# assignment_4

May 3, 2025

## 1 STA-6543 Assignment 4

Jason Gillette

### 1.1 Question 3

(a) Explain how k-fold cross-validation is implemented.

In k-fold cross-validation, the data set is randomly divided into k equal-sized folds. Chapter 5, section 1 of the text details the process as follows:

```
i. For each of the k iterations, one fold is set aside as the validation set, and the model is

ii. The model is then used to make predictions on the held-out fold, and an error is computed

iii. After all k iterations, the cross-validation error is estimated by averaging the k indivi
```

(b) What are the advantages and disadvantages of k-fold cross-validation relative to:

i. The validation set approach?

The advantage of k-fold cross validation over traditional validation is that it uses the entire dataset for both training and validation, which provides a more stable and reliable estimate of test error. Cross validated models tend to better fit and are less prone to variability caused by training-testing split. However, the disadvantage of cross-validation is it can be more resource intensive and complex to set up, although this is increasingly abstracted by various libraries.

ii. Leave One Out Cross Validation (LOOCV)?

The advantage of k-fold over LOOCV, is that k-fold is significantly less computationally expensive. Where k-fold may split on multiple observation (e.g. k+5, K=10), LOOCV performs a split of every single observation, training on n-1 observations for n iterations. Because the test hold-out is different across every iteration, LOOCV may also result in higher variance.

### 1.2 Question 5.

In Chapter 4, we used logistic regression to predict the probability of default using income and balance on the Default data set. We will now estimate the test error of this logistic regression model using the validation set approach. Do not forget to set a random seed before beginning your analysis.

[1]:
```
from ISLP import load_data
import pandas as pd

# Load the Default dataset
default = load_data('Default')
default.head()
```

[1]:

| | default | student | balance | income |
|---|---|---|---|---|
| 0 | No | No | 729.526495 | 44361.625074 |
| 1 | No | Yes | 817.180407 | 12106.134700 |
| 2 | No | No | 1073.549164 | 31767.138947 |
| 3 | No | No | 529.250605 | 35704.493935 |
| 4 | No | No | 785.655883 | 38463.495879 |

### 1.3 Question 5a.

Fit a logistic regression model that uses income and balance to predict default.

[3]:
```
import statsmodels.api as sm

# data preprocessing - convert 'default' to 0 (No) and 1 (Yes)
default['default01'] = (default['default'] == 'Yes').astype(int)

# Define predictors and response
X = default[['income', 'balance']]
X = sm.add_constant(X)  # add intercept
y = default['default01']

# Fit logistic regression
logit_model = sm.Logit(y, X).fit()

# Show model summary
logit_model.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.078948
         Iterations 10
```

[3]:

| | | | |
|---|---|---|---|
| **Dep. Variable:** | default01 | **No. Observations:** | 10000 |
| **Model:** | Logit | **Df Residuals:** | 9997 |
| **Method:** | MLE | **Df Model:** | 2 |
| **Date:** | Sat, 03 May 2025 | **Pseudo R-squ.:** | 0.4594 |
| **Time:** | 18:47:43 | **Log-Likelihood:** | -789.48 |
| **converged:** | True | **LL-Null:** | -1460.3 |
| **Covariance Type:** | nonrobust | **LLR p-value:** | 4.541e-292 |

| | coef | std err | z | P> \|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **const** | -11.5405 | 0.435 | -26.544 | 0.000 | -12.393 | -10.688 |
| **income** | 2.081e-05 | 4.99e-06 | 4.174 | 0.000 | 1.1e-05 | 3.06e-05 |
| **balance** | 0.0056 | 0.000 | 24.835 | 0.000 | 0.005 | 0.006 |

Possibly complete quasi-separation: A fraction 0.14 of observations can be perfectly predicted. This might indicate that there is complete quasi-separation. In this case some parameters will not be identified.

The model shows that balance is a strong predictor of default, while income has a weaker (larger coefficient 0.0056 vs 0.00002, higher z-statistic 24.8 vs 4.2), though statistically significant, effect. The model explains nearly half of the deviance (R2 = 0.4594).

The warning on quasi-separation suggests caution, as some cases may be too easy to classify perfectly. Not sure what to make of this?

### 1.4 Question 5bi.

Using the validation set approach, estimate the test error of this model. In order to do this, you must perform the following steps:

i. Split the sample set into a training set and a validation set

[4]:
```
import numpy as np
from sklearn.model_selection import train_test_split

# Set seed for reproducibility
np.random.seed(42)

# Define predictors and response
X = default[['income', 'balance']]
y = default['default01']

# train/validation split
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.5,␣
↪random_state=1)
```

### 1.5 Question 5bii.

Fit a multiple logistic regression model using only the training observations.

[6]:
```
import statsmodels.api as sm

# Add intercept to training and validation sets
X_train_sm = sm.add_constant(X_train)
X_val_sm = sm.add_constant(X_val)

# Fit logistic regression
logit_model = sm.Logit(y_train, X_train_sm).fit()
```

```
Optimization terminated successfully.
         Current function value: 0.079028
         Iterations 10
```

### 1.6 Question 5biii.

iii. Obtain a prediction of default status for each individual in the validation set by computing the posterior probability of default for that individual, and classifying the individual to the default category if the posterior probability is greater than 0.5.

[11]:
```
# Predict probabilities for validation set
probs = logit_model.predict(X_val_sm)

# Classify: default if P(default) > 0.5
y_pred = (probs > 0.5).astype(int)
print(y_pred)
```

```
3472    0
5095    0
9504    0
5786    0
8758    0
       ..
1625    1
990     0
2408    0
2568    0
5075    0
Length: 5000, dtype: int64
```

### 1.7 Question 5biv.

Compute the validation set error, which is the fraction of the observations in the validation set that are misclassified.

[ ]:
```
# misclassification rate
val_error = np.mean(y_pred != y_val)
print(f"Validation Set Error: {val_error:.4f}")
```

```
Validation Set Error: 0.0250
```

Error rate of 2.5%, model correctly classified 97.5% of validation observations using the 0.5 threshold.

### 1.8 Question 5c.

Repeat the process in (b) three times, using three different splits of the observations into a training set and a validation set. Comment on the results obtained.

[10]:
```python
# Track results
errors = []

# TODO: Is this question is asking to randomize the observations per split, not␣
 ↪the split size???
for seed in [42, 43, 44]:
    np.random.seed(seed)

    # Train/test split
    X_train, X_val, y_train, y_val = train_test_split(
        default[['income', 'balance']],
        default['default01'],
        test_size=0.5,
        random_state=seed
    )

    # Add intercept
    X_train_sm = sm.add_constant(X_train)
    X_val_sm = sm.add_constant(X_val)

    # Fit model
    model = sm.Logit(y_train, X_train_sm).fit(disp=False)

    # Predict and classify
    probs = model.predict(X_val_sm)
    preds = (probs > 0.5).astype(int)

    # Compute error
    error = np.mean(preds != y_val)
    errors.append(error)
    print(f"Seed {seed} - Validation Set Error: {error:.4f}")
```

```
Seed 42 - Validation Set Error: 0.0258
Seed 43 - Validation Set Error: 0.0282
Seed 44 - Validation Set Error: 0.0252
```

Relatively stable test error found across different random splits confirms that the model indicating it generalizes well. The small variability in error (within ~0.2%) implies low variance in performance.

### 1.9 Question 5d.

Now consider a logistic regression model that predicts the probability of default using income, balance, and a dummy variable for student. Estimate the test error for this model using the validation set approach. Comment on whether or not including a dummy variable for student leads to a reduction in the test error rate.

[ ]:
```python
# Convert student' to binary dummy variable
default['student01'] = (default['student'] == 'Yes').astype(int)
```

```python
# Set seed
np.random.seed(42)

# define predictors / response
X = default[['income', 'balance', 'student01']]
y = default['default01']

# Train/test split
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.5,␣
 ↪random_state=42)

# add intercept
X_train_sm = sm.add_constant(X_train)
X_val_sm = sm.add_constant(X_val)

# Fit model
model = sm.Logit(y_train, X_train_sm).fit()

# predict and classify
probs = model.predict(X_val_sm)
preds = (probs > 0.5).astype(int)

# Compute test error
val_error = np.mean(preds != y_val)
print(f"Validation Set Error with student dummy var: {val_error:.4f}")
```

```
Optimization terminated successfully.
         Current function value: 0.077900
         Iterations 10
Validation Set Error with student dummy var: 0.0256
```

Adding student dummy variable did not significantly reduce the test error. In fact, the error remained pretty much the same / increased slightly (e.g. 2.50% to 2.56%). This suggests that student provides little additional predictive value beyond what is already captured by balance and income.

### 1.10 Question 6.

We continue to consider the use of a logistic regression model to predict the probability of default using income and balance on the Default data set. In particular, we will now compute estimates for the standard errors of the income and balance logistic regression coefficients in two different ways: (1) using the bootstrap, and (2) using the standard formula for computing the standard errors in the `sm.GLM()` function. Do not forget to set a random seed before beginning your analysis.

### 1.11 Question 6a.

Using the summarize() and sm.GLM() functions, determine the estimated standard errors for the coefficients associated with income and balance in a multiple logistic regression model that uses

both predictors.

[ ]:
```
from ISLP import load_data
from ISLP.models import summarize
import statsmodels.api as sm

# load
default = load_data('Default')

# Convert default to binary response
default['default01'] = (default['default'] == 'Yes').astype(int)

# define predictors and response
X = default[['income', 'balance']]
X = sm.add_constant(X)  # add intercept
y = default['default01']

# Fit logistic regression model binomial / logistic
model = sm.GLM(y, X, family=sm.families.Binomial())
results = model.fit()

# summary
summarize(results) # TODO: balance std err 0.0???
```

[ ]:

| | coef | std err | z | P>\|z\| |
|---|---|---|---|---|
| const | -11.540500 | 0.435000 | -26.544 | 0.0 |
| income | 0.000021 | 0.000005 | 4.174 | 0.0 |
| balance | 0.005600 | 0.000000 | 24.835 | 0.0 |

## 2 Question 6b.

Write a function, boot_fn(), that takes as input the Default data set as well as an index of the observations, and that outputs the coefficient estimates for income and balance in the multiple logistic regression model.

[ ]:
```
import statsmodels.api as sm
import numpy as np

def boot_fn(data, index):
    """
    Fit a logistic regression model on a subset of the data defined by `index`,
    and returns the estimated coefficients for income and balance.
    param: data (DataFrame): The Default dataset.
    param: index (array-like): Row indices to include in the sample.
    returns: ndarray: Coefficients for [intercept, income, balance].
    """
    # select resampled data
```

```
    sample = data.iloc[index]

    # Define predictors and response
    X = sample[['income', 'balance']]
    X = sm.add_constant(X)
    y = (sample['default'] == 'Yes').astype(int)

    # fit model
    model = sm.Logit(y, X).fit(disp=False)

    return model.params.values  # [intercept, income, balance]
```

### 2.1 Question 6c.

Following the bootstrap example in the lab, use your boot_fn() function to estimate the standard errors of the logistic regression coefficients for income and balance.

[23]:
```
from ISLP import load_data
import statsmodels.api as sm
import numpy as np

# reload Default data and convert default to binary
default = load_data('Default')
default['default01'] = (default['default'] == 'Yes').astype(int)

def boot_coeffs(D, idx):
    """
    Returns both income and balance coefficients from logistic regression.
    """
    sample = D.loc[idx]
    X = sample[['income', 'balance']]
    X = sm.add_constant(X)
    y = sample['default01']
    model = sm.GLM(y, X, family=sm.families.Binomial()).fit()
    return model.params[['income', 'balance']].values  # returns a vector


# Run bootstrap func
rng = np.random.default_rng(seed=42)
B = 100
coefs = np.zeros((B, 2))  # 2 for income and balance

for b in range(B):
    idx = rng.choice(default.index, size=len(default), replace=True)
    coefs[b] = boot_coeffs(default, idx)

# Compute standard errors
```

```
bootstrap_se = coefs.std(axis=0)
print(f"Bootstrap SEs:\nIncome: {bootstrap_se[0]}\nBalance: {bootstrap_se[1]}")
```

```
Bootstrap SEs:
Income: 5.243875934638313e-06
Balance: 0.00023046818518135682
```

### 2.2 Question 6d.

Comment on the estimated standard errors obtained using the `sm.GLM()` function and using the bootstrap.

balance SE from 6(a) is different, but still small and in agreement with standard formula. Maybe a pandas issue in 6(a)?

### 2.3 Question 9.

We will now consider the Boston housing data set, from the ISLP library.

[25]:
```
## Load Boston housing data
boston = load_data('Boston')
```

### 2.4 Question 9a.

Based on this data set, provide an estimate for the population mean of medv. Call this estimate μˆ.

[26]:
```
# Compute the estimate of the population mean
mu_hat = boston['medv'].mean()
print(f"Estimated population mean of medv: {mu_hat:.4f}")
```

```
Estimated population mean of medv: 22.5328
```

### 2.5 Question 9b.

Provide an estimate of the standard error of mu_hat. Interpret this result. Hint: We can compute the standard error of the sample mean by dividing the sample standard deviation by the square root of the number of observations

[27]:
```
# Number of observations
n = boston.shape[0]

# Sample standard deviation
std_dev = boston['medv'].std(ddof=1)  # use ddof=1 for sample SD???

# SE of the mean
se_mu = std_dev / np.sqrt(n)

print(f"Sample mean: {mu_hat:.4f}") # same
print(f"Standard error: {se_mu:.4f}")
```

```
Sample mean: 22.5328
Standard error: 0.4089
```

### 2.6 Question 6c.

Now estimate the standard error of μˆ using the bootstrap. How does this compare to your answer from (b)?

[29]:
```python
# new bootstrap func
def boot_mu(D, idx):
    return D['medv'].iloc[idx].mean()

# redefine boot_se
def boot_SE(func, D, n=None, B=1000, seed=0):
    rng = np.random.default_rng(seed)
    first_, second_ = 0, 0
    n = n or D.shape[0]

    for _ in range(B):
        idx = rng.choice(D.index, size=n, replace=True)
        value = func(D, idx)
        first_ += value
        second_ += value**2

    return np.sqrt(second_ / B - (first_ / B)**2)

bootstrap_se = boot_SE(boot_mu, boston, B=100, seed=0)
print(f"Bootstrap standard error: {bootstrap_se:.4f}")
```

```
Bootstrap standard error: 0.4315
```

The bootstrap standard error of the sample mean was estimated to be 0.4315, which is slightly higher than the analytical standard error of approximately 0.4089. This small difference suggests that that both methods provide consistent estimates.

### 2.7 Question 9d.

Based on your bootstrap estimate from (c), provide a 95 % confidence interval for the mean of medv. Compare it to the results obtained by using Boston[‘medv’].std() and the two standard error rule (3.9). Hint: You can approximate a 95 % confidence interval using the formula [μˆ − 2SE(μˆ), μˆ + 2SE(μˆ)].

[32]:
```python
## bootstap CI
ci_lower = mu_hat - 2 * bootstrap_se
ci_upper = mu_hat + 2 * bootstrap_se
print(f"95% CI (Bootstrap): [{ci_lower:.4f}, {ci_upper:.4f}]")

## analytical CI
ci_lower_analytical = mu_hat - 2 * se_mu
```

```
ci_upper_analytical = mu_hat + 2 * se_mu
print(f"95% CI (Analytical): [{ci_lower_analytical:.4f}, {ci_upper_analytical:.
  ↪4f}]")
```

```
95% CI (Bootstrap): [21.6699, 23.3957]
95% CI (Analytical): [21.7151, 23.3505]
```

Both intervals are very similar, both centered around the sample mean (~22.53), with the bootstrap interval being slightly wider.

### 2.8 Question 9e.

Based on this data set, provide an estimate, μˆmed, for the median value of medv in the population.

[33]:
```
mu_hat_median = boston['medv'].median()
print(f"Estimated population median: {mu_hat_median:.4f}")
```

```
Estimated population median: 21.2000
```

### 2.9 Question 9f.

We now would like to estimate the standard error of μˆmed. Unfortunately, there is no simple formula for computing the standard error of the median. Instead, estimate the standard error of the median using the bootstrap. Comment on your findings.

[39]:
```
# median bootstrp
def boot_median(D, idx):
    return D['medv'].iloc[idx].median()

# boot SE again
def boot_SE(func, D, n=None, B=100, seed=0):
    rng = np.random.default_rng(seed)
    first_, second_ = 0, 0
    n = n or D.shape[0]

    for _ in range(B):
        idx = rng.choice(D.index, size=n, replace=True)
        value = func(D, idx)
        first_ += value
        second_ += value**2

    return np.sqrt(second_ / B - (first_ / B)**2)

# Run
bootstrap_se_median = boot_SE(boot_median, boston, B=100, seed=0)
print(f"Bootstrap standard error of median: {bootstrap_se_median:.4f}")
```

```
Bootstrap standard error of median: 0.3444
```

We repeatedly sampled from the population and calculated the median medv, the typical deviation from the true median would be around 0.34 units. This is slightly lower than the standard error of the mean (~0.41), suggesting that the median is a relatively good estimator.

### 2.10 Question 9g.

Based on this data set, provide an estimate for the tenth percentile of medv in Boston census tracts. Call this quantity μˆ0.1. (You can use the `np.percentile()` function.)

[40]:
```
import numpy as np

# Compute the 10th percentile
mu_hat_0_1 = np.percentile(boston['medv'], 10)
print(f"Estimated 10th percentile of medv: {mu_hat_0_1:.4f}")
```

```
Estimated 10th percentile of medv: 12.7500
```

### 2.11 Question 9h.

Use the bootstrap to estimate the standard error of μˆ0.1. Comment on your findings.

[41]:
```
# percentile bootstrap
def boot_percentile_10(D, idx):
    return np.percentile(D['medv'].iloc[idx], 10)


# boot SE again
def boot_SE(func, D, n=None, B=100, seed=0):
    rng = np.random.default_rng(seed)
    first_, second_ = 0, 0
    n = n or D.shape[0]

    for _ in range(B):
        idx = rng.choice(D.index, size=n, replace=True)
        value = func(D, idx)
        first_ += value
        second_ += value**2

    return np.sqrt(second_ / B - (first_ / B)**2)

# Run
bootstrap_se_percentile_10 = boot_SE(boot_percentile_10, boston, B=100, seed=0)
print(f"Bootstrap standard error of 10th percentile:␣
 ↪{bootstrap_se_percentile_10:.4f}")
```

```
Bootstrap standard error of 10th percentile: 0.4894
```

The standard error of the 10th percentile is ~0.48 which is significantly worse than median, and only slightly worse than mean standard error.