podkohlxo

April 21, 2024

1 Worksheet 21

Name: JINGYI ZHANG UID: U26578499

1.0.1 Topics

- Logistic Regression
- Gradient Descent

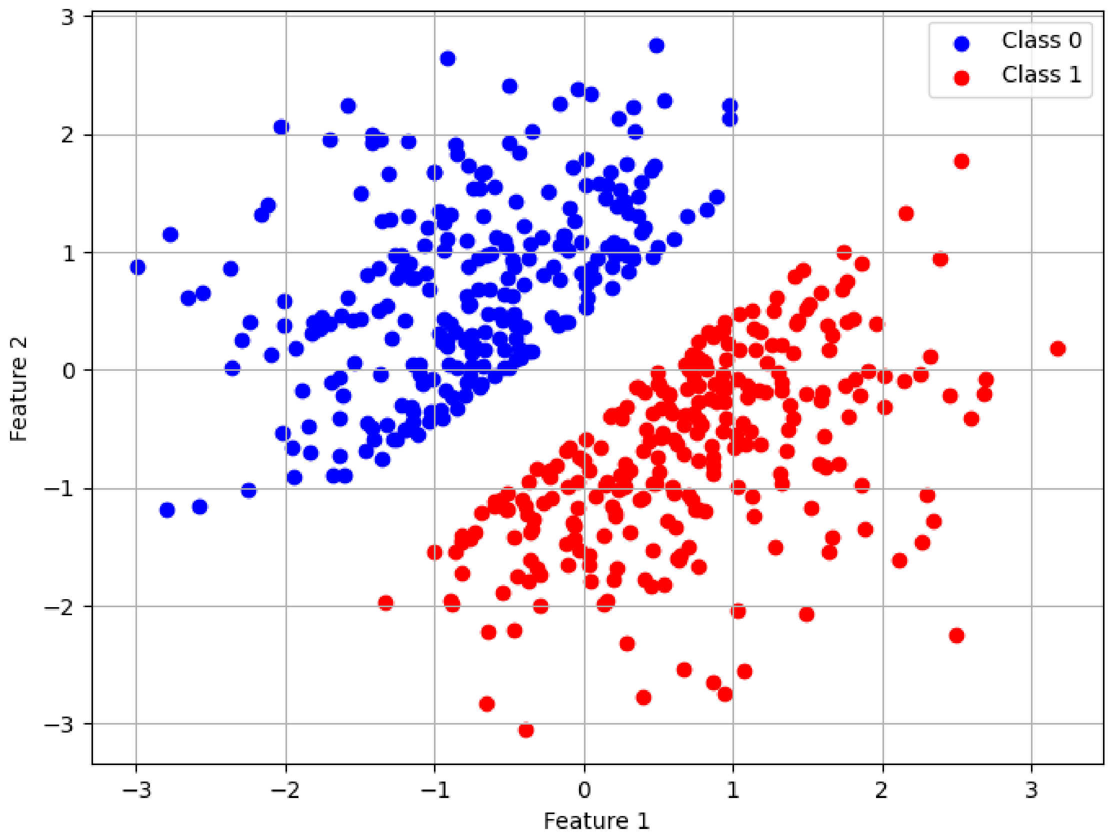
1.1 Logistic Regression

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn.datasets as datasets
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import PolynomialFeatures

centers = [[0, 0]]
t, _ = datasets.make_blobs(n_samples=750, centers=centers, cluster_std=1,
 ↪random_state=0)

# LINE
def generate_line_data():
    # create some space between the classes
    X = np.array(list(filter(lambda x : x[0] - x[1] < -.5 or x[0] - x[1] > .5,
 ↪t)))
    Y = np.array([1 if x[0] - x[1] >= 0 else 0 for x in X])
    return X, Y

# CIRCLE
def generate_circle_data(t):
    # create some space between the classes
    X = np.array(list(filter(lambda x : (x[0] - centers[0][0])**2 + (x[1] -
 ↪centers[0][1])**2 < 1 or (x[0] - centers[0][0])**2 + (x[1] -
 ↪centers[0][1])**2 > 1.5, t)))
    Y = np.array([1 if (x[0] - centers[0][0])**2 + (x[1] - centers[0][1])**2 >=
 ↪1 else 0 for x in X])
```

```
    return X, Y

# XOR
def generate_xor_data():
    X = np.array([
        [0,0],
        [0,1],
        [1,0],
        [1,1]])
    Y = np.array([x[0]^x[1] for x in X])
    return X, Y
```

a) Using the above code, generate and plot data that is linearly separable.

[2]:
```
X, Y = generate_line_data()

plt.figure(figsize=(8, 6))
plt.scatter(X[Y == 0][:, 0], X[Y == 0][:, 1], color='blue', label='Class 0')
plt.scatter(X[Y == 1][:, 0], X[Y == 1][:, 1], color='red', label='Class 1')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



b) Fit a logistic regression model to the data a print out the coefficients.

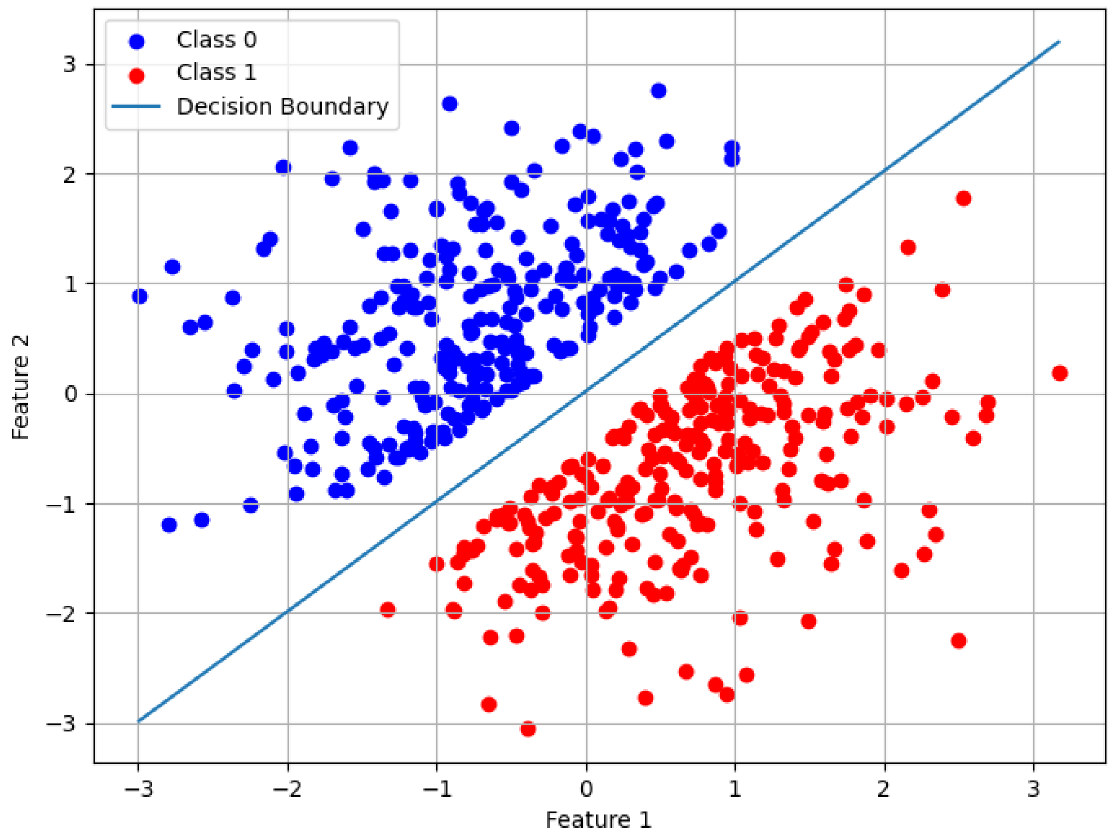
[3]:
```
model = LogisticRegression().fit(X, Y)
model.coef_,model.intercept_
```

[3]:
```
(array([[ 4.11128306, -4.10408124]]), array([0.06146435]))
```

c) Using the coefficients, plot the line through the scatter plot you created in a). (Note: you need to do some math to get the line in the right form)

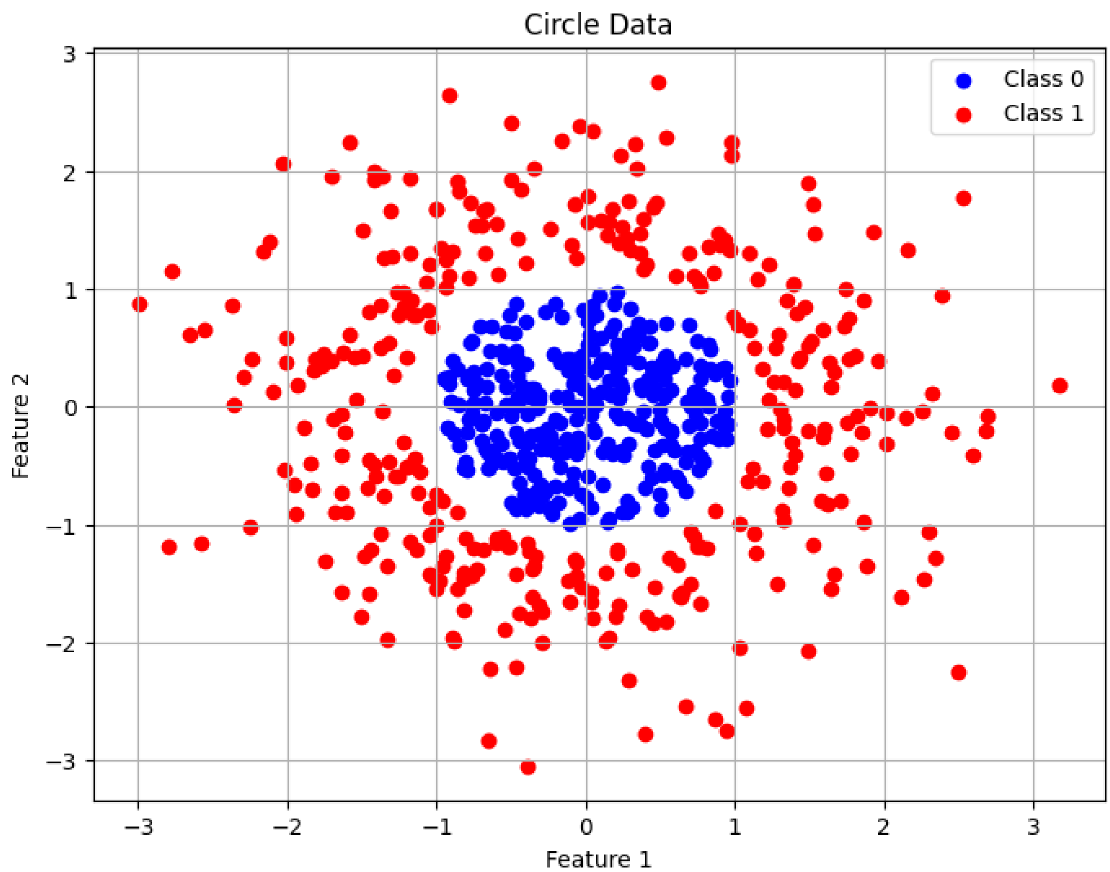
[4]:
```
plt.figure(figsize=(8, 6))
plt.scatter(X[Y == 0][:, 0], X[Y == 0][:, 1], color='blue', label='Class 0')
plt.scatter(X[Y == 1][:, 0], X[Y == 1][:, 1], color='red', label='Class 1')

feature1 = np.array([X[:, 0].min(), X[:, 0].max()])
feature2 = -(model.intercept_[0] + model.coef_[0][0] * feature1) / model.
↪coef_[0][1]

plt.plot(feature1, feature2, label='Decision Boundary')

plt.xlabel('Feature 1')
```

```
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



d) Using the above code, generate and plot the CIRCLE data.

[5]:
```
X, Y = generate_circle_data(t)

plt.figure(figsize=(8, 6))
plt.scatter(X[Y == 0][:, 0], X[Y == 0][:, 1], color='blue', label='Class 0')
plt.scatter(X[Y == 1][:, 0], X[Y == 1][:, 1], color='red', label='Class 1')
plt.title('Circle Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



e) Notice that the equation of an ellipse is of the form

$$ax^2 + by^2 = c$$

Fit a logistic regression model to an appropriate transformation of X.

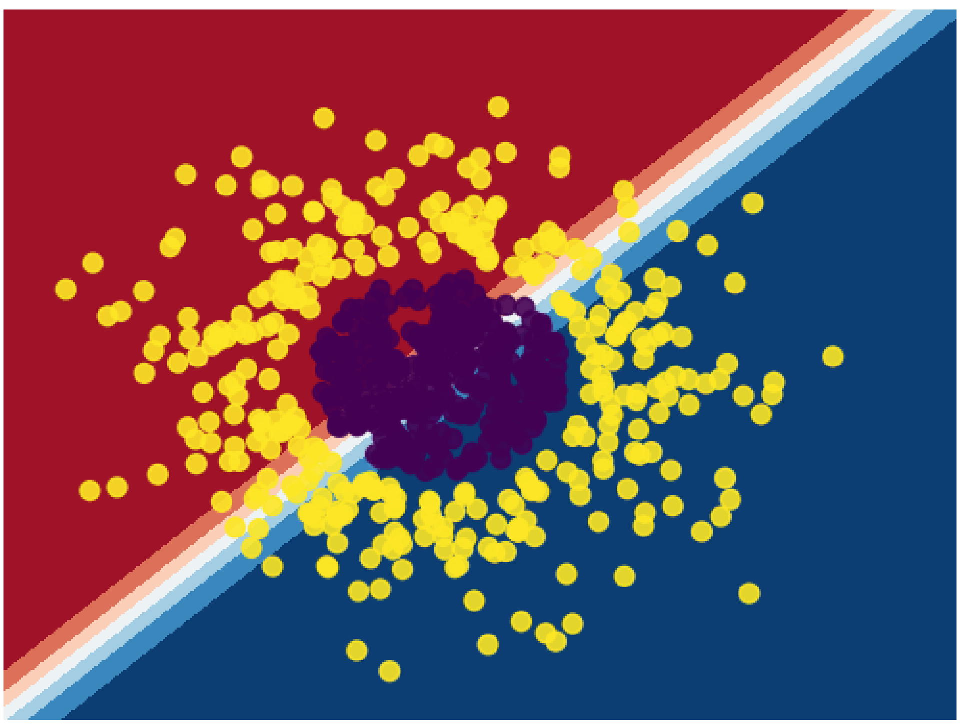
[6]:
```
#model = ...
from sklearn.preprocessing import PolynomialFeatures

# Redefine polynomial features to only include squares of each feature
poly_square = PolynomialFeatures(degree=2, include_bias=False,␣
↪interaction_only=False)
X_circle_square = poly_square.fit_transform(X)[:, [2, 4]]  # Select only x1^2␣
↪and x2^2

# Fit the logistic regression model to the square-only transformed data
model_circle_square = LogisticRegression()
model_circle_square.fit(X_circle_square, Y)

# Coefficients for the square-only features
```

```python
print("Coefficients:", model_circle_square.coef_)
print("Intercept:", model_circle_square.intercept_)
```

```
Coefficients: [[4.91410958 4.97630742]]
Intercept: [-6.45841785]
```

f) Plot the decision boundary using the code below.

[7]:
```python
# create a mesh to plot in
h = .02  # step size in the mesh
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
meshData = np.c_[xx.ravel(), yy.ravel()]
fig, ax = plt.subplots()
A = model.predict_proba(meshData)[:, 1].reshape(xx.shape)
Z = model.predict(meshData).reshape(xx.shape)
ax.contourf(xx, yy, A, cmap="RdBu", vmin=0, vmax=1)
ax.axis('off')

# Plot also the training points
ax.scatter(X[:, 0], X[:, 1], c=Y, s=50, alpha=0.9)
plt.show()
```

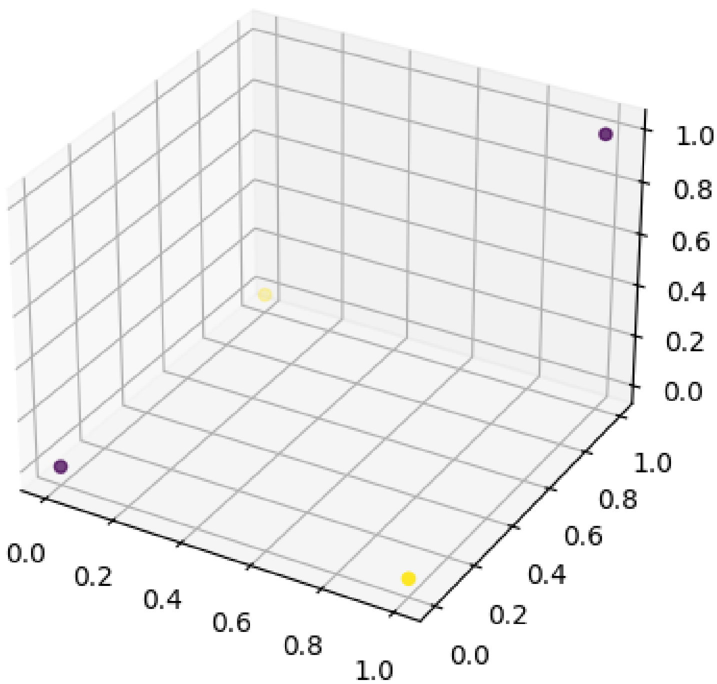


g) Plot the XOR data. In this 2D space, the data is not linearly separable, but by introducing a new feature

$$x_3 = x_1 * x_2$$

(called an interaction term) we should be able to find a hyperplane that separates the data in 3D. Plot this new dataset in 3D.

[8]:
```
from mpl_toolkits.mplot3d import Axes3D

X, Y = generate_xor_data()
ax = plt.axes(projection='3d')
plt.figure(figsize=(8, 6))
ax.scatter3D(X[: , 0], X[: , 1], X[: , 0]* X[: , 1], c=Y)
plt.show()
```




```
<Figure size 800x600 with 0 Axes>
```

h) Apply a logistic regression model using the interaction term. Plot the decision boundary.

[9]:
```
poly = PolynomialFeatures(interaction_only=True)
lr = LogisticRegression(verbose=0)
model = make_pipeline(poly, lr).fit(X, Y)

# create a mesh to plot in
h = .02  # step size in the mesh
```

```
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
meshData = np.c_[xx.ravel(), yy.ravel()]

fig, ax = plt.subplots()
A = model.predict_proba(meshData)[:, 1].reshape(xx.shape)
Z = model.predict(meshData).reshape(xx.shape)
ax.contourf(xx, yy, A, cmap="RdBu", vmin=0, vmax=1)
ax.axis('off')

# Plot also the training points
ax.scatter(X[:, 0], X[:, 1], color=Y, s=50, alpha=0.9)
plt.show()
```

[10]:
```
'''
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LogisticRegression
%matplotlib widget
```

```
for i in range(20000):
    for solver in ['lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag',␣
 ↪'saga']:
        X_transform = PolynomialFeatures(interaction_only=True,␣
 ↪include_bias=False).fit_transform(X)
        model = LogisticRegression(verbose=0, solver=solver, random_state=i,␣
 ↪max_iter=10000)
        model.fit(X_transform, Y)
        print(model.score(X_transform, Y))
        if model.score(X_transform, Y) > .75:
            print("random state = ", i)
            print("solver = ", solver)
            break

print(model.coef_)
print(model.intercept_)

xx, yy = np.meshgrid([x / 10 for x in range(-1, 11)], [x / 10 for x in␣
 ↪range(-1, 11)])
z = - model.intercept_ / model.coef_[0][2] - model.coef_[0][0] * xx / model.
 ↪coef_[0][2] - model.coef_[0][1] * yy / model.coef_[0][2]

ax = plt.axes(projection='3d')
ax.scatter3D(X[: , 0], X[: , 1], X[: , 0]* X[: , 1], c=Y)
ax.plot_surface(xx, yy, z, alpha=0.5)
plt.show()
'''
```

[10]:
```
'\nimport numpy as np\nimport matplotlib.pyplot as plt\nfrom
mpl_toolkits.mplot3d import Axes3D\nfrom sklearn.preprocessing import
PolynomialFeatures\nfrom sklearn.linear_model import
LogisticRegression\n%matplotlib widget\nfor i in range(20000):\n    for solver
in [\'lbfgs\', \'liblinear\', \'newton-cg\', \'newton-cholesky\', \'sag\',
\'saga\']:\n        X_transform = PolynomialFeatures(interaction_only=True,
include_bias=False).fit_transform(X)\n        model =
LogisticRegression(verbose=0, solver=solver, random_state=i, max_iter=10000)\n
model.fit(X_transform, Y)\n        print(model.score(X_transform, Y))\n
if model.score(X_transform, Y) > .75:\n            print("random state = ", i)\n
print("solver = ", solver)\n
break\n\nprint(model.coef_)\nprint(model.intercept_)\n\nxx, yy = np.meshgrid([x
/ 10 for x in range(-1, 11)], [x / 10 for x in range(-1, 11)])\nz = -
model.intercept_ / model.coef_[0][2] - model.coef_[0][0] * xx /
model.coef_[0][2] - model.coef_[0][1] * yy / model.coef_[0][2]\n\nax =
plt.axes(projection=\'3d\')\nax.scatter3D(X[: , 0], X[: , 1], X[: , 0]* X[: ,
1], c=Y)\nax.plot_surface(xx, yy, z, alpha=0.5)\nplt.show()\n'
```

i) Using the code below that generates 3 concentric circles, fit a logisitc regression model to it and plot the decision boundary.

[11]:
```
t, _ = datasets.make_blobs(n_samples=1500, centers=centers, cluster_std=2,
                                random_state=0)

# CIRCLES
def generate_circles_data(t):
    def label(x):
        if x[0]**2 + x[1]**2 >= 2 and x[0]**2 + x[1]**2 < 8:
            return 1
        if x[0]**2 + x[1]**2 >= 8:
            return 2
        return 0
    # create some space between the classes
    X = np.array(list(filter(lambda x : (x[0]**2 + x[1]**2 < 1.8 or x[0]**2 +
 ↪x[1]**2 > 2.2) and (x[0]**2 + x[1]**2 < 7.8 or x[0]**2 + x[1]**2 > 8.2), t)))
    Y = np.array([label(x) for x in X])
    return X, Y

X, Y = generate_circles_data(t)

poly = PolynomialFeatures(2)
lr = LogisticRegression(verbose=2)
model = make_pipeline(poly, lr).fit(X, Y)


#
h = 0.02
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

#
Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

#
plt.contourf(xx, yy, Z, alpha=0.8, cmap='viridis')
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', s=20, cmap='viridis')
plt.title("Decision Boundary for 3 Concentric Circles")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```

```
RUNNING THE L-BFGS-B CODE

           * * *
```

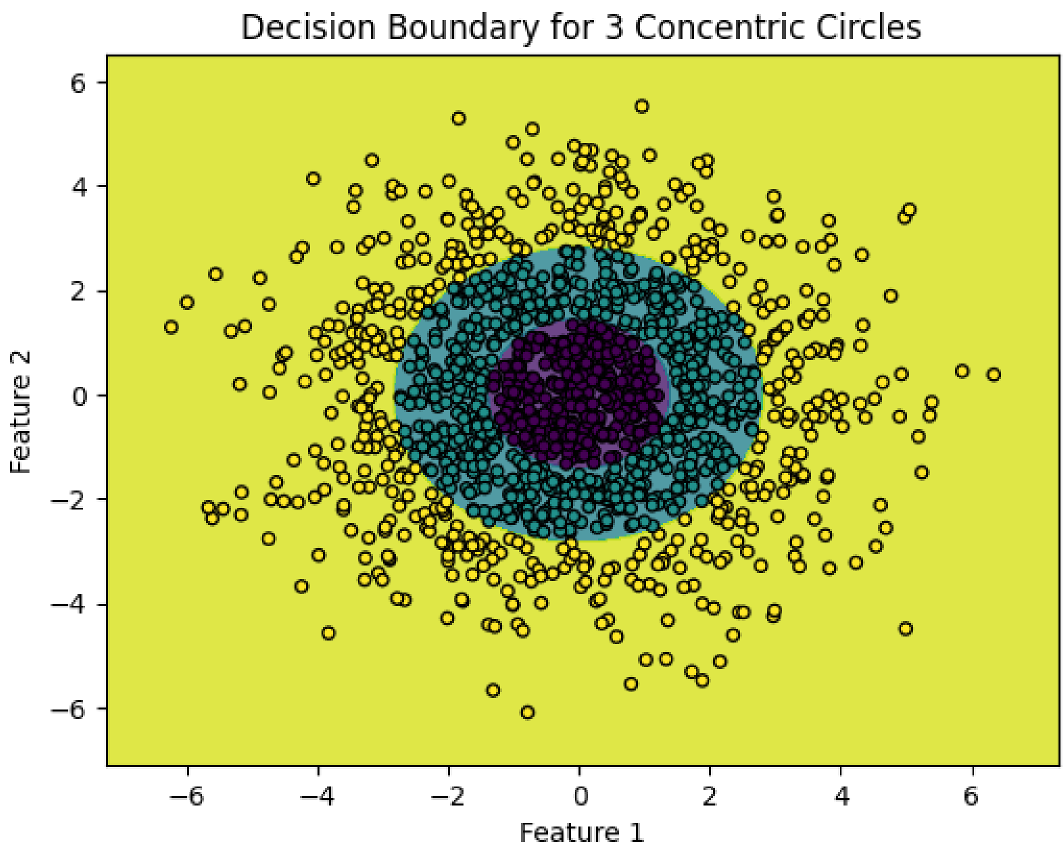
```
Machine precision = 2.220D-16
 N =           21     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.09861D+00    |proj g|=  1.56782D+00

At iterate    1    f=  8.95383D-01    |proj g|=  5.58383D-01

At iterate    2    f=  8.27195D-01    |proj g|=  3.32189D-01

At iterate    3    f=  7.96120D-01    |proj g|=  6.06694D-01

At iterate    4    f=  7.66577D-01    |proj g|=  1.62504D-01

At iterate    5    f=  7.38227D-01    |proj g|=  1.74906D-01

At iterate    6    f=  6.22098D-01    |proj g|=  1.59122D-01

At iterate    7    f=  4.78311D-01    |proj g|=  1.20393D-01

At iterate    8    f=  2.88632D-01    |proj g|=  2.29261D-01

At iterate    9    f=  2.73371D-01    |proj g|=  3.32278D-01

At iterate   10    f=  1.61027D-01    |proj g|=  1.86164D-01

At iterate   11    f=  1.31057D-01    |proj g|=  8.20086D-02

At iterate   12    f=  1.09857D-01    |proj g|=  4.31233D-02

At iterate   13    f=  9.54732D-02    |proj g|=  3.14860D-02

At iterate   14    f=  8.82113D-02    |proj g|=  3.19970D-02

At iterate   15    f=  8.65265D-02    |proj g|=  2.58420D-02

At iterate   16    f=  8.51986D-02    |proj g|=  7.71859D-03

At iterate   17    f=  8.49323D-02    |proj g|=  4.32702D-03

At iterate   18    f=  8.47892D-02    |proj g|=  2.54731D-03

At iterate   19    f=  8.45880D-02    |proj g|=  4.72692D-03

At iterate   20    f=  8.43495D-02    |proj g|=  9.76915D-03
```

```
At iterate   21    f=  8.39228D-02    |proj g|=  1.41123D-02

At iterate   22    f=  8.30751D-02    |proj g|=  1.33183D-02

At iterate   23    f=  8.27346D-02    |proj g|=  1.95171D-02

At iterate   24    f=  8.16237D-02    |proj g|=  1.19741D-02

At iterate   25    f=  7.96364D-02    |proj g|=  7.19066D-03

At iterate   26    f=  7.76664D-02    |proj g|=  1.85242D-02

At iterate   27    f=  7.46684D-02    |proj g|=  3.03654D-02

At iterate   28    f=  7.05634D-02    |proj g|=  5.30341D-02

At iterate   29    f=  6.54279D-02    |proj g|=  3.59637D-02

At iterate   30    f=  6.18421D-02    |proj g|=  2.28137D-02

At iterate   31    f=  5.85717D-02    |proj g|=  6.21250D-03

At iterate   32    f=  5.74600D-02    |proj g|=  5.64093D-03

At iterate   33    f=  5.58955D-02    |proj g|=  1.31517D-02

At iterate   34    f=  5.38386D-02    |proj g|=  1.53870D-02

At iterate   35    f=  5.32192D-02    |proj g|=  1.92596D-02

At iterate   36    f=  5.19024D-02    |proj g|=  3.39938D-03

At iterate   37    f=  5.17519D-02    |proj g|=  1.98737D-03

At iterate   38    f=  5.16150D-02    |proj g|=  1.95118D-03

At iterate   39    f=  5.15147D-02    |proj g|=  2.96761D-03

At iterate   40    f=  5.14626D-02    |proj g|=  2.59971D-03

At iterate   41    f=  5.13984D-02    |proj g|=  1.62724D-03

At iterate   42    f=  5.13370D-02    |proj g|=  9.99570D-04

At iterate   43    f=  5.12664D-02    |proj g|=  1.37624D-03

At iterate   44    f=  5.11763D-02    |proj g|=  1.60635D-03
```

```
At iterate   45    f=  5.10465D-02    |proj g|=  1.34433D-03

At iterate   46    f=  5.10047D-02    |proj g|=  4.95510D-03

At iterate   47    f=  5.08214D-02    |proj g|=  4.46364D-03

At iterate   48    f=  5.07167D-02    |proj g|=  1.08309D-03

At iterate   49    f=  5.06903D-02    |proj g|=  8.37843D-04

At iterate   50    f=  5.06685D-02    |proj g|=  4.53040D-04

At iterate   51    f=  5.06555D-02    |proj g|=  1.17846D-03

At iterate   52    f=  5.06408D-02    |proj g|=  7.58418D-04

At iterate   53    f=  5.06250D-02    |proj g|=  9.22698D-04

At iterate   54    f=  5.05228D-02    |proj g|=  1.71929D-03

At iterate   55    f=  5.03782D-02    |proj g|=  1.97601D-03

At iterate   56    f=  5.02844D-02    |proj g|=  2.05844D-03

At iterate   57    f=  5.01694D-02    |proj g|=  9.05956D-04

At iterate   58    f=  5.01232D-02    |proj g|=  7.69174D-04

At iterate   59    f=  5.01149D-02    |proj g|=  8.32515D-04

At iterate   60    f=  5.01087D-02    |proj g|=  1.84661D-04

At iterate   61    f=  5.01070D-02    |proj g|=  1.57103D-04

At iterate   62    f=  5.01059D-02    |proj g|=  4.65643D-04

At iterate   63    f=  5.01039D-02    |proj g|=  2.51792D-04

At iterate   64    f=  5.01027D-02    |proj g|=  1.24677D-04

At iterate   65    f=  5.01016D-02    |proj g|=  1.21590D-04

At iterate   66    f=  5.01008D-02    |proj g|=  9.65235D-05

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
```

```
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
   21     66     75      1     0     0   9.652D-05   5.010D-02
  F =   5.0100827208558775E-002

CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
```

```
 This problem is unconstrained.
```



1.2 Gradient Descent

Recall in Linear Regression we are trying to find the line

$$y = X\beta$$

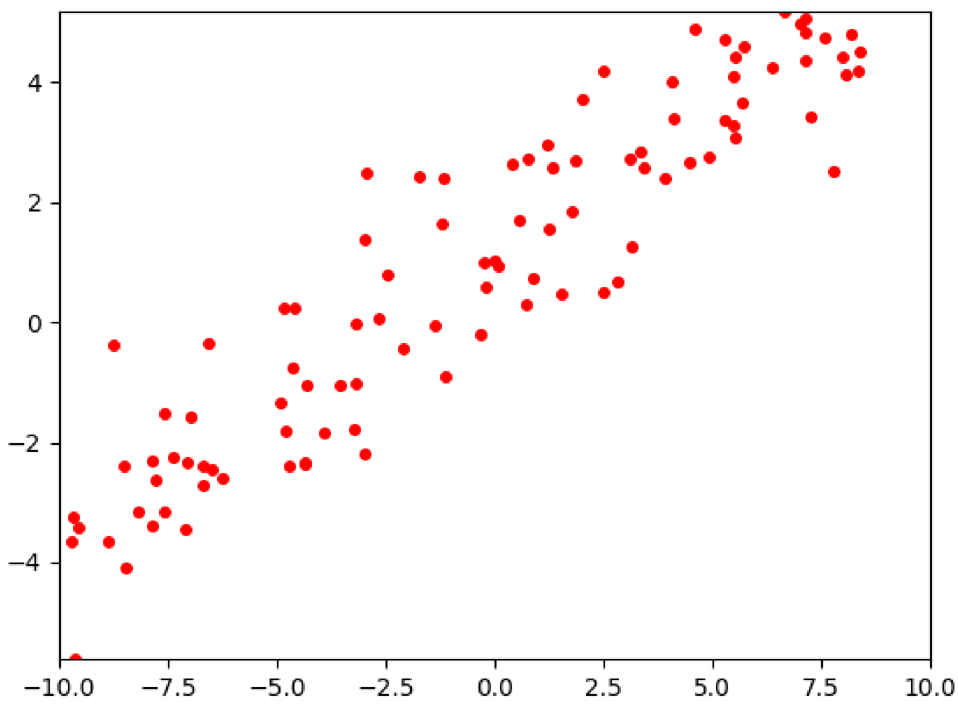
that minimizes the sum of square distances between the predicted `y` and the `y` we observed in our dataset:

$$\mathcal{L}(\ ) = \|\mathbf{y} - X\ \|^2$$

We were able to find a global minimum to this loss function but we will try to apply gradient descent to find that same solution.

a) Implement the `loss` function to complete the code and plot the loss as a function of beta.

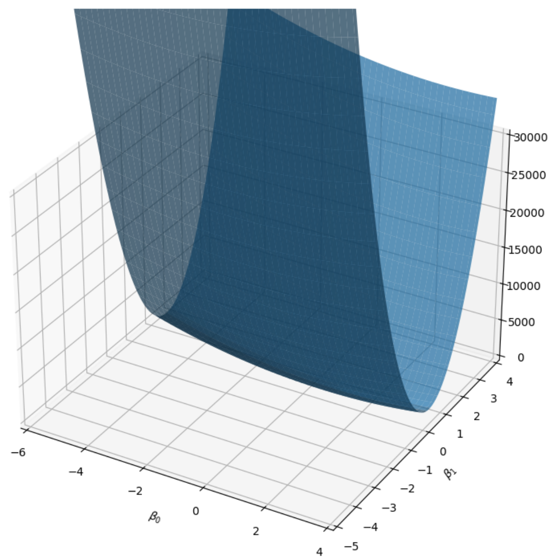
[12]:
```
%matplotlib widget
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt

beta = np.array([ 1 , .5 ])
xlin = -10.0 + 20.0 * np.random.random(100)
X = np.column_stack([np.ones((len(xlin), 1)), xlin])
y = beta[0]+(beta[1]*xlin)+np.random.randn(100)

fig, ax = plt.subplots()
ax.plot(xlin, y,'ro',markersize=4)
ax.set_xlim(-10, 10)
ax.set_ylim(min(y), max(y))
plt.show()
```



[13]:

```
b0 = np.arange(-5, 4, 0.1)
b1 = np.arange(-5, 4, 0.1)
b0, b1 = np.meshgrid(b0, b1)

def loss(X, y, beta):
    predictions = X.dot(beta)
    errors = y - predictions
    return np.sum(errors ** 2)

def get_cost(B0, B1):
    res = []
    for b0, b1 in zip(B0, B1):
        line = []
        for i in range(len(b0)):
            beta = np.array([b0[i], b1[i]])
            line.append(loss(X, y, beta))
        res.append(line)
    return np.array(res)

cost = get_cost(b0, b1)
```

```python
# Creating figure
fig = plt.figure(figsize =(14, 9))
ax = plt.axes(projection ='3d')
ax.set_xlim(-6, 4)
ax.set_xlabel(r'$\beta_0$')
ax.set_ylabel(r'$\beta_1$')
ax.set_ylim(-5, 4)
ax.set_zlim(0, 30000)

# Creating plot
ax.plot_surface(b0, b1, cost, alpha=.7)

# show plot
plt.show()
```



Since the loss is

$$\mathcal{L}(\ ) = \|\mathbf{y} - X\ \|^2 = \beta^T X^T X\beta - 2\ ^T X^T\mathbf{y} + \mathbf{y}^T\mathbf{y}$$

The gradient is

$$\nabla_\beta \mathcal{L}(\ ) = 2X^T X\beta - 2X^T\mathbf{y}$$

b) Implement the gradient function below and complete the gradient descent algorithm

[14]:

```
import numpy as np
from PIL import Image as im
import matplotlib.pyplot as plt

TEMPFILE = "temp.png"

def snap(betas, losses):
    # Creating figure
    fig = plt.figure(figsize =(14, 9))
    ax = plt.axes(projection ='3d')
    ax.view_init(20, -20)
    ax.set_xlim(-5, 4)
    ax.set_xlabel(r'$\beta_0$')
    ax.set_ylabel(r'$\beta_1$')
    ax.set_ylim(-5, 4)
    ax.set_zlim(0, 30000)

    # Creating plot
    ax.plot_surface(b0, b1, cost, color='b', alpha=.7)
    ax.plot(np.array(betas)[:,0], np.array(betas)[:,1], losses, 'o-', c='r',␣
 ↪markersize=10, zorder=10)
    fig.savefig(TEMPFILE)
    plt.close()
    return im.fromarray(np.asarray(im.open(TEMPFILE)))


def gradient(X, y, beta):
    return 2 * X.T.dot(X.dot(beta) - y)


def gradient_descent(X, y, beta_hat, learning_rate, epochs, images):
    losses = [loss(X, y, beta_hat)]
    betas = [beta_hat]

    for _ in range(epochs):
        images.append(snap(betas, losses))
        beta_hat = beta_hat - learning_rate * gradient(X, y, beta_hat)

        losses.append(loss(X, y, beta_hat))
        betas.append(beta_hat)

    return np.array(betas), np.array(losses)


beta_start = np.array([-5, -2])
learning_rate = 0.0002 # try .0005
images = []
```

```
betas, losses = gradient_descent(X, y, beta_start, learning_rate, 10, images)

images[0].save(
    'gd.gif',
    optimize=False,
    save_all=True,
    append_images=images[1:],
    loop=0,
    duration=500
)
```

c) Use the code above to create an animation of the linear model learned at every epoch.

[15]:
```
def snap_model(beta):
    xplot = np.linspace(-10,10,50)
    yestplot = beta[0] + beta[1] * xplot
    fig, ax = plt.subplots()
    ax.plot(xplot, yestplot,'b-',lw=2)
    ax.plot(xlin, y,'ro',markersize=4)
    ax.set_xlim(-10, 10)
    ax.set_ylim(min(y), max(y))
    fig.savefig(TEMPFILE)
    plt.close()
    return im.fromarray(np.asarray(im.open(TEMPFILE)))



def gradient_descent(X, y, beta_hat, learning_rate, epochs, images):
    losses = [loss(X, y, beta_hat)]
    betas = [beta_hat]

    for _ in range(epochs):
        images.append(snap_model(beta_hat))
        beta_hat = beta_hat - learning_rate * gradient(X, y, beta_hat)

        losses.append(loss(X, y, beta_hat))
        betas.append(beta_hat)

    return np.array(betas), np.array(losses)



images = []
betas, losses = gradient_descent(X, y, beta_start, learning_rate, 100, images)

images[0].save(
    'model.gif',
    optimize=False,
    save_all=True,
```

```
    append_images=images[1:],
    loop=0,
    duration=200
)
```

In logistic regression, the `loss` is the negative log-likelihood

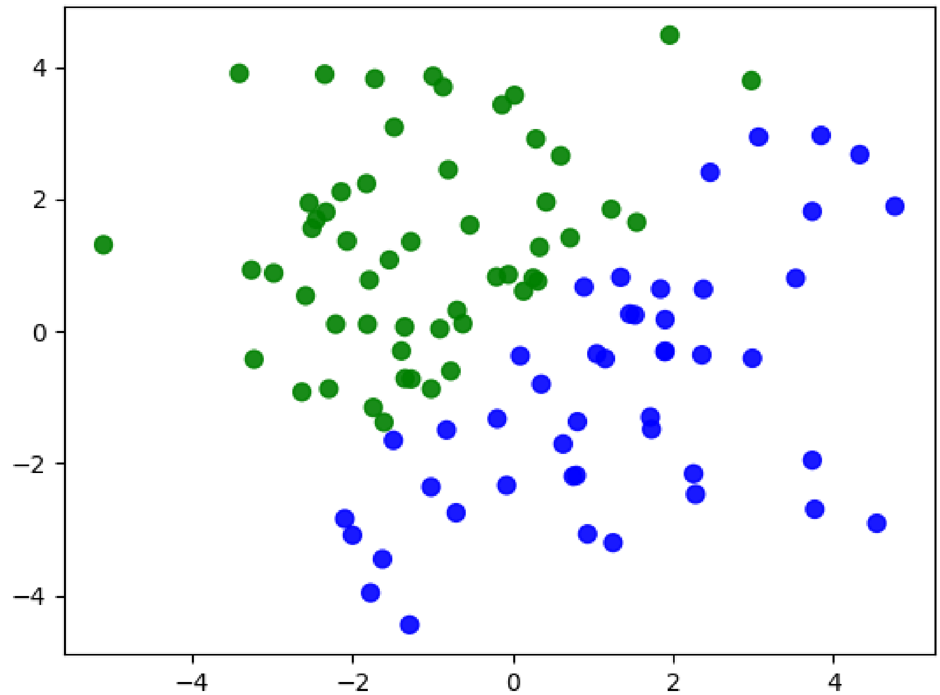
$$l( ) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\sigma(x_i\beta)) + (1-y_i)\log(1-\sigma(x_i\beta))$$

the gradient of which is:

$$\nabla_\beta l( ) = \frac{1}{N}\sum_{i=1}^{N} x_i(y_i - \sigma(x_i\beta))$$

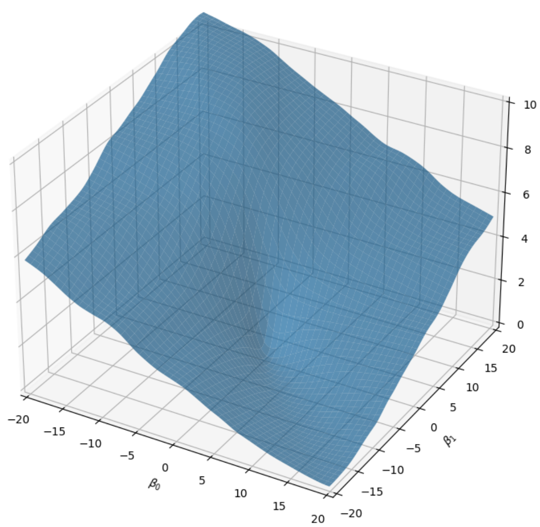
d) Plot the loss as a function of b.

[16]:
```
%matplotlib widget
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
import sklearn.datasets as datasets

centers = [[0, 0]]
t, _ = datasets.make_blobs(n_samples=100, centers=centers, cluster_std=2,␣
 ↪random_state=0)

# LINE
def generate_line_data():
    # create some space between the classes
    X = t
    Y = np.array([1 if x[0] - x[1] >= 0 else 0 for x in X])
    return X, Y

X, y = generate_line_data()

cs = np.array([x for x in 'gb'])
fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], color=cs[y].tolist(), s=50, alpha=0.9)
plt.show()
```



[17]:
```
b0 = np.arange(-20, 20, 0.1)
b1 = np.arange(-20, 20, 0.1)
b0, b1 = np.meshgrid(b0, b1)


def sigmoid(x):
    e = np.exp(x)
    return e / (1 + e)


def loss(X, y, beta):
    z = np.dot(X, beta)
    pred_prob = sigmoid(z)
    return -np.mean(y * np.log(pred_prob + 1e-5) + (1 - y) * np.log(1 -␣
 ↪pred_prob + 1e-5))


def get_cost(B0, B1):
    res = []
    for b0, b1 in zip(B0, B1):
        line = []
```

```
        for i in range(len(b0)):
            beta = np.array([b0[i], b1[i]])
            line.append(loss(X, y, beta))
        res.append(line)
    return np.array(res)

cost = get_cost(b0, b1)

# Creating figure
fig = plt.figure(figsize =(14, 9))
ax = plt.axes(projection ='3d')
ax.set_xlim(-20, 20)
ax.set_xlabel(r'$\beta_0$')
ax.set_ylabel(r'$\beta_1$')
ax.set_ylim(-20, 20)
ax.set_zlim(0, 10)

# Creating plot
ax.plot_surface(b0, b1, cost, alpha=.7)

# show plot
plt.show()
```



e) Plot the loss at each iteration of the gradient descent algorithm.

[18]:

```
import numpy as np
from PIL import Image as im
import matplotlib.pyplot as plt

TEMPFILE = "temp.png"

def snap(betas, losses):
    # Creating figure
    fig = plt.figure(figsize =(14, 9))
    ax = plt.axes(projection ='3d')
    ax.view_init(10, 10)
    ax.set_xlabel(r'$\beta_0$')
    ax.set_ylabel(r'$\beta_1$')
    ax.set_ylim(-20, 20)
    ax.set_zlim(0, 10)

    # Creating plot
    ax.plot_surface(b0, b1, cost, color='b', alpha=.7)
    ax.plot(np.array(betas)[:,0], np.array(betas)[:,1], losses, 'o-', c='r',␣
↪markersize=10, zorder=10)
    fig.savefig(TEMPFILE)
    plt.close()
    return im.fromarray(np.asarray(im.open(TEMPFILE)))



def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def gradient(X, y, beta):
    z = X.dot(beta)
    y_pred = sigmoid(z)
    gradient = X.T.dot(y_pred - y)
    return gradient


def gradient_descent(X, y, beta_hat, learning_rate, epochs, images):
    losses = [loss(X, y, beta_hat)]
    betas = [beta_hat]

    for _ in range(epochs):
        images.append(snap(betas, losses))
        beta_hat = beta_hat - learning_rate * gradient(X, y, beta_hat)

        losses.append(loss(X, y, beta_hat))
        betas.append(beta_hat)
```

```
    return np.array(betas), np.array(losses)


beta_start = np.array([-5, -2])
learning_rate = 0.1
images = []
betas, losses = gradient_descent(X, y, beta_start, learning_rate, 10, images)

images[0].save(
    'gd_logit.gif',
    optimize=False,
    save_all=True,
    append_images=images[1:],
    loop=0,
    duration=500
)
```

f) Create an animation of the logistic regression fit at every epoch.

[19]:
```
beta_start = np.array([-5, -2])
learning_rate = 0.1
images = []
betas, losses = gradient_descent(X, y, beta_start, learning_rate, 10, images)

images[0].save(
    'gd_logit.gif',
    optimize=False,
    save_all=True,
    append_images=images[1:],
    loop=0,
    duration=500
)
```

g) Modify the above code to evaluate the gradient on a random batch of the data. Overlay the true loss curve and the approximation of the loss in your animation.

[20]:
```
import numpy as np
from sklearn.utils import shuffle

def mini_batch_gradient_descent(X, y, beta_hat, learning_rate, epochs,
 ↪batch_size, images):
    full_losses = [loss(X, y, beta_hat)]
    batch_losses = []
    betas = [beta_hat.copy()]

    for epoch in range(epochs):
        X, y = shuffle(X, y)  # Shuffle the dataset
```

```
        for i in range(0, X.shape[0], batch_size):
            X_batch = X[i:i + batch_size]
            y_batch = y[i:i + batch_size]

            # Calculate gradient using the batch
            grad = gradient(X_batch, y_batch, beta_hat)
            beta_hat -= learning_rate * grad
            betas.append(beta_hat.copy())

            # Calculate and store batch loss for visualization
            batch_current_loss = loss(X_batch, y_batch, beta_hat)
            batch_losses.append(batch_current_loss)

            # Calculate full dataset loss after the update for true trajectory visualization
            full_current_loss = loss(X, y, beta_hat)
            full_losses.append(full_current_loss)

            # Snapshot current state for animation
            images.append(snap(betas, full_losses, batch_losses))

    return np.array(betas), np.array(full_losses), np.array(batch_losses)


def snap(betas, full_losses, batch_losses):
    min_length = min(len(betas), len(full_losses), len(batch_losses))
    betas = np.array(betas)[:min_length]
    full_losses = np.array(full_losses)[:min_length]
    batch_losses = np.array(batch_losses)[:min_length]

    fig = plt.figure(figsize=(14, 9))
    ax = plt.axes(projection='3d')
    ax.view_init(10, 10)
    ax.set_xlabel(r'$\beta_0$')
    ax.set_ylabel(r'$\beta_1$')
    ax.set_ylim(-20, 20)
    ax.set_zlim(0, max(max(full_losses), max(batch_losses), 1))

    ax.plot_surface(b0, b1, cost, color='blue', alpha=.7)
    ax.plot(betas[:, 0], betas[:, 1], full_losses, 'o-', color='red', markersize=10, zorder=10)
    ax.plot(betas[:, 0], betas[:, 1], batch_losses, 'o-', color='green', markersize=5, zorder=5)

    fig.savefig(TEMPFILE)
    plt.close()
    return im.fromarray(np.asarray(im.open(TEMPFILE)))
```

```
beta_start = np.array([-5, -2], dtype=np.float64)
learning_rate = 0.1
images = []

betas, full_losses, batch_losses = mini_batch_gradient_descent(X, y,␣
↪beta_start, learning_rate, 10, 10, images)

images[0].save(
    'mini_batch_gd_logit.gif',
    optimize=False,
    save_all=True,
    append_images=images[1:],
    loop=0,
    duration=500
)
```

h) Below is a sandox where you can get intuition about how to tune gradient descent parameters:

[21]:
```
import numpy as np
from PIL import Image as im
import matplotlib.pyplot as plt

TEMPFILE = "temp.png"

def snap(x, y, pts, losses, grad):
    fig = plt.figure(figsize =(14, 9))
    ax = plt.axes(projection ='3d')
    ax.view_init(20, -20)
    ax.plot_surface(x, y, loss(np.array([x, y])), color='r', alpha=.4)
    ax.plot(np.array(pts)[:,0], np.array(pts)[:,1], losses, 'o-', c='b',␣
↪markersize=10, zorder=10)
    ax.plot(np.array(pts)[-1,0], np.array(pts)[-1,1], -1, 'o-', c='b', alpha=.
↪5, markersize=7, zorder=10)

    # Plot Gradient Vector
    X, Y, Z = [pts[-1][0]], [pts[-1][1]], [-1]
    U, V, W = [-grad[0]], [-grad[1]], [0]
    ax.quiver(X, Y, Z, U, V, W, color='g')
    fig.savefig(TEMPFILE)
    plt.close()
    return im.fromarray(np.asarray(im.open(TEMPFILE)))

def loss(x):
    return np.sin(sum(x**2)) # change this
```

```python
def gradient(x):
    return 2 * x * np.cos(sum(x**2)) # change this

def gradient_descent(x, y, init, learning_rate, epochs):
    images, losses, pts = [], [loss(init)], [init]
    for _ in range(epochs):
        grad = gradient(init)
        images.append(snap(x, y, pts, losses, grad))
        init = init - learning_rate * grad
        losses.append(loss(init))
        pts.append(init)
    return images

init = np.array([-.5, -.5]) # change this
learning_rate = 1.394 # change this
x, y = np.meshgrid(np.arange(-2, 2, 0.1), np.arange(-2, 2, 0.1)) # change this
images = gradient_descent(x, y, init, learning_rate, 12)

images[0].save(
    'gradient_descent.gif',
    optimize=False,
    save_all=True,
    append_images=images[1:],
    loop=0,
    duration=500
)
```