COMP9417 – Machine Learning

Homework 1: Classification models – Regularized Logistic Regression and the Perceptron

Wanqing Yang - z5325987

Question 1. Regularized Logistic Regression

(a) When $y_i = 0$ and $\tilde{y}_i = -1$, objective(1) and objective(2) are equal to each other, which is $(\hat{w}, \hat{c}) = arg \min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} log\big(1 + exp(w^T x_i + c)\big)$

When $y_i = 1$ and $\tilde{y}_i = 1$, objective(1) and objective(2) are also equal to each other, which is $(\hat{w}, \hat{c}) = arg \min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} log\big(1 + exp(-w^T x_i + c)\big)$

Therefore, the two objectives (1) and (2) are identical.
For λ, the larger the value, the greater the weight of the second term has.
For C, the larger the value, the less weight of the second term has.

Therefore, C plays the same role as $\frac{1}{\lambda}$

(b) (I) $LogSumExp(x_1, \dots, x_n) = log(e^{x_1} + \cdots + e^{x_n})$
$= log(e^{x_1 - x_*} e^{x_*} + \cdots + e^{x_n - x_*} e^{x_*}) = log(e^{x_*}) + log(e^{x_1 - x_*} + e^{x_n - x_*})$
$= x_* + log(e^{x_1 - x_*} + \cdots + e^{x_n - x_*})$

(II) Because $x_* = max(x_1, \dots, x_n)$
Then $x_i \leq x_*$
Therefore, $0 < e^{x_i - x_*} \leq e^0 = 1$

(III) For $log(e^{x_1} + \cdots + e^{x_n})$, if x_i is a very small number, e^{x_i} might be rounded to 0. At this point, a numerical overflow occurs when we try to compute $loge^{x_i} = log0$, and we are going to get INF.
But after the transformation, each term inside of the log is a number greater than 0 and less than equal to 1, so numerical overflow is avoided.

(c)

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X_ = pd.read_csv('songs.csv', usecols=[2, 3, 4, 6, 8, 9, 11, 12, 13, 14])
Y_ = pd.read_csv('songs.csv', usecols=[16])
X_ = X_.values
Y_ = Y_.values
Y_ = Y_.flatten()
n = len(X_)
X = []
Y = []
```

```
for i in range(0, n):
    if np.isnan(X_[i]).any():
        continue
    if Y_[i] == 5:
        X.append(X_[i])
        Y.append(1)
    elif Y_[i] == 9:
        X.append(X_[i])
        Y.append(-1)
X = np.array(X)
Y = np.array(Y)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=23)
x_tmp = X_train
scl = StandardScaler().fit(x_tmp)
X_train = scl.transform(X_train)
X_test = scl.transform(X_test)
print("X_train first row: ", X_train[0][0], X_train[0][1], X_train[0][2])
print("X_train last row: ", X_train[-1][0], X_train[-1][1], X_train[-1][2])
print("X_test first row: ", X_test[0][0], X_test[0][1], X_test[0][2])
print("X_test last row: ", X_test[-1][0], X_test[-1][1], X_test[-1][2])
print("Y_train first row: ", Y_train[0])
```

```
print("Y_train last row: ", Y_train[-1])
print("Y_test first row: ", Y_test[0])
print("Y_test last row: ", Y_test[-1])
```

The output of this code is that:

```
X_train first row:  -0.935558428070434 0.675192977391112 1.3849984956609747
X_train last row:  -1.1330147946544558 -1.0945887725876495 0.9670244850142993
X_test first row:  -0.29382523667236327 1.3600510492219735 0.2630682565567403
X_test last row:  -0.29382523667236327 -1.0539041346571034 -1.348331547646891
Y_train first row:  -1
Y_train last row:  1
Y_test first row:  -1
Y_test last row:  1
```

(d) X_train is used to train the model, and X_test is used to make predictions to ensure the

accuracy of the model.

If this model were in the real world, it would not include the contents of X_test except for data leakage.

Therefore, X_test cannot be used to fit the model.

(e)

```
def reg_log_loss(W, C, X, y):
    w = W[1:]
    c = W[0]
    term1 = 0.5 * (np.linalg.norm(w) ** 2)
    sum = 0.0
    n = len(X)
    for i in range(0, n):
        sum += np.logaddexp(0, -y[i]*(np.dot(w.T, X[i]) + c))
    term2 = sum * C
    return term1 + term2
```

```
c = 1.2
w = 0.35 * np.ones(X_train.shape[1])
W = np.insert(w, 0, c)
loss = reg_log_loss(W = W, C = 0.001, X = X_train, y = y_train)
print(loss)
```

The loss is `3.2987851381957265`

(f)

```
def reg_log_loss(W, C, X, y):
    w = W[1:]
    c = W[0]
    term1 = 0.5 * np.dot(w.T, w)
    sum = 0.0
    n = len(X)
    for i in range(0, n):
        sum += np.logaddexp(0, -y[i]*(np.dot(w.T, X[i]) + c))
    term2 = sum * C
    return term1 + term2

def reg_log_fit(X, y, C):
    w = 0.1 * np.ones(X.shape[1])
    W0 = np.insert(w, 0, -1.4)
    g = lambda x: reg_log_loss(x, C, X, y)
    res = minimize(g, W0, method='Nelder-Mead', tol=1e-6)
    return res.x

def sigmoid(x):
    return 1.0 / (1 + np.exp(-x))
```

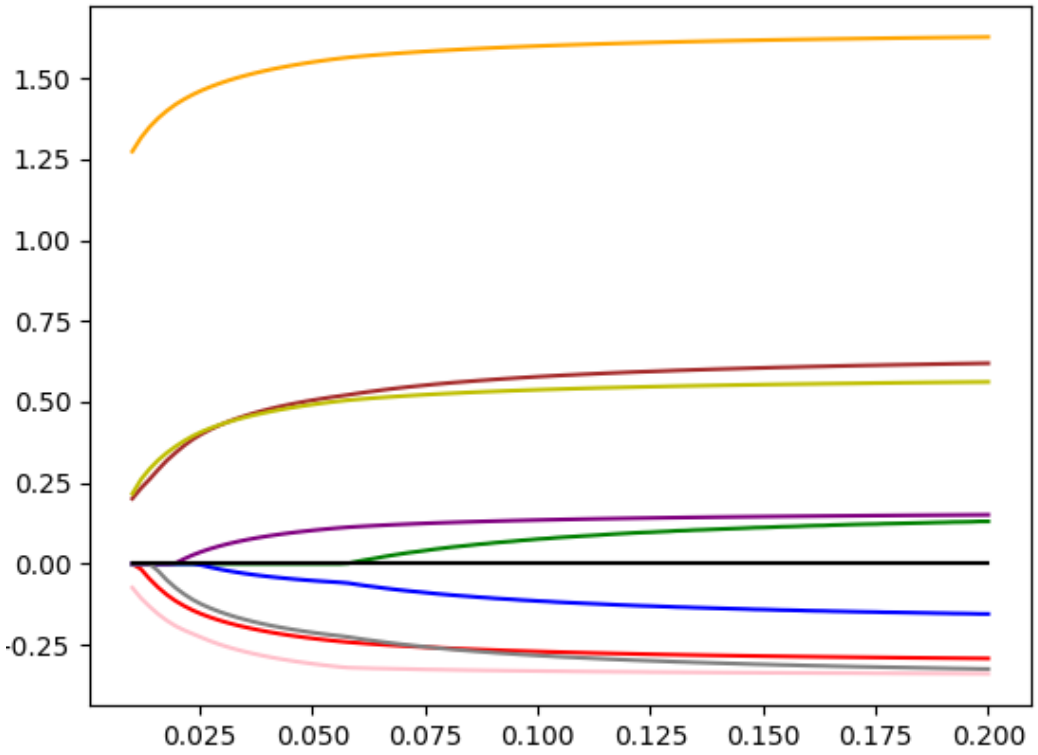
```
res_x = reg_log_fit(X_train, y_train, C = 0.4)
w_pre = res_x[1:]
c_pre = res_x[0]
y_pre = []
n = len(X_train)
for i in range(0, n):
    y_pre.append(sigmoid(np.dot(w_pre.T, X_train[i]) + c_pre))
y_pre = np.array(y_pre)
LogLoss = log_loss(y_train, y_pre)
print("train and test loss of resulting model: ", LogLoss)
clf = LogisticRegression(C=1, tol=1e-6, penalty='l2', solver='liblinear')
clf.fit(X_train, y_train)
y_pre_clf = clf.predict(X_train)
LogLoss_clf = log_loss(y_train, y_pre_clf)
print("train and test loss of logistic regression model: ", LogLoss_clf)
```

The train and test loss of resulting model is `0.4115689537829064`

The train and test loss of logistic regression model is `5.7141804840144115`

(g)

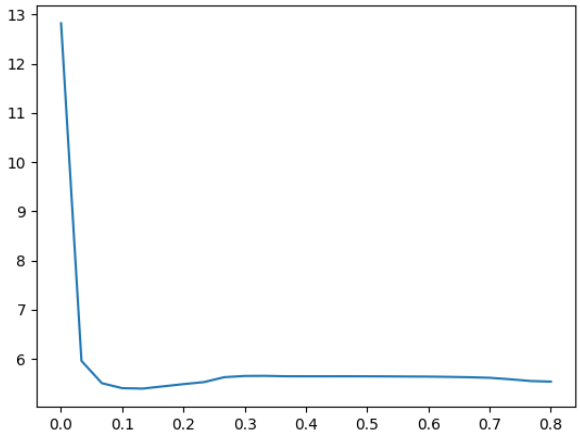
```
Cs = np.linspace(0.01, 0.2, num=100)
num = len(Cs)
coef = []
for i in range(0, num):
    clf_g = LogisticRegression(penalty='l1', solver='liblinear', C=Cs[i])
    clf_g.fit(X_train, y_train)
    length = len(clf_g.coef_[0])
    tmp = []
    for j in range(0, length):
        tmp.append(clf_g.coef_[0][j])
    tmp = np.array(tmp)
    coef.append(tmp)
coef = np.array(coef)
print(coef)
dic = {0:"red", 1:"brown", 2:"green", 3:"blue", 4:"orange", 5:"pink", 6:"purple", 7:"grey", 8:"black", 9:"y"}
for i in range(0, 10):
    tmp = []
    for j in range(0, num):
        tmp.append(coef[j][i])
    tmp = np.array(tmp)
    plt.plot(Cs, tmp, color=dic[i])
plt.show()
```



Because L1 regularization is sparse, it can set most parameters to 0, thus obtaining more important features. Based on this plot, the most important feature is the orange (5^{th}) feature.

(h)

```
X_train_20 = X_train[:544]
y_train_20 = y_train[:544]
Cs = np.linspace(0.0001, 0.8, num = 25)
LogLoss_C = []
for k in range(0, len(Cs)):
    sum = 0.0
    for i in range(0, 544):
        X_train_i = np.delete(X_train_20, i, 0)
        y_train_i = np.delete(y_train_20, i, 0)
        clf = LogisticRegression(penalty='l1', solver='liblinear', C=Cs[k])
        clf.fit(X_train_i, y_train_i)
        y_pre_i = clf.predict(X_train_i)
        LogLoss_i = log_loss(y_train_i, y_pre_i)
        sum += LogLoss_i
    avg = sum / 544
    LogLoss_C.append(avg)
LogLoss_C = np.array(LogLoss_C)
plt.plot(Cs, LogLoss_C)
plt.show()
```



By observing the image, it can be seen that log_loss gets the minimum value when c is around 0.13.

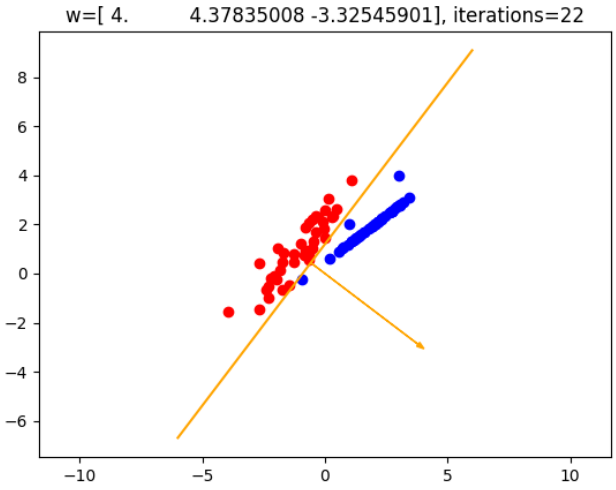
Therefore, the optimal value of C is 0.13

Question 2. Perceptron Learning Variants

(a)

```
def perceptron(X, y, max_iter = 100):
    np.random.seed(1)
    w = np.zeros(3)
    t = 1
    while t <= max_iter:
        misc_set = []
        for i in range(0, len(X)):
            if y[i] * (np.dot(w, X[i])) <= 0:
                misc_set.append(i)
        if len(misc_set) == 0:
            break
        idx = np.random.randint(0, len(misc_set))
        w = w + y[misc_set[idx]] * X[misc_set[idx]]
        t += 1
    return w, t-1
```

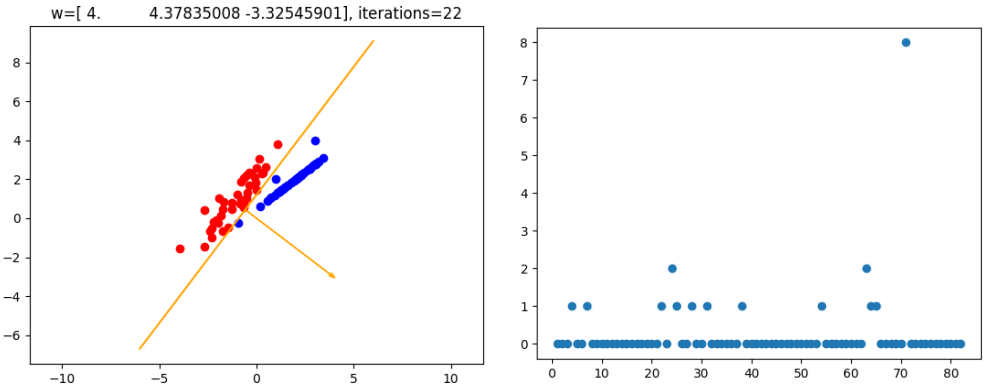
```
X = pd.read_csv("PerceptronX.csv", header=None)
y = pd.read_csv("Perceptrony.csv", header=None)
X = np.array(X)
y = np.array(y)
w, nmb_iter = perceptron(X, y)
fig, ax = plt.subplots()
plot_perceptron(ax, X, y, w)
ax.set_title(f"w={w}, iterations={nmb_iter}")
plt.savefig("name.png", dpi=300)
plt.show()
```



(b)

```
a, nmb_iter = dual_perceptron(X, y)
w = np.zeros(3)
y = y.flatten()
for i in range(0, len(X)):
    w = w + a[i] * X[i] * y[i]
fig, ax = plt.subplots()
plot_perceptron(ax, X, y, w)
ax.set_title(f"w={w}, iterations={nmb_iter}")
plt.savefig("name.png", dpi=300)
plt.show()
n = len(X)
xx = np.arange(1, n+1)
plt.scatter(xx, a)
plt.show()
```

```
def dual_perceptron(X, y, max_iter = 100):
    np.random.seed(1)
    n = len(X)
    w = np.zeros(n)
    t = 1
    while t <= max_iter:
        misc_set = []
        for i in range(0, len(X)):
            sum = 0
            for j in range(0, len(X)):
                sum += y[j]*w[j] * np.dot(X[j], X[i])
            if y[i] * sum <= 0:
                misc_set.append(i)
        if len(misc_set) == 0:
            break
        idx = np.random.randint(0, len(misc_set))
        w[misc_set[idx]] = w[misc_set[idx]] + 1
        t += 1
    return w, t-1
```

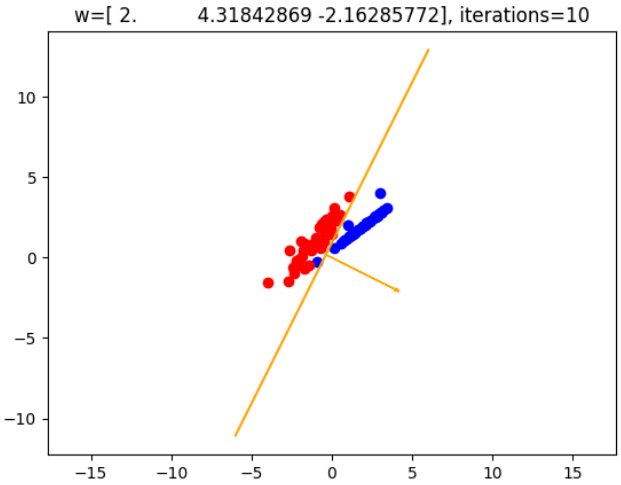


My plot (a) and (b) are same. This is because in the dual form, $\sum_{j=1}^{N} \alpha_j y_j$ corresponds to the change in w.

(c)

```
w, nmb_iter = r_perceptron(X, y)
print(w)
fig, ax = plt.subplots()
plot_perceptron(ax, X, y, w)
ax.set_title(f"w={w}, iterations={nmb_iter}")
plt.savefig("name.png", dpi=300)
plt.show()
```

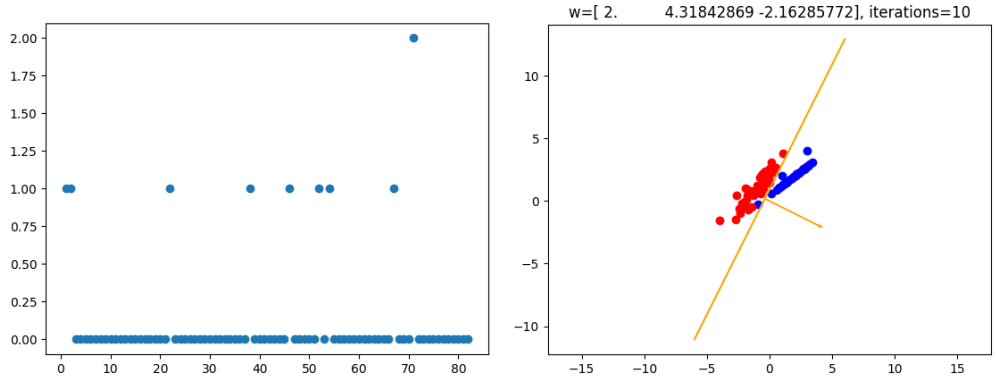
```
def r_perceptron(X, y, max_iter = 100):
    np.random.seed(1)
    w = np.zeros(3)
    t = 1
    I = np.zeros(n)
    while t <= max_iter:
        misc_set = []
        for i in range(0, len(X)):
            if y[i] * (np.dot(w, X[i])) + 2 * I[i] <= 0:
                misc_set.append(i)
        if len(misc_set) == 0:
            break
        idx = np.random.randint(0, len(misc_set))
        w = w + y[misc_set[idx]] * X[misc_set[idx]]
        t += 1
        I[misc_set[idx]] = 1
    return w, t - 1
```



(d)

```
a, nmb_iter = r_dual_perceptron(X, y)
w = np.zeros(3)
y = y.flatten()
for i in range(0, len(X)):
    w = w + a[i] * X[i] * y[i]
fig, ax = plt.subplots()
plot_perceptron(ax, X, y, w)
ax.set_title(f"w={w}, iterations={nmb_iter}")
plt.savefig("name.png", dpi=300)
plt.show()
n = len(X)
xx = np.arange(1, n+1)
plt.scatter(xx, a)
plt.show()
```

```
def r_dual_perceptron(X, y, max_iter = 100):
    np.random.seed(1)
    n = len(X)
    w = np.zeros(n)
    t = 1
    I = np.zeros(n)
    while t <= max_iter:
        misc_set = []
        for i in range(0, len(X)):
            sum = 0
            for j in range(0, len(X)):
                sum += y[j] * w[j] * np.dot(X[j], X[i])
            if y[i] * sum + 2 * I[i] <= 0:
                misc_set.append(i)
        if len(misc_set) == 0:
            break
        idx = np.random.randint(0, len(misc_set))
        w[misc_set[idx]] = w[misc_set[idx]] + 1
        t += 1
        I[misc_set[idx]] = 1
```



$input$: $(x_1, y_1), \ldots, (x_n, y_n), r > 0$

$initialize$: $\alpha^{(0)} = (0,0,\ldots,0) \in R^n, I^{(0)} = (0,0,\ldots,0) \in R^n,$

$for\ \ t = 1, \ldots, max_i ter$:

$if\ there\ is\ an\ index\ i\ such\ that\ y_{i\,\sum_{i=1}^{n} y_j \alpha_j <x_j, x_i> + I_i r \leq 0}$:

$\alpha_i^{(t+1)} = \alpha_i^{(t)} + 1;\ t = t + 1;\ I\_i = 1$

$else$:

$output \alpha^{(t)}, t$

(e) The additive term is used to mark whether (x_i, y_i) has been used previously. Its advantages is that it shortens the time complexity of perceptron algorithm, and its disadvantage is that the results may not be accurate. By comparing the plots of (a) and (c), it can be seen that the number of iterations changes from 22 to 10, and there is a misclassification point in (c) but not in (a).