COMP9417 - Machine Learning
Homework 2: Newton's Method and Mean Squared Error of Estimators

Introduction In homework 1, we considered Gradient Descent (and coordinate descent) for minimizing a regularized loss function. In this homework, we consider an alternative method known as Newton's algorithm. We will first run Newton's algorithm on a simple toy problem, and then implement it from scratch on a real life classification problem. We will also dig deeper into the idea of estimation and comparing estimators based on bias, variance and MSE.
Points Allocation There are a total of 28 marks.

- Question 1 a): 1 mark
- Question 1 b): 1 mark
- Question 1 c): 1 mark
- Question 2 a): 2 marks
- Question 2 b): 2 marks
- Question 2 c): 5 marks
- Question 2 d): 1 mark
- Question 2 e): 3 marks
- Question 2 f): 5 marks
- Question 2 g): 1 mark
- Question 3 a): 3 marks
- Question 3 b): 2 marks
- Question 3 c): 1 mark

What to Submit

- A **single PDF** file which contains solutions to each question. For each question, provide your solution in the form of text and requested plots. For some questions you will be requested to provide screen shots of code used to generate your answer — only include these when they are explicitly asked for.
- **.py file(s) containing all code you used for the project, which should be provided in a separate .zip file.** This code must match the code provided in the report.

- You may be deducted points for not following these instructions.
- You may be deducted points for poorly presented/formatted work. Please be neat and make your solutions clear. Start each question on a new page if necessary.
- You **cannot** submit a Jupyter notebook; this will receive a mark of zero. This does not stop you from developing your code in a notebook and then copying it into a .py file though, or using a tool such as **nbconvert** or similar.
- We will set up a Moodle forum for questions about this homework. Please read the existing questions before posting new questions. Please do some basic research online before posting questions. Please only post clarification questions. Any questions deemed to be *fishing* for answers will be ignored and/or deleted.
- Please check Moodle announcements for updates to this spec. It is your responsibility to check for announcements about the spec.
- Please complete your homework on your own, do not discuss your solution with other people in the course. General discussion of the problems is fine, but you must write out your own solution and acknowledge if you discussed any of the problems in your submission (including their name(s) and zID).
- As usual, we monitor all online forums such as Chegg, StackExchange, etc. Posting homework questions on these site is equivalent to plagiarism and will result in a case of academic misconduct.

When and Where to Submit

- Due date: Week 7, Monday **July 11th**, 2022 by **5pm**. Please note that the forum will not be actively monitored on weekends.
- Late submissions will incur a penalty of 5% per day **from the maximum achievable grade**. For example, if you achieve a grade of 80/100 but you submitted 3 days late, then your final grade will be $80 - 3 \times 5 = 65$. Submissions that are more than 5 days late will receive a mark of zero.
- Submission must be done through Moodle, no exceptions.

Question 1. Introduction to Newton's Method

Note: throughout this question do not use any existing implementations of any of the algorithms discussed unless explicitly asked to in the question. Using existing implementations can result in a grade of zero for the entire question. In homework 1 we studied gradient descent (GD), which is usually referred to as a first order method. Here, we study an alternative algorithm known as Newton's algorithm, which is generally referred to as a second order method. Roughly speaking, a second order method makes use of both first and second derivatives. Generally, second order methods are much more accurate than first order ones. Given a twice differentiable function $g : \mathbb{R} \to \mathbb{R}$, Newton's method generates a sequence $\{x^{(k)}\}$ iteratively according to the following update rule:

$$x^{(k+1)} = x^{(k)} - \frac{g'(x^{(k)})}{g''(x^{(k)})}, \qquad k = 0, 1, 2, \ldots, \tag{1}$$

For example, consider the function $g(x) = \frac{1}{2}x^2 - \sin(x)$ with initial guess $x^{(0)} = 0$. Then

$$g'(x) = x - \cos(x), \qquad \text{and} \qquad g''(x) = 1 + \sin(x),$$

and so we have the following iterations:

$$\begin{aligned}
x^{(1)} &= x^{(0)} - \frac{x^{(0)} - \cos(x^0)}{1 + \sin(x^{(0)})} = 0 - \frac{0 - \cos(0)}{1 + \sin(0)} = 1 \\
x^{(2)} &= x^{(1)} - \frac{x^{(1)} - \cos(x^1)}{1 + \sin(x^{(1)})} = 1 - \frac{1 - \cos(1)}{1 + \sin(1)} = 0.750363867840244 \\
x^{(3)} &= 0.739112890911362 \\
&\vdots
\end{aligned}$$

and this continues until we terminate the algorithm (as a quick exercise for your own benefit, code this up, plot the function and each of the iterates). We note here that in practice, we often use a different update called the *dampened* Newton method, defined by:

$$x^{(k+1)} = x^{(k)} - \alpha\frac{g'(x_k)}{g''(x_k)}, \qquad k = 0, 1, 2, \ldots. \tag{2}$$

Here, as in the case of GD, the step size α has the effect of 'dampening' the update.

(a) Consider the twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$. The Newton steps in this case are now:

$$x^{(k+1)} = x^{(k)} - (H(x^{(k)}))^{-1}\nabla f(x^{(k)}), \qquad k = 0, 1, 2, \ldots, \tag{3}$$

where $H(x) = \nabla^2 f(x)$ is the Hessian of f. Explain heuristically (in a couple of sentences) how the above formula is a generalization of equation (1) to functions with vector inputs. *what to submit: Some commentary*

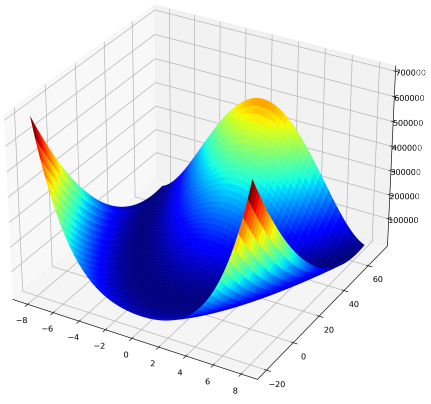
> **Solution:**
> Some justification that the gradient is the analog of the first derivative, the Hessian is the analog of the second derivative and that taking inverse is the analog of division.

(b) Consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by

$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2.$$

Create a 3D plot of the function using `mplot3d` (see lab0 for example). Further, compute the gradient and Hessian of f. *what to submit: A single plot, the code used to generate the plot, the gradient and Hessian calculated along with all working. Add a copy of the code to solutions.py*

Solution:



We have

$$\nabla f(x, y) = \begin{bmatrix} -400xy + 400x^3 + 2x - 2 \\ 200y - 200x^2 \end{bmatrix}, \qquad \nabla^2 f(x, y) = \begin{bmatrix} -400y + 1200x^2 + 2 & -400x \\ -400x & 200 \end{bmatrix}.$$

The code used in this section:

```
from mpl_toolkits import mplot3d
x = np.linspace(-5, 5, 1000)
y = np.linspace(-5, 5, 1000)
X, Y = np.meshgrid(x,y)


def func(x,y):
    return 100 * (y-x**2)**2 + (1-x)**2

Z = func(X, Y)

fig = plt.figure(figsize=(10,10))
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
plt.savefig("figures/1b.png", dpi=400)
plt.show()

```

(c) Using `NumPy` only, implement the (undampened) Newton algorithm to find the minimizer of the function in the previous part, using an initial guess of $x^{(0)} = (-1.2, 1)^T$. Terminate the algorithm when $\left\|\nabla f(x^{(k)})\right\|_2 \leq 10^{-6}$. Report the values of $x^{(k)}$ for $k = 0, 1, \ldots, K$ where K is your final iteration. *what to submit: your iterations, and a screen shot of your code. Add a copy of the code to solutions.py*

Solution:

The global minimizer is $(1, 1)$, the iterations are:

$$\begin{aligned}
&\text{iteration } 0, x_0 = [-1.2, 1.] \\
&\text{iteration } 1, x_1 = [-1.1752809, 1.38067416] \\
&\text{iteration } 2, x_2 = [0.76311487, -3.17503385] \\
&\text{iteration } 3, x_3 = [0.76342968, 0.58282478] \\
&\text{iteration } 4, x_4 = [0.99999531, 0.94402732] \\
&\text{iteration } 5, x_5 = [0.9999957, 0.99999139]
\end{aligned}$$

The code used in this section:

```
def grad_func(x):
    return np.array([-400*x[0]*x[1] + 400 * x[0]**3 + 2*x[0] - 2, 200 * x[1] - 200 * x
    [0]**2])

def Hessian(x):
    H = np.empty((2,2))
    H[0,0] = -400 * x[1] + 1200 * x[0]**2 +2
    H[0,1] = -400 * x[0]
    H[1,0] = -400 * x[0]
    H[1,1] = 200
    return H

def sqd_norm(b):
    return np.linalg.norm(b, ord=2)**2

x = np.array([-1.2, 1])
k = 0
tol = 10**(-6)
while sqd_norm(grad_func(x)) > tol:
    print(f"iteration {k}, x_{k} = {x}")
    x = x - np.linalg.inv(Hessian(x)) @ grad_func(x)
    k += 1

print(f"iteration {k}, x_{k} = {x}")


```

Question 2. Solving Logistic Regression Numerically

Note: throughout this question do not use any existing implementations of any of the algorithms discussed unless explicitly asked to do so in the question. Using existing implementations can result in a grade of zero for the entire question. In this question we will compare gradient descent and Newton's algorithm for solving the logistic regression problem. Recall that in logistic regresion,

our goal is to minimize the log-loss, also referred to as the cross entropy loss. For an intercept $\beta_0 \in \mathbb{R}$, parameter vector $\beta = (\beta_1, \ldots, \beta_p)^T \in \mathbb{R}^p$, target $y_i \in \{0, 1\}$, and feature vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})^T \in \mathbb{R}^p$ for $i = 1, \ldots, n$, the (ℓ_2-regularized) log-loss that we will work with is:

$$L(\beta_0, \beta) = \frac{1}{2} \|\beta\|_2^2 + \frac{\lambda}{n} \sum_{i=1}^{n} \left[y_i \ln \left(\frac{1}{\sigma(\beta_0 + \beta^T x_i)} \right) + (1 - y_i) \ln \left(\frac{1}{1 - \sigma(\beta_0 + \beta^T x_i)} \right) \right],$$

where $\sigma(z) = (1 + e^{-z})^{-1}$ is the logistic sigmoid, and λ is a hyper-parameter that controls the amount of regularization. Note that you are provided with an implementation of this loss in `helper.py`.

(a) Suppose that you were going to solve this problem using gradient descent and chose to update each coordinate individually. Derive gradient descent updates for each of the components $\beta_0, \beta_1, \ldots, \beta_p$ for step size α and regularization parameter λ. That is, derive explicit expressions for the terms † in the following:

$$\begin{aligned}
\beta_0^{(k)} &= \beta_0^{(k-1)} - \alpha \times \dagger \\
\beta_1^{(k)} &= \beta_1^{(k-1)} - \alpha \times \dagger \\
\beta_2^{(k)} &= \beta_2^{(k-1)} - \alpha \times \dagger \\
&\vdots \\
\beta_p^{(k)} &= \beta_p^{(k-1)} - \alpha \times \dagger
\end{aligned}$$

Make your expression as simple as possible, and be sure to include all your working. **Hint: If you haven't done so already, take a look at Question 4 of Tutorial 3.** *what to submit: your coordinate level GD updates along with any working.*

Solution:

This is pretty much identical to the question in tutorial 3:

$$\begin{aligned}
\beta_0^{(k)} &= \beta_0^{(k-1)} - \alpha \times \left(-\frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) \right) \\
&= \beta_0^{(k-1)} + \alpha \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) \\
\beta_1^{(k)} &= \beta_1^{(k-1)} - \alpha \times \left(\beta_1^{(k-1)} - \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{i1} \right) \\
&= \beta_1^{(k-1)} + \alpha \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{i1} - \alpha \beta_1^{(k-1)} \\
\beta_2^{(k)} &= \beta_2^{(k-1)} - \alpha \times \left(\beta_2^{(k-1)} - \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{i2} \right) \\
&= \beta_2^{(k-1)} + \alpha \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{i2} - \alpha \beta_2^{(k-1)} \\
&\vdots \\
\beta_p^{(k)} &= \beta_p^{(k-1)} - \alpha \times \left(\beta_p^{(k-1)} - \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{ip} \right) \\
&= \beta_p^{(k-1)} + \alpha \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)})) x_{ip} - \alpha \beta_p^{(k-1)}.
\end{aligned}$$

(b) For the non-intercept components $\beta_1, \ldots, \beta_p$, re-write the gradient descent updates of the previous question in vector form, i.e. derive an explicit expression for the term † in the following:

$$\beta^{(k)} = \beta^{(k-1)} - \alpha \times \dagger$$

Your expression should only be in terms of β_0, β, x_i and y_i. Next, let $\gamma = [\beta_0, \beta^T]^T$ be the $(p+1)$-dimensional vector that combines the intercept with the coefficient vector β, write down the update

$$\gamma^{(k)} = \gamma^{(k-1)} - \alpha \times \dagger.$$

Note: This final expression will be our vectorized implementation of gradient descent. The point of the above exercises is just to be careful about the differences between intercept and non-intercept parameters. Doing GD on the coordinates is extremely innefficient in practice. *what to submit: your vectorized GD updates along with any working.*

Solution:

$$\begin{aligned}\beta^{(k)} &= \beta^{(k-1)} - \alpha \times \left(\beta^{(k-1)} - \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)}) x_i \right) \\ &= \beta^{(k-1)} - \alpha\beta^{(k-1)} + \alpha \frac{\lambda}{n} \sum_{i=1}^{n} (y_i - \sigma(\beta_0^{(k-1)} + x_i^T \beta^{(k-1)}) x_i \\ &= \beta^{(k-1)} - \alpha\beta^{(k-1)} + \alpha \frac{\lambda}{n} X^T (y - \sigma(\beta_0^{(k-1)} 1_n + X^T \beta^{(k-1)})),\end{aligned}$$

where the sigmoid in the final line is applied element-wise. Either of the final two expressions is sufficient for full marks. Now, we also immediately have

$$\gamma^{(k)} = \gamma^{(k-1)} - \alpha \times \begin{bmatrix} -\frac{\lambda}{n}(y - \sigma(\beta_0^{(k-1)} 1_n + X^T \beta^{(k-1)})) \\ \beta^{(k-1)} - \frac{\lambda}{n} X^T (y - \sigma(\beta_0^{(k-1)} 1_n + X^T \beta^{(k-1)})) \end{bmatrix},$$

where 1_n is the vector of ones.

(c) Derive the (dampened) Newton updates for the logistic regression problem with step size α. You should do this in vector form as in the previous question. Make sure to include all your working. For notational ease, you might find it helpful to write $z_i^{(k)} = \beta_0^{(k)} + x_i^T \beta^{(k)}$. **Hint: When doing this question, first solve it without the regularization term (i.e. ignore the ridge penalty term and take $\lambda = 1$). Once you have a form for the Hessian in that case, it should be more straightforward to extend it to the the more general setting needed here.** *what to submit: your vectorized dampened Newton updates along with any working.*

Solution:

Based on the result in Tutorial 3, the Hessian matrix is given by

$$H(\beta_0, \beta) = \begin{bmatrix} \frac{\lambda}{n} \sum_{i=1}^{n} \sigma(z_i)(1 - \sigma(z_i)) & \frac{\lambda}{n} \sum_{i=1}^{n} \sigma(z_i)(1 - \sigma(z_i)) x_i^T \\ \frac{\lambda}{n} \sum_{i=1}^{n} \sigma(z_i)(1 - \sigma(z_i)) x_i & I_p + \frac{\lambda}{n} \sum_{i=1}^{n} \sigma(z_i)(1 - \sigma(z_i)) x_i x_i^T \end{bmatrix}.$$

In matrix notation (which is not required here, but useful to know), let A be the $n \times n$ diagonal matrix with i-th element $\sigma(z_i)(1 - \sigma(z_i))$, then

$$H(\beta_0, \beta) = \begin{bmatrix} \frac{\lambda}{n} 1_n^T A 1_n & \frac{\lambda}{n} 1_n^T A X \\ \frac{\lambda}{n} X^T A 1_n & I_p + \frac{\lambda}{n} X^T A X \end{bmatrix},$$

where 1_n is the n-dimensional vector of ones.

Let $\gamma^{(k)} = [\beta_0^{(k)}, (\beta^{(k)})^T]^T \in \mathbb{R}^{p+1}$. We can then write $H(\beta_0, \beta) = H(\gamma)$, and similarly the gradient $\nabla L(\beta_0, \beta) = L(\gamma)$. Further, let $v = [v_1, \ldots, v_n]^T$. The Newton steps are then

$$\gamma^{(k)} = \gamma^{(k-1)} - \alpha [H(\gamma^{(k-1)})]^{-1} \begin{bmatrix} -\frac{\lambda}{n} 1_n^T (y - \sigma(z^{(k-1)})) \\ \beta^{(k-1)} - \frac{\lambda}{n} X^T (y - \sigma(z^{(k-1)})). \end{bmatrix}$$

(d) We will now compare the performance of GD and the Newton algorithm on a real dataset using the derived updates in the previous parts. To do this, we will work with the `songs.csv` dataset.

The data contains information about various songs, and also contains a class variable outlining the genre of the song. If you are interested, you can read more about the data here, though a deep understanding of each of the features will not be crucial for the purposes of this assessment. Load in the data and preform the follwing preprocessing:

(I) Remove the following features: "Artist Name", "Track Name", "key", "mode", "time_signature", "instrumentalness"

(II) The current dataset has 10 classes, but logistic regression in the form we have described it here only works for binary classification. We will restrict the data to classes 5 (hiphop) and 9 (pop). After removing the other classes, re-code the variables so that the target variable is $y = 1$ for hiphop and $y = 0$ for pop.

(III) Remove any remaining rows that have missing values for any of the features. Your remaining dataset should have a total of 3886 rows.

(IV) Use the `sklearn.model_selection.train_test_split` function to split your data into X_train, X_test, Y_train and Y_test. Use a `test_size` of 0.3 and a `random_state` of 23 for reproducibility.

(V) Fit the `sklearn.preprocessing.StandardScaler` to the resulting training data, and then use this object to scale both your train and test datasets.

(VI) Print out the first and last row of X_train, X_test, y_train, y_test (but only the first three columns of X_train, X_test).

What to submit: the print out of the rows requested in (VI). A copy of your code in solutions.py

Solution:

- first row X_train: [-0.93555843, 0.67519298, 1.3849985]
- last row X_train: [-1.13301479,-1.09458877, 0.96702449]
- first row X_test: [-0.29382524, 1.36005105, 0.26306826]
- last row X_test: [-0.29382524,-1.05390413,-1.34833155]
- first row y_train: 0
- last row y_train: 1
- first row y_test: 0
- last row y_test: 1

The code for this section is:

```
1
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 from sklearn.model_selection import train_test_split
6 from sklearn.preprocessing import StandardScaler
7 df = pd.read_csv("songs.csv")
8 df = df.drop(columns=["Artist Name", "Track Name", "key", "mode", "time_signature", "
      instrumentalness"])
9
```

```
df = df[(df.Class == 9) + (df.Class==5)]  # 5 = hiphop, 9 = pop
df.Class.replace({5:1, 9:-1}, inplace=True)
df.dropna(inplace=True)
df.isnull().sum()
df.shape

df.head()

y = df.Class.to_numpy()
X = df.iloc[:, :-1].to_numpy()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state
    =23)

scaler = StandardScaler().fit(X_train)
scaled_X_train = scaler.transform(X_train)
scaled_X_test = scaler.transform(X_test)

scaled_X_train[[0,-1], :3]

print('\item first row X\_train: ', np.array2string(scaled_X_train[0,:3], separator=',
    '))
print('\item last row X\_train: ', np.array2string(scaled_X_train[-1,:3], separator=',
    '))
print('\item first row X\_test: ', np.array2string(scaled_X_test[0,:3], separator=',')
    )
print('\item last row X\_test: ', np.array2string(scaled_X_test[-1,:3], separator=',')
    )
print('\item first row y\_train: ', np.array2string(y_train[0], separator=','))
print('\item last row y\_train: ', np.array2string(y_train[-1], separator=','))
print('\item first row y\_test: ', np.array2string(y_test[0], separator=','))
print('\item last row y\_test: ', np.array2string(y_test[-1], separator=','))

```

A quick aside: Backtracking Line Search: In homework 1, we chose the step-size parameter in our implementations naively by looking at a grid of step-sizes and choosing the best one. In practice, this is obviously not feasible (computational cost of training the model multiple times for a large number of candidate step-sizes). One very popular and empirically successful approach is known as Backtracking Line Search (BLS). In BLS, the step-size is chosen adaptively at each iteration of the algorithm by checking a given criteria. In particular, choose parameters $0 < a \leq 0.5$ and $0 < b < 1$. At iteration k of the algorithm (where we are minimizing the function $f(x)$), the current iterate is $x^{(k)}$, set the step-size to $\alpha = 1$. Then, while

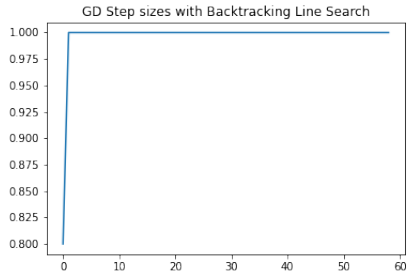
$$f(x^{(k)} - \alpha \nabla f(x^{(k)})) > f(x) - a\alpha \|\nabla f(x^{(k)})\|_2^2,$$

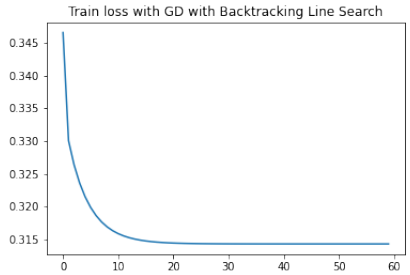
shrink the step-size according to $\alpha = b\alpha$, otherwise use the current step-size α in your update. We will not explain why this is a sensible thing to do here, but you will be able to find information about it online if you are interested. (Of course, we can also discuss it during one of the consultation sessions). Importantly however, the BLS idea can be applied in both gradient descent and the dampened Newton algorithm, and you will be required to use the BLS in your implementations of both algorithms in the following questions.

(e) Run gradient descent with backtracking line search on the training dataset for 60 epochs and $\lambda = 0.5$, and initalize $\beta_0^{(0)} = 0, \beta^{(0)} = 0_p$, where 0_p is the p-dimensional vector of zeroes. For your BLS implementation take $a = 0.5, b = 0.8$. Report your train and test losses, as well as plots of your step-sizes at each iteration, and train loss at each iteration. **Hint: if you need a sanity check here, the best thing to do is use sklearn to fit logistic regression models. This should give you an idea of what kind of loss your implementation should be achieving (if your implementation does as well or better, then you are on the right track)** *what to submit: two plots, one for the step-sizes and the other for the train losses. Report your train and test losses, and a screen shot of any code used in this section, as well as a copy of your code in solutions.py.*

Solution:

The train loss is 0.31429682466558384, the test loss is 0.31751509349581103.



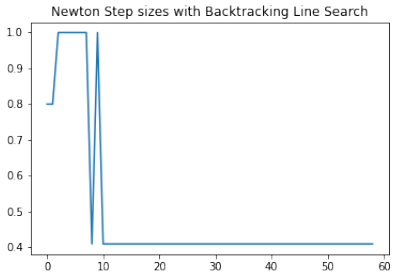


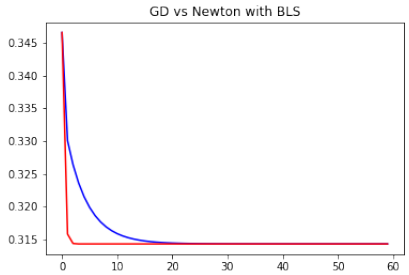

```
# Backtracking Line Search
def BackTrack(x, f, grad_f, a, b, X, y, L):
    t = 1
    LHS = f(x - t * grad_f(x, X, y, L), X, y, L)
    RHS = f(x, X, y, L) - a * t * np.linalg.norm(grad_f(x, X, y, L), ord=2)**2
    while LHS > RHS:
        t *= b
```

```
        LHS = f(x - t * grad_f(x, X, y, L), X, y, L)
        RHS = f(x, X, y, L) - a * t * np.linalg.norm(grad_f(x, X, y, L), ord=2)**2
    return t

# implementing GD for logreg
n = X_train.shape[0]
p = X_train.shape[1]
n_epochs = 60
n_iter = n_epochs
lam = .5

gammas_GD = np.zeros((n_iter, p+1))
alphas = np.zeros(n_iter)
for j in range(1, n_iter):
    alphas[j-1] = BackTrack(gammas_GD[j-1], loss, calc_grad, 0.5, 0.8, X_train,
    y_train, lam)
    gammas_GD[j] = gammas_GD[j-1] - alphas[j-1] * calc_grad(gammas_GD[j-1], X_train,
    y_train, lam)

gamma_GD = gammas_GD[-1]

print("train loss: ", loss(gamma_GD, X_train, y_train, lam))
print("test  loss: ", loss(gamma_GD, X_test, y_test, lam))



plt.plot(np.arange(n_iter), alphas)
plt.title("GD Step sizes with Backtracking Line Search")
plt.show()

plt.plot(np.arange(n_iter), [loss(g, X_train, y_train, lam) for g in gammas_GD])
plt.title("Train loss with GD with Backtracking Line Search")
plt.show()
```

(f) Run the dampened Newton algorithm with backtracking line search on the training dataset for 60 epochs and $\lambda = 0.5$. Use the same parameters for your BLS algorithm as in the previous question. Use the same initialization for β_0, β as in the previous question Report your train and test losses, as well as plots of your step-sizes at each iteration. Further, provide a single plot showing the train loss for both GD and Newton algorithms (use labels/legends to make your plot easy to read). *what to submit: two plots, one for the step-sizes and the other for the train losses (of both GD and Newton). Report your train and test losses, and a screen shot of any code used in this section, as well as a copy of your code in solutions.py.*

Solution:
The train loss is 0.31429681501971396, the test loss is 0.3175178211990147.






```python
# implementing Newton for logreg
n = X_train.shape[0]
p = X_train.shape[1]

def calc_hess(gamma, X, lam):
    n, p = X.shape
    z = np.dot(X, gamma[1:]) + gamma[0]
    sig_z = sigmoid(z)
    A = np.diag(sig_z * (1-sig_z))
    one_vec = np.ones(n)

    H = np.zeros((p+1, p+1))
    H[0,0] = (lam/n) * A.sum()
    H[0, 1:] = (lam/n) * (one_vec.T @ A @ X)
    H[1:, 0] = (lam/n) * (X.T @ A @ one_vec)
    H[1:, 1:] = np.eye(p) + (lam/n) * (X.T@A@X)

    return H

gammas_N = np.zeros((n_iter, p+1))
alphas = np.zeros(n_iter)

for j in range(1, n_iter):
    alphas[j-1] = BackTrack(gammas_N[j-1], loss, calc_grad, 0.5, 0.8, X_train, y_train
    , lam)
    H = calc_hess(gammas_N[j-1], X_train, lam)
    Hinv = np.linalg.inv(H)
    gammas_N[j] = gammas_N[j-1] - alphas[j-1] * Hinv @ calc_grad(gammas_N[j-1],
    X_train, y_train, lam)

gamma_N = gammas_N[-1]

print("train loss: ", loss(gamma_N, X_train, y_train, lam))
print("test  loss: ", loss(gamma_N, X_test, y_test, lam))



plt.plot(np.arange(n_iter), alphas)
plt.title("Newton Step sizes with Backtracking Line Search")
plt.show()

plt.plot(np.arange(n_iter), [loss(g, X_train, y_train, lam) for g in gammas_N])
plt.title("Train loss for Newton with Backtracking Line Search")
plt.show()
```

```
43
44 losses_GD = [loss(g, X_train, y_train, lam) for g in gammas_GD]
45 losses_N = [loss(g, X_train, y_train, lam)  for g in gammas_N]
46 plt.plot(np.arange(n_iter), losses_GD, color='blue', label="GD")
47 plt.plot(np.arange(n_iter), losses_N, color='red', label="Newton")
48 plt.legend()
49 plt.show()
```

(g) In general, it turns out that Newton's method is much better than GD, in fact convergence of the Newton algorithm is quadratic, whereas convergence of GD is linear (much slower than quadratic). Given this, why do you think gradient descent and its variants (e.g. SGD) are much more popular for solving machine learning problems? *what to submit: some commentary*

Solution:
Some mention of the computational complexity of Newton's method, need to compute the Hessian makes it much more computationally expensive relative to GD.

Question 3. More on MLE

Let $X_1, \ldots, X_n \overset{\text{i.i.d.}}{\sim} N(\mu, \sigma^2)$. Recall that in Tutorial 2 we showed that the MLE estimators of μ, σ^2 were $\hat{\mu}_{\text{MLE}}$ and $\hat{\sigma}^2_{\text{MLE}}$ where

$$\hat{\mu}_{\text{MLE}} = \overline{X}, \qquad \text{and} \qquad \hat{\sigma}^2_{\text{MLE}} = \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2.$$

In this question, we will explore these estimators in more depth.

(a) Find the bias and variance of both $\hat{\mu}_{\text{MLE}}$ and $\hat{\sigma}^2_{\text{MLE}}$. **Hint:** You may use without proof the fact that

$$\text{var}\left(\frac{1}{\sigma^2}\sum_{i=1}^{n}(X_i - \overline{X})^2\right) = 2(n-1)$$

what to submit: the bias and variance of the estimators, along with your working.

Solution:
For $\hat{\mu}_{\text{MLE}}$ the calculations follow identically to part (a) of Tutorial 2 Question 2, except that now we have

$$\text{var}(\hat{\mu}_{\text{MLE}}) = \frac{\sigma^2}{n},$$

which is just a generalization of the result in part (a) to any choice of σ. Now, we have shown

already that $\hat{\sigma}^2_{\text{MLE}} = \frac{1}{n}\sum_{i=1}^n (X_i - \overline{X})^2$. Therefore,

$$\begin{aligned}
\text{bias}(\hat{\sigma}^2_{\text{MLE}}) &= \mathbb{E}\left(\frac{1}{n}\sum_{i=1}^n (X_i - \overline{X})^2\right) - \sigma^2 \\
&= \frac{1}{n}\sum_{i=1}^n (\mathbb{E}(X_i^2) - 2\mathbb{E}(X_i\overline{X}) + \mathbb{E}(\overline{X}^2)) - \sigma^2 \\
&= \frac{1}{n}\sum_{i=1}^n \left\{(\sigma^2 + \mu^2) - 2\left(\frac{\sigma^2}{n} + \mu^2\right) + \left(\frac{\sigma^2}{n} + \mu^2\right)\right\} - \sigma^2 \\
&= \sigma^2\frac{n-1}{n} - \sigma^2 \\
&= -\frac{\sigma^2}{n},
\end{aligned}$$

where we have used the fact that

$$\text{var}(X_i) = \mathbb{E}(X_i^2) - [\mathbb{E}(X_i)]^2 \implies \mathbb{E}(X_i^2) = \text{var}(X_i) + [\mathbb{E}(X_i)]^2 = \sigma^2 + \mu^2,$$

and that

$$\mathbb{E}(X_i\overline{X}) = \frac{1}{n}\sum_{j=1}^n \mathbb{E}(X_iX_j) = \frac{1}{n}\left(\mathbb{E}(X_i^2) + (n-1)\mathbb{E}(X_iX_j)\right) = \frac{1}{n}\left(\mathbb{E}(X_i^2) + (n-1)\mathbb{E}(X_i)\mathbb{E}(X_j)\right)$$

this shows that the MLE estimator of σ^2 is negatively biased, which means that it tends to under-estimate the true parameter. Now, for the variance of this estimator, we rely on the provided hint to find that

$$\begin{aligned}
\text{var}(\hat{\sigma}^2_{\text{MLE}}) &= \text{var}\left(\frac{1}{n}\sum_{i=1}^n (X_i - \overline{X})^2\right) \\
&= \frac{\sigma^4}{n^2}\text{var}\left(\frac{1}{\sigma^2}\sum_{i=1}^n (X_i - \overline{X})^2\right) \\
&= \frac{2(n-1)\sigma^4}{n^2}
\end{aligned}$$

(b) Your friend tells you that they have a much better estimator for σ^2, namely:

$$\tilde{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^n (X_i - \overline{X})^2.$$

Discuss whether this estimator is better or worse than the MLE estimator:

$$\hat{\sigma}^2_{\text{MLE}} = \frac{1}{n}\sum_{i=1}^n (X_i - \overline{X})^2.$$

Be sure to include a detailed analysis of the bias and variance of both estimators, and describe what occurs to each of these quantities (for each of the estimators) as the sample size n increases (use plots). For your plots, you can assume that $\sigma = 1$.

what to submit: the bias and variance of the new estimator. A plot comparing the bias of both estimators as a function of the sample size n, a plot comparing the variance of both estimators as a function of the sample size n, use labels/legends in your plots. A copy of the code used here in solutions.py

Solution:

First, we note that

$$\tilde{\sigma}^2 = \frac{n}{n-1}\hat{\sigma}^2_{\text{MLE}}.$$

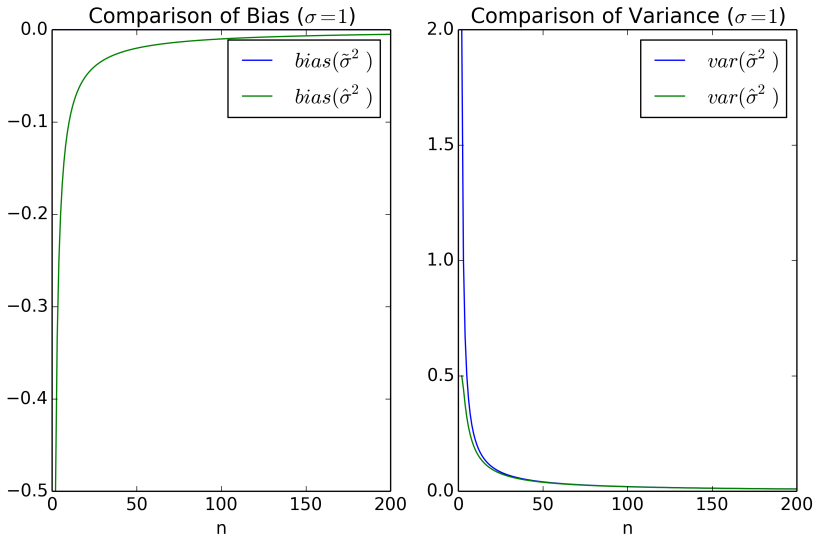
Now, we have

$$\begin{aligned}
\text{bias}(\tilde{\sigma}^2) &= \mathbb{E}\left(\frac{n}{n-1}\hat{\sigma}^2_{\text{MLE}}\right) - \sigma^2 \\
&= \frac{n}{n-1}\mathbb{E}\left(\hat{\sigma}^2_{\text{MLE}}\right) - \sigma^2 \\
&= \frac{n}{n-1}\sigma^2\frac{n-1}{n} - \sigma^2 \\
&= 0,
\end{aligned}$$

so $\tilde{\sigma}^2$ is an unbiased estimator. Next, we have

$$\begin{aligned}
\text{var}(\tilde{\sigma}^2) &= \text{var}\left(\frac{n}{n-1}\hat{\sigma}^2_{\text{MLE}}\right) \\
&= \frac{n^2}{(n-1)^2}\text{var}\left(\hat{\sigma}^2_{\text{MLE}}\right) \\
&= \frac{n^2}{(n-1)^2}\frac{2(n-1)\sigma^4}{n^2} \\
&= \frac{2\sigma^4}{n-1}.
\end{aligned}$$

We can now compare both estimators by plotting their bias and variance for a range of sample sizes n. We see that as the sample size gets larger, the estimators have identical bias and variance, which is to be expected. We see that the bias of $\tilde{\sigma}^2$ is smaller than that of the MLE for small sample sizes, whereas its variance is larger than that of the MLE for small sample sizes. Note that this should answer the natural question that comes up of why we divide by $n-1$ when we are computing the sample variance.



(c) Compute and then plot the MSE of the two estimators considered in the previous part. For your plots, you can assume that $\sigma = 1$. Provide some discussion as to which estimator is better (according to their MSE), and what happens as the sample size n gets bigger. *what to submit: the MSEs of the two variance estimators. A plot comparing the MSEs of the estimators as a function of the sample size n, and some commentary. Use labels/legends in your plots. A copy of the code used here in solutions.py*

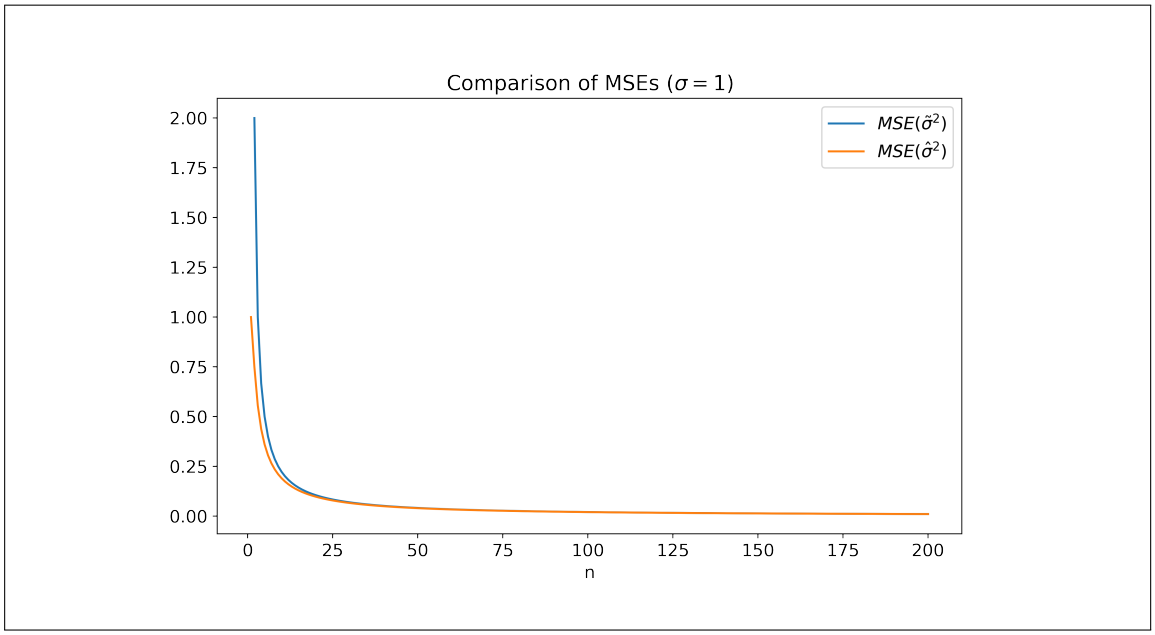
Solution:
We have

$$\text{MSE}(\tilde{\sigma}^2) = \text{var}(\tilde{\sigma}^2) = \frac{2\sigma^4}{n-1}$$

and

$$\text{MSE}(\hat{\sigma}^2_{\text{MLE}}) = \frac{\sigma^4}{n^2} + \frac{2(n-1)\sigma^4}{n^2} = \frac{(2n-1)\sigma^4}{n^2}.$$

From the plot, we see that the MLE is superior in terms of MSE, but the difference is negligible as n grows.


Comparison of MSEs ($\sigma = 1$)
$MSE(\tilde{\sigma}^2)$
$MSE(\hat{\sigma}^2)$
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
25
50
75
100
125
150
175
200
n