Homework 4

PSTAT 131

Resampling

For this assignment, we will be working with **two** of our previous data sets - one for classification and one for regression. For the classification problem, our goal is (once again) to predict which passengers would survive the Titanic shipwreck. For the regression problem, our goal is (also once again) to predict abalone age.

Loead the data from data/titanic.csv and data/abalone.csv into Python and refresh your memory about the variables they contain using their attached codebooks.

Make sure to change survived and pclass to factors, as before, and make sure to generate the age variable as rings + 1.5!

Remember that you'll need to set a seed at the beginning of the document to repoduce your results.

In [3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.exceptions import ConvergenceWarning
import warnings
warnings.filterwarnings('ignore', category=ConvergenceWarning)
warnings.filterwarnings('ignore', category=UserWarning)
```

Section 1: Regression (abalone age)

Question 1

Follow the instructions from Homework 2 to split the data set, stratifying on the outcome variable, age. You can choose the proportions to split the data into. Use k-fold cross-validation to create 5 folds from the training set.

In [6]:
```
from sklearn.model_selection import train_test_split
```

In [7]:
```
#Load in the datasets
abalone_data = pd.read_csv("data/abalone.csv")

#Add age target variable
abalone_data["age"] = abalone_data["rings"] + 1.5

#Dropping 'rings' feature due to structural collinearity
abalone_data = abalone_data.drop("rings", axis = 1)
```

In [8]:
```
#Check appropriate changes were made
abalone_data.head(1)
```

Out[8]:

	type	longest_shell	diameter	height	whole_weight	shucked_weight	viscera_weight	shell_weight	age
0	M	0.455	0.365	0.095	0.514	0.2245	0.101	0.15	16.5

In [9]:
```
#Split features and target variable
X = abalone_data.drop("age", axis = 1)
y = abalone_data["age"]

#Create quantiles bins for stratified sampling
age_binned = pd.qcut(y, q = 4, labels = False)

#Split train/test data w/ stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, stratify = age_binned, random_state
```

In [10]:
```
#Dummy code any categorical predictors
coded_X_train = pd.get_dummies(X_train, columns = ['type'])
coded_X_train
```

Out[10]:

	longest_shell	diameter	height	whole_weight	shucked_weight	viscera_weight	shell_weight	type_F	type_I	type_M
3951	0.465	0.390	0.110	0.6355	0.1815	0.1570	0.2250	1	0	0
3423	0.630	0.475	0.150	1.1720	0.5360	0.2540	0.3160	1	0	0
3257	0.505	0.385	0.110	0.6550	0.3185	0.1500	0.1850	0	0	1
2830	0.525	0.430	0.135	0.8435	0.4325	0.1800	0.1815	1	0	0
182	0.560	0.450	0.160	1.0235	0.4290	0.2680	0.3000	1	0	0
...	...	...	...	...	...	...	...	...	...	...
651	0.335	0.245	0.090	0.1665	0.0595	0.0400	0.0600	0	1	0
2555	0.370	0.290	0.080	0.2545	0.1080	0.0565	0.0700	0	1	0
3727	0.505	0.400	0.150	0.7750	0.3445	0.1570	0.1850	0	0	1
805	0.405	0.305	0.120	0.3185	0.1235	0.0905	0.0950	0	0	1
3198	0.450	0.335	0.140	0.4780	0.1865	0.1150	0.1600	0	0	1

3341 rows × 10 columns

In [11]:
```
#Centering & Scaling all predictors
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
cols = coded_X_train.columns
X_train_scaled = sc.fit_transform(coded_X_train)
X_train_scaled = pd.DataFrame(X_train_scaled, columns = cols)
X_train_scaled
```

Out[11]:

	longest_shell	diameter	height	whole_weight	shucked_weight	viscera_weight	shell_weight	type_F	type_I	type_M
0	-0.485566	-0.176117	-0.686432	-0.387921	-0.792995	-0.208390	-0.097979	1.500000	-0.695866	-0.759889
1	0.883863	0.677664	0.249637	0.701865	0.800942	0.672978	0.550822	1.500000	-0.695866	-0.759889
2	-0.153583	-0.226339	-0.686432	-0.348311	-0.177002	-0.271994	-0.383166	-0.666667	-0.695866	1.315981
3	0.012408	0.225663	-0.101389	0.034587	0.335576	0.000594	-0.408120	1.500000	-0.695866	-0.759889
4	0.302893	0.426552	0.483654	0.400219	0.319839	0.800186	0.436747	1.500000	-0.695866	-0.759889
...	...	...	...	...	...	...	...	...	...	...
3336	-1.564510	-1.632567	-1.154467	-1.340596	-1.341543	-1.271484	-1.274376	-0.666667	1.437059	-0.759889
3337	-1.274025	-1.180565	-1.388484	-1.161842	-1.123472	-1.121560	-1.203079	-0.666667	1.437059	-0.759889
3338	-0.153583	-0.075672	0.249637	-0.104557	-0.060098	-0.208390	-0.383166	-0.666667	-0.695866	1.315981
3339	-0.983540	-1.029898	-0.452415	-1.031840	-1.053780	-0.812627	-1.024837	-0.666667	-0.695866	1.315981
3340	-0.610059	-0.728563	0.015620	-0.707849	-0.770513	-0.590013	-0.561408	-0.666667	-0.695866	1.315981

3341 rows × 10 columns

In [12]:
```
#Applying changes to testing set for later evaluation
coded_X_test = pd.get_dummies(X_test, columns = ['type'])

cols_x = coded_X_test.columns

X_test_scaled = sc.fit_transform(coded_X_test)
X_test_scaled = pd.DataFrame(X_test_scaled, columns = cols_x)
```

> We'll use GridSearchCV(cv = 5) during our model tuning / evaluation step to run k-fold cross validation instead of splitting it now (easier and cleaner code)

Question 2

In your own words, explain what we are doing when we perform k-fold cross-validation:

- What is k-fold cross-validation?
- Why should we use it, rather than simply comparing our model results on the entire training set?
- If we split the training set into two and used one of those two splits to evaluate/compare our models, what resampling method would we be using?

> K-fold cross validation provides the user with a way to validate/predict the approximate performance of their machine learning models by splitting the training data set further in "K" folds, fitting the model to one of those folds, and testing on the rest (k-1 folds). This process is then repeated k times. Performing k-fold cross validation helps ensure that the model can generalize well to new data, and prevents data leakage & overfitting (which would happen if we use the entire training set as a means of validation). Splitting the training set into two and using one of the two splits to evaluate/compare our models is called validation set sampling. While it is the simplest way to do resampling, it is less robust to overfitting and leads to less data to train your model.

Question 3

Set up the three models:

- K-nearest Neighbors with the KNeighborsRegressor function, tuning n_neighbors
- linear regression
- elastic net linear regression, tuning penalty and mixture

Use GridSearchCV to set up grids of values for all of the parameters we're tuning. Use values of neighbors from 1 to 10, the default values of penalty, and values of mixture from 0 to 1. Set up 10 levels of each.

How many models total, **across all folds**, will we be fitting ot the abalone data? To answer, think about how many folds there are, how many combinations of model parameters there are, and how many models you'll fit to each fold.

In [17]:
```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import GridSearchCV

#Define model & tuning grid
knn_model = KNeighborsRegressor()
param_grid_knn = {
    'n_neighbors': range(1,11)
}
#Evaluate w/ 5-fold CV
grid_search_mae_knn = GridSearchCV(knn_model, param_grid_knn, cv = 5, scoring = 'neg_root_mean_squared_error')
grid_search_mae_knn.fit(X_train_scaled, y_train);
```

In [18]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
#Define the model
lr_model = LinearRegression()

#Evaluate w/ 5-fold CV
neg_rmse_cv_scores_lr = cross_val_score(lr_model, X_train_scaled, y_train, cv = 5, scoring = 'neg_root_mean_squ

#Turn - into +
rmse_cv_scores_lr = -neg_rmse_cv_scores_lr
```

In [19]:
```
from sklearn.linear_model import ElasticNet
#Define the model & tuning grid
en_model = ElasticNet(random_state = 3)
param_grid_en = {
    'alpha': [0, 0.1, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1],    # Penalty strength
    'l1_ratio': [0, 0.1, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]  # Mixture of L1 (Lasso) and L2 (Ridge)
}
#Evaluate w/ 5-fold CV
grid_search_en = GridSearchCV(en_model, param_grid_en, cv=5, scoring='neg_root_mean_squared_error')
grid_search_en.fit(X_train_scaled, y_train);
```

Question 4 & Question 5

Fit all the models you created in Question 3 to your folded data. Print the mean and standard errors of the performance metric ***root mean squared error (RMSE)*** for each model across folds.

Decide which of the models has performed the best. Explain how/why you made this decision. Note that each value of the tuning parameter(s) is considered a different model; for instance, KNN with k = 4 is one model, KNN with K = 2 another.

In [21]:
```
#KNN Model
best_index_knn = grid_search_mae_knn.best_index_
best_mean_rmse_knn = -grid_search_mae_knn.cv_results_['mean_test_score'][best_index_knn]
best_std_rmse_knn = grid_search_mae_knn.cv_results_['std_test_score'][best_index_knn]
best_params_knn = grid_search_mae_knn.best_params_
print(f'Best KNN Model: n_neighbors = {best_params_knn["n_neighbors"]}')
print(f'Best Mean RMSE = {best_mean_rmse_knn:.4f}, Standard Deviation = {best_std_rmse_knn:.4f}')


#LR Model
print("\nLinear Regression Model Results:")
lr_rmse_mean = np.mean(rmse_cv_scores_lr)
lr_rmse_std = np.std(rmse_cv_scores_lr)
print(f'Linear Regression: Mean RMSE = {lr_rmse_mean}, Standard Deviation = {lr_rmse_std}')

#EN Model
best_index = grid_search_en.best_index_
best_score = -grid_search_en.best_score_
best_std = grid_search_en.cv_results_['std_test_score'][best_index]
best_params = grid_search_en.best_params_
print(f"Best RMSE: {best_score:}")
print(f"Standard Deviation of Best RMSE: {best_std:.4f}")
print(f"Best Hyperparameters: {best_params}")
```

```
Best KNN Model: n_neighbors = 10
Best Mean RMSE = 2.2799, Standard Deviation = 0.0818

Linear Regression Model Results:
Linear Regression: Mean RMSE = 2.2763147912114916, Standard Deviation = 0.12284541628270419
Best RMSE: 2.276314754048024
Standard Deviation of Best RMSE: 0.1228
Best Hyperparameters: {'alpha': 0, 'l1_ratio': 0}
```

> While all scores were relatively similar, it seems the Elastic Net model performed the best of all the models with a mean RMSE score of **2.27631475** - barely beating out the Linear Regression model.

Question 6

Fit your chosen model to the entire **training set**.

Lastly, access the performance of your chosen model on your testing set. Compare your model's **testing** RMSE to its average RMSE across folds.

In [24]:
```
from sklearn.metrics import mean_squared_error
#Fit & predict test set
best_model = ElasticNet(random_state = 3, alpha = 0, l1_ratio = 0)
best_model.fit(X_train_scaled, y_train)
y_pred = best_model.predict(X_test_scaled)

# Calculate the RMSE on the test set
test_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f'Test RMSE: {test_rmse}')
```

```
Test RMSE: 2.1092843449499514
```

> After fitting the model to the entire training set and evaluating its performance on the testing dataset, the tuned Elastic Net model did slightly better than its average RMSE score across folds. With a RMSE score of **~2.109**, the model does well in predicting abalone age given the available features.

Section 2: Classification (Titanic Survival)

In [27]:
```
import pandas as pd
import numpy as np
from collections import Counter
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=RuntimeWarning)
```

Question 7

Follow the instructions from Homework 3 to split the data set, stratifying on the outcome variable, survived. You can choose the proportions to split the data into. Use k-fold cross validation to create 5 folds from the training set.

In [29]:
```
#Load in data
titanic_data = pd.read_csv("data/titanic.csv")
X = titanic_data.drop("survived", axis = 1)
y = titanic_data["survived"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .2, random_state = 3, stratify = y)
```

In [30]:
```
from sklearn.impute import KNNImputer
#Prep Train/test data for imputation
train_data = pd.concat([X_train, y_train], axis = 1)
test_data = pd.concat([X_test, y_test], axis = 1)

#KKN imputer to handle missing age values
imputer = KNNImputer(n_neighbors=5)
train_data[['age']] = imputer.fit_transform(train_data[['age']])
test_data[['age']] = imputer.fit_transform(test_data[['age']])

#Grabbing appropriate features and dummy coding categorical variables
model_cols = ['survived','pclass', 'sex', 'age', 'sib_sp', 'parch', 'fare']
train_data = train_data[model_cols]
adj_train_data = pd.get_dummies(train_data, columns = ['pclass', 'sex'])
test_data = test_data[model_cols]
adj_test_data = pd.get_dummies(test_data, columns = ['pclass', 'sex'])
```

Question 8

Follow the instructions from Homework 3 - but this time, use SMOTE() to oversample such that there are equal proportions of the Yes and No levels.

In [32]:
```
#Separate features and target variable for the training set
X_train = adj_train_data.drop('survived', axis=1)
y_train = adj_train_data['survived']

#Separate features and target variable for the test set (not applying SMOTE to the test set)
X_test = adj_test_data.drop('survived', axis=1)
y_test = adj_test_data['survived']
```

In [33]:
```
from imblearn.over_sampling import SMOTE
#Apply SMOTE to the training data
smote = SMOTE(random_state=3)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

#Confirming oversample
counter = Counter(y_train_smote)
print(counter)
```

```
Counter({'Yes': 439, 'No': 439})
```

In [34]:
```
#Remapping survival 'Yes' indicates survival (1) and 'No' indicates non-survival (0)
y_train_smote = y_train_smote.map({'Yes': 1, 'No': 0})
y_test = y_test.map({'Yes': 1, 'No': 0})
```

Question 9 & 10

Set up the three models:

- K-Nearest neighbors with the KNeighborsRegressor function, tuning n_neighbors
- logistic regression
- elastic net **logistic** regression, tuning penalty and mixture

Set up the grids, etc. the same way you did in Question 3. Note that you can use the same grids of parameter values without having to recreate them. Fit all the models you created in Question 9 to your folded data.

In [36]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

#Define model & tuning grid
knn_model = KNeighborsClassifier()
param_grid_knn = {
    'n_neighbors': range(1,11)
}
#Evaluate w/ 5-fold CV
grid_search_mae_knn = GridSearchCV(knn_model, param_grid_knn, cv = 5, scoring = 'roc_auc')
grid_search_mae_knn.fit(X_train_smote, y_train_smote);
```

In [37]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

#Define logistic regression model
logreg_model = LogisticRegression(random_state=3)

#Evaluate with 5-fold cross-validation using ROC AUC as the scoring metric
auc_cv_scores_logreg = cross_val_score(logreg_model, X_train_smote, y_train_smote, cv=5, scoring='roc_auc')
```

In [38]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

en_logreg_model = LogisticRegression(penalty='elasticnet', solver='saga', random_state=3)
param_grid_en = {
    'C': [0, 0.1, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1],
    'l1_ratio': [0, 0.1, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]  #Mixture of L1 (Lasso) and L2 (Ridge) where l1
}

grid_search_en_logreg = GridSearchCV(en_logreg_model, param_grid_en, cv=5, scoring = 'roc_auc')
grid_search_en_logreg.fit(X_train_smote, y_train_smote);
```

Question 11

Print the means and standard errors of the performance metric: area under the ROC curve for each model across folds.

Decide which of the models has performed the best. Explain how/why you made this decision.

In [40]:
```
#KNeighbors Classifier Model
best_index_knn = grid_search_mae_knn.best_index_
best_mean_auc_knn = grid_search_mae_knn.cv_results_['mean_test_score'][best_index_knn]
best_std_auc_knn = grid_search_mae_knn.cv_results_['std_test_score'][best_index_knn]
best_params_knn = grid_search_mae_knn.best_params_
print(f'Best KNN Model: n_neighbors = {best_params_knn["n_neighbors"]}')
print(f'Best Mean ROC AUC = {best_mean_auc_knn:.4f}, Standard Deviation = {best_std_auc_knn:.4f}')

#Logistic Regression model
mean_auc_logreg = np.mean(auc_cv_scores_logreg)
std_auc_logreg = np.std(auc_cv_scores_logreg)
print(f'Logistic Regression: Mean ROC AUC = {mean_auc_logreg:.4f}, Standard Deviation = {std_auc_logreg:.4f}')

#Elastic Net Logistic Regression model
best_index_en_logreg = grid_search_en_logreg.best_index_
best_mean_auc_en_logreg = grid_search_en_logreg.cv_results_['mean_test_score'][best_index_en_logreg]
best_std_auc_en_logreg = grid_search_en_logreg.cv_results_['std_test_score'][best_index_en_logreg]
best_params_en_logreg = grid_search_en_logreg.best_params_
print(f'Best ElasticNet Logistic Regression Model: C = {best_params_en_logreg["C"]}, l1_ratio = {best_params_en
print(f'Best Mean ROC AUC = {best_mean_auc_en_logreg:.4f}, Standard Deviation = {best_std_auc_en_logreg:.4f}')
```

```
Best KNN Model: n_neighbors = 4
Best Mean ROC AUC = 0.8127, Standard Deviation = 0.0476
Logistic Regression: Mean ROC AUC = 0.8672, Standard Deviation = 0.0541
Best ElasticNet Logistic Regression Model: C = 0.3, l1_ratio = 0
Best Mean ROC AUC = 0.7492, Standard Deviation = 0.0260
```

> After evaluating the models based on the mean ROC AUC score and its standard deviation, the best performing model was the Logistic Regression model with a mean ROC AUC score of .8672 and sd of .0541. This means that the model distinguishes between the positive and negative classes with a relatively high degree of accuracy and consistency. While it does not have the lowest ROC AUC std, it's difference is negligible in the context of the problem and should still generalize well to new data.

Question 12

Fit your chosen model to the entire **training set.** Assess the performance of your chosen model on your **testing set.** Compare your model's **testing** ROC AUC to its average ROC AUC across folds.

In [43]:
```
from sklearn.metrics import roc_auc_score

best_model_classifier = LogisticRegression(random_state=3)
best_model_classifier.fit(X_train_smote, y_train_smote)
best_pred_proba = best_model_classifier.predict_proba(X_test)[:,1]
roc_auc_test = roc_auc_score(y_test, best_pred_proba)
print(f'ROC AUC on the Test set: {roc_auc_test:f}')
```

```
ROC AUC on the Test set: 0.900000
```

> The Logistic Regression model performed well on the test set with a ROC AUC score of 0.9000, which is slightly higher than its average cross-validated performance of 0.8672. The consistency in performance between cross-validation and the test set indicates that the model generalizes well to unseen data and has not overfit the training data.