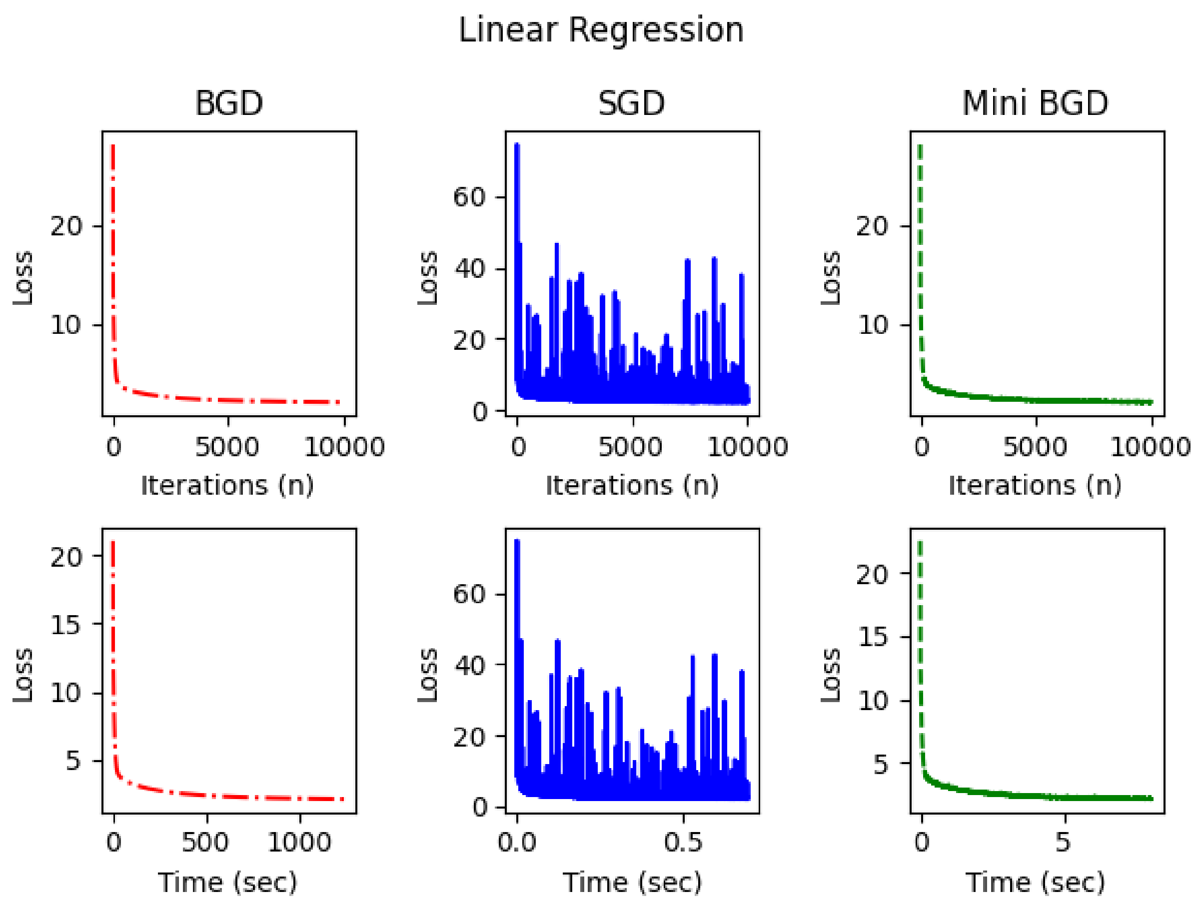
Linear Regression

1) Code

```
def linear_function(theta_, x_) -> float:
  return np.inner(theta_, x_)

def linear_loss_all(theta_, X_, y_, N, lambd) -> float:
  sum = 0
  for i in range(N):
    new_x_ = np.append([1], X_[i])
    sum = sum + (linear_function(theta_, new_x_)-y_[i])**2
  return 0.5 * sum


'''
The rest of the functions are the same for both regressions.
'''
def find_graident(theta_, x_, y, function, lambd):
  new_x_ = np.append([1], x_)
  return (function(theta_, new_x_)-y) * new_x_ + (lambd * theta_)

def mini_batch_gradient_decent(
    theta_:np.ndarray,
    X_:np.ndarray,
    y_:np.ndarray,
    N:int,
    learning_rate:float,
    batch_size:int,
    function,
    lambd:float
  ):
  batch = random.sample(range(N), k=batch_size)
  gradient = sum([ find_graident(theta_, X_[i], y_[i], function, lambd)
                for i in batch])
  new_theta_ = theta_ - ((learning_rate * gradient)/batch_size)
  return new_theta_

def gradient_decent(
  X_:np.ndarray,
  y_:np.ndarray,
  N:int,

  iterations:int,
  learning_rate:float,
  batch_size:int,
  function:str,
  lambd:float
  ):
```

```
function_types = {"linear":(linear_function, linear_loss_all), "logistic":(si
f, j = function_types[function]
theta_ = np.array([0.001 for i in range(X_.shape[1]+1)])
losses = []
times = []
losses.append(j(theta_, X_, y_, N, lambd)/N)
for i in range(iterations):
  start = time.time()
  theta_ = mini_batch_gradient_decent(theta_, X_, y_, N, learning_rate, batch
  end = time.time()
  losses.append(j(theta_, X_, y_, N, lambd)/N)
  times.append(end - start)
return (losses, times)
```

2) Plots



3) Findings relative to the convergence theory

I found that a bad guess for the inital weights/$\underline{\theta}$ would cause my regression to never converge. After some testing, I found that setting 0.001 for all θ_i, the result of the regression would converge.

In addtion to the intial weights, the learning rate/α heavily impacted whether the regression would converge. Setting it to 0.00000005 helped the results converge the fastest without diverging.

Logistic Regression

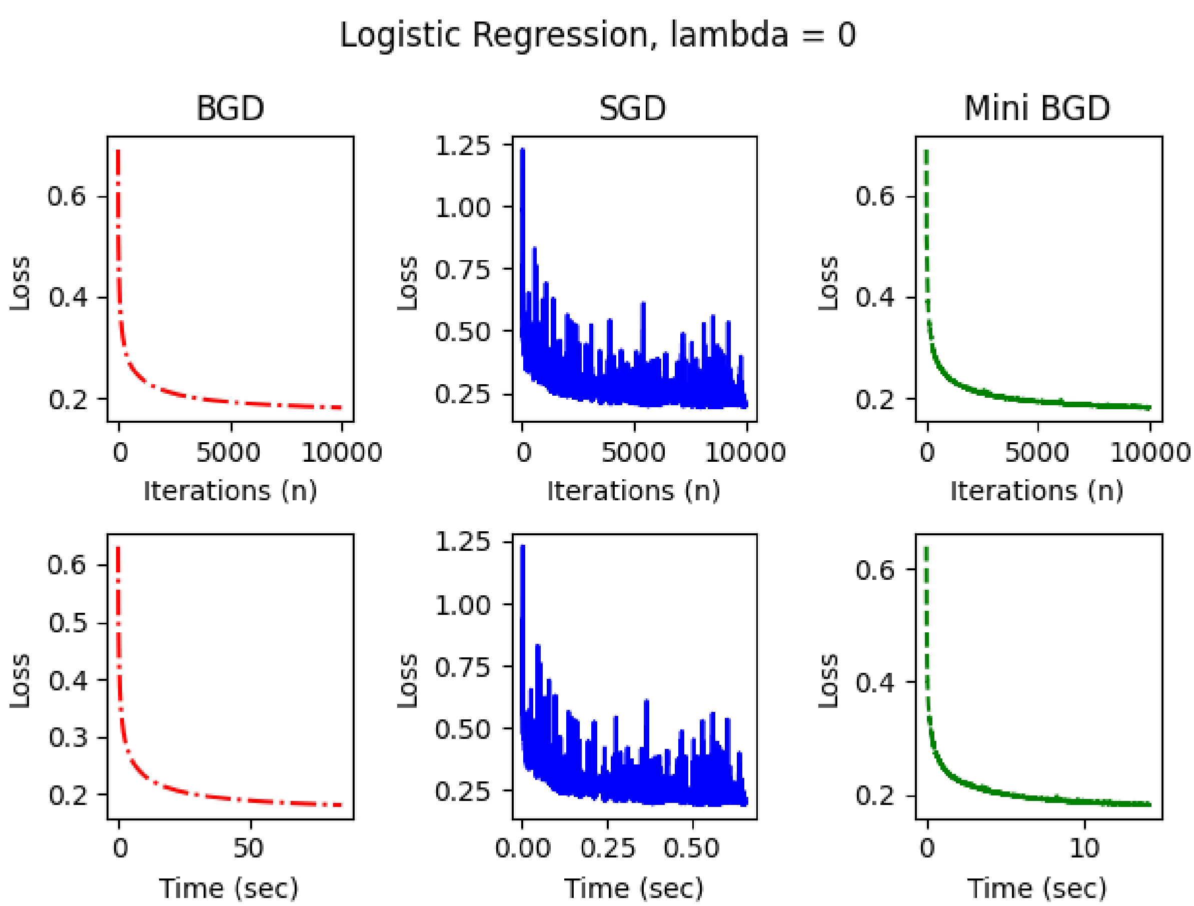
1) Code

```
def sigmoid_function(theta_, x_) -> float:
  return np.power(1 + math.exp(-1 * np.inner(theta_, x_)), -1)

def logistic_loss_all(theta_, X_, y_, N, lambd) -> float:
  sum = 0
  for i in range(N):
    reg = 0
    for theta in theta_:
      reg += theta**2
    new_x_ = np.append([1], X_[i])
    sum = sum - (
      y_[i] * math.log(sigmoid_function(theta_, new_x_)) +
      (1-y_[i]) * math.log(1-sigmoid_function(theta_, new_x_))
    ) + (lambd * reg)
  return sum

'''
The rest of the functions are the same for both regressions.
'''
def find_graident(theta_, x_, y, function, lambd):
  new_x_ = np.append([1], x_)
  return (function(theta_, new_x_)-y) * new_x_ + (lambd * theta_)

def mini_batch_gradient_decent(
    theta_:np.ndarray,
    X_:np.ndarray,
    y_:np.ndarray,
    N:int,
    learning_rate:float,
    batch_size:int,
    function,
    lambd:float
  ):
  batch = random.sample(range(N), k=batch_size)
  gradient = sum([ find_graident(theta_, X_[i], y_[i], function, lambd)
                for i in batch])
  new_theta_ = theta_ - ((learning_rate * gradient)/batch_size)
  return new_theta_


def gradient_decent(
  X_:np.ndarray,
  y_:np.ndarray,
  N:int,
  iterations:int,
```

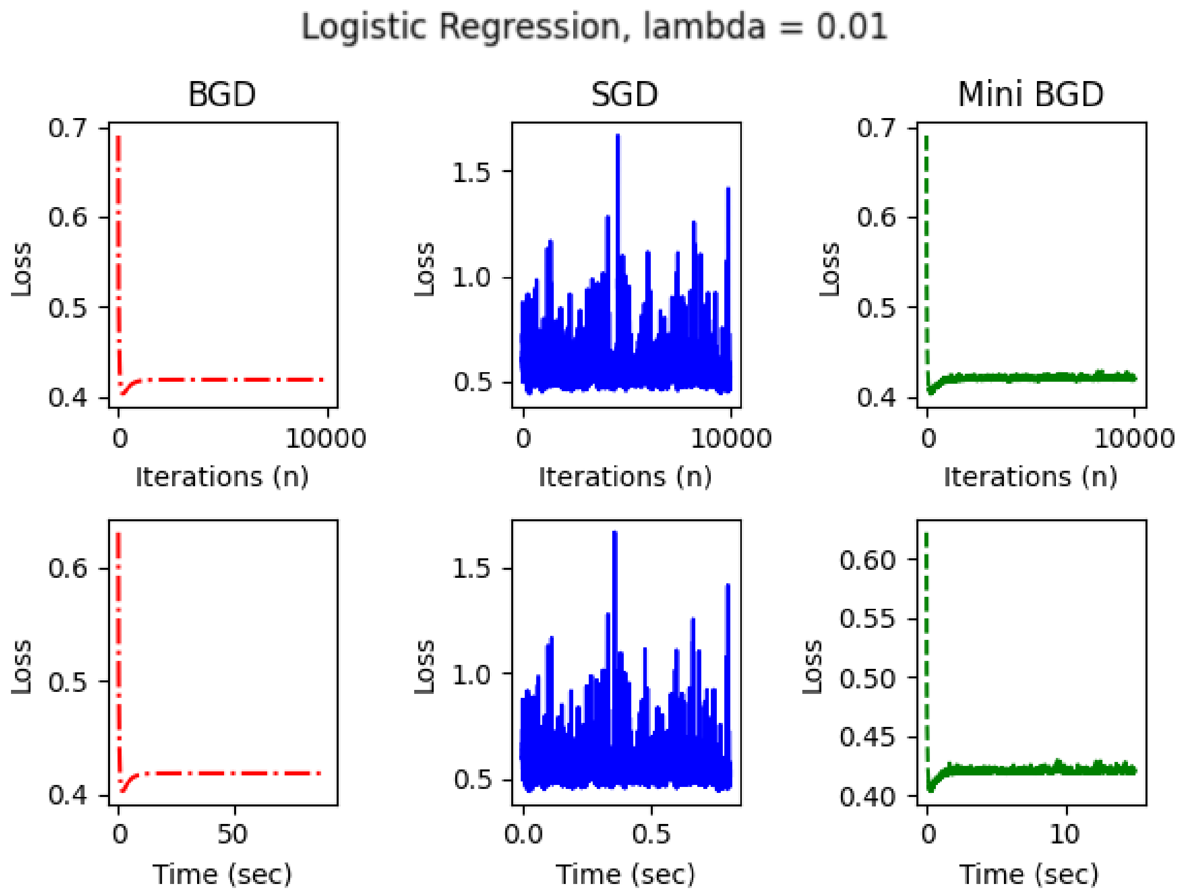
```
    learning_rate:float,
    batch_size:int,
    function:str,
    lambd:float
    ):
    function_types = {"linear":(linear_function, linear_loss_all), "logistic":(si
    f, j = function_types[function]
    theta_ = np.array([0.001 for i in range(X_.shape[1]+1)])
    losses = []
    times = []
    losses.append(j(theta_, X_, y_, N, lambd)/N)
    for i in range(iterations):
      start = time.time()
      theta_ = mini_batch_gradient_decent(theta_, X_, y_, N, learning_rate, batch
      end = time.time()
      losses.append(j(theta_, X_, y_, N, lambd)/N)
      times.append(end - start)
    return (losses, times)
```

2) Plots

2a)



2a)



3) Findings relative to the convergence theory

Like linear regression, I found that a bad guess for the inital weights/$\underline{\theta}$ would cause my regression to never converge. After some testing, I found that setting 0.001 for all θ_i, the result of the regression would converge.

In addtion to the intial weights, the learning rate/α heavily impacted whether the regression would converge. The results from using 0.00000005 from linear regression showed that logistic regression converged slowly. Setting α to 0.2, sped up the process and did not diverge.

The rest of the code:

This contains imports, misc helper functions, formatting data, and displaying results of regressions.

```
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
import time

def generate_ionosphere_data():
  sheet = pd.read_csv("ionosphere.csv")
  X_ = sheet.to_numpy()
```

```
  y_ = X_[:,34]
  for i in range(y_.shape[0]):
    if y_[i] == "g":
      y_[i] = 1
    else:
      y_[i] = 0
  X_ = np.delete(X_, np.s_[34:35], axis=1)
  return (X_, y_)

def generate_air_quality_data():
  sheet = pd.read_excel("AirQualityUCI.xlsx")
  X_ = sheet.to_numpy()
  X_ = np.delete(X_, np.s_[0:2], axis=1)

  for i in range(X_.shape[1]):
    idx_to_change = []
    sum_ = 0
    for j in range(X_.shape[0]):
      val = X_[j][i]
      if (val == -200):
        idx_to_change.append(j)
      else:
        sum_ += val

    avg = sum_ / (X_.shape[0]-len(idx_to_change))
    for j in range(len(idx_to_change)):
      X_[idx_to_change[j]][i] = avg

  y_ = X_[:,3]
  X_ = np.delete(X_, np.s_[3:4], axis=1)
  return (X_, y_)
def cumulate(times) -> None:
  for i in range(1, len(times), 1):
    times[i] = times[i] + times[i-1]

def run_regression(func:str):
  n = 10000
  itr = range(n+1)
  if (func == "linear"):
    learning_rate = 0.00000005
    X_, y_ = generate_air_quality_data()
  elif (func == "logistic"):
    learning_rate = 0.2
    X_, y_ = generate_ionosphere_data()
  else:
    exit(1)
  lambd = 0.01
  N = X_.shape[0]

  figure, axis = plt.subplots(2,3)

  losses, times = gradient_decent(X_, y_, N, n, learning_rate, N, func, lambd)
```

```python
    cumulate(times)
    axis[0, 0].set_title("BGD")
    axis[0, 0].plot(itr, losses, 'r-.', label='BGD')
    axis[0, 0].set(xlabel="Iterations (n)", ylabel="Loss")
    axis[1, 0].plot(times, losses[1:], 'r-.', label='BGD')
    axis[1, 0].set(xlabel="Time (sec)", ylabel="Loss")

    losses, times = gradient_decent(X_, y_, N, n, learning_rate, 1, func, lambd)
    cumulate(times)
    axis[0, 1].set_title("SGD")
    axis[0, 1].plot(itr, losses, 'b-', label='SGD')

    axis[0, 1].set(xlabel="Iterations (n)", ylabel="Loss")
    axis[1, 1].plot(times, losses[1:], 'b-', label='SGD')
    axis[1, 1].set(xlabel="Time (sec)", ylabel="Loss")

    losses, times = gradient_decent(X_, y_, N, n, learning_rate, 50, func, lambd)
    cumulate(times)
    axis[0, 2].set_title("Mini BGD")
    axis[0, 2].plot(itr, losses, 'g--', label='Mini BGD')
    axis[0, 2].set(xlabel="Iterations (n)", ylabel="Loss")
    axis[1, 2].plot(times, losses[1:], 'g--', label='Mini BGD')
    axis[1, 2].set(xlabel="Time (sec)", ylabel="Loss")

    figure.suptitle("Logistic Regression, lambda = 0.01")
    figure.tight_layout()
    plt.show()

  def main():
    run_regression("linear")
    run_regression("logistic")
```

The entire program can be found on Github