In [ ]:
```
from sklearn.datasets import fetch_20newsgroups_vectorized
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
import numpy as np
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.multiclass import OneVsOneClassifier
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsOneClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.linear_model import Lars
import numpy as np
import itertools
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import pandas as pd
from sklearn import svm, datasets
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
import warnings
warnings.filterwarnings("ignore")
import pprint
import csv
import codecs
import urllib.request
from collections import Counter
import glob
import codecs
import re
import pandas as pd
import math
from cmath import exp
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
from sklearn import datasets
import statsmodels.api as sm
from pylab import rcParams
from matplotlib.pyplot import *
from numpy.linalg import inv
from statsmodels.tsa.seasonal import seasonal_decompose
# sns.reset_orig()
np.random.seed(0)
sns.set_theme(context='notebook')
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
from sklearn import linear_model
from sklearn.metrics import mean_squared_error
import numpy as np
np.random.seed(0)
from sklearn.base import BaseEstimator, RegressorMixin
from sklearn.linear_model import LinearRegression
```

```
from numpy import isnan, isinf, float64
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
from sklearn.utils.estimator_checks import *
import warnings
from sklearn.utils.validation import DataConversionWarning
warnings.warn("Test DataConversionWarning", DataConversionWarning)
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.datasets import fetch_20newsgroups
from sklearn.decomposition import TruncatedSVD
```

In the previous exercise sheets, you have implemented various algorithms from scratch; this exercise focuses on introducing you to a high-level Machine Learning library (sklearn).

1. Load the 20news-group-vectorized dataset from sklearn. This dataset consists of 130107 predictors with 20 classes. Perform the following experiments:

In [ ]:

```
# # Load the training dataset
newsgroups_train = fetch_20newsgroups_vectorized(subset='train', return_X_y=True)
newsgroups_test = fetch_20newsgroups_vectorized(subset='test', return_X_y=True)
xtrain, ytrain = newsgroups_train
xtest, ytest = newsgroups_test
```

normalize

a) Multiclass Classification

In [ ]:

```
clf = MultinomialNB()
clf.fit(xtrain, ytrain)
pred = clf.predict(xtest)
print('Classification Accuracy %:', accuracy_score(ytest, pred)*100)
print('Classification Report: ', classification_report(ytest, pred))
```

```
Classification Accuracy %: 70.52575677110993
Classification Report:                precision    recall  f1-score   support

           0       0.85      0.24      0.37       319
           1       0.71      0.60      0.65       389
           2       0.79      0.65      0.71       394
           3       0.63      0.75      0.69       392
           4       0.86      0.68      0.76       385
           5       0.88      0.68      0.77       395
           6       0.90      0.72      0.80       390
           7       0.71      0.92      0.80       396
           8       0.84      0.91      0.87       398
           9       0.86      0.85      0.86       397
          10       0.90      0.93      0.91       399
          11       0.52      0.96      0.67       396
          12       0.78      0.52      0.63       393
          13       0.82      0.76      0.79       396
          14       0.83      0.81      0.82       394
          15       0.34      0.98      0.51       398
          16       0.66      0.80      0.73       364
          17       0.96      0.72      0.82       376
          18       1.00      0.17      0.29       310
          19       1.00      0.01      0.02       251

    accuracy                           0.71      7532
   macro avg       0.79      0.68      0.67      7532
weighted avg       0.79      0.71      0.69      7532
```

b) Use Logistic Regression Algorithm and perform

i. One vs Rest

In [ ]:

```
clf = OneVsRestClassifier(LogisticRegression())
clf.fit(xtrain, ytrain)
pred = clf.predict(xtest)
print('Classification Accuracy %:', accuracy_score(ytest, pred)*100)
print('Classification Report: ', classification_report(ytest, pred))
```

```
Classification Accuracy %: 72.65002655337229
Classification Report:                precision    recall  f1-score   support

           0       0.64      0.61      0.63       319
           1       0.63      0.68      0.65       389
           2       0.72      0.66      0.69       394
           3       0.68      0.62      0.65       392
           4       0.71      0.70      0.71       385
           5       0.72      0.69      0.70       395
           6       0.72      0.86      0.79       390
           7       0.80      0.78      0.79       396
           8       0.79      0.88      0.83       398
           9       0.69      0.82      0.75       397
          10       0.86      0.86      0.86       399
          11       0.86      0.81      0.84       396
          12       0.63      0.62      0.62       393
          13       0.65      0.63      0.64       396
          14       0.85      0.85      0.85       394
          15       0.70      0.89      0.78       398
          16       0.62      0.80      0.69       364
          17       0.85      0.79      0.82       376
          18       0.68      0.46      0.55       310
          19       0.72      0.27      0.40       251

    accuracy                           0.73      7532
   macro avg       0.73      0.71      0.71      7532
weighted avg       0.73      0.73      0.72      7532
```

ii. One vs One

In [ ]:

```
clf = OneVsOneClassifier(LogisticRegression())
clf.fit(xtrain, ytrain)
pred = clf.predict(xtest)
print('Classification Accuracy %:', accuracy_score(ytest, pred))
print('Classification Report: ', classification_report(ytest, pred))
```

```
Classification Accuracy %: 0.6460435475305364
Classification Report:                precision    recall  f1-score   support

           0       0.54      0.50      0.52       319
           1       0.52      0.64      0.57       389
           2       0.70      0.57      0.63       394
           3       0.62      0.56      0.59       392
           4       0.67      0.61      0.64       385
           5       0.67      0.64      0.65       395
           6       0.66      0.85      0.74       390
           7       0.69      0.69      0.69       396
           8       0.69      0.78      0.73       398
           9       0.58      0.68      0.63       397
          10       0.85      0.77      0.81       399
          11       0.86      0.70      0.77       396
          12       0.52      0.54      0.53       393
          13       0.48      0.55      0.51       396
          14       0.84      0.75      0.79       394
          15       0.62      0.78      0.69       398
          16       0.54      0.73      0.62       364
          17       0.80      0.72      0.76       376
          18       0.57      0.40      0.47       310
          19       0.66      0.21      0.32       251

    accuracy                           0.65      7532
   macro avg       0.65      0.63      0.63      7532
weighted avg       0.66      0.65      0.64      7532
```

c) Use Linear Discriminant Analysis Algorithm and perform

i. one vs. rest

LDA was running indefinitely till kernel crashes, which I beleive could have been a result of overload on RAM due to extremely high number of computations. Hence, I reduced the dataset dimensions, using TrauncatedSVD. TruncatedSVD is a dimensionality reduction method that can be used to reduce the number of features in a dataset while preserving as much information as possible. It is a variant of Singular Value Decomposition (SVD), which is a method for decomposing a matrix into its singular vectors and singular values. TruncatedSVD works by first computing the SVD of the input matrix, and then keeping only the top k singular vectors and corresponding singular values, where k is a user-specified parameter. This results in a reduced matrix with fewer columns (features) but which still contains most of the information from the original matrix.

In [ ]:
```
# Create a TruncatedSVD instance with the desired number of dimensions (k)
svd = TruncatedSVD(n_components=1000)
# Use the fit_transform method to reduce the training set to k dimensions
xtrain_reduced = svd.fit_transform(xtrain)
xtest_reduced = svd.transform(xtest)
```

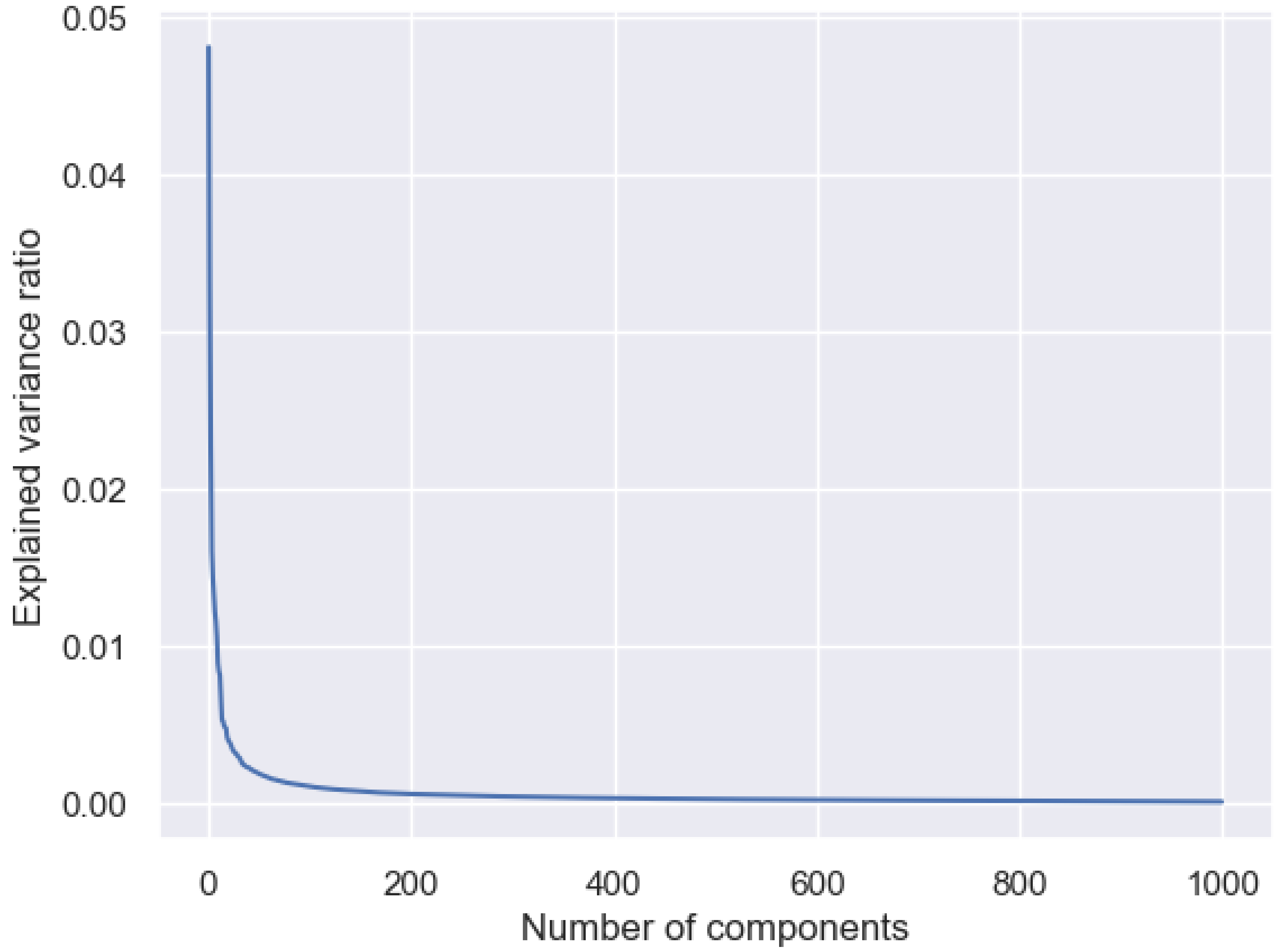
In [ ]:
```
# from sklearn.datasets import fetch_20newsgroups
# from sklearn.decomposition import TruncatedSVD
# import matplotlib.pyplot as plt

# # Load the 20 newsgroups dataset
# data = fetch_20newsgroups()

# # Apply TruncatedSVD to the data
# svd = TruncatedSVD(n_components=100)
# X_reduced = svd.fit_transform(data.data)

# Compute the explained variance ratio for each component
evr = svd.explained_variance_ratio_

# Plot the EVR as a function of the number of components
plt.plot(evr)
plt.xlabel('Number of components')
plt.ylabel('Explained variance ratio')
plt.show()
```



In [ ]:
```
# Create a LinearDiscriminantAnalysis object
lda = OneVsRestClassifier(LinearDiscriminantAnalysis(solver='lsqr', shrinkage='auto',
# Fit the model to the data
lda.fit(xtrain_reduced, ytrain)
# Use the trained model to predict the class of new data
pred = lda.predict(xtest_reduced)
print('Classification Accuracy %:', accuracy_score(ytest, pred))
print('Classification Report: ', classification_report(ytest, pred))
```

```
Classification Accuracy %: 0.7818640467339352
Classification Report:                precision    recall  f1-score   support

           0       0.71      0.63      0.66       319
           1       0.64      0.74      0.68       389
           2       0.71      0.70      0.70       394
           3       0.60      0.71      0.65       392
           4       0.78      0.76      0.77       385
           5       0.84      0.69      0.75       395
           6       0.83      0.84      0.83       390
           7       0.87      0.84      0.86       396
           8       0.97      0.89      0.93       398
           9       0.92      0.88      0.90       397
          10       0.94      0.91      0.93       399
          11       0.97      0.85      0.90       396
          12       0.60      0.77      0.68       393
          13       0.85      0.80      0.82       396
          14       0.93      0.86      0.89       394
          15       0.73      0.90      0.80       398
          16       0.71      0.86      0.78       364
          17       0.97      0.73      0.83       376
          18       0.65      0.59      0.62       310
          19       0.53      0.52      0.52       251

    accuracy                           0.78      7532
   macro avg       0.79      0.77      0.78      7532
weighted avg       0.79      0.78      0.78      7532
```

ii. One vs One

In [ ]:

```
clf = OneVsOneClassifier(LinearDiscriminantAnalysis(solver='lsqr', shrinkage='auto',
clf.fit(xtrain_reduced, ytrain)
pred = clf.predict(xtest_reduced)
print('Classification Accuracy %:', accuracy_score(ytest, pred))
print('Classification Report: ', classification_report(ytest, pred))
```

```
Classification Accuracy %: 0.7668613913967074
Classification Report:                precision    recall  f1-score   support

           0       0.70      0.69      0.69       319
           1       0.63      0.69      0.66       389
           2       0.72      0.68      0.70       394
           3       0.62      0.66      0.64       392
           4       0.75      0.75      0.75       385
           5       0.77      0.70      0.73       395
           6       0.82      0.86      0.84       390
           7       0.83      0.83      0.83       396
           8       0.94      0.89      0.91       398
           9       0.90      0.85      0.87       397
          10       0.93      0.90      0.92       399
          11       0.90      0.84      0.87       396
          12       0.65      0.71      0.67       393
          13       0.74      0.72      0.73       396
          14       0.85      0.82      0.84       394
          15       0.76      0.88      0.82       398
          16       0.70      0.84      0.76       364
          17       0.90      0.79      0.84       376
          18       0.63      0.56      0.59       310
          19       0.55      0.51      0.52       251

    accuracy                           0.77      7532
   macro avg       0.76      0.76      0.76      7532
weighted avg       0.77      0.77      0.77      7532
```

2. Variable Selection via Forward and Backward Search

Load the dataset regression.npy, the dataset consists of over 100 predictors. We generated the regression dataset such that only a few predictors are relevant. Perform the following experiments using the least angle regression algorithm

1. Forward Search

In [ ]:

```
with open('regression.npy', 'rb') as f:
    X = np.load(f)
    y = np.load(f)
import warnings
warnings.filterwarnings("ignore")

from sklearn.pipeline import make_pipeline
from sklearn import preprocessing
from sklearn.feature_selection import SequentialFeatureSelector
n, m = X.shape
split = int(0.8 * n)
p = np.random.permutation(n)
ones = np.ones(shape=X.shape[0]).reshape(-1, 1)
x_data = preprocessing.normalize(X) # NORMALIZING HERE!!!!!!!!!!
x_data = np.concatenate((ones,x_data), 1)
```

```
x_train1 = x_data[p[:split]]
y_train1 = y[p[:split]]
x_valid1 = x_data[p[split:]]
y_valid1 = y[p[split:]]
```

Forward search with SKLEARN

In [ ]:
```
model = linear_model.Lars().fit(x_train1, y_train1)
sfs = SequentialFeatureSelector(model, n_features_to_select= 50, direction="forward")
features_boolean = sfs.get_support().tolist()

feature_list = []
for i in range(len(features_boolean)):
    if features_boolean[i]:
        feature_list.append(i)

print("Features obtained:", feature_list)
lars_new_model = Lars()
lars_new_model.fit(x_train1[:,feature_list],y_train1)
predictions = lars_new_model.predict(x_valid1[:,feature_list])
mseScore = mean_squared_error(y_valid1,predictions)
print("MSE from Forward Search:\n",mseScore)
```

```
Features obtained: [0, 1, 2, 3, 4, 6, 7, 12, 14, 17, 21, 22, 25, 26, 27, 29, 30, 31,
32, 33, 34, 35, 36, 40, 41, 42, 46, 47, 49, 50, 51, 54, 55, 60, 61, 62, 65, 69, 72, 7
4, 76, 78, 79, 80, 81, 84, 86, 90, 96, 97]
MSE from Forward Search:
 8.644464865020298
```

In [ ]:
```
def performance(x_train, y_train, x_valid, y_valid):
    model = linear_model.Lars().fit(x_train, y_train)
    coef = model.coef_
    predictions= model.predict(x_valid)
    error_computed = mean_squared_error(y_valid, predictions)
    return error_computed, coef, model
```

Custom Implementation

In [ ]:
```
def forward_search(x_train, y_train, x_valid, y_valid):
    M = x_train.shape[1]; totalFeatures = set(range(M)); features = set(); initial_er
    while best_feature is not None:
        best_feature = None; best_error = initial_error
        for f in totalFeatures - features:
            add_new_feature = list(features | {f})
            error_computed, coef, _ = performance(x_train[:, add_new_feature], y_trai
            if error_computed < best_error:
                best_feature = f
                best_error = error_computed
        if best_error < initial_error:
            features = features | {best_feature}
            initial_error = best_error
    return features, best_error, coef
```

In [ ]:
```
featuresForward, mseForward, fwdCoef = forward_search(x_train1 , y_train1 , x_valid1
print('forward search features: \n', featuresForward)
print('forward best MSE: \n', mseForward)
```

```
forward search features:
 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 17, 18, 19, 20, 21, 22, 24, 25, 26, 27,
29, 31, 32, 34, 36, 39, 41, 43, 44, 45, 47, 49, 51, 52, 53, 54, 55, 56, 58, 61, 62, 6
3, 64, 65, 66, 69, 74, 75, 76, 77, 78, 80, 81, 82, 84, 86, 88, 89, 90, 93, 97, 100}
forward best MSE:
 6.261936532911846
```

2. Backward Search

Backward Search with SKLEARN

In [ ]:
```
model = linear_model.Lars().fit(x_train1, y_train1)
sfs = SequentialFeatureSelector(model, n_features_to_select= 50, direction="backward"
features_boolean = sfs.get_support().tolist()

feature_list = []
for i in range(len(features_boolean)):
    if features_boolean[i]:
        feature_list.append(i)

print("Features obtained:", feature_list)
lars_new_model = Lars()
lars_new_model.fit(x_train1[:,feature_list],y_train1)
predictions = lars_new_model.predict(x_valid1[:,feature_list])
mseScore = mean_squared_error(y_valid1,predictions)
print("MSE from Backward Search:\n ",mseScore)
```

```
Features obtained: [0, 1, 2, 3, 4, 6, 7, 12, 14, 17, 21, 22, 25, 26, 27, 29, 30, 31,
32, 33, 34, 35, 36, 40, 41, 42, 46, 47, 49, 50, 51, 54, 55, 60, 61, 62, 65, 69, 72, 7
4, 76, 78, 79, 80, 81, 84, 86, 90, 96, 97]
MSE from Backward Search:
  8.644464865020298
```

Custom Implementation

In [ ]:
```
def backward_search(x_train, y_train, x_valid, y_valid):
    M = x_train.shape[1]
    features = set(range(M))
    best_feature = 1
    error = np.inf
    while best_feature is not None:
        best_feature = None
        best_error = error
        for f in features:
            add_new_feature = list(features-{f})
            error_computed, coef, _ = performance(x_train[:, :len(add_new_feature)],
            if error_computed < best_error:
                best_feature = f
                best_error = error_computed
        if best_error < error:
            features = features - {best_feature}
            error = best_error
    return features, best_error, coef
```

In [ ]:
```
featuresBackward, mseBackward, bckCoef = backward_search(x_train1 , y_train1 , x_vali
print('backward search features: \n', featuresBackward)
print('backward best MSE: \n', mseBackward)
```

```
backward search features:
 {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 4
6, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 6
7, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 8
8, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
backward best MSE:
 6.340086059859352
```

Print out the indices of the selected features, compare the outputs of the two methods. Are the indices the same?

The features from ForwardSearch and BackwardSearch are not the same

In [ ]:
```
print('forward search features: \n', featuresForward)
print('backward search features: \n', featuresBackward)
```

```
forward search features: 
 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 17, 18, 19, 20, 21, 22, 24, 25, 26, 27, 29, 31, 32, 34, 36, 39, 41, 43, 44, 45, 47, 49, 51, 52, 53, 54, 55, 56, 58, 61, 62, 6
3, 64, 65, 66, 69, 74, 75, 76, 77, 78, 80, 81, 82, 84, 86, 88, 89, 90, 93, 97, 100}
backward search features: 
 {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 4
6, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 6
7, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 8
8, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

3 Regularization

Variable selection via forward and backward search drops some predictors; in some cases, we don't want to remove these predictors. Rather we want their coefficients to be small as possible. We are going to test the effect of the regularization term alpha. Try the following alpha values: [10, 1, 0.1, 0.0001, 0.00001], use regression.npy dataset

In [ ]:
```
mseList = []
alphaList = []
```

1. GridSearch

a) Ridge regression

In [ ]:
```
model = Ridge()
param_grid = {'alpha': [10, 1, 0.1, 0.0001, 0.00001 ]}
grid_search = GridSearchCV(model, param_grid, cv=5)
grid_search.fit(x_train1, y_train1)
print(grid_search.best_params_)
optimumAlpha = grid_search.best_params_['alpha']

model = Ridge(alpha = optimumAlpha)
model.fit(x_train1, y_train1)
gRidgeCoef = model.coef_
# print('coeffs are: ', model.coef_)

pred = model.predict(x_valid1)
score = mean_squared_error(y_valid1, pred)
print('Mean squared error is: ', score)
# score = model.score(x_valid1, y_valid1)
# print('R2 error is: ', score)
```

```
mseList.append(score)
alphaList.append(optimumAlpha)
```

```
{'alpha': 0.0001}
Mean squared error is:  6.285025173533575
```

b) Lasso

In [ ]:

```
model = Lasso()
param_grid = {'alpha': [10, 1, 0.1, 0.0001, 0.00001 ]}
grid_search = GridSearchCV(model, param_grid, cv=5)
grid_search.fit(x_train1, y_train1)
print(grid_search.best_params_)
optimumAlpha = grid_search.best_params_['alpha']

model = Lasso(alpha = optimumAlpha)
model.fit(x_train1, y_train1)
gLassoCoef = model.coef_
# print('coeffs are: ', model.coef_)

pred = model.predict(x_valid1)
score = mean_squared_error(y_valid1, pred)
print('Mean squared error is: ', score)
# score = model.score(x_valid1, y_valid1)
# print('R2 error is: ', score)
mseList.append(score)
alphaList.append(optimumAlpha)
```

```
{'alpha': 1e-05}
Mean squared error is:  6.285763008577142
```

c) Elastic-Net

In [ ]:

```
model = ElasticNet()
param_grid = {'alpha': [10, 1, 0.1, 0.0001, 0.00001 ]}
grid_search = GridSearchCV(model, param_grid, cv=5)
grid_search.fit(x_train1, y_train1)
print(grid_search.best_params_)
optimumAlpha = grid_search.best_params_['alpha']

model = ElasticNet(alpha = optimumAlpha)
model.fit(x_train1, y_train1)
gElasticCoef = model.coef_
pred = model.predict(x_valid1)
score = mean_squared_error(y_valid1, pred)
print('Mean squared error is: ', score)
# score = model.score(x_valid1, y_valid1)
# print('R2 error is: ', score)
mseList.append(score)
alphaList.append(optimumAlpha)
```

```
{'alpha': 1e-05}
Mean squared error is:  6.285619301264818
```

2. RandomSearch

a) Ridge regression

In [ ]:

```
model = Ridge()
```

```
random_search = RandomizedSearchCV(model, param_grid, cv=5)
random_search.fit(x_train1, y_train1)
print(random_search.best_params_)
optimumAlpha = random_search.best_params_['alpha']

model = Ridge(alpha = optimumAlpha)
model.fit(x_train1, y_train1)
pred = model.predict(x_valid1)
score = mean_squared_error(y_valid1, pred)
print('Mean squared error is: ', score)
# score = model.score(x_valid1, y_valid1)
# print('R2 error is: ', score)
mseList.append(score)
alphaList.append(optimumAlpha)
```

```
{'alpha': 0.0001}
Mean squared error is:  6.285025173533575
```

b) Lasso

In [ ]:

```
model = Lasso()
random_search = RandomizedSearchCV(model, param_grid, cv=5)
random_search.fit(x_train1, y_train1)
print(random_search.best_params_)
optimumAlpha = random_search.best_params_['alpha']

model = Lasso(alpha = optimumAlpha)
model.fit(x_train1, y_train1)
pred = model.predict(x_valid1)
score = mean_squared_error(y_valid1, pred)
print('Mean squared error is: ', score)
# score = model.score(x_valid1, y_valid1)
# print('R2 error is: ', score)
mseList.append(score)
alphaList.append(optimumAlpha)
```

```
{'alpha': 1e-05}
Mean squared error is:  6.285763008577142
```

c) Elastic-Net

In [ ]:

```
model = ElasticNet()
random_search = RandomizedSearchCV(model, param_grid, cv=5)
random_search.fit(x_train1, y_train1)
print(random_search.best_params_)
optimumAlpha = random_search.best_params_['alpha']

model = ElasticNet(alpha = optimumAlpha)
model.fit(x_train1, y_train1)
pred = model.predict(x_valid1)
# print('coeffs are: ', )
score = mean_squared_error(y_valid1, pred)
print('Mean squared error is: ', score)
# score = model.score(x_valid1, y_valid1)
# print('R2 error is: ', score)
mseList.append(score)
alphaList.append(optimumAlpha)
```

```
{'alpha': 1e-05}
Mean squared error is:  6.285619301264818
```

In [ ]:

```
for i in list(zip(mseList, alphaList)):
    print(i)
```

```
(6.285025173533575, 0.0001)
(6.285763008577142, 1e-05)
(6.285619301264818, 1e-05)
(6.285025173533575, 0.0001)
(6.285763008577142, 1e-05)
(6.285619301264818, 1e-05)
```

3. Briefly discuss the effect of high and low values of alpha. Then, return the best three models with their respective alpha values using the appropriate metric

High values of alpha will result in more regularization, which can help prevent overfitting and improve the stability and interpretability of the model. However, if the value of alpha is too high, it can also cause underfitting by reducing the model's flexibility and ability to fit the data. On the other hand, low values of alpha will result in less regularization, allowing the model to have more flexibility and potentially fit the data better. However, if the value of alpha is too low, the model may overfit the data and produce unstable or unreliable results.

4. Compare the best three models to the models from question 2. What do you observe?

Since the results are the same for Gridsearchcv and Randomsearch, which makes sense as these algorithms find the optimum value for alpha, which lead to the minimization of the cost. I will pick the first three alpha parameters.

a) Ridge regression b) Lasso c) Elastic-Net

In [ ]:
```
mseBar = mseList[:3]
mseBar.append(mseForward)
mseBar.append(mseBackward)
```

In [ ]:
```
print('Ridge, Lasso, ElasticNet, ForwardSearch, FackwardSearch\n', mseBar)
```

```
Ridge, Lasso, ElasticNet, ForwardSearch, FackwardSearch
 [6.285025173533575, 6.285763008577142, 6.285619301264818, 6.261936532911846, 6.34008
6059859352]
```

In [ ]:
```
mseBar.index(min(mseBar))
print('Minimum MSE is from Forward Search: ', mseBar[3])
```

```
Minimum MSE is from Forward Search:  6.261936532911846
```

As per the results, the best performing model in this case if Forward Search which gives the best results, which could be due to the fact that the weights might have diminished but not gone to 0 entirely which might be reflected as this slight increase in the error as compared to Forward Search which completely removes the columns that do not positively affect the minimization of the error. However, this is not a global case, and usually this leads to poor solutions and curating a list of features which are not the best, however in this case it performed good.

5. Get the indices of the top k coefficient of these three models and compare them with the features selected via the forward and back search method. Feel free to try different values of k

From the outputs below, we can observer for all the models all the dominant features are the same at the positions 0 1 2 3 4 5

In [ ]:
```
def max_k(arr, k):
    # Compute the indices of the k largest values in the array
    indices = np.argpartition(arr, -k)[-k:]
    # Extract the maximum k values and their positions from the array
    max_k_values = arr[indices]
    max_k_positions = np.argsort(arr[indices])[::-1]
    # Sort the array of maximum k values according to the positions of the maximum k
    sorted_max_k_values = max_k_values[np.argsort(max_k_positions)]
    return sorted_max_k_values, max_k_positions
```

In [ ]:
```
# gRidgeCoef[1:10]
```

In [ ]:
```
# gLassoCoef[1:10]
```

In [ ]:
```
# gElasticCoef[1:10]
```

In [ ]:
```
# fwdCoef[1:10]
```

In [ ]:
```
# bckCoef[1:10]
```

In [ ]:
```
k = 5
```

In [ ]:
```
vals, coefs = max_k(gRidgeCoef, k)
print('coefficient values: {} \n positions: {}'.format(vals, coefs))
```

```
coefficient values: [61.1434036  41.48622485 40.75673856 32.34406106  4.44094882]
 positions: [4 3 2 1 0]
```

In [ ]:
```
vals, coefs = max_k(gLassoCoef, k)
print('coefficient values: {} \n positions: {}'.format(vals, coefs))
```

```
coefficient values: [61.10246672 41.44295062 40.71421231 32.30407641  4.39925742]
 positions: [4 3 2 1 0]
```

In [ ]:
```
vals, coefs = max_k(gElasticCoef, k)
print('coefficient values: {} \n positions: {}'.format(vals, coefs))
```

```
coefficient values: [40.57667876 41.29854761  4.33418192 32.1861859  60.92444053]
 positions: [2 3 4 1 0]
```

In [ ]:
```
vals, coefs = max_k(fwdCoef, k)
print('coefficient values: {} \n positions: {}'.format(vals, coefs))
```

```
coefficient values: [39.5918596  38.89117224  2.14385536 59.31896068 30.50583422]
 positions: [2 4 3 1 0]
```

In [ ]:
```
vals, coefs = max_k(bckCoef, k)
print('coefficient values: {} \n positions: {}'.format(vals, coefs))
```

```
coefficient values: [41.92919566 41.156475    4.94033556 61.67931821 32.89247886]
 positions: [2 4 3 1 0]
```

Sklearn is a very powerful high-level Machine learning API. It has a clean implementation of almost all the popular machine-learning algorithms. This task will introduce you to how to write a custom estimator in sklearn.

1. Create a python class called MyLinearRegression
2. Ensure your class inherit from sklearn BaseEstimator and RegressorMixin
3. Implement fit(X,Y) method, and returns self
4. Implement predict(X) method
5. Use check estimator() method to know if your estimator(MyLinearRegression) is valid
6. Fit the dataset below using your custom estimator. Remember 80:20 split

In [ ]:
```
class MyLinearRegression(BaseEstimator, RegressorMixin):
    def fit(self, X, y):
        X, y = check_X_y(X, y)
        reg = LinearRegression()

        reg.fit(X, y)
        # self.set_params = reg.get_params
        self.coef_ = reg.coef_
        self.intercept_ = reg.intercept_
        return self

    def predict(self, X):
        check_is_fitted(self)
        X = check_array(X)
        try:
            y_pred = X @ self.coef_
        except:
            print('x shape is:', X.shape)
            cof = np.asanyarray(self.coef_)
            print('cof shape: ', cof.shape)
            y_pred = np.zeros(X.shape[1])
        y_pred = y_pred.astype(float64)
        if any(isnan(y_pred)) or any(isinf(y_pred)):
            raise ValueError('predict() produced NaN or inf values!')

        return np.ravel(y_pred)


reg = MyLinearRegression()

#training data
x_train = np.array([0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5]).reshape(-1, 1)
y_train = np.array([6.0, 4.83, 3.7, 3.15, 2.41, 1.83, 1.49, 1.21]).reshape(-1, 1)

#test data
X_test = np.array([4.5, 5.0, 4.0]).reshape(-1, 1)
y_test = np.array([0.73, 0.64,  0.96]).reshape(-1, 1)

sel = reg.fit(x_train, y_train)
print('betas',sel.coef_,sel.intercept_)

y_pred = reg.predict(X_test)
score = reg.score(X_test, y_test)
mse = mean_squared_error(y_test, y_pred)
print('R2 score is: ', score)
print('mean squares error is: ', mse)
```

```
betas [-1.34714286] 5.435
R2 score is:  -2584.778304773561
mean squares error is:  46.946241666666644
```

In [ ]:
```
from sklearn.utils.estimator_checks import check_estimator

estimator = MyLinearRegression()
gen = check_estimator(estimator, generate_only=True)
```

In [ ]:
```
from sklearn.utils.estimator_checks import check_estimator
```

```
estimator = MyLinearRegression()
# check_estimator(estimator)
try:
    check_estimator(estimator)
except:
    print('Some tests failed!')
```

```
Some tests failed!
```

In [ ]: