Homework 1

Machine Learning and Deep Learning for Data Science

2022-24790 Sungwoo PARK
Graduate School of Data Science

April 6, 2022

Setup and Acknowledgement

For this homework, I worked on the `Colaboratory` powered by `Google`, which is a web IDE for `Python` based on the `Jupyter notebook` environment.

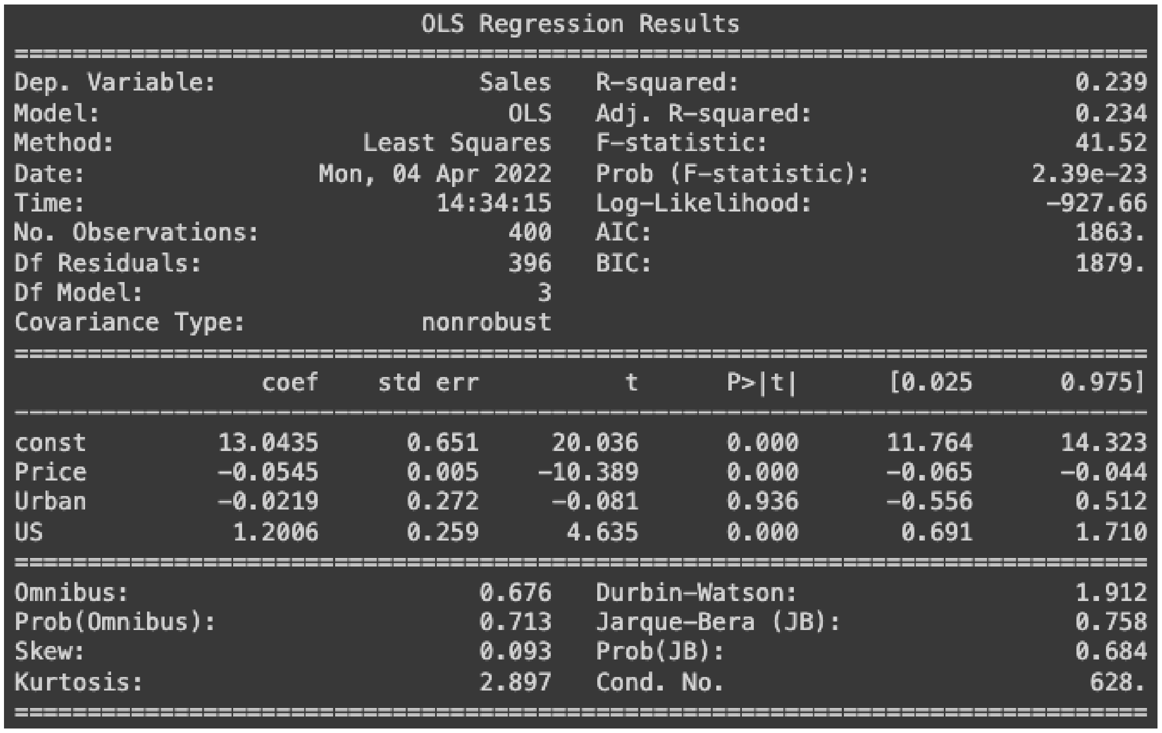
Furthermore, I used the `statsmodels` and the `scikit-learn` for solving the problems, and both are widely used for data analysis. Specifically, the former one provides various options for strict interpretation, so that it is useful for traditional statistical analysis. On the other hand, the latter one is more adopted for prediction, since many machine learning algorithms or performance metrics are well implemented in it. Every main task was done by the `statsmodels` and the `scikit-learn` was used just for verification.

Finally, I set my random seed as `20140801` and discussed with other classmates for this homework: 유지상, 이다예, 이세라, 임상수, 최광호, and they are all from the GSDS.

Problem 1

(a)

From the given dataset, it is seen that the variables `Urban` and `US` are both binary categorical variables, containing one of the two values: `Yes` or `No`. Thus before fitting a model, I converted them into numerical variables, mapping `Yes` to 1 and `No` to 0. After that, I split the samples into the set of exaplanatory variables X, consisting of `Price`, `Urban`, and `US`, and the target variable Y. Then I regressed Y on X by fitting an OLS(Ordinary Least Squares) model. The result is as shown in the figure below.



```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Sales   R-squared:                       0.239
Model:                            OLS   Adj. R-squared:                  0.234
Method:                 Least Squares   F-statistic:                     41.52
Date:                Mon, 04 Apr 2022   Prob (F-statistic):           2.39e-23
Time:                        14:34:15   Log-Likelihood:                -927.66
No. Observations:                 400   AIC:                             1863.
Df Residuals:                     396   BIC:                             1879.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         13.0435      0.651     20.036      0.000      11.764      14.323
Price         -0.0545      0.005    -10.389      0.000      -0.065      -0.044
Urban         -0.0219      0.272     -0.081      0.936      -0.556       0.512
US             1.2006      0.259      4.635      0.000       0.691       1.710
==============================================================================
Omnibus:                        0.676   Durbin-Watson:                   1.912
Prob(Omnibus):                  0.713   Jarque-Bera (JB):                0.758
Skew:                           0.093   Prob(JB):                        0.684
Kurtosis:                       2.897   Cond. No.                         628.
==============================================================================
```

Figure 1: The result of a multiple regression

As we can see at the upper-right part of the figure 1, the R^2 was computed as 0.239.

(b)

The coefficient of a linear regression model is typically interpreted as "an average amount of change in the target value as a unit of an explanatory variable increases." Following this, the coefficient of `Price`, which is computed as -0.0545, is an average amount of change in sales as a unit of `Price` - which is measured by $1 - increases. This interpretation makes sense since the customers tend to purchase less as the price goes higher than before. The fact that the amount of this coefficient is so low may attribute to the difference in scale. The standard deviation of the target is estimated as 2.82, while that of the `Price` is estimated as 23.68, almost 10 times greater. If we normalize them so that both variables have a unit standard deviation, the coefficient would be estimated much higher.

On the other hand, when it comes to binary, or dummy, variables, the interpretation of coefficient becomes different. Since those binary digits, 1 and 0, have no numerical meaning but just to discriminate between two groups. Therefore, the correct interpretation of the coefficient of a dummy variable should be "an average amount of difference in the target variable of the group represented by 1, compared to the group represented by 0." Following this, we can interpret the coefficients of `Urban` and `US` like the statements below:

- People who live in the urban area purchased 0.0219 unit less than those who do not.
- People from the United States purchased 1.2 units more than those who are not.

(c)

The equation form of this model is:

$$\texttt{Sales} = \beta_0 + \beta_1 \cdot \texttt{Price} + \beta_2 \cdot D_{\texttt{Urban}} + \beta_3 \cdot D_{\texttt{US}} + \epsilon \tag{1}$$

where β_0 denotes the intercept, $D_{\texttt{Urban}}$ denotes a dummy variable, with binary digits, that represents whether an observation lives in the urban area or not, and $D_{\texttt{US}}$ denotes whether an observation is from the United States or not. ϵ is an error term, which is assumed to follow the normal distribution $N\left(0, \sigma^2\right)$.

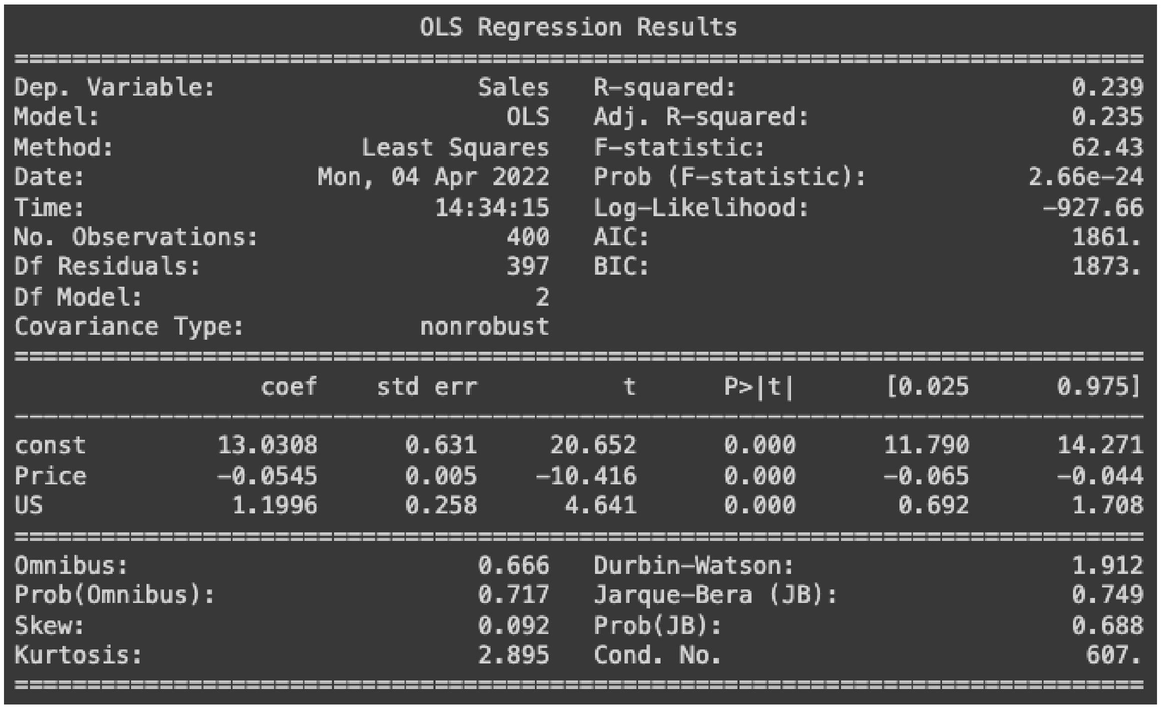
(d)

We can determine whether to reject the null hypothesis or not based on the p-value of each coefficient. The p-value represents the probability for the coefficient to be estimated as its actual estimate from the fitted model under the null hypothesis. Typically, if the probability is less than 0.05, we can reject the null hypothesis.

From the figure 1, we can see that the p-value of the `Price` and `US` are less than 0.05, so we can reject the null hypothesis for each of them. However, the p-value of `Urban` is computed almost close to 1, thus the null hypothesis cannot be rejected. In other words, there is little evidence that whether an observation lives in the urban area or not is associated with the outcome.

(e)

The result estimated from the smaller model, without `Urban`, is as shown in the figure below.



```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Sales   R-squared:                       0.239
Model:                            OLS   Adj. R-squared:                  0.235
Method:                 Least Squares   F-statistic:                     62.43
Date:                Mon, 04 Apr 2022   Prob (F-statistic):           2.66e-24
Time:                        14:34:15   Log-Likelihood:                -927.66
No. Observations:                 400   AIC:                             1861.
Df Residuals:                     397   BIC:                             1873.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         13.0308      0.631     20.652      0.000      11.790      14.271
Price         -0.0545      0.005    -10.416      0.000      -0.065      -0.044
US             1.1996      0.258      4.641      0.000       0.692       1.708
==============================================================================
Omnibus:                        0.666   Durbin-Watson:                   1.912
Prob(Omnibus):                  0.717   Jarque-Bera (JB):                0.749
Skew:                           0.092   Prob(JB):                        0.688
Kurtosis:                       2.895   Cond. No.                         607.
==============================================================================
```

Figure 2: The result of a multiple regression with the smaller model

(f)

The first model's R^2 is computed a bit higher than that of the second model. In fact, the R^2 of the first model is computed as 0.23928 and that of the second model is computed as 0.23926 when I verified it again using the `scikit-learn` package. However, considering the well-known fact that "the bigger the model is, the higher the R^2 is", it seems to be hard to tell which one is better. Instead, when we see the adjusted R^2, which considers the model size as well, the smaller model scored the higher metric. Although the difference seems to be small, but it is bigger than the difference in R^2 between the models. Therefore, we cannot say that one model is better than the other based on just R^2, while we can tell that the second model is better in terms of adjusted R^2.

(g)

The 95% confidence interval for each coefficient, including the intercept, is as below.

- Intercept: $[11.79032, 14.271265]$

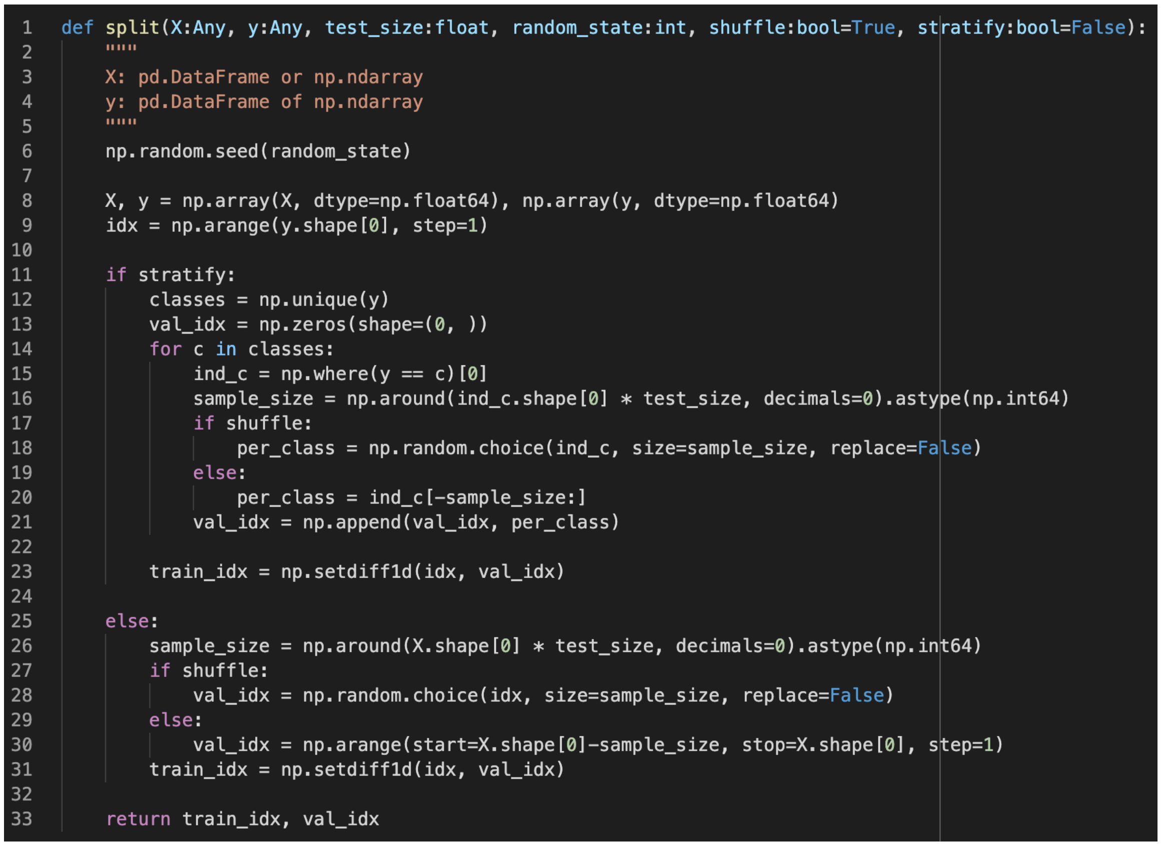
- Price: $[-0.06476, -0.044195]$
- US: $[0.69152, 1.707766]$

Problem 2

(a)

First of all, since the target variable, default, and the variable student are both categorical variables containing two values: Yes and No, so I converted both of them into dummy variables.

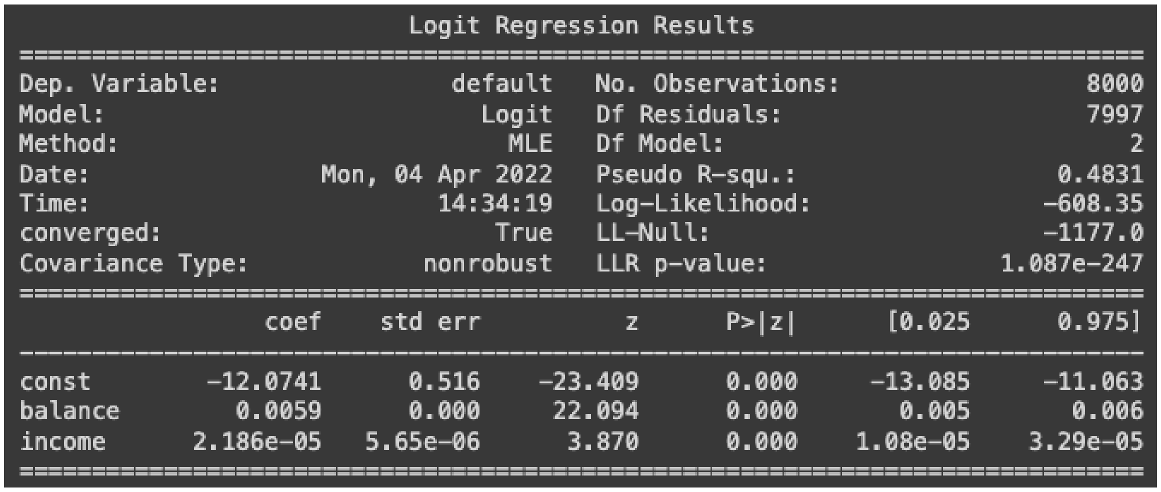
Next, I split the samples into the training set and the validation set, setting the proportion at 0.2, which means that the size of the validation set is 20% of the total samples. I implemented the split function manually using Numpy, and the code snippet is as below.



```python
def split(X:Any, y:Any, test_size:float, random_state:int, shuffle:bool=True, stratify:bool=False):
    """
    X: pd.DataFrame or np.ndarray
    y: pd.DataFrame of np.ndarray
    """
    np.random.seed(random_state)

    X, y = np.array(X, dtype=np.float64), np.array(y, dtype=np.float64)
    idx = np.arange(y.shape[0], step=1)

    if stratify:
        classes = np.unique(y)
        val_idx = np.zeros(shape=(0, ))
        for c in classes:
            ind_c = np.where(y == c)[0]
            sample_size = np.around(ind_c.shape[0] * test_size, decimals=0).astype(np.int64)
            if shuffle:
                per_class = np.random.choice(ind_c, size=sample_size, replace=False)
            else:
                per_class = ind_c[-sample_size:]
            val_idx = np.append(val_idx, per_class)

        train_idx = np.setdiff1d(idx, val_idx)

    else:
        sample_size = np.around(X.shape[0] * test_size, decimals=0).astype(np.int64)
        if shuffle:
            val_idx = np.random.choice(idx, size=sample_size, replace=False)
        else:
            val_idx = np.arange(start=X.shape[0]-sample_size, stop=X.shape[0], step=1)
        train_idx = np.setdiff1d(idx, val_idx)

    return train_idx, val_idx
```

Figure 3: Split function implemented manually

With the training set, I fitted the logistic regression model provided by the statsmodels package. The result is as shown in the figure below.



```
                           Logit Regression Results
==============================================================================
Dep. Variable:                default   No. Observations:                 8000
Model:                          Logit   Df Residuals:                     7997
Method:                           MLE   Df Model:                            2
Date:                Mon, 04 Apr 2022   Pseudo R-squ.:                  0.4831
Time:                        14:34:19   Log-Likelihood:                -608.35
converged:                       True   LL-Null:                       -1177.0
Covariance Type:            nonrobust   LLR p-value:                 1.087e-247
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -12.0741      0.516    -23.409      0.000     -13.085     -11.063
balance        0.0059      0.000     22.094      0.000       0.005       0.006
income      2.186e-05   5.65e-06      3.870      0.000    1.08e-05    3.29e-05
==============================================================================
```

Figure 4: The result of a logistic regression

Finally, I also estimated the test error, which is computed as the misclassification rate. To obtain this, I compared the predicted label, each of whose values equals to 1 if the predicted probability is greater than 0.5 and equals to 0 otherwise, with the true label of the validation set. Then I counted the number of the predicted labels that are equal to the corresponding true values and divided it by the total number of observations in the validation set. As the final step, I subtracted the divided value from 1.

The hold-out validation error was computed as 0.0315, or 3.15%.

(b)

The functioal form of the logistic regression follows that of the sigmoid function, and it looks like the equation below.

$$\texttt{default} = \frac{\exp\left(\beta_0 + \beta_1 \cdot \texttt{balance} + \beta_2 \cdot \texttt{income}\right)}{1 + \exp\left(\beta_0 + \beta_1 \cdot \texttt{balance} + \beta_2 \cdot \texttt{income}\right)} \tag{2}$$

After modifying this equation we can get the equation as below.

$$\frac{\texttt{default}}{1 - \texttt{default}} = \exp\left(\beta_0 + \beta_1 \cdot \texttt{balance} + \beta_2 \cdot \texttt{income}\right) \tag{3}$$

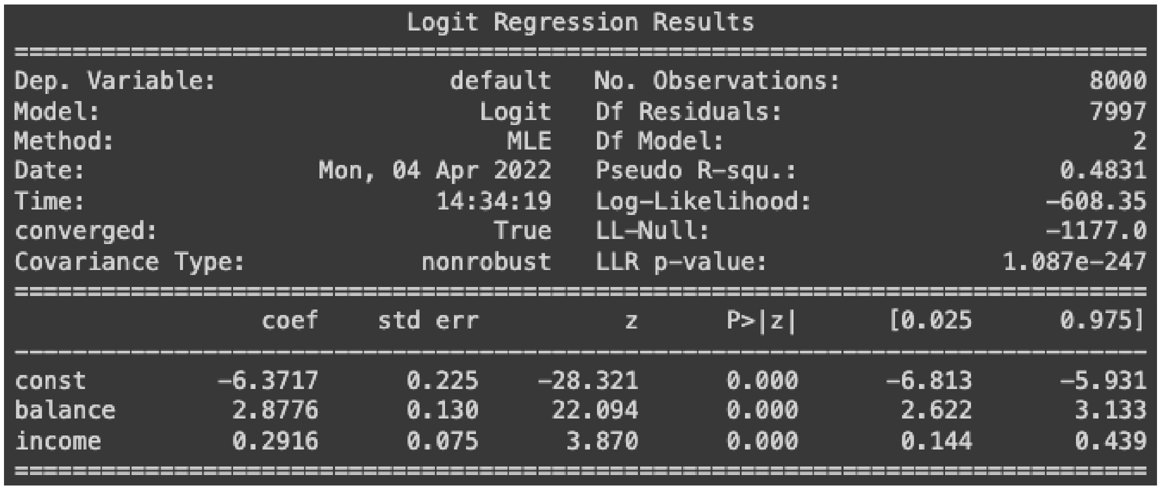
By taking the logarithm to both sides,

$$\log\left(\frac{\texttt{default}}{1 - \texttt{default}}\right) = \beta_0 + \beta_1 \cdot \texttt{balance} + \beta_2 \cdot \texttt{income} \tag{4}$$

This implies that the β_j denotes the average amount of change in the log-odds of an observation being likely to be default as the variable j increases by one unit. Now, the coefficients from the figure 4 can be interpreted as:

- A unit($1) increase in the amount of `balance` increases the log-odds of an observation being likely to be default by 0.0059.
- A unit($1) increase in the amount of `income` increases the log-odds of an observation being likely to be default by 2.186×10^{-5}.

However, the values of both coefficients are too small, and this may attribute to the scale of both variables. So I estimated another logistic regression model after normalizing both explanatory variables, and the result is like the figure below. Note that the normalization of each variable is implemented by subtracting its mean and divide the centered variable by its standard deviation, so that the variable has zero mean and a unit standard deviation after the normalization.



```
                           Logit Regression Results
==============================================================================
Dep. Variable:                default   No. Observations:                 8000
Model:                          Logit   Df Residuals:                     7997
Method:                           MLE   Df Model:                            2
Date:                Mon, 04 Apr 2022   Pseudo R-squ.:                  0.4831
Time:                        14:34:19   Log-Likelihood:                -608.35
converged:                       True   LL-Null:                       -1177.0
Covariance Type:            nonrobust   LLR p-value:                 1.087e-247
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -6.3717      0.225    -28.321      0.000      -6.813      -5.931
balance        2.8776      0.130     22.094      0.000       2.622       3.133
income         0.2916      0.075      3.870      0.000       0.144       0.439
==============================================================================
```

Figure 5: The result of a logistic regression after normalization

Now, the interpretation is changed as:

- 1 standard deviation(about $483.71) increase in the amount of `balance` increases the log-odds of an observation being likely to be default by 2.8776.
- 1 standard deviation(about $13336.64) increase in the amount of `income` increases the log-odds of an observation being likely to be default by 0.2916.

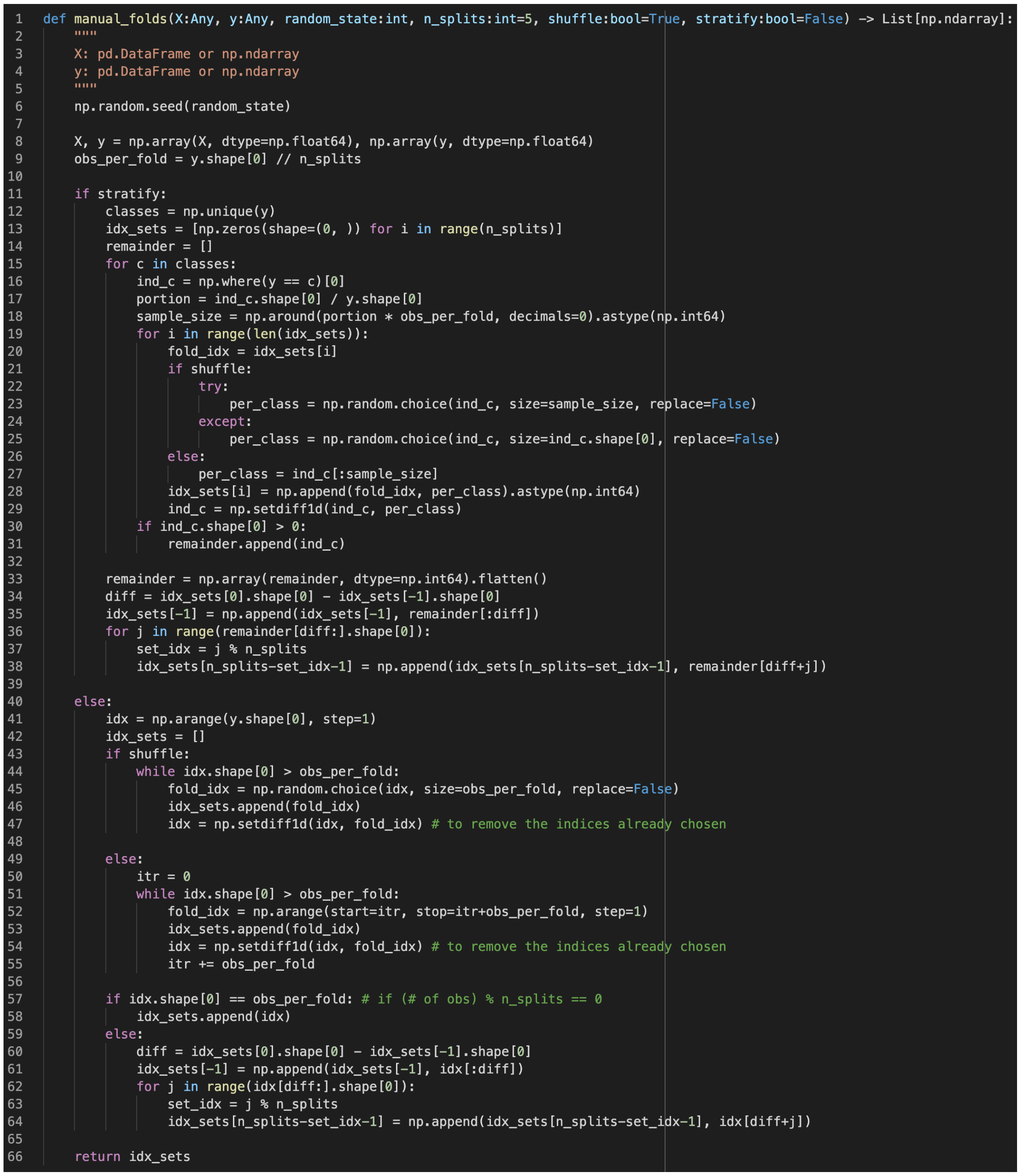
(c)

I implemented the K-fold cross-validation by defining a function using `Numpy`. First, I defined a function named `manual_folds` that returns a list of sets where each set contains the indices of each fold. The function requires arguments as listed below:

- `X`, `y`: The original dataset to perform the cross-validation.
- `n_splits`: The number of folds.
- `randon_state`: A random seed.
- `shuffle`: Whether to implement sampling under random process or not. It is set `True` by default.
- `stratify`: Whether to implement stratified sampling or not. It is set `False` by default.

If `shuffle` is `True`, the amount of $\frac{n}{k}$ indices are randomly chosen, where n denotes the total number of observations and k denotes the number of folds. Important thing is that the indices should be sampled without replacement, since the observations should not be overlapped in the k-fold cross-validation. To do this, the indices already chosen were removed. On the other hand, if `shuffle` is `False`, then the indices are chosen sequentially.

The figure below is the code snippet for the k-fold split function.



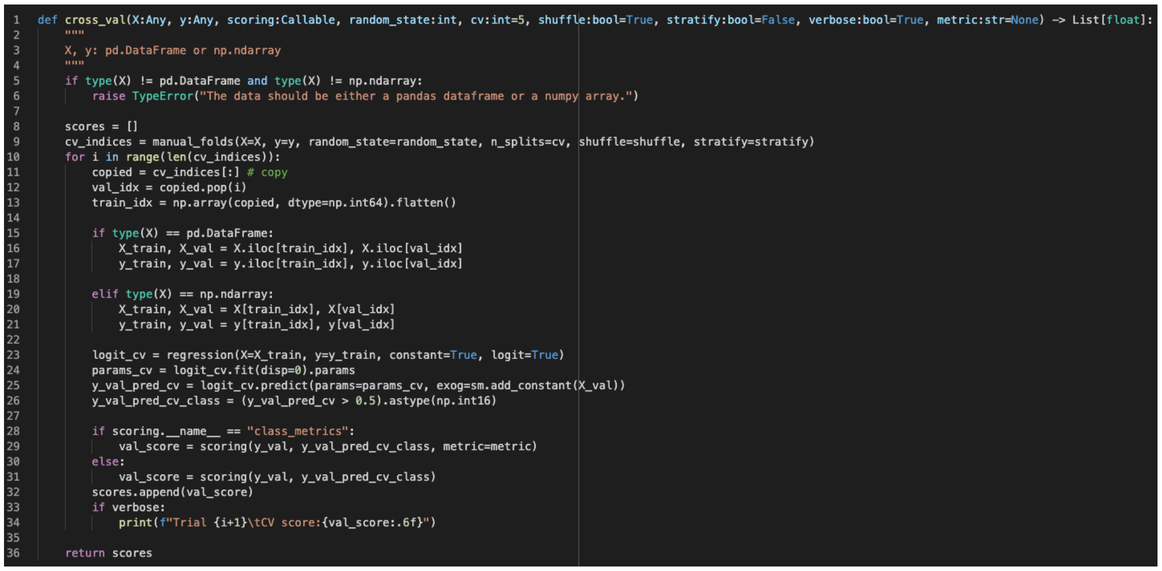
```python
def manual_folds(X:Any, y:Any, random_state:int, n_splits:int=5, shuffle:bool=True, stratify:bool=False) -> List[np.ndarray]:
    """
    X: pd.DataFrame or np.ndarray
    y: pd.DataFrame or np.ndarray
    """
    np.random.seed(random_state)

    X, y = np.array(X, dtype=np.float64), np.array(y, dtype=np.float64)
    obs_per_fold = y.shape[0] // n_splits

    if stratify:
        classes = np.unique(y)
        idx_sets = [np.zeros(shape=(0, )) for i in range(n_splits)]
        remainder = []
        for c in classes:
            ind_c = np.where(y == c)[0]
            portion = ind_c.shape[0] / y.shape[0]
            sample_size = np.around(portion * obs_per_fold, decimals=0).astype(np.int64)
            for i in range(len(idx_sets)):
                fold_idx = idx_sets[i]
                if shuffle:
                    try:
                        per_class = np.random.choice(ind_c, size=sample_size, replace=False)
                    except:
                        per_class = np.random.choice(ind_c, size=ind_c.shape[0], replace=False)
                else:
                    per_class = ind_c[:sample_size]
                idx_sets[i] = np.append(fold_idx, per_class).astype(np.int64)
                ind_c = np.setdiff1d(ind_c, per_class)
            if ind_c.shape[0] > 0:
                remainder.append(ind_c)

        remainder = np.array(remainder, dtype=np.int64).flatten()
        diff = idx_sets[0].shape[0] - idx_sets[-1].shape[0]
        idx_sets[-1] = np.append(idx_sets[-1], remainder[:diff])
        for j in range(remainder[diff:].shape[0]):
            set_idx = j % n_splits
            idx_sets[n_splits-set_idx-1] = np.append(idx_sets[n_splits-set_idx-1], remainder[diff+j])

    else:
        idx = np.arange(y.shape[0], step=1)
        idx_sets = []
        if shuffle:
            while idx.shape[0] > obs_per_fold:
                fold_idx = np.random.choice(idx, size=obs_per_fold, replace=False)
                idx_sets.append(fold_idx)
                idx = np.setdiff1d(idx, fold_idx) # to remove the indices already chosen

        else:
            itr = 0
            while idx.shape[0] > obs_per_fold:
                fold_idx = np.arange(start=itr, stop=itr+obs_per_fold, step=1)
                idx_sets.append(fold_idx)
                idx = np.setdiff1d(idx, fold_idx) # to remove the indices already chosen
                itr += obs_per_fold

        if idx.shape[0] == obs_per_fold: # if (# of obs) % n_splits == 0
            idx_sets.append(idx)
        else:
            diff = idx_sets[0].shape[0] - idx_sets[-1].shape[0]
            idx_sets[-1] = np.append(idx_sets[-1], idx[:diff])
            for j in range(idx[diff:].shape[0]):
                set_idx = j % n_splits
                idx_sets[n_splits-set_idx-1] = np.append(idx_sets[n_splits-set_idx-1], idx[diff+j])

    return idx_sets
```

Figure 6: Function for the k-fold split

With the list of sets of indices returned from this function, cross-validation is implemented by repeating the following algorithm:

1. Copy the list of sets of indices.
2. Pop out the `i`th element of the list and set it as indices of a validation set, and aggregate the rest and set it as indices of a training set.
3. Train the pre-defined model and measure the validation score.

The figure below is the code snippet for the cross-validation function.



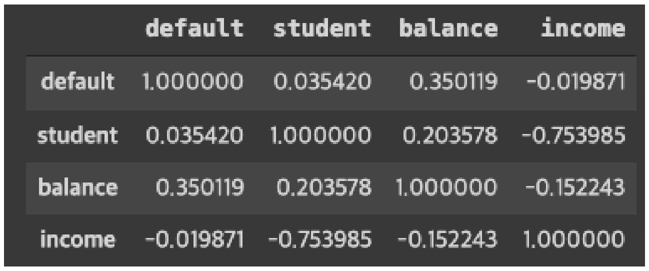
```python
def cross_val(X:Any, y:Any, scoring:Callable, random_state:int, cv:int=5, shuffle:bool=True, stratify:bool=False, verbose:bool=True, metric:str=None) -> List[float]:
    """
    X, y: pd.DataFrame or np.ndarray
    """
    if type(X) != pd.DataFrame and type(X) != np.ndarray:
        raise TypeError("The data should be either a pandas dataframe or a numpy array.")

    scores = []
    cv_indices = manual_folds(X=X, y=y, random_state=random_state, n_splits=cv, shuffle=shuffle, stratify=stratify)
    for i in range(len(cv_indices)):
        copied = cv_indices[:] # copy
        val_idx = copied.pop(i)
        train_idx = np.array(copied, dtype=np.int64).flatten()

        if type(X) == pd.DataFrame:
            X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
            y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

        elif type(X) == np.ndarray:
            X_train, X_val = X[train_idx], X[val_idx]
            y_train, y_val = y[train_idx], y[val_idx]

        logit_cv = regression(X=X_train, y=y_train, constant=True, logit=True)
        params_cv = logit_cv.fit(disp=0).params
        y_val_pred_cv = logit_cv.predict(params=params_cv, exog=sm.add_constant(X_val))
        y_val_pred_cv_class = (y_val_pred_cv > 0.5).astype(np.int16)

        if scoring.__name__ == "class_metrics":
            val_score = scoring(y_val, y_val_pred_cv_class, metric=metric)
        else:
            val_score = scoring(y_val, y_val_pred_cv_class)
        scores.append(val_score)
        if verbose:
            print(f"Trial {i+1}\tCV score:{val_score:.6f}")

    return scores
```

Figure 7: Function for the k-fold cross-validation

The validation error of the cross-validation is computed as 0.0263, or 2.63%, which is much lower than that of the hold-out validation in the part (a). However, it is hard to tell that the cross-validation is a better way than the hold-out method because of the randomness of the training-validation split.

(d)

By including another dummy variable `student`, the 5-fold cross-validation error is computed as 0.0271, or 2.71%, which is higher than the error without the variable. This implies that including the variable do not help reduce the test error rate, and the variable is not relevant to the prediction model as well. Furthermore, the correlation among the variables also seems to affect the test error rate. The below table shows the correlation coefficients among all variables.



	default	student	balance	income
default	1.000000	0.035420	0.350119	-0.019871
student	0.035420	1.000000	0.203578	-0.753985
balance	0.350119	0.203578	1.000000	-0.152243
income	-0.019871	-0.753985	-0.152243	1.000000

Figure 8: Correlation coefficient table

As we can see, the `student` and `income` are highly correlated in a negative way. Thus, by including `student`, the model might have suffered from the multicollinearity problem. This problem is likely to make a regression model unreliable, *i.e.* making the variance of the model higher, thus making the model vulnerable to overfitting due to the 'bias-variance trade-off.' The fact that overfitting could damage the predictive power of a model is well-known.

For verification, I tried the cross-validation again with only using `student` and `income`. In the experiment, the validation error was computed as 3.33%, much higher than the error from any other cases. This result seems to support the argument I have made above.

Extra work

Comparison with stratified sampling

An important thing to be considered is that the distribution of the target variable, `default`, is strongly imbalanced; the proportion of observations which were actually in default was only 3.33%. Therefore, it is critical to keep that proportion in both the training set and the validation set. I used the stratified sampling method to do this and compared the result with that above.

The largest improvement was shown in the hold-out validation method, where the misclassification rate reduced from 3.15% to 2.5%, by almost 21%. The cross-validation error has reduced just a little bit, from 2.63% to 2.62%. Interesting thing is that the performance of the cross-validation showed better performance than the hold-out method without stratified sampling, while the result under the stratified sampling was the opposite. Moreover, the results seem to confirm that the cross-validation shows more robustness than the hold-out validation, because the randomness of split is offset by averaging the scores in the cross-validation.

However, the relation between the models with or without the `student` has not changed: the model with the variable scored the validation error rate of 2.66%, still higher than that from the model without it.

Using different performance metrics

The purpose of this model is to capture which kind of people are likely to be in default. This implies that predicting `1` correctly is much more critical to this model than predicting `0`. However, the misclassifcation rate, or accuracy rate more precisely, does not care about the specific value of each estimate; it considers only whether the actual label is the same as the predicted label or not.

To address this problem, I also used other metrics, which do care about the specific value of each predicted label as well: *precision*, *recall*, *FPR*(False Positive Rate), *FNR*(False Negative Rate), and the *F1-score*. Intuitively, the precision measures how correctly the model classifies the true `1`'s compared to the false `1`'s, the labels whose actual value is not `1` but predicted as `1`.

The recall measures how much the model captures the true `1`'s among the entire labels whose actual value is `1`. FPR, on the other hand, measures how much the model misclassifies the actual `0`'s as `1`, and FNR measures the opposite case. Finally, the F1-score is computed as the harmonic mean of the precision and recall. It is known that for the precision, recall, f1-score, the closer the values go to 1, the better the model is.

Like the misclassification rate, all these metrics scored higher under the stratified sampling than otherwise. However, it is hard to say that the logistic regression is the best model for this dataset when we see the recall, FNR and the f1-score. They were all scored poor; the highest recall and f1-score were 0.3582 and 0.4898, respectively, and the lowest FNR was 0.6428. These scores are all computed under the "stratified and hold-out validation" setting. This implies that this model is not that good at capturing the observations that are truly likely to be default among those who actually are. Furthermore, from the FNR and FPR, it can be seen that there were more cases where the actual `1`'s were misclassified as `0` than the opposite case. It is also noteworthy that the precision score for the dataset with the `student`, especially under the non-stratified sampling, suffered from the `division-by-zero` problem. From these results, it could be learned that it is important, especially in classification with an imbalanced dataset, to split the training set and the validation set keeping the proportion of each class.

The specific scores are shown as in the table below.

Stratified	**Validation**	**Student**	**Precision**	**Recall**	**FPR**	**FNR**	**F1**
No	Hold-out	No	.5185	.2188	.0067	.7812	.3077
No	CV	No	.7382	.3276	.0039	.6724	.4528
No	CV	Yes	`NaN`	0	0	1	`NaN`
Yes	Hold-out	No	.7742	.3582	.0036	.6418	.4898
Yes	CV	No	.7400	.3246	.0038	.6754	.4496
Yes	CV	Yes	.7207	.3186	.0040	.6814	.4403

Table 1: Metric scores computed under each setting