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)
```

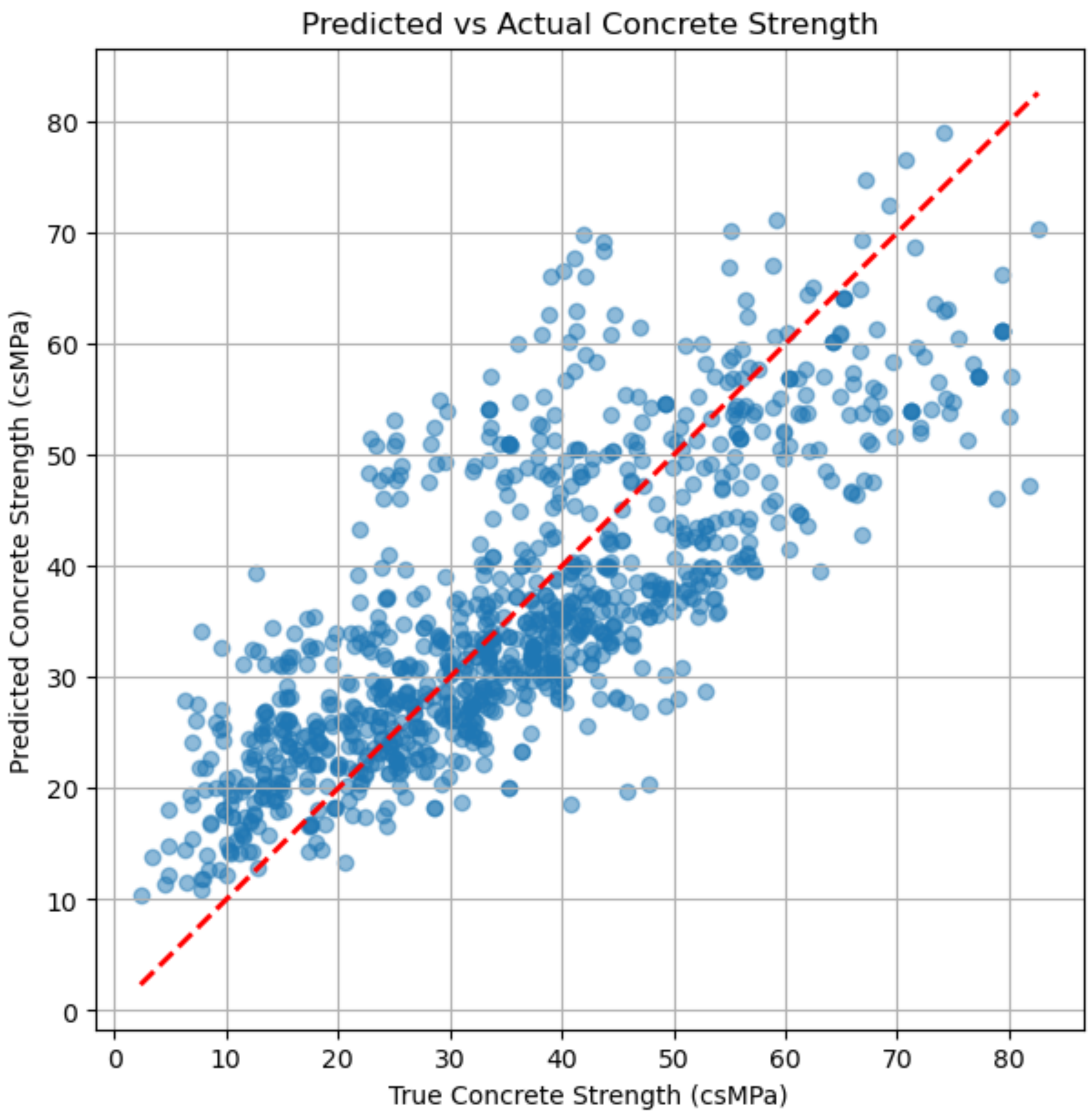
In [4]:
```
# Function to calculate optimal beta
def get_optimal_beta(X, Y):
    XTX = X.T @ X
    XTX_inv = np.linalg.inv(XTX)
    XTX_invX = XTX_inv @ X.T
    beta0 = XTX_invX @ Y
    return beta0

# 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.

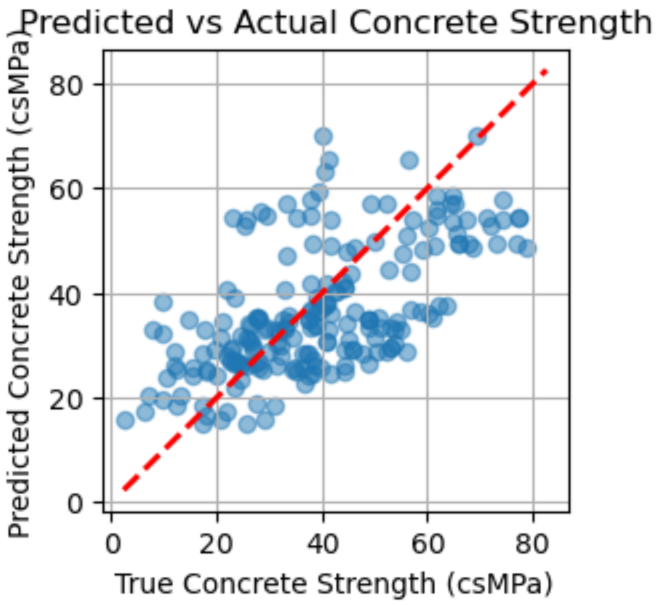
Note: The code in this section is a continuation of code in part 1, meaning that the code in part 1 should be run before running this code

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)
```

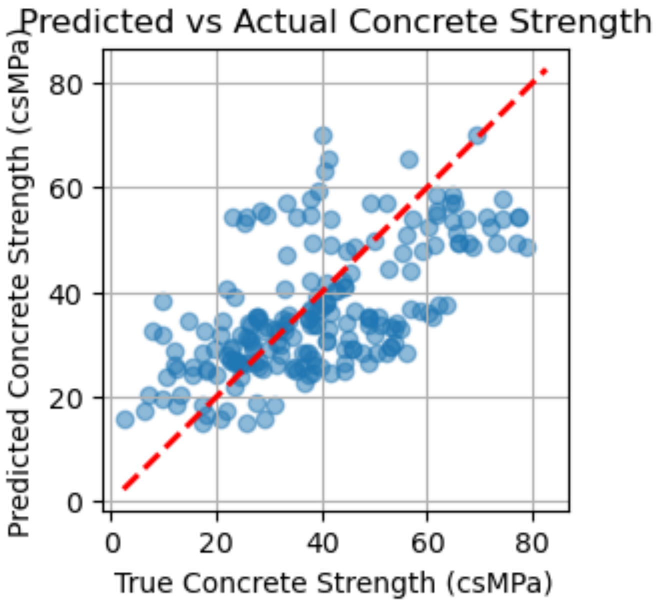
```
(824, 9) (824, 1)
(206, 9) (206, 1)
```

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))
```

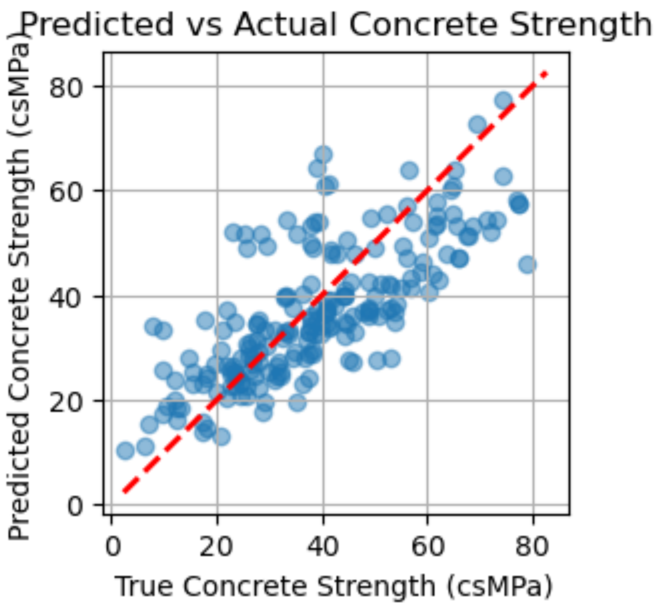
```
Running Number of columns = 7
Training Set X Shape: (824, 7)
beta shape: (7, 1)
185.65027353877565
```



```
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.

Problem 2: Multinomial Logistic Regression from Pre-trained Feature Extractor

Author: Yankun (Alex) Meng

In [1]:
```
import ssl
import torch
import torchvision
import matplotlib.pyplot as plt
from torchvision import transforms
from pretrained_model.Encoder import extractor
```

Step 1

In this problem, you will implement a logistic regression model from scratch to classify MNIST given a pre-trained feature extractor. Specifically, you will walk through the following steps:

Step 1: Load the pre-trained weights to your feature extractor. The code defining the architecture of your feature extractor is included in the **attachment**.

In [2]:
```
# Instantiate the model
device = "cpu" # i will just use cpu
feature_extractor = extractor().to(device)
```

In [3]:
```
# load the pretrained weights into the extractor
state_dict = torch.load("./pretrained_model/feature_extractor_weights.pth", map_loc
feature_extractor.load_state_dict(state_dict)
```

Out[3]:
```
<All keys matched successfully>
```

In [4]:
```
feature_extractor.eval()

for p in feature_extractor.parameters():
    p.requires_grad = False
```

Step 2

Step 2: Extract the latent representation (denoted as $h \in \mathbb{R}^k$) from each sample in MNIST (denoted as $x \in \mathbb{R}^{784}$) using the pre-trained feature extractor.

In [5]:
```
ssl._create_default_https_context = ssl._create_unverified_context
```

In [6]:
```
# Load MNIST from torchvision
train = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transforms.ToTensor()
)

test = torchvision.datasets.MNIST(
    root='./data',
    train=False,
    download=True,
    transform=transforms.ToTensor()
)
```

In [7]:
```
x, y = train[0] # (image, label)
print(x.shape)
print(y)
```

```
torch.Size([1, 28, 28])
5
```

In [8]:
```
# Create Data Loader wrappers
trainloader = torch.utils.data.DataLoader(
    train, batch_size=64, shuffle=True
)

testloader = torch.utils.data.DataLoader(
    test, batch_size=64, shuffle=False
)
```

Now we are ready to extract the features using our pretrained feature extractor.

In [9]:
```
# (H, Y)
H_train, Y_train = [], []
H_test,  Y_test  = [], []
```

In [10]:
```
# Loop Assignment
with torch.no_grad():
    # train set
    for x, y in trainloader:
        h = feature_extractor(x)
        H_train.append(h)
        Y_train.append(y)

    # test set
    for x, y in testloader:
        h = feature_extractor(x)
        H_test.append(h)
        Y_test.append(y)
```

In [11]:
```
# Concatenate all batches
H_train = torch.cat(H_train, dim=0)
Y_train = torch.cat(Y_train, dim=0)
H_test = torch.cat(H_test, dim=0)
Y_test = torch.cat(Y_test, dim=0)
print(H_train.shape)
print(Y_train.shape)
print(H_test.shape)
print(Y_test.shape)
```

```
torch.Size([60000, 256])
torch.Size([60000])
torch.Size([10000, 256])
torch.Size([10000])
```

Step 3: Derive the gradient of W and b with respect to the *cross-entropy* loss between the label and prediction $\hat{y}$ of the model:

$$\hat{y} = \sigma(W^T h + b), \tag{1}$$

where $\sigma(\cdot)$ denotes the softmax function.

$$\hat{\mathbf{y}} = \sigma(\mathbf{z}) = \sigma(\mathbf{W}^\top \mathbf{h} + \mathbf{b})$$

For a single sample with one-hot label $\mathbf{y} \in \mathbb{R}^C$, the cross-entropy loss is

$$L = -\sum_{c=1}^{C} y_c \log \hat{y}_c$$

For a batch of N samples, the average loss is

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log \hat{y}_{i,c}$$

We first compute the derivative of the loss with respect to the input to softmax $\mathbf{z}$

$$\begin{aligned}
\hat{y}_c &= \frac{e^{z_c}}{\sum_{j=1}^{C} e^{z_j}} \\
L &= -\sum_{k=1}^{C} y_k \log \hat{y}_k \\
\frac{\partial L}{\partial z_c} &= \sum_{k=1}^{C} \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_c} \\
\frac{\partial L}{\partial \hat{y}_k} &= -\frac{y_k}{\hat{y}_k} \\
\frac{\partial \hat{y}_k}{\partial z_c} &= \begin{cases} \hat{y}_c(1-\hat{y}_c), & k = c \\ -\hat{y}_k \hat{y}_c, & k \neq c \end{cases} \\
\frac{\partial L}{\partial z_c} &= -\frac{y_c}{\hat{y}_c} \cdot \hat{y}_c(1-\hat{y}_c) + \sum_{k \neq c} -\frac{y_k}{\hat{y}_k}(-\hat{y}_k \hat{y}_c) \\
&= \hat{y}_c - y_c
\end{aligned}$$

Vectorized for the batch

$$\frac{\partial L}{\partial \mathbf{Z}} = \hat{\mathbf{Y}} - \mathbf{Y},$$

where $\hat{\mathbf{Y}}, \mathbf{Y} \in \mathbb{R}^{N \times C}$.

Using the chain rule

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{Z}} \cdot \frac{\partial \mathbf{Z}}{\partial \mathbf{W}}$$

Since $\mathbf{Z} = \mathbf{H}\mathbf{W} + \mathbf{1}_N \mathbf{b}^\top$

$$\frac{\partial \mathbf{Z}}{\partial \mathbf{W}} = \mathbf{H}^\top$$

Thus, the gradient w.r.t. weights for the batch is

$$\boxed{\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{N} \mathbf{H}^\top (\hat{\mathbf{Y}} - \mathbf{Y})}$$

Similarly, the gradient w.r.t. bias is

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{Z}} \cdot \frac{\partial \mathbf{Z}}{\partial \mathbf{b}}$$

Since $\frac{\partial \mathbf{Z}}{\partial \mathbf{b}} = \mathbf{1}_N$ (broadcasted sum over batch)

$$\boxed{\frac{\partial L}{\partial \mathbf{b}} = \frac{1}{N} \sum_{i=1}^{N} (\hat{\mathbf{y}}_i - \mathbf{y}_i)}$$

with resulting shape $\mathbb{R}^C$.

Step 4: Manually implement Stochastic Gradient Descent (SGD) from *scratch (using matrix operations only)*. Use the gradient derived in **Step 3** to run training and update the parameters W and b. Finally, report the classification accuracy on the test set.

In [12]:
```python
# Parameters
num_classes = 10
num_features = H_train.shape[1]  # 256 from extractor
lr = 0.1
num_epochs = 20
batch_size = 256
```

In [13]:
```python
# Initialize weights and bias
W = torch.zeros((num_features, num_classes), dtype=torch.float32)
b = torch.zeros(num_classes, dtype=torch.float32)
```

In [14]:
```python
# Convert labels to one-hot
def to_one_hot(y, C):
    return torch.eye(C)[y]  # shape: [N, C]
```

```python
Y_train_onehot = to_one_hot(Y_train, num_classes)
Y_test_onehot  = to_one_hot(Y_test, num_classes)
```

In [15]:
```python
# Softmax function
def softmax(z):
    # subtract max for numerical stability
    exp_z = torch.exp(z - z.max(dim=1, keepdim=True)[0])
    return exp_z / exp_z.sum(dim=1, keepdim=True)
```

In [16]:
```python
# Cross-entropy loss
def cross_entropy(pred, target):
    return - (target * torch.log(pred + 1e-8)).sum(dim=1).mean()
```

In [17]:
```python
# Number of training samples
num_train = H_train.shape[0]

for epoch in range(num_epochs):
    # Shuffle dataset
    perm = torch.randperm(num_train)
    H_shuffled = H_train[perm]
    Y_shuffled = Y_train_onehot[perm]

    # Mini-batch SGD
    for i in range(0, num_train, batch_size):
        H_batch = H_shuffled[i:i+batch_size]  # [batch, features]
        Y_batch = Y_shuffled[i:i+batch_size]  # [batch, C]

        # Forward pass: linear + softmax
        z = H_batch @ W + b           # [batch, C]
        y_hat = softmax(z)            # [batch, C]

        # Compute gradients manually (from Step 3)
        dW = H_batch.T @ (y_hat - Y_batch) / H_batch.shape[0]  # [features, C]
        db = (y_hat - Y_batch).mean(dim=0)                      # [C]

        # Update parameters
        W -= lr * dW
        b -= lr * db

    # Compute training loss per epoch
    z_train = H_train @ W + b
    y_hat_train = softmax(z_train)
    loss = cross_entropy(y_hat_train, Y_train_onehot)
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")
```

```
Epoch 1/20, Loss: 0.0281
Epoch 2/20, Loss: 0.0181
Epoch 3/20, Loss: 0.0139
Epoch 4/20, Loss: 0.0115
Epoch 5/20, Loss: 0.0099
Epoch 6/20, Loss: 0.0087
Epoch 7/20, Loss: 0.0079
Epoch 8/20, Loss: 0.0072
Epoch 9/20, Loss: 0.0066
Epoch 10/20, Loss: 0.0061
Epoch 11/20, Loss: 0.0057
Epoch 12/20, Loss: 0.0054
Epoch 13/20, Loss: 0.0051
Epoch 14/20, Loss: 0.0048
Epoch 15/20, Loss: 0.0046
Epoch 16/20, Loss: 0.0044
Epoch 17/20, Loss: 0.0042
Epoch 18/20, Loss: 0.0040
Epoch 19/20, Loss: 0.0038
Epoch 20/20, Loss: 0.0037
```

In [18]:
```
# ---- Evaluate on test set ----
z_test = H_test @ W + b
y_hat_test = softmax(z_test)
y_pred = torch.argmax(y_hat_test, dim=1)

accuracy = (y_pred == Y_test).float().mean()
print(f"Test set classification accuracy: {accuracy.item() * 100:.2f}%")
```

```
Test set classification accuracy: 99.08%
```

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 Θ 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,

$$\begin{aligned}\frac{d}{dr}F_R(r) &= f_R(r)\\ &= re^{\frac{-r^2}{2}}, r \geq 0\end{aligned}$$

So in summary,

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

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

$$\begin{aligned}\implies f_{\Theta,R}(\theta, r) &= f_\Theta(\theta)f_R(r)\\ &= \frac{1}{2\pi}re^{\frac{-r^2}{2}} \qquad \text{where } 0 \leq \theta < 2\pi, r \geq 0\end{aligned}$$

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

$$\begin{aligned}f_{\Theta,R}(\theta, r) &= f_{X,Y}(x, y) \cdot r\\ \frac{1}{2\pi}re^{\frac{-r^2}{2}} &= f_{X,Y}(x, y) \cdot r\\ \frac{1}{2\pi}e^{\frac{-r^2}{2}} &= f_{X,Y}(x, y)\end{aligned}$$

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

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

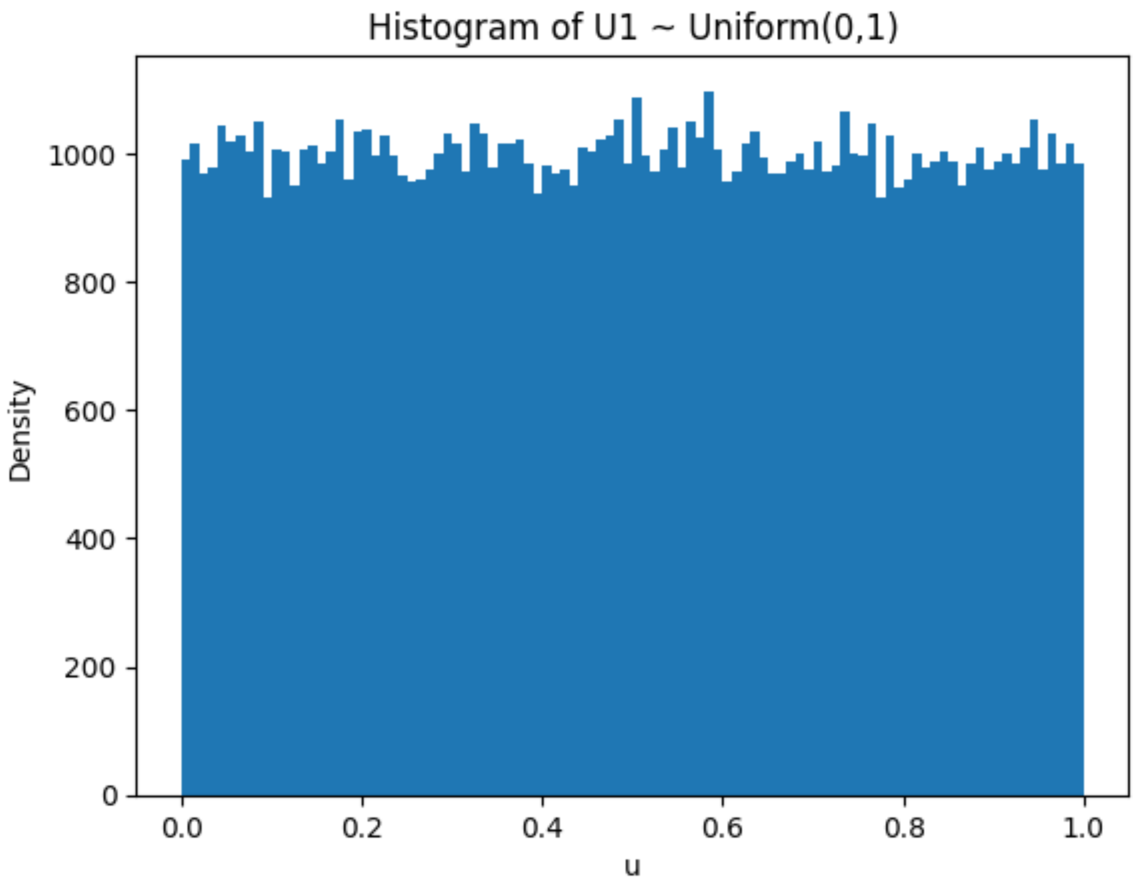
which was exactly the pdf of the standard bivariate normal distribution (equation (1)) defined in the beginning.

Plotting

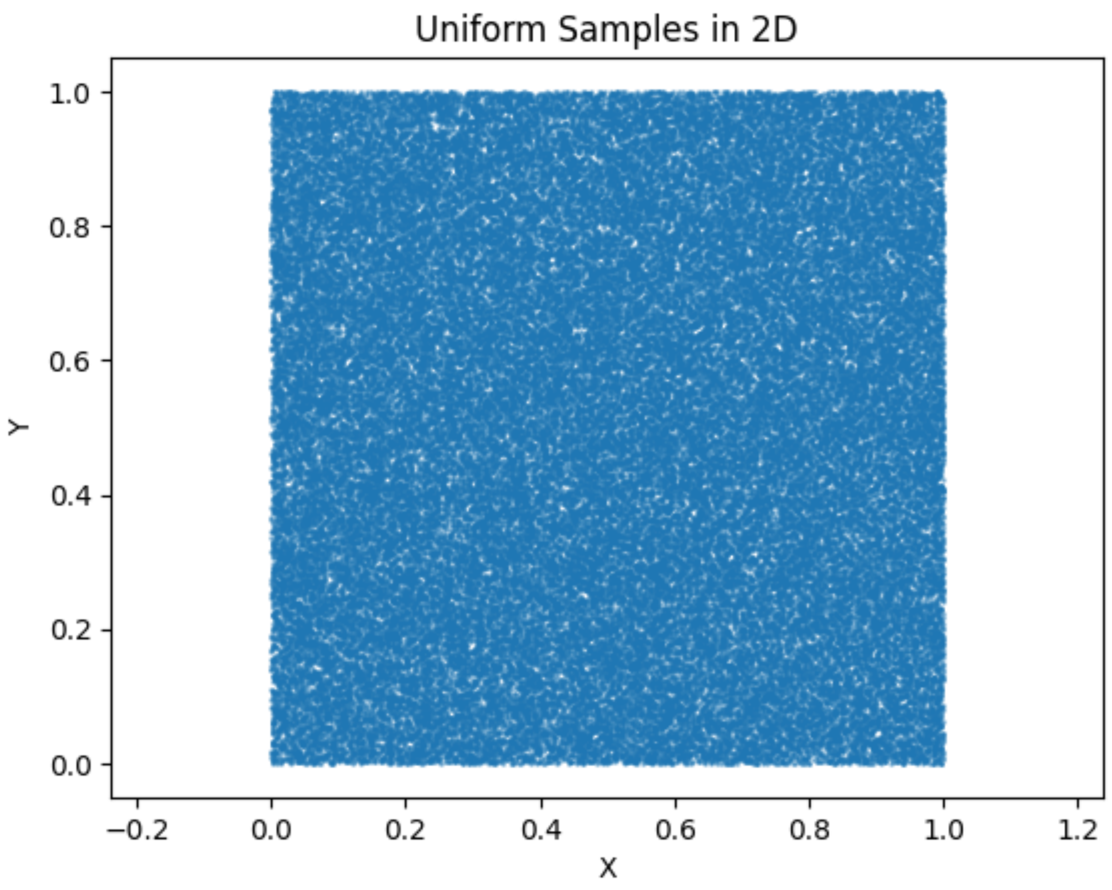
In [44]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
# 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()
```



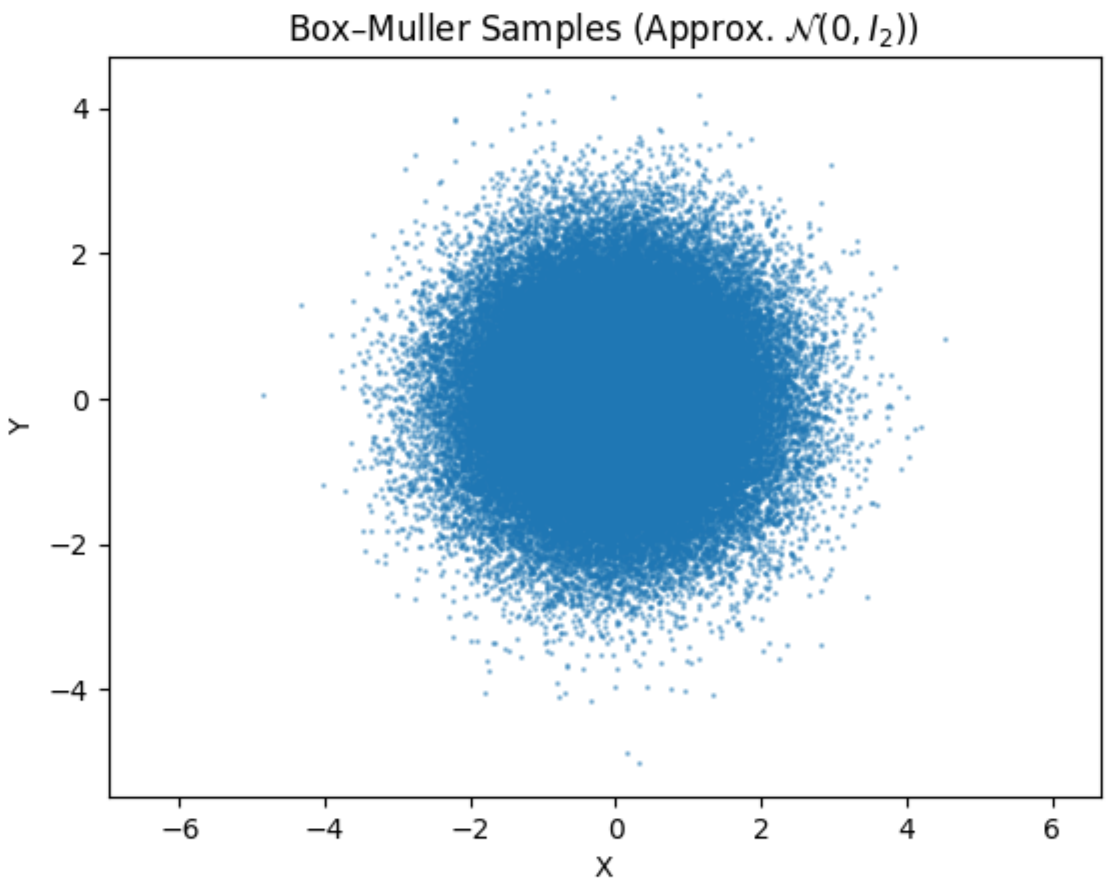
In [54]:
```
# 2D scatter plot (joint distribution)
plt.figure()
plt.scatter(U1, U2, s=1, alpha=0.4)
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Uniform Samples in 2D")
plt.axis("equal")
plt.show()
```



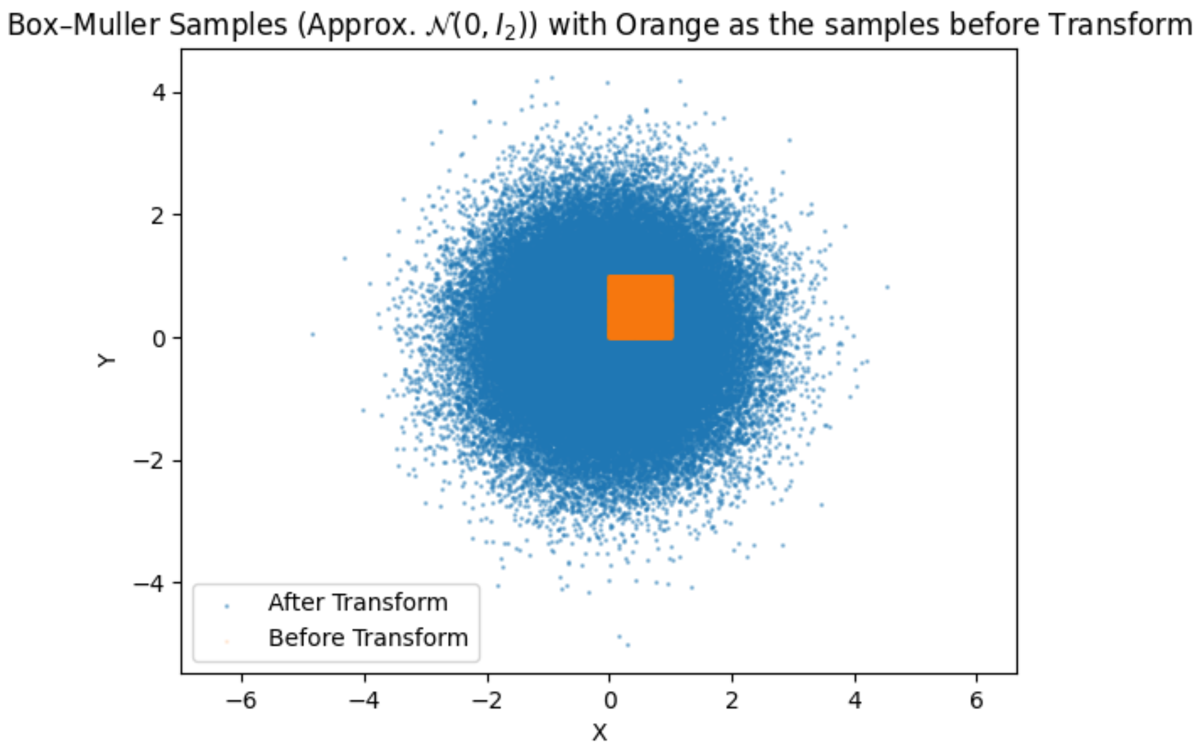
Indeed, these 100,000 draws looks pretty uniformly distributed to me.

In [47]:
```
# 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]:
```
# 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()
```



```
In [59]: # 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()
```



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.