Wasil Engel
12231558
PSet 3

I collaborated with Gina Son.

To see more screenshots of the plots I created for the coding part of the assignment, please see my Jupyter notebook on Github here: https://github.com/Wasil-UChi/Machine-Learning

Ch 5 RESAMPLING METHODS

# 6 p.199

```
path = '/Users/wasilengel/Desktop/School/Harris/Machine Learning/Pset 3/Data-Default.csv'

default = pd.read_csv(path)

default.head() # 10000 rows × 5 columns

# a

encoding_dict = {"Yes": 1, "No": 0}
default["default"] = default["default"].map(encoding_dict)
default.head()

X = default[["balance", "income"]]
X = sm.add_constant(X)
y = default["default"]

results = sm.Logit(y, X).fit()
print(results.summary())

# The SE for balance are 0.000 and for income 4.99e-06.

# b

def get_indices(data, num_samples):
   np.random.seed(1)
   positive_data = data[data["default"] == 1]
   negative_data = data[data["default"] == 0]
```

```
    positive_indices = np.random.choice(positive_data.index, int(num_samples / 4), replace =
True)
    negative_indices = np.random.choice(negative_data.index, int(3*num_samples / 4), replace
= True)
    total = np.concatenate([positive_indices, negative_indices])
    np.random.shuffle(total)
    return total

def boot_fn(data, index):
    np.random.seed(1)
    X = data[["balance", "income"]].loc[index]
    y = data["default"].loc[index]
    lr = LogisticRegression()
    lr.fit(X, y)
    intercept = lr.intercept_
    coef_balance = lr.coef_[0][0]
    coef_income = lr.coef_[0][1]
    return [intercept, coef_balance, coef_income]

intercept, coef_balance, coef_income = boot_fn(default, get_indices(default, 100))
print(f"Intercept is {intercept}, the coeff of balance is {coef_balance}, the coeff for income is
{coef_income}")

# c (purpose: compare SE -- however, unable because Python)

def boot(data, func, R):
    total_coeff_balance = []
    total_coeff_income = []
    for i in range(R):
        bootstrap = resample(data, replace=True, n_samples=(0.3*default.size), random_state=i,
stratify = data['default'])
        params = func(data, bootstrap.index)
        total_coeff_balance.append(params[0])
        total_coeff_income.append(params[1])
    return (np.mean(total_coeff_balance), np.mean(total_coeff_income))

total_coeff_balance, total_coeff_income = boot(default, boot_fn, 1000)

print(f"The total coeff of balance is {total_coeff_balance} and the total coeff of income is
{total_coeff_income}")

# d

# Since this is a simplified version of the R boot function, I am unable
# to get the standard errors, however, I would expect them to be very
# similiar -- unlike the coefficient estimates/ means, which are at
# -6.995 for balance and at 0.004 for income and thus different from
# the coefficients in part a) which vary unless I set seed in boot_fn().
```

# 8 **p.200f.**

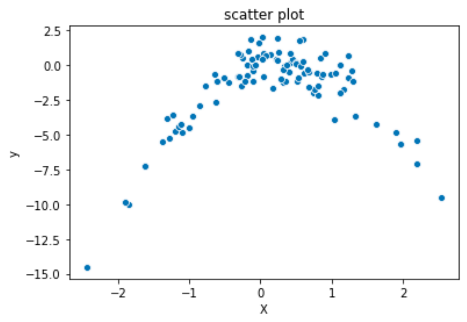
```
# a

np.random.seed(1)
y = np.random.normal(size = 100)
X = np.random.normal(size = 100)
y = X - 2 * (X ** 2) + np.random.normal(size = 100)

# n is 100 -> number of observations/ units
# p is 2 -> number of independent variables/ features (X, X^2)
# equation is y = X - 2X^2 + e // where e = error term

# b

sns.scatterplot(X, y)
plt.xlabel("X")
plt.ylabel("y")
plt.title("scatter plot")
```



```
# Comment on what you find: simulated resembles a quadratic function that
# ranges from -15 to 2.5 along the y-axis and -2 to 2 along the x-axis.

# c

np.random.seed(1)

for i in range (1, 5):
    poly = PolynomialFeatures(i, include_bias = False)
    predictors = poly.fit_transform(X.reshape(-1, 1))
    lr = LinearRegression()
    error = -1 * cross_val_score(lr, predictors, y, cv = len(X), scoring =
"neg_mean_squared_error").mean()
    print(f"For model {i}, error is {error}")

# d

np.random.seed(50374)
for i in range (1, 5):
    poly = PolynomialFeatures(i, include_bias = False)
    predictors = poly.fit_transform(X.reshape(-1, 1))
    lr = LinearRegression()
    error = -1 * cross_val_score(lr, predictors, y, cv = len(X), scoring =
"neg_mean_squared_error").mean()
    print(f"For model {i}, error is {error}")

# Our results are identical because in LOOCV, there is no random sampling,
# instead, it is being trained on the same n-1 observations/ folds (and
# then just tested on the observations that is being left-out).

# e

# Yes, I expected the second model to have the smallest LOOCV test error
# because of its underlying quadratic form, which we have seen in b (see
# scatterplot). This is what best fits the second model, which is
# the one with a quadratic form too: Y = β0 + β1X + β2X2 + e.

# f

for i in range (1, 5):
    poly = PolynomialFeatures(i)
    predictors = poly.fit_transform(X.reshape(-1, 1))
    results = sm.OLS(y, predictors).fit()
    print(results.summary())

# When I run the four model in a OLS regression, the coefficient estimates
# for x1 is statistically significant for the first/ linear model(0.003).
# Beyond that, the x1 and x2 estimates are even more highly significant
# for the second (quadratic) model, third, and fourth models (at a level
# of 0.000). Note how neither x3 in model 3 nor x3 and x4 are significant
# at conventional significance levels. This suggests that our results
# agree with our previous findings from CV where especially the quadratic
# function yielded the smallest LOOCV test error (see answer to part e).
```

Ch 6 LINEAR MODEL SELECTION & REGULARIZATION

# 11 **p.264**

```
# a

## Explore

boston = load_boston()

data = pd.DataFrame(boston.data, columns=boston.feature_names)
data.head()

predictors = data.drop("CRIM", axis = 1)
X = data.drop("CRIM", axis = 1)
y = data["CRIM"]
predictors.head()

for feature in predictors.columns:
    sns.scatterplot(predictors[feature], data["CRIM"])
    plt.title(feature)
    plt.show()

## Best subset selection

# Quite frankly, this does not add up for me. Why are we hand-selecting
# features for best subset selection when our outcome should be a subset
# of variables that best suit our model? Hmm. Instead, we calculate the
# error rate here for a model that we choose based on visual correlation?
# But how is that best subset selection as demonstrated in Friday's lab?

hand_selected_features = ["NOX", "DIS", "RAD", "LSTAT"]
# Based on explored/ visualized data correlation from scatterplots

results_dict = {}

lin_reg = LinearRegression()
error = cross_val_score(lin_reg, predictors[hand_selected_features], y, cv=5, scoring =
"neg_mean_squared_error")
print("Error for best subset selection is", -np.mean(error))
results_dict["Best_subset"] = -np.mean(error)

## Forward stepwise selection
## Option 1:

P = len(X.columns)
used_pred = []
M = []
M_scores = []

for K in range(P):
   best_score = -1000
   best_pred = None

   # Inner loop
   for var in X.columns:

      # Skips if predictor already used
      if var not in used_pred:
         predictors = used_pred[:]
         predictors.append(var)

         score = np.mean(cross_val_score(lin_reg, X[predictors], y, cv = 5, scoring = 
'neg_mean_squared_error'))
         if score > best_score:
            best_score = score
            best_pred = var

   # Updates the list of used predictors and list of Mk models
   used_pred.append(best_pred)
   M.append(used_pred[:])
   M_scores.append(best_score)

best_M = M_scores.index(max(M_scores))
print('Predictors that make the best model are: ', M[best_M])

## Option 2:

sfs1 = SequentialFeatureSelector(lin_reg,
          k_features="best",
          forward=True,
          scoring='neg_mean_squared_error',
          cv=5)

sfs1.fit(X, y)
sfs1.k_feature_names_

## Backwards stepwise selection
```

```
sfs1 = SequentialFeatureSelector(lin_reg,
        k_features="best",
        forward=False,
        scoring='neg_mean_squared_error',
        cv=5)

sfs1.fit(X, y)
sfs1.k_feature_names_

# In the forwards subset selection, we start out with no Xs and add
# those X that most fit and repeat the process until all Xs are in the
# model. In the backwards subset selection, we start with all Xs and
# then remove X's that least fit and repeat the process until there are
# no Xs left in the model. Doing that,
# 1. forwards selection yields: ['RAD', 'LSTAT', 'ZN'] -- same by the
# way for both of the options I explored, while
# 2. backward selection yields: ('ZN', 'NOX', 'DIS', 'RAD', 'LSTAT').
# Moving forward, I'll use only those three variables that appear to
# contribute to best model fit resulting from both methods, namely:
# ['RAD', 'LSTAT', 'ZN']

# Going back to best subset selection, as I said before, what we
# learned in lab on how to do it just does not add up for me (cf.
# above): why are we hand-selecting features when our outcome itself
# should be a subset of variables that render best fit?

# b

# Evaluate model performances for each of the Xs I got in a)
# using K-Fold-CV -> so that is for: ['RAD', 'LSTAT', 'ZN']

error_list = []
for power in range(1, 11):
   X = data['RAD']
   y = data["CRIM"]
   poly = PolynomialFeatures(power, include_bias = False)
   X = poly.fit_transform(X.to_frame())
   lr = LinearRegression()
   error_list.append(-1*cross_val_score(lr, X, y, cv=10,
scoring="neg_mean_squared_error").mean())
print("K Fold CV")
print('RAD')
mini = min(error_list)
print(f"min MSE is: {mini}")
```

```
pd.DataFrame({"Degree": np.arange(1,11), "CV Mean Squared Error": error_list})

error_list = []
for power in range(1, 11):
   X = data['LSTAT']
   y = data["CRIM"]
   poly = PolynomialFeatures(power, include_bias = False)
   X = poly.fit_transform(X.to_frame())
   lr = LinearRegression()
   error_list.append(-1*cross_val_score(lr, X, y, cv=10,
scoring="neg_mean_squared_error").mean())
print("K Fold CV")
print('LSTAT')
mini = min(error_list)
print(f"min MSE is: {mini}")
pd.DataFrame({"Degree": np.arange(1,11), "CV Mean Squared Error": error_list})

error_list = []
for power in range(1, 11):
   X = data['ZN']
   y = data["CRIM"]
   poly = PolynomialFeatures(power, include_bias = False)
   X = poly.fit_transform(X.to_frame())
   lr = LinearRegression()
   error_list.append(-1*cross_val_score(lr, X, y, cv=10,
scoring="neg_mean_squared_error").mean())
print("K Fold CV")
print('ZN')
mini = min(error_list)
print(f"min MSE is: {mini}")
pd.DataFrame({"Degree": np.arange(1,11), "CV Mean Squared Error": error_list})

# Judging by the lowest CV MSE which I printed out on top of each table
# for the variables I previously identified in part a), I would propose
# a model with 2 degrees because that yields the lowest *TOTAL* CV MSE.

# The reason is that even though ZN and RAD both suggest that 7 is the
# model resulting in the lowest CV MSE, we need to take the sum of total
# test errors into account should we do that. Note how for ZN and RAD
# the MSE rates are pretty much constant for all polynomial model degrees.
# They differ by less than one percentage point for the degree values we
# care about, therefore it doesn't matter to much for them what model
# degree we pick. This is very different for LSTAT. Here, the CV MSE
# differs notably across the degrees which is why I prioritize choosing
# its lowest CV MSE because that returns the lowest CV MSE for the model
# overall. Since it's lowest for 2 degrees for LSTAT, I shall use a
# quadratic model function.

# c

# The forwards and backward subset selection methods both return a subset
# of the variables which is what the name suggests too: # ['RAD', 'LSTAT',
# 'ZN']. As I already elaborated on in greater detail in part a):

# "In the forwards subset selection, we start out with no Xs and add
# those X that most fit and repeat the process until all Xs are in the
# model. In the backwards subset selection, we start with all Xs and
# then remove X's that least fit and repeat the process until there are
# no Xs left in the model. Doing that,
# 1. forwards selection yields: ['RAD', 'LSTAT', 'ZN'] -- same by the
# way for both of the options I explored, while
# 2. backward selection yields: ('ZN', 'NOX', 'DIS', 'RAD', 'LSTAT').
# Moving forward, I'll use only those three variables that appear to
# contribute to best model fit resulting from both methods, namely:
# ['RAD', 'LSTAT', 'ZN']""
```