Assignment 1

Sebastian Lie
A0164657U

1. Min cost Multi commodity flow

Let $x_{i,j}^{(k)}$ be the amount of k commodity that is sent from node i to node j. The optimisation model to solve the min cost multi commodity flow problem is as follows.

$$
\begin{aligned}
&\text{minimize} && \sum_{(i,j)\in\mathcal{E}} \sum_{k}^{L} c_{i,j}^{(k)} x_{i,j}^{(k)} \\
&\text{subject to} && \sum_{k}^{L} x_{i,j}^{(k)} \leq u_{i,j},\ \forall (i,j) \in \mathcal{E} \\
&\text{and} && x_{i,j}^{(k)} \geq 0,\ \forall (i,j) \in \mathcal{E},\ 1 \leq k \leq L \\
&\text{and} && \sum_{j} x_{i,j}^{(k)} - \sum_{j} x_{j,i}^{(k)} = \begin{cases} d_k,\ \text{if } i = s_k,\ 1 \leq k \leq L \\ -d_k,\ \text{if } i = t_k,\ 1 \leq k \leq L \\ 0 \text{ else} \end{cases}
\end{aligned}
$$

2. Minimise f over feasible set

(a)

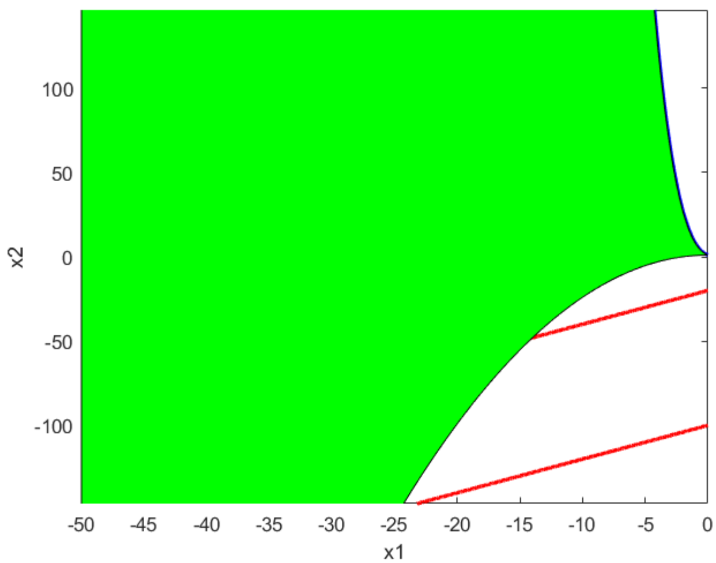

Figure 1: Feasible Region, S

Where the feasible region is in green, and the red lines denote the function to be minimised at different values, with the 2nd red line having a smaller value of f(x) than the red line on top.

(b)

let $C_1 = \{\mathbf{x} \in \mathbb{R}^2 \mid (1-x_1)^3 - x_2 \geq 0\}$ and $C_2 = \{\mathbf{x} \in \mathbb{R}^2 \mid x_2 + 0.25x_1^2 - 1 \geq 0\}$

Then, since both C_1 and C_2 are of the form: $\{\mathbf{x} \in \mathbb{R}^2 \mid g(x) \geq 0\}$

using proposition 2.8, we can conclude that both C_1 and C_2 are closed. Then since the feasible set is the intersection of C_1 and C_2, and the intersection of 2 closed sets is closed, $C_1 \cap C_2$, the feasible set, is also closed.

(c)

We follow the same sets as in (b): C_1 and C_2.

First we investigate if C_1 is bounded. For C_1, we can see that $x_2 \leq (1-x_1)^3$, and at $(1-x_1)^3 - x_2 = 0$, $x_2 = (1-x_1)^3 \in C_1$ by letting $x_1 \to \infty$ and $x_1 \to -\infty$, we have that

$$\lim_{x_1 \to \infty} x_2 = \lim_{x_1 \to \infty} (1-x_1)^3 = -\infty$$

and

$$\lim_{x_1 \to -\infty} x_2 = \lim_{x_1 \to -\infty} (1-x_1)^3 = \infty$$

respectively.

And thus we have that for C_1, $x_2 \in (-\infty, \infty)$. Therefore, C_1 is unbounded.

For C_2, we can see that $x_2 \geq 1 - 0.25x_1^2$, i.e x_2 is lower bounded by $1 - 0.25x_1^2$, and x_2 is upper bounded by $-1 + 0.25x_1^2$. so by letting $x_1 \to -\infty$ or $x_1 \to \infty$, at $x_2 + 0.25x_1^2 - 1 = 0$, $x_2 = 1 - 0.25x_1^2 \in C_2$, we obtain the lower bound of x_2:

$$\lim_{x_1 \to \infty} x_2 = \lim_{x_1 \to -\infty} 1 - 0.25x_1^2 = -\infty$$

and the upper bound of x_2:

$$\lim_{x_1 \to \infty} x_2 = \lim_{x_1 \to \infty} 0.25x_1^2 - 1 = \infty$$

And thus we have that for C_2, $x_2 \in (-\infty, \infty)$. Therefore, C_2 is unbounded.

Since the feasible set must fulfill the conditions of both C_1 and C_2, we can denote the feasible set by $C_1 \cap C_2$. Then, for C_{12}, $x_2 \leq (1 - x_1)^3$ and $x_2 \geq 1 - 0.25x_1^2$, i.e x_2 in the feasible set is lower bounded by $1 - 0.25x_1^2$ and upper bounded by $(1 - x_1)^3$. Then as $x_1 \to -\infty$,

$$\lim_{x_1 \to \infty} 1 - 0.25x_1^2 = -\infty$$

and

$$\lim_{x_1 \to \infty} (1 - x_1)^3 = \infty$$

Which indicates that for the feasible set, $x_2 \in (\infty, \infty)$, and therefore, $C_1 \cap C_2$ is unbounded.

(d)

No, a minimiser for $f(x) = -2x_1 + x_2$ does not exist. Since x_2 is unbounded, as $f(x) \to -\infty$, $x_2 \to -\infty$,and f(x) moves along the lower bound of the feasible set, $1 - 0.25x_1^2$, along which there is no stationary point, and thus within any neighbourhood, we can always find a value of f(x) smaller.

3. Existence of Global Minimiser

Let $\mathbf{0}$ denote the 0 vector. Let us begin by proving that $f(\mathbf{0}) < 1$. We proceed by contradiction.

First, we note that $\|\mathbf{x}\|_\infty \geq 0$ by definition, since it is the maximum of the absolute values in a vector.

Suppose $f(\mathbf{0}) \geq 1$. Then $\exists\ r$ such that $\|\mathbf{0}\|_\infty \geq r \implies r = 0$. Then by the properties of f, $\exists\ \hat{\mathbf{x}}$ s.t $\|\hat{\mathbf{x}}\|_\infty < r = 0$. This is not possible, since $\|\mathbf{x}\|_\infty \geq 0$, and thus we obtain a contradiction, and can conclude that $f(\mathbf{0}) < 1$.

Now, let us choose $r = 1$. Then we can construct a set $C_1 = \{\mathbf{x} \in \mathbb{R}^n | \ \|\mathbf{x}\|_\infty \leq 1\}$ where $\mathbf{0} \in C_1$, since $\|\mathbf{0}\|_\infty < 1$.

Thus there exists at least 1 $\hat{\mathbf{x}} \in C_1$ where $f(\hat{\mathbf{x}}) < 1$ since we have $f(\mathbf{0}) < 1$, and by the properties of f. Additionally, we have that $\forall\ \mathbf{y} \notin C_1,\ \|\mathbf{y}\|_\infty > 1$ where $\mathbf{y} \in \mathbb{R}^n$, and by the properties of f, $f(\mathbf{y}) \geq 1$. This implies that $\forall \mathbf{y} \notin C_1$ and $\hat{\mathbf{x}} \in C_1, f(\mathbf{y}) > f(\hat{\mathbf{x}})$, since $f(\hat{\mathbf{x}}) < 1$.

We then observe that C_1 is closed and bounded, and thus by Weierstrass's theorem, $\exists\ \mathbf{x}^*$ such that $\forall \mathbf{x} \in C_r, f(\mathbf{x}^*) \leq f(\mathbf{x})$, i.e $\mathbf{x}^*$ is the global minimiser on C_1.

Therefore, we have that $f(\mathbf{x}^*) \leq f(\hat{\mathbf{x}}) < f(\mathbf{y})$ for $\mathbf{y} \notin C_1$, and so we conclude that $f(\mathbf{x}^*) \leq f(\mathbf{x})\ \forall \mathbf{x} \in \mathbb{R}^n$, thus f has a global minimiser, $\mathbf{x}^*$.

4. Convexity of negative log likelihood

(a)

To show convexity, we first we obtain $H_f(x)$ by differentiating $f(x) = log(1 + e^{-x})$

$$\nabla f(x) = \frac{-e^{-x}}{1+e^{-x}} = \frac{-1}{1+e^{x}}$$

Differentiating once more,

$$H_f(x) = \frac{e^x}{(1+e^x)^2}$$

We observe that $\forall x \in \mathbb{R}$, $e^x > 0$, and $e^{-x} = \frac{1}{e^x} > 0$.
Therefore,

$$H_f(x) = \frac{e^x}{(1+e^x)^2} > 0$$

$H_f(x)$ is positive definite and thus by theorem 3.2, $f(x) = log(1 + e^{-x})$ is convex over $\mathbb{R}$.

(b)

First we note, as seen in example 3.7, Chapter 3 of our lecture notes, for any affine function $h(\boldsymbol{w}) = \boldsymbol{a}^T\boldsymbol{x} + b$ with $\boldsymbol{a} \in \mathbb{R}, b \in \mathbb{R}$,

$$h(\lambda \boldsymbol{w_0} + (1-\lambda)\boldsymbol{w_1}) = \lambda h(\boldsymbol{w_0}) + (1-\lambda)h(\boldsymbol{w_1})$$

for $\lambda \in [0,1], \boldsymbol{w_0}, \boldsymbol{w_1} \in \mathbb{R}^n$.
Thus, for any $f(\boldsymbol{w}) = g(h(\boldsymbol{w}))$ where g is a convex function, and an affine function $h(\boldsymbol{w}) = A\boldsymbol{w} + \boldsymbol{b}$ with $A \in \mathbb{R}^{mxp}, \boldsymbol{b} \in \mathbb{R}$we have

$$\begin{aligned} f(\lambda \boldsymbol{w_0} + (1-\lambda)\boldsymbol{w_1}) &= g(h(\lambda \boldsymbol{w_0} + (1-\lambda)\boldsymbol{w_1})) \\ &= g(\lambda h(\boldsymbol{w_0}) + (1-\lambda)h(\boldsymbol{w_1})) \\ &\leq \lambda g(h(\boldsymbol{w_0})) + (1-\lambda)g(h(\boldsymbol{w_1})) \end{aligned}$$

and thus by Definition 3.2, f, the composition of a convex and affine function, is convex.

(c)

Since $f(x) = log(1+e^{-x})$ is convex, and $h(\mathbf{w}) = c\mathbf{w}^T\mathbf{x_i}$ for some $c \in \mathbb{R}$ is an affine function, using (b), $f(h(\mathbf{w}))$ is convex. Since the sum of convex functions is convex by corollary 3.1, let $c = y_i$, and we have that $l(\mathbf{w}) = \sum_{i=1}^{n} log(1 + e^{-y_i\mathbf{w}^T\mathbf{x_i}}$ is convex.

5. Logistic Regression, Spam email

(a)

$$\nabla l(\mathbf{w}) = \sum_{i=1}^{n} \frac{-y_i\mathbf{X_i}e^{-y_i\mathbf{w}^T\mathbf{X_i}}}{1 + e^{-y_i\mathbf{w}^T\mathbf{X_i}}}$$
$$= \sum_{i=1}^{n} -\frac{y_i\mathbf{X_i}}{1 + e^{y_i\mathbf{w}^T\mathbf{X_i}}}$$

(b)

Written in python. Gradient function was tested with test_grad() function

```python
import numpy as np
import math
from scipy.io import loadmat
import time

### DSA3102 CONVEX OPTIMISATION HW1 ###
## Author: Sebastian Lie

mat = loadmat('HW1data.mat')

testx = mat["Xtest"]
trainx = mat["Xtrain"]  # (3065,57)
testy = mat["ytest"]  # (3065,1)
trainy = mat["ytrain"]

## exact line search methods ##

def gprime(w,d,t,X,y):
    res = 0
    for i in range(len(X)):
        exp_part = np.exp(-y[i] * np.dot(w,X[i])) * np.exp(-y[i] * t * np.dot(
    d,X[i]))
        numerator = -y[i]*np.dot(np.transpose(d),X[i])*exp_part
        res = res +(numerator/(1+exp_part))
    print(res)
    return res[0]

def gprimeprime(w,d,t,X,y):
    res = 0
    for i in range(len(X)):
        exp_part = np.exp(-y[i] * np.dot(np.transpose(w),X[i])) * np.exp(-y[i]
     *t* np.dot(np.transpose(d),X[i]))
        numerator = (-y[i]*np.dot(np.transpose(d),X[i])**2)*exp_part
```

```
        res = res +(numerator/(1+exp_part)**2)
    print(res[0])
    return res[0]

def newtons(w, d, tol):
    t = 1
    while abs(gprime(w,d,t,trainx,trainy)) > tol:

        t = t - (gprime(w,d,t,trainx,trainy)/gprimeprime(w,d,t,trainx,trainy))
    return t

def golden_search(w, d, a, b, maxit, tol):

    phi = (sqrt(5.0) - 1)/2.0
    lam = b - phi*(b - a)
    mu = a + phi *(b-a)
    flam = loglikelihood(w+lam*d,trainx,trainy)
    fmu = loglikelihood(w+mu*d,trainx,trainy)
    for i in range(maxit):
        if flam > fmu:
            a = lam
            lam = mu
            mu = a + phi*(b-a)
            fmu = loglikelihood(w+mu*d,trainx,trainy)
        else:
            b = mu
            mu = lam
            fmu = flam
            lam = b - phi*(b-a)
            flam = loglikelihood(w+lam*d,trainx,trainy)

        if (b-a) <= tol:
            break
    return (b-a)/2

def bisection(a,b,tol):

    maxit = 10000
    la = loglikelihood(a, trainx, trainy)
    lb = loglikelihood(b, trainx, trainy)

    for i in range(maxit):
        x = (a+b)/2
        lx = loglikelihood(x, trainx,trainy)
        if (lx * lb <= 0):
            a = x
            la = lx
        else:
            b = x
            lb = lx
        if (b-a) < tol:
            break
    return x
```

```
def armijo(alpha_bar,w,d,beta,sigma):
    fx0 = loglikelihood(w,trainx,trainy)
    alpha = alpha_bar
    delta = np.dot(loglikelihood_grad(w,trainx,trainy),d)
    while loglikelihood(w+alpha*d,trainx,trainy) >  fx0 + alpha*sigma*delta:
        alpha = beta * alpha
    return alpha

######################################

## Objective function and its gradient ##

def loglikelihood(w, X, y):
    result = 0
    for i in range(len(X)):
        exp_part = np.exp(-y[i] * np.dot(np.transpose(w),X[i]))
        result += np.log(1+exp_part)
    return result

def loglikelihood_grad(w, X, y):
    grad = np.zeros(57)
    for i in range(len(X)):
        exp_part = np.exp(-y[i] * np.dot(np.transpose(w),X[i]))
        numerator = -y[i]*exp_part*X[i]
        grad = grad +(numerator/(1+exp_part))
    return grad

######################################

## Steepest descent ##


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


def steepest_descent(X,y,w0,maxit,tol,*line_search_params):  # use varargs
    step_size = 1
    w = w0  # inital guess
    line_search_method = line_search_params[0]
    obj_value_list = list()
    num_iter = 0
    for i in range(maxit):

        grad = loglikelihood_grad(w,trainx,trainy)
        d = -grad
        norm_d = np.linalg.norm(d)
        if norm_d < tol:
            break
        obj_value = loglikelihood(w,trainx,trainy)[0]
        obj_value_list.append(obj_value)
        print("Iteration {0}:obj = {1:9.3f}".format(i,norm_d))
        if norm_d < tol:
            break
        else:
```

```

            if line_search_method == "armijo":
                beta = line_search_params[1]
                sigma = line_search_params[2]
                w_prev = w
                step_size = armijo(step_size,w,d,beta,sigma)

            elif line_search_method == "fixed":
                step_size = line_search_params[1]
                w_prev = w

            elif line_search_method == "exact":
                step_size = newtons(w,d,0.1)

            elif line_search_method == "diminishing":
                step_size = line_search_params[1]/math.sqrt(i)

            else:  # default is armijos
                w_prev = w
                step_size = armijo(step_size,w,d,0.7,0.2)

            # update
            w = w +  step_size * d
            num_iter += 1

    return w, obj_value_list, num_iter


def predict(w, xtest, ytest):
    if len(w) != len(xtest):
        w = np.transpose(w)
    yhat = sigmoid(np.dot(xtest,w))
    yhat = np.fromiter(map(lambda x: 1 if x > 0.5 else -1,yhat),dtype=np.double)
    correct = 0
    for i in range(len(yhat)):
      if yhat[i] == ytest[i]:
        correct += 1
    return correct/len(yhat)

## Helper functions that produce useful things ##

# Vimpt
def test_grad(): # proof that grad function works well enough
    alp = 1*10**(-8)
    x = np.random.rand(57)
    differences = list()
    for i in range(57):
        e0 = np.zeros(57)
        e0[i] = 1 # test 0th part
        diff = loglikelihood_grad(x,trainx,trainy)[i] - (loglikelihood(x+alp*e0,trainx,trainy)-loglikelihood(x, trainx,trainy))/alp
        differences.append(round(diff[0],5))
    return differences
'''
```

```
Produced:
[-0.01087, -0.00607, -0.00211, -0.00653, -0.00775, -0.00695, -0.01511, -0.00564, -0.01248, -0.00927, -0.01077, -0.00991, -0.00842, -0.00685, -0.00771, -0.007,
-0.01319, -0.01054, -0.0055, -0.00613, -0.01242, -0.00824, -0.00657, -0.00809, -0.00528, -0.00606, -0.0072, -0.00767, -0.00747, -0.0079, -0.00731, -0.00784,
-0.00672, -0.00775, -0.00796, -0.01021, -0.00604, -0.00764, -0.00567, -0.00743, -0.00753,
-0.00705, -0.00747, -0.00642, -0.00621, -0.00564, -0.00703, -0.00686, -0.005, -0.00376, -0.00468, 0.00207, -0.00642, -0.00411, -0.00089, -0.00049, -0.00891]
'''

def GridSearch():

    # find best armijo parameters and initial solution
    for b in np.arange(0.1,1,0.1): # use arange cos need to iterate through floats
        for s in np.arange(0.1,0.5,0.1):
            start =time.time()
            wA = steepest_descent(trainx,trainy,np.zeros(57),100000,0.5,"armijo",b,s)
            end = time.time()
            acc_dict["w0 = 0, Armijo beta = {0}, sigma = {1}".format(b,s)] = (predict(wA,testx,testy),end-start)
    for b in np.arange(0.1,1,0.1):
        for s in np.arange(0.1,0.5,0.1):
            start =time.time()
            wA = steepest_descent(trainx,trainy,np.ones(57),100000,0.5,"armijo",b,s)
            end =time.time()
            acc_dict["w0 = 1, Armijo beta = {0}, sigma = {1}".format(b,s)] = (predict(wA,testx,testy),end-start)
    for b in np.arange(0.1,1,0.1):
        for s in np.arange(0.1,0.5,0.1):
            start =time.time()
            wA = steepest_descent(trainx,trainy,-1*np.ones(57),100000,0.5,"armijo",b,s)
            end = time.time()
            acc_dict["w0 = -1, Armijo beta = {0}, sigma = {1}".format(b,s)] = (predict(wA,testx,testy),end-start)


def plot_results():

    wA, armijo_obj_values, num_iter1 = steepest_descent(trainx,trainy,np.ones(57),100000,100,"armijo",0.7,0.2)
    wA2, armijo_obj_values2, num_iter2 = steepest_descent(trainx,trainy,np.ones(57),100000,100,"armijo",0.7,0.1)

    fig, ax = plt.subplots()
    ax.plot(armijo_obj_values, range(1,num_iter1+1), 'r',label="beta = 0.7, sigma = 0.2")
```

```
    ax.plot(armijo_obj_values2, range(1,num_iter2+1), 'b',label="beta = 0.7, sigma = 0.1")
    plt.xlabel("Objective function values, Armijo's Step Size Strategy")
    plt.ylabel("Number of iterations")
    legend = ax.legend(loc='upper right',fontsize='small')
    plt.show()

    wf, fixed_obj_values, num_iter3 = steepest_descent(trainx,trainy,np.ones(57),100000,150,"fixed",0.001)
    wf2, fixed_obj_values2, num_iter4 = steepest_descent(trainx,trainy,np.ones(57),100000,150,"fixed",0.0005)

    fig, ax = plt.subplots()
    ax.plot(fixed_obj_values, range(1,num_iter3+1), 'r',label="Step Size = 0.001")
    ax.plot(fixed_obj_values2, range(1,num_iter4+1), 'b',label="Step Size = 0.0005")
    plt.xlabel("Objective function values, Fixed Step Size Strategy")
    plt.ylabel("Number of iterations")
    legend = ax.legend(loc='upper right',fontsize='small')
    plt.show()

#GridSearch()
#plot_results()
```

(c)

I tried fixed step size, exact line search using newton's method, and armijo step size strategies.

Although I am fairly certain I implemented newton's method correctly, I continually get math overflow errors, and am unable to obtain a solution using newton's method. It is possible that I am simply trying the values that cause overflow, but I have tried a combination of values and come to the conclusion that the twice differentiated function of g(t) gives huge values that cannot be dealt with. Exact line search using the bisection method or golden section search is also quite difficult since the intervals to use are not easily found, and so are not reasonable step size strategies to use. Thus, I rule out exact line search.

I then tried fixed step sizes, but at first ran into math overflow errors as well. However, I realised my step sizes were simply too big. I then used step sizes 0.001, 0.0001 and 0.0005, and was able to obtain a solution for these. Using these fixed step sizes, and a stopping criterion of 200, I was able to obtain 93% accuracy, and a relatively fast convergence of 100-200 iterations. However, when I observed the value of the norm of d through each iteration using fixed step size, I realised that the norm of d was not monotonically decreasing, e.g at iteration 83 norm d= 594, but at iteration 95 norm d = 740, and it would jump periodically. Investigating further, I lowered the stopping criterion to 100 and observed the objective function values at every iteration. As we can see by the plot in part g, the objective function does not monotonically decrease! Not only that, but it cannot hit the stopping criterion of 100 (Or it might take an inane amount of iterations): I ran it for

30,000+ iterations before finally manually halting the program. This suggests that fixed step sizes are not optimal, and it will be almost impossible for fixed step size strategy to obtain a low value of norm d (grad at current w).

Using armijo's I was able to obtain similar results as compared to fixed step sizes, obtaining about 93% accuracy, although steepest descent with armijo's rule converged more slowly than a fixed step size strategy. However, when I searched for the best parameters and initial solutions, I found that with a few conditions, Armijo's was able to halt with a stopping criterion of 100 within 178 iterations with 93.6% accuracy! This, coupled with the fact that the objective function value under armijo's rule monotonically decreases (see part g), leads me to conclude that armijo's rule is the best step size strategy here(that I have tried, at least).

(d)

To choose the initial solution, I considered 3 possibilities. Positive, negative and 0. To simplify my picks, I tried steepest descent with armijo's rule and different values of beta and sigma with a vector of 1s, -1s, and 0s, and a stopping criterion of 100.
Test accuracy wise, 0 has the lowest, with 0.929 accuracy for nearly all combinations of beta and sigma. It also is the slowest, requiring a high number of iterations in general(around 600+ at minimum) to hit it's stopping criterion of 100.
Next is -1. It has around 0.929-0.932 test accuracy, and the time taken can vary, taking 26 seconds at the fastest, and 152 seconds at its slowest. However, the vector of 1s is far and away the best choice. All combinations of beta and sigma took less than 50 seconds to complete, and the best combination for test accuracy had a 0.9368 accuracy, which ran in 24 secs, a major leg up above both 0s and -1s. Thus, I choose a vector of 1s for my initial solution.

(e)

I initially chose a stopping criterion of 10, however, it took an unreasonable amount of time to compute on some step size strategies, and I increased it to 200. I found that at 200, I was able to achieve test accuracies of about 92%, and at a stopping criterion of 10 I got a best test accuracy of 94.3%, but at the cost of 1500 iterations more. Thus, I compromised, and used 100 as a stopping criterion, which at best gave me a 93.7% test accuracy and a 94+% train accuracy. Thus it gives me high accuracy of predictions, at a relatively short runtime. However, I cannot deny that a lower stopping criterion gives higher test accuracy, thus in the table for part f, I have included 1 version of steepest descent with armijo's rule and a stopping criterion of 10.

(f)

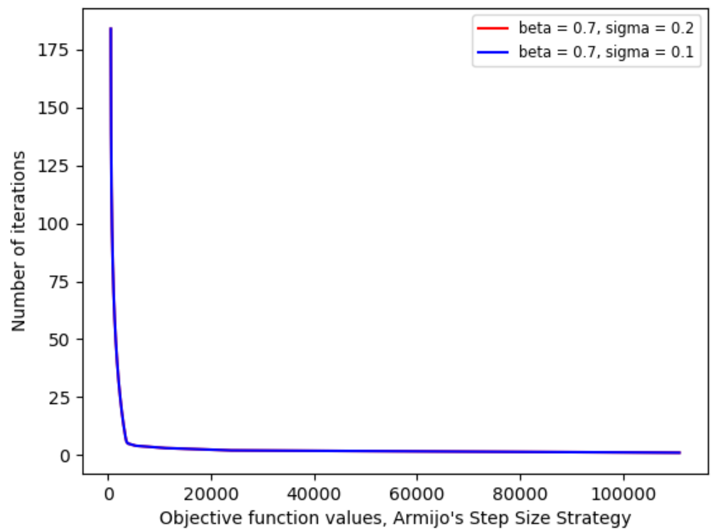

Figure 2: Plot of objective function vs number of iterations using Armijo's step size strategy

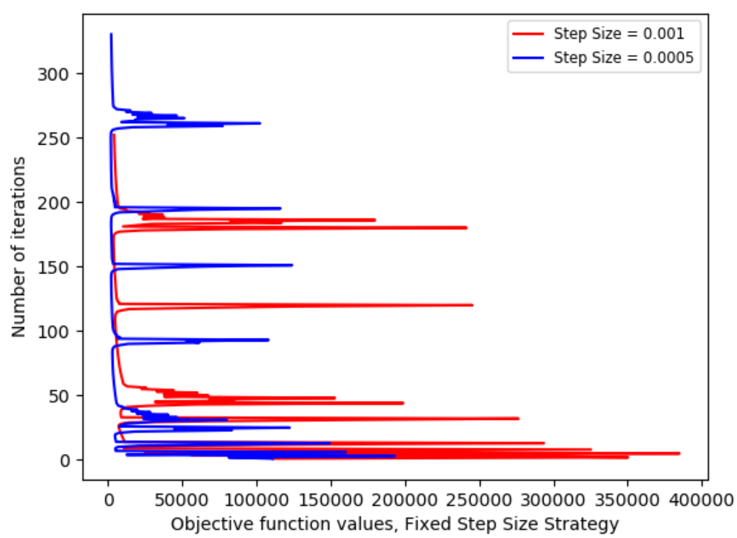

Figure 3: Plot of objective function vs number of iterations using fixed step size strategy

(g)

These results are some of the best of the results I have obtained exploring different combinations of beta and sigma values for my chosen step size strategy, armijo's rule. Initial solution is a vector of 1s, and with a stopping criterion of 100 except for the last entry.

Table 1: Accuracy, iterations for steepest descent with armijo's rule, initial = 1

Step Size Strategy	Time Taken	Iterations	Train Accuracy	Test Accuracy
Armijo beta = 0.7, sigma = 0.1	24.866	178	0.9452	0.9368
Armijo beta = 0.7, sigma = 0.4	23.955	172	0.9455	0.9355
Armijo beta = 0.6, sigma = 0.1	28.107	193	0.9442	0.9361
Armijo beta = 0.7, sigma = 0.1, (10)	364.127	1623	0.9481	0.9433

Where the last entry has the same armijo's parameters as the first entry, albeit with a stopping criterion of 10 instead of 100.

Appendix

```
# made using grid search, contains results talked about in
# parts c,d,e
acc_dict = {'Armijo beta = 0.1, sigma = 0.1, initial = 0':
    (0.9290364583333334, 195.51127672195435, 1414),
            'Armijo beta = 0.1, sigma = 0.2, initial = 0':
    (0.9290364583333334, 196.40087056159973, 1414),
            'Armijo beta = 0.1, sigma = 0.3, initial = 0':
    (0.9290364583333334, 195.7096071243286, 1414),
            'Armijo beta = 0.1, sigma = 0.4, initial = 0':
    (0.9290364583333334, 195.77836728096008, 1414),
            'Armijo beta = 0.2, sigma = 0.1, initial = 0':
    (0.9290364583333334, 121.56685972213745, 883),
            'Armijo beta = 0.2, sigma = 0.2, initial = 0':
    (0.9290364583333334, 121.77729296684265, 883),
            'Armijo beta = 0.2, sigma = 0.3, initial = 0':
    (0.9290364583333334, 121.65076088905334, 883),
            'Armijo beta = 0.2, sigma = 0.4, initial = 0':
    (0.9290364583333334, 121.62672233581543, 883),
            'Armijo beta = 0.3, sigma = 0.1, initial = 0':
    (0.9290364583333334, 266.47731757164, 1935),
            'Armijo beta = 0.3, sigma = 0.2, initial = 0':
    (0.9290364583333334, 266.22399044036865, 1937),
            'Armijo beta = 0.3, sigma = 0.3, initial = 0':
    (0.9290364583333334, 265.9267325401306, 1940),
            'Armijo beta = 0.3, sigma = 0.4, initial = 0':
    (0.9290364583333334, 269.9959063529968, 1940),
            'Armijo beta = 0.4, sigma = 0.1, initial = 0':
    (0.9290364583333334, 118.86411881446838, 863),
            'Armijo beta = 0.4, sigma = 0.2, initial = 0':
    (0.9290364583333334, 118.42331576347351, 863),
            'Armijo beta = 0.4, sigma = 0.3, initial = 0':
    (0.9290364583333334, 118.39830565452576, 863),
            'Armijo beta = 0.4, sigma = 0.4, initial = 0':
    (0.9290364583333334, 118.93395256996155, 863),
            'Armijo beta = 0.5, sigma = 0.1, initial = 0':
    (0.9290364583333334, 99.81205224990845, 724),
            'Armijo beta = 0.5, sigma = 0.2, initial = 0':
    (0.9290364583333334, 99.76034188270569, 724),
            'Armijo beta = 0.5, sigma = 0.3, initial = 0':
    (0.9290364583333334, 99.97069764137268, 724),
            'Armijo beta = 0.5, sigma = 0.4, initial = 0':
    (0.9290364583333334, 199.71015739440918, 1448),
            'Armijo beta = 0.6, sigma = 0.1, initial = 0':
    (0.9290364583333334, 89.46670746803284, 649),
            'Armijo beta = 0.6, sigma = 0.2, initial = 0':
    (0.9290364583333334, 89.38792634010315, 649),
            'Armijo beta = 0.6, sigma = 0.3, initial = 0':
    (0.9290364583333334, 148.95760369300842, 1082),
            'Armijo beta = 0.6, sigma = 0.4, initial = 0':
    (0.9290364583333334, 149.07728338241577, 1082),
            'Armijo beta = 0.7, sigma = 0.1, initial = 0':
    (0.9290364583333334, 84.01133441925049, 607),
```

```
            'Armijo beta = 0.7, sigma = 0.2, initial = 0': (0.9290364583333334, 120.2623519897461, 867),
            'Armijo beta = 0.7, sigma = 0.3, initial = 0': (0.9290364583333334, 119.86942911148071, 868),
            'Armijo beta = 0.7, sigma = 0.4, initial = 0': (0.9290364583333334, 120.01704430580139, 868),
            'Armijo beta = 0.8, sigma = 0.1, initial = 0': (0.9290364583333334, 101.90744209289551, 730),
            'Armijo beta = 0.8, sigma = 0.2, initial = 0': (0.9290364583333334, 101.70398688316345, 730),
            'Armijo beta = 0.8, sigma = 0.3, initial = 0': (0.9290364583333334, 101.40980052947998, 731),
            'Armijo beta = 0.8, sigma = 0.4, initial = 0': (0.9290364583333334, 126.6762318611145, 914),
            'Armijo beta = 0.9, sigma = 0.1, initial = 0': (0.9290364583333334, 89.11970686912537, 637),
            'Armijo beta = 0.9, sigma = 0.2, initial = 0': (0.9290364583333334, 89.3410484790802, 637),
            'Armijo beta = 0.9, sigma = 0.3, initial = 0': (0.9290364583333334, 99.19865036010742, 708),
            'Armijo beta = 0.9, sigma = 0.4, initial = 0': (0.9290364583333334, 110.02369856834412, 787),
            'Armijo beta = 0.1, sigma = 0.1, initial = 1': (0.9361979166666666, 34.913686752319336, 254),
            'Armijo beta = 0.1, sigma = 0.2, initial = 1': (0.9348958333333334, 30.813579320907593, 223),
            'Armijo beta = 0.1, sigma = 0.3, initial = 1': (0.9348958333333334, 30.897355318069458, 223),
            'Armijo beta = 0.1, sigma = 0.4, initial = 1': (0.9348958333333334, 30.549256324768066, 223),
            'Armijo beta = 0.2, sigma = 0.1, initial = 1': (0.9348958333333334, 40.04691982269287, 292),
            'Armijo beta = 0.2, sigma = 0.2, initial = 1': (0.9348958333333334, 37.74006009101868, 275),
            'Armijo beta = 0.2, sigma = 0.3, initial = 1': (0.9348958333333334, 37.774964809417725, 275),
            'Armijo beta = 0.2, sigma = 0.4, initial = 1': (0.9348958333333334, 37.694212913513184, 275),
            'Armijo beta = 0.3, sigma = 0.1, initial = 1': (0.935546875, 44.94977164268494, 324),
            'Armijo beta = 0.3, sigma = 0.2, initial = 1': (0.935546875, 43.878644704818726, 317),
            'Armijo beta = 0.3, sigma = 0.3, initial = 1': (0.935546875, 44.396249771118164, 317),
            'Armijo beta = 0.3, sigma = 0.4, initial = 1': (0.9348958333333334, 37.20848369598389, 271),
            'Armijo beta = 0.4, sigma = 0.1, initial = 1': (0.9348958333333334, 29.413305044174194, 214),
            'Armijo beta = 0.4, sigma = 0.2, initial = 1': (0.9348958333333334, 29.47915530204773, 214),
            'Armijo beta = 0.4, sigma = 0.3, initial = 1': (0.9348958333333334, 29.70651149749756, 215),
            'Armijo beta = 0.4, sigma = 0.4, initial = 1': (0.9348958333333334, 29.619826555252075, 215),
```

```
            'Armijo beta = 0.5, sigma = 0.1, initial = 1':
    (0.9348958333333334, 31.935614585876465, 232),
            'Armijo beta = 0.5, sigma = 0.2, initial = 1':
    (0.9348958333333334, 32.078205585479736, 233),
            'Armijo beta = 0.5, sigma = 0.3, initial = 1':
    (0.9348958333333334, 32.3614678382873 5, 234),
            'Armijo beta = 0.5, sigma = 0.4, initial = 1':
    (0.9348958333333334, 39.08145785331726, 281),
            'Armijo beta = 0.6, sigma = 0.1, initial = 1':
    (0.9361979166666666, 28.10730767250061, 193),
            'Armijo beta = 0.6, sigma = 0.2, initial = 1': (0.935546875,
    34.708186626434326, 235),
            'Armijo beta = 0.6, sigma = 0.3, initial = 1': (0.935546875,
    31.874711513519287, 230),
            'Armijo beta = 0.6, sigma = 0.4, initial = 1': (0.935546875,
    31.88076162338257, 230),
            'Armijo beta = 0.7, sigma = 0.1, initial = 1':
    (0.9368489583333334, 24.866475105285645, 178),
            'Armijo beta = 0.7, sigma = 0.2, initial = 1':
    (0.9368489583333334, 24.9862003326416, 178),
            'Armijo beta = 0.7, sigma = 0.3, initial = 1':
    (0.9361979166666666, 24.76472806930542, 177),
            'Armijo beta = 0.7, sigma = 0.4, initial = 1': (0.935546875,
    23.955928325653076, 172),
            'Armijo beta = 0.8, sigma = 0.1, initial = 1': (0.935546875,
    24.680019855499268, 175),
            'Armijo beta = 0.8, sigma = 0.2, initial = 1': (0.935546875,
    25.08188533782959, 177),
            'Armijo beta = 0.8, sigma = 0.3, initial = 1': (0.935546875,
    24.94025468826294, 176),
            'Armijo beta = 0.8, sigma = 0.4, initial = 1':
    (0.9361979166666666, 29.952916860580444, 212),
            'Armijo beta = 0.9, sigma = 0.1, initial = 1': (0.935546875,
    25.394136667251587, 175),
            'Armijo beta = 0.9, sigma = 0.2, initial = 1': (0.935546875,
    25.42099952697754, 175),
            'Armijo beta = 0.9, sigma = 0.3, initial = 1': (0.935546875,
    25.49980592727661, 176),
            'Armijo beta = 0.9, sigma = 0.4, initial = 1': (0.935546875,
    28.44990372657776, 197),
            'Armijo beta = 0.1, sigma = 0.1, initial = -1': (0.9296875,
    152.00957250595093, 1101),
            'Armijo beta = 0.1, sigma = 0.2, initial = -1': (0.9296875,
    151.39109420776367, 1101),
            'Armijo beta = 0.1, sigma = 0.3, initial = -1': (0.9296875,
    152.66269373893738, 1110),
            'Armijo beta = 0.1, sigma = 0.4, initial = -1': (0.9296875,
    159.33481121063232, 1164),
            'Armijo beta = 0.2, sigma = 0.1, initial = -1': (0.9296875,
    91.6049964427948, 666),
            'Armijo beta = 0.2, sigma = 0.2, initial = -1': (0.9296875,
    91.62103128433228, 666),
            'Armijo beta = 0.2, sigma = 0.3, initial = -1': (0.9296875,
    91.52620959281921, 666),
```

```
            'Armijo beta = 0.2, sigma = 0.4, initial = -1': (0.9296875, 91.57408046722412, 666),
            'Armijo beta = 0.3, sigma = 0.1, initial = -1': (0.9309895833333334, 73.21619772911072, 531),
            'Armijo beta = 0.3, sigma = 0.2, initial = -1': (0.9296875, 61.06863021850586, 445),
            'Armijo beta = 0.3, sigma = 0.3, initial = -1': (0.9296875, 61.27112674713135, 445),
            'Armijo beta = 0.3, sigma = 0.4, initial = -1': (0.9296875, 61.77374505996704, 449),
            'Armijo beta = 0.4, sigma = 0.1, initial = -1': (0.9309895833333334, 37.69517493247986, 274),
            'Armijo beta = 0.4, sigma = 0.2, initial = -1': (0.9309895833333334, 37.940553426742554, 275),
            'Armijo beta = 0.4, sigma = 0.3, initial = -1': (0.9309895833333334, 38.10611057281494, 275),
            'Armijo beta = 0.4, sigma = 0.4, initial = -1': (0.9309895833333334, 38.23473882675171, 278),
            'Armijo beta = 0.5, sigma = 0.1, initial = -1': (0.9309895833333334, 39.05155420303345, 283),
            'Armijo beta = 0.5, sigma = 0.2, initial = -1': (0.9309895833333334, 38.90597224235535, 283),
            'Armijo beta = 0.5, sigma = 0.3, initial = -1': (0.9309895833333334, 39.05856370925903, 284),
            'Armijo beta = 0.5, sigma = 0.4, initial = -1': (0.9309895833333334, 40.121748208999634, 292),
            'Armijo beta = 0.6, sigma = 0.1, initial = -1': (0.9303385416666666, 28.228463649749756, 204),
            'Armijo beta = 0.6, sigma = 0.2, initial = -1': (0.9322916666666666, 26.34553861618042, 190),
            'Armijo beta = 0.6, sigma = 0.3, initial = -1': (0.9303385416666666, 42.08839464187622, 303),
            'Armijo beta = 0.6, sigma = 0.4, initial = -1': (0.9303385416666666, 42.59607529640198, 303),
            'Armijo beta = 0.7, sigma = 0.1, initial = -1': (0.9309895833333334, 38.732409715652466, 278),
            'Armijo beta = 0.7, sigma = 0.2, initial = -1': (0.9303385416666666, 36.570191860198975, 263),
            'Armijo beta = 0.7, sigma = 0.3, initial = -1': (0.9322916666666666, 32.94886827468872, 236),
            'Armijo beta = 0.7, sigma = 0.4, initial = -1': (0.9303385416666666, 32.43821382522583, 233),
            'Armijo beta = 0.8, sigma = 0.1, initial = -1': (0.9309895833333334, 31.97647714614868, 227),
            'Armijo beta = 0.8, sigma = 0.2, initial = -1': (0.9309895833333334, 34.62938165664673, 246),
            'Armijo beta = 0.8, sigma = 0.3, initial = -1': (0.9322916666666666, 33.516358852386475, 237),
            'Armijo beta = 0.8, sigma = 0.4, initial = -1': (0.9303385416666666, 33.26203012466431, 236),
            'Armijo beta = 0.9, sigma = 0.1, initial = -1': (0.9309895833333334, 30.66697907447815, 213),
            'Armijo beta = 0.9, sigma = 0.2, initial = -1': (0.9309895833333334, 33.178274393081665, 231),
```

```
            'Armijo beta = 0.9, sigma = 0.3, initial = -1':
    (0.9309895833333334, 32.55791258811951, 226),
            'Armijo beta = 0.9, sigma = 0.4, initial = -1':
    (0.9322916666666666, 34.49678921699524, 240)}
```