STK-IN4300 Statistical learning methods in Data Science: Assignment 2

Anh-Nguyet Lise Nguyen

November 17, 2019

1 Problem 1

1.1 Problem 1.1

[194]:
```
import numpy as np
import scipy.stats
import pandas as pd
import matplotlib.pyplot as plt
import sklearn.metrics as sklm
import sklearn.linear_model as skllm
import sklearn.model_selection as sklms
import sklearn.preprocessing as sklpre
import sklearn.neighbors as skln
import sklearn.tree as skt
import sklearn.ensemble as skle
import sklearn.neural_network as sknn
import pygam
```

[18]:
```
%load_ext rpy2.ipython
```

```
The rpy2.ipython extension is already loaded. To reload it, use:
  %reload_ext rpy2.ipython
```

[19]:
```
# Import the .csv file for the ozone data into a data frame
df = pd.read_csv("data/ozone_496obs_25vars.txt", header=0, sep=" ")
```

[20]:
```
# Extract variables except SEX
variables = df.loc[:,  (df.columns != "SEX")]

# Onehotting the SEX categorical variable
onehot_sex = pd.get_dummies(df.loc[:,"SEX"]).set_axis(
    ["MALE", "FEMALE"], axis=1, inplace=False
)
# Inserting MALE and FEMALE into variable data frame
variables = variables.join(onehot_sex)
```

[21]:
```
# Split into training and test set
variables_train_, variables_test_ = sklms.train_test_split(
    variables, test_size=0.5, stratify=variables["FSATEM"]
)
```

[22]:
```
# Which columns to scale:
col_continuous = np.array(["ALTER", "AGEBGEW", "FLGROSS", "FNOH24", "FLTOTMED",␣
 ↪"FO3H24", "FTEH24", "FLGEW",  "FFVC"])

# Defining the scaler (subtracts mean, divides by standard deviation):
scaler = sklpre.StandardScaler().fit(variables_train_[col_continuous].values)

# Standardizing the data using the scaler defined above in the continuous␣
 ↪variables
scaled_cols_train = scaler.transform(variables_train_[col_continuous])
scaled_cols_test = scaler.transform(variables_test_[col_continuous])
```

[23]:
```
# List of all variable names
feature_names = np.array(variables.columns.values, dtype=str)

# List of categorical variables
col_categorical = []
for item in feature_names:
    if item not in col_continuous:
        col_categorical.append(item)

# Data frame of the categorical variables
variables_categorical_train = variables_train_.loc[:, col_categorical]
variables_categorical_test = variables_test_.loc[:, col_categorical]

# Data frame of the continuous variables
variables_continuous_train = pd.DataFrame(scaled_cols_train,␣
 ↪columns=col_continuous)
variables_continuous_test = pd.DataFrame(scaled_cols_test,␣
 ↪columns=col_continuous)
```

[24]:
```
# Merging the categorical and continuous variables:

variables_final_train = variables_categorical_train.
 ↪join(variables_continuous_train)
variables_final_test = variables_categorical_test.join(variables_continuous_test)

variables_train_array = np.concatenate([variables_categorical_train,␣
 ↪variables_continuous_train], axis=1)
variables_final_train = pd.DataFrame(variables_train_array,␣
 ↪columns=(list(col_categorical)+list(col_continuous)))
```

```
variables_test_array = np.concatenate([variables_categorical_test,␣
 ↪variables_continuous_test], axis=1)
variables_final_test = pd.DataFrame(variables_test_array,␣
 ↪columns=(list(col_categorical)+list(col_continuous)))
```

[25]:
```
# Exporting the final preprocessed data frames as .csv
variables_final_train.to_csv("data/ozone_train.csv", index=False)
variables_final_test.to_csv("data/ozone_test.csv", index=False)
```

The categorical variables were not scaled and only SEX was encoded into FEMALE and MALE. This was not necessary for the other categorical variables as they were boolean.

1.2 Problem 1.2

[26]:
```
# Import the .csv for the training set made in Problem 1.1
df_train = pd.read_csv("data/ozone_train.csv")

# Extract explanatory features X and outcome y
X_train = df_train.loc[:, df_train.columns != "FFVC"]
y_train = df_train.loc[:, "FFVC"]
```

[27]:
```
# Import the .csv for the test set made in Problem 1.1
df_test = pd.read_csv("data/ozone_test.csv")

# Extract explanatory features X and outcome y
X_test = df_test.loc[:, df_test.columns != "FFVC"]
y_test = df_test.loc[:, "FFVC"]
```

[28]:
```
# Problem 1.2

def summary_OLS(X, y, feature_names):
    array_1D = None

    if len(np.shape(X)) == 1:
        array_1D = True
        X = X.reshape(-1,1)

    OLS = skllm.LinearRegression(fit_intercept=False).fit(X, y)
    y_pred = OLS.predict(X)
    coefficients = np.append(OLS.intercept_, OLS.coef_)


    newX = np.append(np.ones((len(X),1)), X, axis=1)
    MSE = (sum((y - y_pred)**2))/(len(newX)-len(newX[0]))
```

```
    var = MSE*(np.linalg.pinv(np.dot(newX.T,newX)).diagonal())
    std = np.sqrt(var)
    ts_b = coefficients/ std

    p = np.zeros(len(ts_b))
    for index, i in enumerate(ts_b):
        p[index] = 2*(1-scipy.stats.t.cdf(np.abs(i),(len(newX)-1)))

    if array_1D:
        summary = pd.DataFrame(zip(coefficients, std, p), columns=["Feature",␣
↪"Standard deviance", "p-values"]
                               , index=(["intercept"] + [feature_names]))

    else:
        summary = pd.DataFrame(zip(coefficients, std, p),␣
↪columns=["Coefficients", "Standard deviance", "p-values"],
                           index=(["intercept"] + list(feature_names)))

    return summary
```

[29]:
```
summary_train = summary_OLS(X_train, y_train, X_train.columns.values)

summary_test = summary_OLS(X_test, y_test, X_test.columns.values)

summary_train
```

[29]:

	Coefficients	Standard deviance	p-values
intercept	0.000000	0.064966	1.000000e+00
ADHEU	-0.085491	0.174001	6.236304e-01
HOCHOZON	-0.275212	0.114273	1.675705e-02
AMATOP	0.074196	0.100874	4.627138e-01
AVATOP	-0.053700	0.096761	5.794137e-01
ADEKZ	-0.133412	0.100308	1.847384e-01
ARAUCH	-0.035821	0.093061	7.006273e-01
FSNIGHT	-0.034033	0.152263	8.233179e-01
FMILB	-0.196045	0.154182	2.047412e-01
FTIER	0.055690	0.153182	7.165033e-01
FPOLL	-0.011709	0.198942	9.531140e-01
FSPT	-0.115361	0.212263	5.872887e-01
FSATEM	0.290896	0.233122	2.132759e-01
FSAUGE	0.100089	0.128319	4.361371e-01
FSPFEI	0.137500	0.248114	5.799572e-01
FSHLAUF	-0.004113	0.184195	9.822043e-01
MALE	0.471512	0.053396	2.220446e-16
FEMALE	0.007291	0.050103	8.844238e-01

```
ALTER          0.037743          0.048568  4.378254e-01
AGEBGEW        0.065523          0.041005  1.113365e-01
FLGROSS        0.482368          0.064714  1.526557e-12
FNOH24        -0.203376          0.053352  1.741477e-04
FLTOTMED       0.003050          0.041275  9.411556e-01
FO3H24         0.079722          0.087616  3.637616e-01
FTEH24        -0.038663          0.078167  6.213140e-01
FLGEW          0.261821          0.061050  2.580945e-05
```

1.3 Problem 1.3

I'm choosing to use criterias for p-value in backward elimination and forward selection.

1.3.1 Backward elimination

[30]:
```
# Problem 1.3

def backward_elimination(X, y, alpha):
    # Make copies of X and y
    X_copy = X.copy()
    y_copy = y.copy()

    feature_names = X_copy.columns.values
    # Calculating coefficients, standard deviations and p-values as summary
    # of initial X and y
    summary = summary_OLS(X_copy, y_copy, feature_names)
    p_values = summary.loc[:, "p-values"]

    # Find maximum p-value
    p_max = np.max(p_values)

    # Iterate until p <= alpha
    while p_max > alpha:
        # Calculating coefficients, standard deviations and p-values using␣
↪summary_OLS function
        feature_names = X_copy.columns.values
        summary = summary_OLS(X_copy, y_copy, feature_names)
        summary.drop("intercept", axis=0, inplace=True)
        p_values = summary.loc[:, "p-values"]

        # Locating index with maximum p-value and
        # dropping corresponding feature
        p_max_index = p_values.idxmax(axis=1)
        X_copy.drop(p_max_index, axis=1, inplace=True)

        # Update maximum p-value
        p_max = p_values.loc[p_max_index]
```

```
    feature_names = X_copy.columns.values

    return X_copy, summary_OLS(X_copy, y_copy, feature_names)
```

[31]:
```
X_backelim_1_train, summary_backelim_1_train = backward_elimination(X_train,␣
 ↪y_train, 1e-1)
print("Backward elimination using threshold 0.1 \n\n", summary_backelim_1_train)
```

```
Backward elimination using threshold 0.1

            Coefficients  Standard deviance      p-values
intercept       0.000000           0.079393  1.000000e+00
HOCHOZON       -0.265769           0.091128  3.866320e-03
FMILB          -0.235040           0.106927  2.886814e-02
MALE            0.428085           0.075915  4.662965e-08
AGEBGEW         0.064363           0.039638  1.057023e-01
FLGROSS         0.512045           0.055718  0.000000e+00
FNOH24         -0.182559           0.044339  5.229183e-05
FLGEW           0.251048           0.056676  1.419891e-05
```

[202]:
```
X_backelim_2_train, summary_backelim_2_train = backward_elimination(X_train,␣
 ↪y_train, 1e-3)
summary_backelim_2_train

print("Backward elimination using threshold 0.001 \n\n",␣
 ↪summary_backelim_2_train)
```

```
Backward elimination using threshold 0.001

            Coefficients  Standard deviance  p-values
intercept       0.000000           0.043006       1.0
FLGROSS         0.738259           0.043006       0.0
```

1.3.2 Forward selection

[34]:
```
def forward_selection(X, y, alpha):
    X_copy = X.copy()
    y_copy = y.copy()
    p_max = alpha - 1
    p_min = 0
    feature_include = []
    feature_names = np.array(list(X_copy.columns.values))
    while p_max < alpha and len(feature_names)>0:

        n_features = len(feature_names)
        p_values = np.zeros(n_features)
```

```
        for i, feature in enumerate(feature_names):

            features = np.append(feature_include, feature_names[i])

            X_feature = X_copy.loc[:, features].values
            summary = summary_OLS(X_feature, y_copy, features)

            summary.drop((["intercept"] + feature_include ), axis=0,␣
 ↪inplace=True)


            p_one = summary.loc[:, "p-values"].values
            p_values[i] = p_one

        p_min_index = np.argmin(p_values)
        p_min = p_values[p_min_index]

        feature_min_p = feature_names[p_min_index]
        feature_include.append(feature_min_p)
        feature_names = feature_names[feature_names != feature_min_p]


        X_feature = X_copy.loc[:, features].values

        summary_ = summary_OLS(X_feature, y_copy, feature_include)

        p_max = summary.loc[:, "p-values"].values.max()

        X_final = X_copy.loc[:, feature_include]

    return X_final, summary_OLS(X_final, y_copy, feature_include)
```

[35]:
```
X_forsec_1_train, summary_forsec_1_train = forward_selection(X_train, y_train,␣
 ↪1e-1)
print("Forward selection using threshold 0.1\n \n",summary_forsec_1_train)
```

```
Forward selection using threshold 0.1

            Coefficients  Standard deviance      p-values
intercept       0.000000           0.040738  1.000000e+00
FLGROSS         0.518356           0.060276  8.881784e-16
FLGEW           0.283676           0.060013  3.831985e-06
FNOH24         -0.135803           0.041170  1.114813e-03
```

[36]:
```
X_forsec_2_train, summary_forsec_2_train = forward_selection(X_train, y_train,␣
 ↪1e-3)
print(f"Forward selection using threshold 0.001\n \n",summary_forsec_2_train)
```

```
Forward selection using threshold 0.001

            Coefficients  Standard deviance      p-values
intercept       0.000000           0.040738  1.000000e+00
FLGROSS         0.518356           0.060276  8.881784e-16
FLGEW           0.283676           0.060013  3.831985e-06
FNOH24         -0.135803           0.041170  1.114813e-03
```

It is interesting to note that using the same criteria α for p-value in both backward elimination and forward selection, for $\alpha = 0.1$ the number of variables in the model is higher for backward elimination (6) than forward selection (3). For $\alpha = 0.001$ the backward selection model contains three variables, while the forward selection contains two variables. The variable FLGROSS is included in every model and seems to be the most significant. Looking at the low p-values of the remaining variables of the new models, I think it is likely that these models will perform better than the full models, as eliminating the other variables with higher p-values is like eliminating noise.

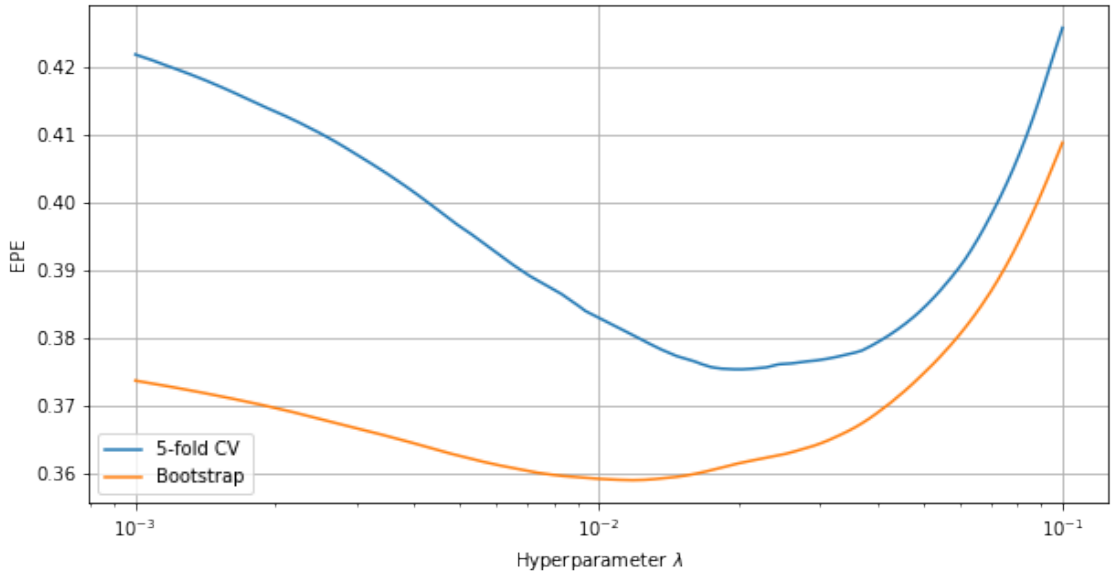
1.4 Problem 1.4

[37]:
```python
def bootstrap(y, n_iter=1000):
    N = len(y)
    N_train = int(0.8*N)
    N_test = int(0.2*N)

    all_indices = np.arange(N)

    indices_bootstrapped = []
    for i in range(n_iter):
        indices_train = np.random.choice(all_indices, size=N_train, replace=True)
        indices_test = np.random.choice(all_indices, size=N_test, replace=True)

        indices_bootstrapped.append([indices_train, indices_test])

    return indices_bootstrapped
```

[38]:
```python
shrinkage_param = np.logspace(-3, -1, 1000)

# Lasso with 5-fold cross validation
reg_lasso_cv = skllm.LassoCV(alphas=shrinkage_param, cv=5, n_jobs=-1,␣
 ↪fit_intercept=False).fit(X_train, y_train)
# calculate mean square error for the test on each fold,
# then take the mean over all folds axis=1
MSE_lasso_cv = np.mean(reg_lasso_cv.mse_path_, axis=1)
```

[39]:
```
a = bootstrap(y_train, n_iter=100)
reg_lasso_bootstrap = skllm.LassoCV(alphas=shrinkage_param, cv=a, n_jobs=-1,␣
 ↪fit_intercept=False).fit(X_train, y_train)
MSE_lasso_bootstrap = np.mean(reg_lasso_bootstrap.mse_path_, axis=1)
```

[40]:
```
figure_lasso = plt.figure(figsize=(10,5))
plt.semilogx(shrinkage_param, MSE_lasso_cv[::-1], label="5-fold CV")
plt.semilogx(shrinkage_param, MSE_lasso_bootstrap[::-1], label="Bootstrap")
plt.xlabel(r"Hyperparameter $\lambda$")
plt.ylabel("EPE")
plt.legend()
plt.grid()
plt.show()
```



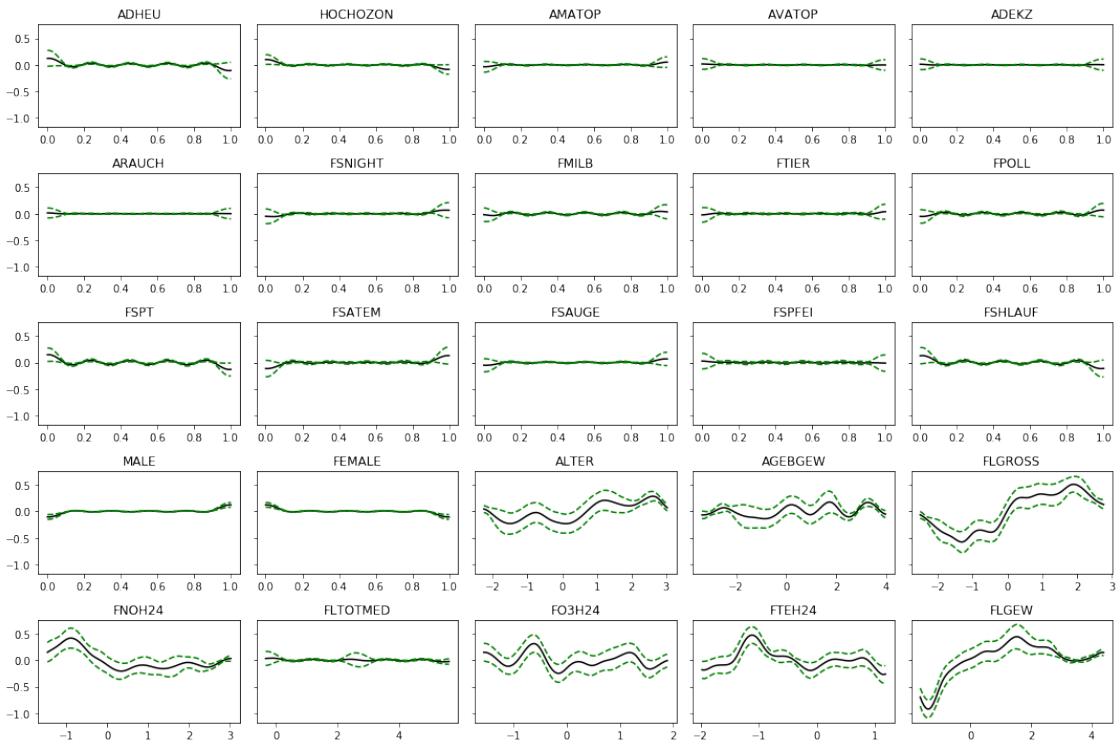
[41]:
```
print(f"The best shrinkage parameter for Lasso using CV is {reg_lasso_cv.alpha_:.
 ↪2e} "+
      f"with R2 score of {reg_lasso_cv.score(X_test,y_test):.2f}")


print(f"The best shrinkage parameter for Lasso using bootstrap is␣
 ↪{reg_lasso_bootstrap.alpha_:.2e} "+
      f"with R2 score of {reg_lasso_bootstrap.score(X_test,y_test):.2f}")
```

```
The best shrinkage parameter for Lasso using CV is 2.01e-02 with R2 score of
0.60
The best shrinkage parameter for Lasso using bootstrap is 1.17e-02 with R2 score
of 0.60
```

From the plot, the EPE found using bootstrap is noticably lower than the EPE found using 5-fold cross validation. This is due to the severe underestimation of the EPE by the bootstrapping method, as the training and test sets are not independent. This can be improved by using e.g. 0.632 bootstrap.

1.5 Problem 1.5

[42]:
```
# Checking for linear dependence

gam_check = pygam.LinearGAM().fit(X_train.values, y_train.values)
# Plotting
fig, axes = plt.subplots(5, 5, figsize=[15, 10], sharey=True)
for i, ax in enumerate(axes.ravel()):
    XX = gam_check.generate_X_grid(i)
    pdep, confi = gam_check.partial_dependence(term=i, X=XX, width=0.95,␣
 ↪meshgrid=False)
    ax.plot(XX[:, i], pdep, c="black")
    ax.plot(XX[:, i], confi[:, 0], c="green", ls="--")
    ax.plot(XX[:, i], confi[:, 1], c="green", ls="--")
    ax.set_title(X_train.columns.values[i])
fig.tight_layout()
plt.show()
```



Judging from the above plots, the variables that seem nonlinear to me are FLGROSS, FTEH24 and

FLGEW.

First we try to fit a GAM:

[43]:
```
nonlinear_features = ["FLGROSS", "FTEH24","FLGEW"]

# In pygam, pygam.l indicates linear, pygam.s is splines

# Initializing pygam for the first variable ADHEU, which is linear:
gam_feature_type =  pygam.l(0)

# Then for the rest of the features:
for i, feature in enumerate(X_train.columns.values[1:]):
    # Splines for non-linear features
    if feature in nonlinear_features:
        gam_feature_type +=  pygam.s(i, spline_order=3, n_splines=5)
    #  And linear for the others
    else:
        gam_feature_type += pygam.l(i)


# Grid search over penalties in log space and train the model
penalties = np.logspace(-5, 2, 150)
gam = pygam.LinearGAM(gam_feature_type).gridsearch(
    X_train.values, y_train.values, lam=penalties
)
```

```
100% (150 of 150) |######################| Elapsed Time: 0:00:14 Time:  0:00:14
```

There apparently is a bug in the pygam library where the p-values are incorrectly calculated and underestimated, so I have chosen not to include those results as the p-values would be lower than they should and would not be meaningful to interpret.

[44]:
```
r2_test_gam = gam._estimate_r2(X_test.values, y_test.
 ↪values)["explained_deviance"]
r2_train_gam = gam._estimate_r2(X_train.values, y_train.
 ↪values)["explained_deviance"]
print(f"R^2 score test: {r2_test_gam:.2f}, R^2 score train: {r2_train_gam:.2f}")
```

```
R^2 score test: 0.57, R^2 score train: 0.64
```

Now we will try to add non-linear terms to the linear model:

[45]:
```
# First need to undo the scaling and centering of the data for training and test␣
 ↪set:
variables_train_new = scaler.inverse_transform(
    variables_final_train.loc[:, col_continuous]
)
```

```
variables_test_new = scaler.inverse_transform(
    variables_final_test.loc[:, col_continuous]
)

# Turn into dataframes
variables_train_new_df = pd.DataFrame(variables_train_new,␣
 ↪columns=list(col_continuous))
variables_test_new_df = pd.DataFrame(variables_test_new,␣
 ↪columns=list(col_continuous))
```

[46]:
```
X_train_new = variables_train_new_df.loc[:, variables_train_new_df.columns!
 ↪="FFVC"]
X_test_new = variables_test_new_df.loc[:, variables_test_new_df.columns!="FFVC"]

features_continuous = col_continuous[col_continuous!="FFVC"]

X_nonlinear_train = []
X_nonlinear_test = []
nonlinear_names = []

for i, feature in enumerate(features_continuous):
    if feature in nonlinear_features:
        # Adding 2nd degree to the features I identified as nonlinear earlier
        X_nonlinear_train.append(X_train_new.values[:, i]**2)
        X_nonlinear_test.append(X_test_new.values[:, i]**2)
        nonlinear_names.append(feature + "^2")
        # Adding 3rd degree to the features I identified as nonlinear earlier
        X_nonlinear_train.append(X_train_new.values[:, i]**3)
        X_nonlinear_test.append(X_test_new.values[:, i]**3)
        nonlinear_names.append(feature + "^3")

X_nonlinear_train = np.array(X_nonlinear_train).T
X_nonlinear_test = np.array(X_nonlinear_test).T
```

[47]:
```
# Now we need to scale and center the new non-linear terms:
scaler_nonlinear = sklpre.StandardScaler().fit(X_nonlinear_train)

X_nonlinear_train_scaled = scaler_nonlinear.transform(X_nonlinear_train)
X_nonlinear_test_scaled = scaler_nonlinear.transform(X_nonlinear_test)

X_nonlinear_train_scaled_df = pd.DataFrame(X_nonlinear_train_scaled, columns =␣
 ↪nonlinear_names)
X_nonlinear_test_scaled_df = pd.DataFrame(X_nonlinear_test_scaled, columns =␣
 ↪nonlinear_names)
```

[48]:
```
X_train_nonlinear = X_train.join(X_nonlinear_train_scaled_df)

X_test_nonlinear = X_test.join(X_nonlinear_test_scaled_df)


summary_nonlinear = summary_OLS(X_train_nonlinear, y_train, X_train_nonlinear.
 ↪columns.values)

OLS_nonlinear = skllm.LinearRegression(fit_intercept=False).
 ↪fit(X_train_nonlinear, y_train)
print(f"R^2 score for nonlinear model: {OLS_nonlinear.score(X_test_nonlinear,␣
 ↪y_test)}")

summary_nonlinear
```

```
R^2 score for nonlinear model: 0.6224037436990644
```

[48]:

	Coefficients	Standard deviance	p-values
intercept	0.000000	0.066745	1.000000e+00
ADHEU	-0.079155	0.174881	6.512181e-01
HOCHOZON	-0.234284	0.118380	4.891669e-02
AMATOP	0.053766	0.101719	5.975740e-01
AVATOP	-0.019458	0.098088	8.429183e-01
ADEKZ	-0.094412	0.100758	3.496681e-01
ARAUCH	-0.015254	0.093362	8.703518e-01
FSNIGHT	-0.024180	0.151812	8.735840e-01
FMILB	-0.198726	0.154743	2.002638e-01
FTIER	0.074665	0.153674	6.274909e-01
FPOLL	-0.072878	0.200577	7.166599e-01
FSPT	-0.070002	0.215073	7.450951e-01
FSATEM	0.304387	0.234076	1.946837e-01
FSAUGE	0.122639	0.129272	3.437038e-01
FSPFEI	0.131439	0.246728	5.947012e-01
FSHLAUF	-0.095158	0.186268	6.099001e-01
MALE	0.434777	0.054070	3.708145e-14
FEMALE	-0.041283	0.051016	4.191775e-01
ALTER	0.035502	0.049072	4.700763e-01
AGEBGEW	0.057251	0.041682	1.708311e-01
FLGROSS	-11.572593	28.336824	6.833382e-01
FNOH24	-0.184276	0.054835	9.010599e-04
FLTOTMED	-0.000139	0.041100	9.972979e-01
FO3H24	0.089253	0.096183	3.543357e-01
FTEH24	1.206839	1.150865	2.953705e-01
FLGEW	4.426434	1.472596	2.921211e-03
FLGROSS^2	22.744971	56.815319	6.892583e-01
FLGROSS^3	-10.737684	28.523355	7.069038e-01
FTEH24^2	-2.522997	2.480757	3.101356e-01

```
FTEH24^3       1.288299           1.353470  3.421041e-01
FLGEW^2       -7.658038           2.772327  6.170810e-03
FLGEW^3        3.589892           1.357829  8.721709e-03
```

Looking at the R^2 score for the non-linear model, it does not seem to perform better than our previous models either, but seems to perform on par with the others. What is interesting is that the p-value for FLGEW^2 and FLGEW^3 seem to be rather low, so perhaps using these non-linear features with a penalized regression method such as the Lasso or the forward selection and backward elimination methods would yield better results.

1.6 Problem 1.6

Apparently, Python doesn't have any well-known libraries for boosting, so this was done using R with Jupyter Notebook. Using %%R at the top of the cell transforms the entire cell into an R cell.

[49]:
```
%%R -i df_train -i df_test

library(compboost)
library(ggplot2)
```

[50]:
```
%%R

linear_boost <- boostLinear(data = df_train, target = "FFVC", loss =␣
 ↪LossQuadratic$new(), trace=10)

linear_boost$getEstimatedCoef()
```

```
  1/100: risk = 0.47
 10/100: risk = 0.33
 20/100: risk = 0.26
 30/100: risk = 0.23
 40/100: risk = 0.21
 50/100: risk = 0.2
 60/100: risk = 0.19
 70/100: risk = 0.19
 80/100: risk = 0.18
 90/100: risk = 0.18
100/100: risk = 0.18


Train 100 iterations in 0 Seconds.
Final risk based on the train set: 0.18

$AGEBGEW_linear
              [,1]
[1,] 4.023172e-19
[2,] 1.385251e-02
```

```
$ALTER_linear
              [,1]
[1,] -6.196189e-18
[2,]  1.678067e-02

$FEMALE_linear
           [,1]
[1,]  0.1794215
[2,] -0.3588430

$FLGEW_linear
              [,1]
[1,] -8.123703e-17
[2,]  2.272637e-01

$FLGROSS_linear
              [,1]
[1,] -1.194482e-16
[2,]  4.878293e-01

$FNOH24_linear
              [,1]
[1,] -1.304551e-18
[2,] -6.963044e-02

$offset
[1] 7.27017e-16
```

[51]:
```
%%R
spline_boost <- boostSplines(data = df_train, target = "FFVC", loss =␣
 ↪LossQuadratic$new(), trace=10)

table(spline_boost$getSelectedBaselearner())
```

```
  1/100: risk = 0.47
 10/100: risk = 0.32
 20/100: risk = 0.24
 30/100: risk = 0.21
 40/100: risk = 0.19
 50/100: risk = 0.18
 60/100: risk = 0.17
 70/100: risk = 0.16
 80/100: risk = 0.15
 90/100: risk = 0.15
100/100: risk = 0.14
```

```
Train 100 iterations in 0 Seconds.
Final risk based on the train set: 0.14


AGEBGEW_spline   ALTER_spline  FEMALE_spline   FLGEW_spline FLGROSS_spline
            10             11              9             14             30
 FNOH24_spline  FO3H24_spline  FTEH24_spline    MALE_spline
             4              6             12              4
```

For component-wise boosting with trees, I wasn't able to find a similar method in the compboost package for R. Looking online, I did manage to find mboost which says it uses "component-wise (penalized) least squares estimates or regression trees as base-learners", so I hope it is correct to use this package for this purpose. I was not able to download this package, however. I tried troubleshooting this, but was not able to make it work. Sorry!

[52]:
```
%%R
library(mboost)

tree_boost <- blackboost(data=df_train, target="FFVC")
```

```
R[write to console]: Error in library(mboost) : there is no package called
'mboost'
Calls: <Anonymous> -> <Anonymous> -> withVisible -> library


Error in library(mboost) : there is no package called 'mboost'
Calls: <Anonymous> -> <Anonymous> -> withVisible -> library
```

1.7 Problem 1.7

[66]:
```
# MSE for full OLS model
OLS = skllm.LinearRegression()


OLS_full = OLS.fit(X_train, y_train)

y_OLS_train = OLS_full.predict(X_train)
MSE_OLS_train = sklm.mean_squared_error(y_train, y_OLS_train)

y_OLS_test = OLS_full.predict(X_test)
MSE_OLS_test = sklm.mean_squared_error(y_test, y_OLS_test)

print(f"MSE test for full OLS model: {MSE_OLS_test:.2f}")
print(f"MSE train for full OLS model: {MSE_OLS_train:.2f}\n")


# MSE for backward elimination model
OLS_backelim_1 = OLS.fit(X_backelim_1_train, y_train)
```

```
y_OLS_backelim_1_train = OLS_backelim_1.predict(X_backelim_1_train)
MSE_OLS_backelim_1_train = sklm.mean_squared_error(y_train,␣
 ↪y_OLS_backelim_1_train)

features_backelim_1 = X_backelim_1_train.columns.values

X_backelim_1_test = X_test.loc[:, features_backelim_1]
y_OLS_backelim_1_test = OLS_backelim_1.predict(X_backelim_1_test)
MSE_OLS_backelim_1_test = sklm.mean_squared_error(y_test, y_OLS_backelim_1_test)

print(f"MSE train for backwards elimination model using threshold = 0.1:␣
 ↪{MSE_OLS_backelim_1_train:.2f}")
print(f"MSE test for backwards elimination model using threshold = 0.1:␣
 ↪{MSE_OLS_backelim_1_test:.2f}\n")


# MSE for backward elimination model.
OLS_backelim_2 = OLS.fit(X_backelim_2_train, y_train)

y_OLS_backelim_2_train = OLS_backelim_2.predict(X_backelim_2_train)
MSE_OLS_backelim_2_train = sklm.mean_squared_error(y_train,␣
 ↪y_OLS_backelim_2_train)

features_backelim_2 = X_backelim_2_train.columns.values

X_backelim_2_test = X_test.loc[:, features_backelim_2]
y_OLS_backelim_2_test = OLS_backelim_2.predict(X_backelim_2_test)
MSE_OLS_backelim_2_test = sklm.mean_squared_error(y_test, y_OLS_backelim_2_test)

print(f"MSE train for backwards elimination model using threshold = 0.01:␣
 ↪{MSE_OLS_backelim_2_train:.2f}")
print(f"MSE test for backwards elimination model using threshold = 0.01:␣
 ↪{MSE_OLS_backelim_2_test:.2f}\n")

# MSE for forward selection model
OLS_forsec_1 = OLS.fit(X_forsec_1_train, y_train)

y_OLS_forsec_1_train = OLS_forsec_1.predict(X_forsec_1_train)
MSE_OLS_forsec_1_train = sklm.mean_squared_error(y_train, y_OLS_forsec_1_train)

features_forsec_1 = X_forsec_1_train.columns.values

X_forsec_1_test = X_test.loc[:, features_forsec_1]
y_OLS_forsec_1_test = OLS_forsec_1.predict(X_forsec_1_test)
MSE_OLS_forsec_1_test = sklm.mean_squared_error(y_test, y_OLS_forsec_1_test)
```

```
print(f"MSE train for forward selection model using threshold = 0.1:␣
 ↪{MSE_OLS_forsec_1_train:.2f}")
print(f"MSE test for forward selection model using threshold = 0.1:␣
 ↪{MSE_OLS_forsec_1_test:.2f}\n")


OLS_forsec_2 = OLS.fit(X_forsec_2_train, y_train)

y_OLS_forsec_2_train = OLS_forsec_2.predict(X_forsec_2_train)
MSE_OLS_forsec_2_train = sklm.mean_squared_error(y_train, y_OLS_forsec_2_train)

features_forsec_2 = X_forsec_2_train.columns.values

X_forsec_2_test = X_test.loc[:, features_forsec_2]
y_OLS_forsec_2_test = OLS_forsec_2.predict(X_forsec_2_test)
MSE_OLS_forsec_2_test = sklm.mean_squared_error(y_test, y_OLS_forsec_2_test)


print(f"MSE train for forward selection model using threshold = 0.01:␣
 ↪{MSE_OLS_forsec_2_train:.2f}")
print(f"MSE test for forward selection model using threshold = 0.01:␣
 ↪{MSE_OLS_forsec_2_test:.2f}\n")

# MSE for Lasso model

# reg_lasso_cv instance was already defined and fitted earlier, so we can just␣
 ↪call on it again
y_lasso_cv_train = reg_lasso_cv.predict(X_train)
MSE_lasso_cv_train = sklm.mean_squared_error(y_train, y_lasso_cv_train)

y_lasso_cv_test = reg_lasso_cv.predict(X_test)
MSE_lasso_cv_test = sklm.mean_squared_error(y_train, y_lasso_cv_test)

print(f"MSE train for Lasso model using CV: {MSE_lasso_cv_train:.2f}")
print(f"MSE test for Lasso model using CV: {MSE_lasso_cv_test:.2f}\n")


# reg_lasso_bootstrap instance was also defined and fitted earlier
y_lasso_bootstrap_train = reg_lasso_bootstrap.predict(X_train)
MSE_lasso_bootstrap_train = sklm.mean_squared_error(y_train,␣
 ↪y_lasso_bootstrap_train)

y_lasso_bootstrap_test = reg_lasso_bootstrap.predict(X_test)
```

```
MSE_lasso_bootstrap_test = sklm.mean_squared_error(y_test,␣
 ↪y_lasso_bootstrap_test)


print(f"MSE train for Lasso model using bootstrapping:␣
 ↪{MSE_lasso_bootstrap_train:.2f}")
print(f"MSE test for Lasso model using bootstrapping: {MSE_lasso_bootstrap_test:.
 ↪2f}\n")



# MSE for GAM model

# gam instance was already defined and fitted earlier
y_gam_train = gam.predict(X_train)
MSE_gam_train = sklm.mean_squared_error(y_train, y_gam_train)

y_gam_test = gam.predict(X_test)
MSE_gam_test = sklm.mean_squared_error(y_train, y_gam_test)


print(f"MSE train for GAM model: {MSE_gam_train:.2f}")
print(f"MSE test for GAM model: {MSE_gam_test:.2f}\n")

# MSE for non-linear model

# OLS_nonlinear instance was already defined and fitted earlier

y_nonlinear_train = OLS_nonlinear.predict(X_train_nonlinear)
MSE_nonlinear_train = sklm.mean_squared_error(y_train, y_nonlinear_train)

y_nonlinear_test = OLS_nonlinear.predict(X_test_nonlinear)
MSE_nonlinear_test = sklm.mean_squared_error(y_test, y_nonlinear_test)

print(f"MSE train for nonlinear model: {MSE_nonlinear_train:.2f}")
print(f"MSE test for nonlinear model: {MSE_nonlinear_test:.2f}\n")


# MSE for linear boosting model

y_linearboost_train = %R linear_boost$predict(df_train)
MSE_linearboost_train = sklm.mean_squared_error(y_train, y_linearboost_train)

y_linearboost_test = %R linear_boost$predict(df_test)
MSE_linearboost_test = sklm.mean_squared_error(y_test, y_linearboost_test)
```

```
print(f"MSE train for linear boosting model: {MSE_linearboost_train:.2f}")
print(f"MSE test for linear boosting model: {MSE_linearboost_test:.2f}\n")


# MSE for splines boosting model

y_splineboost_train = %R spline_boost$predict(df_train)
MSE_splineboost_train = sklm.mean_squared_error(y_train, y_splineboost_train)

print(f"MSE train for spline boosting model: {MSE_splineboost_train:.2f}")

#y_splineboost_test = %R spline_boost$predict(df_test)
#MSE_splineboost_test = sklm.mean_squared_error(y_test, y_splineboost_test)
# Sadly, trying to get test error crashes the program? So I guess I can't answer␣
 ↪that part
```

```
MSE test for full OLS model: 0.36
MSE train for full OLS model: 0.32

MSE train for backwards elimination model using threshold = 0.1: 0.34
MSE test for backwards elimination model using threshold = 0.1: 0.36

MSE train for backwards elimination model using threshold = 0.01: 0.45
MSE test for backwards elimination model using threshold = 0.01: 0.44

MSE train for forward selection model using threshold = 0.1: 0.40
MSE test for forward selection model using threshold = 0.1: 0.43

MSE train for forward selection model using threshold = 0.01: 0.40
MSE test for forward selection model using threshold = 0.01: 0.43

MSE train for Lasso model using CV: 0.35
MSE test for Lasso model using CV: 1.63

MSE train for Lasso model using bootstrapping: 0.34
MSE test for Lasso model using bootstrapping: 0.35

MSE train for GAM model: 0.36
MSE test for GAM model: 1.58

MSE train for nonlinear model: 0.31
MSE test for nonlinear model: 0.33

MSE train for linear boosting model: 0.36
MSE test for linear boosting model: 0.34

MSE train for spline boosting model: 0.28
```

For the boost model errors, I needed to export the models written in R to Python. This was done using %R.

The lowest MSE train was gained using boosted splines. Sadly, I can't seem to get the MSE for the test set and compare. A lot of the models seem to perform on par with each other. The full OLS model, Lasso model using bootstrap, boosted linear model, and non-linear model seem to perform similarly well.

The Lasso model using CV and GAM model seem to have overfitted the data and give far higher test error than train.

I earlier said I thought the backwards elimination and forward selection models would perform better than the standard OLS model, but it seems my threshold was too low and I ended up eliminating too many variables. Raising the threshold would probably improve these models. The backwards elimination model using the threshold = 0.1 actually performed better than the backward elimination with lower threshold and forward selection, which supports the claim that the models were too strict.

2 Problem 2

[67]:
```
%%R
library(mlbench)
data(PimaIndiansDiabetes)
```

[110]:
```
df_pima = %R PimaIndiansDiabetes
X_pima = df_pima.loc[:, df_pima.columns!="diabetes"]
y_pima = df_pima.loc[:, "diabetes"]

# Making pos = True, neg = False, then turning into integers of 1 and 0
y_pima = (y_pima.values == "pos").astype(np.int)

# Dividing into training and test sets with same proportion of positive/negative␣
 ↪diabetes and training size 2/3
Xp_train_, Xp_test_, yp_train, yp_test = sklms.train_test_split(X_pima, y_pima,␣
 ↪stratify=y_pima, test_size=1/3)
```

[94]:
```
# Define the scaler
scaler_p = sklpre.StandardScaler().fit(Xp_train_)

# Scale the data with respect to Xp_train
Xp_train_scaled = scaler_p.transform(Xp_train_)
Xp_test_scaled = scaler_p.transform(Xp_test_)

# Turn back into data frames
Xp_train = pd.DataFrame(Xp_train_scaled, columns=X_pima.columns)
Xp_test = pd.DataFrame(Xp_test_scaled, columns=X_pima.columns)
```

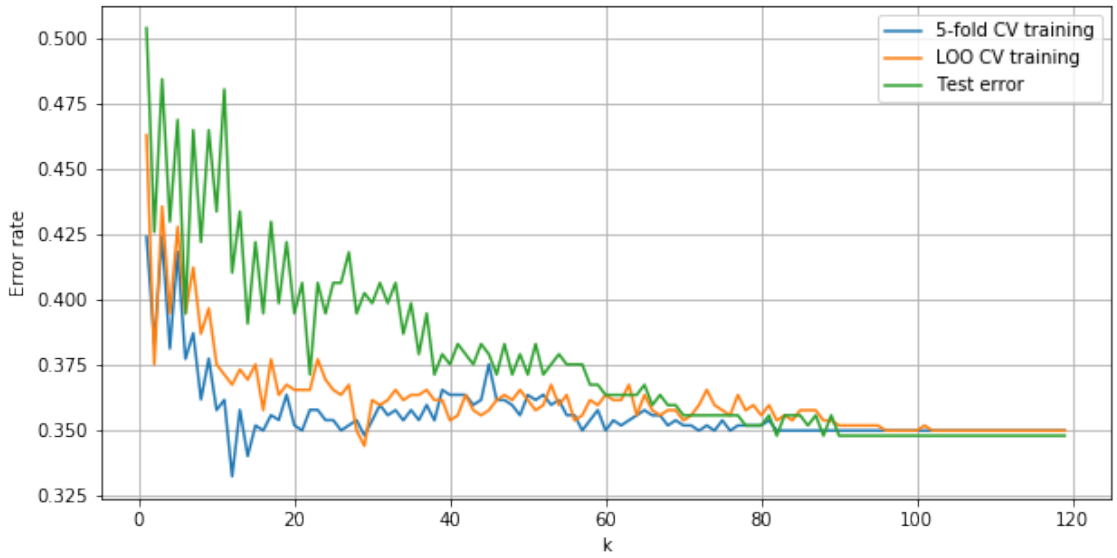
2.1 Problem 2.1

[145]:
```
# Creating instance of kNN Classifier
kNN_model = skln.KNeighborsClassifier()

# Creating dictionary of k-values for the grid search
k_values = np.arange(1, 120)
k_grid = {"n_neighbors": k_values}

# 5-fold CV grid search
kNN_gridsearch_5 = sklms.GridSearchCV(kNN_model, param_grid=k_grid, cv=5,␣
 ↪iid=False)

# LOO CV grid search
loo = sklms.LeaveOneOut()
kNN_gridsearch_loo = sklms.GridSearchCV(kNN_model, param_grid=k_grid, cv=loo,␣
 ↪iid=False)
```

[146]:
```
# Fit the models to the training data with the most optimal k value
kNN_train_cv = kNN_gridsearch_5.fit(Xp_train, yp_train)

kNN_train_loo = kNN_gridsearch_loo.fit(Xp_train, yp_train)
```

[147]:
```
# Retrieving error rates from the CV results
kNN_train_cv_err = 1 - kNN_train_cv.cv_results_["mean_test_score"]
kNN_train_loo_err = 1 - kNN_train_loo.cv_results_["mean_test_score"]


# Calculating test error rates
kNN_test_err = np.zeros_like(k_values, dtype=float)

for k in k_values:
    kNN_test = skln.KNeighborsClassifier(n_neighbors=k).fit(Xp_train, yp_train)
    kNN_test_err[k-1] = 1 - kNN_test.score(Xp_test, yp_test)
```

[148]:
```
figure_kNN = plt.figure(figsize=(10,5))
plt.plot(k_values, kNN_train_cv_err, label="5-fold CV training")
plt.plot(k_values, kNN_train_loo_err, label="LOO CV training")
plt.plot(k_values, kNN_test_err, label="Test error")
plt.legend()
plt.xlabel("k")
plt.ylabel("Error rate")
plt.grid()

plt.show()
```



5-fold CV and LOO seem to underestimate the error rate by a lot. The shape seems to be somewhat similar, which is the most important part, as we are more interested in finding the value for k that minimizes the error. These methods are okay for finding the most optimal k, but it is better to look at test error when you need to consider the actual error in the model.

However, in this case, it seems that the kNN model is not able to get a lower error rate than approximately 0.350. The data set itself contains 768 samples, with 268 positive outcomes and we have $268/768 \approx 0.349$. What probably happens is that kNN starts predicting only 0's (negative outcomes) when k is sufficiently large, and because approximately 35% of the data outcomes are 1's, the error rate becomes 35%.

2.2 Problem 2.2

Considering the bug with the GAM library in Python, I can't do a subset selection, but I will still fit the model.

[186]:
```
gam_splines = pygam.s(0)

for i in range(1, len(Xp_train.columns.values)):
    gam_splines += pygam.s(i, spline_order=2, n_splines=3)

penalties = np.logspace(-5, 0, 150)
with np.errstate(over="ignore"):
    gam_p = pygam.LogisticGAM(gam_splines).gridsearch(
        Xp_train.values, yp_train, lam=penalties
    )
```

```
100% (150 of 150) |######################| Elapsed Time: 0:00:14 Time:  0:00:14
```

[187]:
```
print(f"Train error rate using GAM: {1-gam_p.accuracy(Xp_train, yp_train)}")
print(f"Test error rate using GAM: {1-gam_p.accuracy(Xp_test, yp_test)}")
```

```
Train error rate using GAM: 0.34765625
Test error rate using GAM: 0.3671875
```

GAM seems to perform better than kNN on the training test, but not by a lot. The test set error rate is noticably higher, indicating perhaps a slight overfit. Using subset selection would probably improve the model, but the p-value related bug in pyGAM would have left me with all the variables regardless as it is prone to underestimating the values.

2.3 Problem 2.3

[214]:
```
# Trees
tree = skt.DecisionTreeClassifier()

tree_depths = {"max_depth": np.arange(1, 20)}
tree_gridsearch = sklms.GridSearchCV(
    tree, param_grid=tree_depths, cv=5, n_jobs=-1, iid=False
).fit(Xp_train, yp_train)

print(f"Best tree depth: {tree_gridsearch.best_params_}")
print(f"Tree train error rate: {1-tree_gridsearch.score(Xp_train, yp_train)}")
print(f"Tree test error rate: {1-tree_gridsearch.score(Xp_test, yp_test)}")
```

```
Best tree depth: {'max_depth': 2}
Tree train error rate: 0.314453125
Tree test error rate: 0.40234375
```

[215]:
```
# Bagging
bagging = skle.BaggingClassifier()

n_trees = {"n_estimators": np.arange(1, 100)}
bagging_gridsearch = sklms.GridSearchCV(
    bagging, param_grid=n_trees, cv=5, n_jobs=-1, iid=False
).fit(Xp_train, yp_train)

print(f"Best number of base estimators (bagging): {bagging_gridsearch.
 ↪best_params_}")
print(f"Bagging train error rate: {1-bagging_gridsearch.score(Xp_train,␣
 ↪yp_train)}")
print(f"Bagging test error rate: {1-bagging_gridsearch.score(Xp_test, yp_test)}")
```

```
Best number of base estimators (bagging): {'n_estimators': 38}
Bagging train error rate: 0.001953125
Bagging test error rate: 0.43359375
```

Note: This is only for probability. Scikit-learn does not have an implementation for consensus votes as it tends to perform worse than probability.

[216]:
```
# Random forest
randomforest = skle.RandomForestClassifier()

hyperparams_forest = {"n_estimators": np.arange(1, 100), "max_depth": np.
 ↪arange(1,30)}

randomforest_search = sklms.RandomizedSearchCV(
    randomforest, param_distributions=hyperparams_forest, n_iter=200,cv=5,␣
 ↪n_jobs=-1, iid=False
).fit(Xp_train, yp_train)

print(f"Best number of base estimators and max depth (random forest):␣
 ↪{randomforest_search.best_params_}")
print(f"Random forest train error rate: {1 - randomforest_search.score(Xp_train,␣
 ↪yp_train)}")
print(f"Random forest test error rate: {1 - randomforest_search.score(Xp_test,␣
 ↪yp_test)}")
```

```
Best number of base estimators and max depth (random forest): {'n_estimators':
30, 'max_depth': 8}
Random forest train error rate: 0.103515625
Random forest test error rate: 0.3984375
```

[217]:
```
# Neural network
NN = sknn.MLPClassifier(hidden_layer_sizes=(100, 50))

hyperparams_nn = {
    "learning_rate_init": np.logspace(-4,0, 100),
    "alpha": np.logspace(-4, 0, 100),
    "batch_size":  np.arange(10, 400)
}

NN_search = sklms.RandomizedSearchCV(
    NN, param_distributions=hyperparams_nn, cv=5, n_iter=150, n_jobs=-1,␣
 ↪iid=False
).fit(Xp_train, yp_train)

print(f"Best learning rate, shrinkage alpha and batch size (NN): {NN_search.
 ↪best_params_}")
print(f"Neural network train error rate: {1 - NN_search.score(Xp_train,␣
 ↪yp_train)}")
print(f"Neural network test error rate: {1 - NN_search.score(Xp_test, yp_test)}")
```

```
Best learning rate, shrinkage alpha and batch size (NN): {'learning_rate_init':
0.15556761439304723, 'batch_size': 296, 'alpha': 0.00017475284000076836}
Neural network train error rate: 0.349609375
Neural network test error rate: 0.34765625
```

[199]:
```
adaboost = skle.AdaBoostClassifier()

hyperparams_ada = {"n_estimators": np.arange(1, 300), "learning_rate": np.
 ↪logspace(-4, 0, 100)}

adaboost_search = sklms.RandomizedSearchCV(
    adaboost, param_distributions=hyperparams_ada, cv=5, n_iter=150, n_jobs=-1,␣
 ↪iid=False
).fit(Xp_train, yp_train)

print(f"Best number of estimators and learning rate (AdaBoost): {adaboost_search.
 ↪best_params_}")
print(f"AdaBoost train error rate: {1 - adaboost_search.score(Xp_train,␣
 ↪yp_train)}")
print(f"AdaBoost test error rate: {1 - adaboost_search.score(Xp_test, yp_test)}")
```

```
Best number of estimators and learning rate (AdaBoost): {'n_estimators': 62,
'learning_rate': 0.002364489412645407}
AdaBoost train error rate: 0.349609375
AdaBoost test error rate: 0.34765625
```

2.4 Problem 2.4

I would probably use AdaBoost, as the tree methods (Decision trees, bagging, random forests) seem to easily overfit and the neural network requires more hyperparameter tuning. AdaBoost and Neural network seem to perform practically identically in this case, but AdaBoost actually requires one fewer hyperparameter to tune, which makes it a lot more practical to use. kNN was basically useless.

2.5 Problem 2.5

[200]:
```
%%R
library(mlbench)
data(PimaIndiansDiabetes2)
```

[208]:
```
df_pima2 = %R PimaIndiansDiabetes

# Dropping all NA values
df_pima2.dropna(inplace=True)

X_pima2 = df_pima2.loc[:, df_pima2.columns!="diabetes"]
y_pima2 = df_pima2.loc[:, "diabetes"]

# Making pos = True, neg = False, then turning into integers of 1 and 0
y_pima2 = (y_pima2.values == "pos").astype(int)
```

```
Xp2_train_, Xp2_test_, yp2_train, yp2_test = sklms.train_test_split(X_pima2,␣
 ↪y_pima2, stratify=y_pima2, test_size=1/3)


# Define the scaler
scaler_p2 = sklpre.StandardScaler().fit(Xp2_train_)

# Scale the data with respect to Xp2_train
Xp2_train_scaled = scaler_p2.transform(Xp2_train_)
Xp2_test_scaled = scaler_p2.transform(Xp2_test_)

# Turn back into data frames
Xp2_train = pd.DataFrame(Xp2_train_scaled, columns=X_pima2.columns)
Xp2_test = pd.DataFrame(Xp2_test_scaled, columns=X_pima2.columns)
```

Now to repeat all the methods:

[221]:
```
# GAM
gam_splines2 = pygam.s(0)

for i in range(1, len(Xp2_train.columns.values)):
    gam_splines2 += pygam.s(i, spline_order=2, n_splines=3)

with np.errstate(over="ignore"):
    gam_p2 = pygam.LogisticGAM(gam_splines2).gridsearch(
        Xp2_train.values, yp2_train, lam=penalties
    )

print(f"Train error rate using GAM: {1-gam_p2.accuracy(Xp2_train, yp2_train)}")
print(f"Test error rate using GAM: {1-gam_p2.accuracy(Xp2_test, yp2_test)}\n")


# kNN
kNN2_gridsearch_5 = sklms.GridSearchCV(kNN_model, param_grid=k_grid, cv=5,␣
 ↪iid=False).fit(Xp2_train, yp2_train)

print(f"Best k (kNN): {kNN2_gridsearch_5.best_params_}")
print(f"kNN train error rate (kNN): {1-kNN2_gridsearch_5.score(Xp2_train,␣
 ↪yp2_train)}")
print(f"kNN test error rate (kNN): {1-kNN2_gridsearch_5.score(Xp2_test,␣
 ↪yp2_test)}\n")


# Trees
tree_gridsearch2 = sklms.GridSearchCV(
    tree, param_grid=tree_depths, cv=5, n_jobs=-1, iid=False
).fit(Xp2_train, yp2_train)
```

```
print(f"Best tree depth: {tree_gridsearch2.best_params_}")
print(f"Tree train error rate: {1-tree_gridsearch2.score(Xp2_train, yp2_train)}")
print(f"Tree test error rate: {1-tree_gridsearch2.score(Xp2_test, yp2_test)}\n")


# Bagging
bagging_gridsearch2 = sklms.GridSearchCV(
    bagging, param_grid=n_trees, cv=5, n_jobs=-1, iid=False
).fit(Xp2_train, yp2_train)

print(f"Best number of base estimators (bagging): {bagging_gridsearch2.
 ↪best_params_}")
print(f"Bagging train error rate: {1-bagging_gridsearch2.score(Xp2_train,␣
 ↪yp2_train)}")
print(f"Bagging test error rate: {1-bagging_gridsearch2.score(Xp2_test,␣
 ↪yp2_test)}\n")


# Random forest
randomforest_search2 = sklms.RandomizedSearchCV(
    randomforest, param_distributions=hyperparams_forest, n_iter=200,cv=5,␣
 ↪n_jobs=-1, iid=False
).fit(Xp2_train, yp2_train)

print(f"Best number of base estimators and max depth (random forest):␣
 ↪{randomforest_search2.best_params_}")
print(f"Random forest train error rate: {1 - randomforest_search2.
 ↪score(Xp2_train, yp2_train)}")
print(f"Random forest test error rate: {1 - randomforest_search2.score(Xp2_test,␣
 ↪yp2_test)}\n")


# Neural network
NN_search2 = sklms.RandomizedSearchCV(
    NN, param_distributions=hyperparams_nn, cv=5, n_iter=150, n_jobs=-1,␣
 ↪iid=False
).fit(Xp2_train, yp2_train)

print(f"Best learning rate, shrinkage alpha and batch size (NN): {NN_search2.
 ↪best_params_}")
print(f"Neural network train error rate: {1 - NN_search2.score(Xp2_train,␣
 ↪yp2_train)}")
print(f"Neural network test error rate: {1 - NN_search2.score(Xp2_test,␣
 ↪yp2_test)}\n")
```

```
# Adaptive boosting

adaboost_search2 = sklms.RandomizedSearchCV(
    adaboost, param_distributions=hyperparams_ada, cv=5, n_iter=150, n_jobs=-1,␣
 ↪iid=False
).fit(Xp2_train, yp2_train)

print(f"Best number of estimators and learning rate (AdaBoost):␣
 ↪{adaboost_search2.best_params_}")
print(f"AdaBoost train error rate: {1 - adaboost_search2.score(Xp2_train,␣
 ↪yp2_train)}")
print(f"AdaBoost test error rate: {1 - adaboost_search2.score(Xp2_test,␣
 ↪yp2_test)}\n")
```

```
100% (150 of 150) |######################| Elapsed Time: 0:00:16 Time:  0:00:16

Train error rate using GAM: 0.2265625
Test error rate using GAM: 0.19921875

Best k (kNN): {'n_neighbors': 11}
kNN train error rate (kNN): 0.248046875
kNN test error rate (kNN): 0.26953125

Best tree depth: {'max_depth': 2}
Tree train error rate: 0.2734375
Tree test error rate: 0.27734375

Best number of base estimators (bagging): {'n_estimators': 22}
Bagging train error rate: 0.005859375
Bagging test error rate: 0.22265625

Best number of base estimators and max depth (random forest): {'n_estimators':
60, 'max_depth': 29}
Random forest train error rate: 0.0
Random forest test error rate: 0.1953125


/home/lise/.local/share/virtualenvs/STK-
IN4300-assignment2-hpF4PA7F/lib/python3.6/site-
packages/sklearn/neural_network/multilayer_perceptron.py:566:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

Best learning rate, shrinkage alpha and batch size (NN): {'learning_rate_init':
0.00010974987654930556, 'batch_size': 84, 'alpha': 0.011497569953977368}
Neural network train error rate: 0.205078125
Neural network test error rate: 0.203125
```

```
Best number of estimators and learning rate (AdaBoost): {'n_estimators': 154,
'learning_rate': 0.02915053062825179}
AdaBoost train error rate: 0.234375
AdaBoost test error rate: 0.22265625
```

The tree ensemble methods (bagging, random forest) still seem to be overfitting, but the single tree appears to perform better, although this could also just be a coincidence. GAM and kNN are still performing worse than Adaboost and Neural networks, but at least do not seem to be overfitting as much as bagging and random forest. For the chosen hyperparameters, the neural network performance is actually better than AdaBoost, with an error rate of 0.20 compared to 0.22. It is possible that tuning the hyperparameters of the neural network and AdaBoost further would improve performances, but for the given parameters here, the neural network is better. Using randomized search for the hyperparameters also shortens the time needed to find optimal parameters, so I would be inclined to go for the neural network over AdaBoost.