MANE 4962 HW 3

Lucas Zhou 662005044

File Loaded

```
from google.colab import files
uploaded = files.upload()
```

Browse... 2 files selected.
housing_prices.txt(text/plain) - 1408 bytes, last modified: n/a - 100% done
HW3.1.jpg(image/jpeg) - 7952 bytes, last modified: n/a - 100% done

```
Saving housing_prices.txt to housing_prices.txt
Saving HW3.1.jpg to HW3.1.jpg
```

Problem 1:

Image segmentation is a process to highlight useful regions in images. Use the skimage.io module to load the following image. Afterwards, segment the image into multiple useful regions using the k-means clustering method. The segmented image should highlight, for example, the dashboard, the driver's arms, cars ahead etc., by grouping similar pixels together. You do not need to split the data into train and test set for this problem.

```
from skimage.io import imread
from skimage.color import rgb2lab
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt


# Load the image
image = imread('/content/HW3.1.jpg')

# Convert the image from RGB to Lab color space
```

```
image_lab = rgb2lab(image)

# Reshape the image to a 2D array of Lab color values
pixels = image_lab.reshape((-1, 3))

# Apply K-means clustering
kmeans = KMeans(n_clusters=6, random_state=42)
kmeans.fit(pixels)
labels = kmeans.labels_

# Reshape labels back to the original image dimensions
segmented_image = labels.reshape(image.shape[0], image.shape[1])

# Display the segmented image
plt.imshow(segmented_image, cmap='viridis')
plt.axis('off')
plt.show()
```

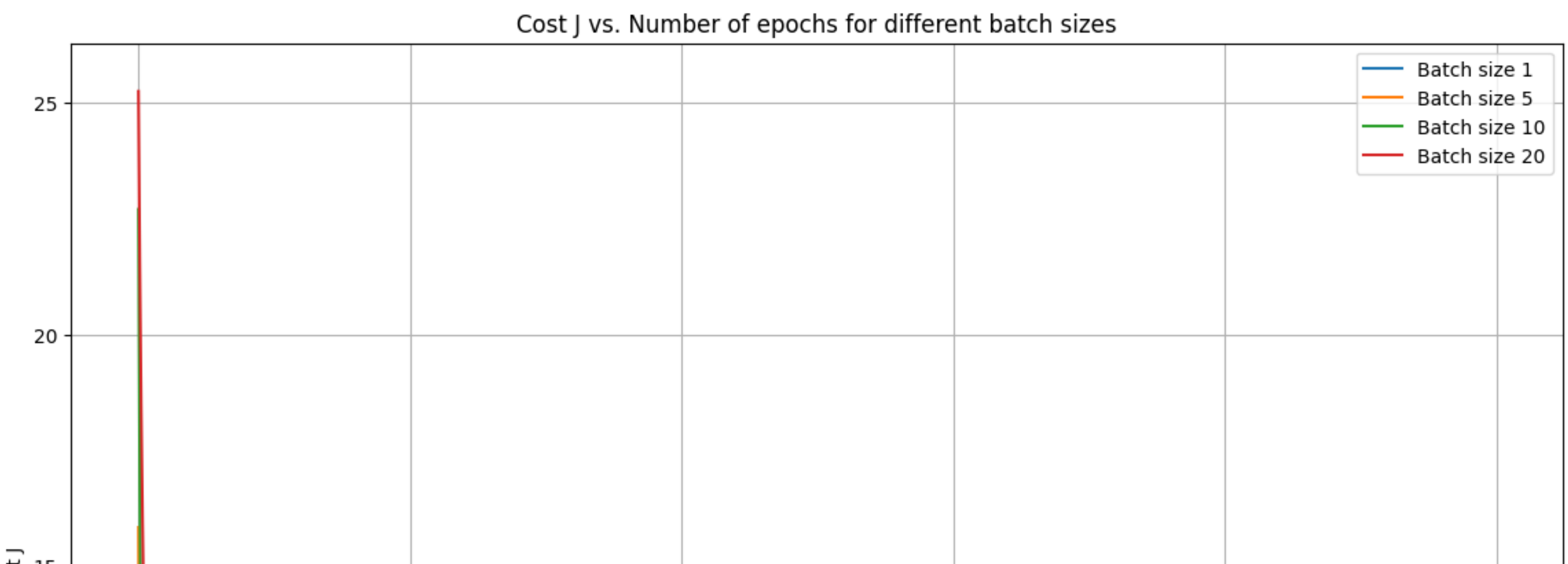
```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
```

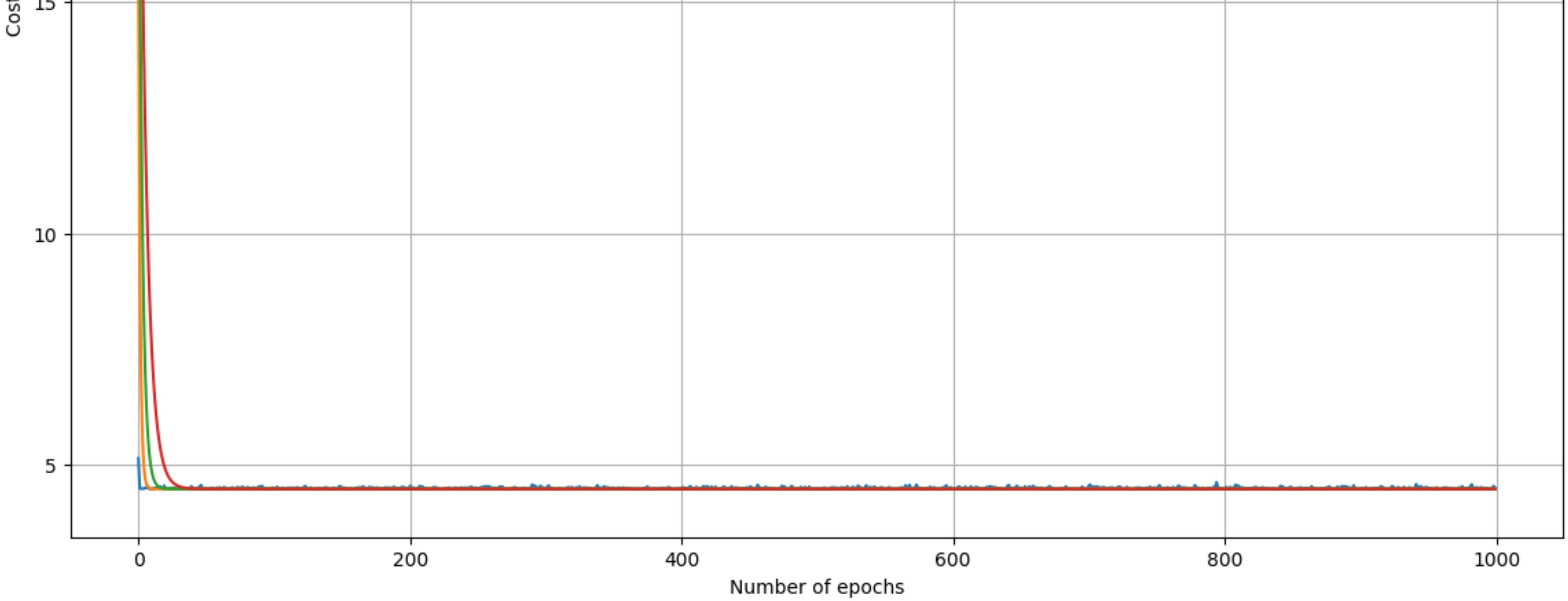
Problem 2:

Problem 2:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load the data
data_path = '/content/housing_prices.txt'
data = pd.read_csv(data_path, header=None, delimiter=',', skiprows=1)

# Extracting features (population) and target values (price)
X = data.iloc[:, 0].values  # Population
y = data.iloc[:, 1].values  # Housing price

# Normalize the features (population)
X_normalized = (X - np.mean(X)) / np.std(X)

# Adding a column of ones to the features to act as bias (intercept term)
X_b = np.c_[np.ones((len(X), 1)), X_normalized]

# Parameters for mini-batch gradient descent
learning_rate = 0.01
n_epochs = 1000
batch_sizes = [1, 5, 10, 20]
cost_history_batch = {}

# Mini-batch gradient descent function
def mini_batch_gradient_descent(X, y, learning_rate, batch_size, n_epochs):
    m = len(y)
    theta = np.random.randn(2, 1)  # Random initialization of weights
    cost_history = []

    for epoch in range(n_epochs):
        shuffled_indices = np.random.permutation(m)
        X_shuffled = X[shuffled_indices]
        y_shuffled = y[shuffled_indices].reshape(-1, 1)

        for i in range(0, m, batch_size):
            xi = X_shuffled[i:i + batch_size]
            yi = y_shuffled[i:i + batch_size]

            gradients = 2/batch_size * xi.T.dot(xi.dot(theta) - yi)
```

```
            theta = theta - learning_rate * gradients

            y_predict = X_b.dot(theta)
            cost = (1/(2*m)) * np.sum(np.square(y_predict - y.reshape(-1, 1)))
        cost_history.append(cost)

    return theta, cost_history

# Plotting the cost function for each batch size
plt.figure(figsize=(14, 10))

for batch_size in batch_sizes:
    theta, cost_history = mini_batch_gradient_descent(X_b, y, learning_rate, batch_size, n_epochs)
    cost_history_batch[batch_size] = cost_history
    plt.plot(range(len(cost_history)), cost_history, label=f'Batch size {batch_size}')

plt.xlabel('Number of epochs')
plt.ylabel('Cost J')
plt.title('Cost J vs. Number of epochs for different batch sizes')
plt.legend()
plt.grid()
plt.show()
```



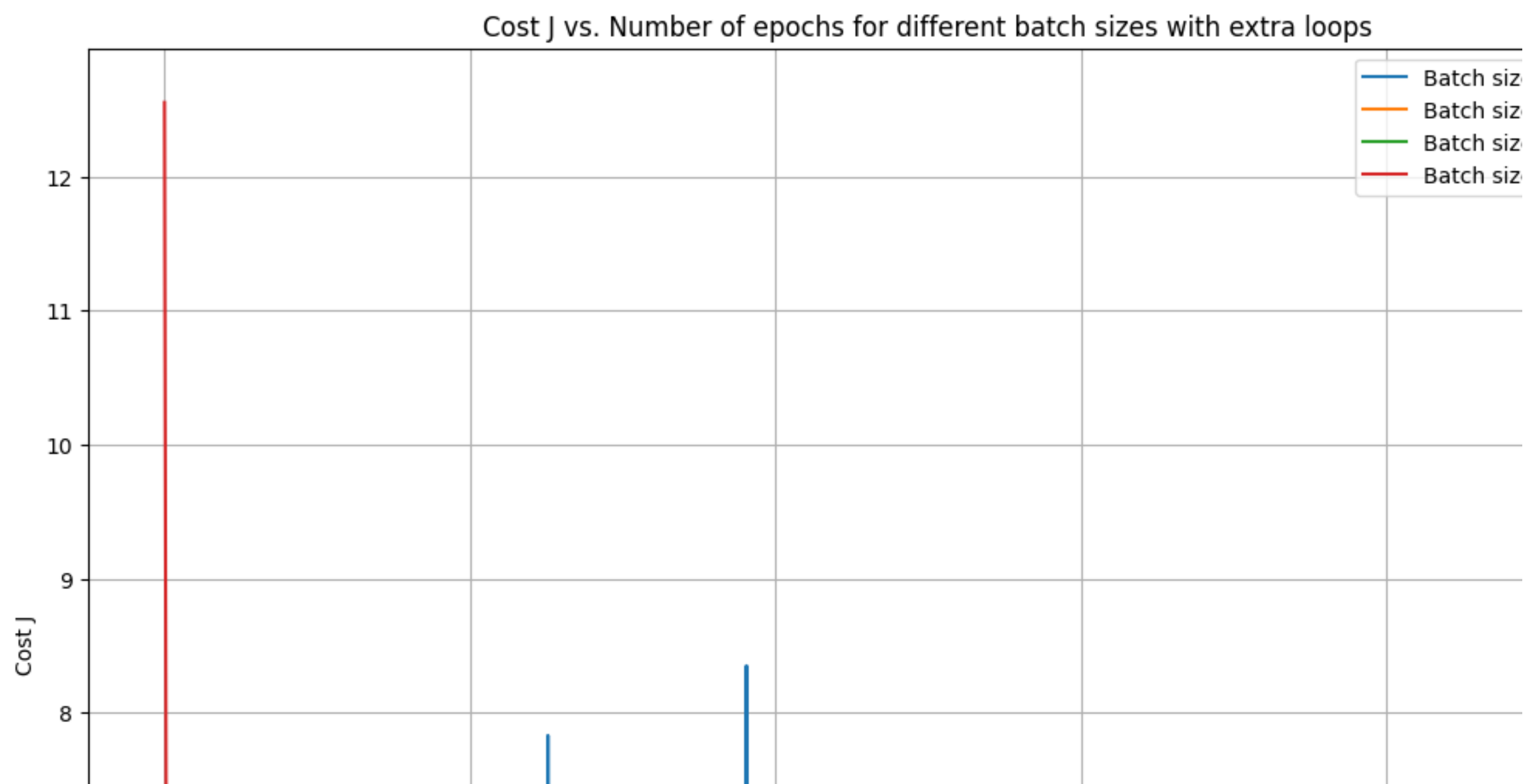


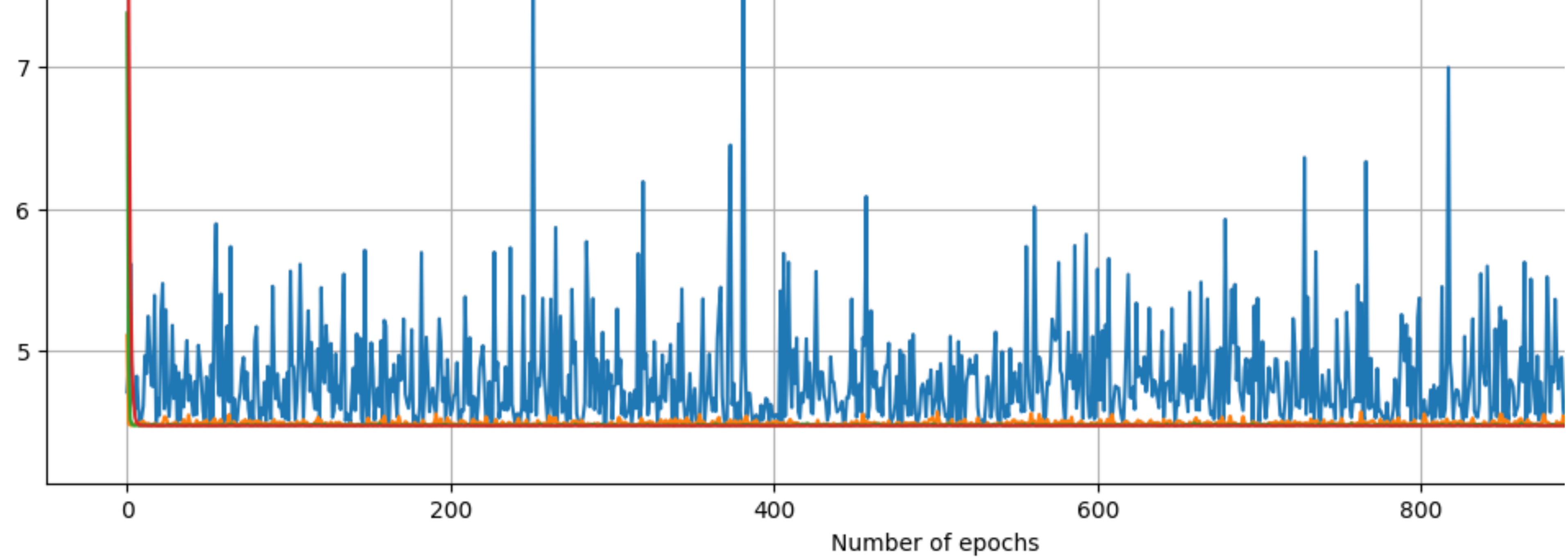
With extra loop

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load the data
data_path = '/content/housing_prices.txt'
data = pd.read_csv(data_path, header=None, delimiter=',', skiprows=1)

# Extracting features (population) and target values (price)
X = data.iloc[:, 0].values  # Population
y = data.iloc[:, 1].values  # Housing price
```

```
# Normalize the features (population)
X_normalized = (X - np.mean(X)) / np.std(X)

# Adding a column of ones to the features to act as bias (intercept term)
X_b = np.c_[np.ones((len(X), 1)), X_normalized]

# Parameters for mini-batch gradient descent
learning_rate = 0.01
n_epochs = 1000
batch_sizes = [1, 5, 10, 20]
cost_history_batch = {}

# Mini-batch gradient descent function
def mini_batch_gradient_descent_with_extra_loop(X, y, learning_rate, batch_size, n_epochs, extra_loops):
    m = len(y)
    theta = np.random.randn(2, 1)  # Random initialization of weights
    cost_history = []

    for epoch in range(n_epochs):
        shuffled_indices = np.random.permutation(m)
        X_shuffled = X[shuffled_indices]
        y_shuffled = y[shuffled_indices].reshape(-1, 1)

        for i in range(0, m, batch_size):
            xi = X_shuffled[i:i + batch_size]
            yi = y_shuffled[i:i + batch_size]

            for _ in range(extra_loops):  # Extra loop
                gradients = 2/batch_size * xi.T.dot(xi.dot(theta) - yi)
                theta = theta - learning_rate * gradients

            y_predict = X.dot(theta)
            cost = (1/(2*m)) * np.sum(np.square(y_predict - y.reshape(-1, 1)))
        cost_history.append(cost)

    return theta, cost_history

# Example usage
learning_rate = 0.01
n_epochs = 1000
batch_sizes = [1, 5, 10, 20]
extra_loops = 5  # Number of times the update step is repeated per batch
```

```python
cost_history_batch = {}

plt.figure(figsize=(14, 10))

for batch_size in batch_sizes:
    theta, cost_history = mini_batch_gradient_descent_with_extra_loop(X_b, y, learning_rate, batch_size, n_epochs, extra_loops)
    cost_history_batch[batch_size] = cost_history
    plt.plot(range(len(cost_history)), cost_history, label=f'Batch size {batch_size} with extra loops')

plt.xlabel('Number of epochs')
plt.ylabel('Cost J')
plt.title('Cost J vs. Number of epochs for different batch sizes with extra loops')
plt.legend()
plt.grid()
plt.show()
```





Problem 3:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import RFE
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import numpy as np

# Load dataset
data = load_breast_cancer()

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, test_size=0.3, random_state=42)  # 70% training and 30%

# Create a logistic regression model
logreg = LogisticRegression()

# Recursive Feature Elimination for feature selection
rfe = RFE(logreg, n_features_to_select=2)
rfe = rfe.fit(X_train, y_train)
```

```
rfe = rfe.fit(X_train, y_train)

# Identify which two features are considered to be the most important.
selected_features = np.where(rfe.support_ == True)[0]
print("Selected features:", selected_features)

# Retrain the model with the selected features
logreg.fit(X_train[:, selected_features], y_train)

# Predict test set
y_pred = logreg.predict(X_test[:, selected_features])

# Evaluate the model
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("Accuracy Score:", accuracy_score(y_test, y_pred))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (sta
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (sta
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (sta
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (sta
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (sta
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (sta
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (sta
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (sta
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

Problem 4:

Double-click (or enter) to edit

```
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import pandas as pd
```

```
import numpy as np

# Load and prepare the dataset
data = pd.read_csv('/content/housing_prices.txt')
X = data.iloc[:, 0].values.reshape(-1, 1)  # Features
y = data.iloc[:, 1].values.reshape(-1, 1)  # Labels

# Normalize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into a training set and a validation set
X_train, X_val, y_train, y_val = train_test_split(X_scaled, y, test_size=0.3, random_state=42)

# Build the neural network
model = tf.keras.Sequential([
    tf.keras.layers.Dense(2, activation='relu', input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(1)
])

# Compile the network
model.compile(optimizer='sgd', loss='mse')

# Train the network
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_val, y_val))

# Plot the training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.show()

# Predict the price of a house in a city with population of 165,000 (scaled accordingly)
population_165k_scaled = scaler.transform([[16.5]])  # The population feature scaled
predicted_price = model.predict(population_165k_scaled)
print("Predicted price of house:", predicted_price)

# Calculate a regression metric (Root Mean Squared Error, for example)
y_pred = model.predict(X_val)
mse = tf.keras.losses.MeanSquaredError()
rmse = np.sqrt(mse(y_val, y_pred).numpy())
print("Root Mean Squared Error on Validation Set:", rmse)
```

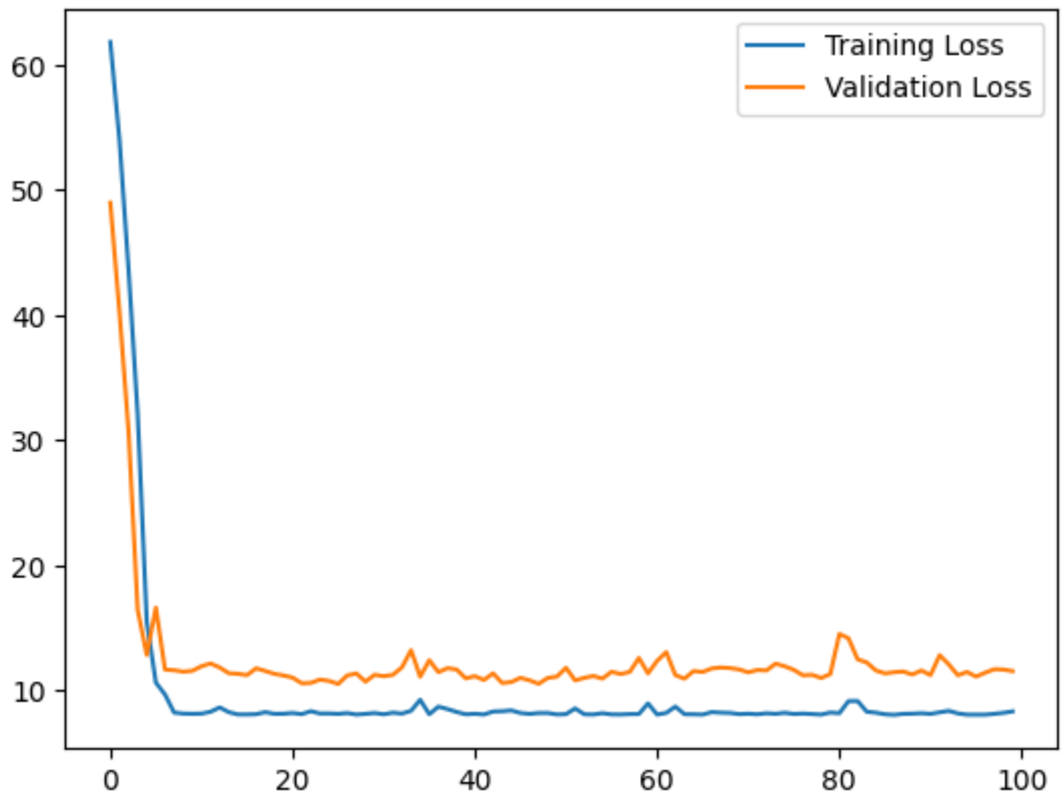
```
Epoch 1/100
3/3 [==============================] - 1s 183ms/step - loss: 61.8107 - val_loss: 48.9762
Epoch 2/100
3/3 [==============================] - 0s 42ms/step - loss: 54.0265 - val_loss: 40.2932
Epoch 3/100
3/3 [==============================] - 0s 24ms/step - loss: 43.6392 - val_loss: 30.7408
Epoch 4/100
3/3 [==============================] - 0s 28ms/step - loss: 32.1063 - val_loss: 16.4625
Epoch 5/100
3/3 [==============================] - 0s 23ms/step - loss: 15.4819 - val_loss: 12.8520
Epoch 6/100
3/3 [==============================] - 0s 26ms/step - loss: 10.6462 - val_loss: 16.6331
Epoch 7/100
3/3 [==============================] - 0s 30ms/step - loss: 9.6925 - val_loss: 11.6862
Epoch 8/100
3/3 [==============================] - 0s 42ms/step - loss: 8.2531 - val_loss: 11.6372
Epoch 9/100
3/3 [==============================] - 0s 38ms/step - loss: 8.1725 - val_loss: 11.5187
Epoch 10/100
3/3 [==============================] - 0s 67ms/step - loss: 8.1572 - val_loss: 11.5903
Epoch 11/100
3/3 [==============================] - 0s 40ms/step - loss: 8.1682 - val_loss: 11.9516
Epoch 12/100
3/3 [==============================] - 0s 50ms/step - loss: 8.3205 - val_loss: 12.1779
Epoch 13/100
3/3 [==============================] - 0s 43ms/step - loss: 8.6661 - val_loss: 11.8534
Epoch 14/100
3/3 [==============================] - 0s 34ms/step - loss: 8.2870 - val_loss: 11.3969
Epoch 15/100
3/3 [==============================] - 0s 37ms/step - loss: 8.1053 - val_loss: 11.3514
Epoch 16/100
3/3 [==============================] - 0s 25ms/step - loss: 8.1044 - val_loss: 11.2437
Epoch 17/100
3/3 [==============================] - 0s 35ms/step - loss: 8.1257 - val_loss: 11.7923
Epoch 18/100
3/3 [==============================] - 0s 34ms/step - loss: 8.2879 - val_loss: 11.5736
Epoch 19/100
3/3 [==============================] - 0s 23ms/step - loss: 8.1538 - val_loss: 11.3453
Epoch 20/100
3/3 [==============================] - 0s 22ms/step - loss: 8.1594 - val_loss: 11.2190
Epoch 21/100
3/3 [==============================] - 0s 25ms/step - loss: 8.2188 - val_loss: 11.0476
Epoch 22/100
3/3 [==============================] - 0s 24ms/step - loss: 8.1294 - val_loss: 10.5930
Epoch 23/100
3/3 [==============================] - 0s 23ms/step - loss: 8.3639 - val_loss: 10.6218
```

```
Epoch 24/100
3/3 [==============================] - 0s 30ms/step - loss: 8.1812 - val_loss: 10.8875
Epoch 25/100
3/3 [==============================] - 0s 36ms/step - loss: 8.1840 - val_loss: 10.7819
Epoch 26/100
3/3 [==============================] - 0s 41ms/step - loss: 8.1506 - val_loss: 10.5418
Epoch 27/100
3/3 [==============================] - 0s 76ms/step - loss: 8.2148 - val_loss: 11.2224
Epoch 28/100
3/3 [==============================] - 0s 37ms/step - loss: 8.0977 - val_loss: 11.3770
Epoch 29/100
3/3 [==============================] - 0s 62ms/step - loss: 8.1495 - val_loss: 10.7216
Epoch 30/100
3/3 [==============================] - 0s 47ms/step - loss: 8.2175 - val_loss: 11.2731
Epoch 31/100
3/3 [==============================] - 0s 47ms/step - loss: 8.1160 - val_loss: 11.1600
Epoch 32/100
3/3 [==============================] - 0s 44ms/step - loss: 8.2451 - val_loss: 11.2674
Epoch 33/100
3/3 [==============================] - 0s 51ms/step - loss: 8.1715 - val_loss: 11.8642
Epoch 34/100
3/3 [==============================] - 0s 38ms/step - loss: 8.3956 - val_loss: 13.2371
Epoch 35/100
3/3 [==============================] - 0s 60ms/step - loss: 9.2710 - val_loss: 11.1157
Epoch 36/100
3/3 [==============================] - 0s 48ms/step - loss: 8.1201 - val_loss: 12.4301
Epoch 37/100
3/3 [==============================] - 0s 71ms/step - loss: 8.7291 - val_loss: 11.4843
Epoch 38/100
3/3 [==============================] - 0s 71ms/step - loss: 8.5315 - val_loss: 11.8021
Epoch 39/100
3/3 [==============================] - 0s 44ms/step - loss: 8.3012 - val_loss: 11.6926
Epoch 40/100
3/3 [==============================] - 0s 40ms/step - loss: 8.1168 - val_loss: 10.9902
Epoch 41/100
3/3 [==============================] - 0s 47ms/step - loss: 8.1561 - val_loss: 11.1581
Epoch 42/100
3/3 [==============================] - 0s 43ms/step - loss: 8.1006 - val_loss: 10.8593
Epoch 43/100
3/3 [==============================] - 0s 45ms/step - loss: 8.3374 - val_loss: 11.3879
Epoch 44/100
3/3 [==============================] - 0s 50ms/step - loss: 8.3622 - val_loss: 10.6121
Epoch 45/100
3/3 [==============================] - 0s 48ms/step - loss: 8.4256 - val_loss: 10.6921
Epoch 46/100
3/3 [==============================] - 0s 67ms/step - loss: 8.2201 - val_loss: 11.0268
```

```
Epoch 47/100
3/3 [==============================] - 0s 66ms/step - loss: 8.1482 - val_loss: 10.8303
Epoch 48/100
3/3 [==============================] - 0s 52ms/step - loss: 8.2154 - val_loss: 10.5520
Epoch 49/100
3/3 [==============================] - 0s 57ms/step - loss: 8.2188 - val_loss: 11.0206
Epoch 50/100
3/3 [==============================] - 0s 69ms/step - loss: 8.1038 - val_loss: 11.1241
Epoch 51/100
3/3 [==============================] - 0s 56ms/step - loss: 8.1284 - val_loss: 11.8324
Epoch 52/100
3/3 [==============================] - 0s 40ms/step - loss: 8.5845 - val_loss: 10.8238
Epoch 53/100
3/3 [==============================] - 0s 129ms/step - loss: 8.1181 - val_loss: 11.0247
Epoch 54/100
3/3 [==============================] - 0s 91ms/step - loss: 8.1018 - val_loss: 11.1764
Epoch 55/100
3/3 [==============================] - 0s 63ms/step - loss: 8.2028 - val_loss: 10.9652
Epoch 56/100
3/3 [==============================] - 0s 40ms/step - loss: 8.1078 - val_loss: 11.5153
Epoch 57/100
3/3 [==============================] - 0s 58ms/step - loss: 8.1025 - val_loss: 11.3348
Epoch 58/100
3/3 [==============================] - 0s 115ms/step - loss: 8.1327 - val_loss: 11.4997
Epoch 59/100
3/3 [==============================] - 0s 56ms/step - loss: 8.1391 - val_loss: 12.6139
Epoch 60/100
3/3 [==============================] - 0s 22ms/step - loss: 8.9869 - val_loss: 11.3842
Epoch 61/100
3/3 [==============================] - 0s 22ms/step - loss: 8.0921 - val_loss: 12.3851
Epoch 62/100
3/3 [==============================] - 0s 23ms/step - loss: 8.2261 - val_loss: 13.0813
Epoch 63/100
3/3 [==============================] - 0s 23ms/step - loss: 8.7312 - val_loss: 11.2493
Epoch 64/100
3/3 [==============================] - 0s 26ms/step - loss: 8.1226 - val_loss: 10.9615
Epoch 65/100
3/3 [==============================] - 0s 21ms/step - loss: 8.1171 - val_loss: 11.5708
Epoch 66/100
3/3 [==============================] - 0s 43ms/step - loss: 8.0947 - val_loss: 11.4990
Epoch 67/100
3/3 [==============================] - 0s 27ms/step - loss: 8.2949 - val_loss: 11.7784
Epoch 68/100
3/3 [==============================] - 0s 57ms/step - loss: 8.2477 - val_loss: 11.8535
Epoch 69/100
3/3 [==============================] - 0s 27ms/step - loss: 8.2248 - val_loss: 11.8072
```

```
Epoch 70/100
3/3 [==============================] - 0s 34ms/step - loss: 8.1312 - val_loss: 11.6971
Epoch 71/100
3/3 [==============================] - 0s 31ms/step - loss: 8.1598 - val_loss: 11.4680
Epoch 72/100
3/3 [==============================] - 0s 26ms/step - loss: 8.1204 - val_loss: 11.6554
Epoch 73/100
3/3 [==============================] - 0s 29ms/step - loss: 8.1965 - val_loss: 11.6185
Epoch 74/100
3/3 [==============================] - 0s 23ms/step - loss: 8.1527 - val_loss: 12.1610
Epoch 75/100
3/3 [==============================] - 0s 23ms/step - loss: 8.2373 - val_loss: 11.9508
Epoch 76/100
3/3 [==============================] - 0s 23ms/step - loss: 8.1420 - val_loss: 11.6652
Epoch 77/100
3/3 [==============================] - 0s 23ms/step - loss: 8.1731 - val_loss: 11.2306
Epoch 78/100
3/3 [==============================] - 0s 30ms/step - loss: 8.1369 - val_loss: 11.2663
Epoch 79/100
3/3 [==============================] - 0s 61ms/step - loss: 8.0939 - val_loss: 11.0184
Epoch 80/100
3/3 [==============================] - 0s 27ms/step - loss: 8.2562 - val_loss: 11.3274
Epoch 81/100
3/3 [==============================] - 0s 98ms/step - loss: 8.2075 - val_loss: 14.5363
Epoch 82/100
3/3 [==============================] - 0s 87ms/step - loss: 9.1698 - val_loss: 14.1862
Epoch 83/100
3/3 [==============================] - 0s 64ms/step - loss: 9.1747 - val_loss: 12.5056
Epoch 84/100
3/3 [==============================] - 0s 51ms/step - loss: 8.3241 - val_loss: 12.2567
Epoch 85/100
3/3 [==============================] - 0s 24ms/step - loss: 8.2448 - val_loss: 11.6183
Epoch 86/100
3/3 [==============================] - 0s 14ms/step - loss: 8.1174 - val_loss: 11.3828
Epoch 87/100
3/3 [==============================] - 0s 14ms/step - loss: 8.0730 - val_loss: 11.4792
Epoch 88/100
3/3 [==============================] - 0s 15ms/step - loss: 8.1513 - val_loss: 11.5324
Epoch 89/100
3/3 [==============================] - 0s 15ms/step - loss: 8.1654 - val_loss: 11.2931
Epoch 90/100
3/3 [==============================] - 0s 14ms/step - loss: 8.2005 - val_loss: 11.6131
Epoch 91/100
3/3 [==============================] - 0s 14ms/step - loss: 8.1503 - val_loss: 11.2455
Epoch 92/100
3/3 [==============================] - 0s 14ms/step - loss: 8.2755 - val_loss: 12.8405
```

```
Epoch 93/100
3/3 [==============================] - 0s 13ms/step - loss: 8.3918 - val_loss: 12.1001
Epoch 94/100
3/3 [==============================] - 0s 14ms/step - loss: 8.1854 - val_loss: 11.2332
Epoch 95/100
3/3 [==============================] - 0s 20ms/step - loss: 8.0901 - val_loss: 11.4974
Epoch 96/100
3/3 [==============================] - 0s 14ms/step - loss: 8.0860 - val_loss: 11.1255
Epoch 97/100
3/3 [==============================] - 0s 15ms/step - loss: 8.0865 - val_loss: 11.4406
Epoch 98/100
3/3 [==============================] - 0s 15ms/step - loss: 8.1527 - val_loss: 11.7155
Epoch 99/100
3/3 [==============================] - 0s 15ms/step - loss: 8.2314 - val_loss: 11.6893
Epoch 100/100
3/3 [==============================] - 0s 14ms/step - loss: 8.3477 - val_loss: 11.5645
```




```
1/1 [==============================] - 0s 83ms/step
Predicted price of house: [[16.034851]]
1/1 [==============================] - 0s 21ms/step
Root Mean Squared Error on Validation Set: 3.400657
```