

COMP9417 – Machine Learning and Data Mining  
T2–2021

## Homework 2: Logistic Regression & Optimization

Daksh Mukhra



Q1

a) we first shall rewrite  $L(\beta, \beta_0)$  in a form we can further analyse.

$$\begin{aligned}
 L(\beta, \beta_0) &= \sum_{i=1}^n y_i \ln \left( \frac{1}{s(\beta_0 + \beta^T x_i)} \right) + (1-y_i) \ln \left( \frac{1}{1 - s(\beta_0 + \beta^T x_i)} \right) \\
 &= \sum_{i=1}^n y_i \ln \left( \frac{1}{s(\beta_0 + \beta^T x_i)} \right) + \ln \left( \frac{1}{1 - s(\beta_0 + \beta^T x_i)} \right) - y_i \ln \left( \frac{1}{1 - s(\beta_0 + \beta^T x_i)} \right) \\
 &= \sum_{i=1}^n \ln \left( \frac{1}{1 - s(\beta_0 + \beta^T x_i)} \right) + y_i \ln \left( \frac{1 - s(\beta_0 + \beta^T x_i)}{s(\beta_0 + \beta^T x_i)} \right)
 \end{aligned}$$

$$\rightarrow \text{But we know } s(z) = (1 + e^{-z})^{-1}$$

$$e^{-z} = \frac{1 - s(z)}{s(z)} \Rightarrow -z = \ln \left( \frac{1 - s(z)}{s(z)} \right)$$

Hence:

$$= \sum_{i=1}^n -\ln(1 - s(\beta_0 + \beta^T x_i)) - y_i \ln \left( \frac{s(\beta_0 + \beta^T x_i)}{1 - s(\beta_0 + \beta^T x_i)} \right)$$

$$= -1 \sum_{i=1}^n \ln \left( 1 - \frac{e^{-(\beta_0 + \beta^T x_i)}}{1 + e^{-(\beta_0 + \beta^T x_i)}} \right) + y_i (\beta_0 + \beta^T x_i)$$

$$\rightarrow \text{as } s(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{e^z + e^0} = \frac{e^z}{e^z + 1}$$

$$\therefore z = \ln \left( \frac{s(z)}{1 - s(z)} \right)$$

We have:

$$= -\sum_{i=1}^n -\ln(1 + e^{\beta_0 + \beta^T x_i}) + y_i (\beta_0 + \beta^T x_i)$$

$$\begin{aligned}
 &\ln \left( 1 - \frac{1}{1 + e^{-(\beta_0 + \beta^T x_i)}} \right) \\
 &= \ln \left( \frac{e^{-(\beta_0 + \beta^T x_i)}}{1 + e^{-(\beta_0 + \beta^T x_i)}} \right) \\
 &= \ln \left( \left( \frac{1 + e^{-(\beta_0 + \beta^T x_i)}}{e^{-(\beta_0 + \beta^T x_i)}} \right)^{-1} \right) \\
 &= -\ln \left( e^{-(\beta_0 + \beta^T x_i)} + 1 \right)
 \end{aligned}$$

$L(\beta_0, \beta)$

$$= \sum_{i=1}^n \ln(1 + e^{\beta_0 + \beta^T x_i}) - y_i (\beta_0 + \beta^T x_i) \quad (*)$$

$\rightarrow$  now consider what happens to each minimization criterion

when  $y=0, \tilde{y}=-1$ ,  $\$ y=1, \tilde{y}=1$ .

$\rightarrow$  When  $y=1, \tilde{y}=1$

① if  $y=1$  the loss for each observation can be written as:

$$\rightarrow \ln(1 + e^{\beta_0 + \beta^T x_i}) - y_i (\beta_0 + \beta^T x_i)$$

$$\rightarrow \ln(1 + e^{\beta_0 + \beta^T x_i}) - (1)(\beta^T x_i + \beta_0)$$

$$\rightarrow \ln(1 + e^{\beta_0 + \beta^T x_i}) - (\beta^T x_i + \beta_0).$$

② for sklearn minimization criterion, when  $\tilde{y} = 1$  then the loss for each observation can be written as.

$$\rightarrow \ln(1 + e^{(-\tilde{y}_i(w^T x_i + c))}) \text{ ignoring } \|w\|_2$$

$$\rightarrow \ln(1 + \frac{1}{\exp(w^T x_i + c)}) = \ln(\frac{\exp(w^T x_i + c) + 1}{\exp(w^T x_i + c)})$$

$$\rightarrow \ln(\exp(w^T x_i + c) + 1) - \ln(\exp(w^T x_i + c))$$

$$\rightarrow \ln(1 + e^{w^T x_i + c}) - (w^T x_i + c)$$

$\rightarrow$  Note that the form of the loss is the same for each obs of the form  $(x_i, y=1)$  between the two criterions.

when  $y = 0, \tilde{y} = -1$ .

③ if  $y = 0$  the loss for each obs can be written as:

$$\rightarrow \ln(1 + e^{\beta_0 + \beta^T x_i}) + 0 // \text{using } L(\beta, \beta_0) (*) \text{ from prev pg.}$$

$$\rightarrow \ln(1 + e^{\beta_0 + \beta^T x_i})$$

④ if  $\tilde{y} = -1$  the loss for each obs in the sklearn minimization case can be written as:  $\ln(1 + \exp(-(-1)w^T x_i + c)))$

$$\rightarrow \ln(1 + \exp(w^T x_i + c))$$

$$\rightarrow \ln(1 + e^{c + w^T x_i})$$

$\rightarrow$  Note the form of the loss for each criterion is the same.

$\Rightarrow$  As the form of loss is the same between the minimization criterions for each respective observation set:

$(x_i : y=0 \vee \tilde{y}=-1), \quad \times$

$(x_i : y=1 \vee \tilde{y}=1)$

they will produce the same coefficients during the minimization procedures as you're effectively minimizing the same criterion. Hence  $(\hat{\beta}_0 = c \neq \hat{\beta} = \hat{w})$

## Question 1. Regularized Logistic Regression & the Bootstrap

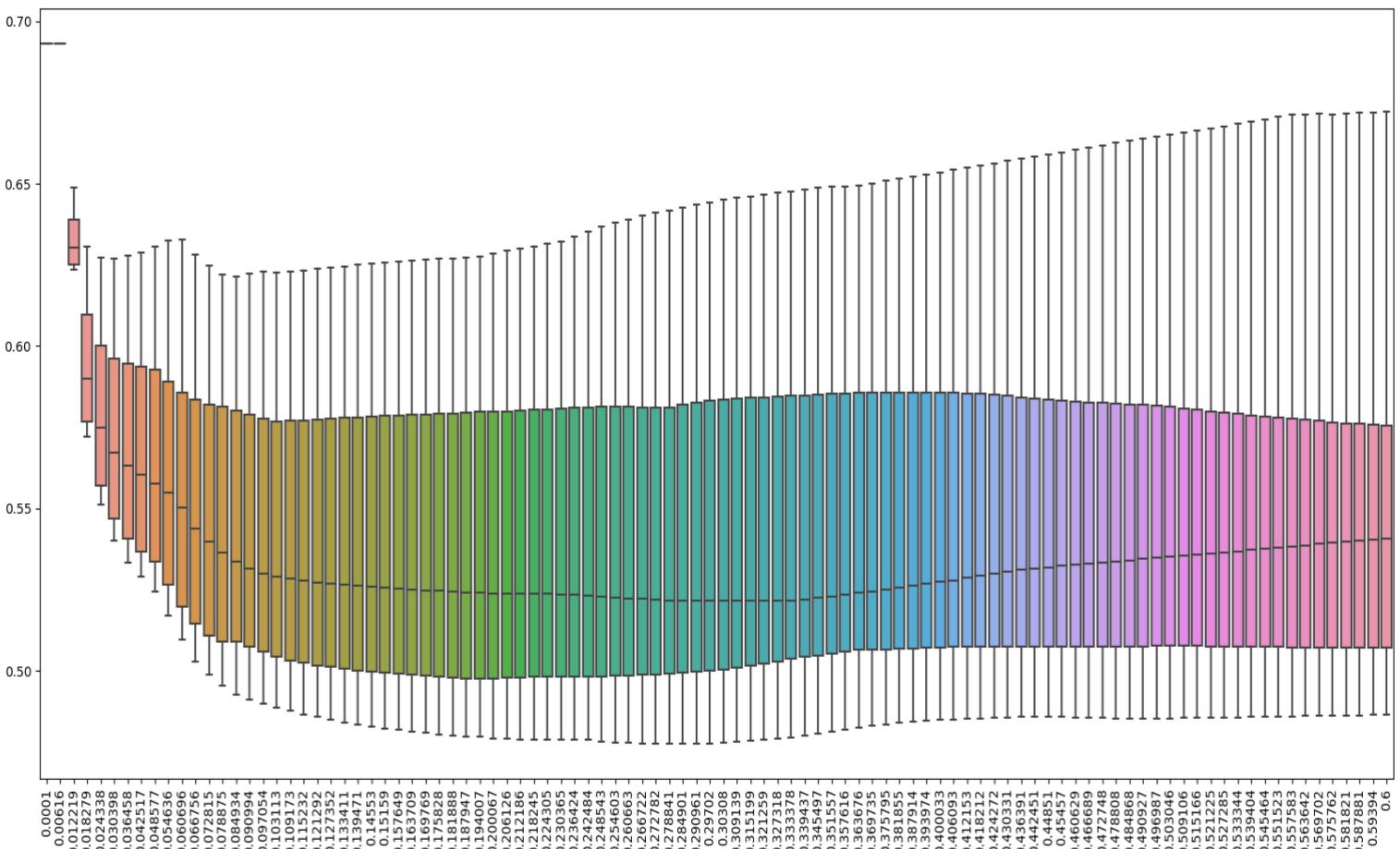
### Q1 a) continued:

A vector ( $w$  in this case) is said to be sparse when most of its cells ( $w_i$ 's in this case) are zero. This is the advantage of L1 regularization which minimizes a function that penalizes large values of the parameters and takes some weights to zero. Most often the function is  $\lambda \sum \Theta_j^2$ , which is some constant  $\lambda$  times the sum of the squared parameter values  $\Theta_j^2$ . The larger  $\lambda$  is the less likely it is that the parameters will be increased in magnitude simply to adjust for small perturbations in the data. Here the parameter  $C$  is simply the inverse of regularization strength in Logistic Regression i.e.,  $C=1/\lambda$ .

We can see that large values of  $C$  give more freedom to the model. Conversely, smaller values of  $C$  constrain the model more. In the L1 penalty case, this leads to sparser solutions. (scikit-learn.org)

Q1 b) Graph indicating the cross-validation scores for each  $C$  value in the form of a box plot.

Choice of  $C$  for each 10 Fold CV



```
MIN avg =  0.5397000191672515 -----
best C =  0.18794747474747472 -----
train accuracy = 0.752 -----
test accuracy = 0.74 -----
best para grd search {'C': 0.1818878787878777}
```

The value of C that gives the best CV performance is 0.18794. The training accuracy is 0.752 and Test accuracy is 0.74. Also depicted to is the minimum average cross validation score out of all the C choices and the best C value found through sklearn grid search.

Code used for this section:

```
19  data = pd.read_csv("Q1.csv")
20  X = data.iloc[:, :-1]
21  Y = data.iloc[:, -1]
22
23  X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.23, train_size=0.77, shuffle=False, stratify=None)
24  data_train = pd.concat([X_train, Y_train], axis=1)
25
26  #cross validation on training set ...
27  def cross_validation_split(dataset, folds):
28      dataset_split = [] # construct an array of arrays
29      df_copy = dataset
30      fold_size = int(df_copy.shape[0] / folds)
31
32      j = 0 # counter for jth row
33      for i in range(folds):
34          fold = []
35          # while loop to add elements to the folds
36          while len(fold) < fold_size:
37              # save the selected line
38              fold.append(df_copy.loc[j].values.tolist())
39              # delete the selected line from dataframe not to select again
40              df_copy = df_copy.drop(j)
41              #print(df_copy)
42              j = j+1
43          # save the fold
44          dataset_split.append(np.asarray(fold))
45  return dataset_split
46
47  def cross_validation_split_Y(dataset, folds):
48      dataset_split = []
49      df_copy = dataset
50      fold_size = int(df_copy.shape[0] / folds)
51      j = 0
52      for i in range(folds):
53          fold = []
54          while len(fold) < fold_size:
55              fold.append(df_copy.loc[j].tolist())
56              df_copy = df_copy.drop(j)
57              j = j+1
58          dataset_split.append(np.asarray(fold))
59  return dataset_split
60
61  data_k_X=cross_validation_split(X_train,10)
62  data_k_Y=cross_validation_split(Y_train,10)
63
64  # CV to find best C parameter
65  C_s = np.linspace(start=0.0001, stop=0.6, num=100, endpoint=True)
66  print(C_s)
67
68  logreg = LogisticRegression(penalty='l1', solver='liblinear', random_state=0)
69  avg_error_C = []
70  log_errors_all = []
```

Code continued:

```
64 # CV to find best C parameter
65 C_s = np.linspace(start=0.0001, stop=0.6, num=100, endpoint=True)
66 print(C_s)
67
68 logreg = LogisticRegression(penalty='l1', solver='liblinear', random_state=0)
69 avg_error_C = []
70 log_errors_all = []
71
72 i = 0
73 for a in C_s:
74     log_errors = []
75     while i < 10:
76         # get x and y for the ith fold
77         fold_X = data_k_X[i]
78         fold_Y = data_k_Y[i]
79         # create a 'ith removed' fold
80         ith_removed_X = np.delete(data_k_X, i, axis=0) # dataset without i-th observation
81         ith_removed_Y = np.delete(data_k_Y, i, axis=0)
82         #shaping
83         new_arr_X = ith_removed_X.reshape(-1, ith_removed_X.shape[-1])
84         ith_removed_X_df = pd.DataFrame(new_arr_X)
85         ith_removed_Y = ith_removed_Y.reshape(-1)
86         #fit model to df with out the ith fold
87         logreg.set_params(C = a) # set to current C
88         logreg.fit(ith_removed_X_df, ith_removed_Y)
89
90         # error based on the dropped fold i.e predict using the dropped fold
91         y_pred_probs_t = logreg.predict_proba(fold_Y)
92         loss_test = log_loss(fold_Y, y_pred_probs_t)
93
94         log_errors.append(loss_test)
95         i = i+1
96     i = 0
97     avg = (sum(log_errors))/10 # find the Leave one out error average for lambda = a
98     avg_error_C.append(avg)
99     log_errors_all.append(log_errors)
100
101 df = pd.DataFrame(log_errors_all)
102
103 #box plot creation
104 df_new = df.T
105 round_C_s = [round(num, 6) for num in C_s]
106 df_new.columns = round_C_s
107 print(df_new)
108
109 fig, ax = plt.subplots(figsize=(20,10))
110 sns.boxplot(data=df_new)
111 plt.title('Choice of C for each 10 Fold CV', fontsize=20, y=1.02)
112 plt.xticks(rotation='vertical')
113 plt.tight_layout()
114 plt.show()
```

```
119 # find best C
120 minavg = min(avg_error_C)
121 minavg_index = avg_error_C.index(min(avg_error_C))
122 best_C = C_s[minavg_index]
123 print(f"MIN avg = {minavg} ")
124 print(f"best C = {best_C} ")
125 # Re-fit the model with this chosen C, and report both train and test accuracy
126 logreg.set_params(C = best_C)
127 logreg.fit(X_train, Y_train)
128 #traning error
129 score_train = accuracy_score(Y_train, logreg.predict(X_train))
130 # test error
131 score_test = accuracy_score(Y_test, logreg.predict(X_test))
132 print(f"train accuracy = {score_train} ")
133 print(f"test accuracy = {score_test} ")
```

### Q1 c)

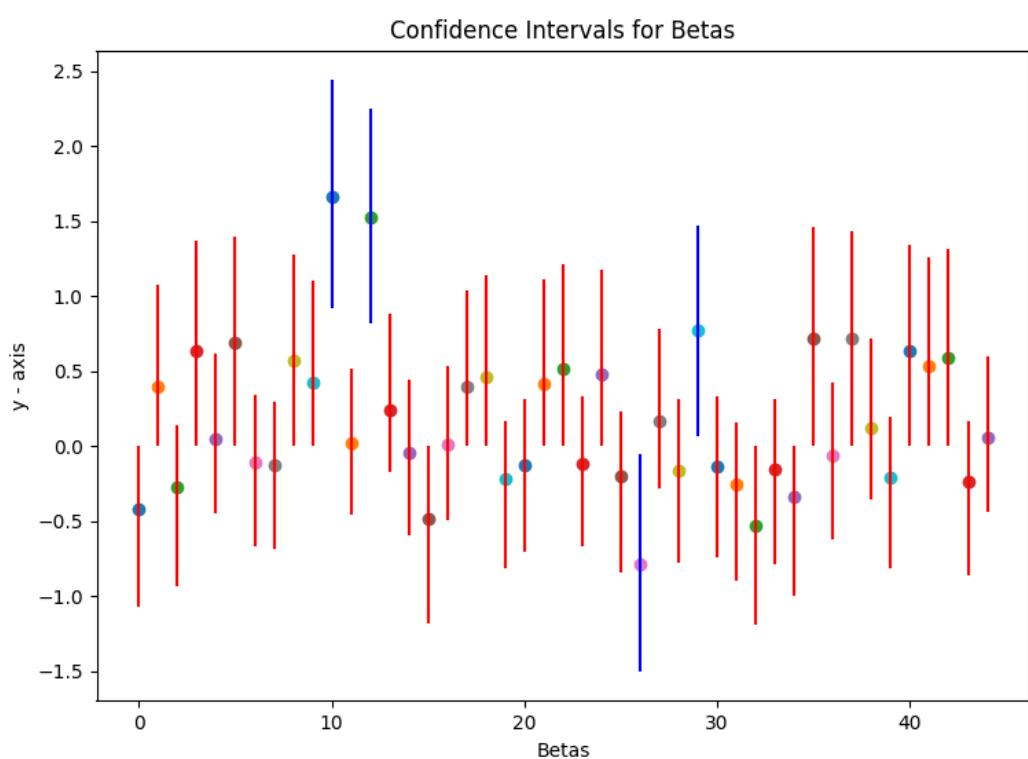
Code used to produce approximately consistent results compared to Q1 b)

```
137 #Q1 c)
138 parameter_candidates = [
139     {'C': C_s},
140 ]
141 grid_lr = GridSearchCV(estimator= LogisticRegression(penalty='l1' , solver='liblinear',random_state=0),
142                         cv=10,scoring='neg_log_loss' ,param_grid=parameter_candidates)
143 grid_lr.fit(X_train, Y_train)
144 best_parameters = grid_lr.best_params_
145 print(f"best para grd search {best_parameters}")
146
```

The modified code above produces the best C value of 0.181887 which is very similar to the result obtained in Q1b of 0.18794. The first reason the modified code obtains closer results to cross validation procedure made from scratch is due specifying the scoring function to be the neg\_log\_loss which was not specified in the original code. However, although it produces approximate results it doesn't produce the exact same result due to grid search using randomized folds in its cross-validation procedure whereas the from scratch procedure used a fixed fold selection procedure.

### Q1 d)

Plot produced for this section



Code used to produce the above plot:

```
149 #Q1 d)
150 np.random.seed(12)
151 i = 0
152 coeffs = list()
153 coeffs_avg = list()
154 logreg = LogisticRegression(penalty='l1', C=1, solver='liblinear', random_state=0)
155
156 while i < 10000:
157     choices_X = list()
158     choices_Y = list()
159     features = X_train.columns
160     response = ['Y']
161     choice_indices = np.random.choice(len(data_train), 500, replace=True)
162     print(f"--{i}--")
163
164     for j in choice_indices:
165         choices_X.append(X_train.loc[j].values.tolist())
166         choices_Y.append(Y_train.loc[j].tolist())
167     choices_X_df = pd.DataFrame(choices_X, columns=features)
168     logreg.fit(choices_X_df, choices_Y)
169     # dont need intercept
170     temp = logreg.coef_
171     coeffs.append(temp[0]) # use .tolist if need be
172     #print(coeffs)
173     i = i+1
174
175 coeff_np = np.array(coeffs)
176 print("===== mean for each beta")
177 mean_for_each_beta = np.mean(coeff_np, axis = 0)
178 print(np.mean(coeff_np, axis = 0))
179 print("===== lower quantiles for each beta ")
180 L_for_each_beta = np.quantile(coeff_np, .05, axis =0)
181 print(L_for_each_beta)
182 print("===== upper quantiles for each beta")
183 U_for_each_beta = np.quantile(coeff_np, .95, axis =0)
184 print(U_for_each_beta)
185
186
187 #plotting
188 plt.figure(figsize=(10,10))
189 i = 0
190 while i < 45:
191     if L_for_each_beta[i] <= 0 <= U_for_each_beta[i]:
192         plt.vlines(i, L_for_each_beta[i], U_for_each_beta[i], colors='r', linestyles='solid')
193         plt.scatter(i, mean_for_each_beta[i])
194     else:
195         plt.vlines(i, L_for_each_beta[i], U_for_each_beta[i], colors='b', linestyles='solid')
196         plt.scatter(i, mean_for_each_beta[i])
197     i = i+1
198 plt.xlabel(r"Betas")
199 plt.ylabel("y - axis")
200 plt.title("Confidence Intervals for Betas")
201 plt.show()
```

### Q1 e)

*What do the confidence intervals tell you about the underlying data generating distribution?*

The data generating distribution refers to the relationship between the response Y and the covariates X and this ultimately captured in the Beta Coefficients. From the confidence intervals we can be 95% confident that the true coefficient parameter falls between a certain specified range. If the range contains zero, then we hypothesise that there is no relationship between that covariate attached to the beta and y variable. Hence, we can narrow down all the covariates that do matter by looking at the blue lines on the plot given in Q1 e)

*How does this relate to the choice of C when running regularized logistic regression on this data?* This relates to the choice of C as the greater the C the more freedom a model is given i.e., more Beta coefficients remain significant while if the C is very low more coefficients turn to zero which is exactly what's happening in our case with C=1.

*Is regularization necessary?* In our case yes. The exception is if you know the data generating process and can model it exactly. Then you merely estimate the model parameters. In general, you will not know the process, so you will have to approximate with a flexible enough model. **We also don't want the model to memorize the training dataset, we want a model that generalizes well to new, unseen data.** Regularization methods introduce bias into the regression solution that can reduce variance considerably. The lower variance solutions produced by regularization techniques provide superior MSE performance which is better for out of sample prediction and avoids overfitting.

Q 2) a) deriving an equation for the explicit gradient update.

→ The general framework :  $x^{k+1} = x^k - \alpha_k \nabla f(x_k)$   $k=0,1,2\dots$   
where  $\alpha_k > 0$ .

$$\rightarrow f(x) = \frac{1}{2} \|Ax - b\|_2^2 = \frac{1}{2} (Ax - b)^T (Ax - b).$$

$$= \frac{1}{2} (x^T A^T - b^T)(Ax - b)$$

$$= \frac{1}{2} (x^T A^T A x - x^T A^T b - b^T A x + b^T b)$$

→ But  $x^T A^T b$  is a scalar, we know the transpose of a scalar is itself.  $\therefore (x^T A^T b)^T = x^T A^T b$   
 $b^T A x = x^T A^T b$ .

$$\rightarrow f(x) = \frac{1}{2} (x^T A^T A x - 2b^T A x + b^T b)$$

$$\nabla f(x) = \frac{\partial}{\partial x} \left[ x^T A^T x - 2(A^T b)^T x + b^T b \right] \times \frac{1}{2}.$$

$$= (A^T A x + (A^T A)^T x - 2A^T b) \times \frac{1}{2}.$$

$$= (A^T A x + A^T A x - 2A^T b) \times \frac{1}{2}.$$

$$= (2A^T A x - 2A^T b) \times \frac{1}{2}$$

$$\nabla f(x) = A^T A x - A^T b.$$

Hence our GD equation is:

$$x^{k+1} = x^k - \alpha_k [A^T A x^k - A^T b]$$

We can also ~~derive~~ derive an explicit solution for the minimum

vector by setting  $\nabla f(x) = 0$ .

$$\rightarrow \nabla f(x) = 0 \Rightarrow A^T A x = A^T b$$

$$x = (A^T A)^{-1} A^T b.$$

## Question 2. Gradient Based Optimization

### Q2 a) continued

Results:

```
====first 5=====
k = 0 --- x(k) = [1 1 1 1]
k = 1 --- x(k) = [1. 0.5 0. 1.5]
k = 2 --- x(k) = [ 1.2 0.25 -0.25 1.45]
k = 3 --- x(k) = [ 1.345 0.125 -0.36 1.44 ]
k = 4 --- x(k) = [ 1.4565 0.0625 -0.4075 1.459 ]
====last 5=====
k = 218 --- x(k) = [ 3.99699850e+00 -6.82054860e-17 -5.61531549e-04 2.99812156e+00]
k = 219 --- x(k) = [ 3.99709142e+00 -6.82054860e-17 -5.44147417e-04 2.99817971e+00]
k = 220 --- x(k) = [ 3.99718146e+00 -1.12614407e-16 -5.27301471e-04 2.99823607e+00]
k = 221 --- x(k) = [ 3.99726872e+00 -6.82054860e-17 -5.10977048e-04 2.99829068e+00]
k = 222 --- x(k) = [ 3.99735328e+00 -1.12614407e-16 -4.95158004e-04 2.99834359e+00]
```

Code used to produce the results:

```
# Q2 a
def get_gradient(x):
    A = np.array([[1, 0, 1, -1], [-1, 1, 0, 2], [0, -1, -2, 1]])
    b = np.array([1, 2, 3])
    # print(A)
    # print(b)
    AtA = np.matmul(np.transpose(A), A)
    AtAx = np.matmul(AtA, x)
    gradient = -np.matmul(np.transpose(A), b) + AtAx
    return gradient

def get_condition(x_curr):
    grad = get_gradient(x_curr)
    return la.norm(grad)

def gradient_descent(start_x, step):
    x_s = []
    condition = get_condition(start_x)
    x_s.append([0, start_x])

    k = 1
    xk = start_x

    while (condition >= 0.001):
        grad = get_gradient(xk)
        x_new = xk - step * grad
        x_s.append([k, x_new])

        condition = get_condition(x_new)
        xk = x_new # setting to new values
        k = k+1
        print(k)
    return x_s

start_x = np.array([1, 1, 1, 1])
results = gradient_descent(start_x , 0.1)
print("====first 5=====")
for num in results[:5]:
    print(f"k = {num[0]} --- x(k) = {num[1]}")

print("====last 5=====")
for num in results[-5:]:
    print(f"k = {num[0]} --- x(k) = {num[1]}")
```

Q2 b) deriving an explicit solution for  $\alpha_k$ .

→ Method of steepest descent / Exact line search.

→ we have to evaluate  $f(x^k - \alpha \nabla f(x^k))$

by taking the derivative wrt to  $\alpha$  & solve for  $\alpha$ .

$$f(x^k - \alpha \nabla f(x^k)) =$$

Note from 2a):  $f(x) = \frac{1}{2} (Ax - B)^T (Ax - B)$   
 $= \frac{1}{2} (x^T A^T A x - 2(A^T B)^T x + B^T B).$

$$\Rightarrow = \frac{1}{2} \left[ (x^k - \alpha \nabla f(x^k))^T A^T A (x^k - \alpha \nabla f(x^k)) \right. \\ \left. - 2(A^T B)^T (x^k - \alpha \nabla f(x^k)) + B^T B \right].$$

$$= \frac{1}{2} \left[ (x^{kT} - \alpha \nabla f(x^k)^T) A^T A (x^k - \alpha \nabla f(x^k)) \right. \\ \left. - 2(B^T A)(x^k - \alpha \nabla f(x^k)) + B^T B \right]$$

$$= \frac{1}{2} \left[ (x^{kT} A^T A - \alpha \nabla f(x^k)^T A^T A) (x^k - \alpha \nabla f(x^k)) \right. \\ \left. - 2 [B^T A x^k - \alpha B^T A \nabla f(x^k)] + B^T B \right]$$

$$= \frac{1}{2} \left[ x^{kT} A^T A x^k - \alpha x^{kT} A^T A \nabla f(x^k) - \alpha \nabla f(x^k)^T A^T A x^k + \alpha^2 \nabla f(x^k)^T A^T A \nabla f(x^k) \right. \\ \left. - 2 B^T A x^k + 2 \alpha B^T A \nabla f(x^k) + B^T B \right].$$

$$= \frac{1}{2} \left[ \alpha^2 \nabla f(x^k)^T A^T A \nabla f(x^k) + \alpha [-x^{kT} A^T A \nabla f(x^k) + \nabla f(x^k)^T A^T A x^k \right. \\ \left. + 2 \alpha B^T A \nabla f(x^k)] \right. \\ \left. + x^{kT} A^T A x^k - 2 B^T A x^k + B^T B \right]$$

But this can be written as a quadratic.

hence

$$= \frac{1}{2} [\alpha^2 + d\alpha + c] \stackrel{\text{arg min}}{\nabla} f(x^k - \alpha \nabla f(x^k)) = -\frac{d}{2a}.$$

Note:  $d = \nabla f(x^k)^T A^T A \nabla f(x^k)$  is just some constant



$$\bullet d = -x^k)^T A^T A \nabla f(x^k) - \nabla f(x^k)^T A^T A x^k + 2B^T A \nabla f(x^k)$$

which is some constant

$$\bullet c = x^k)^T A^T A x^k - 2B^T A x^k + B^T A \quad \text{which is a constant.}$$

$$\therefore \alpha_k = \frac{-d}{2a} = \frac{-[ -x^k)^T A^T A \nabla f(x^k) - \nabla f(x^k)^T A^T A x^k + 2B^T A \nabla f(x^k)]}{2 \nabla f(x^k)^T A^T A \nabla f(x^k)}$$

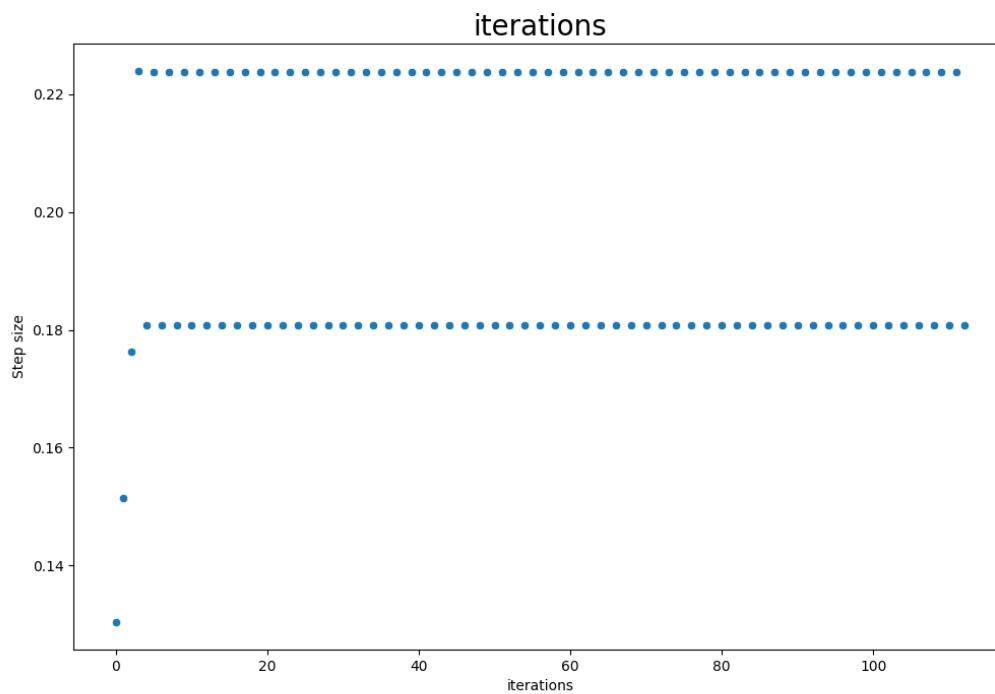
$$\alpha_k = \frac{x^k)^T A^T A \nabla f(x^k) + \nabla f(x^k)^T A^T A x^k + 2B^T A \nabla f(x^k)}{2 \nabla f(x^k)^T A^T A \nabla f(x^k)}.$$

## Q2 b) continued

Results

```
====first 5=====
k = 0 --- x(k) = [1 1 1 1]
k = 1 --- x(k) = [ 1.          0.34782609 -0.30434783  1.65217391]
k = 2 --- x(k) = [ 1.39494706  0.08452804 -0.33726008  1.32305136]
k = 3 --- x(k) = [ 1.50092477  0.01000748 -0.49745676  1.52586073]
k = 4 --- x(k) = [ 1.74348592e+00 -1.19743517e-03 -3.75654918e-01  1.49120345e+00]
====last 5=====
k = 109 --- x(k) = [ 3.99737876e+00  7.65011837e-18 -5.33380212e-04  2.99844552e+00]
k = 110 --- x(k) = [ 3.99762774e+00 -9.17438206e-17 -3.93561204e-04  2.99841486e+00]
k = 111 --- x(k) = [ 3.99769691e+00 -9.17438206e-17 -4.68641603e-04  2.99863420e+00]
k = 112 --- x(k) = [ 3.99791567e+00 -1.91137760e-16 -3.45793018e-04  2.99860726e+00]
k = 113 --- x(k) = [ 3.99797645e+00 -1.91137760e-16 -4.11760592e-04  2.99879997e+00]
```

Plot of ak over all iterations



Code used to plot:

```
131 start_x = np.array([1,1,1,1])
132 results = gradient_descent(start_x , 0.1)
133 print("====first 5=====")
134 for num in results[:5]:
135     print(f"k = {num[0]} --- x(k) = {num[1]}")
136
137 print("====last 5=====")
138 for num in results[-5:]:
139     print(f"k = {num[0]} --- x(k) = {num[1]}")
140
141 sns.scatterplot(data = steps)
142 plt.title('iterations', fontsize=20)
143 plt.xlabel('iterations')
144 plt.ylabel('Step size')
145 plt.show()
```

Code used for GD algorithm:

```
67 # Q2 b
68 def get_gradient(x):
69     A = np.array([[1 , 0 , 1 , -1],[-1, 1 ,0 , 2],[0, -1, -2, 1]])
70     b = np.array([1,2,3])
71     # print(A)
72     # print(b)
73     AtA = np.matmul(np.transpose(A),A)
74     AtAx = np.matmul(AtA,x)
75     gradient = -np.matmul(np.transpose(A),b) + AtAx
76     return gradient
77
78 steps = []
79 def get_step(grad_at_x, x_curr):
80     A = np.array([[1 , 0 , 1 , -1],[-1, 1 ,0 , 2],[0, -1, -2, 1]])
81     b = np.array([1,2,3])
82
83     term1 = -2*np.matmul(np.transpose(b),A)
84     term2 = np.matmul(term1,grad_at_x)
85
86     term3 = np.matmul(np.transpose(x_curr),np.transpose(A))
87     term4 = np.matmul(term3, A)
88     term5 = np.matmul(term4, grad_at_x)
89
90     term6 = np.matmul(np.transpose(grad_at_x), np.transpose(A))
91     term7 = np.matmul(term6, A)
92     term8 = np.matmul(term7 , x_curr)
93
94     numerator = term2 + term5 + term8
95
96     term9 = 2*np.matmul(np.transpose(grad_at_x),np.transpose(A))
97     term10 = np.matmul(term9,A)
98     term11 = np.matmul(term10,grad_at_x)
99
100    denominator = term11
101
102    step = numerator/denominator
103    steps.append(step)
104    return step
105
106 def get_condition(x_curr):
107     grad = get_gradient(x_curr)
108     return la.norm(grad)
109
110 def gradient_descent(start_x, step):
111     x_s = []
112     condition = get_condition(start_x)
113     x_s.append([0,start_x])
114
115     k = 1
116     xk = start_x
117
118     while (condition >= 0.001):
119         grad = get_gradient(xk)
120         step = get_step(grad, xk)
121         x_new = xk - step * grad
122         x_s.append([k,x_new])
123
124         condition = get_condition(x_new)
125         xk = x_new # setting to new values
126
127         print(k)
128         k = k+1
129     return x_s
```

## Q2 c)

*Comment on the differences you observed.*

The key differences observed was the number of steps it took to reach a convergence and the size of the step, which aligns with our definitions gradient descent will point in the direction of steepest descent while steepest descent (exact line search) takes a step

*Why would we prefer steepest descent over gradient descent?*

Fig 1

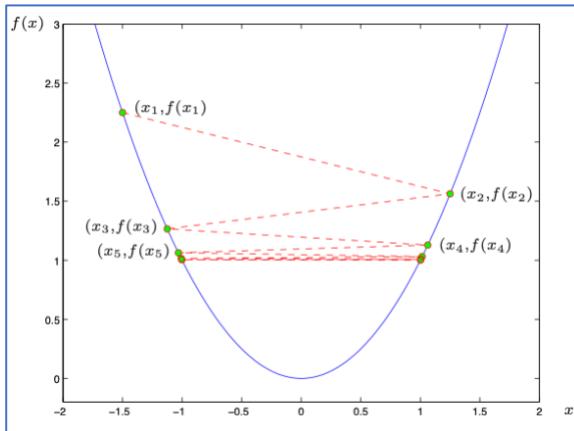
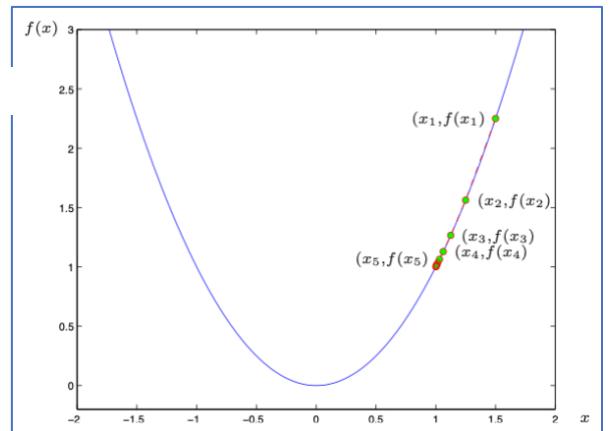


Fig 2



In steepest descent were finding the optimal step size at each iteration of the algorithm, However gradient descent step sizes are constant meaning one of the situations represented by the above figures may occur , either step is too long ( fig1) or too short (fig2) i.e. decrease in  $f$  is not proportional to the size of the directional derivative

*Why would you prefer gradient descent over steepest descent?*

There can be many issues with steepest descent (exact line search) such as Exact line search requires univariate minimization it's also generally very expensive and not cost effective. Also exact line search may not be much better than an approximate line search methods.

*Finally, explain why this is a reasonable condition to terminate use to terminate the algorithm.*

The termination condition is reasonable as  $\|\nabla f(x^k)\|$  is describing the gradient at each  $x$  if  $\nabla f(x^k) = 0$  then we have found the minimum as in our case the gradient may never converge to zero a small value is needed such as 0.0001 so that algorithm stops and we can achieve a ‘good’ enough estimate to the true minimum.

## Q2 d)

Code used for data pre-processing

```
152 #Q2d
153
154 df = pd.read_csv("Q2.csv")
155 df = df.dropna()
156 x_df = df.drop(columns=['transactiondate','latitude','longitude','price'])
157 y_df = df[["price"]]
158 scaler = MinMaxScaler()
159 scaled_x_df = scaler.fit_transform(x_df)
160
161 print(scaled_x_df)
162
163 scaled_x_df = pd.DataFrame(data=scaled_x_df)
164 print(scaled_x_df)
165 scaled_x_df["1"] = 1
166 first_col = scaled_x_df.pop("1")
167 scaled_x_df.insert(0, "1", first_col)
168
169 print(scaled_x_df)
170
171 x_train, x_test, y_train, y_test = train_test_split(scaled_x_df, y_df, test_size=0.5, shuffle=False)
172 print(x_train)
173
174 y_train = np.asarray(y_train)
175 y_test = np.asarray(y_test)
176 x_train = np.asarray(x_train)
177 x_test = np.asarray(x_test)
178 print(f"x trian first row {x_train[0]}")
179 print(f"x trian last row {x_train[-1]}")
180 print(f"y test first 1 {y_test[0]}")
181 print(f"y test last row 1 {y_test[-1]}")
182
183 print(f"y trian first row {y_train[0]}")
184 print(f"y trian last row {y_train[-1]}")
185 print(f"y test first 1 {y_test[0]}")
186 print(f"y test last row 1 {y_test[-1]}")
```

Q2 e)

Training loss 5.532811641693115

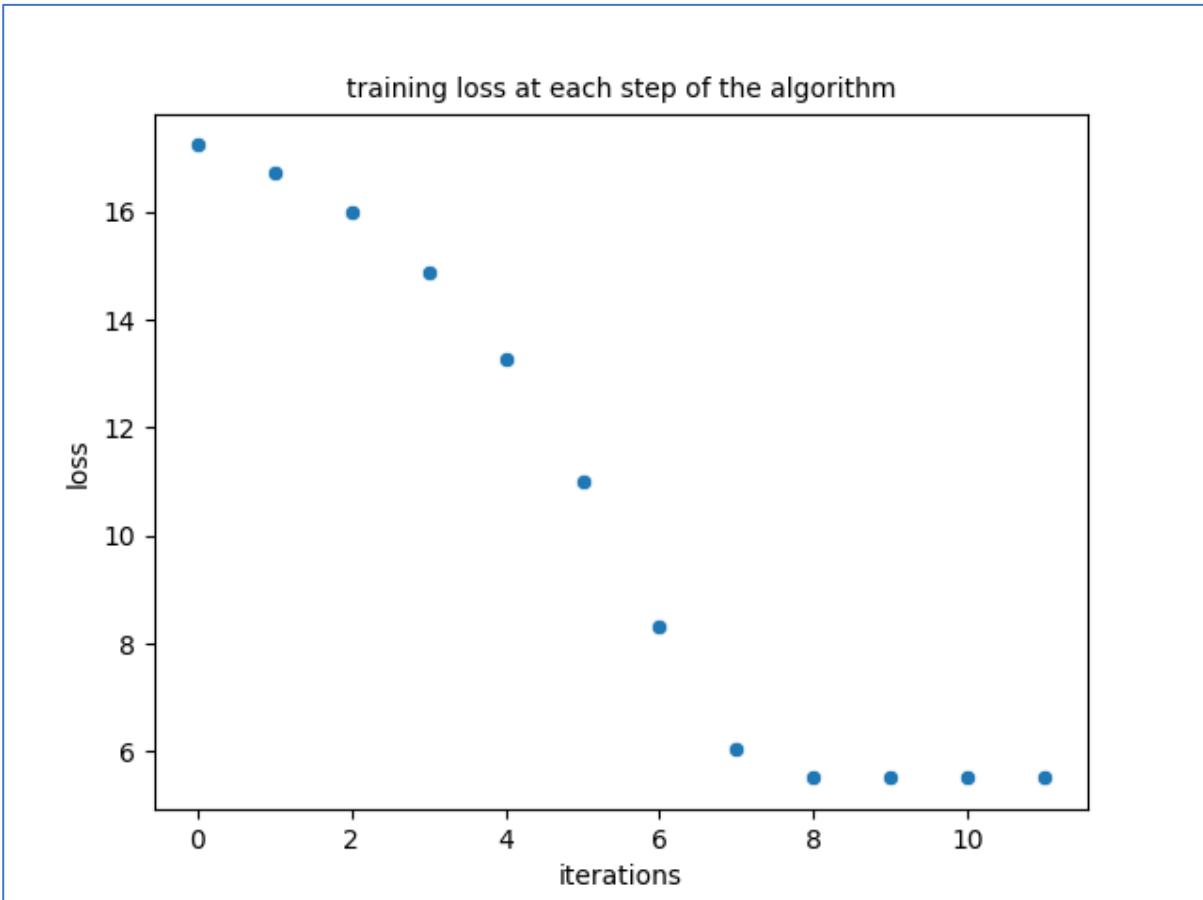
Testing loss 5.052703857421875

Final Iteration: 11

Absolute loss difference between final weight vector and vector before it: 6.4373016e-05

Final weight vector: [18.337734 18.337734 18.337734 18.337734]

Plot produced:



See code next page

Code used to produce results

```
188 # Q2e
189 def get_loss(x,y,w):
190     WtX = jnp.matmul(w, jnp.transpose(x))
191     sqrt_loss = 0.25*((y-WtX)**2)
192     sqrt_loss += 1
193     loss = jnp.sqrt(sqrt_loss)
194     loss -= 1
195
196     return jnp.sum(loss) # changes the shape of the array its just 1 number still
197
198 def get_loss_sum(x,y,w):
199     k = x.shape[0]
200     losses = []
201     res = []
202     for i in range(k):
203         loss = get_loss(x[i],y[i],w)
204         losses.append(loss)
205     res = jnp.mean(jnp.array(losses))
206     return res
207
208 def get_gradient(x,y,w):
209     k = x.shape[0]
210     losses = []
211     res = []
212     for i in range(k):
213         loss = grad(get_loss)(x[i],y[i],w)
214         losses.append(loss)
215     res = jnp.mean(jnp.array(losses))
216     return res
217 loss_all = []
218 def gradient_descent_train(w, x, y, step=1,termination_val=0.0001):
219     results = {}
220
221     converged = False
222     wk = w
223     k=0
224     initial_loss = get_loss_sum(x, y, wk)
225     cond = initial_loss
226     while cond >= 0.0001:
227         grad = get_gradient(x, y, wk)
228         wk_new = wk - step * grad
229         results[k] = wk_new
230
231         cond = abs(get_loss_sum(x, y, wk_new) - get_loss_sum(x, y, wk))
232         loss_for_wk = get_loss_sum(x, y, wk)
233         numpy_array = float(np.asarray(loss_for_wk))
234
235         loss_all.append(numpy_array)
236         print("Iteration: ", k)
237         #jnp mean is just changing shape of result here
238         print("weight_diff: ", jnp.mean(cond))
239         wk = wk_new
240         k+=1
241     return results, wk
242
243 w0 = np.array([1,1,1,1])
244 initial_step = 1
245 res, final_w = gradient_descent_train(w0, x_train, y_train, initial_step)
246 print(final_w)
247 print("-----loss all ")
248 print(loss_all)
249
250 #getting train loss
251 train_loss = float(np.asarray(get_loss_sum(x_train, y_train, final_w)))
252 test_loss = float(np.asarray(get_loss_sum(x_test, y_test, final_w)))
253
254 print(f"train_loss {train_loss}")
255 print(f"test_loss {test_loss}")
256
257
258 sns.scatterplot(data = loss_all)
259 plt.title('training loss at each step of the algorithm', fontsize=10)
260 plt.xlabel('iterations')
261 plt.ylabel('loss')
262 plt.show()
```

## Q2 g) Gradient Boosting Algorithms

Another broad category of gradient based algorithms is gradient boost algorithms. In boosting, we build an ensemble of classifiers or regressors incrementally. In each step, we add a new sub-model that tries to compensate for the errors made by the previous sub-models. To avoid overfitting, boosting is typically done using simple sub-models: the classical choice is small decision trees (“decision stumps”). Gradient boosting is when we optimize the boosted ensemble with respect to some loss function. different types of loss functions can be used resulting in a flexible technique that can be applied to regression, multi-class classification.

The difference between gradient descent and gradient boosting is that in GD you modify the predictions by e.g., adding a delta to the model coefficients; in gradient boosting you modify the predictions by adding a delta directly to the predictions. An example of a weak learner used is Decision trees. Trees are added together one at a time based on minimizing some loss, and existing trees in the model are not changed. XGBoost is one of the most popular variants of gradient boosting.