532

Assignment 7 – DEVIN BRESSER   1a, 1b, 1c — See code

```
11]: # Problem 1a - i
     # Simply add 0.001 to every entry in A
     A = A+0.001

     # Problem 1a - ii
     # For each entry in each column of A, divide it by the sum down that column
     # Compute the sum of each column
     column_sums = [np.sum(A[:,col]) for col in range(np.shape(A)[1])]

     # Divide each entry in each column by the sum of its column
     A = A / column_sums

     # Verify that each column sums to 1:
     print(f"Column 1 sum: {np.round(np.sum(A[:,0]))}")
     print(f"Column 2 sum: {np.round(np.sum(A[:,1]))}")
     print(f"Column 2 sum: {np.round(np.sum(A[:,2]))}")

     # Problem 1a - iii
     # Compute 1st eigenvalue (should be 1) and associated eigenvector
     s, E = eigs(csc_matrix(A), k = 1)
     # (Add np.real to remove annoying +0j components)
     print(f"\nFirst eigenvalue: {np.real(s)}\nAssociated eigenvector:\n {np.real(E)}")
```

```
Column 1 sum: 1.0
Column 2 sum: 1.0
Column 2 sum: 1.0

First eigenvalue: [1.]
Associated eigenvector:
 [[0.00849655]
 [0.00852945]
 [0.00849655]
 ...
 [0.02157236]
 [0.00866803]
 [0.00849655]]
```

```
49]: # Problem 1b
     # Know that the first eigenvector, P, gives importance of pages.

     # Turn P into a dictionary so we can keep track of indices
     P = dict(enumerate(np.real(E)))

     # Sort dictionary P by value while preserving index
     sorted_P = sorted(P.items(), key=lambda x: x[1], reverse=True)

     # Display first n items of sorted P
     n=10
     print("First n most important items:\n ")
     for index, item in enumerate(sorted_P[:n]):
         print(f"#{index + 1}: {item}")
         print()  # This will add a new line between each item

     print(f"1st most important article title: {nodes_dict.get(5089)}")

     # Conclusion: Item # 5089, title simply "Wisconsin" is the most important.

     # Problem 1c
     # From the above, we notice that the 3rd most important page is item #1345 in nodes_dict

     print(f"3rd most important article title: {nodes_dict.get(1345)}")

     # Hooray for Madison!

     # For fun... also noticed that index#2230 is UW-Madison:
     print(f"4th most important article title: {nodes_dict.get(2230)}")
```

```
First n most important items:

#1: (5089, array([0.58556416]))

#2: (2312, array([0.44693652]))

#3: (1345, array([0.07074235]))

#4: (2230, array([0.04778512]))

#5: (379, array([0.03021724]))

#6: (2545, array([0.02981724]))

#7: (517, array([0.02588714]))

#8: (1380, array([0.02586532]))

#9: (4354, array([0.02467508]))

#10: (1603, array([0.02397484]))

1st most important article title: "Wisconsin"
3rd most important article title: "Madison, Wisconsin"
4th most important article title: "University of Wisconsinâ\u80\u93Madison"
```

General: $\min_{\underline{w}} \sum_i \underbrace{\ell_i(\underline{w})}_{\text{loss on ith training sample.}} + \lambda \cdot \underbrace{r(\underline{w})}_{\text{regularizer}}$

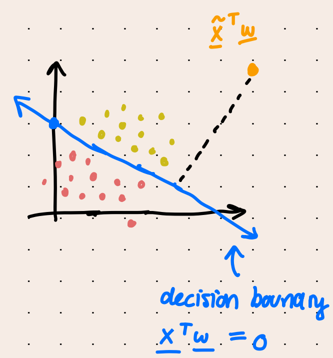
p2.)

a.) Ridge Regression $\ell_i(\underline{w}) = (\underline{x}_i \underline{w}^T - y_i)^2$

$$\rightarrow \min_{\underline{w}} \sum_{i=1}^{n} (\underline{x}_i^T \underline{w} - y_i)^2 + \lambda \|\underline{w}\|_2^2$$

Logistic Regression $\ell_i(\underline{w}) = \log(1 - \exp(-y_i \underline{x}^T \underline{w}))$

$$\rightarrow \min_{\underline{w}} \sum_{i=1}^{n} \log(1 + \exp(-y_i \underline{x}^T \underline{w})) + \lambda \|\underline{w}\|_2^2.$$



→ Let's say we have some easy to classify data point $\underline{\tilde{x}}^T \underline{\tilde{w}}$

→ $|\underline{\tilde{x}}^T \underline{\tilde{w}}|$ is large.

→ $\ell_i(\underline{w}) = \log(1 + \exp(-y_i \underline{\tilde{x}}^T \underline{w}))$

→ $\ell_i(\underline{w}) \approx \log(1 + \exp(\text{large negative number}))$

→ $\ell_i(\underline{w}) \approx \log(1 + \text{very small \#}) \approx \log(1) = 0.$

So, $\ell_i(\underline{w})$ is small as $\underline{x}_i^T \underline{w}$ is large.

Compare with squared error loss function:

→ $\ell_i(\underline{w}) = (\underline{\tilde{x}}^T \underline{w} - y_i)^2$

→ $\ell_i(\underline{w}) \approx (\text{large difference})^2 \rightarrow \underline{\text{large}}\ \ell_i(\underline{w}).$

b) Compute an expression for the gradient (with respect to $\boldsymbol{w}$) of the ℓ_2 regularized logistic loss:

$$\min_{\boldsymbol{w}} \sum_{i=1}^{n} \log\left(1 + e^{-y_i \boldsymbol{x}_i^T \boldsymbol{w}}\right) + \lambda ||\boldsymbol{w}||_2^2$$

$\rightarrow \nabla_{\underline{w}} \log(1 + \exp(-y \underline{x}^T \underline{w}) + \underbrace{\lambda ||\underline{w}||_2^2}$ $\rightarrow$ $\nabla_{\underline{w}} \lambda ||\underline{w}||_2^2 = 2\lambda \underline{w}$

$\rightarrow 2\lambda \underline{w} + \nabla_{\underline{w}} \log(1 + \exp(-y \underline{x}^T \underline{w}))$ $\rightarrow$ Apply derivative of natural log & chain rule:

$\rightarrow \nabla_{w} \log(a(\underline{w})) = \frac{1}{a(\underline{w})} \nabla_{\underline{w}} a(\underline{w})$

$\rightarrow 2\lambda \underline{w} + \frac{1}{1 + \exp(-y \underline{x}^T \underline{w})} \cdot \nabla_{\underline{w}} \cancel{1} + \exp(-y \underline{x}^T \underline{w})$

$\rightarrow 2\lambda \underline{w} + \frac{1}{1 + \exp(-y \underline{x}^T \underline{w})} \cdot \exp(-y \underline{x}^T \underline{w}) \cdot -y \underline{x}$ $\rightarrow$ Apply identity $\frac{d}{dx} e^x = e^x$ & chain rule.

$\rightarrow \nabla \underline{x}^T \underline{w} = \underline{x}$.

$\rightarrow 2\lambda \underline{w} + \frac{-y \underline{x} \exp(-y \underline{x}^T \underline{w})}{1 + \exp(-y \underline{x}^T \underline{w})}$

$$\rightarrow \nabla_{\underline{w}} \ell(\underline{w}) = \sum_{i=1}^{n} \frac{-y_i \underline{x}_i \exp(-y_i \underline{x}_i^T \underline{w})}{1 + \exp(-y_i \underline{x}_i^T \underline{w})} + 2\lambda \underline{w}$$

2c.) See code ↓

```python
# Problem 2c

# define the function as derived in problem 2b:

# this is the gradient for a single data point
def gradient(w, x_i, y_i):

    w = np.array(w)
    z = -y_i * np.dot(x_i, w) # for simplicity
    grad = (-y_i * x_i * np.exp(z) / (1 + np.exp(z)))

    return grad

# sum up the gradients over all data points (+ reg term)
def sum_grad(w):

    w = np.array(w)
    total = np.zeros(2)
    n = len(x_train)
    lmda = 1
    reg = 2*lmda*w

    for i in range(n):
        part = gradient(w, x_train[i], y_train[i])
        total += part

    return total+reg

def logistic(w, x_i, y_i):

    w = np.array(w)
    z = -y_i * np.dot(x_i, w) # for simplicity
    return np.log(1 + np.exp(z))
```

```python
def objective_function(w, method):

    lmda = 1
    reg = lmda * np.linalg.norm(w)**2

    # This part is for problem 2c
    if(method=="logistic"):
        w = np.array(w)
        total = 0
        n = len(x_train)

        for i in range(n):
            part = logistic(w, x_train[i], y_train[i])
            total += part

        return total+reg

    #This part is for problem 2e
    if(method=="squares"):
        w = np.array(w)
        term1 = (w.T @ x_train.T @ x_train @ w)
        term2 = (-2 * w.T @ x_train.T @ y_train)
        term3 = (y_train.T @ y_train)

        return term1 + term2 + term3 + reg

    return 0
```

```python
# Problem 2c

# Now apply Gradient Descent algorithm
# w(k+1) = w(k) - tau * grad(f(w))

# In this case, tau must be between 0 and 2/||X||op^2
U,s,VT = np.linalg.svd(x_train)
print(s)
print(f"(2/largest sigma's) squared: {2/(s**2)}")
# So, 0 < tau < 0.00175

# Pick tau arbitrarily as half of the upper bound
print(f"Select tau: {(2/(s[0]**2))*0.5}")
```

```
[33.81643539 13.0646017 ]
(2/largest sigma's) squared: [0.00174894 0.01171757]
Select tau: 0.0008744688564408182
```

In [818]:

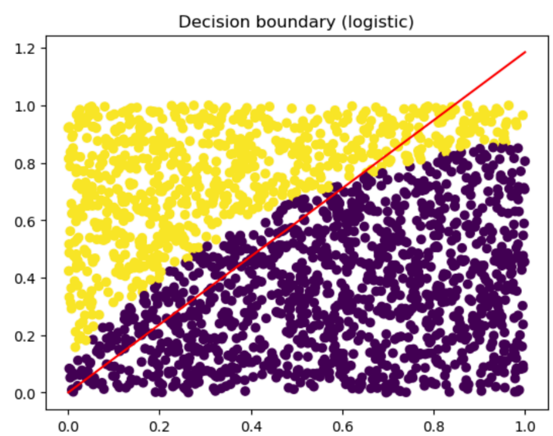
```python
# Problem 2c

def gradient_descent(n, method):

    w = np.zeros(2)
    tau = 0.000874  # per commentary above

    # This part is for problem 2c
    if(method=="logistic"):

        for i in range(n):
            sum_grad_w = sum_grad(w)
            w_new = w - (tau * sum_grad_w)  # update w

            obj_val_prev = objective_function(w, "logistic")
            obj_val_new = objective_function(w_new, "logistic")

            if(obj_val_new > obj_val_prev):
                print(f"Objective function value increased, stopping at iteration {i+1}")
                break

            w = w_new

        return w

    # This part is for problem 2e
    if(method=="squares"):

        for i in range(n):
            w = w.reshape(-1,1)
            w_new = w - (tau * (x_train.T @ (x_train @ w - y_train)))

            obj_val_prev = objective_function(w, "squares")
            obj_val_new = objective_function(w_new, "squares")

            print(f"w_current: {w}, w_new: {w_new})")
            print(f"obj: {obj_val_prev}, obj_new: {obj_val_new}")
            if(obj_val_new > obj_val_prev):
                print(f"Objective function value increased, stopping at iteration {i+1}")
                break

            w = w_new

        return w

    return 0
```

2d.)

n [12]:

```python
# Problem 2d

w_min_logistic = gradient_descent(10000, "logistic")

x1_values = np.linspace(np.min(x_train[:,0]), np.max(x_train[:,0]), 100)
x2_values = -w_min_logistic[0]/w_min_logistic[1] * x1_values
plt.plot(x1_values, x2_values, color='red')


n_train = np.size(y_train)

plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.title('Decision boundary (logistic)')
plt.show()

print(f"Computed w_min using logistic gradient descent: {w_min_logistic}")
```

Objective function value increased, stopping at iteration 2219



Computed w_min using logistic gradient descent: [-7.6296405 6.4364439]

12]:

```python
# Problem 2d

def error_rate(x_train, y_train, w):

    # compute predicted labels
    y_pred = np.sign(np.dot(x_train, w))

    # compare predicted labels to true labels
    errors = np.sum(y_pred != y_train[:, 0])

    # compute error rate
    error_rate = errors / len(y_train)

    return error_rate

# Compute error rate
err_rate_logistic = error_rate(x_train, y_train, w_min_logistic)
print(f"For w values of: {w_min_logistic}")
print(f"The error rate is: {err_rate_logistic}")
```

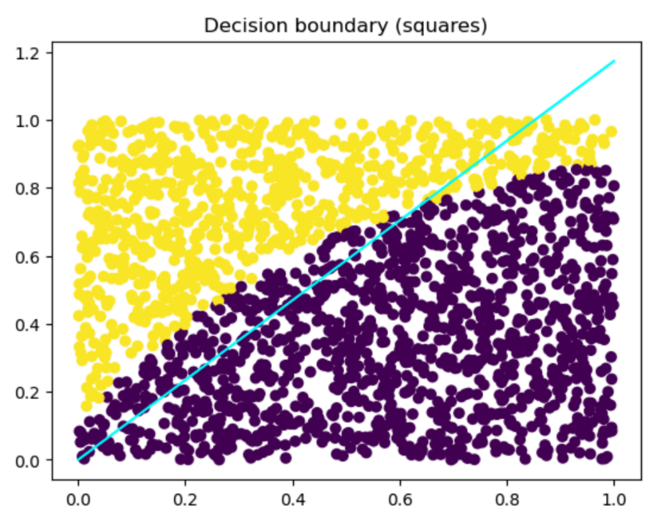
```
For w values of: [-7.6296405  6.4364439]
The error rate is: 0.1145
```

2e.)

16]:

```python
# Problem 2e

w_min_squares = gradient_descent(10000, "squares")

x1_values = np.linspace(np.min(x_train[:,0]), np.max(x_train[:,0]), 100)
x2_values = -w_min_squares[0]/w_min_squares[1] * x1_values
plt.plot(x1_values, x2_values, color='cyan')

n_train = np.size(y_train)

plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.title('Decision boundary (squares)')
plt.show()

print(f"Computed w_min using logistic gradient descent: {w_min_squares}")
print(f"Computed w_min_LS using pseudoinverse method: {np.linalg.inv(x_train.T @ x_train) @ 

# Comment: The decision boundary looks about the same in the base case.
```

```
Objective function value increased, stopping at iteration 33
```

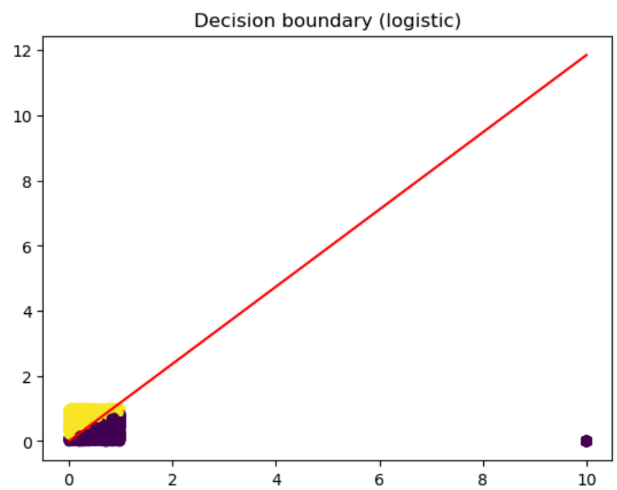



```
Computed w_min using logistic gradient descent: [[-1.9847562 ]
 [ 1.69211052]]
Computed w_min_LS using pseudoinverse method: [[-1.99533655]
 [ 1.70256733]]
```

2f.)

```
[12]: # Problem [redacted] 2f

w_min_logistic = gradient_descent(10000, "logistic")

x1_values = np.linspace(np.min(x_train[:,0]), np.max(x_train[:,0]), 100)
x2_values = -w_min_logistic[0]/w_min_logistic[1] * x1_values
plt.plot(x1_values, x2_values, color='red')


n_train = np.size(y_train)

plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.title('Decision boundary (logistic)')
plt.show()
```

```
Objective function value increased, stopping at iteration 2244
 and w-value [-7.62964176  6.43644501]
```



2f

```
[13]: # Problem [redacted]

def error_rate(x_train, y_train, w):

    # compute predicted labels
    y_pred = np.sign(np.dot(x_train, w))

    # compare predicted labels to true labels
    errors = np.sum(y_pred != y_train[:, 0])

    # compute error rate
    error_rate = errors / len(y_train)

    return error_rate

# Compute error rate
err_rate_logistic = error_rate(x_train, y_train, w_min_logistic)
print(f"For w values of: {w_min_logistic}")
print(f"The error rate is: {err_rate_logistic}")
```

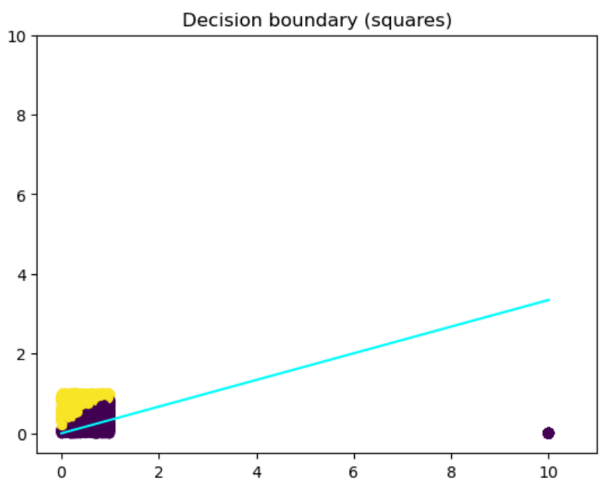
```
For w values of: [-7.62964174  6.43644499]
The error rate is: 0.07633333333333334
```

↑ Logistic

2f.)

In [15]:
```python
# Problem 2f - LS

w_min_squares = gradient_descent(10000, "squares")

x1_values = np.linspace(np.min(x_train[:,0]), np.max(x_train[:,0]), 100)
x2_values = -w_min_squares[0]/w_min_squares[1] * x1_values
plt.plot(x1_values, x2_values, color='cyan')

n_train = np.size(y_train)

plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.title('Decision boundary (squares)')
plt.xlim(-0.5,11)
plt.ylim(-0.5,10)

plt.show()

# Comment: The decision boundary looks about the same in the base case.
```

```
Objective function value increased, stopping at iteration 987
 and w-value [[-0.10556255]
 [ 0.31533253]]
```



In [10]:
```python
# Problem [redacted] 2f

# Compute error rate
w_min_squares = np.squeeze(w_min_squares.reshape(1,-1))
err_rate_squares = error_rate(x_train, y_train, w_min_squares)
print(f"For w values of: {w_min_squares}")
print(f"The error rate is: {err_rate_squares}")

# So the two classifiers work equally as well in the base case.
```

```
For w values of: [-0.10556255  0.31533253]
The error rate is: 0.3
```

Comment:

The error rate increases substantially under the squared error loss function in this case. This is because the decision boundary skews heavily towards the outliers.