1.

$x^t[A]x < 0$ the matrix not semidefinite

$x^t[A]x = \sum_{i=1}^{d} \sum_{j=1}^{d} A^{(i,j)}x^{(i)}x^{(j)}$ =

$x^t[\begin{smallmatrix}1 & 2\\ 2 & 1\end{smallmatrix}]x = \begin{matrix}x^{(1)}\\ x^{(2)}\end{matrix}\begin{bmatrix}1 & 2\\ 2 & 1\end{bmatrix}x^{(1)} \quad x^{(2)} = (x^{(1)})^2 + 4x^{(1)}x^{(2)} + (x^{(2)})^2$

If $x^{(1)}$ = 0 and $x^{(2)}$=2 then $x^t[A]x$ would be < 0 so it would not be positive semidefinite

Consider A to be $\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}$ then $x^t[A]x = \begin{matrix}x^{(1)}\\ x^{(2)}\end{matrix}\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}x^{(1)} \quad x^{(2)}$ =

$4(x^{(1)})^2 + x^{(1)}x^{(2)} + 4(x^{(2)})^2$ which would be >0 for all values of $x^{(1)}$and $x^{(2)}$ so our new A would be PSD

2.

a. $\frac{d^2}{d\theta_{\mathrm{p}} d\theta_{\mathrm{n}}}$ $of\ n_1 log(\theta_{y=1}) + (\mathrm{n} - \mathrm{n1})\log(1-\theta_{y=1}) + \sum_{j=1}^{d}[r_0^j * log(\theta_{j|0}) +$ $(\mathrm{n} - n_1 - r_0^j)log(1-\theta_{j|0}) + r_1^j * log(\theta_{j|1}) + (\mathrm{n} - \mathrm{n1} - r_1^j)log(1- \theta_{j|1})]$ =

$$\sum_{j=1}^{d}[\, r_1^j/\theta_{j|1} - (\, n_1 - r_1^j)/(1- \theta_{j|1})\,]\ d\theta_{j|1}$$

$$\sum_{j=1}^{d}[\, r_0^j/\theta_{j|0} - (\, n - n_1 - r_1^j)/(1- \theta_{j|0})\,]\ d\theta_{j|0}$$

$$n_1/(\theta_{y=1}) - (n - n_1)/(1-\theta_{y=1})\ d\theta_{y=1}$$

A taking the 2d derivative $\frac{d^2}{d\theta_p d\theta_q}$ where p!=q we can see that taking the derivative of any of the above with any theta value not already used in the simplification of the problem would result in 0

b. -f(θ) = g (θ) = -$n_1 log(\theta_{y=1})$-$(\mathrm{n} - \mathrm{n1})\log(1-\theta_{y=1})$-$\sum_{j=1}^{d}\left[r_0^j * log(\theta_{j|0}) +\right.$ $\left.(\mathrm{n} - \mathrm{n}_1 - \mathrm{r}_0^{\mathrm{j}})\log(1-\Theta_{\mathrm{j|0}}) + \mathrm{r}_1^{\mathrm{j}} * \log(\Theta_{\mathrm{j|1}}) + (\mathrm{n} - \mathrm{n1} - \mathrm{r}_1^{\mathrm{j}})\log(1- \Theta_{\mathrm{j|1}})\right]$

$$\sum_{j=1}^{d}[\, r_1^j/\theta_{j|1}\ + (\, n_1 - r_1^j)/(1- \theta_{j|1})\,]\ d\theta_{j|1}$$

$$[\, r_1^j/(\theta_{j|1})\hat{}2\ + (\, n_1 - r_1^j)/(1- \theta_{j|1})\hat{}2\,]\ \frac{d(2)}{d\theta_{j|1}(2)} \leq 0 \Rightarrow g(\theta) \geq 0$$

$$\sum_{j=1}^{d}[\, r_0^j/\theta_{j|0} -(\, n - n_1 - r_1^j)/(1- \theta_{j|0})\,]\ d\theta_{j|0}$$

$$-r_0^j/(\theta_{j|0})\hat{}2 - (\, n - n_1 - r_1^j)/(1- \theta_{j|0})\hat{}2\ \frac{d(2)}{d\theta_{j|0}(2)} \leq 0 \Rightarrow g(\theta) \geq 0$$

$$-n_1/(\theta_{y=1}) + (n - n_1)/(1-\theta_{y=1})\ d\theta_{y=1}$$

$$n_1/(\theta_{y=1})\text{^}2\text{-}(n - n_1)/(1-\theta_{y=1})\text{^}2\ \frac{d(2)}{d\theta_{y=1}(2)} \leq 0 \Rightarrow g(\theta) \geq 0$$

3.

a. $\frac{1}{n}\sum_{\mathrm{i}=1}^{n}\left(y_i - \theta_i^T x_i\right)^2 = \frac{1}{n}\sum_{\mathrm{i}=1}^{n}(y_i\text{-h(x)})^2 = \frac{1}{n}\sum_{\mathrm{i}=1}^{n}(y_i\text{-}(\theta_0^T + \theta^T y_{\mathrm{x}}))^2$ ->
$\frac{1}{n}\sum_{\mathrm{i}=1}^{n} 2(y_i\text{-}(\theta_0^T + \theta^T x_i))(\text{-}x_i)\frac{d}{d\theta(\mathrm{j})}$

b. $\frac{d}{d\theta(\mathrm{i})d\theta(\mathrm{j})}\ \frac{1}{n}\sum_{j=1}^{n} 2(y_{\mathrm{j}}\text{-}(\theta_0^T + \theta^T x_{\mathrm{j}}))(\text{-}x_i)\frac{d}{d\theta(\mathrm{j})} = \frac{1}{n}\sum_{j=1}^{n} 2\left(x_{\mathrm{j}}\right)(\text{-}x_i)\frac{d}{d\theta(\mathrm{j})}\ \frac{d}{d\theta(\mathrm{i})} =$
$\frac{2}{n}\sum_{j=1}^{n}(x_{\mathrm{k}})(x_{\mathrm{k}})$ ^(T)=||x||^2 which is greater than or equal to 0 for all x so the function is psd so it is convex

###

1. data_mat=np.insert(data_mat,0,1.,axis =1)

```
from sklearn.feature_extraction.text import CountVectorizer

## Transform to bag of words representation.
vectorizer = CountVectorizer(analyzer = "word", tokenizer = None, preprocessor = None, stop_words = None, max_features = 4500)
data_features = vectorizer.fit_transform(sents_processed)

print ('The original size: ',data_features.shape)
```

```
The original size:  (3000, 4500)
```

```
## STUDENT: YOUR CODE STARTS HERE
# Task: Append '1' to the beginning of each vector.
# Hint: You can use data_features.toarray() to transform data_features into a numpy array
# The output should be a numpy array named data_mat

data_mat = data_features.toarray()

data_mat=np.insert(data_mat,0,1.,axis =1)

## STUDENT: CODE ENDS
print ('The updated size: ',data_mat.shape)
```

```
The updated size:  (3000, 4501)
```

2.

```
## STUDENT: Start of code ###

 #   - yx
 #  _____________
 #  ((e^yθ^Tx)+1)

 #initialize ld(theta)
 derivatives=np.zeros(weights.size)

 for i in range(labels.size):
    #dot product of weigths with each column of feature matrix
    column_product=np.dot(weights,feature_matrix[i,:])
```

```
        temp=labels[i]*column_product
        value = 1/(1+np.exp(temp))
        #take the sum for every road
        derivatives = derivatives+(- labels[i]*feature_matrix[i,:]*value)

    return derivatives

    # End of code ###
```

```
def weight_derivative(weights, feature_matrix, labels):
    # Input:
    # weights: weight vector w, a numpy vector of dimension d
    # feature_matrix: numpy array of size n by d, where n is the number of data points, and d is the feature dimension
    # labels: true labels y, a numpy vector of dimension d, each with value -1 or +1
    # Output:
    # Derivative of the regression cost function with respect to the weight w, a numpy array of dimension d

    ## STUDENT: Start of code ###

    #    - yx
    #   __________
    #    ((e^yθ^Tx)+1)

    #initialize ld(theta)
    derivatives=np.zeros(weights.size)

    for i in range(labels.size):
        #dot product of weigths with each column of feature matrix
        column_product=np.dot(weights,feature_matrix[i,:])
        temp=labels[i]*column_product
        value = 1/(1+np.exp(temp))
        #take the sum for every road
        derivatives = derivatives+(- labels[i]*feature_matrix[i,:]*value)

    return derivatives

    # End of code ###
```

```
# STUDENT: PRINT THE OUTPUT AND COPY IT TO THE SOLUTION FILE
my_weights = np.ones(data_mat.shape[1]) # a weight of all 1s
derivative = weight_derivative(my_weights,train_data,train_labels)


print (derivative[:10])
```

```
[ 1.23415330e+03 -4.13993755e-08  1.00000000e+00  9.99993856e-01
  1.99987630e+00  9.99859072e-01  9.52574127e-01  3.59772270e+01
  2.99996572e+00 -1.38879439e-11]
```

3.

```
temp_weight = [0.5]*(len(train_data[0]))
initial_weights = np.array(temp_weight)
#print(initial_weights)


step_size = 5
tolerance = 2
# end of code
```

```
#Initialize the weights, step size and tolerance
# Start of code
#STUDENT: Specify the initial_weights, step_size, and tolera

temp_weight = [0.5]*(len(train_data[0]))
initial_weights = np.array(temp_weight)
#print(initial_weights)


step_size = 1
tolerance = 1
# end of code

# Use the regression_gradient_descent function to calculate
final_weights = gradient_descent(train_data,train_labels, in

# end of code
print ("Here are the final weights after convergence:")
print (final_weights)
```

```
Iteration:  573 gradient_magnitude:  3.6682392304496507
Iteration:  574 gradient_magnitude:  2.213136375610675
Iteration:  575 gradient_magnitude:  1.9803704159457487
Iteration:  576 gradient_magnitude:  1.8783445398539662
Iteration:  577 gradient_magnitude:  1.8131520656107953
Iteration:  578 gradient_magnitude:  1.7906836176702727
Iteration:  579 gradient_magnitude:  1.7802176657823705
Iteration:  580 gradient_magnitude:  1.7763035017143993
Iteration:  581 gradient_magnitude:  1.774618305284305
Iteration:  582 gradient_magnitude:  1.7739270604126407
Iteration:  583 gradient_magnitude:  1.7735934127788686
Iteration:  584 gradient_magnitude:  1.7733865413467367
Iteration:  585 gradient_magnitude:  1.7731850853899929
Iteration:  586 gradient_magnitude:  1.7728986819359303
Iteration:  587 gradient_magnitude:  1.7723980736553682
Iteration:  588 gradient_magnitude:  1.771442601239768
Iteration:  589 gradient_magnitude:  1.7695302631054801
Iteration:  590 gradient_magnitude:  1.7655867273725743
Iteration:  591 gradient_magnitude:  1.7573107327870083
Iteration:  592 gradient_magnitude:  1.740345037915812
Iteration:  593 gradient_magnitude:  1.710658997369197
Iteration:  594 gradient_magnitude:  1.6765140698262924

Iteration:  595 gradient_magnitude:  1.6515548064717083
Iteration:  596 gradient_magnitude:  1.6338579665542021
Iteration:  597 gradient_magnitude:  1.619519775674744
Iteration:  598 gradient_magnitude:  1.605520519064422
Iteration:  599 gradient_magnitude:  1.5814491956242431
Iteration:  600 gradient_magnitude:  1.4789406846914526
Iteration:  601 gradient_magnitude:  1.114876333894972
Iteration:  602 gradient_magnitude:  0.8802789949735388
Here are the final weights after convergence:
[ 9.36895823e-02  1.38697761e+01 -2.95000490e+01 ... -1.67588608e+02
  5.00000000e-01 -3.14998337e+01]
```

4.

STUDENT: CODE STARTS HERE
Pull out the parameters (theta_0, theta) of the logistic regression model

theta = final_weights #gradient_descent(train_data,train_labels, initial_weights, step_size, tolerance)
theta0 = theta[0]
theta = np.delete(theta,0)

STUDENT: CODE ENDS HERE

5.

```
## STUDENT: YOUR CODE HERE
 predict_array = []

 for i in range(len(feature_matrix)):

     dot = np.dot(feature_matrix[i],weights)

     z = -1*dot
     predict = 1/(1+pow(np.e,z))
     predict_array.append(predict)
 return np.array(predict_array)
 ## STUDENT: CODE ENDS
```

```
def model_predict(feature_matrix,weights):
# Prediction made by logistic regression

    # Input:
    # feature_matrix: numpy array of size n by d+1
    #                 note we have included the du
    # weights: weight vector to start with, a nump
    # Output:
    # labels: predicted labels, a numpy vector of

    ## STUDENT: YOUR CODE HERE
    predict_array = []

    for i in range(len(feature_matrix)):

        dot = np.dot(feature_matrix[i],weights)

        z = -1*dot
        predict = 1/(1+pow(np.e,z))
        predict_array.append(predict)
    return np.array(predict_array)
    ## STUDENT: CODE ENDS
```

#not test error did not work as written so I wrote my own

```python
# STUDENT: copy the output of this section to the solution file
def getError(preds_train,train_labels):
    errs_train =  0
    for i in range(len(preds_train)):
        if((preds_train[i] > 0.0) and (train_labels[i] < 0.0)):
            errs_train+=1
        if((preds_train[i] < 0.0) and (train_labels[i] > 0.0)):
            errs_train+=1
    return errs_train

## Get predictions on training and test data
preds_train = model_predict(train_data,final_weights)
#print(preds_train)
preds_test = model_predict(test_data,final_weights)


## Compute errors

errs_train = np.sum((preds_train > 0.0) is not (train_labels > 0.
errs_test = np.sum((preds_test > 0.0) is not (test_labels > 0.0)]

error_train = getError(preds_train,train_labels)
error_test = getError(preds_test,test_labels)
```

```python
print ("Training error: ", float(errs_train)/len(train_labels))
print ("Test error: ", float(errs_test)/len(test_labels))

print ("Training error: ", float(error_train)/len(train_labels))
print ("Test error: ", float(error_test)/len(test_labels))
```

```
Training error:  0.0004
Test error:  0.002
Training error:  0.1028
Test error:  0.184
```

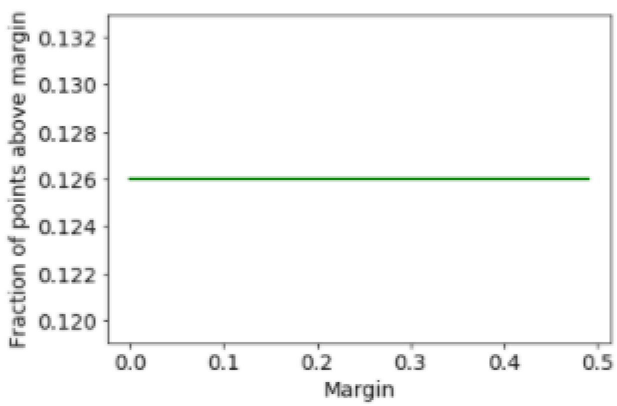
6.

```python
## STUDENT: YOUR CODE HERE
  model_predict_arr = model_predict(feature_matrix, weights)
  count =0;
  for i in range(len(model_predict_arr)):
    if (model_predict_arr[i]>0) and (model_predict_arr[i]<1):
      if(model_predict_arr[i]<0.5-gamma) or (model_predict_arr[i]>0.5+gamma):
        count+=1

  return count
```

```
def margin_counts(feature_matrix, weights, gamma):
## Return number of points for which Pr(y=1) lies in [0, 0.5 - gamma) or (0.5 + gamma, 1

    # Input:
    # feature_matrix: numpy array of size n by d+1, where n is the number of data points
    #                 note we have included the dummy feature as the first column of the
    # weights: weight vector to start with, a numpy vector of dimension d+1
    # gamma: the margin value
    # Output:
    # number of points for which Pr(y=1) lies in [0, 0.5 - gamma) or (0.5 + gamma, 1]

    ## STUDENT: YOUR CODE HERE
    model_predict_arr = model_predict(feature_matrix, weights)
    count =0;
    for i in range(len(model_predict_arr)):
        if (model_predict_arr[i]>0) and (model_predict_arr[i]<1):
            if(model_predict_arr[i]<0.5-gamma) or (model_predict_arr[i]>0.5+gamma):
                count+=1

    return count

    ## STUDENT: CODE ENDS
```

```
gammas = np.arange(0,0.5,0.01)
f = np.vectorize(lambda g: margin_counts(test_data, final_weights,g))
plt.plot(gammas, f(gammas)/500.0, linewidth=2, color='green')
plt.xlabel('Margin', fontsize=14)
plt.ylabel('Fraction of points above margin', fontsize=14)
plt.show()
```

```
C:\Users\kuent\Anaconda3\lib\site-packages\ipykernel_launcher.py:19: Runtim
```



7.

```
## STUDENT: YOUR CODE HERE
  model_predict_arr = model_predict(feature_matrix, weights)
  denominator=len(feature_matrix)
  numerator = 1

  for i in range(len(model_predict_arr)):
    if (model_predict_arr[i]>0) and (model_predict_arr[i]<1):
      if(model_predict_arr[i]<(0.5-gamma)) or (model_predict_arr[i]>(0.5+gamma)):
        denominator+=1
        if(labels[i]<(0.5-gamma)) and (model_predict_arr[i]<(0.5-gamma)):
          numerator+=1
        if(labels[i]>(0.5+gamma)) and (model_predict_arr[i]>(0.5+gamma)):
```

numerator+=1

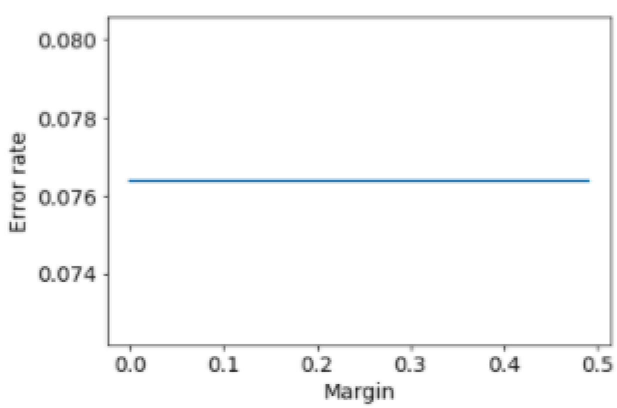
return numerator/denominator

STUDENT: YOUR CODE ENDS

```
def margin_errors(feature_matrix, labels, weights, gamma):
## Return error of predictions that lie in intervals [0, 0.5 - gamma) and (0.5 + gamma, 1]

    # Input:
    # feature_matrix: numpy array of size n by d+1, where n is the number of data points, d
    #                 note we have included the dummy feature as the first column of the fe
    # labels: true labels y, a numpy vector of dimension n
    # weights: weight vector to start with, a numpy vector of dimension d+1
    # gamma: the margin value
    # Output:
    # error of predictions that lie in intervals [0, 0.5 - gamma) and (0.5 + gamma, 1]

    ## STUDENT: YOUR CODE HERE
    model_predict_arr = model_predict(feature_matrix, weights)
    denominator=len(feature_matrix)
    numerator = 1

    for i in range(len(model_predict_arr)):
        if (model_predict_arr[i]>0) and (model_predict_arr[i]<1):
            if(model_predict_arr[i]<(0.5-gamma)) or (model_predict_arr[i]>(0.5+gamma)):
                denominator+=1
                if(labels[i]<(0.5-gamma)) and (model_predict_arr[i]<(0.5-gamma)):
                    numerator+=1
                if(labels[i]>(0.5+gamma)) and (model_predict_arr[i]>(0.5+gamma)):
                    numerator+=1



    return numerator/denominator

    ## STUDENT: YOUR CODE ENDS
```

```
## Plot the result
plt.plot(gammas, f(gammas), linewidth=2)
plt.ylabel('Error rate', fontsize=14)
plt.xlabel('Margin', fontsize=14)
plt.show()
```

```
C:\Users\kuent\Anaconda3\lib\site-packages\ipykernel_launcher
```



8.

STUDENT: YOUR CODE HERE

```
#the first index is the larges value and the last index is the smallest value
#the value that is going to be excluded firrst from the array is in the back

weights= np.delete(final_weights,0)
print(len(weights))
large_positive = []
small_negative = []

for i in range(len(weights)):
   #initialize array
   if (len(large_positive)<10):
      large_positive.append([vocab[i],weights[i]])
      small_negative.append([vocab[i],weights[i]])
   else:
      #larger than smallest value in array
      if (final_weights[i]>large_positive[0][1]):
         #remove small add large
         large_positive.pop(0)
         large_positive.append([vocab[i],weights[i]])
         large_positive.sort(key=lambda x: x[1])
      #smaller than largest value in array
      if (final_weights[i]<small_negative[9][1]):
         #remove large add small
         small_negative.pop(9)
         small_negative.append([vocab[i],weights[i]])
         small_negative.sort(key=lambda x: x[1])

#remove weights from print statement
final_positive = []
final_negative = []
for i in range(10):
   final_positive.append(large_positive[i][0])
   final_negative.append(small_negative[i][0])


print('Top Ten')
print(large_positive)
print('Bottom Ten')
print(small_negative)

## STUDENT: CODE ENDS
```

```python
## STUDENT: YOUR CODE HERE
#the first index is the larges value and the last index is the smallest value
#the value that is going to be excluded firrst from the array is in the back

weights= np.delete(final_weights,0)
print(len(weights))
large_positive = []
small_negative = []

for i in range(len(weights)):
    #initialize array
    if (len(large_positive)<10):
        large_positive.append([vocab[i],weights[i]])
        small_negative.append([vocab[i],weights[i]])
    else:
        #larger than smallest value in array
        if (final_weights[i]>large_positive[0][1]):
            #remove small add large
            large_positive.pop(0)
            large_positive.append([vocab[i],weights[i]])
            large_positive.sort(key=lambda x: x[1])
        #smaller than largest value in array
        if (final_weights[i]<small_negative[9][1]):
            #remove large add small
            small_negative.pop(9)
            small_negative.append([vocab[i],weights[i]])
            small_negative.sort(key=lambda x: x[1])

#remove weights from print statement
final_positive = []
final_negative = []
for i in range(10):
    final_positive.append(large_positive[i][0])
    final_negative.append(small_negative[i][0])


print('Top Ten')
print(large_positive)
print('Bottom Ten')
print(small_negative)

## STUDENT: CODE ENDS
```

Each is accompanied with their theta value

```
Top Ten
[['you', 522.3124682700212], ['well', 1198.4122353355472], ['family', 1267.4999997273922], ['happy', 1507.5298332164937], ['fri
endly', 1609.6197732734986], ['amazing', 1756.0169505164492], ['works', 1953.3096254243846], ['fantastic', 2054.3429738609047],
['loved', 2349.88442359567], ['great', 2382.2088423393316]]
Bottom Ten
[['worst', -2435.1562271465036], ['disappointment', -2236.3075883660986], ['waste', -2118.928162274578], ['disappointing', -182
0.0104102997275], ['avoid', -1655.739111789458], ['bland', -1609.8454208770834], ['fails', -1431.4384892762353], ['aren', -132
2.6989221304673], ['difficult', -1264.999999999803], ['writing', -824.9671458584996]]
```