Data Mining Project 2 Report

309552007 袁鈺勛

A. Readme

Logistic Regression

Class of logistic regression by gradient descent

Argument	Description	Default	Type
-l, --learning_rate	Learning rate	0.1	Float
-r, --regularization	0: without L2 regularization, 1: with L2 regularization	0 (0-1)	Int
-p, --penalty	Hyperparameter of regularization	1.0	Float

Run

Argument	Description	Default
-l, --learning_rate	Learning rate	0.1
-r, --regularization	0: without L2 regularization, 1: with L2 regularization	0 (0-1)
-p, --penalty	Hyperparameter of regularization	1.0
-m, --mode	0: cross validation, 1: prediction	0 (0-1)
-v, --verbosity	verbosity level	0 (0-1)

```
\$ python3 logistic_regression.py [-l learning_rate] [-r (0-1)] [-p penalty] [-m (0-1)] [-v (0-1)]
```

Program file 中有一個 LogisticRegression class，”-l” 可以控制 learning rate，”-r”可以控制是否要使用 regularization，”-p”可以控制 regularization 的 hyperparameter。在 run program 的時候除了上面提到的三個 argument 之外，還有”-m”可以控制是要做 cross validation 還是做 prediction，”-v”可以控制是否要 print 一些額外的 information。Training data 和 testing data 以及 submission 這三個 file 要放在和 program file 同一層的 data directory 下。做完 prediction 的 result 會輸出成 result.csv。

B. Preprocessing

```
new_tr_info, new_ts_info = training_info.copy(), testing_info.copy()
number_of_tr = len(new_tr_info.index)
new_info = new_tr_info.append(new_ts_info)

# Fill NaN with 0
new_info['Comorbidities'] = new_info['Comorbidities'].fillna(0)
new_info['Antibiotics'] = new_info['Antibiotics'].fillna(0)
new_info['Bacteria'] = new_info['Bacteria'].fillna(0)

# Assign unique values to distinct values in the column
new_info['Comorbidities'] = new_info.groupby('Comorbidities').ngroup()

# Assign 0 if it's 0, 1 if it's a string
new_info['Antibiotics'] = new_info.groupby('Antibiotics').ngroup().astype(bool).astype(int)
new_info['Bacteria'] = new_info.groupby('Bacteria').ngroup().astype(bool).astype(int)

# Write them back to training and testing
new_tr_info[['Comorbidities', 'Antibiotics', 'Bacteria']] = new_info[
                                                    ['Comorbidities', 'Antibiotics', 'Bacteria']].iloc[
                                                :number_of_tr]
new_ts_info[['Comorbidities', 'Antibiotics', 'Bacteria']] = new_info[
                                                    ['Comorbidities', 'Antibiotics', 'Bacteria']].iloc[
                                                number_of_tr:]
```

上圖為”predict_info_fixer” function，會將共病症、抗生素、細菌的缺格補上 0，並且幫共病症各種類別給上特定的數值，抗生素和細菌則是非 0 的話給 1，0 的話就給 0。

```
# Group all training data by patient number
sectors = training_tpr.groupby('No')

# Get means of each patient
training_patients = sectors.mean()

# Standardize all columns
ss = StandardScaler()
training_patients[['T', 'P', 'R', 'NBPS', 'NBPD']] = ss.fit_transform(
    training_patients[['T', 'P', 'R', 'NBPS', 'NBPD']])

# Get mean and variance of training data
m = ss.mean_
variance = ss.var_

# Group all testing data by patient number
sectors = testing_tpr.groupby('No')

# Get means of each patient
testing_patients = sectors.mean()

# Scale all columns
testing_patients[['T', 'P', 'R', 'NBPS', 'NBPD']] = (testing_patients[
                                                ['T', 'P', 'R', 'NBPS', 'NBPD']] - m) / np.sqrt(variance)
```

上圖為”predict_tpr_fixer” function，他會將 training data 做標準化後，將標準化時算出的 mean 和 variance 用來將 testing data 標準化，以避免 feature 間單位和數值區間的不同。

C. Feature Selection

```
# Use mutual information to select top k features
list_of_col = SelectKBest(mutual_info_classif, k=k).fit(training_data, training_target).get_support(indices=True)
features = list(map(list(training_data).__getitem__, list_of_col))

return features
```

利用 mutual information 來取出前 k 個和 label 最相關的 feature。

D. Model Construction

```python
# Calculate parameters
num_of_data = len(training_data)

# Set up Φ and group
group = training_data['Target'].to_numpy().reshape((num_of_data, 1))
del training_data['Target']
self.features = list(training_data)
num_of_features = len(self.features)
phi = np.ones((num_of_data, num_of_features + 1))
phi[:, 1:] = training_data.to_numpy()

# Get gradient descent result
self.omega = self.gradient_descent(phi, group, num_of_features)

return self.omega
```

上圖為 model 的”fit” function，他會先取出每個 data entity 對應到的 label，以及取出 training 所用到的 feature，而且會將 data 組成一個新的 phi 用於 gradient descent 的計算。

```python
# Set up initial guess of omega
omega = np.zeros((num_of_features + 1, 1))

# Get optimal weight vector omega
count = 0
while True:
    count += 1
    old_omega = omega.copy()

    # Update omega
    if self.regularization:
        # With L2 penalty
        omega -= self.learning_rate * (
                self.get_delta_j(phi, omega, group) - self.penalty * omega - 0.75 * old_omega) / len(phi)
    else:
        # Without L2 penalty
        omega -= self.learning_rate * (self.get_delta_j(phi, omega, group) - 0.75 * old_omega) / len(phi)

    if np.linalg.norm(omega - old_omega) < 1e-7 or count > 5000:
        break

return omega
```

```python
return phi.T.dot(expit(phi.dot(omega)) - group)
```

上兩張圖分別為 model 的”gradient_descent” function 和”get_delta_j” function，當 gradient descent 收斂的時候便可以得到 optimal weight omega，而且收斂的方式區分為有 regularization 以及沒有 regularization，兩種方法都會使用 momentum term。

```
# Calculate parameters
testing_data = test_data[self.features]
num_of_data = len(testing_data)
num_of_features = len(list(testing_data))

# Set up Φ
phi = np.ones((num_of_data, num_of_features + 1))
phi[:, 1:] = testing_data.to_numpy()

# Get results of gradient descent
weight = weight.reshape((len(weight), 1))
result = expit(phi.dot(weight))
result[result ≥ 0.5] = 1
result[result < 0.5] = 0
result = result.reshape(num_of_data).astype(int)

return result
```

上圖為 model 的"prediction" function，他會將 testing data 組成的 phi 和 weight 餵給 logistic function，機率大於等於 0.5 的就歸類為 label 1，小於 0.5 的就歸類給 label 0。

```
# Calculate parameters
testing_data = test_data[self.features]
num_of_data = len(testing_data)
num_of_features = len(list(testing_data))

# Set up Φ
phi = np.ones((num_of_data, num_of_features + 1))
phi[:, 1:] = testing_data.to_numpy()

# Get results of gradient descent
result = expit(phi.dot(weight)).reshape(num_of_data)

return result
```

上圖為 model 的"pred_prob" function，同樣會將 testing data 以及 weight 餵給 logistic function，並且回傳 data 可能為 label 1 的機率。

E. Validation

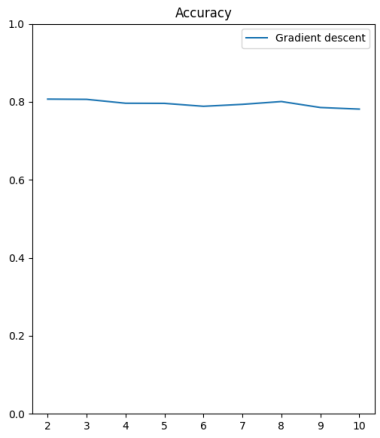
```
# Setup K fold
skf = RepeatedStratifiedKFold(n_repeats=10, random_state=0)
```

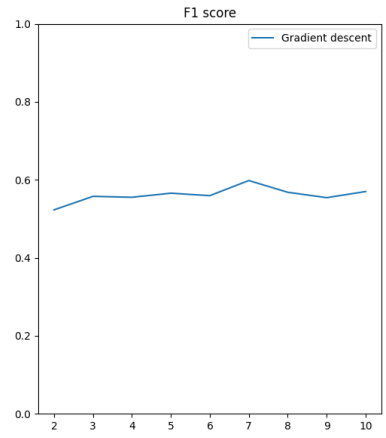
使用 stratified k-fold 來驗證，預設 5 個 fold，跑 10 次來測試。

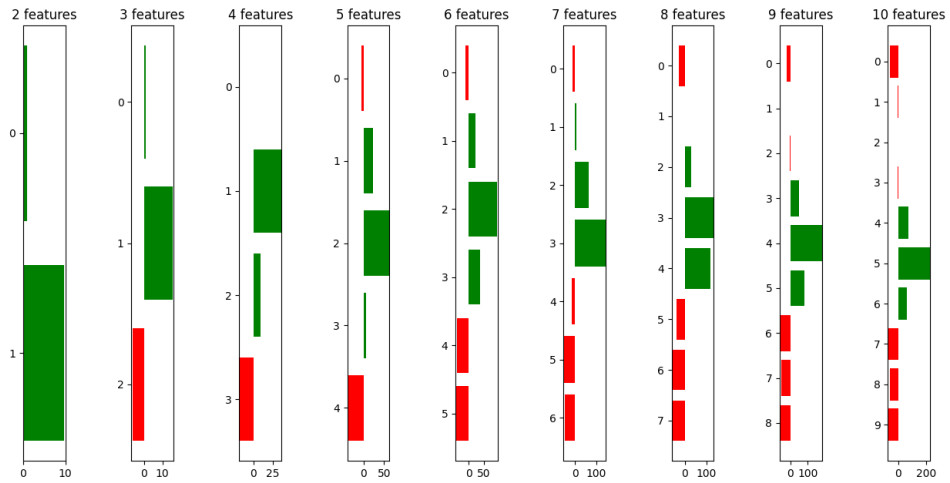
```
for train_index, test_index in skf.split(training_data, training_target):
    # Get training set and testing set
    data_train, target_train = training_data.iloc[train_index.tolist()], training_target.iloc[
        train_index.tolist()]
    data_test, target_test = training_data.iloc[test_index.tolist()], training_target.iloc[test_index.tolist()]

    # Use training set to select features
    for k in range(2, total_features):
        info_log(f'=== Iteration: {iteration}, Num of features: {k} ===')
        features = feature_selection(data_train, target_train, k)
```

每次從 training data set 中取出 training data 以及 testing data，並利用 training data 選取要使用的 feature。

```
=== Accuracy ===
=== Gradient descent ===
2: 0.806875
3: 0.80625
4: 0.79625
5: 0.7959375
6: 0.7884375
7: 0.7934375
8: 0.800625
9: 0.7853125
10: 0.78125


=== F1 score ===
=== Gradient descent ===
2: 0.5228350302943015
3: 0.557486417041236
4: 0.5551275136354518
5: 0.5655815459121346
6: 0.5592895246009338
7: 0.5980255647394759
8: 0.5679440583720459
9: 0.5541210297710737
10: 0.5698802034408847
```

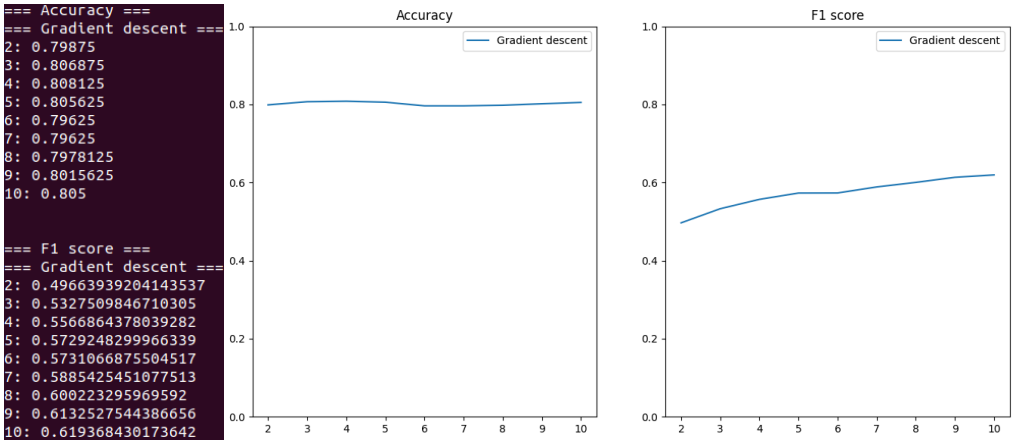


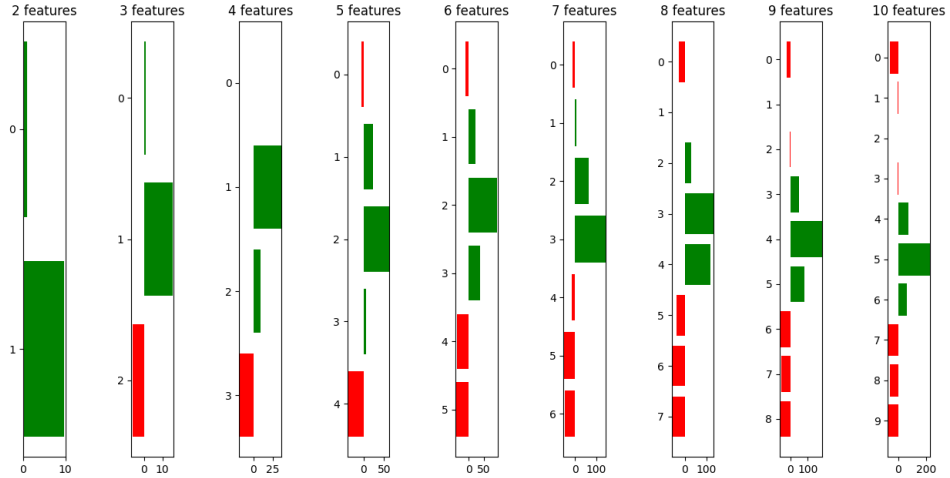




上面三張圖為沒有 regularization 的 validation 結果，左上圖為取不同數量的 feature 得到的各自平均 accuracy 以及平均 f1-score，右上圖即為左上圖的視覺化，第三張即是不同數量 feature 的情況底下得到的平均 weight。

```
=== Accuracy ===
=== Gradient descent ===
2: 0.79875
3: 0.806875
4: 0.808125
5: 0.805625
6: 0.79625
7: 0.79625
8: 0.7978125
9: 0.8015625
10: 0.805


=== F1 score ===
=== Gradient descent ===
2: 0.49663939204143537
3: 0.5327509846710305
4: 0.5566864378039282
5: 0.5729248299966339
6: 0.5731066875504517
7: 0.5885425451077513
8: 0.600223295969592
9: 0.6132527544386656
10: 0.619368430173642
```



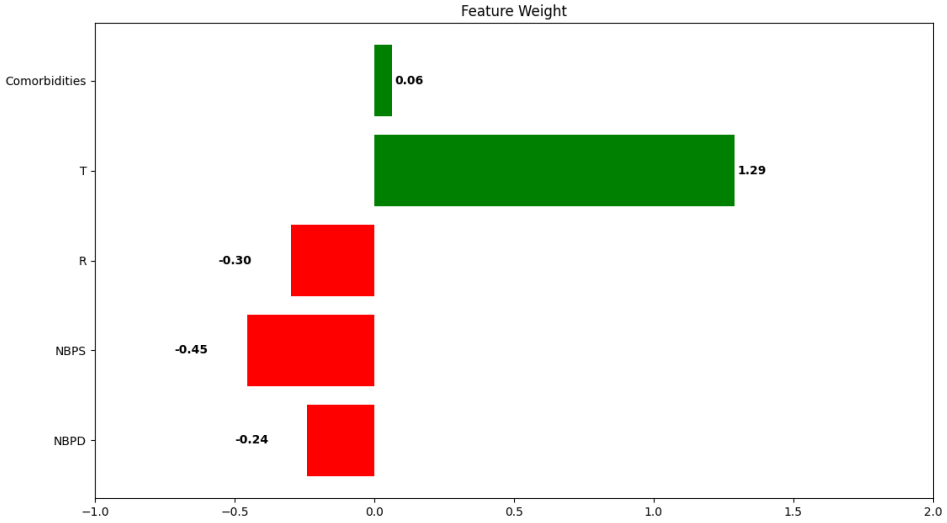


上面三張圖為有 regularization 的 validation 結果，可以看出他相對沒有 regularization 的結果較為穩定，所以在做 prediction 時是採用有 regularization 的結果。

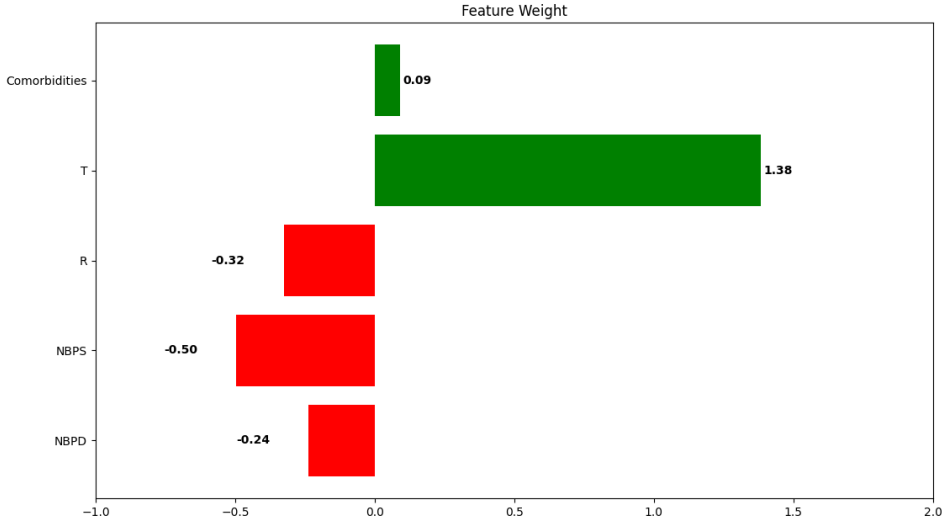
F. Prediction

```python
fig = plt.figure(1)
fig.canvas.set_window_title('Feature Weight')

ax = fig.add_subplot(1, 1, 1)

omega = self.omega.copy().reshape(len(self.omega))
colors = ['g' if value > 0 else 'r' for value in omega[1:]]
y_pos = np.arange(len(self.features))

ax.barh(y_pos, omega[1:], align='center', color=colors)
for idx, value in reversed(list(enumerate(omega[1:]))):
    ax.text(value + 0.01 if value > 0 else value - 0.26, idx, f'{value:.2f}', color='black', fontweight='bold',
            ha='left', va='center')
ax.set_xlim(floor(min(omega[1:])), ceil(max(omega[1:])))
ax.set_yticks(y_pos)
ax.set_yticklabels(self.features)
ax.invert_yaxis()
ax.set_title('Feature Weight')

plt.tight_layout()
plt.show()
```

上圖是 model 的”visualize” function，他會將 fit 得到的 feature 以及對應的 weight 視覺化成下面兩張圖類型的 horizontal bar chart。



上圖為沒有 regularization 方式做 prediction 後得到的 feature weight。



上圖為有 regularization 方式做 prediction 後得到的 feature weight。