Problem 1: Linear Regression on a Simple Dataset

Student: Yankun (Alex) Meng

In this problem, you will implement a multiple linear regression model from scratch. Please download the **Concrete Strength regression** dataset from https://www.kaggle.com/datasets/maajdl/yeh-concret-data.

A multiple linear regression model takes the form:

$$Y = X\hat{\beta} + \hat{\epsilon} \text{ where } \hat{\epsilon} \sim \mathcal{N}(0, \hat{\sigma}I)$$

Here, $X \in \mathbb{R}^{N \times m}$ where each column represents an independent variable, and $Y \in \mathbb{R}^{N \times 1}$ represents the dependent variable to be predicted. We assume that our model minimizes the MSE loss (i.e. $\hat{\beta}_{opt} = argmin_{\hat{\beta}} \|Y - X\hat{\beta}\|_2^2$).

Given the dataset, "concrete compressive strength" is the variable to be predicted.

Problem 1

1. Using all independent variables plus a bias term, find a $\hat{\beta}_0 \in \mathbb{R}^{9 \times 1}$ such that $\|Y - X\hat{\beta}_0\|_2$ is minimized. You may only use matrix operations (e.g., multiplication, decomposition, inverse) to find β_0. **You are not allowed to use any statistical libraries.**

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv("./data/Concrete_Data_Yeh.csv")
print(data.shape)

# Check if there are any null values in the dataset
print(data.isnull().sum())
```

```
(1030, 9)
cement              0
slag                0
flyash              0
water               0
superplasticizer    0
coarseaggregate     0
fineaggregate       0
age                 0
csMPa               0
dtype: int64
```

So The concrete strength dataset has 1030 datapoints, 8 features, and 1 target variable. Which means that the design matrix X has shape (1030, 8), Output Vector Y is (1030, 1), so the bias has shape (1030, 1), and the weight matrix $\hat{\beta}$ has dimensions (8, 1), one for each feature.

If we include the bias term inside the weight matrix however, so the model is more like

$$Y = X\hat{\beta}$$

Now since the first column of $\hat{\beta}$ represents the bias, and all the rest columns represents the coefficients/weights for the features. X has shape (1030, 9) and $\hat{\beta}$ has shape (9, 1).

Here we use the normal equations to solve the problem. We derived it in class but let's derive it again here.

So the MSE error function can be written as

$$E = \frac{1}{2}\sum_{i=1}^{N}(x_i\hat{\beta} - y_i)^2$$

Notice this is just the L2-Norm, and if we write this in dot product form:

$$E = \frac{1}{2}(X\hat{\beta} - Y)^T(X\hat{\beta} - Y)$$

To minimize this we have to take $\frac{\partial E}{\partial \hat{\beta}} = 0$, using the product rule, we have

$$\frac{\partial E}{\partial \hat{\beta}} = X^T(X\hat{\beta} - Y) = 0$$

$$X^TX\hat{\beta} - X^TY = 0$$

$$X^TX\hat{\beta} = X^TY$$

$$\boxed{\hat{\beta} = (X^TX)^{-1}X^TY}$$

So this is the beta that minimizes the MSE function, which is what I'll use for this problem as the model.

In [2]:
```
feature_cols = ['cement', 'slag', 'flyash', 'water',
                'superplasticizer', 'coarseaggregate', 'fineaggregate', 'age']
X = data[feature_cols].to_numpy()
X = np.hstack([
    np.ones((1030, 1)),
    X
])
Y = data['csMPa'].to_numpy()
Y = np.reshape(Y, (1030, 1))
```

In [3]:
```
print(X.shape)
print(Y.shape)
```

```
(1030, 9)
(1030, 1)
```

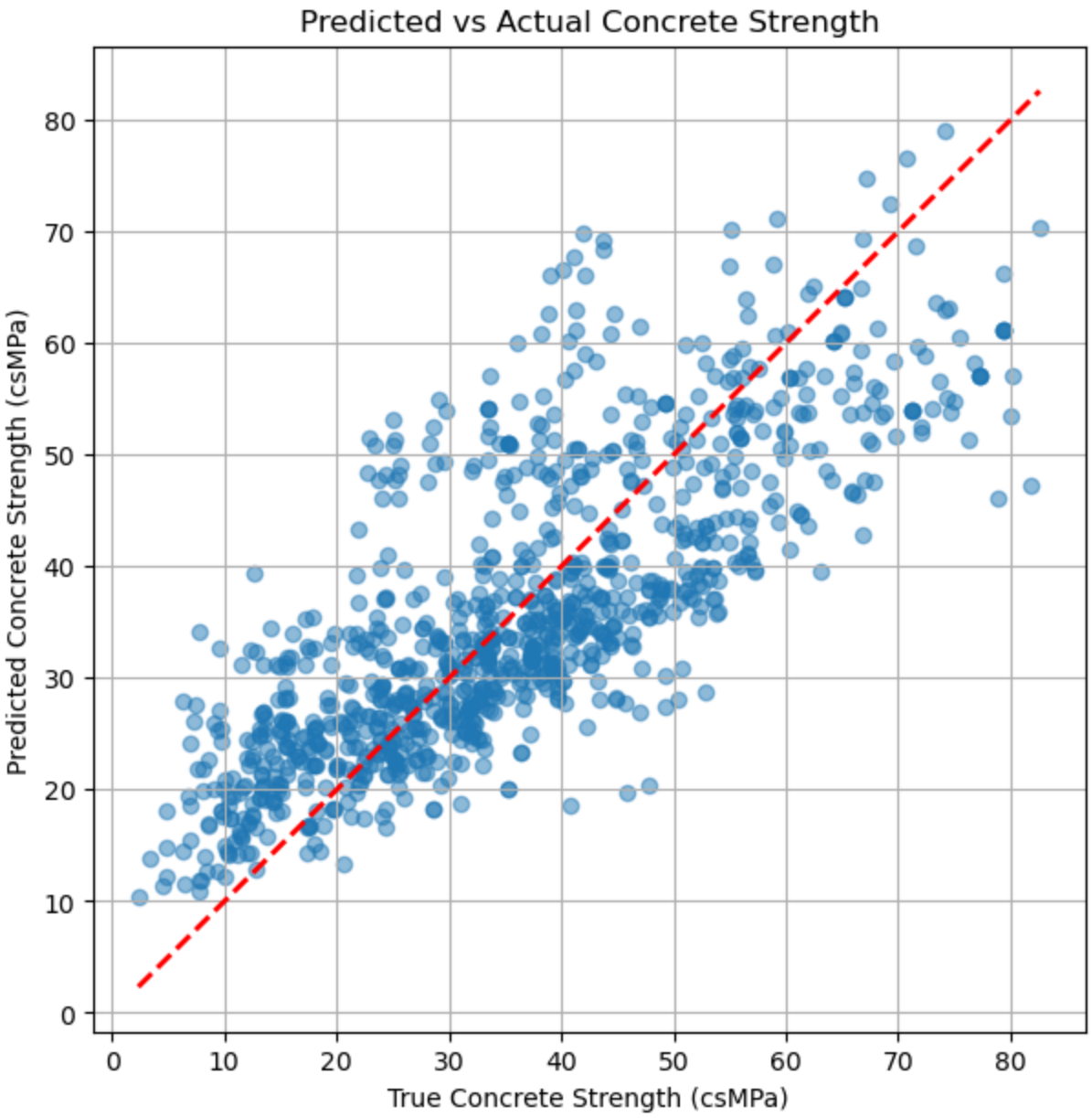
In [4]:
```
# Function to calculate optimal beta
def get_optimal_beta(X, Y):
    XTX = X.T @ X
    XTX_inv = np.linalg.inv(XTX)
    XTX_invX = XTX_inv @ X.T
    beta0 = XTX_invX @ Y
    return beta0

# Calculate optimal beta
beta0 = get_optimal_beta(X, Y)
```

In [5]:
```
print(beta0)
print(beta0.shape)
```

```
[[-2.33312136e+01]
 [ 1.19804334e-01]
 [ 1.03865809e-01]
 [ 8.79343215e-02]
 [-1.49918419e-01]
 [ 2.92224595e-01]
 [ 1.80862148e-02]
 [ 2.01903511e-02]
 [ 1.14222068e-01]]
(9, 1)
```

In [6]:
```
# Make predictions
Y_pred = X @ beta0
```

In [23]:
```
def plot_correlation(Y_true, Y_pred, figsize=(7,7)):
    # Y_pred and Y are both (N, 1) arrays
    plt.figure(figsize=figsize)
    plt.scatter(Y_true, Y_pred, alpha=0.5)
    plt.plot([Y.min(), Y.max()], [Y.min(), Y.max()], 'r--', lw=2)  # y = x line
    plt.xlabel("True Concrete Strength (csMPa)")
    plt.ylabel("Predicted Concrete Strength (csMPa)")
    plt.title("Predicted vs Actual Concrete Strength")
    plt.grid(True)
    plt.show()
plot_correlation(Y, Y_pred)
```



In [8]:
```
# Mean Squared Error
def mse(Y_true, Y_pred):
    return np.square(np.subtract(Y_true,Y_pred)).mean()
print(mse(Y, Y_pred))
```

```
107.19723607486017
```

Problem 2

2. Randomly split the dataset into training set (80%) and a validation set (20%). Use the training set to build models with $\beta \in \mathbb{R}^{i\times 1}$ for each $i \in \{7, 8, 9\}$. Then, compute the MSE loss of your prediction on the validation set for each model.

What do you observe from the results of your three models? Do models with more independent variables always perform better? You may use any combination of independent variables of your choive.

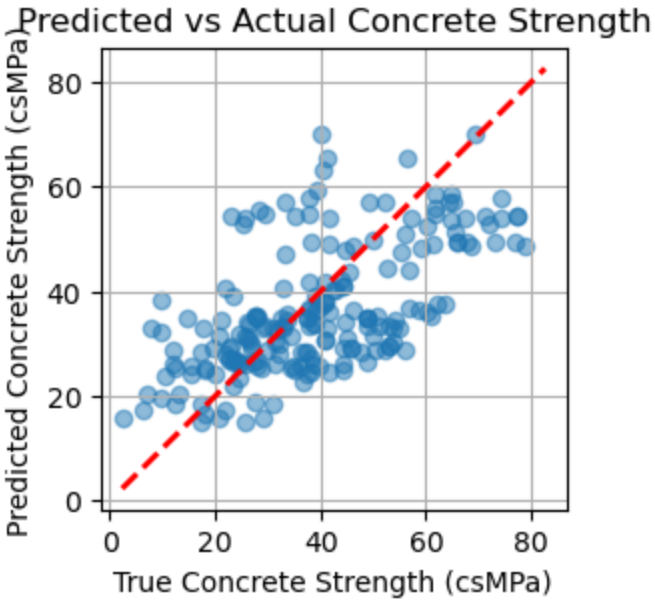
Note: The code in this section is a continuation of code in part 1, meaning that the code in part 1 should be run before running this code

In [9]:
```
# Get the indices array
N = X.shape[0]
indices = np.arange(N)
np.random.seed(11)
np.random.shuffle(indices)

# Split indices for 80-20 train-validation split
train_size = int(0.8 * N)
train_idx = indices[:train_size]
val_idx = indices[train_size:]

# Create training and validation sets using the indices
X_train, Y_train = X[train_idx], Y[train_idx]
X_val, Y_val = X[val_idx], Y[val_idx]

print(X_train.shape, Y_train.shape)
print(X_val.shape, Y_val.shape)
```

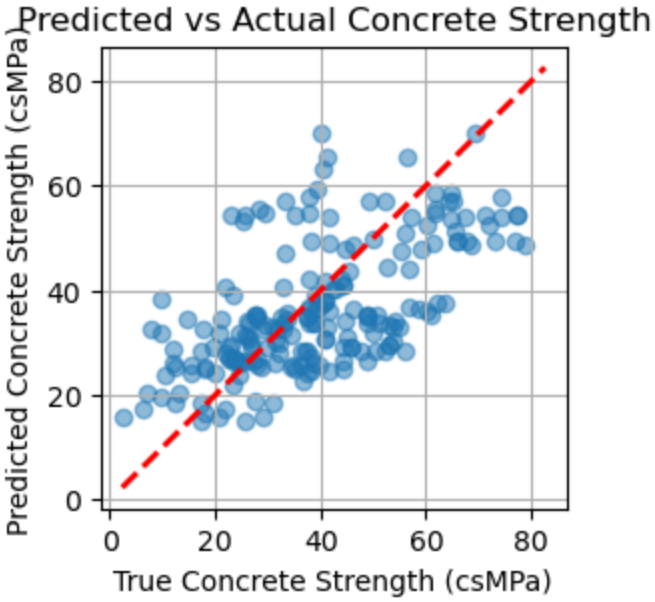
```
(824, 9) (824, 1)
(206, 9) (206, 1)
```

In [24]:
```
number_of_columns = [7, 8, 9]

for c in number_of_columns:
    print(f"Running Number of columns = {c}")
    # Train
    X_train_new = X_train[:, :c]
    X_val_new = X_val[:, :c]
    print(f"Training Set X Shape: {X_train_new.shape}")
    beta_new = get_optimal_beta(X_train_new, Y_train)
    print(f"beta shape: {beta_new.shape}")

    # Predict
    Y_pred_new = X_val_new @ beta_new

    # Evaluate
    mse_val = mse(Y_val, Y_pred_new)
    print(mse_val)

    # Y_pred and Y are both (N, 1) arrays
    plot_correlation(Y_val, Y_pred_new, figsize=(3, 3))
```

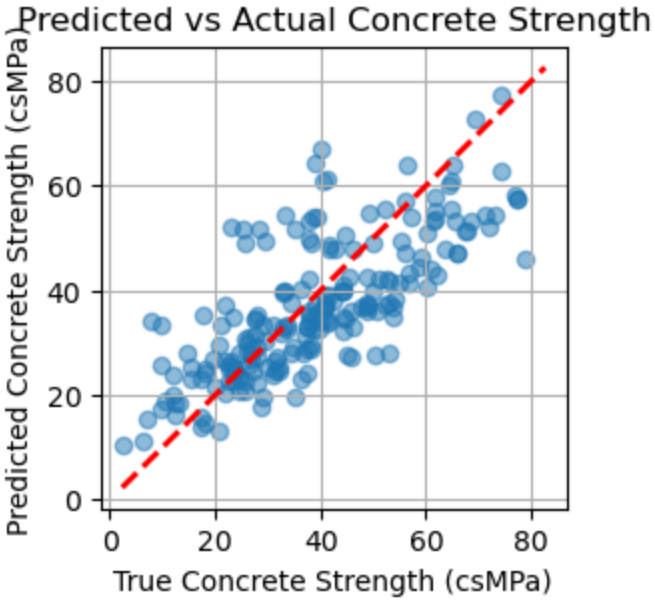
```
Running Number of columns = 7
Training Set X Shape: (824, 7)
beta shape: (7, 1)
185.65027353877565
```




```
Running Number of columns = 8
Training Set X Shape: (824, 8)
beta shape: (8, 1)
185.6864922433817
```




```
Running Number of columns = 9
Training Set X Shape: (824, 9)
beta shape: (9, 1)
127.16494430935373
```



From the results, we observe that the first two models with 7 and 8 features have very similar validation MSEs (~185.65), while the model with 9 features achieves a noticeably lower MSE (~127.16). This shows that adding more independent variables can improve model performance if the additional feature provides useful information. However, the small difference between 7 and 8 features (with 8 even being a teeny-tiny bit higher than 7) also illustrates that more variables do not always lead to better results—irrelevant or redundant features may have little effect or even harm generalization.