One Drug Sensitivity Estimation Based on Adaptive Empirical Conditional Expectation (AECE) II

ZHAO Yuqi
Department of Mathematics
HKUST

FAN Min
Department of Mathematics
HKUST

PAN Hui
Department of ECE
HKUST

Abstract

In the previous project report, we proposed an algorithm called *Adaptive Empirical Conditional Expectation*, focusing on one drug sensitivity prediction. Although the performance of it was pretty good compared with other machine learning algorithms, there are some potential problems and space to improve the algorithm. In this report, we propose an advanced *Adaptive Empirical Conditional Expectation* which adopts bootstrap method to stabilize $\mathbb{E}(y)$ in the prior algorithm. The experiment showed that the revised method could give mean square error 3.044, better than the previous one.

1 Problem Statement and Data Setup

The dataset consists of 642 cancer cell line samples in response to one drug, Afatinib, with 60 features as gene mutation status. The response y as drug sensitivity is measured by logarithmic IC50, and the feature X_{ij} as mutation status is binary indicating whether the corresponding gene j of cell line i is mutated or not.

Therefore the feature matrix X is n by p and the response vector y is n dimensional, where $n = 642$ and $p = 60$.

A random subset of 100 cell line responses are withdrawn to the test sample. The basic problem is to predict the responses of these test samples based on their binary predictors/features.

As we mentioned before, most classic algorithm had bad performance. And our empirical conditional expectation algorithm gave a better result in this dataset. However, the estimation may sensitive to the dataset.

2 Review of Previous Empirical Conditional Estimation

The original regression problem is actually compute conditional expectation $\mathbb{E}(y|X)$ as prediction, where X is the features of a specific sample. Usually, putting some model structure (linear model, nearest neighbors) will give good performance. However in the case that those models do not work well, the reasonable way is to compute the empirical conditional expectation $\hat{\mathbb{E}}(y|X)$. This method is practicable benefit from the binary feature of gene status that we can match the train data to the test one. In this way we can avoid to model the mechanism/structure of drug process.

However, X contain too many dimensions thus lead to small sample size for each configuration and make the estimator unstable. Under this situation, feature selection is necessary to make sure sufficient sample size. We used T-test to construct the feature series to be selected and empirical variance to determined the threshold of feature number. It must be mentioned that the number of feature is adaptive to the test sample, which means for different test sample, we use different features.

2.1 Feature Selection Series via P-value

In this dataset, the feature space is 60-dimensional. But the specific drug, Afatinib, may only take effect to some few genes. we can use T-test to select the target genes. In particular, to feature i, we can test the hypothesis

$$H_0 : \mathbb{E}(y|X_i = 0) = \mathbb{E}(y|X_i = 1) \quad \text{versus} \quad H_1 : \mathbb{E}(y|X_i = 0) \neq \mathbb{E}(y|X_i = 1).$$

The less p-value is, the more significant the drug is to gene mutation feature i. Thus we set the threshold γ to choose the features whose p-value is smaller than γ.

Based on p-value, we can construct a feature series which the first element of this series is the most significant feature for the response value meanwhile the last one is the most insignificant one.

2.2 Threshold Decision

Once we get the feature series, indicated by$[X_1, \cdots, X_{60}]$, we can choose first K features in next step. However, we did not use a unified threshold K because with different test sample configuration the number of matching train sample in train dataset may be varies seriously. Therefore we used different K_i for different test sample feature X_i^{Test} in our algorithm.

Given a special test sample X_i^{Test} or X_i for short, suppose y_i is its true response value need to be predicted. We assumed the model

$$y_i = f(X_i) + \epsilon.$$

Thus we have error decomposition

$$\begin{aligned}
\mathbb{E}(\hat{y}_i - y_i)^2 &= \mathbb{E}(\hat{y}_i - \mathbb{E}\hat{y}_i)^2 + (\mathbb{E}\hat{y}_i - \mathbb{E}y_i)^2 + \mathbb{E}(\mathbb{E}y_i - y_i)^2 \\
&= \mathbb{E}(y(X_1, \cdots, X_{K_i}) - \mathbb{E}y(X_1, \cdots, X_{K_i}))^2 + (\mathbb{E}y(X_1, \cdots, X_{K_i}) - \mathbb{E}y_i)^2 \\
&\quad + \mathbb{E}(\mathbb{E}y_i - y_i)^2 \\
&= \mathrm{Var}(\hat{y}_i) + \mathrm{Bias}(\hat{y}_i)^2 + \mathrm{Var}(y_i)
\end{aligned}$$

We could not change the last term $\mathrm{Var}(y_i)$ since it is intrinsic. But we can change the K_i to make the first two terms as small as possible. To choose a good threshold K_i we must estimate the first two terms.

$\mathrm{Bias}(\hat{y}_i)$

To estimate bias, we need to estimate $\mathbb{E}(y_i)$. We used Expectation condition on full feature to estimate, i.e. $\hat{\mathbb{E}}(y|X_1, \cdots, X_{60})$. However, in train set there may not exist sample such that its configuration matching the given one. In this case we must drop features one by one until there exist sample to obtain existing expectation, for example $\hat{\mathbb{E}}(y|X_1, \cdots, X_{58})$ where 2 features dropped.

$\mathrm{Var}(\hat{y}_i)$

We can estimate the variance of our estimation from train set in terms of different setting K_i.

In particular, we used ‘Early Stopping’ strategy to avoid overfitting. We dropped feature one by one according to the feature series, and computed the sum of empirical variance of $\hat{y}_i$ and bias square $\mathrm{Bias}(\hat{y})_i$, stopped when the sum start increasing. Therefore the threshold K_i would be selected.

2.3 Empirical Conditional Expectation with Adaptive Feature Selection

Once having selected features $[X_1, \cdots, X_{K_i}]$, we can compute the empirical conditional expectation $\hat{\mathbb{E}}(y|X_1, \cdots, X_{K_i})$ for given cell line features x by simply taking the mean of all samples whose selected features are equal to the given ones, in practice

$$\hat{y}_i = \hat{\mathbb{E}}(y|X_1, \cdots, X_{K_i}) = \sum_{k=1}^{n} y_k \prod_{j \leq K_i} \chi_{\{X_{kj} = x_j\}},$$

where χ characteristic function.

2.4 Weakness of the Previous

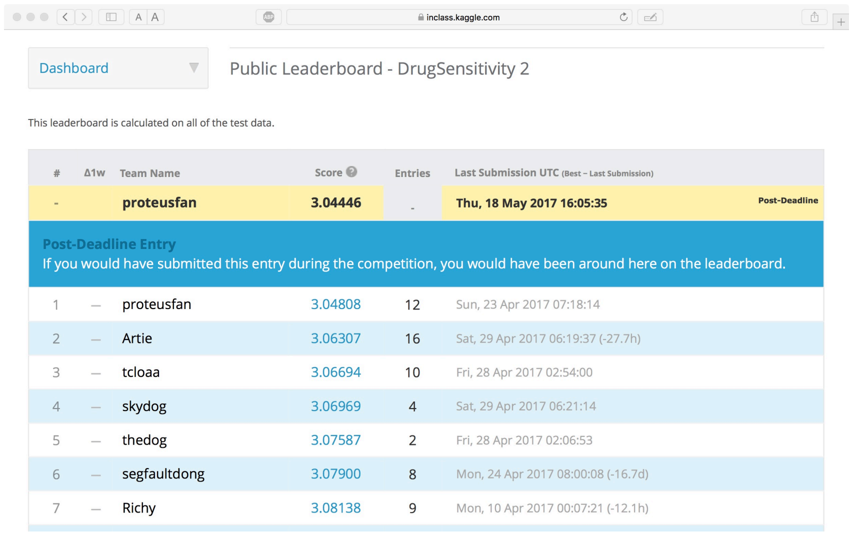
In the description above, we found that the estimation of $\mathbb{E}(y_i)$ might be very sensitive to the dataset. We used the sample mean of $y(X_1, \cdots, X_t)$ with the largest t such that sample size is large than 1. This setting may give rise to problem since the sample size is too small thus the estimation is very unstable. In the improvement setting we use bootstrap method to cover this problem.

3 Advanced Setting with Bootstrap

We used bootstrap method to stabilize the expectation of $y(X_1, \cdots, X_t)$ noted in last section. For each loop, we constructed new train set by random sampling with replacement on original train set for n times. In such way we get the same size train set as the original one. Then, the p-value of each feature is recalculated and the approximation of $y(X_1, \cdots, X_n)$ is approached based on new p-value. The reason why using the new p-value instead of the old one is that we simulate original data probability density by bootstrap and in each loop we can only get information from new dataset (noticed that for each loop, there are around $1/e$ samples not in the new train set. Thus in the process of estimate $\mathbb{E}y_i$). After getting bunch of approximation of $y(X_1, \cdots, X_n)$, we calculate the average of approximation of $y(X_1, \cdots, X_n)$, replacing the old one, and then implement the previous algorithm on original dataset.

4 Experiment

We implement Bootstrap 100 times for experiment and update our IC50s estimation to kaggle in class. The final result mean square error is 3.044, which beat the previous algorithm and others. Moreover, our algorithm is much more stable to reach a high-level performance comparatively.


inclass.kaggle.com

Dashboard

Public Leaderboard - DrugSensitivity 2

This leaderboard is calculated on all of the test data.

#	Δ1w	Team Name	Score	Entries	Last Submission UTC (Best – Last Submission)
-		proteusfan	3.04446	-	Thu, 18 May 2017 16:05:35 Post-Deadline

Post-Deadline Entry
If you would have submitted this entry during the competition, you would have been around here on the leaderboard.

#	Δ1w	Team Name	Score	Entries	Last Submission UTC (Best – Last Submission)
1	—	proteusfan	3.04808	12	Sun, 23 Apr 2017 07:18:14
2	—	Artie	3.06307	16	Sat, 29 Apr 2017 06:19:37 (-27.7h)
3	—	tcloaa	3.06694	10	Fri, 28 Apr 2017 02:54:00
4	—	skydog	3.06969	4	Sat, 29 Apr 2017 06:21:14
5	—	thedog	3.07587	2	Fri, 28 Apr 2017 02:06:53
6	—	segfaultdong	3.07900	8	Mon, 24 Apr 2017 08:00:08 (-16.7d)
7	—	Richy	3.08138	9	Mon, 10 Apr 2017 00:07:21 (-12.1h)

Figure 4.1: The result of Kaggle Competition

5 Conclusion

In this report, we propose a revised framework based on *Adaptive Empirical Conditional Expectation*. This methods is inspired by the feature structure which is in fact binary. Accordingly, we can use training set to match the result of test data and avoid modeling the mechanism of drug process.

Notice this framework is highly interpretive, almost tuning free (the result is robust to hyper-parameter) thanks to adaptive feature selection, and performs well on the given dataset.

Furthermore, this framework is potential with respect to the feature selection process. Notice that we use sample size as hyper-parameter which can also be learned from data via variance, thus the total framework is turning free at all. Besides, the interaction between different gene mutations can also be considered in our framework.

6 Appendix: Python Code

Listing 1: main.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import pandas as pd

# Data Loading
submission = pd.read_csv('./Data/submission.csv')
TrueIC50s = pd.read_csv('./Data/TrueIC50s.csv')
df = pd.read_csv('./Data/OneDrug.csv')
train_index = df.index[np.logical_not(np.isnan(df['IC50s']))]
test_index = df.index[np.isnan(df['IC50s'])]
train_feature = df.values[train_index,3:]
train_IC50s = df.values[train_index,2]
test_feature = df.values[test_index,3:]
test_IC50s = np.zeros(test_feature.shape[0])

# adaptive empirical conditional expectation method + bootstrap
from AECE import aece
from sklearn.metrics import mean_squared_error

# normal method
model = aece()
model.fit(train_feature, train_IC50s)
test_IC50s = model.predict(test_feature)

# mean of bootstrap
model = aece()
model.fit(train_feature, train_IC50s)
test_IC50s = model.predict_bootstrap(test_feature)

mse = mean_squared_error(test_IC50s, TrueIC50s['IC50'])

submission['IC50'] = test_IC50s
submission.to_csv('./Data/aece_bootstrap_mean.csv', index=False)
```

Listing 2: AECE.py

```
# module for drug sensitivity
import numpy as np
from scipy.stats import ttest_ind
import time

class aece(object):
    def __init__(self, num_bootstrap = 100 ,limit = 2):
        self.limit = limit
        self.num_bootstrap = num_bootstrap

    def __pvalue(X, y):
        ttest = [None] * X.shape[1]
        for i in range(X.shape[1]):
            sample1 = y[ X[:,i] == 1 ]
            sample0 = y[ X[:,i] == 0 ]
            if len(sample1) > 1:
                ttest[i] = ttest_ind(sample1, sample0, equal_var = False)
            else:
                ttest[i] = [np.nan]*2
        ttest = np.array(ttest)
        return ttest[:,1]

    def fit(self, X, y):
        self.X = X
        self.y = y
        self.p = aece.__pvalue(X, y)

    def __group_data(X, y):
        # in case there is only one sample
        if len(X.shape) == 1:
            X = X[np.newaxis,:]
        # group all data
        idx = list(range(X.shape[0]))
        group = []
        while len(idx)>0:
            standard = X[idx[0],:]
            group.append(np.array([idx[0],y[idx[0]]])[np.newaxis,:])
            del idx[0]
            if len(idx)>0:
                panel = idx.copy()
                for element in panel:
                    if sum(abs(X[element,:] - standard))==0:
                        group[-1] = np.r_[group[-1],
                                np.array([element,y[element]])[np.newaxis,:]]
                        idx.remove(element)
        group_pos = [int(element[0,0]) for element in group]
        group_feature = X[group_pos,:]
        return group, group_feature

    def __check(sample, group, group_feature):
        # Check
        num_group = len(group)
        for i in range(num_group):
            if sum(abs(sample - group_feature[i])) == 0:
                element = group[i]
                return element.shape[0], element[:,1].mean(), element[:,1].var()
        return 0, np.nan, np.nan

    def predict(self, test_feature):
        t0 = time.time()
        # Data setting
        test_IC50s = np.zeros(test_feature.shape[0])
        test_group, group_feature = aece.__group_data(test_feature,test_IC50s)
        group_relation = [list(map(int,element[:,0])) for element in test_group]
        # Prediction
        p_thresholding = np.sort(np.unique(self.p[~np.isnan(self.p)]))[::-1]
        pred = []
        for sample in group_feature:
            est_mean = np.inf
            est_var = np.inf
            mean_std = []
            for i in range(len(p_thresholding)):
                print('sample %d/%d, thresholding %d/%d.' %
                      (len(pred), len(group_feature)-1, i, len(p_thresholding)-1))
                p_loc = self.p <= p_thresholding[i]
                temp_group, temp_group_feature = aece.__group_data(self.X[:,p_loc],
                                                                   self.y)
                temp_sample = sample[p_loc]
                num_same, temp_mean, temp_var = aece.__check(
                        temp_sample, temp_group, temp_group_feature)
                if num_same >= self.limit:
                    mean_std = mean_std or temp_mean
                    temp_var = (temp_mean-mean_std)**2 + temp_var/(num_same-1)
                    if temp_var <= est_var:
                        est_mean = temp_mean
                        est_var = temp_var
                        if i == len(p_thresholding)-1:
                            pred.append(est_mean)
                    else:
                        pred.append(est_mean)
                        break
            print('prediction:␣%f.' % (pred[-1]))
        # Inverse projection to test_IC50s
        for i in range(len(pred)):
            test_IC50s[group_relation[i]] = pred[i]
        # Time running
        t1 = time.time()
        print('Time␣running:␣%f.' % (t1-t0))
        return test_IC50s

    def __bootstrap_mean(self, sample, sample_counter):
        # Initialization
        mean_std = 0
        # Bootstrap
        for j in range(self.num_bootstrap):
            print('bootstrap:␣%d/%d,␣sample:␣%d/%d.' % (j, self.num_bootstrap-1,
                  sample_counter[0], sample_counter[1]))
            index = list(map(int,len(self.X)*np.random.rand(len(self.X))))
            X_new = self.X[index,:]
            y_new = self.y[index]
            p_new = aece.__pvalue(X_new, y_new)
            p_thresholding = np.sort(np.unique(p_new[~np.isnan(p_new)]))[::-1]
            for i in range(len(p_thresholding)):
                p_loc = p_new <= p_thresholding[i]
                group, group_feature = aece.__group_data(X_new[:,p_loc], y_new)
                temp_sample = sample[p_loc]
                num_same, temp_mean, temp_var = aece.__check(
                        temp_sample, group, group_feature)
                if num_same >= 1:
                    print('%d␣iteration␣in␣this␣phase.' % (i))
                    mean_std += temp_mean
                    break
        mean_std /= self.num_bootstrap
        print('bootstrap␣mean:␣%f.' % (mean_std))
        return mean_std

    def predict_bootstrap(self, test_feature):
        t0 = time.time()
        # Data setting
        test_IC50s = np.zeros(test_feature.shape[0])
        test_group, group_feature = aece.__group_data(test_feature, test_IC50s)
        group_relation = [list(map(int,element[:,0])) for element in test_group]
        # Prediction
```

```
p_thresholding = np.sort(np.unique(self.p[~np.isnan(self.p)]))[::-1]
pred = []
for sample in group_feature:
    est_mean = np.inf
    est_var = np.inf
    mean_std = self.__bootstrap_mean(sample,
                                     (len(pred), len(test_group)-1))
    for i in range(len(p_thresholding)):
        print('sample␣%d/%d,␣thresholding␣%d/%d.' %
              (len(pred), len(group_feature)-1, i, len(p_thresholding)-1))
        p_loc = self.p <= p_thresholding[i]
        temp_group, temp_group_feature = aece.__group_data(self.X[:,p_loc],
                                                           self.y)
        temp_sample = sample[p_loc]
        num_same, temp_mean, temp_var = aece.__check(
                temp_sample, temp_group, temp_group_feature)
        if num_same >= self.limit:
            temp_var = (temp_mean-mean_std)**2 + temp_var/(num_same-1)
            if temp_var <= est_var:
                est_mean = temp_mean
                est_var = temp_var
                if i == len(p_thresholding)-1:
                    pred.append(est_mean)
            else:
                pred.append(est_mean)
                break
    print('prediction:␣%f.' % (pred[-1]))
# Inverse projection to test_IC50s
for i in range(len(pred)):
    test_IC50s[group_relation[i]] = pred[i]
# Time running
t1 = time.time()
print('Time␣running:␣%f.' % (t1-t0))
return test_IC50s
```

References

[1] James C Costello. etc. (2014) A community effort to assess and improve drug sensitivity prediction algorithms.Nature Biotechnology 32, 12021212.