Manuel Agraz Vallejo

AI534 Implementation 1

Deadline: Sunday, Oct. 12, by 11:59pm

Submission Instruction: Submit 1) your completed notebook in ipynb format, and 2) a PDF export of the completed notebook with outputs (the codeblock at the end of the notebook should automatically produce the pdf file).

Overview In this assignment, we will implement and experiment with linear regression models to predict house prices based on various features. We will use the same housing data you explored in the warm-up assignment.

We will implement two versions, one using the closed-form solution, and one using gradient descent.

You may modify the starter code as you see fit, including changing the signatures of functions and adding/removing helper functions. However, please make sure that your TA can understand what you are doing and why.

First lets import the necessary packages and configure the notebook environment.

In [2]:
```
# Install required packages for PDF export (used at the end of the notebook)
# !pip install nbconvert > /dev/null 2>&1
# !pip install pdfkit > /dev/null 2>&1
# !apt-get install -y wkhtmltopdf > /dev/null 2>&1

# Import system and utility libraries
import os
import pdfkit
import contextlib
import sys
# from google.colab import files

# Import data science libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# add more imports if necessary
```

Part 0: (5 pts) Data and preprocessing

Data access

Follow these steps to access the datasets:

1. On Canvas, download the following files:

- `IA1_train.csv` (training data)
- `IA1_val.csv` (validation data)

1. Upload both files to your Google Drive at: `/My Drive/AI534/`
2. Mount Google Drive in Colab using the following code block, which assumes specific file paths for your files.

In [3]:
```
# from google.colab import drive
# drive.mount('/content/gdrive')

# train_path = '/content/gdrive/My Drive/AI534/IA1_train.csv' # DO NOT MODIFY THIS. Please make sure your data
# val_path = '/content/gdrive/My Drive/AI534/IA1_val.csv' # DO NOT MODIFY THIS. Please make sure your data has

train_path = './IA1_train.csv' # DO NOT MODIFY THIS. Please make sure your data has this exact path
val_path = './IA1_dev.csv' # DO NOT MODIFY THIS. Please make sure your data has this exact path
```

Now load the training and validation data.

In [4]:
```
train_df = pd.read_csv(train_path)
val_df = pd.read_csv(val_path)

train_df.head()
```

Out[4]:

	id	date	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	...	sqft_above	sqft_basement
0	7972604355	5/21/2014	3	1.00	1020	7874	1.0	0	0	3	...	1020	0
1	8731951130	6/9/2014	3	2.25	2210	8000	2.0	0	0	4	...	2210	0
2	7885800740	2/18/2015	4	2.50	2350	5835	2.0	0	0	3	...	2350	0
3	4232900940	5/22/2014	3	1.50	1660	4800	2.0	0	0	3	...	1660	0
4	3275850190	9/5/2014	3	2.50	2410	9916	2.0	0	0	4	...	2410	0

5 rows × 21 columns

Preprocessing

Implement the preprocessing function:

1. **Remove** the *ID* column from both training and validation data
2. **Extract date components** Convert the 'date' column into 3 numerical features: 'day', 'month' and 'year'
3. **Create a new feature 'age_since_renovated'** to replace the inconsistent 'yr_renovated'. is set to 0 if the house has not been renovated. This creates an inconsistent meaning to the numerical values. Replace it with a new feature called *age_since_renovated*:

   > if *yr_renovate* != 0
   >
   > > *age_since_renovated = year - yr_renovated*
   >
   > else
   >
   > > *age_since_renovated = year - yr_built*

1. **Normalize features using z-score normalization** (except the target 'price') For each feature 'x': $$ z=\frac{x-\mu}{\sigma} $$

where: μ is the mean of 'x' in the training set σ is the standard deviation of 'x' in the training set

Apply the same μ and σ from the training data to normalize both the training and validation data.

In [5]:

```
def preprocess(train_df, val_df, normalize=True):
    # Your code goes here
    _train_df = train_df.drop(columns="id")
    _val_df = val_df.drop(columns="id")

    #Process date
    _train_df["date"] = pd.to_datetime(_train_df["date"])
    _train_df["day"] = _train_df["date"].dt.day
    _train_df["year"] = _train_df["date"].dt.year
    _train_df["month"] = _train_df["date"].dt.month

    _train_df["age_since_renovated"] = np.where(_train_df["yr_renovated"] != 0, _train_df["year"]-_train_df["yr


    _train_df = _train_df.drop(columns='date')
    _train_df = _train_df.drop(columns='yr_renovated')


    _val_df["date"] = pd.to_datetime(_val_df["date"])
    _val_df["day"] = _val_df["date"].dt.day
    _val_df["year"] = _val_df["date"].dt.year
    _val_df["month"] = _val_df["date"].dt.month

    _val_df["age_since_renovated"] = np.where(_val_df["yr_renovated"] != 0, _val_df["year"]-_val_df["yr_renovat

    _val_df = _val_df.drop(columns='date')
    _val_df = _val_df.drop(columns='yr_renovated')


    #Normalize all columns except price
    if(normalize):
        for col_name in _train_df.drop(columns=['price']).columns:
            mu = _train_df[col_name].mean()
            sigma =_train_df[col_name].std()

            _train_df[col_name] = (_train_df[col_name] - mu) / sigma
            _val_df[col_name] = (_val_df[col_name] - mu) / sigma

    # Add bias column
    bias = pd.Series(1.0, index=_train_df.index, name='bias')
    _train_df = pd.concat([bias, _train_df], axis=1)
```

```
    bias = pd.Series(1.0, index=_val_df.index, name='bias')
    _val_df = pd.concat([bias, _val_df], axis=1)

    return _train_df.drop(columns=['price']), _val_df.drop(columns=['price']), _train_df["price"], _val_df["pri
```

Let's do a quick testing of your normalization, please

1. Estimate and print the new mean and standard deviation of the normalized features for the training data --- this should be 0 and 1 respectively.
2. Estimate and print the new mean and standard deviation of the normalized features for the validation data --- these values will not be 0 and 1, but somewhat close

In [6]:

```
# Apply preprocessing
X_train, X_val, y_train, y_val = preprocess(train_df, val_df)

# Print training set stats
print("Training set (normalized features):")
print("Mean:", X_train.mean().round(2).to_list())
print("Std: ", X_train.std().round(2).to_list())

# Print validation set stats
print("\nValidation set (normalized features):")
print("Mean:", X_val.mean().round(2).to_list())
print("Std: ", X_val.std().round(2).to_list())
```

```
Training set (normalized features):
Mean: [1.0, 0.0, -0.0, 0.0, -0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.0, 0.0, 0.0, 0.0, -0.0, 0.0, 0.0, -0.0, -0.0, 0.0
, -0.0, -0.0]
Std:  [0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0]

Validation set (normalized features):
Mean: [1.0, 0.01, -0.01, 0.01, 0.01, 0.02, 0.02, 0.0, -0.06, 0.06, 0.03, -0.02, 0.02, -0.03, -0.02, 0.02, 0.05,
-0.01, -0.02, 0.01, -0.02, -0.02]
Std:  [0.0, 0.89, 1.0, 0.99, 0.91, 0.99, 1.09, 1.03, 0.98, 1.03, 1.0, 0.98, 1.0, 1.01, 1.0, 0.99, 1.05, 0.79, 1
.01, 1.0, 0.98, 1.0]
```

✍ Question

Why is it import to use the same μ and σ to perform normalization on the training and validation data? What would happen if we use μ and σ estimated using the validation to perform normalization on the validation data?

Answer: If we use the mean and std from the validation data to normalize the validation data, we would be providing information about the validation data through the mean and std. Since we want to validate our model using the validation data, we have to avoid letting the model know how the validation data looks like before hand, we do this by using the training data mean and std. Otherwise our model would "see" how the validation data looks like by the way it is normalized, and this will invalidate any evaluation we do with the validation data.

Part 1 (10 pts) Generate closed-form solution for reference.

Before we implement gradient descent, we'll begin by solving linear regression using the **closed-form solution** as a reference point.

Our data now contains 21 numeric features. Including the bias term w_0, the learned weight vector should have 22 dimensions.

Implement closed-form solution for linear regression

Write a function to compute the weight vector for linear regression using the **closed-form solution** (also known as the normal equation): $$ \mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} $$

You may use NumPy's build-in matrix operations. For numerical stability, we recommend using `np.linalg.pinv()` when computing the inverse.

Your function should take the feature matrix and target vector as input, and return the learned weight vector (\mathbf{w}).

In [7]:

```
def closed_form_linear_regression(X, y):
    """
    Compute weights for linear regression using the closed-form solution.

    Args:
        X (ndarray): Feature matrix of shape (n_samples, n_features)
        y (ndarray): Target vector of shape (n_samples,)

    Returns:
```

```
    w (ndarray): Weight vector of shape (n_features,)
    """
    # Your code goes here

    w=np.linalg.pinv(X.T @ X) @ X.T @ y
    return w
```

Apply and evaluate the model

1. Use your `closed_form_linear_regression()` function to learn weights from the **training data**.
2. Use the learned weights to make predictions on both **training** and **validation** sets.
3. Report the **Mean Squared Error (MSE)** for both sets.
4. Print the learned weight vector (should have 22 values: 21 features + bias).

In [8]:

```
# Your Code goes here
w=closed_form_linear_regression(X_train.to_numpy(), y_train.to_numpy())

y_train_pred = X_train.to_numpy() @ w

mse_train = np.mean((y_train_pred - y_train.to_numpy())**2)

y_val_pred = X_val.to_numpy() @ w

mse_val = np.mean((y_val_pred - y_val.to_numpy())**2)

print(f"MSE_Train: {mse_train}")
print(f"MSE_Val: {mse_val}")

print(f"Learned weight vector shape: {w.shape}, values: {w}")

feature_weight_dict = dict(zip(X_train.columns.tolist(), w.tolist()))
for key in feature_weight_dict:
    print (f"{key}: {feature_weight_dict[key]}")
```

```
MSE_Train: 3.757887089954586
MSE_Val: 4.503508105356855
Learned weight vector shape: (22,), values: [ 5.36167284 -0.28135266  0.3390716   0.76341998  0.05815041  0.018
13676
  0.3281388   0.44675376  0.1998432   1.11544343  0.75623295  0.15546155
 -0.88336171 -0.26341874  0.83661248 -0.30369641  0.14358099 -0.09927428
 -0.05063652  0.17375019  0.05485035 -0.10255779]
bias: 5.361672840000041
bedrooms: -0.2813526588003597
bathrooms: 0.33907160068001474
sqft_living: 0.7634199770395101
sqft_lot: 0.05815041337175126
floors: 0.018136758423721908
waterfront: 0.32813879876650265
view: 0.4467537577479005
condition: 0.19984320339310452
grade: 1.1154434291620294
sqft_above: 0.7562329454505428
sqft_basement: 0.15546154848496263
yr_built: -0.8833617141584924
zipcode: -0.2634187373257927
lat: 0.8366124800413339
long: -0.3036964061101125
sqft_living15: 0.1435809936127316
sqft_lot15: -0.09927428347952527
day: -0.05063651693935773
year: 0.17375019364819907
month: 0.054850349766602025
age_since_renovated: -0.10255779001893855
```

✍ Question

The learned feature weights are often used to understand the importance of the features. The sign of the weights indicates if a feature positively or negatively impact the price, and the magnitude suggests the strength of the impact. Does the sign of all the features match your expection based on your common-sense understanding of what makes a house expensive? Please hightlight any surprises from the results.

Answer: Generally the features match my expectation of what makes a house expensive, however there are some features that caught my attention. First I was expecting the number of bedrooms to have a positive impact on the price however in this dataset it has a negative impact with a weight of -0.28. I was surprised to see that the largest positive weight was attributed to the grade given to the house. I was also surprised by the small positive impact of sqft_lot, i was expecting the size of the land to be of much more importance to the price.

Part 2 (35 pts) Implement and experiment with batch gradient

descent

In this part, you will implement batch gradient descent for linear regression and experiment with it on the given data.

Implement 'batch_gradient_descent' function

Your function should take following **inputs:**

- `X` : training feature matrix (shape: n_samples × d)
- `y` : target vector (shape: n_samples)
- `gamma` : learning rate (\gamma)
- `T` : number of iterations (epochs)
- `epsilon_loss` *(optional)*: convergence threshold for loss (\epsilon_l)
- `epsilon_grad` *(optional)*: convergence threshold for gradient norm (\epsilon_g)

It should output:

1. 'w': the learned $d+1$ - dimensional weight vector
2. 'losses': list of mean squared errors for each training iteration

In [9]:
```
def batch_gradient_descent(X, y, gamma, T, epsilon_loss=None, epsilon_grad=None):
    """
    Perform batch gradient descent for linear regression.

    Args:
        X (ndarray): Feature matrix (n_samples, n_features)
        y (ndarray): Target vector (n_samples,)
        gamma (float): Learning rate
        T (int): Number of iterations (epochs)
        epsilon_loss (float, optional): Convergence threshold for loss
        epsilon_grad (float, optional): Convergence threshold for gradient norm

    Returns:
        w (ndarray): Learned weight vector (d+1, includes bias)
        losses (list): MSE loss at each epoch
    """
    # Your code goes here
    N,d = X.shape
    w = np.random.normal(0, 0.01, d)
    losses = []
    for epoch in range(T):

        #Compute prediction
        y_pred = X @ w

        #Calculate loss
        mse_loss = np.mean((y_pred-y)**2)

        #Calculate gradient of the loss
        gradient_mse = (2/N) * X.T @ (y_pred-y)

        #Perform gradient descent update
        w -= gamma * gradient_mse

        #Store loss
        losses.append(mse_loss)

        if epsilon_loss is not None and epoch > 0:
            if abs(losses[-1] - losses[-2]) < epsilon_loss:
                print(f"Converged at epoch {epoch}")
                break

        if epsilon_grad is not None:
            if np.linalg.norm(gradient_mse) < epsilon_grad:
                print(f"Gradient converged at epoch {epoch}")
                break

    return w, losses
```

Experiment with different learning rate

Use your 'batch_gradient_descent' function to

1. Train models on the training data with learning rates $\gamma = 10^{-i}$ for $i = 0, 1, 2, 3, 4$.
2. Train for up to 3000 iterations (stop early if the loss converges or diverges).
3. For each converging (not necessarily converged yet) learning rate, compute and report the final MSE on the **validation set**.
4. Plot the **training loss curves** (MSE vs. iterations) for all converging learning rates.

- Use different colors for each learning rate
- Include a legend

In [10]:
```python
learning_rates = [10**-i for i in range(0,5)]

w_list = []
train_losses_list = []
converged_rates = []
val_mse_list = []

for i, lr in enumerate(learning_rates):
    print(f"Learning rate: {lr}")

    w, losses = batch_gradient_descent(X_train.to_numpy(), y_train.to_numpy(), gamma=lr, T=3000)

    # Check if converged (no NaN values)
    if not np.any(np.isnan(w)) and not np.any(np.isnan(losses)):
        w_list.append(w)
        train_losses_list.append(losses)  # Fix: was appending w instead of losses
        converged_rates.append(lr)

        # Calculate validation MSE
        y_val_pred = X_val.to_numpy() @ w
        val_mse = np.mean((y_val_pred - y_val.to_numpy())**2)
        val_mse_list.append(val_mse)

print(f"\nConverged learning rates: {converged_rates}")

# Plot training loss curves for converging learning rates
plt.figure(figsize=(12, 8))
colors = ['red', 'blue', 'green', 'orange', 'purple']

for i, (lr, losses) in enumerate(zip(converged_rates, train_losses_list)):
    plt.plot(losses, color=colors[i % len(colors)], label=f'γ = {lr}', linewidth=2)

plt.xlabel('Epoch')
plt.ylabel('Training MSE Loss')
plt.title('Training Loss Curves for Different Learning Rates')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Validation MSE for each converging learning rate
print("\nValidation MSE Results:")
print("-" * 30)
for lr, train_mse, val_mse in zip(converged_rates, train_losses_list, val_mse_list):
    print(f"Learning rate {lr}: Training MSE = {train_mse[-1]:.6f}, Validation MSE = {val_mse:.6f}")
```

```
Learning rate: 1
/home/magraz/venvs/ml_class/lib/python3.12/site-packages/numpy/_core/_methods.py:134: RuntimeWarning: overflow
encountered in reduce
  ret = umr_sum(arr, axis, dtype, out, keepdims, where=where)
/tmp/ipykernel_27290/4213706113.py:27: RuntimeWarning: overflow encountered in square
  mse_loss = np.mean((y_pred-y)**2)
/tmp/ipykernel_27290/4213706113.py:24: RuntimeWarning: overflow encountered in matmul
  y_pred = X @ w
/tmp/ipykernel_27290/4213706113.py:30: RuntimeWarning: invalid value encountered in matmul
  gradient_mse = (2/N) * X.T @ (y_pred-y)
Learning rate: 0.1
Learning rate: 0.01
Learning rate: 0.001
Learning rate: 0.0001

Converged learning rates: [0.1, 0.01, 0.001, 0.0001]
```


```
Validation MSE Results:
------------------------------
Learning rate 0.1: Training MSE = 3.757887, Validation MSE = 4.503508
Learning rate 0.01: Training MSE = 3.757889, Validation MSE = 4.503499
Learning rate 0.001: Training MSE = 3.781112, Validation MSE = 4.525632
Learning rate 0.0001: Training MSE = 13.157934, Validation MSE = 14.424164
```

✍ Question

Which learning rate leads to the best training and validation MSE respectively? Do you observe better training MSE tend to correpsond to better validation MSE? How is this different from the trend shown on page 52 (or vicinity) of the lecture slides (titled 'danger of using training loss to select M') regarding overfitting? Is there any issue with using training loss to pick learning rate in this case?

Answer: The 0.1 learning rate leads to the best training MSE, while the learning rate of 0.01 leads to the best validation MSE. In this case better training MSE does tend to correspond to better validation MSE. The lecture slides mention that this is not always the case and good training MSE performance might be caused by overfitting leading to bad validation MSE performance. In this case there's no issue in using the training loss to pick a learning rate since we do not observe overfitting, this shown by how the validation loss remains very close to the training loss.

Part 3. More exploration

Part 3: More exploration.

3(a). (25 pts) Normalization of features: what is the impact?

In part 1, you were asked to perform z-score normalization of all the features. In this part, we will ask you to first conceptually think about what is the impact this operation on the solution and then use some experiments to varify your conceptual understanding.

✍ Questions.

The normalization process applies a linear transformation to each feature, where the transformed feature x' is simply a linear function of original feature x: $x'=\frac{x-\mu}{\sigma}$.

Let's disect the influence of this transformation on our learned linear regression model.

1. How do you think this transformation will influnce the training and validation MSE we get for the closed-form solution? Why?
2. How do you think this will change the magnitude of the weights of the learned model? Why?
3. How do you think this will change the convergence behavior of the batch gradient descent algorithm? Why?

Answer

1.- It does not directly affect the training and validation MSE as normalization is only a linear transformation it doesn't affect the expressive power of the model.

2.-Normalization is helping by keeping the scale of the input bounded and thus keeping the scale of the weights similar to each other. This is because without normalization features at different scales for example on the 1000-2000 would require much smaller weights than features on a range from 1-10 in order to lead to the same loss. This can lead to weights becoming numerically unstable reaching really high or small values.

3.-Since weights will be similar to each other in scale, the gradients will be smaller and not explode leading to divergence as we saw with the learning rate of 1.

Experimental verification

Now please perform the following experiments to verify your answer to the above questions.

1. Apply 'closed_form_linear_regression' to training data that did not go through the feature normalization step, and report the learned weights and the resulting training and testing MSEs.

2. Apply 'batch_gradient_descent' to training data that did not go through the feature normalization step using different learning rates. Note that the learning rate used in previous section will no longer work here. You will need to search for an appropriate learning rate to get some converging behavior. Plot your MSE loss curve as a function of the epochs once you identify a convergent learning rate. Hint: the learning rate needs to be much, much,much, much, much, much, much smaller (think about each much as an order of manitude) than what was used in part 2). Also unless you let it run for a long time, it is unlikely to converge to the same level of loss values. So use a reasonable upper bound on the # of iterations so that it won't take forever.

In [11]:
```
X_train, X_val, y_train, y_val = preprocess(train_df, val_df, normalize=False)

w=closed_form_linear_regression(X_train.to_numpy(), y_train.to_numpy())

y_train_pred = X_train.to_numpy() @ w

mse_train = np.mean((y_train_pred - y_train.to_numpy())**2)

y_val_pred = X_val.to_numpy() @ w

mse_val = np.mean((y_val_pred - y_val.to_numpy())**2)

print(f"MSE_Train: {mse_train}")
print(f"MSE_Val: {mse_val}")

print(f"Learned weight vector: {w}")

feature_weight_dict = dict(zip(X_train.columns.tolist(), w.tolist()))
for key in feature_weight_dict:
    print (f"{key}: {feature_weight_dict[key]}")
```

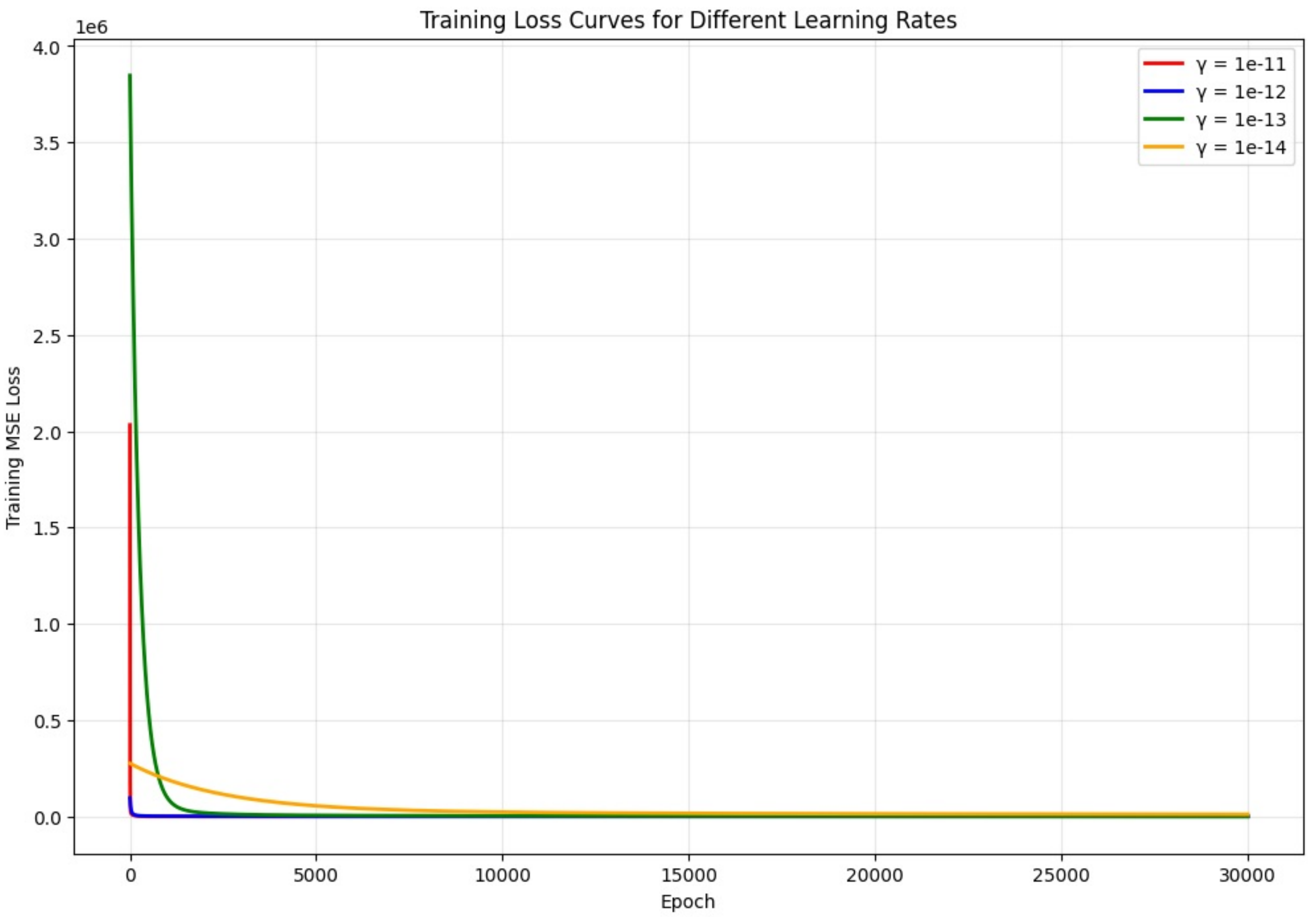
```
MSE_Train: 3.769005285215598
MSE_Val: 4.516144824761274
Learned weight vector: [ 1.80833197e-04 -2.94943580e-01  4.40041821e-01  9.84276146e-04
  1.34948679e-06  4.48302413e-02  4.01762150e+00  5.98760319e-01
  2.90711780e-01  9.51607366e-01  7.63596560e-04  2.20678882e-04
 -3.01274281e-02 -5.53837323e-03  6.00871553e+00 -2.13593881e+00
  1.94481317e-04 -3.43347448e-06 -6.96949118e-03  2.44379973e-02
 -2.32014597e-02 -3.16835515e-03]
bias: 0.00018083319687162147
bedrooms: -0.29494358028579304
bathrooms: 0.44004182081116394
sqft_living: 0.0009842761457062909
sqft_lot: 1.3494867898084868e-06
floors: 0.04483024129080991
waterfront: 4.017621504246485
view: 0.5987603189487873
condition: 0.2907117804141296
grade: 0.9516073660152577
sqft_above: 0.0007635965602713692
sqft_basement: 0.00022067888243741148
yr_built: -0.030127428104602904
zipcode: -0.005538373225630897
lat: 6.00871552746392
long: -2.135938808832332
sqft_living15: 0.00019448131714775948
sqft_lot15: -3.4334744771632282e-06
day: -0.006969491179073965
year: 0.024437997261595394
month: -0.023201459685648535
age_since_renovated: -0.0031683551549142788
```

In [12]:

```python
# Set random seed for reproducibility
np.random.seed(42)

learning_rates = [10**-i for i in range(10,15)]

w_list = []
train_losses_list = []
converged_rates = []
val_mse_list = []



for i, lr in enumerate(learning_rates):
    print(f"Learning rate: {lr}")

    w, losses = batch_gradient_descent(X_train.to_numpy(), y_train.to_numpy(), gamma=lr, T=30000)

    # Check if converged (no NaN or inf values)
    if not (np.any(np.isnan(w)) or np.any(np.isinf(w))) and not (np.any(np.isnan(losses)) or np.any(np.isinf(lo
        w_list.append(w)
        train_losses_list.append(losses)
        converged_rates.append(lr)

        # Calculate validation MSE
        y_val_pred = X_val.to_numpy() @ w
        val_mse = np.mean((y_val_pred - y_val.to_numpy())**2)
        val_mse_list.append(val_mse)

print(f"\nConverged learning rates: {converged_rates}")

# Plot training loss curves for converging learning rates
plt.figure(figsize=(12, 8))
colors = ['red', 'blue', 'green', 'orange', 'purple']

for i, (lr, losses) in enumerate(zip(converged_rates, train_losses_list)):
    plt.plot(losses, color=colors[i % len(colors)], label=f'γ = {lr}', linewidth=2)

plt.xlabel('Epoch')
plt.ylabel('Training MSE Loss')
plt.title('Training Loss Curves for Different Learning Rates')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Validation MSE for each converging learning rate
print("\nValidation MSE Results:")
print("-" * 30)
for lr, train_mse, val_mse in zip(converged_rates, train_losses_list, val_mse_list):
    print(f"Learning rate {lr}: Training MSE = {train_mse[-1]:.6f}, Validation MSE = {val_mse:.6f}")
```

```
Learning rate: 1e-10
```

```
/home/magraz/venvs/ml_class/lib/python3.12/site-packages/numpy/_core/_methods.py:134: RuntimeWarning: overflow encountered in reduce
  ret = umr_sum(arr, axis, dtype, out, keepdims, where=where)
/tmp/ipykernel_27290/4213706113.py:27: RuntimeWarning: overflow encountered in square
  mse_loss = np.mean((y_pred-y)**2)
/tmp/ipykernel_27290/4213706113.py:30: RuntimeWarning: overflow encountered in matmul
  gradient_mse = (2/N) * X.T @ (y_pred-y)
/tmp/ipykernel_27290/4213706113.py:30: RuntimeWarning: invalid value encountered in matmul
  gradient_mse = (2/N) * X.T @ (y_pred-y)
/tmp/ipykernel_27290/4213706113.py:33: RuntimeWarning: invalid value encountered in subtract
  w -= gamma * gradient_mse
Learning rate: 1e-11
Learning rate: 1e-12
Learning rate: 1e-13
Learning rate: 1e-14

Converged learning rates: [1e-11, 1e-12, 1e-13, 1e-14]
```




```
Validation MSE Results:
------------------------------
Learning rate 1e-11: Training MSE = 52.781697, Validation MSE = 59.767885
Learning rate 1e-12: Training MSE = 166.220369, Validation MSE = 151.463909
Learning rate 1e-13: Training MSE = 438.441818, Validation MSE = 360.814639
Learning rate 1e-14: Training MSE = 11144.328994, Validation MSE = 10365.507741
```

✍ Questions

Please revisit the questions above. Does your experiment confirm your expectation? Can you provide explanations to the observed differences (or lack of differences) between the normalized data and unnormalized data? Based on these observations and your understanding of them, please comment on the benefits of normalizing the input features in learning for linear regressions.

Answer

In my answer to the first question I expected the loss to remain similar and only the weights to change in magnitude. The experiment confirmed my expectation since the loss for both train and validation are very similar when using the normalized and non-normalized data. The main change is in the weights, the normalized weights have values in a range [-0.88, 1.17], while the unnormalized weights have values in the range [-2.14, 6.0]. Some weights of the unnormalized weight set are very small such as the sqft_lot15 weight of -3.4334744771632282e-06 these very small weights can cause numerical stability problems when calculating the gradients. Normalization can help prevent this by keeping the weights similar in scale.

3(b). (15 pts) Explore the impact of correlated features

In the warm up exercise, you all have seen some features are highly correlated with one another. For example, there are multiple squared footage related features that are strongly correlated (e.g., *sqft_above* and *sqrt_living* has a correlation coefficient of 0.878). This is referred to as multicollinearity phenomeon, where two or more features are correlated.

There are numerous consequences from multicollinearity. It makes it more challenging to estimate the weights of the features accurately. The weights may become unstable, and their interpretation becomes less clear.

In this part you will work with the pre-processed training set, and perform the following experiments to examine how correlated features affect the stability of learned weights.

Experiment to investigate impact of correlated features

Conduct following experiments.

1. **Create five training subsets**: Randomly subsample 75% of the orginial preprocessed training set to form five slightly different training sets.
2. **Fit models**: Use your 'closed_form_linear_regression' function to train a linear regression model on each of the five training sets.
3. **Report learned weights in a table**:
   - The table should have **five rows** (one for each model)
   - Each column corresponds to a **feature's weight**
   - Include a **header row** with the feature names
4. **Report the variance of weights across models**:
   Include an additional row to the above table to report for each feature, the variance of its learned weight coefficients across the five models. This variance serves as a measure of the **stability** of the weight assigned to each feature. Larger variance suggests lower stability.

   Note: We use 5 random training subset here to get a rough sense of weight stability. For more robust analysis, you could increase this to 10 or more runs.

In [13]:
```
X_train, X_val, y_train, y_val = preprocess(train_df, val_df)

# Set random seed for reproducibility
np.random.seed(42)

# Create five training subsets (75% of original data each)
n_subsets = 5
subset_size = int(0.75 * len(X_train))
weights_list = []
feature_names = X_train.columns.tolist()

print(f"Original training set size: {len(X_train)}")
print(f"Each subset size: {subset_size} (75% of original)")
print()

# Generate and train on five subsets
for i in range(n_subsets):
    # Randomly sample 75% of the data
    indices = np.random.choice(len(X_train), size=subset_size, replace=False)

    X_subset = X_train.iloc[indices]
    y_subset = y_train.iloc[indices]

    # Train model on this subset
    w = closed_form_linear_regression(X_subset.to_numpy(), y_subset.to_numpy())
    weights_list.append(w)

# Convert to numpy array for easier manipulation
weights_array = np.array(weights_list)

# Create DataFrame for better visualization
weights_df = pd.DataFrame(weights_array,
                          columns=feature_names,
                          index=[f'Model_{i+1}' for i in range(n_subsets)])

# Calculate variance for each feature across models
variance_row = weights_array.var(axis=0)
variance_df = pd.DataFrame([variance_row],
                           columns=feature_names,
                           index=['Variance'])

# Combine weights and variance
results_df = pd.concat([weights_df, variance_df])

#Modify pandas display options to show all columns
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
```

```python
pd.set_option('display.max_colwidth', None)

print("\nLearned weights for each model and their variances:")
print("=" * 80)
print(results_df.round(8))

# Show features with highest variance (least stable)
print(f"\nTop 7 features with highest weight variance (least stable):")
print("-" * 60)
variance_sorted = variance_row.argsort()[::-1]
for i in range(7):
    feature_idx = variance_sorted[i]
    feature_name = feature_names[feature_idx]
    variance_val = variance_row[feature_idx]
    print(f"{i+1}. {feature_name}: {variance_val:.6f}")

# Show features with lowest variance (most stable)
print(f"\nTop 7 features with lowest weight variance (most stable):")
print("-" * 60)
for i in range(7):
    feature_idx = variance_sorted[-(i+1)]
    feature_name = feature_names[feature_idx]
    variance_val = variance_row[feature_idx]
    print(f"{i+1}. {feature_name}: {variance_val:.6f}")
```

```
Original training set size: 8000
Each subset size: 6000 (75% of original)


Learned weights for each model and their variances:
================================================================================
              bias  bedrooms  bathrooms  sqft_living  sqft_lot    floors  \
Model_1   5.378996 -0.253589   0.320739     0.767621  0.049930  0.001440
Model_2   5.374076 -0.388834   0.353622     0.805095  0.069618  0.018646
Model_3   5.371488 -0.271897   0.297439     0.821282  0.003784  0.009480
Model_4   5.374208 -0.256012   0.310755     0.737472  0.091133  0.056506
Model_5   5.377052 -0.280844   0.340964     0.776631  0.073349  0.004026
Variance  0.000007  0.002532   0.000411     0.000859  0.000894  0.000405

          waterfront      view  condition     grade  sqft_above  \
Model_1     0.338428  0.467218   0.208293  1.144133    0.774589
Model_2     0.348595  0.450986   0.188901  1.121650    0.796605
Model_3     0.315468  0.473120   0.215730  1.101628    0.802855
Model_4     0.405443  0.357503   0.192420  1.137622    0.721430
Model_5     0.385483  0.389617   0.175580  1.130621    0.766040
Variance    0.001057  0.002109   0.000204  0.000218    0.000831

          sqft_basement  yr_built   zipcode       lat      long  \
Model_1        0.129287 -0.858731 -0.236295  0.845090 -0.278649
Model_2        0.165683 -0.998953 -0.268564  0.819634 -0.298739
Model_3        0.187611 -0.855085 -0.268485  0.842254 -0.306449
Model_4        0.167503 -0.986954 -0.271158  0.828215 -0.280617
Model_5        0.164397 -0.904618 -0.287925  0.845222 -0.302188
Variance       0.000354  0.003783  0.000280  0.000107  0.000131

          sqft_living15  sqft_lot15       day      year     month  \
Model_1        0.086692   -0.088918 -0.069710  0.154263  0.050610
Model_2        0.144418   -0.125605 -0.037518  0.175336  0.062571
Model_3        0.100404   -0.061678 -0.054093  0.190032  0.064391
Model_4        0.173754   -0.132565 -0.079551  0.184103  0.061128
Model_5        0.162915   -0.122468 -0.048618  0.181234  0.065850
Variance       0.001178    0.000723  0.000225  0.000152  0.000029

          age_since_renovated
Model_1             -0.093015
Model_2             -0.179818
Model_3             -0.104203
Model_4             -0.174355
Model_5             -0.095625
Variance             0.001533

Top 7 features with highest weight variance (least stable):
------------------------------------------------------------
1. yr_built: 0.003783
2. bedrooms: 0.002532
3. view: 0.002109
4. age_since_renovated: 0.001532
5. sqft_living15: 0.001178
6. waterfront: 0.001057
7. sqft_lot: 0.000894

Top 7 features with lowest weight variance (most stable):
------------------------------------------------------------
1. bias: 0.000007
2. month: 0.000029
3. lat: 0.000107
4. long: 0.000131
5. year: 0.000152
6. condition: 0.000204
7. grade: 0.000218
```

In [14]:

```python
correlation_matrix = X_train.corr()

# Display correlation matrix
print("Feature Correlation Matrix:")
print("=" * 80)
print(correlation_matrix.round(3))

# Find highly correlated feature pairs (correlation > 0.7)
high_corr_pairs = []
for i in range(len(correlation_matrix.columns)):
    for j in range(i+1, len(correlation_matrix.columns)):
        corr_val = correlation_matrix.iloc[i, j]
        if abs(corr_val) > 0.7:
            high_corr_pairs.append((
                correlation_matrix.columns[i],
                correlation_matrix.columns[j],
                corr_val
            ))

print(f"\nHighly correlated feature pairs (|correlation| > 0.7):")
print("-" * 60)
for feat1, feat2, corr in high_corr_pairs:
    print(f"{feat1} <-> {feat2}: {corr:.3f}")
```

```
Feature Correlation Matrix:
================================================================================
                     bias  bedrooms  bathrooms  sqft_living  sqft_lot  floors  \
bias                  NaN       NaN        NaN          NaN       NaN     NaN
bedrooms              NaN     1.000      0.484        0.562     0.025   0.165
bathrooms             NaN     0.484      1.000        0.750     0.071   0.511
sqft_living           NaN     0.562      0.750        1.000     0.165   0.354
sqft_lot              NaN     0.025      0.071        0.165     1.000  -0.013
floors                NaN     0.165      0.511        0.354    -0.013   1.000
waterfront            NaN    -0.011      0.027        0.068     0.035   0.015
view                  NaN     0.087      0.186        0.281     0.078   0.036
condition             NaN     0.027     -0.140       -0.065    -0.013  -0.267
grade                 NaN     0.339      0.664        0.758     0.102   0.466
sqft_above            NaN     0.463      0.685        0.879     0.177   0.522
sqft_basement         NaN     0.293      0.263        0.417     0.007  -0.253
yr_built              NaN     0.143      0.510        0.315     0.048   0.489
zipcode               NaN    -0.152     -0.199       -0.194    -0.124  -0.060
lat                   NaN    -0.020      0.019        0.037    -0.087   0.051
long                  NaN     0.127      0.220        0.240     0.209   0.122
sqft_living15         NaN     0.382      0.567        0.762     0.139   0.280
sqft_lot15            NaN     0.026      0.083        0.179     0.774  -0.019
day                   NaN     0.002     -0.002       -0.009     0.005  -0.003
year                  NaN    -0.005     -0.032       -0.030     0.005  -0.040
month                 NaN    -0.010      0.005        0.010    -0.007   0.031
age_since_renovated   NaN    -0.157     -0.537       -0.335    -0.048  -0.505

                     waterfront   view  condition  grade  sqft_above  \
bias                        NaN    NaN        NaN    NaN         NaN
bedrooms                 -0.011  0.087      0.027  0.339       0.463
bathrooms                 0.027  0.186     -0.140  0.664       0.685
sqft_living               0.068  0.281     -0.065  0.758       0.879
sqft_lot                  0.035  0.078     -0.013  0.102       0.177
floors                    0.015  0.036     -0.267  0.466       0.522
waterfront                1.000  0.377      0.027  0.041       0.059
view                      0.377  1.000      0.055  0.247       0.172
condition                 0.027  0.055      1.000 -0.146      -0.160
grade                     0.041  0.247     -0.146  1.000       0.755
sqft_above                0.059  0.172     -0.160  0.755       1.000
sqft_basement             0.029  0.260      0.168  0.146      -0.068
yr_built                 -0.047 -0.052     -0.384  0.448       0.422
zipcode                   0.041  0.073      0.023 -0.190      -0.260
lat                      -0.021 -0.004      0.001  0.109      -0.009
long                     -0.049 -0.075     -0.121  0.207       0.343
sqft_living15             0.063  0.277     -0.098  0.713       0.738
sqft_lot15                0.023  0.064     -0.006  0.115       0.191
day                       0.009 -0.005      0.002 -0.017      -0.003
year                     -0.010 -0.001     -0.030 -0.044      -0.032
month                     0.018 -0.003      0.017  0.019       0.017
age_since_renovated       0.031  0.020      0.416 -0.461      -0.431

                     sqft_basement  yr_built  zipcode    lat   long  \
bias                           NaN       NaN      NaN    NaN    NaN
bedrooms                     0.293     0.143   -0.152 -0.020  0.127
bathrooms                    0.263     0.510   -0.199  0.019  0.220
sqft_living                  0.417     0.315   -0.194  0.037  0.240
sqft_lot                     0.007     0.048   -0.124 -0.087  0.209
floors                      -0.253     0.489   -0.060  0.051  0.122
waterfront                   0.029    -0.047    0.041 -0.021 -0.049
view                         0.260    -0.052    0.073 -0.004 -0.075
condition                    0.168    -0.384    0.023  0.001 -0.121
grade                        0.146     0.448   -0.190  0.109  0.207
sqft_above                  -0.068     0.422   -0.260 -0.009  0.343
sqft_basement                1.000    -0.144    0.090  0.096 -0.153
yr_built                    -0.144     1.000   -0.346 -0.143  0.407
zipcode                      0.090    -0.346    1.000  0.255 -0.567
lat                          0.096    -0.143    0.255  1.000 -0.139
long                        -0.153     0.407   -0.567 -0.139  1.000
sqft_living15                0.188     0.327   -0.278  0.042  0.338
sqft_lot15                   0.011     0.068   -0.140 -0.073  0.249
day                         -0.014     0.016   -0.006 -0.028 -0.007
year                        -0.001    -0.003    0.020 -0.038 -0.012
month                       -0.012    -0.006   -0.009  0.016 -0.008
age_since_renovated          0.119    -0.913    0.324  0.133 -0.387

                     sqft_living15  sqft_lot15    day   year  month  \
bias                           NaN         NaN    NaN    NaN    NaN
bedrooms                     0.382       0.026  0.002 -0.005 -0.010
bathrooms                    0.567       0.083 -0.002 -0.032  0.005
sqft_living                  0.762       0.179 -0.009 -0.030  0.010
sqft_lot                     0.139       0.774  0.005  0.005 -0.007
floors                       0.280      -0.019 -0.003 -0.040  0.031
waterfront                   0.063       0.023  0.009 -0.010  0.018
view                         0.277       0.064 -0.005 -0.001 -0.003
condition                   -0.098      -0.006  0.002 -0.030  0.017
grade                        0.713       0.115 -0.017 -0.044  0.019
sqft_above                   0.738       0.191 -0.003 -0.032  0.017
sqft_basement                0.188       0.011 -0.014 -0.001 -0.012
yr_built                     0.327       0.068  0.016 -0.003 -0.006
```

```
zipcode                    -0.278      -0.140 -0.006  0.020 -0.009
lat                         0.042      -0.073 -0.028 -0.038  0.016
long                        0.338       0.249 -0.007 -0.012 -0.008
sqft_living15               1.000       0.171 -0.010 -0.035  0.014
sqft_lot15                  0.171       1.000 -0.002  0.002 -0.006
day                        -0.010      -0.002  1.000 -0.007 -0.064
year                       -0.035       0.002 -0.007  1.000 -0.780
month                       0.014      -0.006 -0.064 -0.780  1.000
age_since_renovated        -0.325      -0.066 -0.014  0.025 -0.009

                     age_since_renovated
bias                                 NaN
bedrooms                          -0.157
bathrooms                         -0.537
sqft_living                       -0.335
sqft_lot                          -0.048
floors                            -0.505
waterfront                         0.031
view                               0.020
condition                          0.416
grade                             -0.461
sqft_above                        -0.431
sqft_basement                      0.119
yr_built                          -0.913
zipcode                            0.324
lat                                0.133
long                              -0.387
sqft_living15                     -0.325
sqft_lot15                        -0.066
day                               -0.014
year                               0.025
month                             -0.009
age_since_renovated                1.000

Highly correlated feature pairs (|correlation| > 0.7):
------------------------------------------------------------
bathrooms <-> sqft_living: 0.750
sqft_living <-> grade: 0.758
sqft_living <-> sqft_above: 0.879
sqft_living <-> sqft_living15: 0.762
sqft_lot <-> sqft_lot15: 0.774
grade <-> sqft_above: 0.755
grade <-> sqft_living15: 0.713
sqft_above <-> sqft_living15: 0.738
yr_built <-> age_since_renovated: -0.913
year <-> month: -0.780
```

✍ Questions

Ideally, we want the learned weight coefficients to be **stable across different runs**, as this indicates a more **reliable and interpretable** model.

- Based on the variances you computed:
  - Do features with **high correlation to others** tend to show **more instability** in their weights across different training subsets?
  - What trends do you observe?
- Use a **correlation matrix** of the input features to support your observations. Which features appear most correlated?
- What implications does this have for interpreting feature importance in your model?

Answer:

For the most part, highly correlated features tend to show more variance in their weights. Looking at the sqft_living and grade features both of which are highly correlated, we can see that their variance is among the top 7 highest. The same can be said for the features yr_built and age_since_renovated. This aligns with the correlation matrix shown above.

The most correlated features are: bathrooms, sqft_living, sqft_above, grade, sqft_lot, yr_built, age_since_renovated, year and month. The features most positively correlated to each other are sqft_living and sqft_above, and the most negatively correlated features to each other are yr_built and age_since_renovated.

Basically having correlated features makes it harder to identify which features are actually responsible for changes in the target. Therefore to improve model performance we can prune some of the highly correlated features and retrain.

Kaggle competition (10 pts)

In this section, you will try to build your best model on the given training data and apply it to the provided test data and submit the predictions for the class-wide competition on Kaggle.

Model restriction. You must use linear regression (without regularization) as your predictive model. No advanced models, such as Ridge, Lasso, tree-based models, neural networks, or other complex learners are not allowed.

Implementation note. For this part, you are allowed to use a standard library implementation (e.g., 'sklearn.linear_model.LinearRegression') to speed up experimentation.

Exploration encouraged. You are encouraged to explore:

- feature engineering such as removing, transforming features, constructing new features based on existing ones, using different encoding for the discrete features;
- training data filtering/modification such as identifying and removing potential outliers in the training data;
- target manipulation such as normalizing, or log transforming the prediction target

Fair play and have fun! The spirit of this competition is for you to learn how far linear regression can go when paired with thoughtful data preparation.

To participate in this competition, use the following link: https://www.kaggle.com/t/7e07d14f327c4ee1babd526d4ccf0701

Team work. You should continue working in the same team for this competition. Make sure to note in your submission your kaggle team name.

How to sumbit. Your submission should include the prediction for every test sample. The file must be a CSV with two columns: `id` and `price`.

- `id` is the unique identifier for each instance as provided in the test data PA1_test1.csv
- `price` is your predicted result. Your file should start with a header row (`id, price`) and followed by N rows, one per test sample.

Competition evluation. The competition has two leaderboards: the public leader board as well as the private leader board. The results on the public leader board are visible through out the competition so that you can tell how well your model works compared to others and use it to pick the best models to make submission for the private leader board. Each team will be allowed to submit 3 entries to be evaluated on the private leaderboard for the final performance. The results on the private leaderboard will be released after the competition is closed.

Points and bonus points. You will get the full 10 points if you

- participate in the competition (successful submissions)
- achieve non-trivial performance (outperform some simple baseline)
- complete the report on the competition below.

You will get **3 nonus points** if your team scored top 3 on the private leader board, or entered the largest number of unique submissions (unique sores).

No late submission. The competition will be closed at 11:59 pm of the due date. No late submission will be allowed for this portion of the assignment to ensure fairness.

In [15]:

```python
#1st Config, only linear regression

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

train_val_path = './PA1_train1.csv'
test_path = './PA1_test1.csv'

train_val_df = pd.read_csv(train_val_path)
test_df = pd.read_csv(test_path)

print("Original dataset shape:", train_val_df.shape)
print(train_val_df.head())

# Split the data into train and validation sets
# Using 80% for training and 20% for validation
train_df_kaggle, val_df_kaggle = train_test_split(
    train_val_df,
    test_size=0.2,
    random_state=42
)

print(f"\nAfter split:")
print(f"Training set shape: {train_df_kaggle.shape}")
print(f"Validation set shape: {val_df_kaggle.shape}")

# Apply preprocessing to the split data
X_train_kaggle, X_val_kaggle, y_train_kaggle, y_val_kaggle = preprocess(
    train_df_kaggle,
    val_df_kaggle,
    normalize=True
```

```python
)

# Train a baseline model
model = LinearRegression()
model.fit(X_train_kaggle, y_train_kaggle)

# Evaluate on validation set
y_val_pred = model.predict(X_val_kaggle)
val_mse = np.mean((y_val_pred - y_val_kaggle)**2)
val_rmse = np.sqrt(val_mse)

print(f"\nBaseline Model Performance:")
print(f"Validation MSE: {val_mse:.2f}")
print(f"Validation RMSE: {val_rmse:.2f}")


# === INFERENCE ON TEST DATA ===
print(f"\nTest data shape: {test_df.shape}")
print("Test data columns:", test_df.columns.tolist())

# Save test IDs before preprocessing
test_ids = test_df['id'].copy()

def preprocess_test_data(test_df, train_stats_df, normalize=True):
    """
    Preprocess test data using statistics from training data
    """
    _test_df = test_df.drop(columns="id")

    # Process date
    _test_df["date"] = pd.to_datetime(_test_df["date"])
    _test_df["day"] = _test_df["date"].dt.day
    _test_df["year"] = _test_df["date"].dt.year
    _test_df["month"] = _test_df["date"].dt.month

    _test_df["age_since_renovated"] = np.where(
        _test_df["yr_renovated"] != 0,
        _test_df["year"] - _test_df["yr_renovated"],
        _test_df["year"] - _test_df["yr_built"]
    )

    _test_df = _test_df.drop(columns='date')

    # Normalize using training data statistics
    if normalize:
        for col_name in _test_df.columns:
            if col_name in train_stats_df.columns:
                mu = train_stats_df[col_name].mean()
                sigma = train_stats_df[col_name].std()
                _test_df[col_name] = (_test_df[col_name] - mu) / sigma

    # Add bias column
    bias = pd.Series(1.0, index=_test_df.index, name='bias')
    _test_df = pd.concat([bias, _test_df], axis=1)

    return _test_df

# Get training statistics for normalization
train_features = train_df_kaggle.drop(columns=['id', 'price'])
train_features["date"] = pd.to_datetime(train_features["date"])
train_features["day"] = train_features["date"].dt.day
train_features["year"] = train_features["date"].dt.year
train_features["month"] = train_features["date"].dt.month
train_features["age_since_renovated"] = np.where(
    train_features["yr_renovated"] != 0,
    train_features["year"] - train_features["yr_renovated"],
    train_features["year"] - train_features["yr_built"]
)
train_features = train_features.drop(columns='date')

# Add bias column
bias = pd.Series(1.0, index=train_features.index, name='bias')
train_features = pd.concat([bias, train_features], axis=1)

# Preprocess test data
X_test = preprocess_test_data(test_df, train_features, normalize=True)

print(f"Processed test features shape: {X_test.shape}")
print("Test feature columns:", X_test.columns.tolist())
print("Training feature columns:", X_train_kaggle.columns.tolist())

# Ensure feature order matches
X_test = X_test[X_train_kaggle.columns]

# Make predictions on test data
test_predictions = model.predict(X_test)

print(f"\nTest predictions shape: {test_predictions.shape}")
print(f"Test predictions range: [{test_predictions.min():.2f}, {test_predictions.max():.2f}]")
```

```python
# Create submission DataFrame
submission_df = pd.DataFrame({
    'id': test_ids,
    'price': test_predictions
})

print(f"\nSubmission DataFrame shape: {submission_df.shape}")
print("Submission preview:")
print(submission_df.head(10))

# Save to CSV
submission_filename = 'kaggle_submission_c1.csv'
submission_df.to_csv(submission_filename, index=False)
print(f"\nSubmission saved to: {submission_filename}")

# Verify the submission format
print(f"\nVerifying submission format:")
verification_df = pd.read_csv(submission_filename)
print(f"Columns: {verification_df.columns.tolist()}")
print(f"Shape: {verification_df.shape}")
print(f"Any missing values: {verification_df.isnull().sum().sum()}")
print("First few rows:")
print(verification_df.head())
```

```
Original dataset shape: (10000, 21)
           id       date  bedrooms  bathrooms  sqft_living  sqft_lot  floors  \
0  3066410850   7/9/2014         4       2.50         2720     10006     2.0
1  9345400350  7/18/2014         2       2.50         2600      5000     1.0
2  7128300060   7/7/2014         5       1.75         1650      3000     1.5
3  2155500030  4/28/2015         4       1.75         1720      9600     1.0
4  3999300080   9/4/2014         6       2.25         3830     11180     1.0

   waterfront  view  condition  grade  sqft_above  sqft_basement  yr_built  \
0           0     0          3      9        2720              0      1989
1           0     0          5      8        1300           1300      1926
2           0     0          3      8        1650              0      1902
3           0     0          4      8        1720              0      1969
4           0     2          5      9        2440           1390      1962

   yr_renovated  zipcode      lat     long  sqft_living15  sqft_lot15   price
0             0    98074  47.6295 -122.042           2720       10759  5.9495
1             0    98126  47.5806 -122.379           2260        5000  6.6500
2             0    98144  47.5955 -122.306           1740        4000  4.4300
3             0    98059  47.4764 -122.155           1660       10720  3.8000
4             0    98008  47.5849 -122.113           2500       10400  8.8700

After split:
Training set shape: (8000, 21)
Validation set shape: (2000, 21)

Baseline Model Performance:
Validation MSE: 4.04
Validation RMSE: 2.01

Test data shape: (5583, 20)
Test data columns: ['id', 'date', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors', 'waterfront', '
view', 'condition', 'grade', 'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat', 'long
', 'sqft_living15', 'sqft_lot15']
Processed test features shape: (5583, 23)
Test feature columns: ['bias', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors', 'waterfront', 'vie
w', 'condition', 'grade', 'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat', 'long',
'sqft_living15', 'sqft_lot15', 'day', 'year', 'month', 'age_since_renovated']
Training feature columns: ['bias', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors', 'waterfront',
'view', 'condition', 'grade', 'sqft_above', 'sqft_basement', 'yr_built', 'zipcode', 'lat', 'long', 'sqft_living
15', 'sqft_lot15', 'day', 'year', 'month', 'age_since_renovated']

Test predictions shape: (5583,)
Test predictions range: [-2.39, 30.99]

Submission DataFrame shape: (5583, 2)
Submission preview:
           id     price
0  6414100192  6.974274
1  1954400510  4.623626
2  1736800520  8.476333
3  9212900260  4.175039
4  1875500060  4.223683
5  8562750320  5.345829
6  9547205180  7.704252
7  2078500320  5.634425
8  5547700270  6.529079
9  8035350320  7.800657

Submission saved to: kaggle_submission_c1.csv

Verifying submission format:
Columns: ['id', 'price']
Shape: (5583, 2)
Any missing values: 0
First few rows:
           id     price
0  6414100192  6.974274
1  1954400510  4.623626
2  1736800520  8.476333
3  9212900260  4.175039
4  1875500060  4.223683
```

In [16]:

```python
#2nd Remove Highly Correlated and low weighted features

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

COLUMNS_TO_DROP = ['date', 'yr_renovated', 'month', 'day', 'sqft_lot', 'sqft_lot15', 'floors']

def preprocess_trim_correlated(train_df, val_df, normalize=True):
    _train_df = train_df.drop(columns="id")
    _val_df = val_df.drop(columns="id")

    #Process date
    _train_df["date"] = pd.to_datetime(_train_df["date"])
    _train_df["day"] = _train_df["date"].dt.day
    _train_df["year"] = _train_df["date"].dt.year
    _train_df["month"] = _train_df["date"].dt.month
```

```
    _train_df["age_since_renovated"] = np.where(_train_df["yr_renovated"] != 0, _train_df["year"]-_train_df["yr_


    _train_df = _train_df.drop(columns=COLUMNS_TO_DROP)

    _val_df["date"] = pd.to_datetime(_val_df["date"])
    _val_df["day"] = _val_df["date"].dt.day
    _val_df["year"] = _val_df["date"].dt.year
    _val_df["month"] = _val_df["date"].dt.month

    _val_df["age_since_renovated"] = np.where(_val_df["yr_renovated"] != 0, _val_df["year"]-_val_df["yr_renovat

    _val_df = _val_df.drop(columns=COLUMNS_TO_DROP)

    #Normalize all columns except price
    if(normalize):
        for col_name in _train_df.drop(columns=['price']).columns:
            mu = _train_df[col_name].mean()
            sigma =_train_df[col_name].std()

            _train_df[col_name] = (_train_df[col_name] - mu) / sigma
            _val_df[col_name] = (_val_df[col_name] - mu) / sigma

    # Add bias column
    bias = pd.Series(1.0, index=_train_df.index, name='bias')
    _train_df = pd.concat([bias, _train_df], axis=1)

    bias = pd.Series(1.0, index=_val_df.index, name='bias')
    _val_df = pd.concat([bias, _val_df], axis=1)

    return _train_df.drop(columns=['price']), _val_df.drop(columns=['price']), _train_df["price"], _val_df["pri

train_val_path = './PA1_train1.csv'
test_path = './PA1_test1.csv'

train_val_df = pd.read_csv(train_val_path)
test_df = pd.read_csv(test_path)

print("Original dataset shape:", train_val_df.shape)
print(train_val_df.head())

# Split the data into train and validation sets['date', 'yr_renovated', 'year', 'month', 'day', 'sqft_above']
# Using 80% for training and 20% for validation
train_df_kaggle, val_df_kaggle = train_test_split(
    train_val_df,
    test_size=0.2,
    random_state=42
)

print(f"\nAfter split:")
print(f"Training set shape: {train_df_kaggle.shape}")
print(f"Validation set shape: {val_df_kaggle.shape}")

# Apply preprocessing to the split data
X_train_kaggle, X_val_kaggle, y_train_kaggle, y_val_kaggle = preprocess_trim_correlated(
    train_df_kaggle,
    val_df_kaggle,
    normalize=True
)

# Train a baseline model['date', 'yr_renovated', 'year', 'month', 'day', 'sqft_above']
model = LinearRegression()
model.fit(X_train_kaggle, y_train_kaggle)

# Evaluate on validation set
y_val_pred = model.predict(X_val_kaggle)
val_mse = np.mean((y_val_pred - y_val_kaggle)**2)

print(f"\nBaseline Model Performance:")
print(f"Validation MSE: {val_mse:.2f}")


# === INFERENCE ON TEST DATA ===
print(f"\nTest data shape: {test_df.shape}")
print("Test data columns:", test_df.columns.tolist())

# Save test IDs before preprocessing
test_ids = test_df['id'].copy()

def preprocess_test_data(test_df, train_stats_df, normalize=True):
    """
    Preprocess test data using statistics from training data
    """
    _test_df = test_df.drop(columns="id")

    # Process date
    _test_df["date"] = pd.to_datetime(_test_df["date"])
    _test_df["day"] = _test_df["date"].dt.day
```

```python
    _test_df["year"] = _test_df["date"].dt.year
    _test_df["month"] = _test_df["date"].dt.month

    _test_df["age_since_renovated"] = np.where(
        _test_df["yr_renovated"] != 0,
        _test_df["year"] - _test_df["yr_renovated"],
        _test_df["year"] - _test_df["yr_built"]
    )

    _test_df = _test_df.drop(columns=COLUMNS_TO_DROP)

    # Normalize using training data statistics
    if normalize:
        for col_name in _test_df.columns:
            if col_name in train_stats_df.columns:
                mu = train_stats_df[col_name].mean()
                sigma = train_stats_df[col_name].std()
                _test_df[col_name] = (_test_df[col_name] - mu) / sigma

    # Add bias column
    bias = pd.Series(1.0, index=_test_df.index, name='bias')
    _test_df = pd.concat([bias, _test_df], axis=1)

    return _test_df

# Get training statistics for normalization
train_features = train_df_kaggle.drop(columns=['id', 'price'])
train_features["date"] = pd.to_datetime(train_features["date"])
train_features["day"] = train_features["date"].dt.day
train_features["year"] = train_features["date"].dt.year
train_features["month"] = train_features["date"].dt.month
train_features["age_since_renovated"] = np.where(
    train_features["yr_renovated"] != 0,
    train_features["year"] - train_features["yr_renovated"],
    train_features["year"] - train_features["yr_built"]
)
train_features = train_features.drop(columns=COLUMNS_TO_DROP)

# Add bias column
bias = pd.Series(1.0, index=train_features.index, name='bias')
train_features = pd.concat([bias, train_features], axis=1)

# Preprocess test data
X_test = preprocess_test_data(test_df, train_features, normalize=True)

print(f"Processed test features shape: {X_test.shape}")
print("Test feature columns:", X_test.columns.tolist())
print("Training feature columns:", X_train_kaggle.columns.tolist())

# Ensure feature order matches
X_test = X_test[X_train_kaggle.columns]

# Make predictions on test data
test_predictions = model.predict(X_test)

print(f"\nTest predictions shape: {test_predictions.shape}")
print(f"Test predictions range: [{test_predictions.min():.2f}, {test_predictions.max():.2f}]")

# Create submission DataFrame
submission_df = pd.DataFrame({
    'id': test_ids,
    'price': test_predictions
})

print(f"\nSubmission DataFrame shape: {submission_df.shape}")
print("Submission preview:")
print(submission_df.head(10))

# Save to CSV
submission_filename = 'kaggle_submission_c2.csv'
submission_df.to_csv(submission_filename, index=False)
print(f"\nSubmission saved to: {submission_filename}")

# Verify the submission format
print(f"\nVerifying submission format:")
verification_df = pd.read_csv(submission_filename)
print(f"Columns: {verification_df.columns.tolist()}")
print(f"Shape: {verification_df.shape}")
print(f"Any missing values: {verification_df.isnull().sum().sum()}")
print("First few rows:")
print(verification_df.head())
```

```
Original dataset shape: (10000, 21)
           id       date  bedrooms  bathrooms  sqft_living  sqft_lot  floors  \
0  3066410850   7/9/2014         4       2.50         2720     10006     2.0
1  9345400350  7/18/2014         2       2.50         2600      5000     1.0
2  7128300060   7/7/2014         5       1.75         1650      3000     1.5
3  2155500030  4/28/2015         4       1.75         1720      9600     1.0
4  3999300080   9/4/2014         6       2.25         3830     11180     1.0

   waterfront  view  condition  grade  sqft_above  sqft_basement  yr_built  \
0           0     0          3      9        2720              0      1989
1           0     0          5      8        1300           1300      1926
2           0     0          3      8        1650              0      1902
3           0     0          4      8        1720              0      1969
4           0     2          5      9        2440           1390      1962

   yr_renovated  zipcode      lat     long  sqft_living15  sqft_lot15   price
0             0    98074  47.6295 -122.042           2720       10759  5.9495
1             0    98126  47.5806 -122.379           2260        5000  6.6500
2             0    98144  47.5955 -122.306           1740        4000  4.4300
3             0    98059  47.4764 -122.155           1660       10720  3.8000
4             0    98008  47.5849 -122.113           2500       10400  8.8700

After split:
Training set shape: (8000, 21)
Validation set shape: (2000, 21)

Baseline Model Performance:
Validation MSE: 4.05

Test data shape: (5583, 20)
Test data columns: ['id', 'date', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors', 'waterfront', '
view', 'condition', 'grade', 'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat', 'long
', 'sqft_living15', 'sqft_lot15']
Processed test features shape: (5583, 17)
Test feature columns: ['bias', 'bedrooms', 'bathrooms', 'sqft_living', 'waterfront', 'view', 'condition', 'grad
e', 'sqft_above', 'sqft_basement', 'yr_built', 'zipcode', 'lat', 'long', 'sqft_living15', 'year', 'age_since_re
novated']
Training feature columns: ['bias', 'bedrooms', 'bathrooms', 'sqft_living', 'waterfront', 'view', 'condition', '
grade', 'sqft_above', 'sqft_basement', 'yr_built', 'zipcode', 'lat', 'long', 'sqft_living15', 'year', 'age_sinc
e_renovated']

Test predictions shape: (5583,)
Test predictions range: [-2.35, 31.22]

Submission DataFrame shape: (5583, 2)
Submission preview:
           id     price
0  6414100192  6.849096
1  1954400510  4.631742
2  1736800520  8.358237
3  9212900260  4.301022
4  1875500060  4.319058
5  8562750320  5.223364
6  9547205180  7.722226
7  2078500320  5.707803
8  5547700270  6.525511
9  8035350320  7.792177

Submission saved to: kaggle_submission_c2.csv

Verifying submission format:
Columns: ['id', 'price']
Shape: (5583, 2)
Any missing values: 0
First few rows:
           id     price
0  6414100192  6.849096
1  1954400510  4.631742
2  1736800520  8.358237
3  9212900260  4.301022
4  1875500060  4.319058
```

In [17]:

```python
#3rd One hot encode zipcode

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

COLUMNS_TO_DROP = ['date', 'yr_renovated', 'zipcode']

def preprocess_trim_correlated(train_df, val_df, normalize=True):
    _train_df = train_df.drop(columns="id").copy()
    _val_df = val_df.drop(columns="id").copy()

    # Process date for both datasets
    for df in [_train_df, _val_df]:
        df["date"] = pd.to_datetime(df["date"])
        df["day"] = df["date"].dt.day
        df["year"] = df["date"].dt.year
```

```python
        df["month"] = df["date"].dt.month
        df["age_since_renovated"] = np.where(
            df["yr_renovated"] != 0,
            df["year"] - df["yr_renovated"],
            df["year"] - df["yr_built"]
        )

    # ONE-HOT ENCODE ZIPCODE FIRST (before normalization and dropping columns)
    ohe_local = OneHotEncoder(sparse_output=False, handle_unknown='ignore')

    # Fit on training data only
    ohe_local.fit(_train_df[["zipcode"]])

    # Transform both train and validation
    train_ohe = ohe_local.transform(_train_df[["zipcode"]])
    val_ohe = ohe_local.transform(_val_df[["zipcode"]])

    # Create DataFrames with proper indices
    train_ohe_df = pd.DataFrame(
        train_ohe,
        columns=ohe_local.get_feature_names_out(),
        index=_train_df.index
    )
    val_ohe_df = pd.DataFrame(
        val_ohe,
        columns=ohe_local.get_feature_names_out(),
        index=_val_df.index
    )

    # Concatenate one-hot encoded features
    _train_df = pd.concat([_train_df, train_ohe_df], axis=1)
    _val_df = pd.concat([_val_df, val_ohe_df], axis=1)

    # Drop columns AFTER one-hot encoding
    _train_df = _train_df.drop(columns=COLUMNS_TO_DROP)
    _val_df = _val_df.drop(columns=COLUMNS_TO_DROP)

    # Normalize all columns except price (skip one-hot encoded columns)
    if normalize:
        # Get columns to normalize (exclude price and one-hot encoded columns)
        cols_to_normalize = [col for col in _train_df.columns
                             if col != 'price' and not col.startswith('zipcode_')]

        for col_name in cols_to_normalize:
            mu = _train_df[col_name].mean()
            sigma = _train_df[col_name].std()

            _train_df[col_name] = (_train_df[col_name] - mu) / sigma
            _val_df[col_name] = (_val_df[col_name] - mu) / sigma

    # Add bias column
    bias_train = pd.Series(1.0, index=_train_df.index, name='bias')
    bias_val = pd.Series(1.0, index=_val_df.index, name='bias')

    _train_df = pd.concat([bias_train, _train_df], axis=1)
    _val_df = pd.concat([bias_val, _val_df], axis=1)

    return (_train_df.drop(columns=['price']),
            _val_df.drop(columns=['price']),
            _train_df["price"],
            _val_df["price"],
            ohe_local)  # Return the fitted encoder

train_val_path = './PA1_train1.csv'
test_path = './PA1_test1.csv'

train_val_df = pd.read_csv(train_val_path)
test_df = pd.read_csv(test_path)

# Split the data
train_df_kaggle, val_df_kaggle = train_test_split(
    train_val_df,
    test_size=0.2,
    random_state=42
)

# Apply preprocessing and get the fitted encoder
X_train_kaggle, X_val_kaggle, y_train_kaggle, y_val_kaggle, fitted_ohe = preprocess_trim_correlated(
    train_df_kaggle,
    val_df_kaggle,
    normalize=True
)

# Train model
model = LinearRegression()
model.fit(X_train_kaggle, y_train_kaggle)

# Evaluate on validation set
y_val_pred = model.predict(X_val_kaggle)
```

```python
val_mse = np.mean((y_val_pred - y_val_kaggle)**2)

print(f"\nModel Performance:")
print(f"Validation MSE: {val_mse:.2f}")

# Save test IDs before preprocessing
test_ids = test_df['id'].copy()

def preprocess_test_data(test_df, ohe_fitted, train_stats_df, normalize=True):
    """
    Preprocess test data using fitted encoder and training statistics
    """
    _test_df = test_df.drop(columns="id").copy()

    # Process date
    _test_df["date"] = pd.to_datetime(_test_df["date"])
    _test_df["day"] = _test_df["date"].dt.day
    _test_df["year"] = _test_df["date"].dt.year
    _test_df["month"] = _test_df["date"].dt.month

    _test_df["age_since_renovated"] = np.where(
        _test_df["yr_renovated"] != 0,
        _test_df["year"] - _test_df["yr_renovated"],
        _test_df["year"] - _test_df["yr_built"]
    )

    # One-hot encode using fitted encoder
    test_ohe = ohe_fitted.transform(_test_df[["zipcode"]])
    test_ohe_df = pd.DataFrame(
        test_ohe,
        columns=ohe_fitted.get_feature_names_out(),
        index=_test_df.index
    )

    _test_df = pd.concat([_test_df, test_ohe_df], axis=1)
    _test_df = _test_df.drop(columns=COLUMNS_TO_DROP)

    # Normalize using training statistics
    if normalize:
        cols_to_normalize = [col for col in _test_df.columns
                             if not col.startswith('zipcode_')]

        for col_name in cols_to_normalize:
            if col_name in train_stats_df.columns:
                mu = train_stats_df[col_name].mean()
                sigma = train_stats_df[col_name].std()
                _test_df[col_name] = (_test_df[col_name] - mu) / sigma

    # Add bias column
    bias = pd.Series(1.0, index=_test_df.index, name='bias')
    _test_df = pd.concat([bias, _test_df], axis=1)

    return _test_df

# Create training stats using the same preprocessing as the training data
train_stats = train_df_kaggle.drop(columns=['id', 'price']).copy()
train_stats["date"] = pd.to_datetime(train_stats["date"])
train_stats["day"] = train_stats["date"].dt.day
train_stats["year"] = train_stats["date"].dt.year
train_stats["month"] = train_stats["date"].dt.month
train_stats["age_since_renovated"] = np.where(
    train_stats["yr_renovated"] != 0,
    train_stats["year"] - train_stats["yr_renovated"],
    train_stats["year"] - train_stats["yr_built"]
)

# Preprocess test data using the fitted encoder from training
X_test = preprocess_test_data(test_df, fitted_ohe, train_stats, normalize=True)

print(f"Training features shape: {X_train_kaggle.shape}")
print(f"Test features shape: {X_test.shape}")

# Ensure feature order matches
X_test = X_test[X_train_kaggle.columns]

# Make predictions on test data
test_predictions = model.predict(X_test)

# Create submission DataFrame
submission_df = pd.DataFrame({
    'id': test_ids,
    'price': test_predictions
})

# Save to CSV
submission_filename = 'kaggle_submission_c3.csv'
submission_df.to_csv(submission_filename, index=False)
print(f"\nSubmission saved to: {submission_filename}")
```

```
Model Performance:
Validation MSE: 2.50
Training features shape: (8000, 91)
Test features shape: (5583, 91)

Submission saved to: kaggle_submission_c3.csv
```

✍ Report on the Kaggle competition

1. **Team name: Manuel Agraz Vallejo**:
2. **Exploration Summary:** Brief describe the approches you tried. 3. **Most Impactful Change:** Which exploration led to the most performance improvement, and why do you think it helped?

I tried 3 approaches:

1. A simple linear regression with only the feature modifications used in the assignment (normalization, transforming date feature and yr_renovated).

2. Along with the previous feature modification, this approach additionally removes features with very small weights and some of the highly correlated features. Features removed: 'month', 'day', 'sqft_lot', 'sqft_lot15', 'floors'

3. Keeping the first approach's feature modifications, this approach one hot encodes the zipcode feature. This makes the feature space larger, also increasing the amount of weights, but makes noticing differences in the zipcode data easier.

Out of these three approaches the one that led to the best performance was the one hot encoding of the zipcode feature. With this approach the model was able to achieve a loss of 2.35 on the test set. I think it helped the most since the new encoding really separated the different zipcodes making it easy for the model to identify which ones are more relevant than others. Without the encoding the zipcodes are all very close to each other, within the 9000s range and its hard for the model to discern between them.

In [20]:
```
#running this code block will convert this notebook and its outputs into a pdf report.
# ⚠ALERT! Exporting colab notebooks into a clean figure-inclusive pdf can be unreliable.
# Sometimes output figures may not appear in your exported file.
#
#If this happens, please assemble your report manually: copy relevant fgures/results
# into a separate documents and save as PDF. Be sure to clearly lablel each figure with
# the corresponding part number (e.g., Part 3(b)).).

# !jupyter nbconvert --to html /content/gdrive/MyDrive/Colab\ Notebooks/IA1-2024.ipynb  # you might need to cha
!jupyter nbconvert --to html '/home/magraz/ml-class/HW2/IA1_2025.ipynb'    # you might need to change this path

# input_html = '/content/gdrive/MyDrive/Colab Notebooks/IA1-2025.html' #you might need to change this path acco
# output_pdf = '/content/gdrive/MyDrive/Colab Notebooks/IA1output.pdf' #you might need to change this path or n

input_html = '/home/magraz/ml-class/HW2/IA1_2025.html' #you might need to change this path accordingly
output_pdf = '/home/magraz/ml-class/HW2/IA1output.pdf' #you might need to change this path or name accordingly

# Convert HTML to PDF
pdfkit.from_file(input_html, output_pdf)

# Download the generated PDF
# files.download(output_pdf)
```

```
usage: jupyter [-h] [--version] [--config-dir] [--data-dir] [--runtime-dir]
               [--paths] [--json] [--debug]
               [subcommand]

Jupyter: Interactive Computing

positional arguments:
  subcommand     the subcommand to launch

options:
  -h, --help     show this help message and exit
  --version      show the versions of core jupyter packages and exit
  --config-dir   show Jupyter config dir
  --data-dir     show Jupyter data dir
  --runtime-dir  show Jupyter runtime dir
  --paths        show all Jupyter paths. Add --json for machine-readable
                 format.
  --json         output paths as machine-readable json
  --debug        output debug information about paths

Available subcommands: kernel kernelspec migrate run troubleshoot

Jupyter command `jupyter-nbconvert` not found.
```

Out[20]:
```
True
```