# 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 $\beta_0$. **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.

```python
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))
```

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

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

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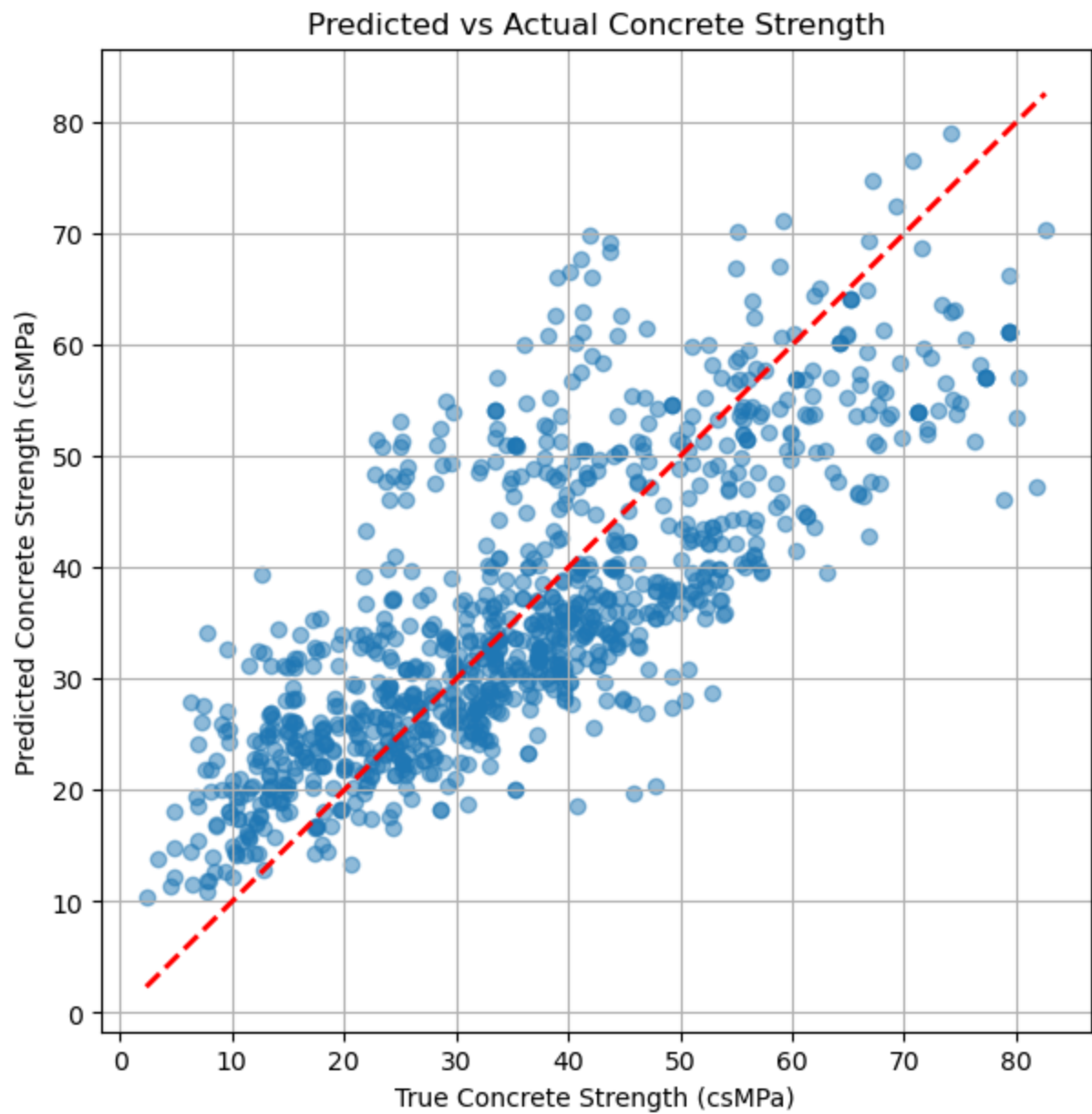
```python
In [5]: print(beta0)
        print(beta0.shape)
```

```
[[-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)
```

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

```python
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)
```

## Predicted vs Actual Concrete Strength



```
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

```python
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)
```

```python
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))
```
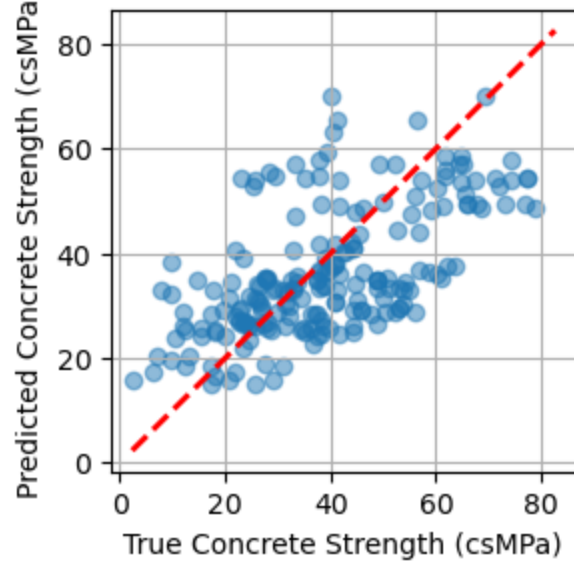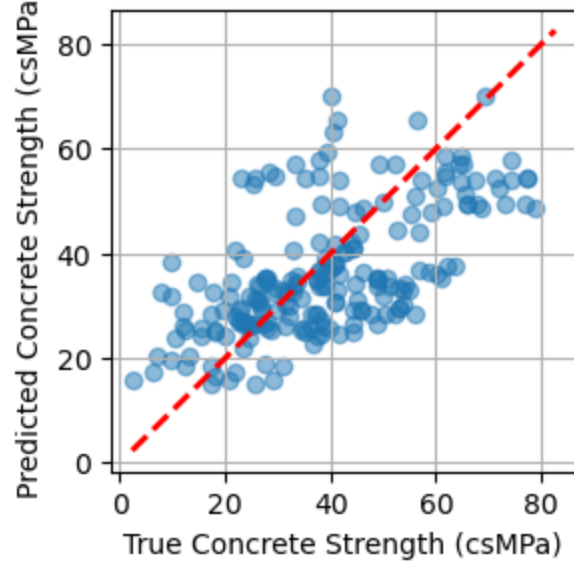
```
Running Number of columns = 7
Training Set X Shape: (824, 7)
beta shape: (7, 1)
185.65027353877565
```
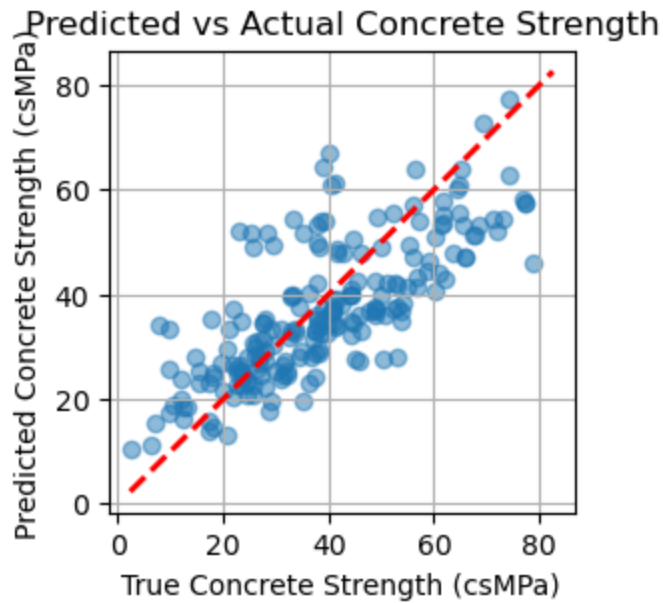
Predicted vs Actual Concrete Strength



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

Predicted vs Actual Concrete Strength



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

**Predicted vs Actual Concrete Strength**

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.

# Problem 3: Transformation from a Uniform Distribution

Student: Yankun (Alex) Meng

The Box-Muller transform is a popular sampling method for generating pairs of independent standard, normally distributed random variables from a pair of independent, standard uniformly distributed random numbers.

Specifically, Let $U_1 \sim Uniform(0, 1)$ and $U_2 \sim Uniform(0, 1)$ and define the random variables:

$$\Theta = 2\pi U_1$$

$$R = \sqrt{-2\ln(U_2)}$$

Show that the random variable vector

$$Z = \begin{pmatrix} R\cos(\Theta) \\ R\sin(\Theta) \end{pmatrix}$$

follows a standard bivariate normal distribution (i.e. $Z \sim \mathcal{N}(0, I_2)$).

Then, create a plot to verify that the transformed distribution is indeed Gaussian. You should start with `np.random.rand()` to sample from a uniform distribution.

## Solution

Suppose Z is the random vector

$$Z = \begin{pmatrix} X \\ Y \end{pmatrix}$$

where $X = R\cos(\Theta)$ and $Y = R\sin(\Theta)$, which is literally just a point in 2D Cartesian space, that is represented in polar coordinates.

The goal is to show that the Joint Probability Density Function (PDF) $f_{X,Y}(x, y)$ produced by a Box Muller Transform is the exact same as the PDF of the standard bivariate normal distribution. Let's derive both.

The Covariance matrix is given as the identity matrix

$$I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

which means that $X$ and $Y$ and independent. So the joint probability density function is defined as

$$f_{X,Y}(x,y) = \frac{1}{2\pi} e^{-\frac{(x^2+y^2)}{2}} \tag{1}$$

So now we just have to show that the PDF resulting from the Box-Muller Transform is exactly that. So what is $f_{\Theta,R}(\theta, r)$, the polar version of $f_{X,Y}(x,y)$?

Since $U_1$ and $U_2$ are independent by assumption, $R$ and $\Theta$ must be independent because $\Theta = f(U_1)$ and $R = g(U_2)$, they are just some functions of the two variables. So

$$f_{\Theta,R}(\theta, r) = f_{\Theta}(\theta) f_R(r)$$

by the definition of independence for joint probability distributions.

We know that $U_1 \sim Uniform(0,1)$, so that $\Theta = 2\pi U_1$ must be $\Theta \sim Uniform(0, 2\pi)$, so its pdf is

$$f_{\Theta}(\theta) = \begin{cases} \frac{1}{2\pi} & 0 \le \theta < 2\pi \\ 0 & \text{otherwise} \end{cases}$$

For $R = \sqrt{-2\ln U_2}$, we can compute the CDF for $R$ and from there derive the pdf.

$$\begin{aligned} F_R(r) &= P(R \le r) \\ &= P(\sqrt{-2\ln U_2} \le r) \\ &= P(-2\ln U_2 \le r^2) \\ &= P(2\ln U_2 \ge -r^2) \\ &= P(\ln U_2 \ge \frac{-r^2}{2}) \\ &= P(U_2 \ge e^{\frac{-r^2}{2}}) \\ &= 1 - P(U_2 \le e^{\frac{-r^2}{2}}) \end{aligned}$$

Since we know that $U_2 \sim Uniform(0,1)$, its pdf is

$$f_{U_2}(u_2) = \begin{cases} 1 & 0 \le u_2 < 1 \\ 0 & \text{otherwise} \end{cases}$$

which means the cumulative distribution function (cdf) is

$$F_{U_2}(u_2) = \begin{cases} u_2 & 0 \le u_2 < 1 \\ 0 & \text{otherwise} \end{cases}$$

Therefore,

$$\begin{aligned} F_R(r) &= 1 - P(U_2 \le e^{\frac{-r^2}{2}}) \\ &= 1 - e^{\frac{-r^2}{2}} \end{aligned}$$

Differentiating the cdf to get the pdf,

$$\frac{d}{dr} F_R(r) = f_R(r)$$

$$= re^{\frac{-r^2}{2}}, r \geq 0$$

So in summary,

$$f_R(r) = re^{\frac{-r^2}{2}}, r \geq 0$$

$$f_\Theta(\theta) = \begin{cases} \frac{1}{2\pi} & 0 \leq \theta < 2\pi \\ 0 & \text{otherwise} \end{cases}$$

$$\implies f_{\Theta,R}(\theta, r) = f_\Theta(\theta) f_R(r)$$

$$= \frac{1}{2\pi} re^{\frac{-r^2}{2}} \qquad \text{where } 0 \leq \theta < 2\pi, r \geq 0$$

In order to convert $f_{\Theta,R}(\theta, r) \to f_{X,Y}(x, y)$, we perform the change of variables

$$X = R\cos\Theta, \quad Y = R\sin\Theta$$

The Jacobian of this transformation is

$$J = \begin{vmatrix} \frac{\partial X}{\partial R} & \frac{\partial X}{\partial \Theta} \\ \frac{\partial Y}{\partial R} & \frac{\partial Y}{\partial \Theta} \end{vmatrix} = \begin{vmatrix} \cos\Theta & -R\sin\Theta \\ \sin\Theta & R\cos\Theta \end{vmatrix} = R(\cos^2\Theta + \sin^2\Theta) = R$$

In order to perform $\{\theta, r\} \to \{x, y\}$, we need to multiply the new distribution with the jacobian of the transformation, which becomes

$$f_{\Theta,R}(\theta, r) = f_{X,Y}(x, y) \cdot r$$

$$\frac{1}{2\pi} re^{\frac{-r^2}{2}} = f_{X,Y}(x, y) \cdot r$$

$$\frac{1}{2\pi} e^{\frac{-r^2}{2}} = f_{X,Y}(x, y)$$

Since $x = rcos(\theta), y = rsin(\theta), r = \sqrt{x^2 + y^2}$ according to rectangular to polar transformation, so

$$\frac{1}{2\pi} e^{\frac{-(x^2+y^2)}{2}} = f_{X,Y}(x, y)$$

which was exactly the pdf of the standard bivariate normal distribution (equation (1)) defined in the beginning.
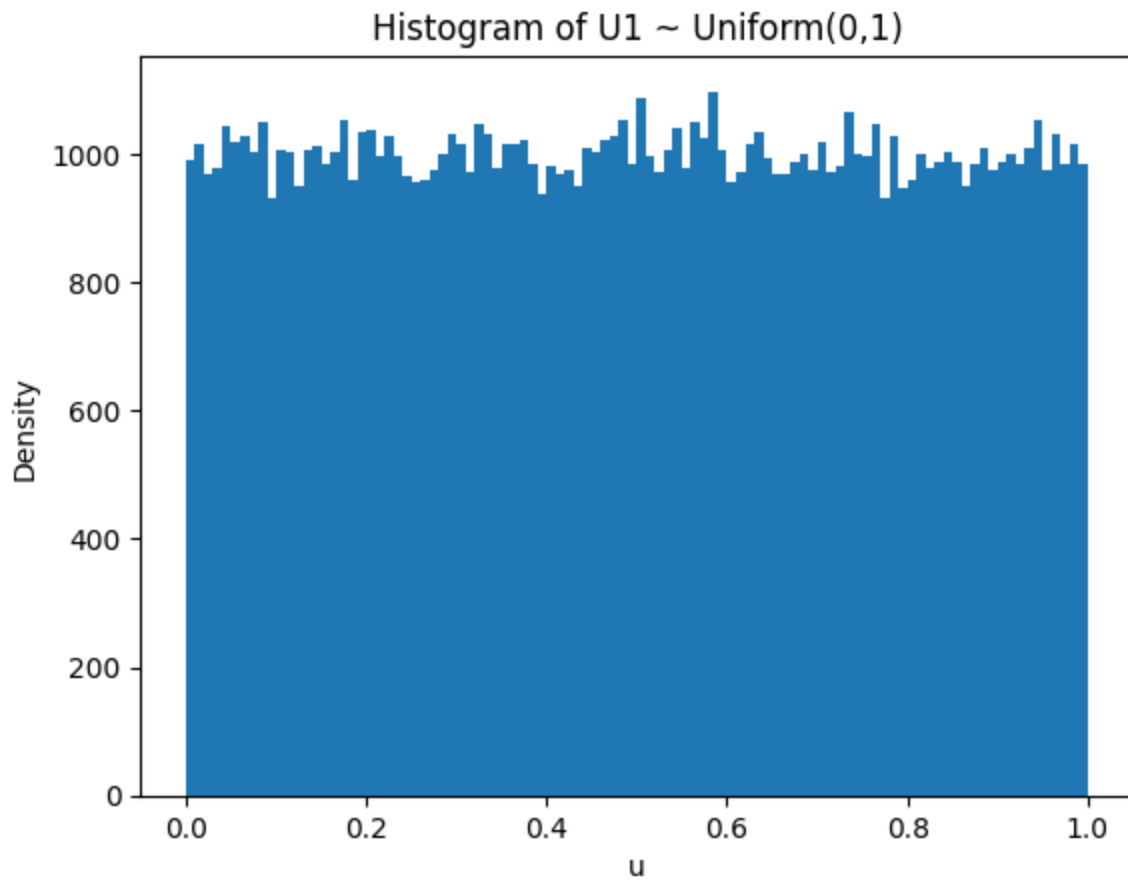
## Plotting

In [44]:
```python
import numpy as np
import matplotlib.pyplot as plt
```
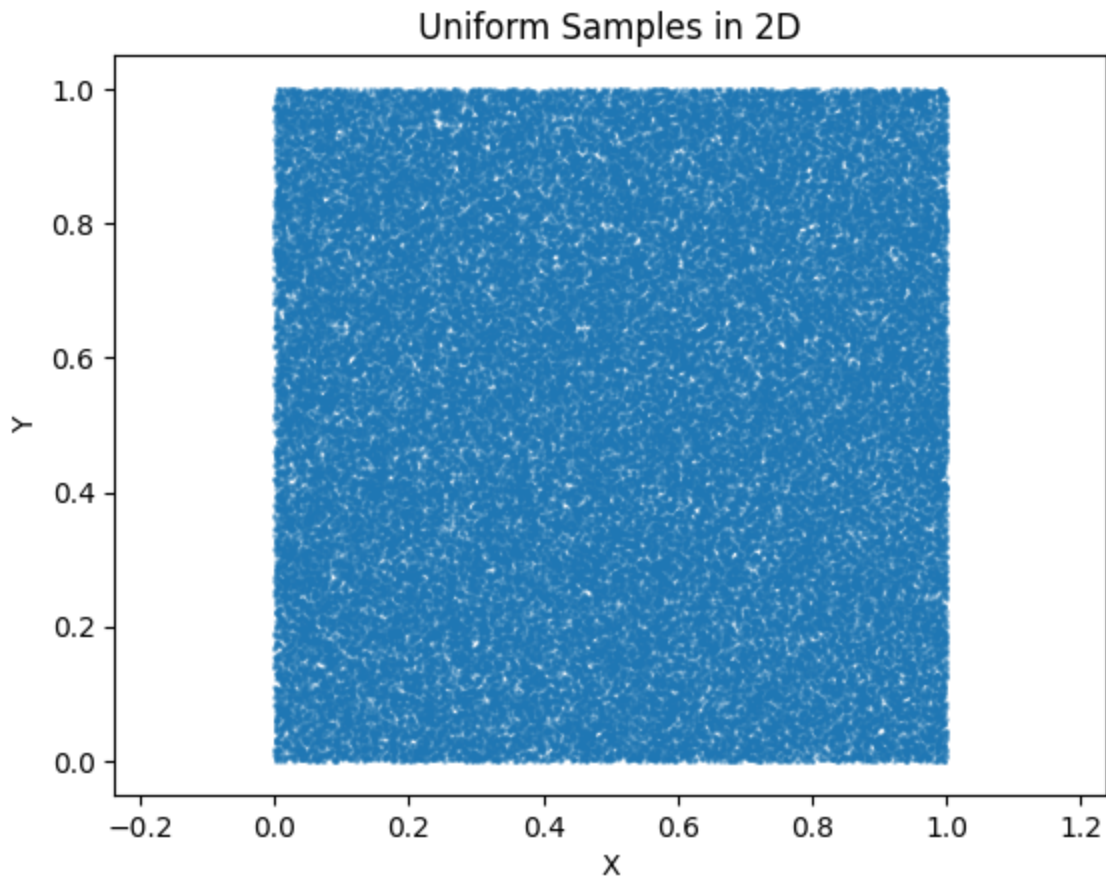
```
# Number of Samples
N = 100000

U1 = np.random.rand(N)
U2 = np.random.rand(N)
```

In [45]:
```
plt.figure()
plt.hist(U1, bins=100)
plt.xlabel("u")
plt.ylabel("Density")
plt.title("Histogram of U1 ~ Uniform(0,1)")
plt.show()
```
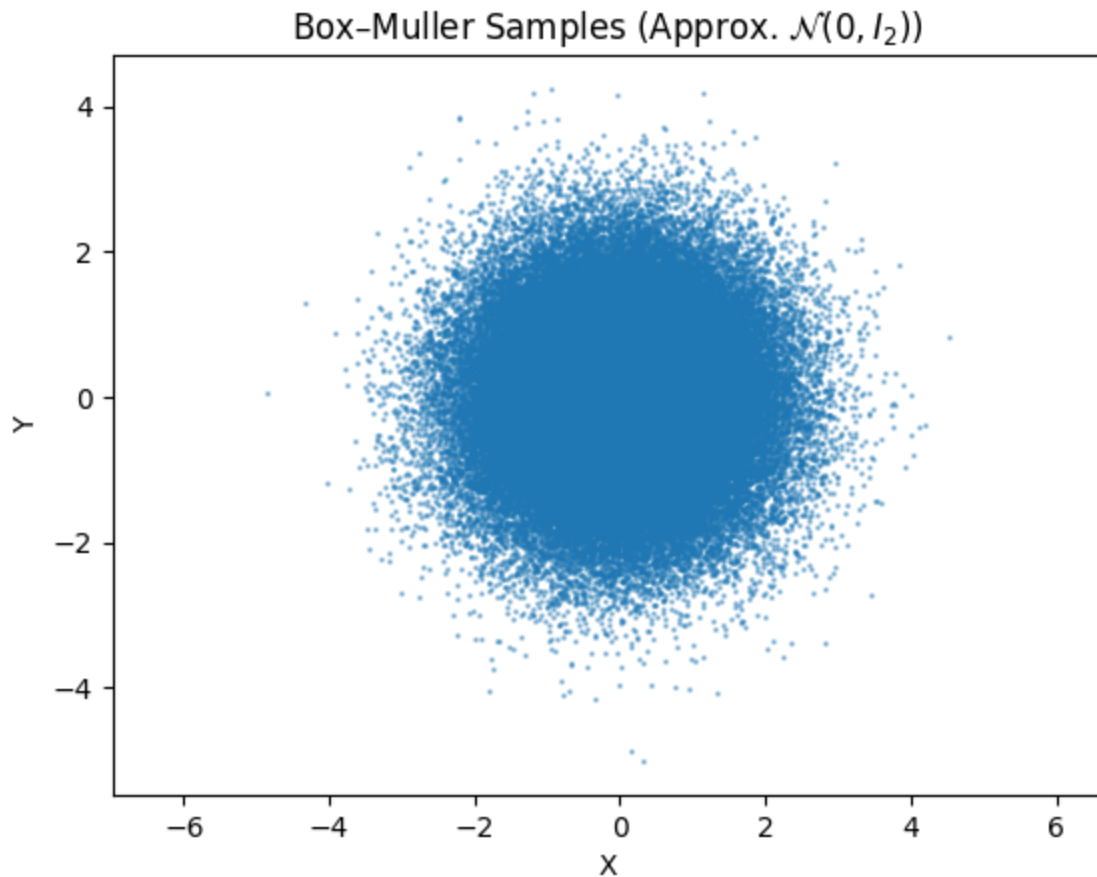


In [54]:
```
# 2D scatter plot (joint distribution)
plt.figure()
plt.scatter(U1, U2, s=1, alpha=0.4)
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Uniform Samples in 2D")
plt.axis("equal")
plt.show()
```

Uniform Samples in 2D

Indeed, these 100,000 draws looks pretty uniformly distributed to me.

In [47]:
```python
# Box-Muller Transform
Theta = 2 * np.pi * U1
R = np.sqrt(-2 * np.log(U2))
X = R * np.cos(Theta)
Y = R * np.sin(Theta)
```
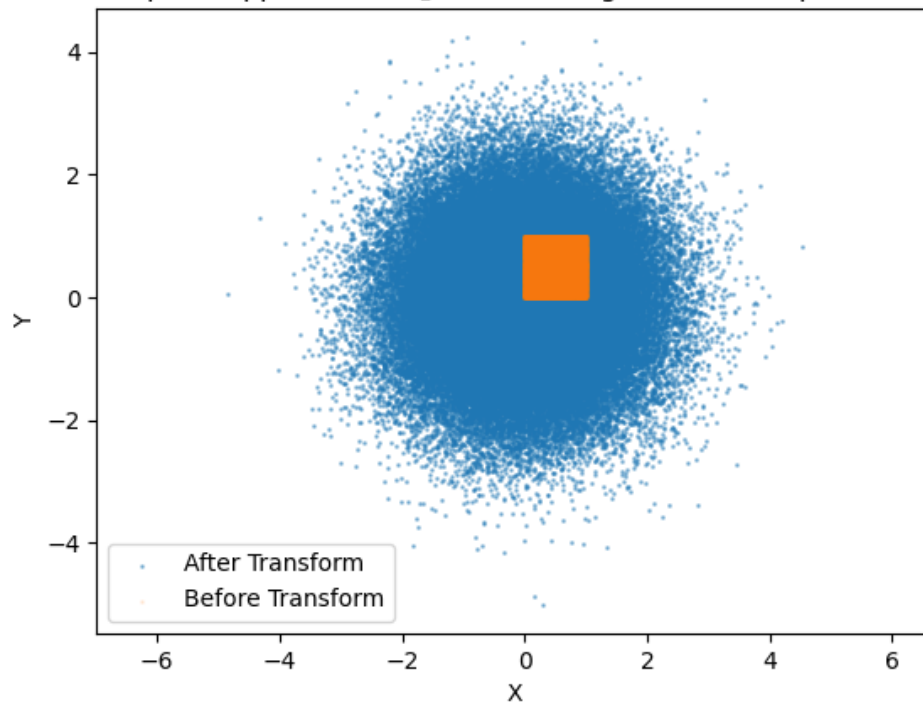
In [53]:
```python
# 2D scatter plot (joint distribution)
plt.figure()
plt.scatter(X, Y, s=1, alpha=0.4)
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Box-Muller Samples (Approx. $\mathcal{N}(0, I_2)$)")
plt.axis("equal")
plt.show()
```

Box–Muller Samples (Approx. $\mathcal{N}(0, I_2)$)

In [59]:
```python
# 2D scatter plot (joint distribution)
plt.figure()
plt.scatter(X, Y, s=1, alpha=0.4)
plt.scatter(U1, U2, s=1, alpha=0.1)
plt.xlabel("X")
plt.ylabel("Y")
plt.legend(["After Transform", "Before Transform"])
plt.title("Box-Muller Samples (Approx. $\mathcal{N}(0, I_2)$) with Orange as the sa
plt.axis("equal")
plt.show()
```

Box–Muller Samples (Approx. $\mathcal{N}(0, I_2)$) with Orange as the samples before Transform

The little orange square of uniformly distributed sample points became a symmetrically distributed gaussian blob of points in this 2D cartesian plane, verifying that Box-Muller algorithm is correct, simple, and fast.