Project Optimization for Machine Learning

Maxence Lasbordes | M2 MASH

In [77]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import torch
```

Question 1

For this project we will use the a9a dataset. This dataset is a binary classification problem with sparse features. The goal is to predict whether an individual makes over 50K a year based on the features provided. The dataset is available at the following URL: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/a9a

In [78]:
```
with open('a9a.txt', 'r') as file:
    lines = file.readlines()

labels = []
features = []

for line in lines:
    parts = line.strip().split()

    labels.append(int(parts[0]))

    feature_dict = {}
    for feat in parts[1:]:
        index, value = feat.split(':')
        feature_dict[int(index)] = float(value)

    features.append(feature_dict)
```

The dataset contains 32561 samples and 123 features. The features are very sparse. We are only focusing on the training set which contains 32561 samples.

In [79]:
```
print('Number of samples:', len(labels))
print('Number of features:', len(features[0]))
print(labels[:5])
print(features[:5])
```

```
Number of samples: 32561
Number of features: 14
[-1, -1, -1, -1, -1]
[{3: 1.0, 11: 1.0, 14: 1.0, 19: 1.0, 39: 1.0, 42: 1.0, 55: 1.0, 64: 1.0, 67: 1.0, 73: 1.0, 75: 1.0, 76: 1.0, 80
: 1.0, 83: 1.0}, {5: 1.0, 7: 1.0, 14: 1.0, 19: 1.0, 39: 1.0, 40: 1.0, 51: 1.0, 63: 1.0, 67: 1.0, 73: 1.0, 74: 1
.0, 76: 1.0, 78: 1.0, 83: 1.0}, {3: 1.0, 6: 1.0, 17: 1.0, 22: 1.0, 36: 1.0, 41: 1.0, 53: 1.0, 64: 1.0, 67: 1.0,
73: 1.0, 74: 1.0, 76: 1.0, 80: 1.0, 83: 1.0}, {5: 1.0, 6: 1.0, 17: 1.0, 21: 1.0, 35: 1.0, 40: 1.0, 53: 1.0, 63:
1.0, 71: 1.0, 73: 1.0, 74: 1.0, 76: 1.0, 80: 1.0, 83: 1.0}, {2: 1.0, 6: 1.0, 18: 1.0, 19: 1.0, 39: 1.0, 40: 1.0
, 52: 1.0, 61: 1.0, 71: 1.0, 72: 1.0, 74: 1.0, 76: 1.0, 80: 1.0, 95: 1.0}]
```

We can see that the dataset is in a sparse format. Only the non-zero features are specified.

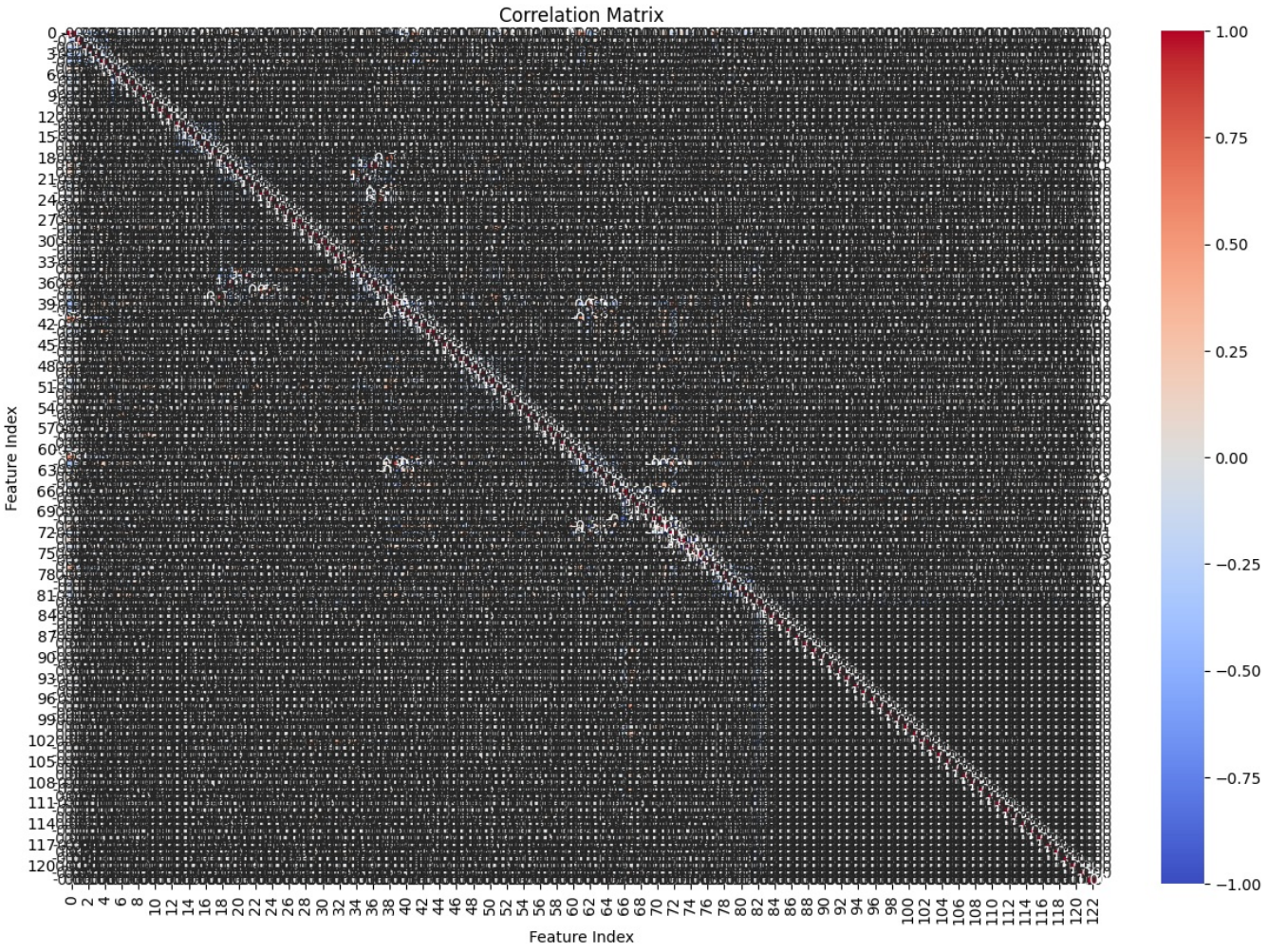
In [80]:
```
max_index = max([max(feat.keys()) for feat in features if feat])
dense_features = np.zeros((len(features), max_index))

# The features are 1-indexed so we shift them to 0-indexed
for i, feat in enumerate(features):
    for index, value in feat.items():
        dense_features[i, index-1] = value
print(dense_features[0])
```

```
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0.
 1. 0. 1. 1. 0. 0. 0. 1. 0. 0. 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.]
```

We now have the full dense matrix, so we can generate the correlation matrix.

In [81]:
```
dataframe = pd.DataFrame(dense_features)
correlation_matrix = dataframe.corr()

plt.figure(figsize=(15, 10))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="coolwarm", vmin=-1, vmax=1)
plt.title("Correlation Matrix")
plt.xlabel("Feature Index")
```

```
plt.ylabel("Feature Index")
plt.show()
```



Because of the huge number of features (123), it is very hard to visualize correctly the correlation matrix.

Qestion 2

We will build the f function and the gradient of the f function.

In [82]:
```
def sigmoid(x):
    return 1 / (1 + torch.exp(-x))

def f_i(x, a_i, y_i):
    return (y_i - sigmoid(torch.dot(a_i, x))) ** 2

def grad_f_i_explicit(x, a_i, y_i):
    exp_term = torch.exp(torch.dot(a_i, x))
    numerator = 2 * exp_term * (exp_term * (y_i - 1) + y_i)
    denominator = (1 + exp_term) ** 3
    return - (numerator / denominator) * a_i
```

We will now compare the explicit computation of the gradient with the gradient from the automatic differentiation.

In [83]:
```
a_i = torch.tensor(dense_features[0], dtype=torch.float32)
y_i = torch.tensor(labels[0], dtype=torch.float32)
x = torch.randn(a_i.size(0), requires_grad=True)


f_value = f_i(x, a_i, y_i)
f_value.backward() # Compute the gradients
grad_f_i_autograd = x.grad # Get the gradients
grad_f_i_explicit = grad_f_i_explicit(x, a_i, y_i)

print("Autograd gradient:", grad_f_i_autograd)
print("Explicit gradient:", grad_f_i_explicit)

# We check if the gradients match with a small tollerance
if torch.allclose(grad_f_i_autograd, grad_f_i_explicit):
    print("\nValidation: Gradients match !!!")
else:
    print("\nValidation: Gradients do not match !!!")
```

```
Autograd gradient: tensor([0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0364, 0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0364, 0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0364, 0.0000, 0.0364, 0.0364, 0.0000, 0.0000, 0.0000, 0.0364, 0.0000,
        0.0000, 0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000])
Explicit gradient: tensor([0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0364, 0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0364, 0.0000, 0.0000, 0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0364, 0.0000, 0.0364, 0.0364, 0.0000, 0.0000, 0.0000, 0.0364, 0.0000,
        0.0000, 0.0364, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       grad_fn=<MulBackward0>)

Validation: Gradients match !!!
```

As expected, the gradients are indeed matching !

Question 3

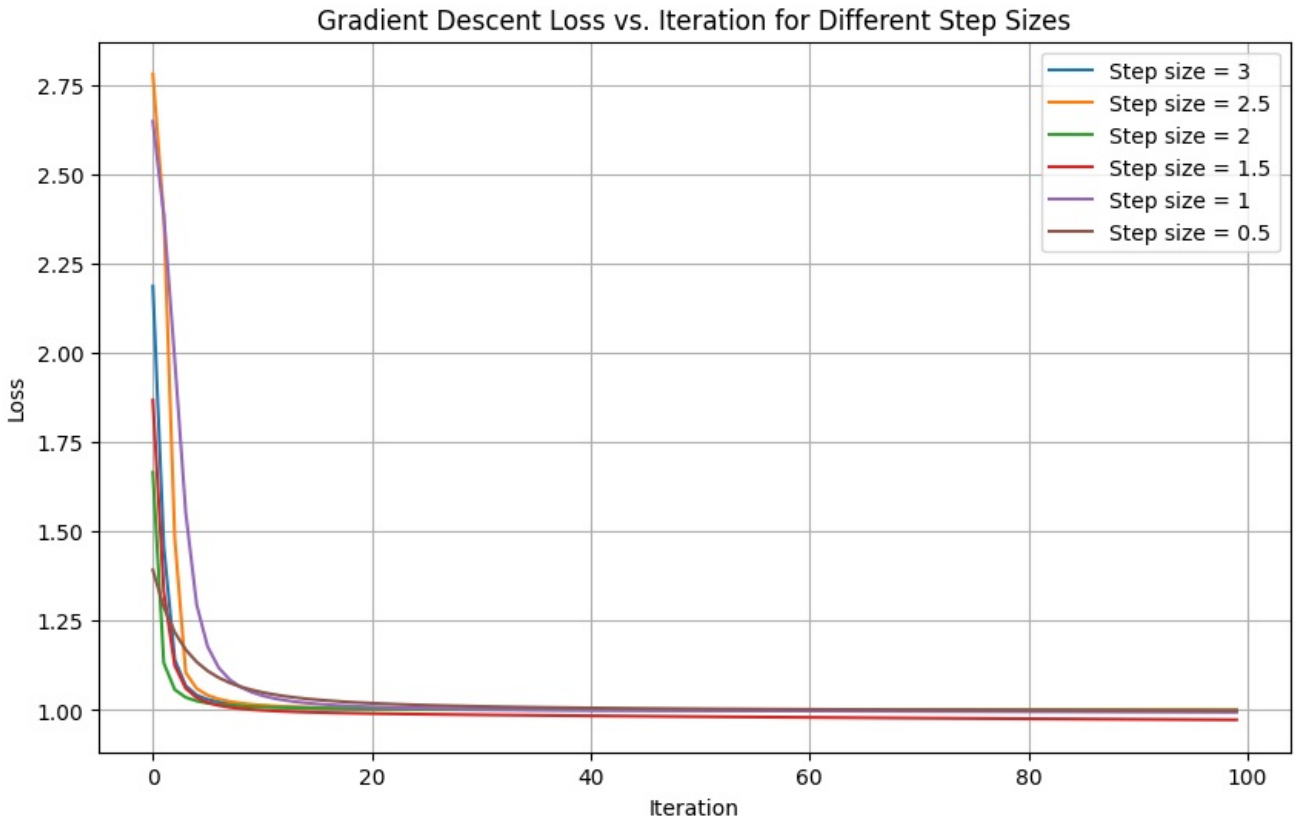
Implementation of gradient descent for our problem. We reused a part of the code from the Lab 2.

In [84]:
```python
def f(x, A, y):
    sigmoid = 1 / (1 + torch.exp(-A @ x))
    return torch.mean((y - sigmoid) ** 2)
```

In [85]:
```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Let's test the function with a small subset of 1000 data points to make it faster
A = torch.tensor(dense_features[:1000], dtype=torch.float32, device=device)
y = torch.tensor(labels[:1000], dtype=torch.float32, device=device)
x = torch.randn(A.size(1), requires_grad=True, device=device)
```

In [86]:
```python
nb_iters = 100
step_sizes = [3, 2.5, 2, 1.5, 1, 0.5]

losses_dict = {stepsize: [] for stepsize in step_sizes}

for stepsize in step_sizes:
    x = torch.randn(A.size(1), requires_grad=True, device=device)
    for i in range(nb_iters):
        loss = f(x, A, y)
        loss.backward()
        with torch.no_grad():
            x -= stepsize*x.grad
            x.grad.zero_()

        losses_dict[stepsize].append(loss.detach().item())

        if i % 10 == 0:
            print(f"Step size {stepsize}, Iteration {i}/{nb_iters} : Loss : {loss.item()}")

# Plot the losses for each step size
plt.figure(figsize=(10, 6))
for stepsize, losses in losses_dict.items():
    plt.plot(range(nb_iters ), losses, label=f"Step size = {stepsize}")

plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Gradient Descent Loss vs. Iteration for Different Step Sizes")
plt.legend()
plt.grid()
plt.show()
```

```
Step size 3, Iteration 0/100 : Loss : 2.186828851699829
Step size 3, Iteration 10/100 : Loss : 1.0111451148986816
Step size 3, Iteration 20/100 : Loss : 1.0033867359161377
Step size 3, Iteration 30/100 : Loss : 1.001236081123352
Step size 3, Iteration 40/100 : Loss : 1.0003330707550049
Step size 3, Iteration 50/100 : Loss : 0.9998317956924438
Step size 3, Iteration 60/100 : Loss : 0.9994914531707764
Step size 3, Iteration 70/100 : Loss : 0.9992251396179199
Step size 3, Iteration 80/100 : Loss : 0.9989950656890869
Step size 3, Iteration 90/100 : Loss : 0.998782217502594
Step size 2.5, Iteration 0/100 : Loss : 2.780731439590454
Step size 2.5, Iteration 10/100 : Loss : 1.013646125793457
Step size 2.5, Iteration 20/100 : Loss : 1.004689335823059
Step size 2.5, Iteration 30/100 : Loss : 1.0026336908340454
Step size 2.5, Iteration 40/100 : Loss : 1.0017834901809692
Step size 2.5, Iteration 50/100 : Loss : 1.001328468322754
Step size 2.5, Iteration 60/100 : Loss : 1.0010470151901245
Step size 2.5, Iteration 70/100 : Loss : 1.0008561611175537
Step size 2.5, Iteration 80/100 : Loss : 1.000718116760254
Step size 2.5, Iteration 90/100 : Loss : 1.0006133317947388
Step size 2, Iteration 0/100 : Loss : 1.6657466888427734
Step size 2, Iteration 10/100 : Loss : 1.0084993839263916
Step size 2, Iteration 20/100 : Loss : 1.0034843683242798
Step size 2, Iteration 30/100 : Loss : 1.0018739700317383
Step size 2, Iteration 40/100 : Loss : 1.0010333061218262
Step size 2, Iteration 50/100 : Loss : 1.0004676580429077
Step size 2, Iteration 60/100 : Loss : 1.0000131130218506
Step size 2, Iteration 70/100 : Loss : 0.9995934963226318
Step size 2, Iteration 80/100 : Loss : 0.999162495136261
Step size 2, Iteration 90/100 : Loss : 0.9986860752105713
Step size 1.5, Iteration 0/100 : Loss : 1.8677486181259155
Step size 1.5, Iteration 10/100 : Loss : 0.9994000196456909
Step size 1.5, Iteration 20/100 : Loss : 0.9904266595840454
Step size 1.5, Iteration 30/100 : Loss : 0.9867252111434937
Step size 1.5, Iteration 40/100 : Loss : 0.9839764833450317
Step size 1.5, Iteration 50/100 : Loss : 0.9815990328788757
Step size 1.5, Iteration 60/100 : Loss : 0.9794658422470093
Step size 1.5, Iteration 70/100 : Loss : 0.9775331020355225
Step size 1.5, Iteration 80/100 : Loss : 0.9757710099220276
Step size 1.5, Iteration 90/100 : Loss : 0.9741526246070862
Step size 1, Iteration 0/100 : Loss : 2.648204803466797
Step size 1, Iteration 10/100 : Loss : 1.040448784828186
Step size 1, Iteration 20/100 : Loss : 1.009045124053955
Step size 1, Iteration 30/100 : Loss : 1.0028530359268188
Step size 1, Iteration 40/100 : Loss : 1.0002825260162354
Step size 1, Iteration 50/100 : Loss : 0.9987536072731018
Step size 1, Iteration 60/100 : Loss : 0.9976173043251038
Step size 1, Iteration 70/100 : Loss : 0.9966363906860352
Step size 1, Iteration 80/100 : Loss : 0.9956998825073242
Step size 1, Iteration 90/100 : Loss : 0.9947459697723389
Step size 0.5, Iteration 0/100 : Loss : 1.3921946287155151
Step size 0.5, Iteration 10/100 : Loss : 1.0496180057525635
Step size 0.5, Iteration 20/100 : Loss : 1.0191868543624878
Step size 0.5, Iteration 30/100 : Loss : 1.0101796388626099
Step size 0.5, Iteration 40/100 : Loss : 1.00595223903656
Step size 0.5, Iteration 50/100 : Loss : 1.0034854412078857
Step size 0.5, Iteration 60/100 : Loss : 1.0018523931503296
Step size 0.5, Iteration 70/100 : Loss : 1.000676155090332
Step size 0.5, Iteration 80/100 : Loss : 0.9997738599777222
Step size 0.5, Iteration 90/100 : Loss : 0.9990465641021729
```



- The expected gradient descent convergence rate for **non-convex functions** is $O\left(\frac{1}{\sqrt{k}}\right)$, where k is the number of iterations. And we can observe this convergence rate empirically on the previous plot for the 1.5 step size.
- Since it is a non convexe optimization problem we don't have a defined best constant stepsize, but experimentally I found that a good constant stepsize seems to be around 1.5.

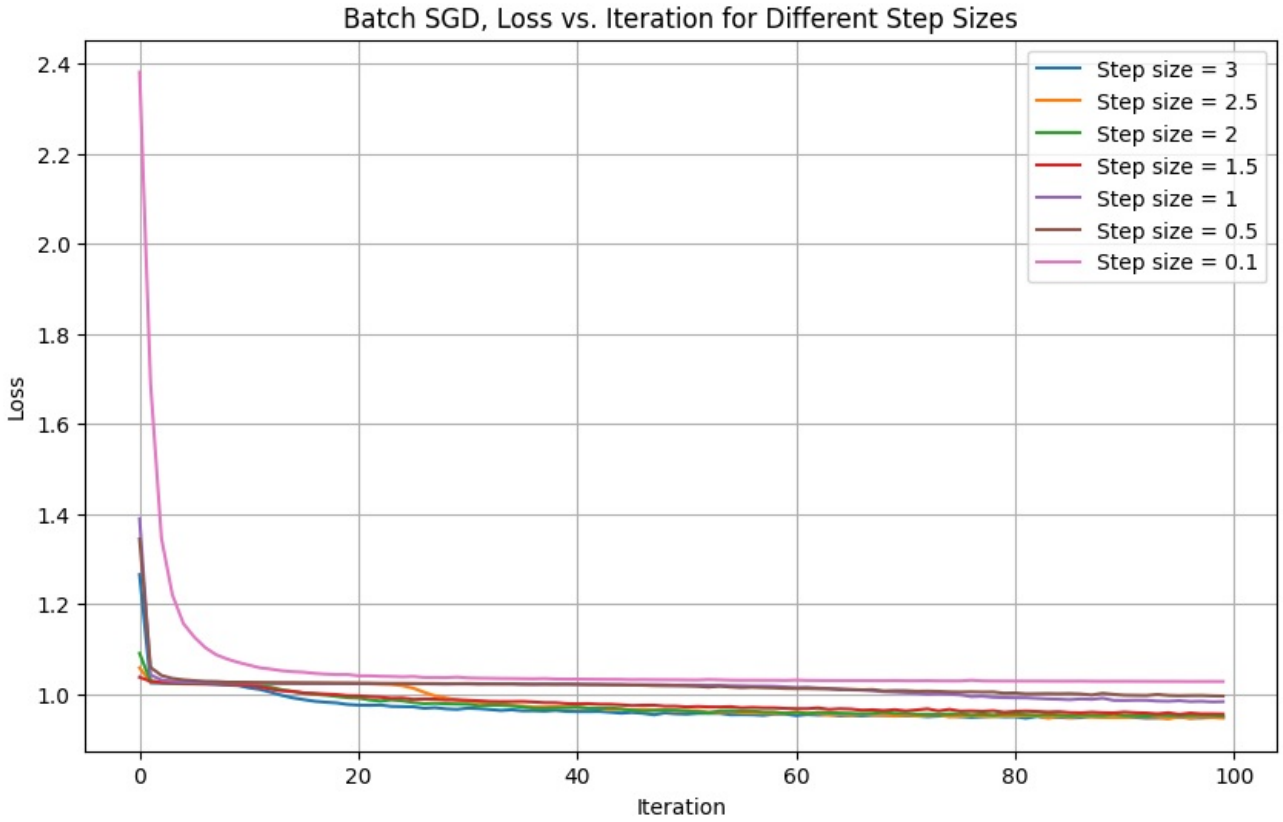
Question 4

Implementation of the Batch Stochastic Gradient Descent with various constant step sizes.

In [ ]:
```python
# Batch SGD

batch_size = 64
nb_iters = 100
step_sizes = [3, 2.5, 2, 1.5, 1, 0.5, 0.1]

losses_dict = {stepsize: [] for stepsize in step_sizes}

for stepsize in step_sizes:
    x = torch.randn(A.size(1), requires_grad=True, device=device)
    for i in range(nb_iters):
        # Shuffle data
        indices = torch.randperm(A.size(0))
        A_shuffled = A[indices]
        y_shuffled = y[indices]

        total_loss = 0
        for b in range(0, A.size(0), batch_size):

            A_batch = A_shuffled[b:b+batch_size]
            y_batch = y_shuffled[b:b+batch_size]

            loss = f(x, A_batch, y_batch)
            total_loss += loss.item()
            loss.backward()

            with torch.no_grad():
                x -= stepsize * x.grad
                x.grad.zero_()

        # Average loss over all batches
        avg_loss = total_loss / (A.size(0) / batch_size)
        losses_dict[stepsize].append(avg_loss)

        if i % 10 == 0:
            print(f"Step size {stepsize}, Iteration {i}/{nb_iters} : Loss : {avg_loss}")

# Plot the losses for each step size
```

```python
plt.figure(figsize=(10, 6))
for stepsize, losses in losses_dict.items():
    plt.plot(range(nb_iters ), losses, label=f"Step size = {stepsize}")

plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Batch SGD, Loss vs. Iteration for Different Step Sizes")
plt.legend()
plt.grid()
plt.show()
```

```
Step size 3, Iteration 0/100 : Loss : 1.2649269065856934
Step size 3, Iteration 10/100 : Loss : 1.0134881896972656
Step size 3, Iteration 20/100 : Loss : 0.975144058227539
Step size 3, Iteration 30/100 : Loss : 0.9683539619445801
Step size 3, Iteration 40/100 : Loss : 0.9608267745971679
Step size 3, Iteration 50/100 : Loss : 0.9551635475158692
Step size 3, Iteration 60/100 : Loss : 0.9518445014953614
Step size 3, Iteration 70/100 : Loss : 0.9516175842285156
Step size 3, Iteration 80/100 : Loss : 0.9494062309265137
Step size 3, Iteration 90/100 : Loss : 0.9496779556274414
Step size 2.5, Iteration 0/100 : Loss : 1.0578110122680664
Step size 2.5, Iteration 10/100 : Loss : 1.0240807609558105
Step size 2.5, Iteration 20/100 : Loss : 1.0229849815368652
Step size 2.5, Iteration 30/100 : Loss : 0.9830973434448242
Step size 2.5, Iteration 40/100 : Loss : 0.9703367347717285
Step size 2.5, Iteration 50/100 : Loss : 0.9628512611389161
Step size 2.5, Iteration 60/100 : Loss : 0.957918155670166
Step size 2.5, Iteration 70/100 : Loss : 0.9530532875061035
Step size 2.5, Iteration 80/100 : Loss : 0.9510421562194824
Step size 2.5, Iteration 90/100 : Loss : 0.9476250877380371
Step size 2, Iteration 0/100 : Loss : 1.0899366149902343
Step size 2, Iteration 10/100 : Loss : 1.019552101135254
Step size 2, Iteration 20/100 : Loss : 0.9908539009094238
Step size 2, Iteration 30/100 : Loss : 0.9770147171020508
Step size 2, Iteration 40/100 : Loss : 0.9706209144592285
Step size 2, Iteration 50/100 : Loss : 0.9616864585876465
Step size 2, Iteration 60/100 : Loss : 0.9597345886230468
Step size 2, Iteration 70/100 : Loss : 0.9568300590515136
Step size 2, Iteration 80/100 : Loss : 0.9551200523376465
Step size 2, Iteration 90/100 : Loss : 0.9522305030822754
Step size 1.5, Iteration 0/100 : Loss : 1.0371061477661132
Step size 1.5, Iteration 10/100 : Loss : 1.018639747619629
Step size 1.5, Iteration 20/100 : Loss : 0.9957461242675781
Step size 1.5, Iteration 30/100 : Loss : 0.9858564071655274
Step size 1.5, Iteration 40/100 : Loss : 0.9779864883422852
Step size 1.5, Iteration 50/100 : Loss : 0.9711552276611328
Step size 1.5, Iteration 60/100 : Loss : 0.9672134742736817
Step size 1.5, Iteration 70/100 : Loss : 0.962116283416748
Step size 1.5, Iteration 80/100 : Loss : 0.9610638618469238
Step size 1.5, Iteration 90/100 : Loss : 0.9596674156188965
Step size 1, Iteration 0/100 : Loss : 1.3889301986694336
Step size 1, Iteration 10/100 : Loss : 1.0241097373962402
Step size 1, Iteration 20/100 : Loss : 1.0231210289001464
Step size 1, Iteration 30/100 : Loss : 1.0222467613220214
Step size 1, Iteration 40/100 : Loss : 1.0212396545410156
Step size 1, Iteration 50/100 : Loss : 1.0195847053527831
Step size 1, Iteration 60/100 : Loss : 1.0141487159729003
Step size 1, Iteration 70/100 : Loss : 1.0026125564575195
Step size 1, Iteration 80/100 : Loss : 0.9914966049194336
Step size 1, Iteration 90/100 : Loss : 0.9854697341918945
Step size 0.5, Iteration 0/100 : Loss : 1.34432071685791
Step size 0.5, Iteration 10/100 : Loss : 1.0254533615112305
Step size 0.5, Iteration 20/100 : Loss : 1.0236455383300782
Step size 0.5, Iteration 30/100 : Loss : 1.023055995941162
Step size 0.5, Iteration 40/100 : Loss : 1.0213259201049805
Step size 0.5, Iteration 50/100 : Loss : 1.017549388885498
Step size 0.5, Iteration 60/100 : Loss : 1.0117757301330566
Step size 0.5, Iteration 70/100 : Loss : 1.007146053314209
Step size 0.5, Iteration 80/100 : Loss : 1.0021842994689942
Step size 0.5, Iteration 90/100 : Loss : 0.9971814270019531
Step size 0.1, Iteration 0/100 : Loss : 2.3806184692382812
Step size 0.1, Iteration 10/100 : Loss : 1.063932373046875
Step size 0.1, Iteration 20/100 : Loss : 1.040199867248535
Step size 0.1, Iteration 30/100 : Loss : 1.0355830497741698
Step size 0.1, Iteration 40/100 : Loss : 1.0327998733520507
Step size 0.1, Iteration 50/100 : Loss : 1.0310940589904785
Step size 0.1, Iteration 60/100 : Loss : 1.0308299407958985
Step size 0.1, Iteration 70/100 : Loss : 1.0291638298034669
Step size 0.1, Iteration 80/100 : Loss : 1.0284778213500976
Step size 0.1, Iteration 90/100 : Loss : 1.027939769744873
```



- In theory, Batch SGD has faster convergence per epoch because we update each parameter multiple times. The results confirm this theory, as the loss evolution is more consistent and lower after 100 epochs for the best constant step sizes (around 2.5).
- The best constant step size seems to be around 2.5 and the best batch size for this step size is 16, as we can see in the next graph computed.

In [227…

```
# Batch SGD

batch_sizes = [16, 32, 64, 128]  # Test these batch sizes
step_size = 2.5  # Since we found it was the best step size
epochs = 100  # Number of iterations

losses_dict = {batch_size: [] for batch_size in batch_sizes}

# Batch SGD with constant step size
for batch_size in batch_sizes:
    x = torch.randn(A.size(1), requires_grad=True, device=device)
    for epoch in range(epochs):
        # Shuffle data
        indices = torch.randperm(A.size(0))
        A_shuffled = A[indices]
        y_shuffled = y[indices]

        total_loss = 0
        for b in range(0, A.size(0), batch_size):
            A_batch = A_shuffled[b:b+batch_size]
            y_batch = y_shuffled[b:b+batch_size]

            loss = f(x, A_batch, y_batch)
            total_loss += loss.item()

            loss.backward()

            with torch.no_grad():
                x -= step_size * x.grad
                x.grad.zero_()

        # Average loss over all batches
        avg_loss = total_loss / (A.size(0) / batch_size)
        losses_dict[batch_size].append(avg_loss)

        print(f"Batch size {batch_size}, Epochs {epoch}/{epochs} : Loss : {avg_loss}")
```

```python
# Plot the losses for each batch size
plt.figure(figsize=(10, 6))
for batch_size, losses in losses_dict.items():
    plt.plot(range(nb_iters), losses, label=f"Batch size = {batch_size}")

plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title(f"Batch SGD, Loss vs. Iteration for Different Batch Sizes (Step Size = {step_size})")
plt.legend()
plt.grid()
plt.show()
```

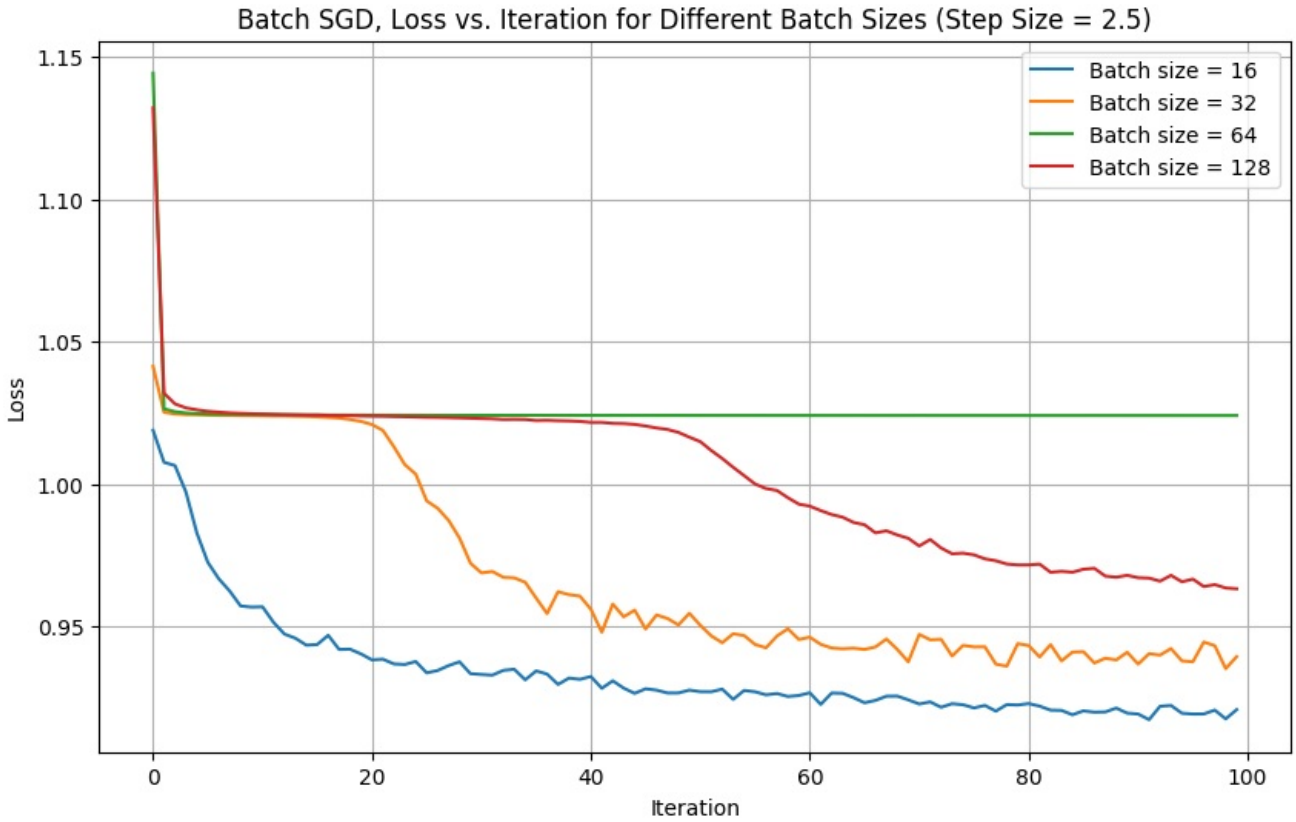
```
Batch size 16, Epochs 0/100 : Loss : 1.0188475141525268
Batch size 16, Epochs 1/100 : Loss : 1.0075865840911866
Batch size 16, Epochs 2/100 : Loss : 1.006454257965088
Batch size 16, Epochs 3/100 : Loss : 0.997158332824707
Batch size 16, Epochs 4/100 : Loss : 0.9827456626892089
Batch size 16, Epochs 5/100 : Loss : 0.9724474220275879
Batch size 16, Epochs 6/100 : Loss : 0.9666523160934448
Batch size 16, Epochs 7/100 : Loss : 0.9623333139419555
Batch size 16, Epochs 8/100 : Loss : 0.9570609474182129
Batch size 16, Epochs 9/100 : Loss : 0.9566926946640014
Batch size 16, Epochs 10/100 : Loss : 0.956782205581665
Batch size 16, Epochs 11/100 : Loss : 0.9513599920272827
Batch size 16, Epochs 12/100 : Loss : 0.9472079372406006
Batch size 16, Epochs 13/100 : Loss : 0.945741093635559
Batch size 16, Epochs 14/100 : Loss : 0.9433106193542481
Batch size 16, Epochs 15/100 : Loss : 0.943521255493164
Batch size 16, Epochs 16/100 : Loss : 0.9467741546630859
Batch size 16, Epochs 17/100 : Loss : 0.9417941637039184
Batch size 16, Epochs 18/100 : Loss : 0.9418806743621826
Batch size 16, Epochs 19/100 : Loss : 0.9402154712677002
Batch size 16, Epochs 20/100 : Loss : 0.9381232786178589
Batch size 16, Epochs 21/100 : Loss : 0.93836270236969
Batch size 16, Epochs 22/100 : Loss : 0.9367076187133789
Batch size 16, Epochs 23/100 : Loss : 0.93650719165802
Batch size 16, Epochs 24/100 : Loss : 0.9375499668121338
Batch size 16, Epochs 25/100 : Loss : 0.933574122428894
Batch size 16, Epochs 26/100 : Loss : 0.9343811540603638
Batch size 16, Epochs 27/100 : Loss : 0.9361155300140381
Batch size 16, Epochs 28/100 : Loss : 0.9374470767974854
Batch size 16, Epochs 29/100 : Loss : 0.9332797203063965
Batch size 16, Epochs 30/100 : Loss : 0.9330026845932007
Batch size 16, Epochs 31/100 : Loss : 0.9327610960006714
Batch size 16, Epochs 32/100 : Loss : 0.9344275102615357
Batch size 16, Epochs 33/100 : Loss : 0.9348259258270264
Batch size 16, Epochs 34/100 : Loss : 0.9311063480377197
Batch size 16, Epochs 35/100 : Loss : 0.9342412881851196
Batch size 16, Epochs 36/100 : Loss : 0.933141996383667
Batch size 16, Epochs 37/100 : Loss : 0.9294971256256104
Batch size 16, Epochs 38/100 : Loss : 0.9317239456176758
Batch size 16, Epochs 39/100 : Loss : 0.931293254852295
Batch size 16, Epochs 40/100 : Loss : 0.9322716655731201
Batch size 16, Epochs 41/100 : Loss : 0.9281302003860473
Batch size 16, Epochs 42/100 : Loss : 0.9307605018615722
Batch size 16, Epochs 43/100 : Loss : 0.9281887254714966
Batch size 16, Epochs 44/100 : Loss : 0.9263521718978882
Batch size 16, Epochs 45/100 : Loss : 0.9279547691345215
Batch size 16, Epochs 46/100 : Loss : 0.9274869756698608
Batch size 16, Epochs 47/100 : Loss : 0.926491280555725
Batch size 16, Epochs 48/100 : Loss : 0.9264987125396729
Batch size 16, Epochs 49/100 : Loss : 0.9274974098205566
Batch size 16, Epochs 50/100 : Loss : 0.9269435958862304
Batch size 16, Epochs 51/100 : Loss : 0.9269292297363281
Batch size 16, Epochs 52/100 : Loss : 0.9278326616287231
Batch size 16, Epochs 53/100 : Loss : 0.9242432346343994
Batch size 16, Epochs 54/100 : Loss : 0.9273742303848267
Batch size 16, Epochs 55/100 : Loss : 0.9269096574783325
Batch size 16, Epochs 56/100 : Loss : 0.9258460378646851
Batch size 16, Epochs 57/100 : Loss : 0.9262973136901855
Batch size 16, Epochs 58/100 : Loss : 0.9252542705535889
Batch size 16, Epochs 59/100 : Loss : 0.9256051912307739
Batch size 16, Epochs 60/100 : Loss : 0.9265588665008545
Batch size 16, Epochs 61/100 : Loss : 0.9224718790054321
Batch size 16, Epochs 62/100 : Loss : 0.9264649362564087
Batch size 16, Epochs 63/100 : Loss : 0.9263658876419067
Batch size 16, Epochs 64/100 : Loss : 0.9248521947860717
Batch size 16, Epochs 65/100 : Loss : 0.9230485172271728
Batch size 16, Epochs 66/100 : Loss : 0.9238869800567627
Batch size 16, Epochs 67/100 : Loss : 0.9253476276397705
Batch size 16, Epochs 68/100 : Loss : 0.9253833894729614
Batch size 16, Epochs 69/100 : Loss : 0.9240663890838623
Batch size 16, Epochs 70/100 : Loss : 0.9226158208847046
Batch size 16, Epochs 71/100 : Loss : 0.9233516206741333
Batch size 16, Epochs 72/100 : Loss : 0.9214580678939819
Batch size 16, Epochs 73/100 : Loss : 0.9227181510925293
Batch size 16, Epochs 74/100 : Loss : 0.9223585224151611
Batch size 16, Epochs 75/100 : Loss : 0.9212378149032593
```

```
Batch size 16, Epochs 76/100 : Loss : 0.9221101999282837
Batch size 16, Epochs 77/100 : Loss : 0.9201105613708496
Batch size 16, Epochs 78/100 : Loss : 0.9224074954986572
Batch size 16, Epochs 79/100 : Loss : 0.9222712993621827
Batch size 16, Epochs 80/100 : Loss : 0.9227609405517578
Batch size 16, Epochs 81/100 : Loss : 0.9218554630279541
Batch size 16, Epochs 82/100 : Loss : 0.9204534120559692
Batch size 16, Epochs 83/100 : Loss : 0.9203589153289795
Batch size 16, Epochs 84/100 : Loss : 0.9188340282440186
Batch size 16, Epochs 85/100 : Loss : 0.9201990728378295
Batch size 16, Epochs 86/100 : Loss : 0.9197371616363526
Batch size 16, Epochs 87/100 : Loss : 0.9198191089630127
Batch size 16, Epochs 88/100 : Loss : 0.9211755714416504
Batch size 16, Epochs 89/100 : Loss : 0.9193979597091675
Batch size 16, Epochs 90/100 : Loss : 0.9190873098373413
Batch size 16, Epochs 91/100 : Loss : 0.9171087436676025
Batch size 16, Epochs 92/100 : Loss : 0.9218139581680298
Batch size 16, Epochs 93/100 : Loss : 0.9221047220230103
Batch size 16, Epochs 94/100 : Loss : 0.9193604879379272
Batch size 16, Epochs 95/100 : Loss : 0.9190868911743164
Batch size 16, Epochs 96/100 : Loss : 0.9191084938049316
Batch size 16, Epochs 97/100 : Loss : 0.9204706649780273
Batch size 16, Epochs 98/100 : Loss : 0.9173860721588135
Batch size 16, Epochs 99/100 : Loss : 0.9206308956146241
Batch size 32, Epochs 0/100 : Loss : 1.0413475952148437
Batch size 32, Epochs 1/100 : Loss : 1.0251579036712646
Batch size 32, Epochs 2/100 : Loss : 1.0245818920135499
Batch size 32, Epochs 3/100 : Loss : 1.0243627128601074
Batch size 32, Epochs 4/100 : Loss : 1.0242928466796875
Batch size 32, Epochs 5/100 : Loss : 1.0241632595062256
Batch size 32, Epochs 6/100 : Loss : 1.0240864791870117
Batch size 32, Epochs 7/100 : Loss : 1.0240767250061036
Batch size 32, Epochs 8/100 : Loss : 1.024015438079834
Batch size 32, Epochs 9/100 : Loss : 1.023988977432251
Batch size 32, Epochs 10/100 : Loss : 1.023947063446045
Batch size 32, Epochs 11/100 : Loss : 1.0238836002349854
Batch size 32, Epochs 12/100 : Loss : 1.0238738994598389
Batch size 32, Epochs 13/100 : Loss : 1.0237787132263183
Batch size 32, Epochs 14/100 : Loss : 1.0237072677612304
Batch size 32, Epochs 15/100 : Loss : 1.023562126159668
Batch size 32, Epochs 16/100 : Loss : 1.0233699893951416
Batch size 32, Epochs 17/100 : Loss : 1.023120578765869
Batch size 32, Epochs 18/100 : Loss : 1.0226289329528808
Batch size 32, Epochs 19/100 : Loss : 1.0219651126861573
Batch size 32, Epochs 20/100 : Loss : 1.0208646450042724
Batch size 32, Epochs 21/100 : Loss : 1.018790594100952
Batch size 32, Epochs 22/100 : Loss : 1.0130590534210204
Batch size 32, Epochs 23/100 : Loss : 1.0067885093688964
Batch size 32, Epochs 24/100 : Loss : 1.0034127502441406
Batch size 32, Epochs 25/100 : Loss : 0.9940190696716309
Batch size 32, Epochs 26/100 : Loss : 0.9914297943115234
Batch size 32, Epochs 27/100 : Loss : 0.9871828651428223
Batch size 32, Epochs 28/100 : Loss : 0.9809648513793945
Batch size 32, Epochs 29/100 : Loss : 0.972102611541748
Batch size 32, Epochs 30/100 : Loss : 0.9687075977325439
Batch size 32, Epochs 31/100 : Loss : 0.969197717666626
Batch size 32, Epochs 32/100 : Loss : 0.9671600875854492
Batch size 32, Epochs 33/100 : Loss : 0.9669552898406982
Batch size 32, Epochs 34/100 : Loss : 0.9654251766204834
Batch size 32, Epochs 35/100 : Loss : 0.959856819152832
Batch size 32, Epochs 36/100 : Loss : 0.9544178638458252
Batch size 32, Epochs 37/100 : Loss : 0.9620879344940185
Batch size 32, Epochs 38/100 : Loss : 0.9610588283538818
Batch size 32, Epochs 39/100 : Loss : 0.9605533237457275
Batch size 32, Epochs 40/100 : Loss : 0.9559364128112793
Batch size 32, Epochs 41/100 : Loss : 0.9478519210815429
Batch size 32, Epochs 42/100 : Loss : 0.9577526588439942
Batch size 32, Epochs 43/100 : Loss : 0.953268196105957
Batch size 32, Epochs 44/100 : Loss : 0.9556118106842041
Batch size 32, Epochs 45/100 : Loss : 0.9490016784667968
Batch size 32, Epochs 46/100 : Loss : 0.9539249801635742
Batch size 32, Epochs 47/100 : Loss : 0.9526771144866943
Batch size 32, Epochs 48/100 : Loss : 0.9504147052764893
Batch size 32, Epochs 49/100 : Loss : 0.9545130977630615
Batch size 32, Epochs 50/100 : Loss : 0.950258752822876
Batch size 32, Epochs 51/100 : Loss : 0.9465669479370117
Batch size 32, Epochs 52/100 : Loss : 0.944152006149292
Batch size 32, Epochs 53/100 : Loss : 0.9473056468963623
Batch size 32, Epochs 54/100 : Loss : 0.9466806087493896
Batch size 32, Epochs 55/100 : Loss : 0.9435538120269775
Batch size 32, Epochs 56/100 : Loss : 0.9423643188476563
Batch size 32, Epochs 57/100 : Loss : 0.9465098819732666
Batch size 32, Epochs 58/100 : Loss : 0.9490618858337402
Batch size 32, Epochs 59/100 : Loss : 0.945294734954834
Batch size 32, Epochs 60/100 : Loss : 0.9461084156036377
Batch size 32, Epochs 61/100 : Loss : 0.9435831489562988
Batch size 32, Epochs 62/100 : Loss : 0.9423033485412597
Batch size 32, Epochs 63/100 : Loss : 0.9420302715301514
Batch size 32, Epochs 64/100 : Loss : 0.9422357311248779
```

```
Batch size 32, Epochs 65/100 : Loss : 0.9417900772094726
Batch size 32, Epochs 66/100 : Loss : 0.9426698513031005
Batch size 32, Epochs 67/100 : Loss : 0.9454519290924073
Batch size 32, Epochs 68/100 : Loss : 0.9418005523681641
Batch size 32, Epochs 69/100 : Loss : 0.9374314498901367
Batch size 32, Epochs 70/100 : Loss : 0.9470857448577881
Batch size 32, Epochs 71/100 : Loss : 0.9451932697296143
Batch size 32, Epochs 72/100 : Loss : 0.9453545684814453
Batch size 32, Epochs 73/100 : Loss : 0.939466739654541
Batch size 32, Epochs 74/100 : Loss : 0.9432157649993896
Batch size 32, Epochs 75/100 : Loss : 0.9426992835998536
Batch size 32, Epochs 76/100 : Loss : 0.9427151851654053
Batch size 32, Epochs 77/100 : Loss : 0.9366458015441894
Batch size 32, Epochs 78/100 : Loss : 0.9359115066528321
Batch size 32, Epochs 79/100 : Loss : 0.9438346748352051
Batch size 32, Epochs 80/100 : Loss : 0.9431691532135009
Batch size 32, Epochs 81/100 : Loss : 0.9391258106231689
Batch size 32, Epochs 82/100 : Loss : 0.9434789600372314
Batch size 32, Epochs 83/100 : Loss : 0.9377711429595947
Batch size 32, Epochs 84/100 : Loss : 0.9408585243225097
Batch size 32, Epochs 85/100 : Loss : 0.9410051612854003
Batch size 32, Epochs 86/100 : Loss : 0.937026704788208
Batch size 32, Epochs 87/100 : Loss : 0.9387033939361572
Batch size 32, Epochs 88/100 : Loss : 0.9381076717376708
Batch size 32, Epochs 89/100 : Loss : 0.9408001232147217
Batch size 32, Epochs 90/100 : Loss : 0.9366483249664307
Batch size 32, Epochs 91/100 : Loss : 0.9403078060150146
Batch size 32, Epochs 92/100 : Loss : 0.9398123111724853
Batch size 32, Epochs 93/100 : Loss : 0.9420973682403564
Batch size 32, Epochs 94/100 : Loss : 0.9377630805969238
Batch size 32, Epochs 95/100 : Loss : 0.9374639854431153
Batch size 32, Epochs 96/100 : Loss : 0.9443856067657471
Batch size 32, Epochs 97/100 : Loss : 0.9431266250610352
Batch size 32, Epochs 98/100 : Loss : 0.9350453128814697
Batch size 32, Epochs 99/100 : Loss : 0.9392665195465087
Batch size 64, Epochs 0/100 : Loss : 1.144400894165039
Batch size 64, Epochs 1/100 : Loss : 1.0263893432617188
Batch size 64, Epochs 2/100 : Loss : 1.0253930892944336
Batch size 64, Epochs 3/100 : Loss : 1.0249359741210937
Batch size 64, Epochs 4/100 : Loss : 1.024729736328125
Batch size 64, Epochs 5/100 : Loss : 1.0245827407836914
Batch size 64, Epochs 6/100 : Loss : 1.0245005569458008
Batch size 64, Epochs 7/100 : Loss : 1.0244314727783204
Batch size 64, Epochs 8/100 : Loss : 1.0243726196289062
Batch size 64, Epochs 9/100 : Loss : 1.0243295555114746
Batch size 64, Epochs 10/100 : Loss : 1.024323501586914
Batch size 64, Epochs 11/100 : Loss : 1.024294937133789
Batch size 64, Epochs 12/100 : Loss : 1.0242535820007324
Batch size 64, Epochs 13/100 : Loss : 1.0242403907775879
Batch size 64, Epochs 14/100 : Loss : 1.0242193641662598
Batch size 64, Epochs 15/100 : Loss : 1.0242081756591797
Batch size 64, Epochs 16/100 : Loss : 1.024188488006592
Batch size 64, Epochs 17/100 : Loss : 1.0241831893920899
Batch size 64, Epochs 18/100 : Loss : 1.0241724548339843
Batch size 64, Epochs 19/100 : Loss : 1.0241615028381348
Batch size 64, Epochs 20/100 : Loss : 1.0241567001342773
Batch size 64, Epochs 21/100 : Loss : 1.0241451683044434
Batch size 64, Epochs 22/100 : Loss : 1.0241409301757813
Batch size 64, Epochs 23/100 : Loss : 1.0241305961608886
Batch size 64, Epochs 24/100 : Loss : 1.0241337394714356
Batch size 64, Epochs 25/100 : Loss : 1.0241226348876953
Batch size 64, Epochs 26/100 : Loss : 1.0241197776794433
Batch size 64, Epochs 27/100 : Loss : 1.0241126098632813
Batch size 64, Epochs 28/100 : Loss : 1.0241114196777343
Batch size 64, Epochs 29/100 : Loss : 1.0241051597595214
Batch size 64, Epochs 30/100 : Loss : 1.0241036338806153
Batch size 64, Epochs 31/100 : Loss : 1.0241013717651368
Batch size 64, Epochs 32/100 : Loss : 1.0240982818603515
Batch size 64, Epochs 33/100 : Loss : 1.0240928535461427
Batch size 64, Epochs 34/100 : Loss : 1.0240890998840333
Batch size 64, Epochs 35/100 : Loss : 1.0240916595458984
Batch size 64, Epochs 36/100 : Loss : 1.0240826988220215
Batch size 64, Epochs 37/100 : Loss : 1.0240809936523438
Batch size 64, Epochs 38/100 : Loss : 1.024076488494873
Batch size 64, Epochs 39/100 : Loss : 1.0240768699645997
Batch size 64, Epochs 40/100 : Loss : 1.0240748748779296
Batch size 64, Epochs 41/100 : Loss : 1.0240753173828125
Batch size 64, Epochs 42/100 : Loss : 1.0240730056762695
Batch size 64, Epochs 43/100 : Loss : 1.0240693778991699
Batch size 64, Epochs 44/100 : Loss : 1.0240715370178222
Batch size 64, Epochs 45/100 : Loss : 1.0240663986206056
Batch size 64, Epochs 46/100 : Loss : 1.024064926147461
Batch size 64, Epochs 47/100 : Loss : 1.0240668411254883
Batch size 64, Epochs 48/100 : Loss : 1.0240650939941407
Batch size 64, Epochs 49/100 : Loss : 1.0240618629455567
Batch size 64, Epochs 50/100 : Loss : 1.02405912399292
Batch size 64, Epochs 51/100 : Loss : 1.0240601539611816
Batch size 64, Epochs 52/100 : Loss : 1.024056728363037
Batch size 64, Epochs 53/100 : Loss : 1.0240541877746583
```

```
Batch size 64, Epochs 54/100 : Loss : 1.024053726196289
Batch size 64, Epochs 55/100 : Loss : 1.0240551834106446
Batch size 64, Epochs 56/100 : Loss : 1.0240519866943358
Batch size 64, Epochs 57/100 : Loss : 1.0240510787963868
Batch size 64, Epochs 58/100 : Loss : 1.0240506820678712
Batch size 64, Epochs 59/100 : Loss : 1.0240493354797364
Batch size 64, Epochs 60/100 : Loss : 1.0240467567443847
Batch size 64, Epochs 61/100 : Loss : 1.0240480117797852
Batch size 64, Epochs 62/100 : Loss : 1.0240477180480958
Batch size 64, Epochs 63/100 : Loss : 1.0240420036315918
Batch size 64, Epochs 64/100 : Loss : 1.024046974182129
Batch size 64, Epochs 65/100 : Loss : 1.0240443153381347
Batch size 64, Epochs 66/100 : Loss : 1.0240424957275391
Batch size 64, Epochs 67/100 : Loss : 1.0240433197021483
Batch size 64, Epochs 68/100 : Loss : 1.024039752960205
Batch size 64, Epochs 69/100 : Loss : 1.0240420875549316
Batch size 64, Epochs 70/100 : Loss : 1.0240457496643067
Batch size 64, Epochs 71/100 : Loss : 1.0240394210815429
Batch size 64, Epochs 72/100 : Loss : 1.0240398406982423
Batch size 64, Epochs 73/100 : Loss : 1.024038600921631
Batch size 64, Epochs 74/100 : Loss : 1.0240377197265624
Batch size 64, Epochs 75/100 : Loss : 1.0240409278869629
Batch size 64, Epochs 76/100 : Loss : 1.0240378494262696
Batch size 64, Epochs 77/100 : Loss : 1.024034526824951
Batch size 64, Epochs 78/100 : Loss : 1.02403462600708
Batch size 64, Epochs 79/100 : Loss : 1.0240391464233398
Batch size 64, Epochs 80/100 : Loss : 1.0240326538085938
Batch size 64, Epochs 81/100 : Loss : 1.02403426361084
Batch size 64, Epochs 82/100 : Loss : 1.0240344848632812
Batch size 64, Epochs 83/100 : Loss : 1.024030200958252
Batch size 64, Epochs 84/100 : Loss : 1.0240312538146972
Batch size 64, Epochs 85/100 : Loss : 1.0240312156677247
Batch size 64, Epochs 86/100 : Loss : 1.0240375366210936
Batch size 64, Epochs 87/100 : Loss : 1.0240318450927735
Batch size 64, Epochs 88/100 : Loss : 1.0240299034118652
Batch size 64, Epochs 89/100 : Loss : 1.0240302734375
Batch size 64, Epochs 90/100 : Loss : 1.0240303955078125
Batch size 64, Epochs 91/100 : Loss : 1.0240300712585448
Batch size 64, Epochs 92/100 : Loss : 1.0240299072265624
Batch size 64, Epochs 93/100 : Loss : 1.0240279846191407
Batch size 64, Epochs 94/100 : Loss : 1.024025966644287
Batch size 64, Epochs 95/100 : Loss : 1.0240277290344237
Batch size 64, Epochs 96/100 : Loss : 1.0240275688171387
Batch size 64, Epochs 97/100 : Loss : 1.0240285568237304
Batch size 64, Epochs 98/100 : Loss : 1.024026954650879
Batch size 64, Epochs 99/100 : Loss : 1.024025619506836
Batch size 128, Epochs 0/100 : Loss : 1.1321980133056642
Batch size 128, Epochs 1/100 : Loss : 1.0318693237304688
Batch size 128, Epochs 2/100 : Loss : 1.0281290435791015
Batch size 128, Epochs 3/100 : Loss : 1.0267247772216797
Batch size 128, Epochs 4/100 : Loss : 1.026037239074707
Batch size 128, Epochs 5/100 : Loss : 1.0255204696655273
Batch size 128, Epochs 6/100 : Loss : 1.0252092056274413
Batch size 128, Epochs 7/100 : Loss : 1.0249520416259765
Batch size 128, Epochs 8/100 : Loss : 1.0247996673583983
Batch size 128, Epochs 9/100 : Loss : 1.0246583251953125
Batch size 128, Epochs 10/100 : Loss : 1.024576789855957
Batch size 128, Epochs 11/100 : Loss : 1.0244595336914062
Batch size 128, Epochs 12/100 : Loss : 1.0243611450195313
Batch size 128, Epochs 13/100 : Loss : 1.0242881164550781
Batch size 128, Epochs 14/100 : Loss : 1.024206672668457
Batch size 128, Epochs 15/100 : Loss : 1.024204330444336
Batch size 128, Epochs 16/100 : Loss : 1.0240781478881835
Batch size 128, Epochs 17/100 : Loss : 1.023958984375
Batch size 128, Epochs 18/100 : Loss : 1.023990119934082
Batch size 128, Epochs 19/100 : Loss : 1.023874008178711
Batch size 128, Epochs 20/100 : Loss : 1.023818145751953
Batch size 128, Epochs 21/100 : Loss : 1.0237725219726563
Batch size 128, Epochs 22/100 : Loss : 1.0236785812377929
Batch size 128, Epochs 23/100 : Loss : 1.0236155471801758
Batch size 128, Epochs 24/100 : Loss : 1.023534523010254
Batch size 128, Epochs 25/100 : Loss : 1.023418342590332
Batch size 128, Epochs 26/100 : Loss : 1.0233709564208984
Batch size 128, Epochs 27/100 : Loss : 1.023294448852539
Batch size 128, Epochs 28/100 : Loss : 1.0231932067871095
Batch size 128, Epochs 29/100 : Loss : 1.0230881042480469
Batch size 128, Epochs 30/100 : Loss : 1.0229760360717775
Batch size 128, Epochs 31/100 : Loss : 1.0228732452392577
Batch size 128, Epochs 32/100 : Loss : 1.0226420669555665
Batch size 128, Epochs 33/100 : Loss : 1.0226821136474609
Batch size 128, Epochs 34/100 : Loss : 1.0226407852172852
Batch size 128, Epochs 35/100 : Loss : 1.0222785034179687
Batch size 128, Epochs 36/100 : Loss : 1.0223433303833007
Batch size 128, Epochs 37/100 : Loss : 1.0222011337280272
Batch size 128, Epochs 38/100 : Loss : 1.0221217880249023
Batch size 128, Epochs 39/100 : Loss : 1.0219649200439453
Batch size 128, Epochs 40/100 : Loss : 1.0216152420043945
Batch size 128, Epochs 41/100 : Loss : 1.0216191177368164
Batch size 128, Epochs 42/100 : Loss : 1.0212737274169923
```

```
Batch size 128, Epochs 43/100 : Loss : 1.021191032409668
Batch size 128, Epochs 44/100 : Loss : 1.0209129333496094
Batch size 128, Epochs 45/100 : Loss : 1.0203397216796875
Batch size 128, Epochs 46/100 : Loss : 1.0196552047729492
Batch size 128, Epochs 47/100 : Loss : 1.0191324310302734
Batch size 128, Epochs 48/100 : Loss : 1.0181022109985352
Batch size 128, Epochs 49/100 : Loss : 1.016414909362793
Batch size 128, Epochs 50/100 : Loss : 1.0148417510986327
Batch size 128, Epochs 51/100 : Loss : 1.0117296447753907
Batch size 128, Epochs 52/100 : Loss : 1.0089281845092772
Batch size 128, Epochs 53/100 : Loss : 1.0058287811279296
Batch size 128, Epochs 54/100 : Loss : 1.0029121780395507
Batch size 128, Epochs 55/100 : Loss : 0.9999563217163085
Batch size 128, Epochs 56/100 : Loss : 0.9983623886108398
Batch size 128, Epochs 57/100 : Loss : 0.9976814422607422
Batch size 128, Epochs 58/100 : Loss : 0.9951349716186524
Batch size 128, Epochs 59/100 : Loss : 0.9928758087158203
Batch size 128, Epochs 60/100 : Loss : 0.9921771850585938
Batch size 128, Epochs 61/100 : Loss : 0.9906088104248046
Batch size 128, Epochs 62/100 : Loss : 0.9892181854248047
Batch size 128, Epochs 63/100 : Loss : 0.9882916259765625
Batch size 128, Epochs 64/100 : Loss : 0.9864367599487305
Batch size 128, Epochs 65/100 : Loss : 0.985632942199707
Batch size 128, Epochs 66/100 : Loss : 0.9827819747924804
Batch size 128, Epochs 67/100 : Loss : 0.9834989471435547
Batch size 128, Epochs 68/100 : Loss : 0.9820356140136719
Batch size 128, Epochs 69/100 : Loss : 0.9808608474731445
Batch size 128, Epochs 70/100 : Loss : 0.9781616058349609
Batch size 128, Epochs 71/100 : Loss : 0.9804493408203125
Batch size 128, Epochs 72/100 : Loss : 0.9774573669433594
Batch size 128, Epochs 73/100 : Loss : 0.975434700012207
Batch size 128, Epochs 74/100 : Loss : 0.9756439514160157
Batch size 128, Epochs 75/100 : Loss : 0.9750560150146484
Batch size 128, Epochs 76/100 : Loss : 0.9736436309814454
Batch size 128, Epochs 77/100 : Loss : 0.9729699630737305
Batch size 128, Epochs 78/100 : Loss : 0.9718226165771484
Batch size 128, Epochs 79/100 : Loss : 0.9715080642700196
Batch size 128, Epochs 80/100 : Loss : 0.9715170822143555
Batch size 128, Epochs 81/100 : Loss : 0.9717886886596679
Batch size 128, Epochs 82/100 : Loss : 0.9689221343994141
Batch size 128, Epochs 83/100 : Loss : 0.9692493896484375
Batch size 128, Epochs 84/100 : Loss : 0.9689401245117187
Batch size 128, Epochs 85/100 : Loss : 0.9699922256469726
Batch size 128, Epochs 86/100 : Loss : 0.9703047561645508
Batch size 128, Epochs 87/100 : Loss : 0.9675798034667968
Batch size 128, Epochs 88/100 : Loss : 0.9671841659545899
Batch size 128, Epochs 89/100 : Loss : 0.9678520431518555
Batch size 128, Epochs 90/100 : Loss : 0.9670132293701171
Batch size 128, Epochs 91/100 : Loss : 0.9668687973022461
Batch size 128, Epochs 92/100 : Loss : 0.9658022918701172
Batch size 128, Epochs 93/100 : Loss : 0.967842300415039
Batch size 128, Epochs 94/100 : Loss : 0.9655851058959961
Batch size 128, Epochs 95/100 : Loss : 0.9665095367431641
Batch size 128, Epochs 96/100 : Loss : 0.9639063186645508
Batch size 128, Epochs 97/100 : Loss : 0.9645827407836914
Batch size 128, Epochs 98/100 : Loss : 0.9634032897949218
Batch size 128, Epochs 99/100 : Loss : 0.9631259384155273
```



Question 5

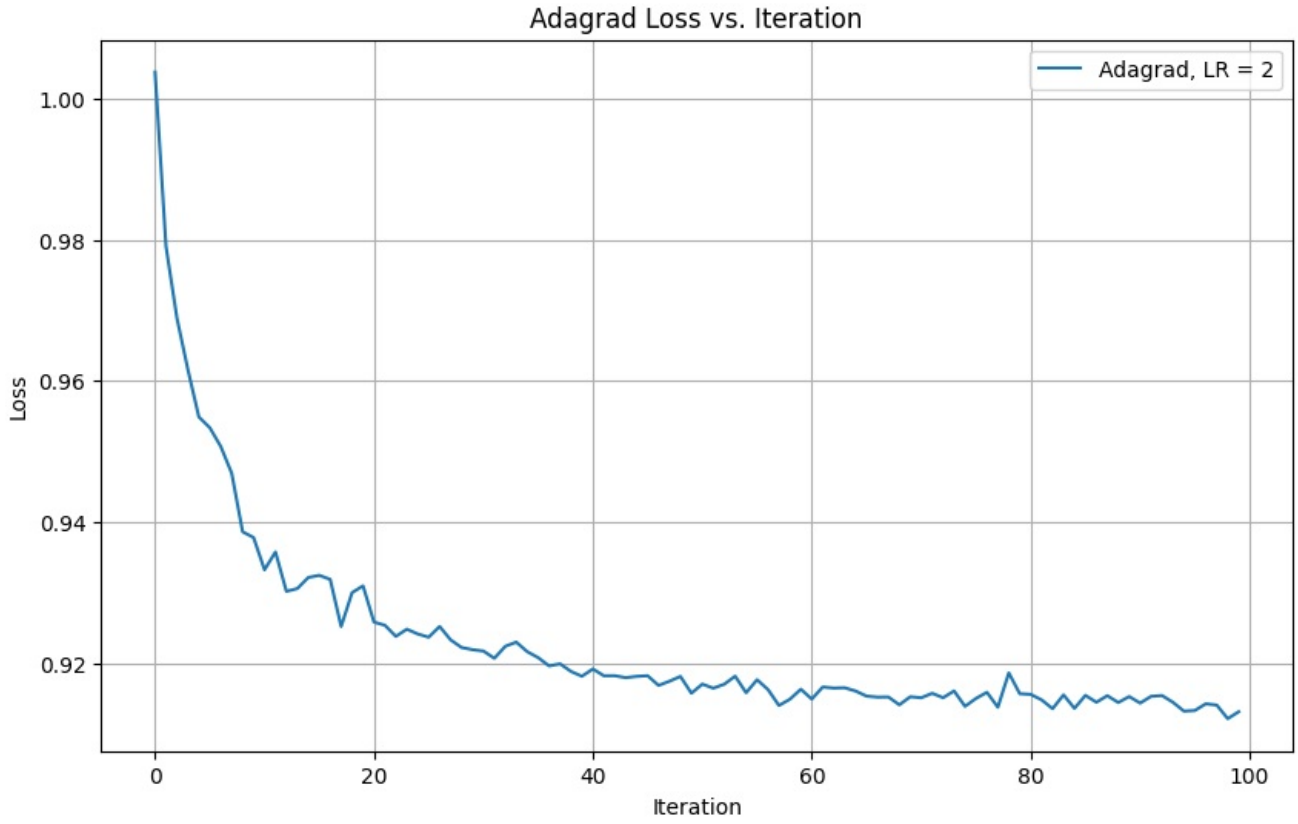
We will now implement Adagrad and compare it with the best result from Batch SGD with batch size of 16 and a learning rate of 2.5.

In [142…

```
# Adagrad implementation

batch_size = 16
nb_iters = 100
lr = 2
epsilon = 1e-8

x = torch.randn(A.size(1), requires_grad=True, device=device)
G = torch.zeros_like(x, device=device)
adagrad_losses = []
for i in range(nb_iters):
    indices = torch.randperm(A.size(0))
    A_shuffled = A[indices]
    y_shuffled = y[indices]
```

```
    total_loss = 0
    for b in range(0, A.size(0), batch_size):
        A_batch = A_shuffled[b:b+batch_size]
        y_batch = y_shuffled[b:b+batch_size]
        loss = f(x, A_batch, y_batch)
        total_loss += loss.item()
        loss.backward()
        G.add_(x.grad.pow(2))
        with torch.no_grad():
            x -= lr / torch.sqrt(G + epsilon) * x.grad
            x.grad.zero_()
    avg_loss = total_loss / (A.size(0) / batch_size)
    adagrad_losses.append(avg_loss)
    if i % 10 == 0:
        print(f"Adagrad, Iteration {i}/{nb_iters} : Loss : {avg_loss}")

# Plot the losses
plt.figure(figsize=(10, 6))
plt.plot(range(nb_iters), adagrad_losses, label=f"Adagrad, LR = {lr}")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Adagrad Loss vs. Iteration")
plt.legend()
plt.grid()
plt.show()
```

```
Adagrad, Iteration 0/100 : Loss : 1.0037726936340332
Adagrad, Iteration 10/100 : Loss : 0.9332298784255981
Adagrad, Iteration 20/100 : Loss : 0.9258254203796387
Adagrad, Iteration 30/100 : Loss : 0.9217158479690551
Adagrad, Iteration 40/100 : Loss : 0.9191737432479858
Adagrad, Iteration 50/100 : Loss : 0.9170286235809326
Adagrad, Iteration 60/100 : Loss : 0.9149073829650879
Adagrad, Iteration 70/100 : Loss : 0.9150953741073609
Adagrad, Iteration 80/100 : Loss : 0.9155781621932984
Adagrad, Iteration 90/100 : Loss : 0.9143404111862182
```



- As expected, the results are slighly better with Adagrad compared to the Batch SGD for the same configuration, but the training is much faster with batch SGD.

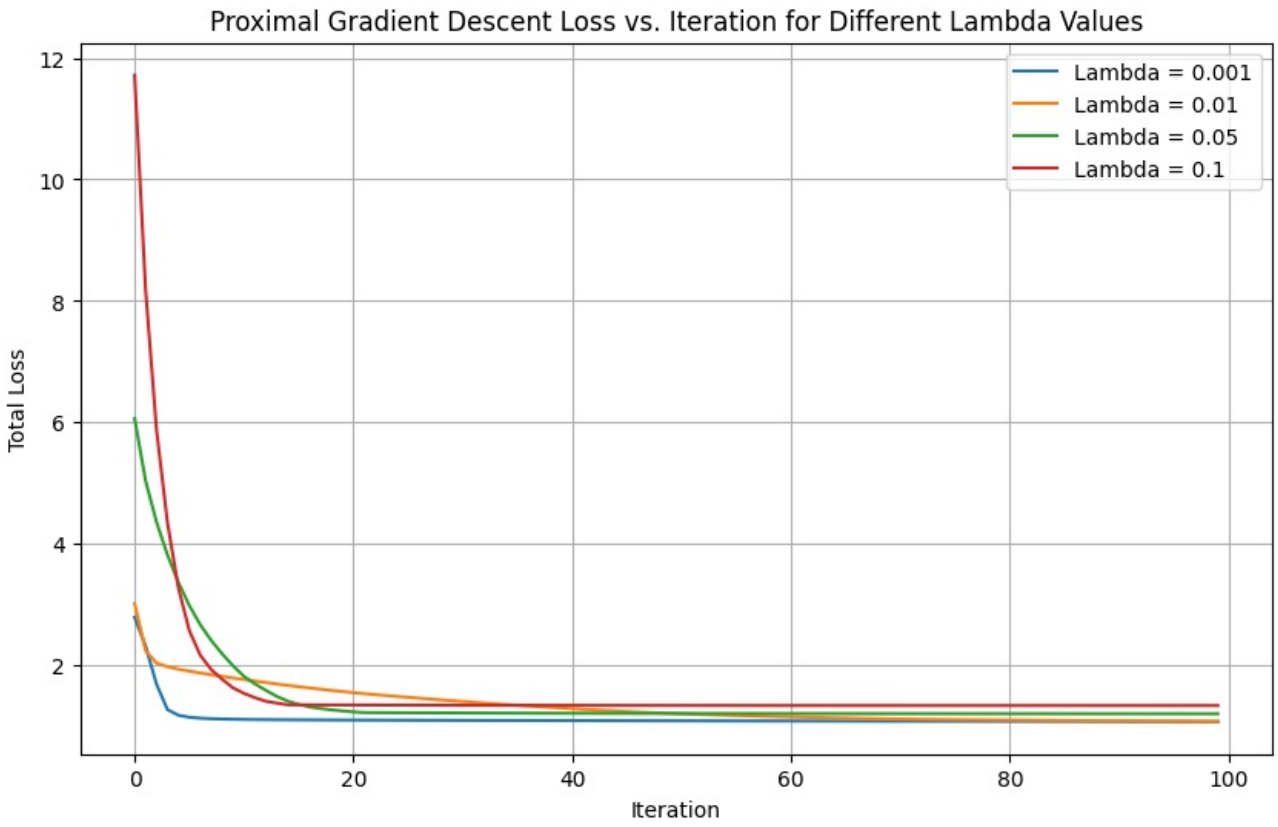
Question 6

We will now implement the proximal gradient approach with the regularized version of the problem. We take 100 iterations and a step size of 2.5 based on previous experiments.

In [ ]:
```
# Proximal Gradient Descent for regularized problem

nb_iters = 100
step_size = 2.5
lambda_values = [0.001, 0.01, 0.05, 0.1]  # Test these lambda values
```

```python
losses_dict = {lambdaa: [] for lambdaa in lambda_values}
sparsity_dict = {lambdaa: 0.0 for lambdaa in lambda_values}

for lambdaa in lambda_values:
    x = torch.randn(A.size(1), requires_grad=True, device=device)

    for i in range(nb_iters):
        loss = f(x, A, y)

        l1_term = lambdaa * torch.norm(x, p=1)
        total_loss = loss + l1_term

        loss.backward()

        with torch.no_grad():
            x_new = x - step_size * x.grad
            x_prox = torch.sign(x_new) * torch.clamp(torch.abs(x_new) - step_size * lambdaa, min=0.0)
            x.copy_(x_prox)
            x.grad.zero_()

        losses_dict[lambdaa].append(total_loss.item())

        if i % 10 == 0:
            print(f"Lambda {lambdaa}, Iteration {i}/{nb_iters} : Total Loss : {total_loss.item()}")

    sparsity = (x == 0).float().mean().item() * 100
    print(f"Lambda = {lambdaa}: Sparsity = {sparsity:.2f}%")

# Plotting the results
plt.figure(figsize=(10, 6))
for lambdaa, losses in losses_dict.items():
    plt.plot(range(nb_iters), losses, label=f"Lambda = {lambdaa}")

plt.xlabel("Iteration")
plt.ylabel("Total Loss")
plt.title("Proximal Gradient Descent Loss vs. Iteration for Different Lambda Values")
plt.legend()
plt.grid()
plt.show()
```

```
Lambda 0.001, Iteration 0/100 : Total Loss : 2.7937822341918945
Lambda 0.001, Iteration 10/100 : Total Loss : 1.1132606267929077
Lambda 0.001, Iteration 20/100 : Total Loss : 1.1026146411895752
Lambda 0.001, Iteration 30/100 : Total Loss : 1.097569227218628
Lambda 0.001, Iteration 40/100 : Total Loss : 1.0936344861984253
Lambda 0.001, Iteration 50/100 : Total Loss : 1.0902553796768188
Lambda 0.001, Iteration 60/100 : Total Loss : 1.0871518850326538
Lambda 0.001, Iteration 70/100 : Total Loss : 1.0842095613479614
Lambda 0.001, Iteration 80/100 : Total Loss : 1.081372618675232
Lambda 0.001, Iteration 90/100 : Total Loss : 1.0786681175231934
Lambda = 0.001: Sparsity = 17.07%
Lambda 0.01, Iteration 0/100 : Total Loss : 3.020785331726074
Lambda 0.01, Iteration 10/100 : Total Loss : 1.769435167312622
Lambda 0.01, Iteration 20/100 : Total Loss : 1.553912878036499
Lambda 0.01, Iteration 30/100 : Total Loss : 1.4077719449996948
Lambda 0.01, Iteration 40/100 : Total Loss : 1.2892885208129883
Lambda 0.01, Iteration 50/100 : Total Loss : 1.2061007022857666
Lambda 0.01, Iteration 60/100 : Total Loss : 1.1538927555084229
Lambda 0.01, Iteration 70/100 : Total Loss : 1.118927240371704
Lambda 0.01, Iteration 80/100 : Total Loss : 1.1026290655136108
Lambda 0.01, Iteration 90/100 : Total Loss : 1.0894405841827393
Lambda = 0.01: Sparsity = 91.87%
Lambda 0.05, Iteration 0/100 : Total Loss : 6.06582498550415
Lambda 0.05, Iteration 10/100 : Total Loss : 1.820993185043335
Lambda 0.05, Iteration 20/100 : Total Loss : 1.2365082502365112
Lambda 0.05, Iteration 30/100 : Total Loss : 1.2194335460662842
Lambda 0.05, Iteration 40/100 : Total Loss : 1.2145264148712158
Lambda 0.05, Iteration 50/100 : Total Loss : 1.2111324071884155
Lambda 0.05, Iteration 60/100 : Total Loss : 1.2089495658874512
Lambda 0.05, Iteration 70/100 : Total Loss : 1.2080544233322144
Lambda 0.05, Iteration 80/100 : Total Loss : 1.2074052095413208
Lambda 0.05, Iteration 90/100 : Total Loss : 1.2069278955459595
Lambda = 0.05: Sparsity = 97.56%
Lambda 0.1, Iteration 0/100 : Total Loss : 11.716358184814453
Lambda 0.1, Iteration 10/100 : Total Loss : 1.5414475202560425
Lambda 0.1, Iteration 20/100 : Total Loss : 1.3502055406570435
Lambda 0.1, Iteration 30/100 : Total Loss : 1.3466498851776123
Lambda 0.1, Iteration 40/100 : Total Loss : 1.34437096118927
Lambda 0.1, Iteration 50/100 : Total Loss : 1.342882513999939
Lambda 0.1, Iteration 60/100 : Total Loss : 1.341903567314148
Lambda 0.1, Iteration 70/100 : Total Loss : 1.3414863348007202
Lambda 0.1, Iteration 80/100 : Total Loss : 1.3413445949554443
Lambda 0.1, Iteration 90/100 : Total Loss : 1.341244101524353
Lambda = 0.1: Sparsity = 98.37%
```



- Lambda = 0.01 seems to be a good value for the regularization parameter as it yields a 91.87% of sparsity for the solution vector and the loss is still low.

Question 7

Let's now implement the BFGS method and compare it with the regular gradient descent on the first problem. We reuse the variables name from the decription of the BFGS technique. We will use a constant step size of 0.1 and 100 iterations.

In [ ]:
```python
# BFGS Method

nb_iters = 100
step_size = 0.01 # constant step size

x = torch.randn(A.size(1), device=device, requires_grad=True)
H = torch.eye(x.size(0), device=x.device)

for i in range(nb_iters):
    loss = f(x, A, y)
    loss.backward()
    g = x.grad.clone()
    x.grad.zero_()

    x_new = (x - step_size * torch.matmul(H, g)).detach().requires_grad_(True)
    f_new = f(x_new, A, y)
    f_new.backward()
    g_new = x_new.grad.clone()
    x_new.grad.zero_()

    s = x_new - x
    v = g_new - g
    dot_product = torch.dot(s, v)
    if dot_product > 0:
        rho = torch.outer(s, v)
        I = torch.eye(H.size(0), device=device)
        H = (I - rho / dot_product) @ H @ (I - rho / dot_product).T + torch.outer(s, s) / dot_product

    if i % 10 == 0 or i == nb_iters - 1:
        print(f"Iteration {i}/{nb_iters} : Loss : {loss.item()}")

    x = x_new.detach().requires_grad_(True)
```

```
Iteration 0/100 : Loss : 1.059018611907959
Iteration 10/100 : Loss : 1.0531277656555176
Iteration 20/100 : Loss : 1.0471994876861572
Iteration 30/100 : Loss : 1.0418123006820679
Iteration 40/100 : Loss : 1.0369892120361328
Iteration 50/100 : Loss : 1.032723307609558
Iteration 60/100 : Loss : 1.0289827585220337
Iteration 70/100 : Loss : 1.0257185697555542
Iteration 80/100 : Loss : 1.0228744745254517
Iteration 90/100 : Loss : 1.020393967628479
Iteration 99/100 : Loss : 1.018429160118103
```

- We observe very similar results between BFGS and the regular gradient descent. The loss is very similar. One thing to note is that the BFGS method is more computationally expensive than the regular gradient descent and we have nan values for the loss with bigger step sizes (I wasn't able to solve this problem).

Question 8

We implement the stochastic gradient variant of the BFGS method. We will use a constant step size of 0.1 and 10 epochs since it is very computationally expensive (we use the explicit formular to compute the gradient this time).

In [92]:
```python
def grad_f(x, A, y):
    exp_term = torch.exp(A @ x)
    numerator = 2 * exp_term * (exp_term * (y - 1) + y)
    denominator = (1 + exp_term) ** 3
    scalar_factors = - (numerator / denominator)
    return torch.mean(scalar_factors.unsqueeze(1) * A, dim=0)
```

In [ ]:
```python
# BFGS Method with stochastic gradient

epochs = 100
step_size = 0.001  # constant step size
batch_size = 128

x = torch.randn(A.size(1), device=device, requires_grad=True)
H = torch.eye(A.size(1), device=device)
for epoch in range(epochs):
    indices = torch.randperm(A.size(0))
    A_shuffled = A[indices]
    y_shuffled = y[indices]
    total_loss = 0
    for b in range(0, A.size(0), batch_size):
        A_batch = A_shuffled[b:b+batch_size]
        y_batch = y_shuffled[b:b+batch_size]

        g = grad_f(x, A_batch, y_batch)
        loss = f(x, A_batch, y_batch)
        total_loss += loss.item()
        x_new = x - step_size * torch.matmul(H, g)
        g_new = grad_f(x_new, A_batch, y_batch)
        s = x_new - x
        v = g_new - g
        dot_product = torch.dot(s, v)

        if dot_product > 0:
            rho = torch.outer(s, v)
            I = torch.eye(H.size(0), device=device)
            H = (I - rho / dot_product) @ H @ (I - rho / dot_product).T + torch.outer(s, s) / dot_product

        x = x_new
    avg_loss = total_loss / (A.size(0) / batch_size)
    if epoch % 10 == 0 or epoch == epochs - 1:
        print(f"Epochs {epoch}/{epochs} : Loss : {avg_loss}")
```

```
Epochs 0/100 : Loss : 2.4262610778808593
Epochs 10/100 : Loss : 1.2706128845214844
Epochs 20/100 : Loss : 1.2389833984375
Epochs 30/100 : Loss : 1.2041190795898438
Epochs 40/100 : Loss : 1.1746941375732423
Epochs 50/100 : Loss : 1.1399949493408204
Epochs 60/100 : Loss : 1.125844223022461
Epochs 70/100 : Loss : 1.1115374298095704
Epochs 80/100 : Loss : 1.1006154937744141
Epochs 90/100 : Loss : 1.090562286376953
Epochs 99/100 : Loss : 1.0829546661376954
```

Question 9

Number of data access and vector updates for the BFGS Method.

In [ ]:
```python
# BFGS Method
```

```
nb_iters = 100
step_size = 0.01 # constant step size

x = torch.randn(A.size(1), device=device, requires_grad=True)
H = torch.eye(x.size(0), device=x.device)

data_access = 0
vector_updates = 0

for i in range(nb_iters):
    loss = f(x, A, y)
    data_access += 1
    loss.backward()
    g = x.grad.clone()
    x.grad.zero_()

    x_new = (x - step_size * torch.matmul(H, g)).detach().requires_grad_(True)
    vector_updates += 1
    f_new = f(x_new, A, y)
    data_access += 1
    f_new.backward()
    g_new = x_new.grad.clone()
    x_new.grad.zero_()

    s = x_new - x
    v = g_new - g
    dot_product = torch.dot(s, v)
    if dot_product > 0:
        rho = torch.outer(s, v)
        I = torch.eye(H.size(0), device=device)
        H = (I - rho / dot_product) @ H @ (I - rho / dot_product).T + torch.outer(s, s) / dot_product

    x = x_new.detach().requires_grad_(True)

print("Data accesses:", data_access)
print("Vector updates:", vector_updates)
```

```
Data accesses: 200
Vector updates: 100
```

Number of data access and vector updates for the Adagrad Method.

In [229…

```
# Adagrad implementation

batch_size = 16
nb_epochs = 100
lr = 2
epsilon = 1e-8

data_access = 0
vector_updates = 0

x = torch.randn(A.size(1), requires_grad=True, device=device)
G = torch.zeros_like(x, device=device)
adagrad_losses = []
for i in range(nb_epochs):
    indices = torch.randperm(A.size(0))
    A_shuffled = A[indices]
    y_shuffled = y[indices]
    total_loss = 0
    for b in range(0, A.size(0), batch_size):
        data_access += 1
        A_batch = A_shuffled[b:b+batch_size]
        y_batch = y_shuffled[b:b+batch_size]
        loss = f(x, A_batch, y_batch)
        total_loss += loss.item()
        loss.backward()
        G.add_(x.grad.pow(2))
        with torch.no_grad():
            x -= lr / torch.sqrt(G + epsilon) * x.grad
            vector_updates += 1
            x.grad.zero_()
    avg_loss = total_loss / (A.size(0) / batch_size)
    adagrad_losses.append(avg_loss)

print("Data accesses:", data_access)
print("Vector updates:", vector_updates)
```

```
Data accesses: 6300
Vector updates: 6300
```

- As we can see the amount of Data accesses and vector uppdates for the Adagrad Method with a batch size of 16 on 200 iterations is 6300 meanwhile for the BFGS method with 100 iterations it is 200 and 100 respectively. This is a huge difference in terms of computational cost and computational speed. The BFGS is much faster compared to the Adagrad method, but the adagrad method is more accurate and the loss is lower at the end.

- We therefore have 1 vector update per iteration (basically per batch update) but 63 per epoch.

Question 10

We combine the proximal gradient implementation from question 6 with the BFGS technique on the regularized problem. Results are very similar to the proximal gradient descent, for a fixed value of lambda which is 0.001 in our case. The loss is slightly lower for the Proximal Gradient approach of question 6 with this configuration.

In [185…

```
# BFGS Method with Proximal gradient

nb_iters = 100
step_size = 0.1
lambdaa = 0.001

x = torch.randn(A.size(1), device=device, requires_grad=True)
H = torch.eye(x.size(0), device=x.device)

for i in range(nb_iters):
    loss = f(x, A, y)
    l1_term = lambdaa * torch.norm(x, p=1)
    total_loss = loss + l1_term

    loss.backward()
    g = x.grad.clone()
    x.grad.zero_()

    with torch.no_grad():
        x_bfgs = x - step_size * torch.matmul(H, g)
        x_prox = torch.sign(x_bfgs) * torch.clamp(torch.abs(x_bfgs) - step_size * lambdaa, min=0.0)

    x_new = x_prox.detach().requires_grad_(True)

    f_new = f(x_new, A, y)
    f_new.backward()
    g_new = x_new.grad.clone()
    x_new.grad.zero_()

    s = x_prox - x.detach()
    v = g_new - g.detach()
    dot_product = torch.dot(s, v)
    if dot_product > 0:
        rho = torch.outer(s, v)
        I = torch.eye(H.size(0), device=device)
        H = (I - rho / dot_product) @ H @ (I - rho / dot_product).T + torch.outer(s, s) / dot_product

    x = x_new

    if i % 10 == 0 or i == nb_iters - 1:
        print(f"Iteration {i}/{nb_iters} : Loss : {total_loss.item()}")
```

```
Iteration 0/100 : Loss : 1.2143223285675049
Iteration 10/100 : Loss : 1.1368565559387207
Iteration 20/100 : Loss : 1.1086654663085938
Iteration 30/100 : Loss : 1.0978338718414307
Iteration 40/100 : Loss : 1.0917375087738037
Iteration 50/100 : Loss : 1.0873628854751587
Iteration 60/100 : Loss : 1.0828802585601807
Iteration 70/100 : Loss : 1.078061580657959
Iteration 80/100 : Loss : 1.0731076002120972
Iteration 90/100 : Loss : 1.0677419900894165
Iteration 99/100 : Loss : 1.0631940364837646
```

Question 11

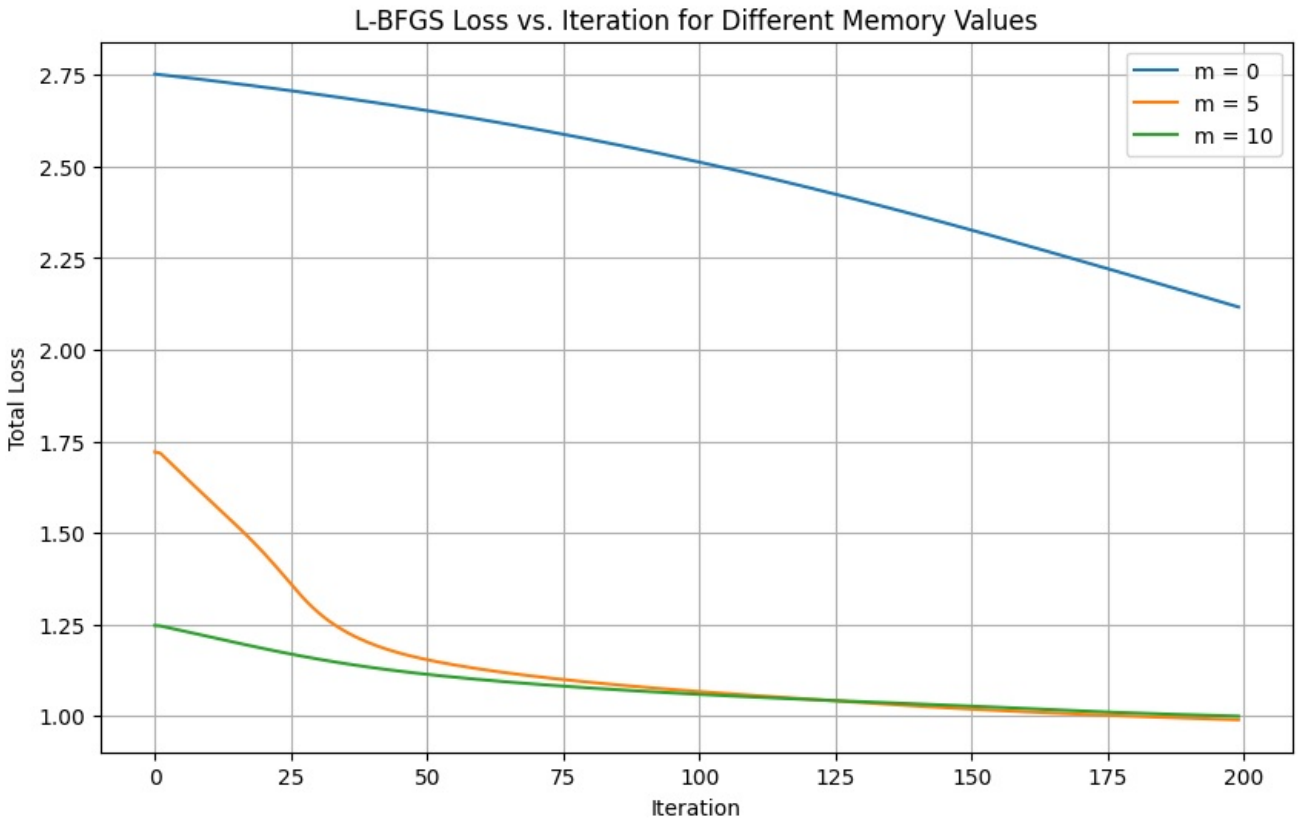
Implementation of the L-BFGS method for various m values.

In [221…

```
# L-BFGS Method

nb_iters = 200
alpha = 0.01
ms = [0, 5, 10]

losses_dict = {m: [] for m in ms}

for m in ms:
    x = torch.randn(A.size(1), device=device, requires_grad=True)
    memory = []
    for k in range(nb_iters):
        if k == 0:
            loss = f(x, A, y)
            loss.backward()
            g = x.grad.clone()
            x.grad.zero_()
```

```
            x_new = (x - alpha * g).detach().requires_grad_(True)
            f_new = f(x_new, A, y)
            f_new.backward()
            g_new = x_new.grad.clone()
            x_new.grad.zero_()

            s = x_new - x
            v = g_new - g
            memory.append((s.detach(), v.detach()))
            x = x_new

        else:
            loss = f(x, A, y)
            loss.backward()
            g = x.grad.clone()
            x.grad.zero_()

            q = g.clone()
            gammas = []

            for l in range(1, min(m, k) + 1):
                s, v = memory[-l]
                if torch.dot(s, v) > 0:
                    gamma = torch.dot(s, q) / torch.dot(v, s)
                    gammas.append(gamma)
                    q -= gamma * v

            gammas = gammas[::-1]

            for l in range(len(gammas)):
                s, v = memory[l]
                if torch.dot(s, v) > 0:
                    delta = torch.dot(v, q) / torch.dot(v, s)
                    q += (gammas[l] - delta) * s

            x_new = (x - alpha * q).detach().requires_grad_(True)
            f_new = f(x_new, A, y)
            f_new.backward()
            g_new = x_new.grad.clone()
            x_new.grad.zero_()

            s = x_new - x
            v = g_new - g

            if k >= m:
                memory.pop(0)
            memory.append((s.detach(), v.detach()))
            x = x_new

        losses_dict[m].append(loss.item())

        if k % 10 == 0 or k == nb_iters - 1:
            print(f"Iteration {k}/{nb_iters} : Loss : {loss}")

plt.figure(figsize=(10, 6))
for m, losses in losses_dict.items():
    plt.plot(range(nb_iters), losses, label=f"m = {m}")

plt.xlabel("Iteration")
plt.ylabel("Total Loss")
plt.title("L-BFGS Loss vs. Iteration for Different Memory Values")
plt.legend()
plt.grid()
plt.show()
```

```
Iteration 0/200 : Loss : 2.7521960735321045
Iteration 10/200 : Loss : 2.7349860668182373
Iteration 20/200 : Loss : 2.7165040969848633
Iteration 30/200 : Loss : 2.6966559886932373
Iteration 40/200 : Loss : 2.6753458976745605
Iteration 50/200 : Loss : 2.6524786949157715
Iteration 60/200 : Loss : 2.6279611587524414
Iteration 70/200 : Loss : 2.6017086505889893
Iteration 80/200 : Loss : 2.5736496448516846
Iteration 90/200 : Loss : 2.5437309741973877
Iteration 100/200 : Loss : 2.511927604675293
Iteration 110/200 : Loss : 2.4782464504241943
Iteration 120/200 : Loss : 2.442734956741333
Iteration 130/200 : Loss : 2.405484914779663
Iteration 140/200 : Loss : 2.366633176803589
Iteration 150/200 : Loss : 2.326359987258911
Iteration 160/200 : Loss : 2.2848849296569824
Iteration 170/200 : Loss : 2.242457628250122
Iteration 180/200 : Loss : 2.1993463039398193
Iteration 190/200 : Loss : 2.155829906463623
Iteration 199/200 : Loss : 2.1165459156036377
Iteration 0/200 : Loss : 1.72097909450531
Iteration 10/200 : Loss : 1.5903719663619995
Iteration 20/200 : Loss : 1.445297360420227
Iteration 30/200 : Loss : 1.2830710411071777
Iteration 40/200 : Loss : 1.1961872577667236
Iteration 50/200 : Loss : 1.1539891958236694
Iteration 60/200 : Loss : 1.1278122663497925
Iteration 70/200 : Loss : 1.108181357383728
Iteration 80/200 : Loss : 1.092246651649475
Iteration 90/200 : Loss : 1.078782320022583
Iteration 100/200 : Loss : 1.0670664310455322
Iteration 110/200 : Loss : 1.056532621383667
Iteration 120/200 : Loss : 1.0466235876083374
Iteration 130/200 : Loss : 1.0368529558181763
Iteration 140/200 : Loss : 1.0274471044540405
Iteration 150/200 : Loss : 1.019110083580017
Iteration 160/200 : Loss : 1.0118350982666016
Iteration 170/200 : Loss : 1.0053967237472534
Iteration 180/200 : Loss : 0.9996223449707031
Iteration 190/200 : Loss : 0.9943934082984924
Iteration 199/200 : Loss : 0.9900833964347839
Iteration 0/200 : Loss : 1.2474678754806519
Iteration 10/200 : Loss : 1.2166744470596313
Iteration 20/200 : Loss : 1.1843606233596802
Iteration 30/200 : Loss : 1.1556272506713867
Iteration 40/200 : Loss : 1.1323109865188599
Iteration 50/200 : Loss : 1.1140737533569336
Iteration 60/200 : Loss : 1.099438190460205
Iteration 70/200 : Loss : 1.0872317552566528
Iteration 80/200 : Loss : 1.07676100730896
Iteration 90/200 : Loss : 1.067601203918457
Iteration 100/200 : Loss : 1.0594666004180908
Iteration 110/200 : Loss : 1.0521444082260132
Iteration 120/200 : Loss : 1.0454614162445068
Iteration 130/200 : Loss : 1.0392544269561768
Iteration 140/200 : Loss : 1.033327579498291
Iteration 150/200 : Loss : 1.027359127998352
Iteration 160/200 : Loss : 1.0207914113998413
Iteration 170/200 : Loss : 1.0139671564102173
Iteration 180/200 : Loss : 1.0084177255630493
Iteration 190/200 : Loss : 1.0036541223526
Iteration 199/200 : Loss : 0.9999207258224487
```



Question 12

We compute the number of accesses to data points and updates at every iteration for the L-BFGS method with m=10.

In [230…

```
# L-BFGS Method

nb_iters = 100
alpha = 0.01
ms = 10

x = torch.randn(A.size(1), device=device, requires_grad=True)
memory = []
data_access = 0
vector_updates = 0

for k in range(nb_iters):
    if k == 0:
        loss = f(x, A, y)
        data_access += 1
        loss.backward()
        g = x.grad.clone()
        x.grad.zero_()

        x_new = (x - alpha * g).detach().requires_grad_(True)
        f_new = f(x_new, A, y)
        data_access += 1
        f_new.backward()
        g_new = x_new.grad.clone()
        x_new.grad.zero_()

        s = x_new - x
        v = g_new - g
        memory.append((s.detach(), v.detach()))
        x = x_new
        vector_updates += 1

    else:
        loss = f(x, A, y)
        data_access += 1
        loss.backward()
        g = x.grad.clone()
        x.grad.zero_()

        q = g.clone()
        gammas = []

        if m > 0:
            for l in range(1, min(m, k) + 1):
                s, v = memory[-l]
                if torch.dot(s, v) > 0:
                    gamma = torch.dot(s, q) / torch.dot(v, s)
```

```
                    gammas.append(gamma)
                    q -= gamma * v

            gammas = gammas[::-1]

            for l in range(len(gammas)):
                s, v = memory[l]
                if torch.dot(s, v) > 0:
                    delta = torch.dot(v, q) / torch.dot(v, s)
                    q += (gammas[l] - delta) * s

        x_new = (x - alpha * q).detach().requires_grad_(True)
        f_new = f(x_new, A, y)
        data_access += 1
        f_new.backward()
        g_new = x_new.grad.clone()
        x_new.grad.zero_()

        s = x_new - x
        v = g_new - g

        if k >= m:
            memory.pop(0)
        memory.append((s.detach(), v.detach()))
        x = x_new
        vector_updates += 1


print("Data accesses:", data_access)
print("Vector updates:", vector_updates)
```

```
Data accesses: 200
Vector updates: 100
```

- For 100 iterations, the L-BFGS method has 200 Data accesses and 100 vector updates, while as previously seen, for the Adagrad Method with a batch size of 16 on 100 iterations the number of data accesses and vector updates is 6300. Again, the L-BFGS is much faster compared to the Adagrad method, and also more memory efficient but the adagrad method is more accurate and the loss is lower at the end.
- Therefore we have 1 vector update per iteration.