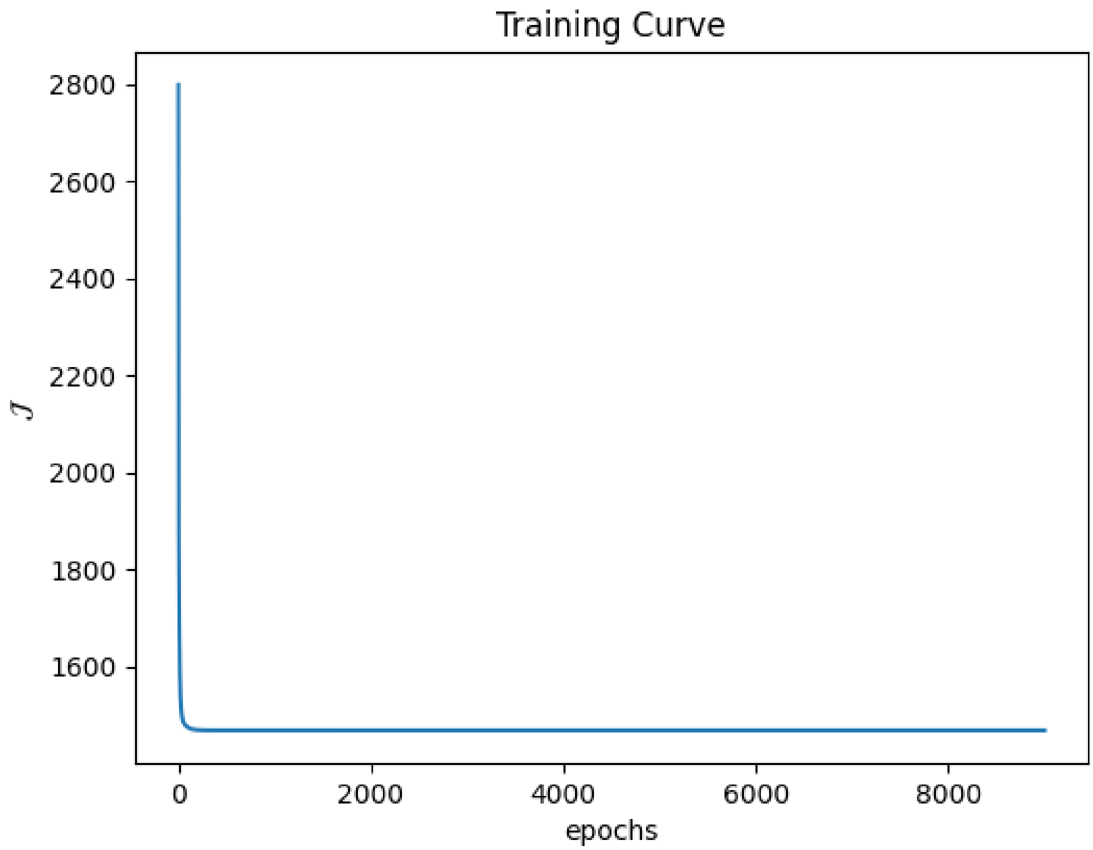
Training Curve
2800
2600
2400
2200
2000
1800
1600
$\mathcal{J}$
0
2000
4000
6000
8000
epochs


Training Curve
2800
2600
2400
2200
2000
1800
1600
1400
$\mathcal{J}$
0
2000
4000
6000
8000
10000
epochs

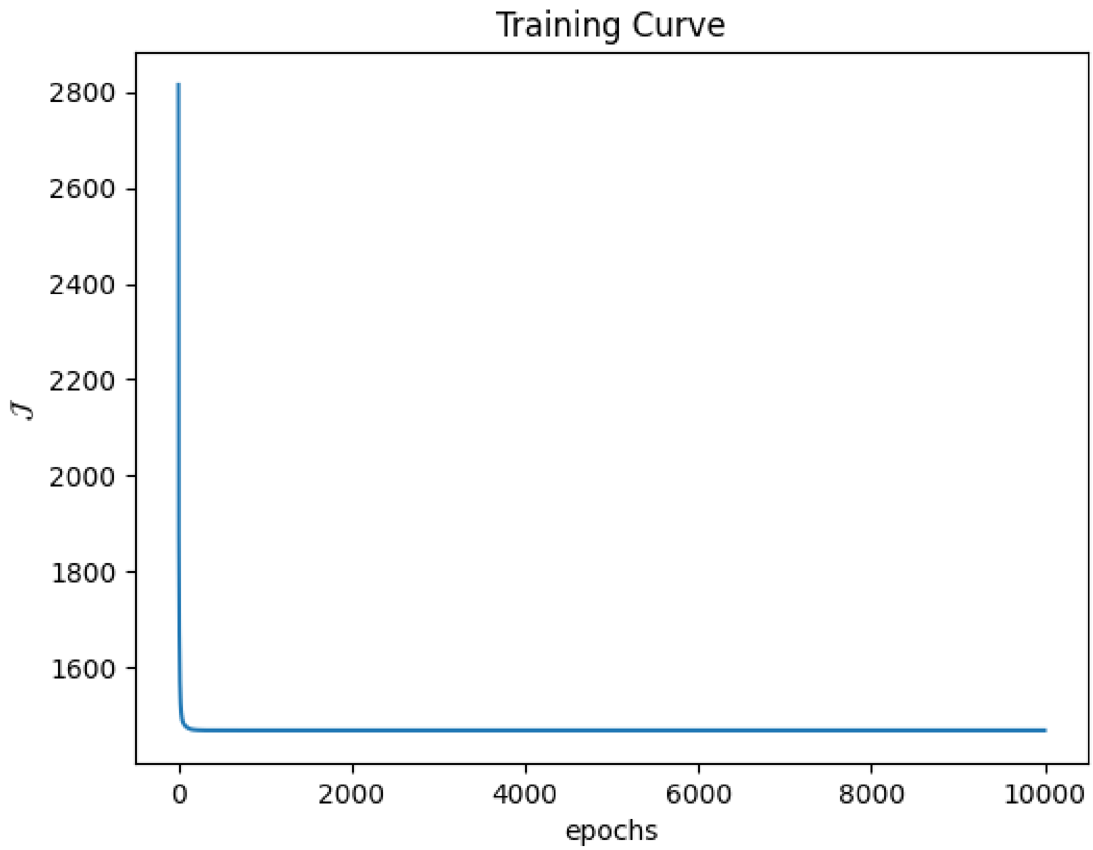


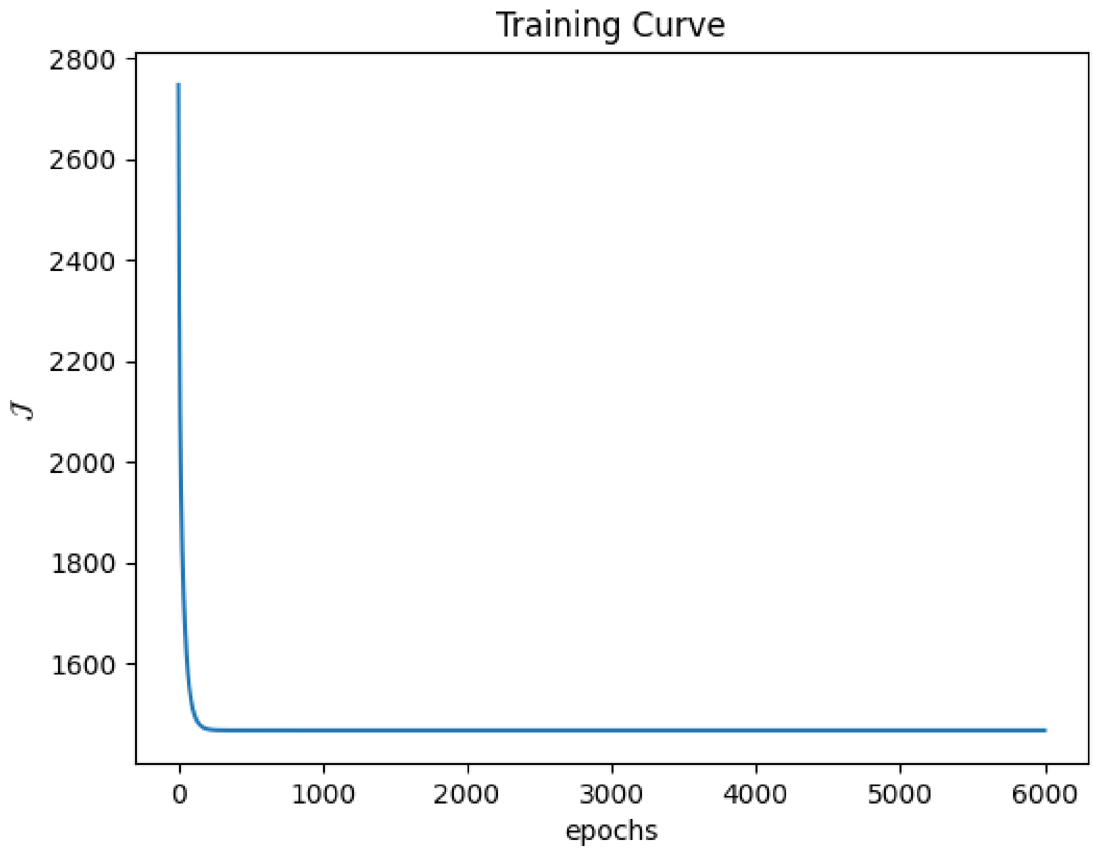
Training Curve
3000
2800
2600
2400
2200
2000
1800
1600
1400
$\mathcal{J}$
0
1000
2000
3000
4000
5000
6000
epochs

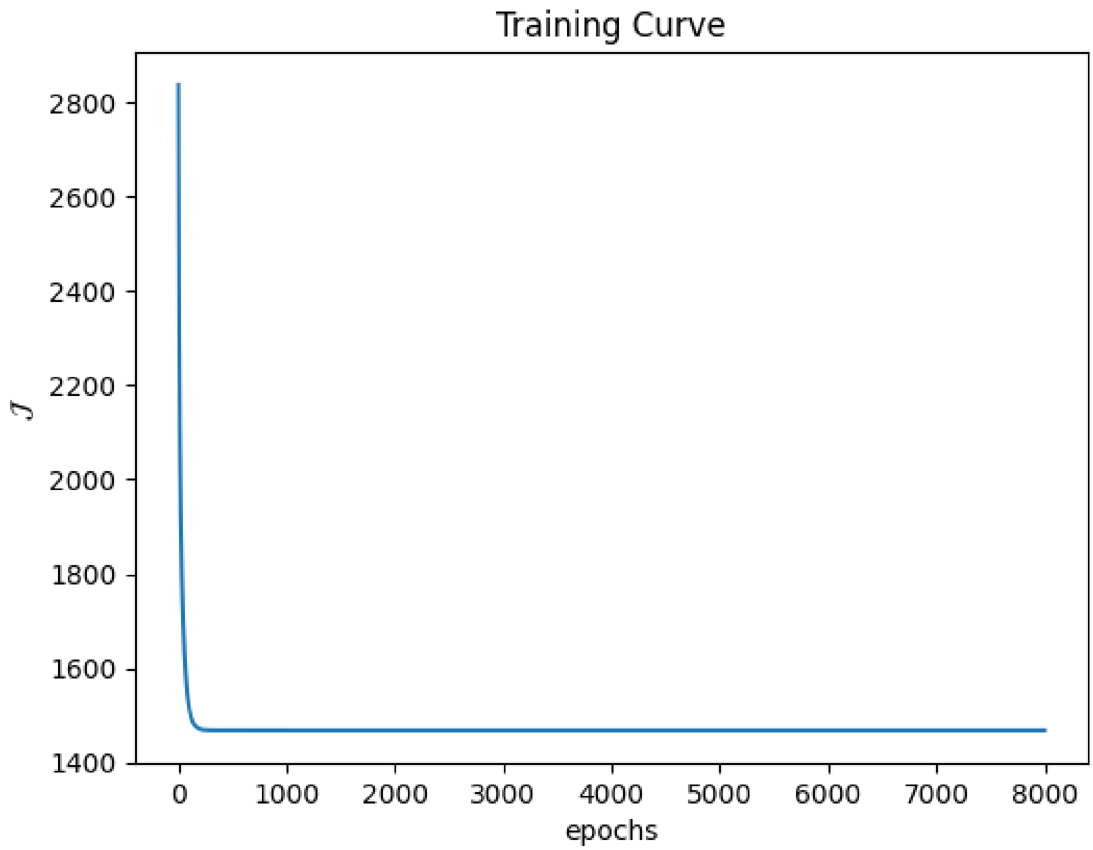

Training Curve
2800
2600
2400
2200
2000
1800
1600
1400
$\mathcal{J}$
0
1000
2000
3000
4000
5000
6000
7000
8000
epochs

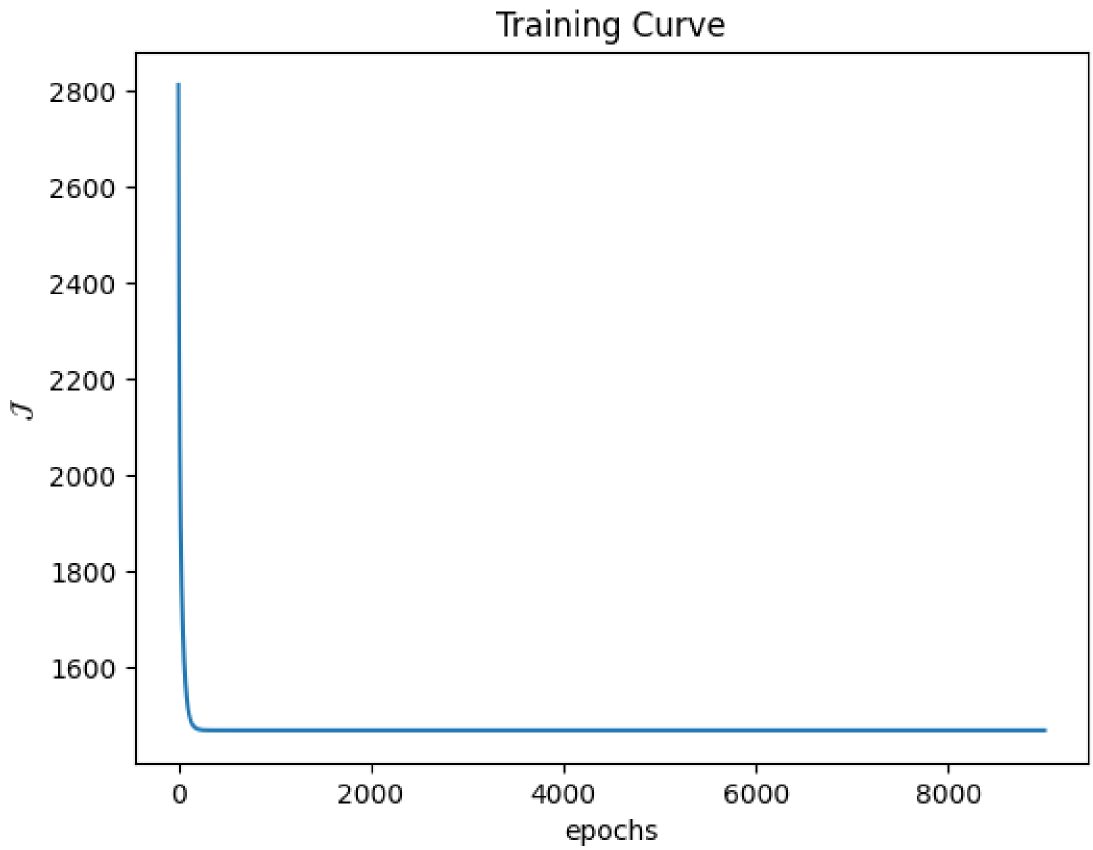


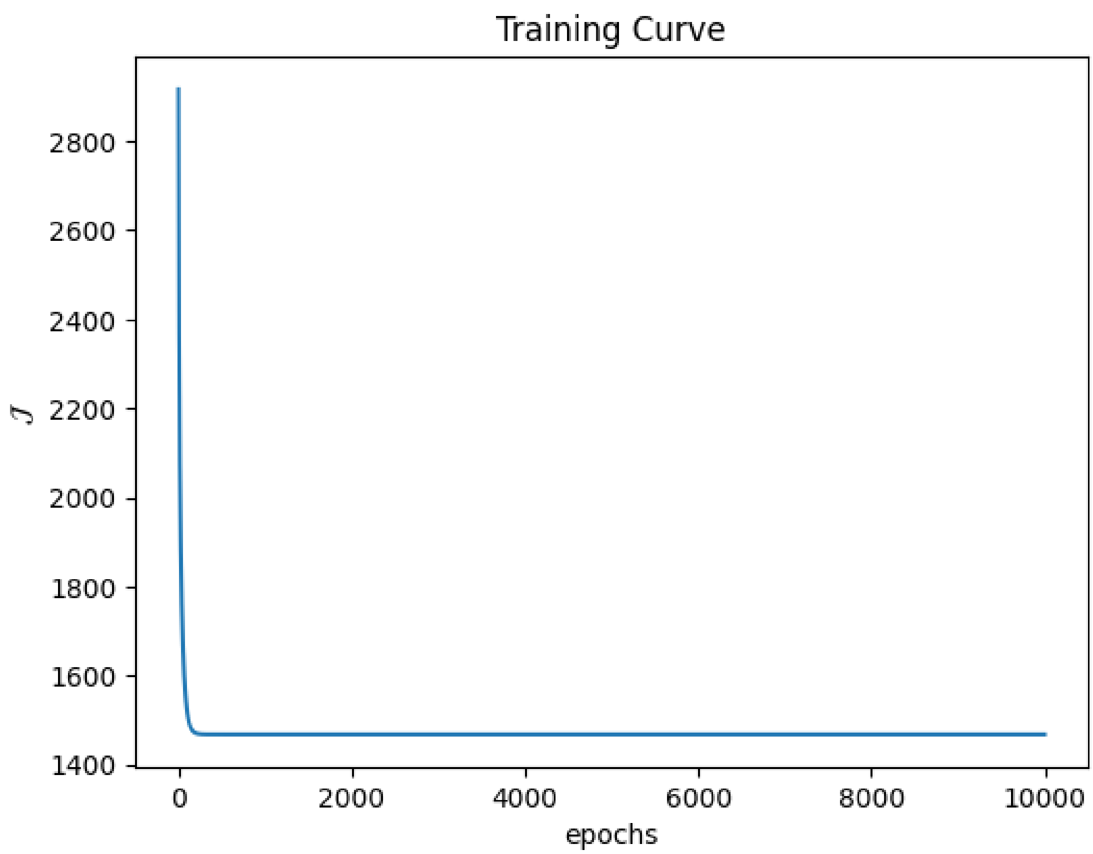
Training Curve
2800
2600
2400
2200
2000
1800
1600
$\mathcal{J}$
0
2000
4000
6000
8000
epochs

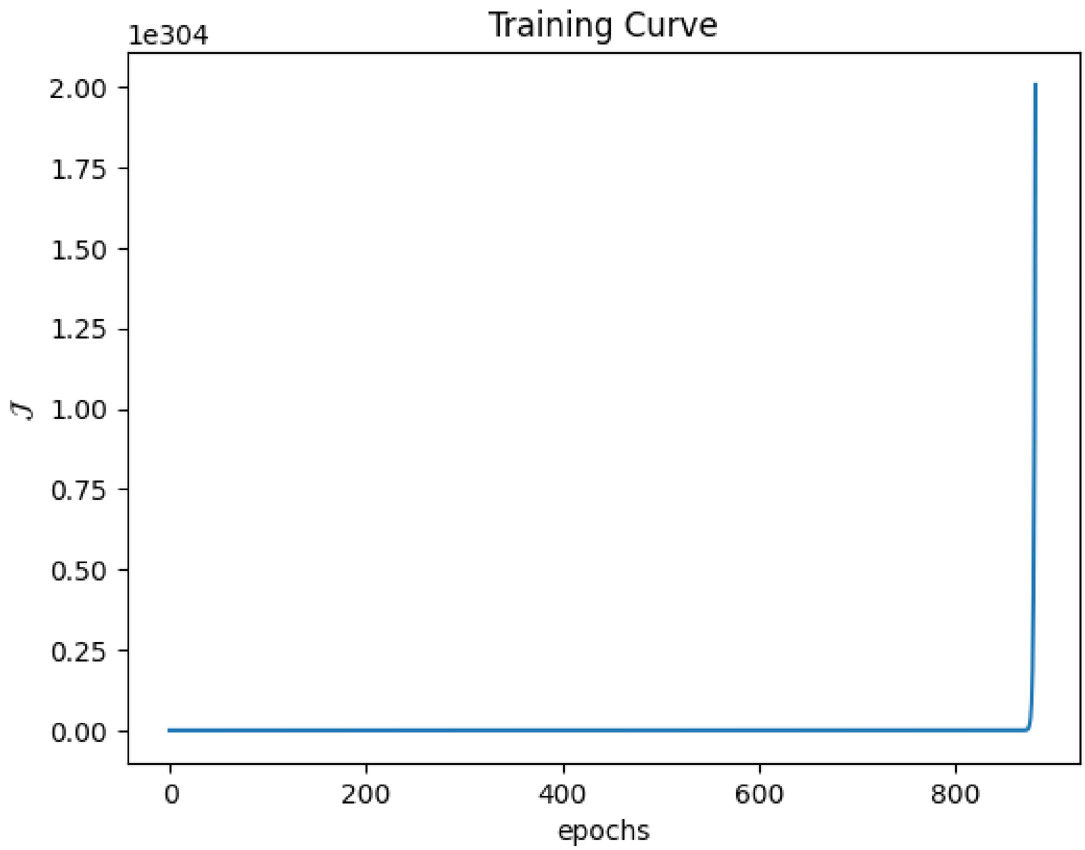

Training Curve
2800
2600
2400
2200
2000
1800
1600
$\mathcal{J}$
0
2000
4000
6000
8000
10000
epochs

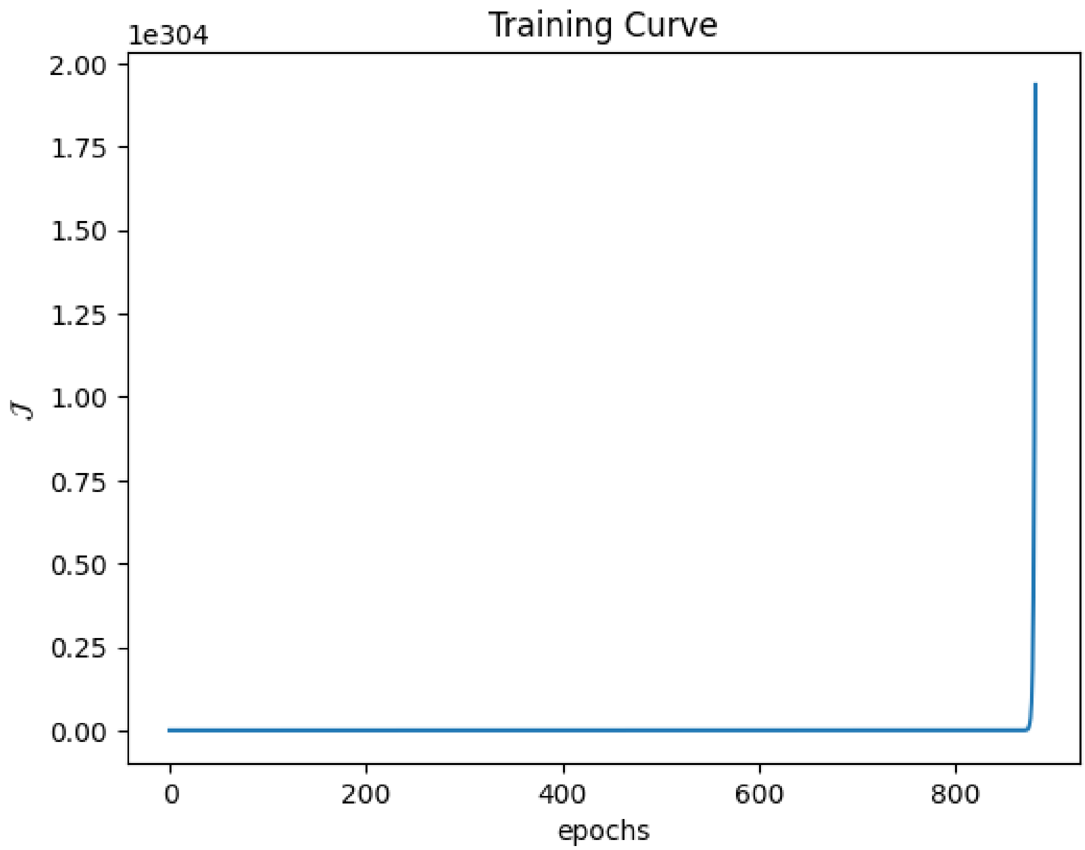

Training Curve
2800
2600
2400
2200
2000
1800
1600
$\mathcal{J}$
0
1000
2000
3000
4000
5000
6000
epochs

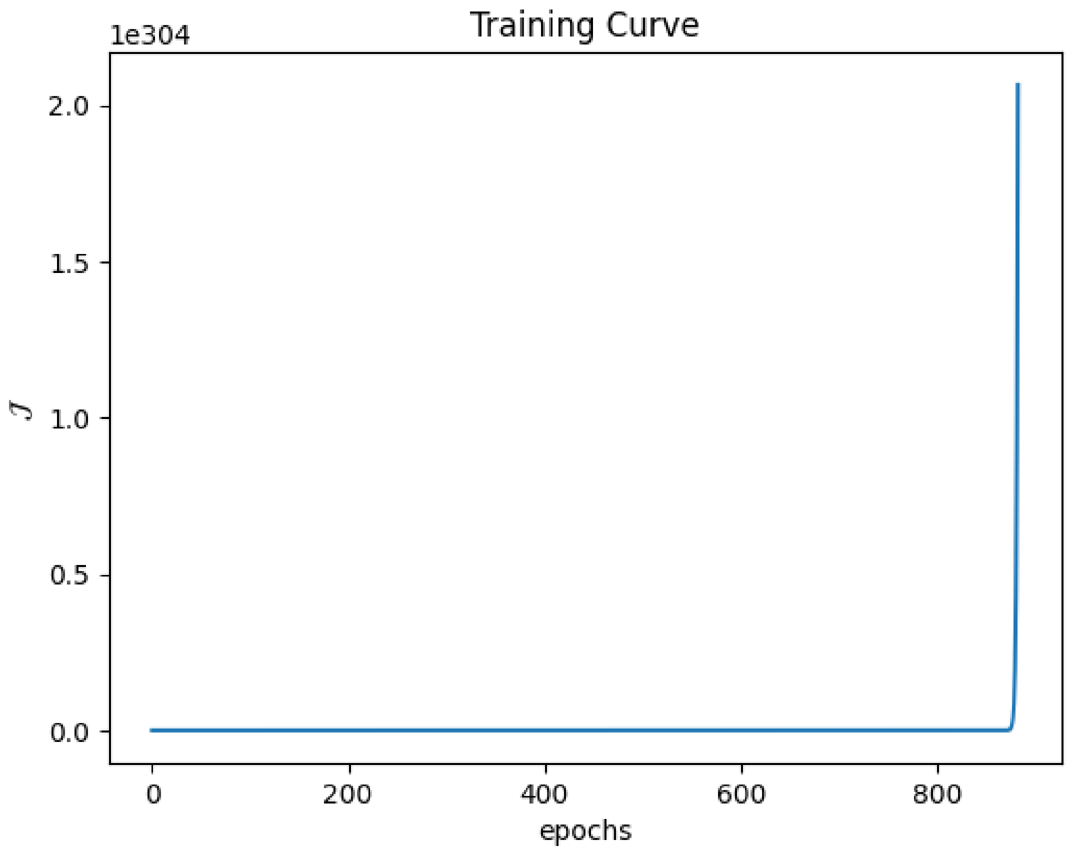

Training Curve
2800
2600
2400
2200
2000
1800
1600
1400
$\mathcal{J}$
0
1000
2000
3000
4000
5000
6000
7000
8000
epochs


Training Curve
2800
2600
2400
2200
2000
1800
1600
$\mathcal{J}$
0
2000
4000
6000
8000
epochs

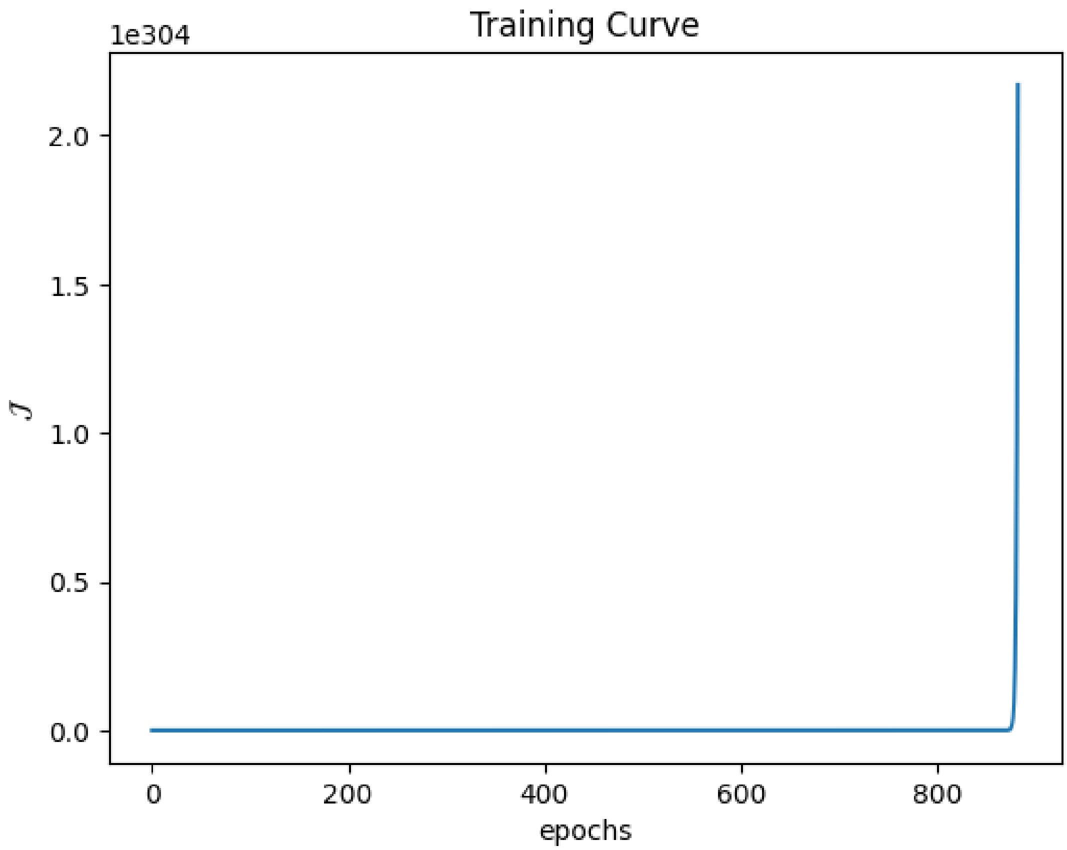

Training Curve
2800
2600
2400
2200
2000
1800
1600
1400
$\mathcal{J}$
0
2000
4000
6000
8000
10000
epochs


Training Curve
1e304
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
$\mathcal{J}$
0
200
400
600
800
epochs


Training Curve
1e304
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
$\mathcal{J}$
0
200
400
600
800
epochs



Training Curve
1e304
2.0
1.5
1.0
0.5
0.0
$\mathcal{J}$
0
200
400
600
800
epochs



6 Testing

[95]:
```
import pandas as pd

def apply_normalization(new_data, params_dict, feature_columns):
    normalized_new_data = []

    for idx, col in enumerate(feature_columns):
        if col in params_dict:
            min_val = params_dict[col]['min_val']
            max_val = params_dict[col]['max_val']
            # Normalize the new data point using the captured min and max values
            normalized_value = (new_data[idx] - min_val) / (max_val - min_val)
            normalized_new_data.append(normalized_value)
        else:
            # If the column was excluded from normalization, retain the
 ↪original value
            normalized_new_data.append(new_data[idx])
```

```
    return normalized_new_data
```

[99]:
```
# Classification
test_input_class = [[3.21, 32.285162, 15393, 85750, 8396, 4, -110.813768, 3,␣
 ↪1995, 3411450]]
feature_columns_class = ['lot_acres', 'latitude', 'taxes', 'zipcode',␣
 ↪'sqrt_ft', 'bedrooms', 'longitude', 'garage', 'year_built', 'sold_price']

# Normalize the test input data
normalized_test_input_class = [apply_normalization(sample,␣
 ↪normalization_params, feature_columns_class) for sample in test_input_class]
y_pred_test_class = knn.predict(normalized_test_input_class, 1)
y_pred_test_class
```

[99]:
```
array([1.])
```

[101]:
```
# Regression
test_input_reg = [[32.285162, 3.21, 15393, 4, -110.813768, 3411450, 6396, 5]]
feature_columns_reg = ['latitude', 'lot_acres', 'taxes', 'bedrooms',␣
 ↪'longitude', 'sold_price', 'sqrt_ft', 'fireplaces']

# Normalize the test input data
normalized_test_input_reg = [apply_normalization(sample, normalization_params,␣
 ↪feature_columns_reg) for sample in test_input_reg]
y_pred_test_reg = knnr.predict(normalized_test_input_reg, 2)
y_pred_test_reg
```

[101]:
```
array([6.25351373])
```