MLPy Workshop 5

Alexandru Girban (S2148980), Hariaksh Pandya (S2692608)

February 14, 2025

1 Week 5 - Regularization

1.1 Aims

By the end of this notebook you will be able to

- perform regulized regression in sklearn
- understand the role of tuning parameter(s)
- use cross-validation for model tuning and comparison.

1. Problem Definition and Setup
2. Exploratory Data Analysis
3. Baseline Model
4. Ridge Regression
5. Lasso Regression
6. ElasticNet Regression

During workshops, you will complete the worksheets together in teams of 2-3, using **pair programming**. You should aim to switch roles between driver and navigator approximately every 15 minutes. When completing worksheets:

- You will have tasks tagged by (CORE) and (EXTRA).
- Your primary aim is to complete the (CORE) components during the WS session, afterwards you can try to complete the (EXTRA) tasks for your self-learning process.

Instructions for submitting your workshops can be found at the end of worksheet. As a reminder, you must submit a pdf of your notebook on Learn by 16:00 PM on the Friday of the week the workshop was given.

---

2 Problem Definition and Setup

2.1 Packages

First, let's load some of the packages you wil need for this workshop (we will load others as we progress).

[1]:
```python
# Data libraries
import pandas as pd
import numpy as np

# Plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Plotting defaults
plt.rcParams['figure.figsize'] = (10,6)
plt.rcParams['figure.dpi'] = 80

# sklearn modules
import sklearn
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV, KFold
```

2.2 User Defined Helper Functions

We will make use of the two helper functions that we used last week.

[2]:
```python
def get_coefs(m):
    """Returns the model coefficients from a Scikit-learn model object as an␣
 ↪array,
    includes the intercept if available.
    """

    # If pipeline, use the last step as the model
    if (isinstance(m, sklearn.pipeline.Pipeline)):
        m = m.steps[-1][1]


    if m.intercept_ is None:
        return m.coef_

    return np.concatenate([[m.intercept_], m.coef_])
```

[3]:
```python
def model_fit(m, X, y, plot = False):
    """Returns the mean squared error, root mean squared error and R^2 value of␣
 ↪a fitted model based
    on provided X and y values.

    Args:
        m: sklearn model object
        X: model matrix to use for prediction
        y: outcome vector to use to calculating rmse and residuals
        plot: boolean value, should fit plots be shown
    """
    y_hat = m.predict(X)
    MSE = mean_squared_error(y, y_hat)
    RMSE = np.sqrt(mean_squared_error(y, y_hat))
    Rsqr = r2_score(y, y_hat)

    Metrics = (round(MSE, 4), round(RMSE, 4), round(Rsqr, 4))

    res = pd.DataFrame(
        data = {'y': y, 'y_hat': y_hat, 'resid': y - y_hat}
    )

    if plot:
        plt.figure(figsize=(12, 6))

        plt.subplot(121)
        sns.lineplot(x='y', y='y_hat', color="grey", data =  pd.
↪DataFrame(data={'y': [min(y),max(y)], 'y_hat': [min(y),max(y)]}))
        sns.scatterplot(x='y', y='y_hat', data=res).set_title("Observed vs␣
↪Fitted values")

        plt.subplot(122)
        sns.scatterplot(x='y_hat', y='resid', data=res).set_title("Fitted␣
↪values vs Residuals")
        plt.hlines(y=0, xmin=np.min(y), xmax=np.max(y), linestyles='dashed',␣
↪alpha=0.3, colors="black")

        plt.subplots_adjust(left=0.0)

        plt.suptitle("Model (MSE, RMSE, Rsq) = " + str(Metrics), fontsize=14)
        plt.show()

    return MSE, RMSE, Rsqr
```

2.3 Data

The data for this week's workshop comes from the Elements of Statistical Learning textbook. The data originally come from a study by Stamey et al. (1989) in which they examined the relationship between the level of prostate-specific antigen (`psa`) and a number of clinical measures in men who were about to receive a prostatectomy. The variables are as follows,

- `lpsa` - log of the level of prostate-specific antigen
- `lcavol` - log cancer volume
- `lweight` - log prostate weight
- `age` - patient age
- `lbph` - log of the amount of benign prostatic hyperplasia
- `svi` - seminal vesicle invasion

- `lcp` - log of capsular penetration
- `gleason` - Gleason score
- `pgg45` - percent of Gleason scores 4 or 5
- `train` - test / train split used in ESL

These data are available in `prostate.csv`, which is included in the workshop materials.

Let's start by reading in the data.

[4]:
```
prostate = pd.read_csv('prostate.csv')
prostate.head()
```

[4]:
```
     lcavol   lweight  age      lbph  svi       lcp  gleason  pgg45      lpsa  \
0 -0.579818  2.769459   50 -1.386294    0 -1.386294        6      0 -0.430783
1 -0.994252  3.319626   58 -1.386294    0 -1.386294        6      0 -0.162519
2 -0.510826  2.691243   74 -1.386294    0 -1.386294        7     20 -0.162519
3 -1.203973  3.282789   58 -1.386294    0 -1.386294        6      0 -0.162519
4  0.751416  3.432373   62 -1.386294    0 -1.386294        6      0  0.371564

  train
0     T
1     T
2     T
3     T
4     T
```

3 Exploratory Data Analysis

Before modelling, we will start with EDA to gain an understanding of the data, through descriptive statistics and visualizations.

3.0.1 Exercise 1 (CORE)

a) Examine the data structure, look at the descriptive statistics, and create a pairs plot. Do any of our variables appear to be categorical / ordinal rather than numeric?

b) Are there any interesting patterns in these data? Which variable appears likely to have the strongest relationship with `lpsa`? Why do you think we are exploring the relationship between these variables and `lpsa` (log of psa) rather than just psa?

[5]:
```
# Part a
prostate.info()
prostate.describe()
```

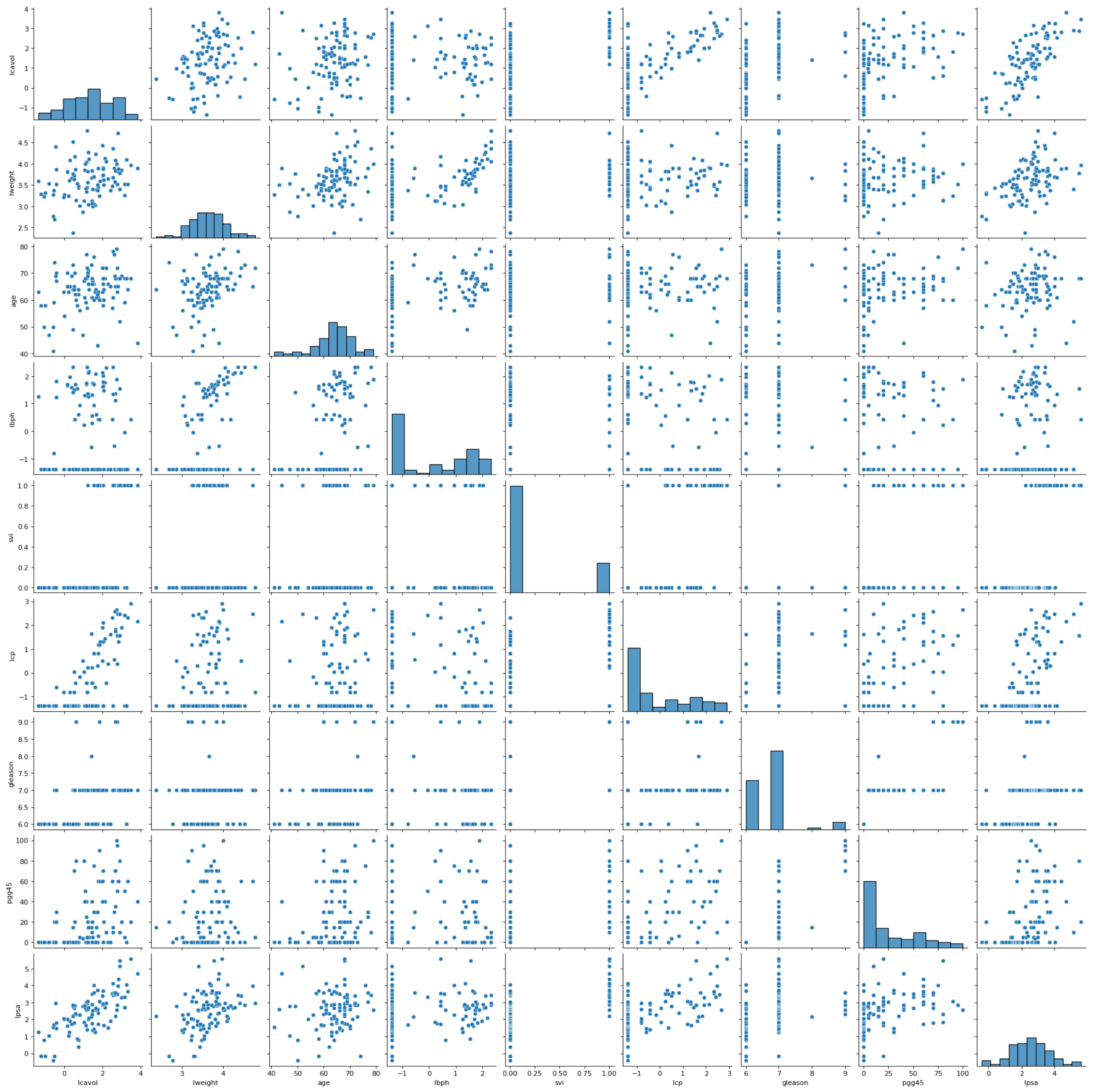
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 97 entries, 0 to 96
Data columns (total 10 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
```

```
 0   lcavol   97 non-null     float64
 1   lweight  97 non-null     float64
 2   age      97 non-null     int64
 3   lbph     97 non-null     float64
 4   svi      97 non-null     int64
 5   lcp      97 non-null     float64
 6   gleason  97 non-null     int64
 7   pgg45    97 non-null     int64
 8   lpsa     97 non-null     float64
 9   train    97 non-null     object
dtypes: float64(5), int64(4), object(1)
memory usage: 7.7+ KB
```

[5]:
```
          lcavol    lweight        age       lbph        svi        lcp  \
count  97.000000  97.000000  97.000000  97.000000  97.000000  97.000000
mean    1.350010   3.628943  63.865979   0.100356   0.216495  -0.179366
std     1.178625   0.428411   7.445117   1.450807   0.413995   1.398250
min    -1.347074   2.374906  41.000000  -1.386294   0.000000  -1.386294
25%     0.512824   3.375880  60.000000  -1.386294   0.000000  -1.386294
50%     1.446919   3.623007  65.000000   0.300105   0.000000  -0.798508
75%     2.127041   3.876396  68.000000   1.558145   0.000000   1.178655
max     3.821004   4.780383  79.000000   2.326302   1.000000   2.904165

         gleason       pgg45       lpsa
count  97.000000   97.000000  97.000000
mean    6.752577   24.381443   2.478387
std     0.722134   28.204035   1.154329
min     6.000000    0.000000  -0.430783
25%     6.000000    0.000000   1.731656
50%     7.000000   15.000000   2.591516
75%     7.000000   40.000000   3.056357
max     9.000000  100.000000   5.582932
```

All the columns contain numerical data, except for the train column which indicates if the datapoint is to be included in the training set or the test set. Of the numerical data, gleason and svi behave like ordinal/binary variables.

[6]:
```python
# Part b
sns.pairplot(data=prostate)
plt.show()
```



lpsa appears to have the strongest linear relation with lcavol, but also with lweight. We are using the logs because of the choice of the model/regression we might try to fit.

3.1 Train-Test Set

For these data we have already been provided a column to indicate which values should be used for the training set and which for the test set. This is encoded by the values in the `train` column - we can use these columns to separate our data and generate our training data: `X_train` and `y_train` as well as our test data `X_test` and `y_test`.

[6]:
```
# Create train and test data frames
train = prostate.query("train == 'T'").drop('train', axis=1)
test = prostate.query("train == 'F'").drop('train', axis=1)
```

[7]:
```python
# Training data
X_train = train.drop(['lpsa'], axis=1)
y_train = train.lpsa

print('X_train:', X_train.shape)
print('y_train:', y_train.shape)
```

```
X_train: (67, 8)
y_train: (67,)
```

[8]:
```python
# Test data
X_test = test.drop('lpsa', axis=1)
y_test = test.lpsa

print("X_test:", X_test.shape)
print("y_test:", y_test.shape)
```

```
X_test: (30, 8)
y_test: (30,)
```

Let's also fix the random seed to make this notebook's output identical at every run

[9]:
```python
# Fix seed
rng = np.random.seed(0)
```

4 Baseline model

Our first task is to fit a baseline model which we will be able to use as a point of comparison for our subsequent models. A good candidate for this is a simple linear regression model that includes all of our features.

[10]:
```python
# Train a linear regression model
from sklearn.linear_model import LinearRegression
lm = LinearRegression().fit(X_train, y_train)
```

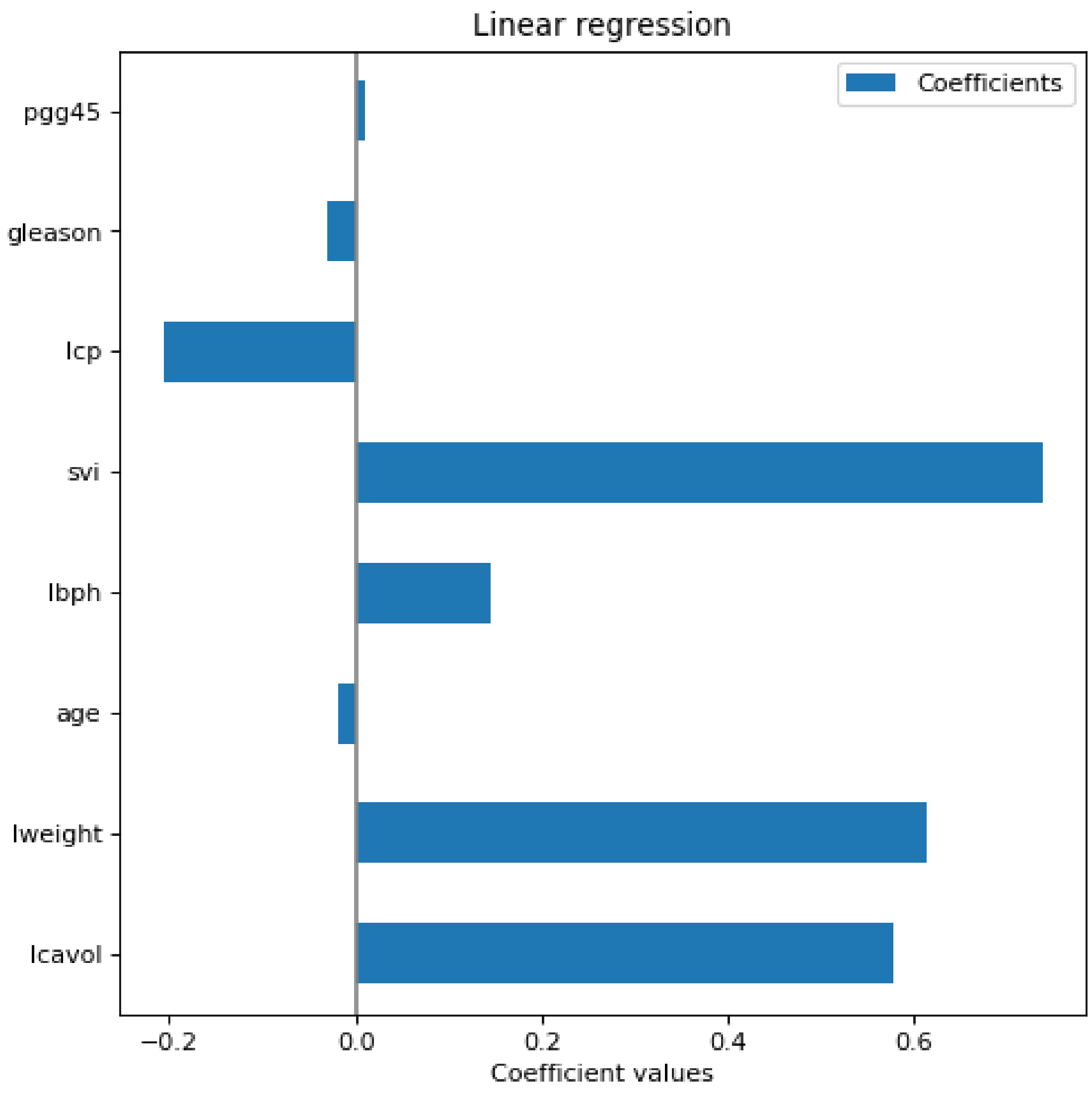
We can extract the coefficients for the model, which correspond to the variables: `lcavol`, `lweight`, `age`, `lbph`, `svi`, `lcp`, `gleason`, and `pgg45` respectively.

[11]:
```python
# Create a data frame of the coeffcients
fe_names = lm.feature_names_in_

coefs = pd.DataFrame(
    np.copy(lm.coef_),
    columns=["Coefficients"],
    index=fe_names,
)

coefs
```

```
# To add intercept
# fe_names = np.append(['intercept'],lm.feature_names_in_)
# coefs = pd.DataFrame(
#     get_coefs(lm),
#     columns=["Coefficients"],
#     index=fe_names,
# )
```

[11]:
```
         Coefficients
lcavol       0.576543
lweight      0.614020
age         -0.019001
lbph         0.144848
svi          0.737209
lcp         -0.206324
gleason     -0.029503
pgg45        0.009465
```

[12]:
```
# Plot of the coefficients
coefs.plot.barh(figsize=(9, 7))
plt.title("Linear regression")
plt.axvline(x=0, color=".5")
plt.xlabel("Coefficient values")
plt.subplots_adjust(left=0.3)
```



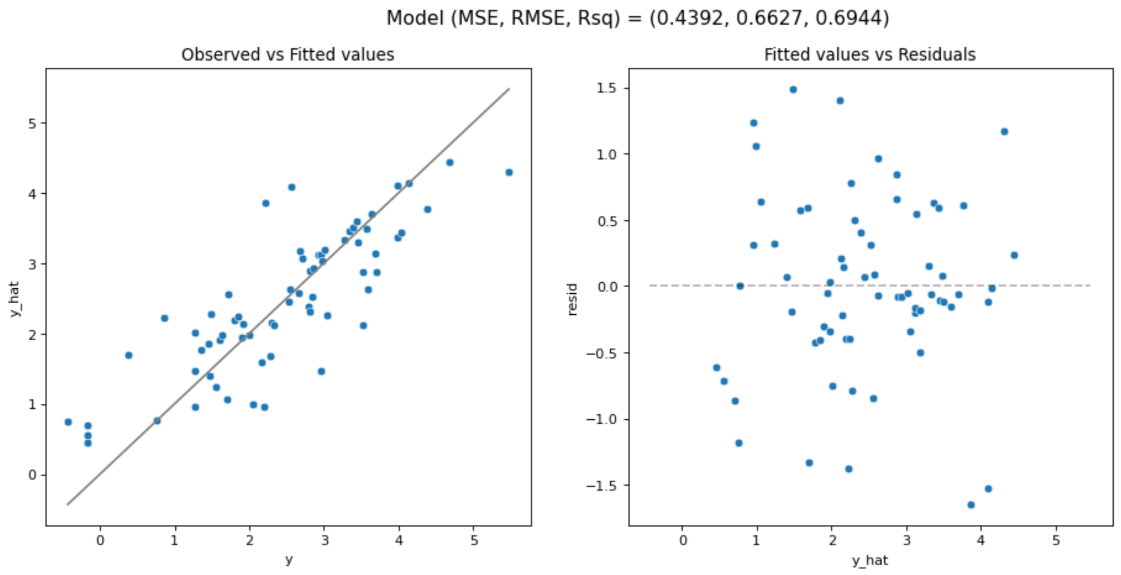
These coefficients have the typical regression interpretation, e.g. for each unit increase in `lcavol` we expect `lpsa` to increase by 0.5765 on average. To evaluate the predictive properities of our model, we will use the `model_fit` helper function.

4.0.1 Exercise 2 (CORE)

Use the `model_fit` function to evaluate both the model fit on the training data and the predictions on the test data.

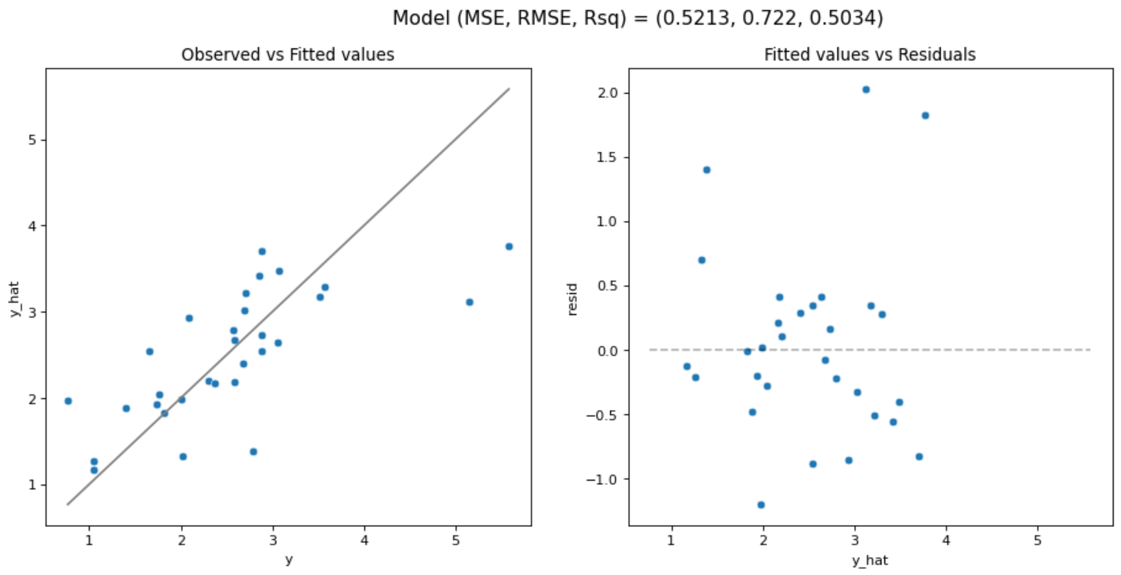
- Based on these plots do you see anything in the fit or residual plot that is potentially concerning?
- Do you expect the MSE on test data to be better or worse than the MSE on the training data?

```
[14]: model_fit(lm, X_train, y_train, True)
```



```
[14]: (0.43919976805833433, 0.662721486039448, 0.6943711796768238)
```

```
[15]: model_fit(lm, X_test, y_test, True)
```



```
[15]: (0.5212740055076021, 0.7219930785731966, 0.503379850238179)
```

Of concern in the plots above we observe of 2 high-leverage outliers in the test data, which may affect the slope of the fit significantly. As expected, we obtain lower MSE and better R^2 on the TRAINING data versus the TEST data.

4.1 Standardization

In subsequent sections we will be exploring the use of the Ridge and Lasso regression models which both penalize larger values of **w**. While not particularly bad, our baseline model had coefficients that ranged from the smallest at 0.0095 to the largest at 0.737 which is about a 78x difference in magnitude. This difference can be made even worse if we were to change the units of one of our features, e.g. changing a measurement in kg to grams would change that coefficient by 1000 which has no effect on the fit of our linear regression model (predictions and other coefficients would be unchanged) but would have a meaningful impact on the estimates given by a Ridge or Lasso regression model, since that coefficient would now dominate the penalty term.

To deal with this issue, the standard approach is to standaridize all features. Additionally, the feature values can now be interpreted as the number of standard deviations each observation is away from that column's mean. Using `sklearn` we can perform this transformation using the `StandardScaler` transformer from the preprocessing submodule.

Keep in mind, that in order to get a realistic idea of the performance of model on the test data, **the mean and standard deviation used to standardize both the training and test sets should be computed from the training data only**. The best way to accomplish this is to include the StandardScaler in a modeling pipeline for your data

4.1.1 Exercise 3 (CORE)

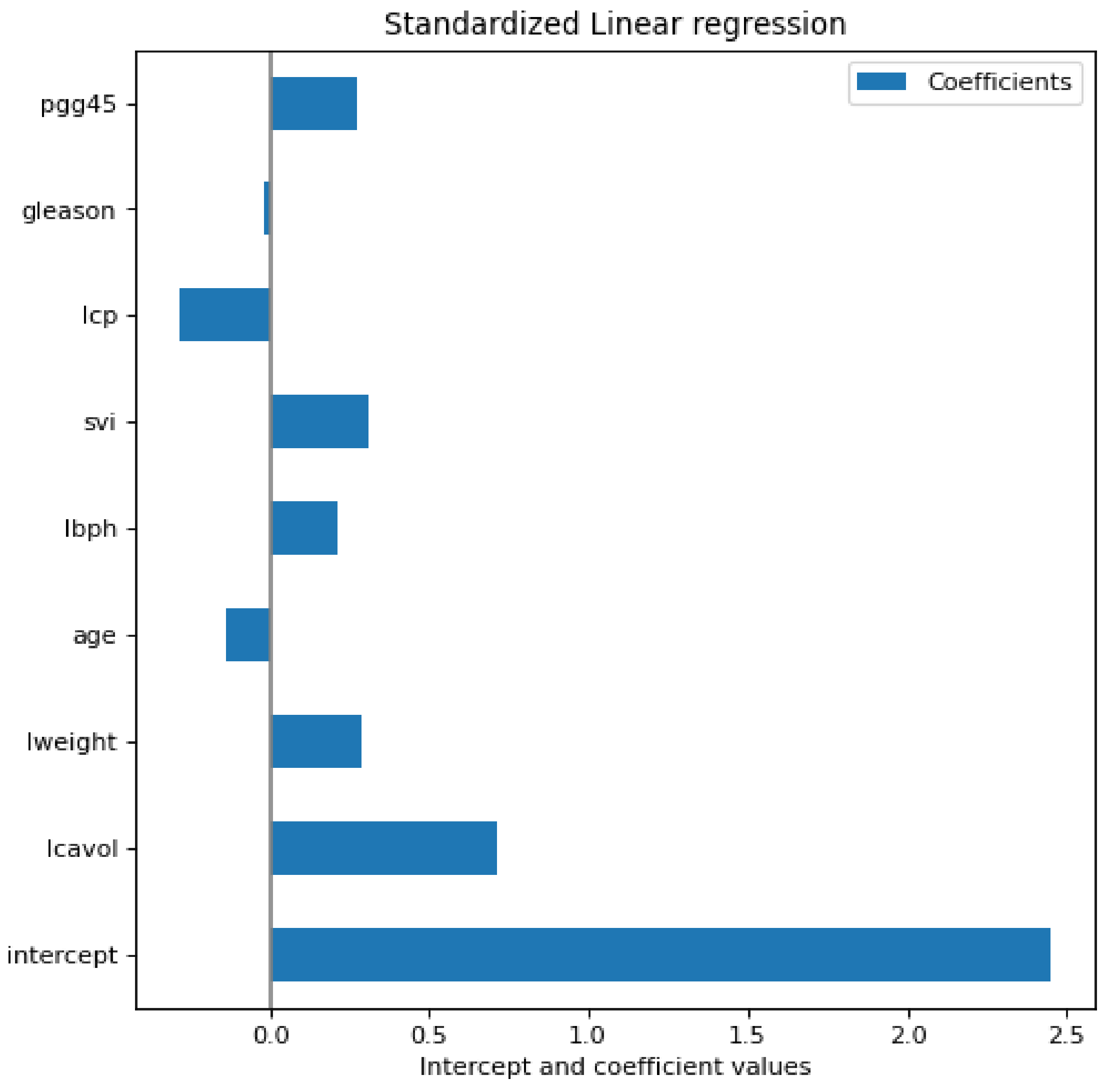
Consider the following pipeline that first standardizes the features before linear regression. Fit the model to the training data. Using this new model what has changed about our model results? Comment on both the model's coefficients as well as its predictive performance. How has the interpretation of coefficients changed?

[16]:
```python
# Linear regression pipeline, including standardization
from sklearn.preprocessing import StandardScaler

lm_s = make_pipeline(
    StandardScaler(),
    LinearRegression()
)

lmst = lm_s.fit(X_train, y_train)

fe_names = np.concatenate([['intercept'], lmst.feature_names_in_])

coefs = pd.DataFrame(
    np.copy(get_coefs(lmst)),
    columns=["Coefficients"],
    index=fe_names,
)
```

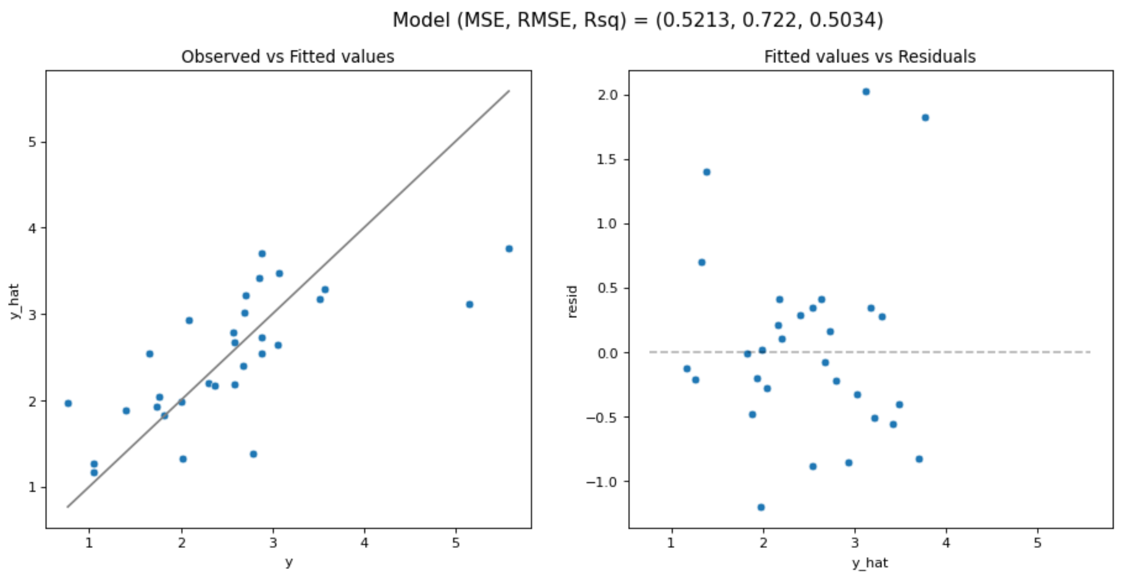
```
coefs
```

[16]:
```
           Coefficients
intercept      2.452345
lcavol         0.711041
lweight        0.290450
age           -0.141482
lbph           0.210420
svi            0.307300
lcp           -0.286841
gleason       -0.020757
pgg45          0.275268
```

[17]:
```
coefs.plot.barh(figsize=(9, 7))
plt.title("Standardized Linear regression")
plt.axvline(x=0, color=".5")
plt.xlabel("Intercept and coefficient values")
plt.subplots_adjust(left=0.3)
```



```
[18]: model_fit(lmst, X_test, y_test, True)
```



```
[18]: (0.5212740055076005, 0.7219930785731955, 0.5033798502381805)
```

While the fit and quality of the model has not changed, the interpretation of the intercept and coefficient values has changed: higher values of the coefficient after using the StandardScaler() does indeed indicate a measure of the sensitivity of the predicted variable with respect to a given feature RELATIVE to the other features in the linear model we are fitting.

Note that by simply adding the `StandardScaler()` step in the pipeline, we have standardized all features, including the binary and ordinal features. This makes interpreting the coefficients of the binary and ordinal features more challenging. Because of this, typically it is preferred to only standardize the numerical variables; in that case, you can use `ColumnTransformer()` to apply standardization only to the numerical variables.

We can check the mean and standard deviation used to standardize the features by accessing the `.mean_` and `.scale_` attributes of the `StandardScaler()`. Notice the values used to transform the binary variable `svi`.

```
[19]: # Extract and print the mean and std used in StandardScaler
ss = StandardScaler().fit(X_train)

fe_names = lmst.feature_names_in_

ss_p = pd.DataFrame(
    np.c_[np.round(ss.mean_,4), np.round(ss.scale_,4)],
    columns=["Mean", "SD"],
    index=fe_names,
)
ss_p
```

[19]:
```
             Mean        SD
lcavol     1.3135    1.2333
lweight    3.6261    0.4730
age       64.7463    7.4460
lbph       0.0714    1.4527
svi        0.2239    0.4168
lcp       -0.2142    1.3902
gleason    6.7313    0.7036
pgg45     26.2687   29.0823
```

[20]:
```
print('After standardizing, the orginal value of 0 for svi is replaced with',np.
 ↪round(-ss.mean_[4]/ss.scale_[4],4) )
print('After standardizing, the orginal value of 1 for svi is replaced with',np.
 ↪round((1-ss.mean_[4])/ss.scale_[4],4) )
```

```
After standardizing, the orginal value of 0 for svi is replaced with -0.5371
After standardizing, the orginal value of 1 for svi is replaced with 1.8619
```

When standardizing all features, if we are interested in interpreting the value of the coefficients of the categorical inputs, we should **unstandardize** the coefficients. Letting $\tilde{\mathbf{x}}$ denote the standardized features and $\hat{\mathbf{w}}$ denote the estimated coeffcients when training with standardized features, we have that:

$$\mathrm{E}[y|\tilde{\mathbf{x}}] = \hat{w}_0 + \hat{w}_1\tilde{x}_1 + ... + \hat{w}_D\tilde{x}_D.$$

Noting that $\tilde{x}_d = (x_d - \bar{x}_d)/s_d$ (where $\bar{x}_d$ and s_d represent the sample mean and standard deviation), we can transform back to the original space:

$$\mathrm{E}[y|\mathbf{x}] = \hat{w}_0 + \hat{w}_1(x_1 - \bar{x}_1)/s_1 + ... + \hat{w}_D(x_D - \bar{x}_D)/s_D.$$

Thus,

$$\mathrm{E}[y|\mathbf{x}] = \left(\hat{w}_0 - \sum_d \bar{x}_d/s_d\right) + \hat{w}_1/s_1x_1 + ... + \hat{w}_D/s_Dx_D.$$

And, the *unstandardized* coefficients are obtain by dividing $\hat{\mathbf{w}}$ by the standard deviations.

4.1.2 Exercise 4 (CORE)

Unstandardize the coefficients and interpret the effect of the binary variable `svi`.

[21]:
```
# Unstandardize the coefficients
unst_coef = coefs[1:]['Coefficients'] / ss_p['SD']

unst_coef
```

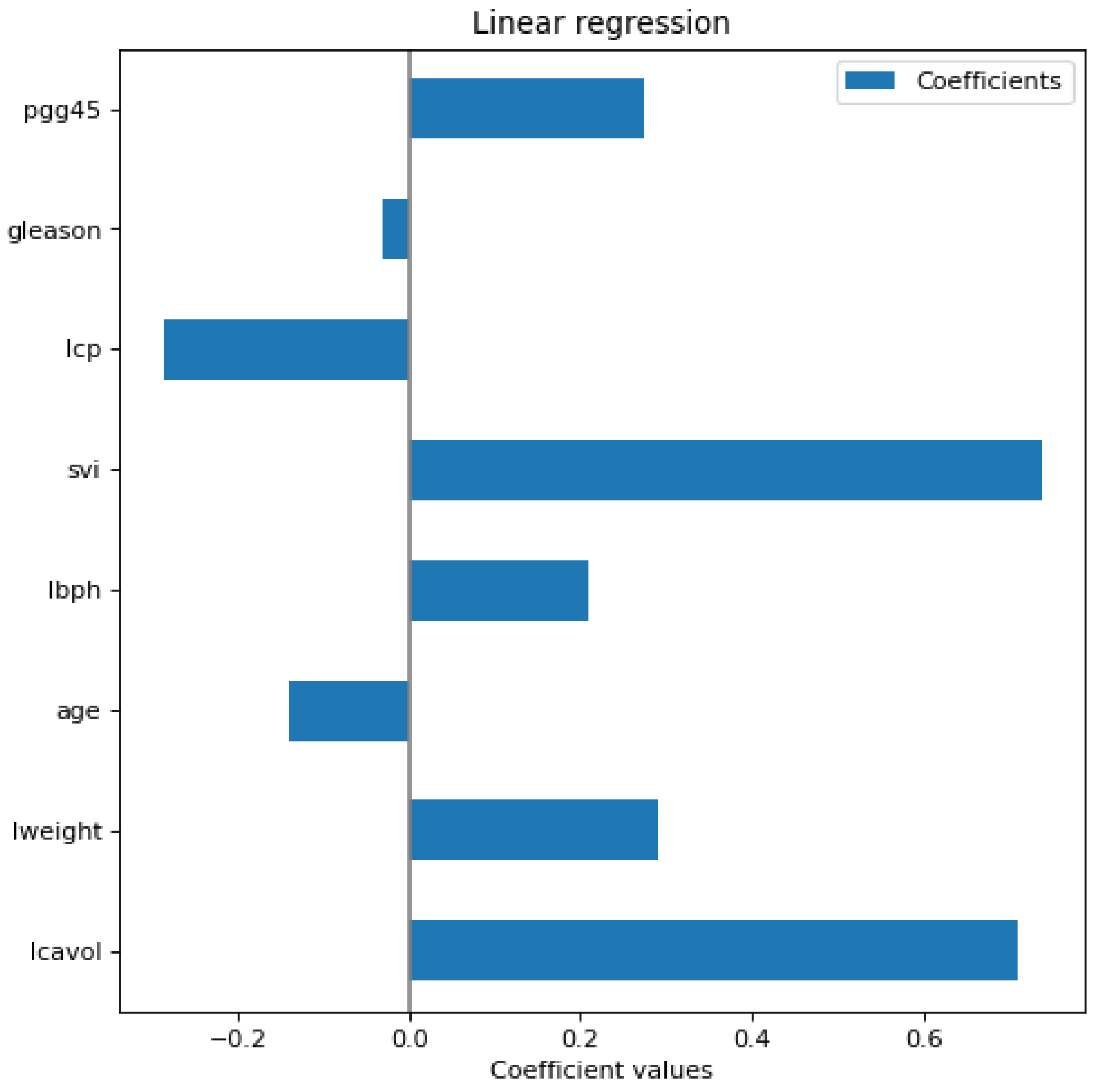
[21]:
```
lcavol      0.576535
lweight     0.614060
age        -0.019001
```

```
lbph       0.144847
svi        0.737285
lcp       -0.206331
gleason   -0.029501
pgg45      0.009465
dtype: float64
```

The unstandardized coefficient associated to svi is 0.737, which is the largest among all the unstandardized coefficients, followed by lcavol and lweight. This indicates the effect of binary variable svi is strongly positive when trying to predict lpsa, svi being TRUE significantly increasing the lpsa prediction in our model in addition to the positive effects of the numerical lcavol & lweight.

Note that in our plot of the coefficients, we want to show the coefficients of the categorical features on the original scale but the coefficients of the numerical features after standardization, for improved interpretation and comparison.

[22]:
```
# In our plot, we want to show the coefficients of the categorical features on␣
  ↪the original scale for improved interpretation
coefs = np.copy(lm_s[1].coef_)
coefs[[4,6]] = coefs[[4,6]]/lm_s['standardscaler'].scale_[[4,6]]

coefs = pd.DataFrame(
    coefs,
    columns=["Coefficients"],
    index=lm_s.feature_names_in_,
)
coefs.plot.barh(figsize=(9, 7))
plt.title("Linear regression")
plt.axvline(x=0, color=".5")
plt.xlabel("Coefficient values")
plt.subplots_adjust(left=0.3)
plt.show()
```



5 Ridge Regression

Ridge regression is a natural extension to linear regression which introduces an ℓ_2 penalty on the coefficients in a standard least squares problem.

The `Ridge` model is provided by the `linear_model` submodule. Note that the penalty parameter (referred to as λ in the lecture notes) is called `alpha` is sklearn, and, as discussed in lectures, this parameter crucially determines the amount of shrinkage towards zero and the weight of the ℓ_2 penalty.

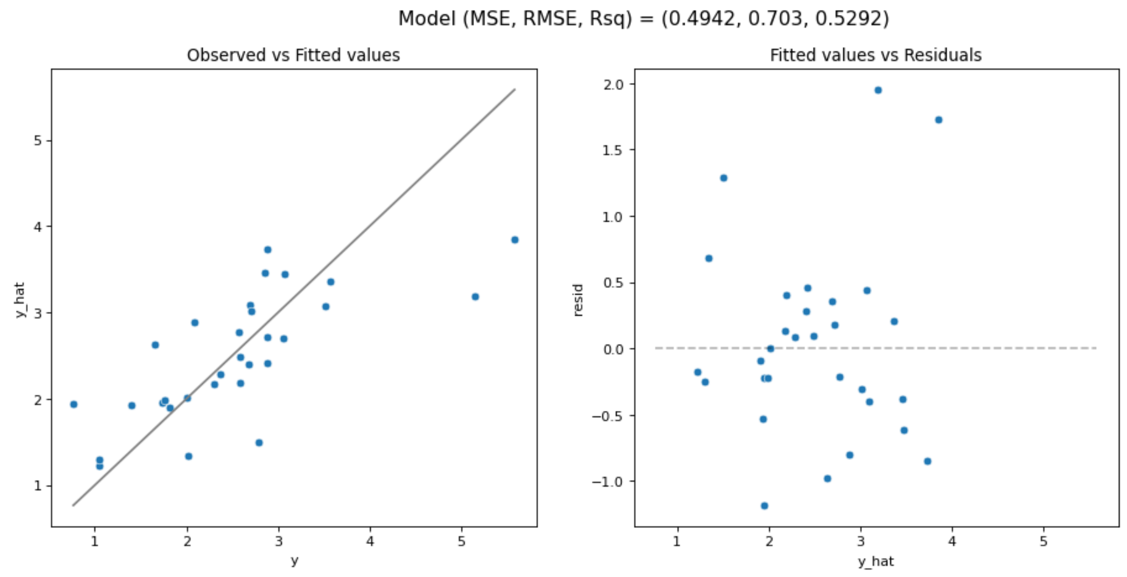
After defining the ridge regression model via, e.g. `Ridge(alpha = 1)`, the usual methods can be called, such as `.fit()` to fit the model and `.predict()` to make predictions.

As for the `LinearRegression()`, after fitting, the intercept and coefficients are stored separately in the attributes `.intercept_` and `.coef_`. In Ridge, this is helpful as it highlights how the penalty is only applied to the coefficient (i.e. we do not want to shrink the intercept).

Let's start by fitting a ridge regression model with $\alpha = 1$.

[23]:
```
from sklearn.linear_model import Ridge
```

[24]:
```
# Selected alpha value
alpha_val = 5

# Ridge pipeline
r = make_pipeline(
    StandardScaler(),
    Ridge(alpha = alpha_val)
).fit(X_train, y_train)

model_fit(r, X_test, y_test, plot = True)
```



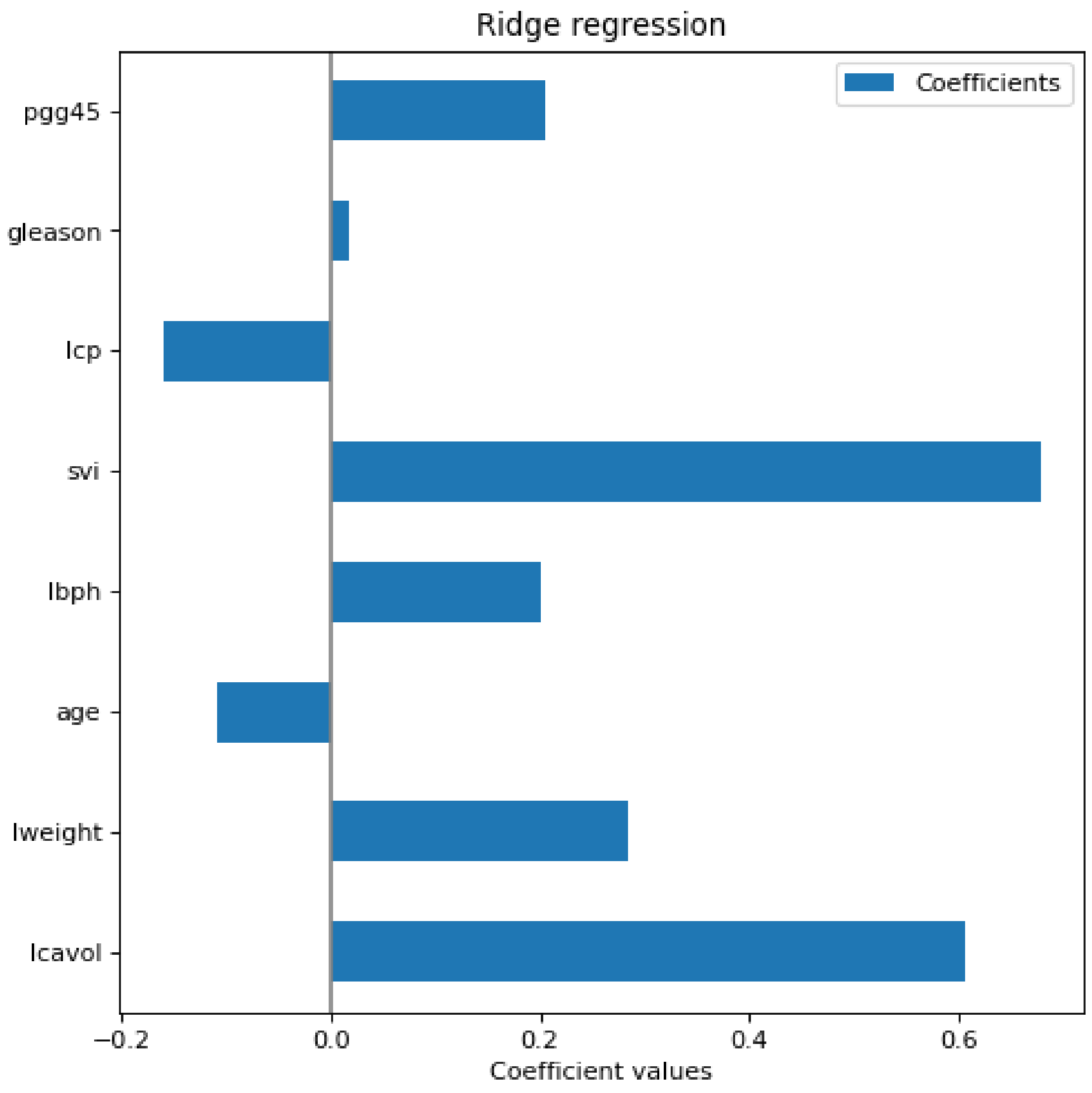
[24]:
```
(0.4941537530376167, 0.7029607051874356, 0.5292174398761034)
```

[25]:
```
# Create dataframe with coefficents, and unstandardize the binary coeffcients
rcoefs = np.copy(r[-1].coef_)
rcoefs[[4,6]] = rcoefs[[4,6]]/r[0].scale_[[4,6]]

rcoefs_ = pd.DataFrame(
    rcoefs,
    columns=["Coefficients"],
    index=r.feature_names_in_,
)
```

```
rcoefs_
```

[25]:
```
         Coefficients
lcavol       0.606103
lweight      0.284181
age         -0.109822
lbph         0.200299
svi          0.678658
lcp         -0.160521
gleason      0.017698
pgg45        0.205013
```

[26]:
```
# Plot of the coefficients
rcoefs_.plot.barh(figsize=(9, 7))
plt.title("Ridge regression")
plt.axvline(x=0, color=".5")
plt.xlabel("Coefficient values")
plt.subplots_adjust(left=0.3)
plt.show()
```



5.0.1 Exercise 5 (CORE)

Adjust the value of `alpha` in the cell above and rerun it. Qualitatively, how does the model fit change as alpha changes? How does the MSE change?

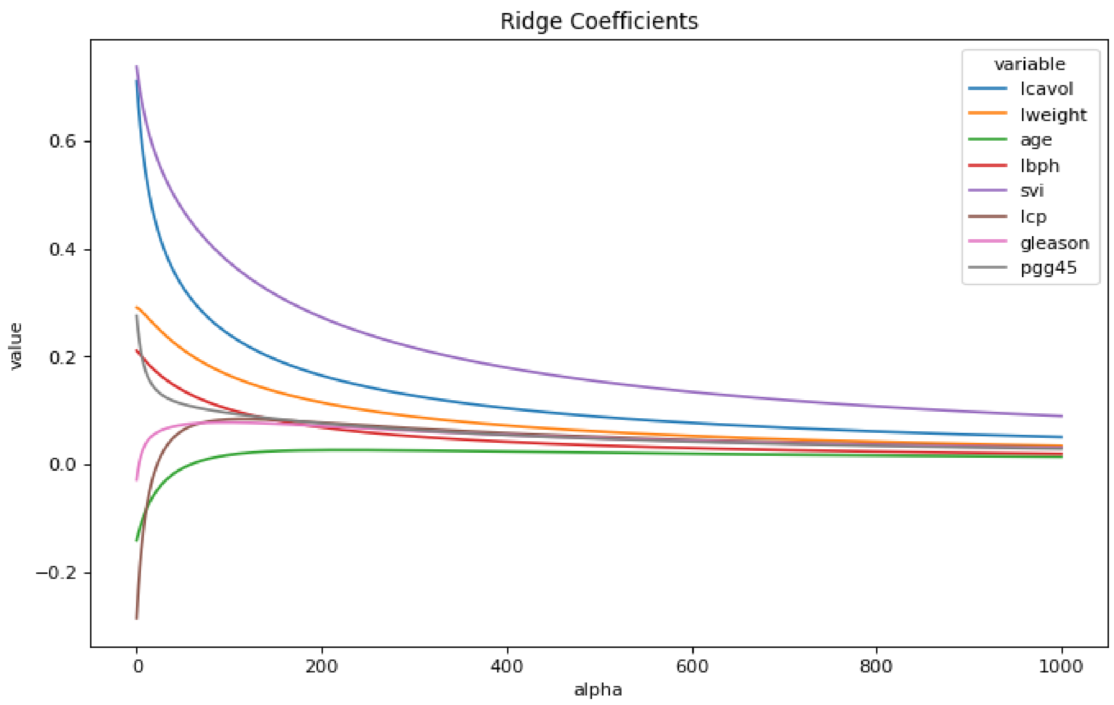
I have changed the value of alpha to alpha = 5. Qualitatively, the MSE has decreased if we perform this change compared to alpha = 1, and the fit has improved as we get a better R^2 value.

5.1 Solution path: Ridge coeffcients as a function of α

A useful way of examining the behavior of Ridge regression models is to plot the **solution path** of the coefficents $\mathbf{w}$ as a function of the penalty parameter α. Since Ridge regression is equivalent to linear regression when $\alpha = 0$, we can see that as we increase the value of α, we are shrinking all of the coefficients in $\mathbf{w}$ towards zero asymptotically α approaches infinity.

[27]:
```
# Grid of alpha values
alphas = np.logspace(-2, 3, num=200) # from 10^-2 to 10^3

ws = [] # Store coefficients
mses_train = [] # Store training mses
mses_test = [] # Store test mses

for a in alphas:
    m = make_pipeline(
        StandardScaler(),
        Ridge(alpha=a)
    ).fit(X_train, y_train)

    w_temp = np.copy(m[1].coef_)
    w_temp[[4,6]] = w_temp[[4,6]]/m[0].scale_[[4,6]]
    ws.append(w_temp)
    mses_train.append(mean_squared_error(y_train, m.predict(X_train)))
    mses_test.append(mean_squared_error(y_test, m.predict(X_test)))
```

[28]:
```
# Create a data frame for plotting
sol_path = pd.DataFrame(
    data = ws,
    columns = X_train.columns # Label columns w/ feature names
).assign(
    alpha = alphas,
).melt(
    id_vars = ('alpha')
)

# Plot solution path of the weights
plt.figure(figsize=(10,6))
ax = sns.lineplot(x='alpha', y='value', hue='variable', data=sol_path)
ax.set_title("Ridge Coefficients")
plt.show()
```



5.1.1 Exercise 6 (CORE)

Based on this plot, which variable(s) seem to be the most important for predicting `lpsa`?

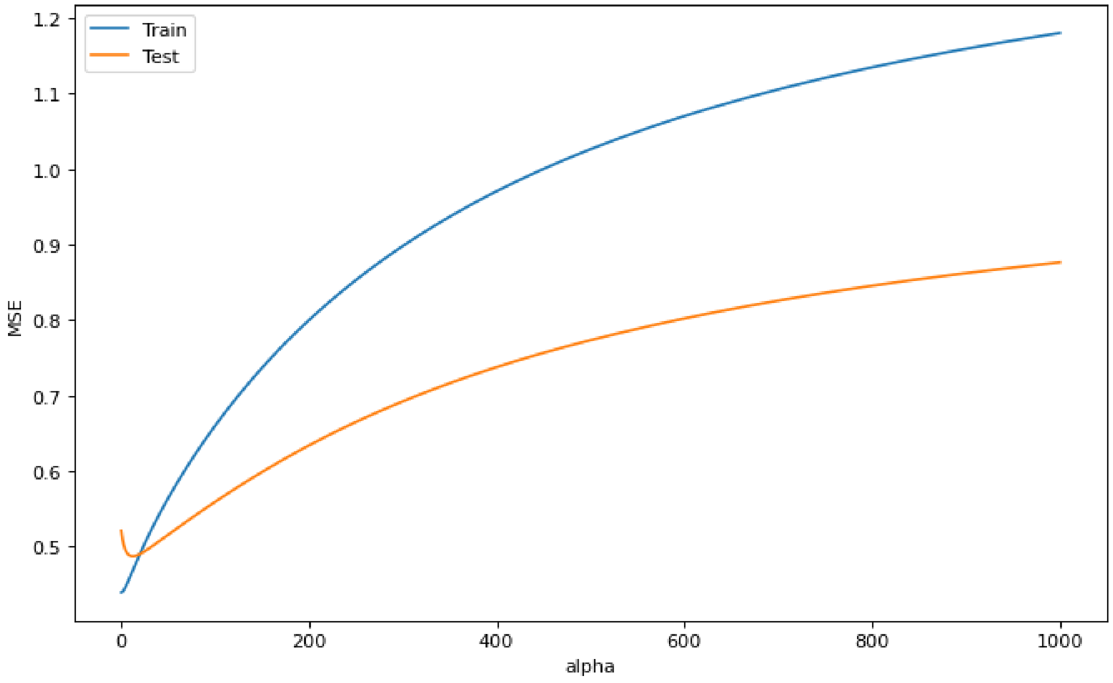
Based on the plot above, we note that the variables which seem most important in predicting are lpsa are: primarily, svi, and secondarily lcavol.

5.1.2 Exercise 7 (CORE)

Run the code below to also plot both the training and test MSE as a function of α. What do you notice about the MSE as we increase α? Which value of α seems better regarding the changes on training and testing MSE values?

[29]:
```
# Path of MSE as function of alpha
mses_path = pd.DataFrame(
    {'alpha': alphas, 'Train': np.asarray(mses_train), 'Test': np.
 ↪asarray(mses_test)}).melt(
    id_vars = ('alpha')
)

# Plot MSE path
plt.figure(figsize=(10,6))
ax = sns.lineplot(x='alpha', y='value', hue='variable', data=mses_path)
ax.set_ylabel("MSE")
# To remove legend title
```

```
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles=handles[0:], labels=labels[0:])
plt.show()
```



For the training data, increasing the alpha always increases the MSE, however when we look at the test data, we see that the optimal/minimal MSE is obtained for a value of alpha around 5. Increasing alpha further results in overfitting.

5.2 Tuning the penalty parameter with cross-validation

We see that the value of α crucially determines the performance of the ridge regression model. While `RidgeRegression()` uses the default value of `alpha=1`, this should never be used in practice. Instead, this parameter can be tuned using cross-validation.

As with the polynomial models from last week, we can use `GridSearchCV` to employ k-fold cross validation to determine an optimal α. Remember, you can use the method `.get_params()` on your pipeline to list the parameters names to specify in `GridSearchCV`.

[30]:
```
# Grid of tuning parameters
alphas = np.linspace(0, 15, num=151)

#Pipeline
m = make_pipeline(
        StandardScaler(),
        Ridge())
# To get the parameter name for grid search
# m.get_params()

# CV strategy
cv = KFold(5, shuffle=True, random_state=1234)

# Grid search
gs = GridSearchCV(m,
    param_grid={'ridge__alpha': alphas},
    cv=cv,
    scoring="neg_mean_squared_error")
gs.fit(X_train, y_train)
```

[30]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=1234, shuffle=True),
             estimator=Pipeline(steps=[('standardscaler', StandardScaler()),
                                       ('ridge', Ridge())]),
             param_grid={'ridge__alpha': array([ 0. ,  0.1,  0.2,  0.3,  0.4,
0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
        2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,
        3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4…
        7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,  8.7,
        8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5,  9.6,  9.7,  9.8,
        9.9, 10. , 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9,
       11. , 11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8, 11.9, 12. ,
       12.1, 12.2, 12.3, 12.4, 12.5, 12.6, 12.7, 12.8, 12.9, 13. , 13.1,
       13.2, 13.3, 13.4, 13.5, 13.6, 13.7, 13.8, 13.9, 14. , 14.1, 14.2,
       14.3, 14.4, 14.5, 14.6, 14.7, 14.8, 14.9, 15. ])},
             scoring='neg_mean_squared_error')
```

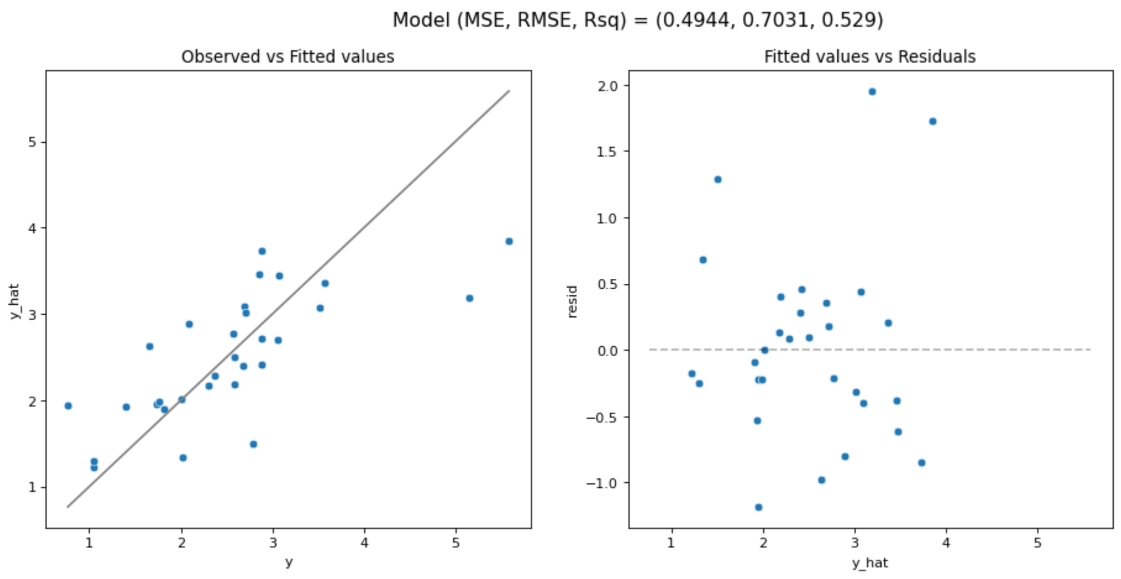
Note that we are passing `sklearn.model_selection.KFold(5, shuffle=True, random_state=1234)` to the `cv` argument rather than leaving it to its default. This is because, while not obvious, the prostate data is structured (sorted by `lpsa` value) and this way we are able to ensure that the folds are properly shuffled. Failing to do this causes *very* unreliable results from the cross validation process.

Once fit, we can examine the results to determine what value of α was chosen as well as examine the results of cross validation.

[31]:
```
print(gs.best_params_)
print(-gs.best_score_)
```

```
{'ridge__alpha': 4.9}
0.7066011634399013
```

[32]:
```
model_fit(gs.best_estimator_, X_test, y_test, plot=True)
```



```
[32]: (0.4944100876726734, 0.7031430065588887, 0.528973228686775)
```

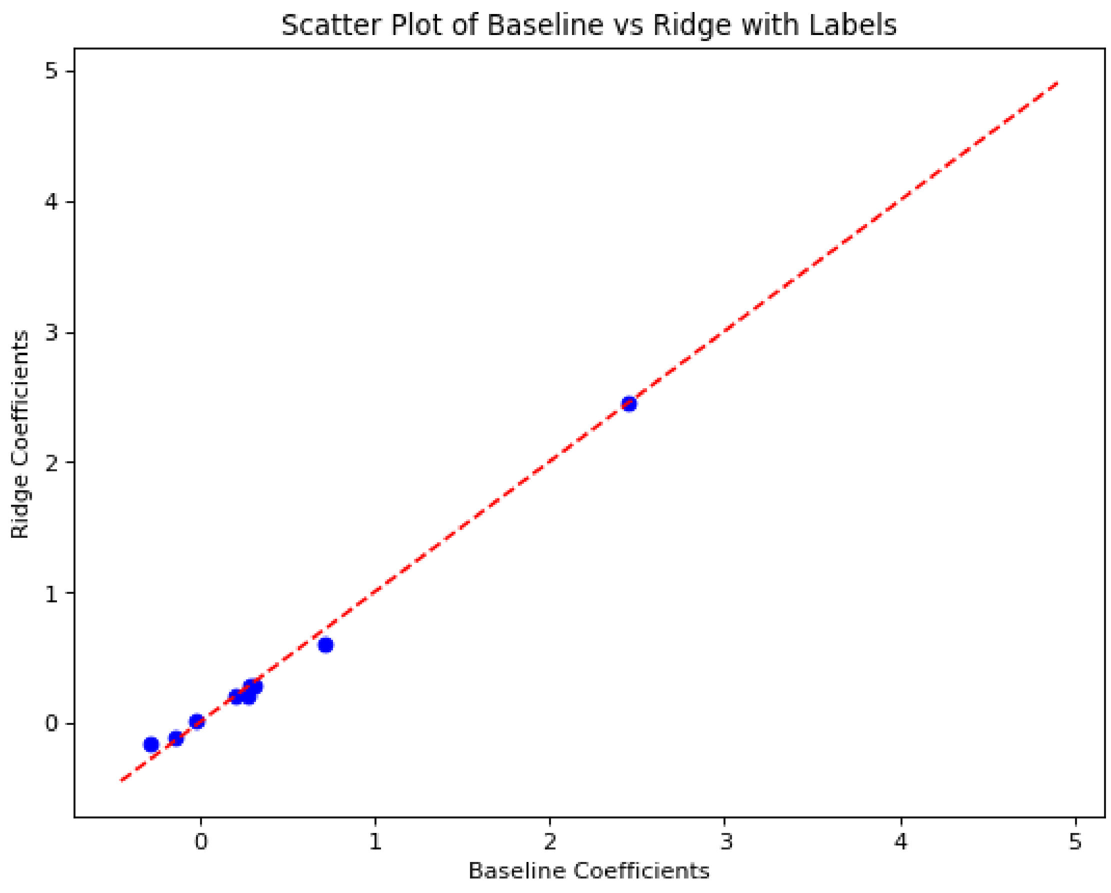
5.2.1 Exercise 8 (CORE)

- How does this model compare to the performance of our baseline model? Is it better or worse?
- How do the model coefficients for this model compare to the baseline model? To answer this plot the coefficients for the baseline model against the coefficients for the ridge model. Are they always higher or lower? Now, use `np.linalg.norm` to compute the ℓ_2 norm of the coeffcients for both models and comment on the results.

It is clear that the previously obtained model is BETTER than the baseline model we used originally, because its R^2 is better: 0.52 vs. 0.5.

```
[33]: fe_names = np.concatenate([['intercept'], lm.feature_names_in_])

baseline_coefs = pd.DataFrame(
    np.copy(get_coefs(lmst)),
    columns=["Coefficients"],
    index=fe_names,
)

ridge_coefs = pd.DataFrame(
    np.copy(get_coefs(gs.best_estimator_)),
    columns=["Coefficients"],
    index=fe_names,
)

coefs = pd.concat([baseline_coefs, ridge_coefs], axis=1)
```

```
# Create scatter plot
plt.figure(figsize=(8, 6))
plt.scatter(coefs.iloc[:, 0], coefs.iloc[:, 1], color='blue')

x_vals = np.linspace(min(coefs.iloc[:, 0] + coefs.iloc[:, 1]), max(coefs.iloc[:
 ↪, 0] + coefs.iloc[:, 1]), 100)
plt.plot(x_vals, x_vals, color='red', linestyle='--', label="y = x")


# Add axis labels and title
plt.xlabel("Baseline Coefficients")
plt.ylabel("Ridge Coefficients")
plt.title("Scatter Plot of Baseline vs Ridge with Labels")

# Show plot
plt.show()
```



[34]:
```
print(np.linalg.norm(ridge_coefs['Coefficients'][1:]))
print(np.linalg.norm(baseline_coefs['Coefficients'][1:]))
```

```
0.807318313759715
0.9524432722339893
```

As we can see sometimes the Ridge coefficients are larger than the baseline ones and viceversa, however they are all relatively close to each other, lying close to x = y. Meanwhile, the l2 norm for the baseline model is 0.95, while for the Ridge model it is 0.80, indicating that the Ridge coefficients are closer among each other, as expected as that is the point behind the correction.

As we saw last week, it is also recommend to plot the CV scores. Although the grid search may report a best value for the parameter corresponding to the maximum CV score (e.g. min CV MSE), if the curve is relatively flat around the minimum, we may prefer the simpler model.

Recall from last week that we can access the cross-validated scores (along with other results for each split) in the attribute `cv_results_`.

[35]:
```
cv_results = pd.DataFrame(gs.cv_results_)
cv_results.head()
```

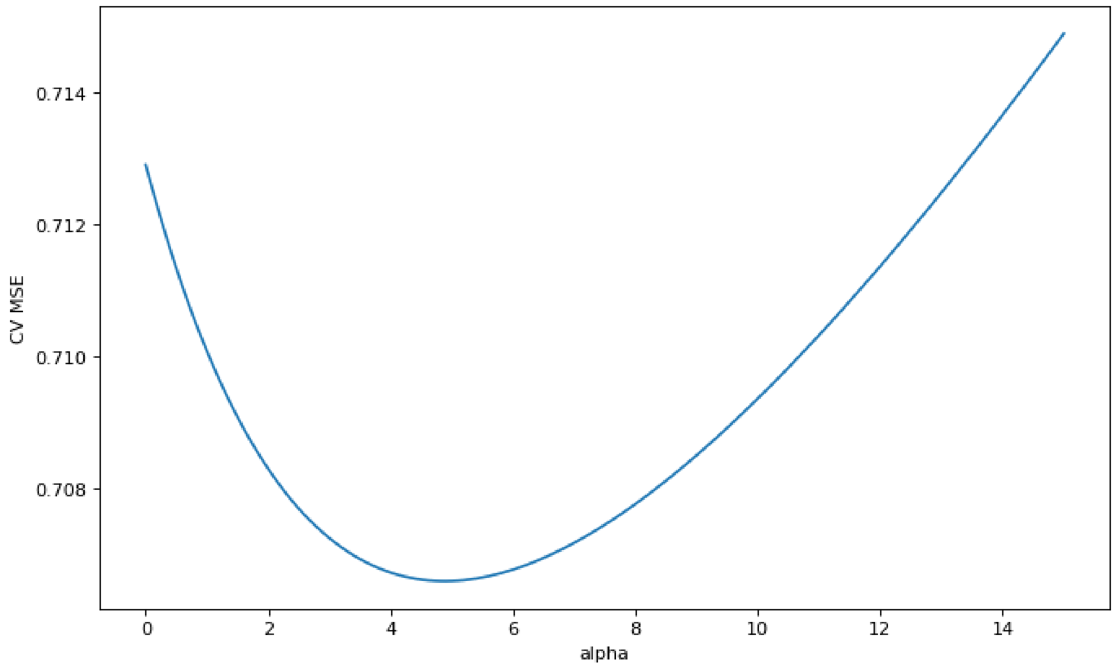
[35]:
```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       0.003351      0.000333         0.001473        0.000074
1       0.003003      0.000150         0.001348        0.000024
2       0.002902      0.000046         0.001338        0.000031
3       0.003818      0.001588         0.001371        0.000065
4       0.002897      0.000019         0.001309        0.000012

   param_ridge__alpha                                 params  \
0                 0.0                 {'ridge__alpha': 0.0}
1                 0.1                 {'ridge__alpha': 0.1}
2                 0.2                 {'ridge__alpha': 0.2}
3                 0.3  {'ridge__alpha': 0.30000000000000004}
4                 0.4                 {'ridge__alpha': 0.4}

   split0_test_score  split1_test_score  split2_test_score  split3_test_score  \
0          -0.933523          -0.684212          -0.732690          -0.424796
1          -0.934023          -0.685383          -0.733913          -0.422857
2          -0.934523          -0.686515          -0.735152          -0.420989
3          -0.935022          -0.687609          -0.736407          -0.419188
4          -0.935519          -0.688668          -0.737675          -0.417453

   split4_test_score  mean_test_score  std_test_score  rank_test_score
0          -0.789296        -0.712903        0.166571              134
1          -0.786632        -0.712561        0.167126              130
2          -0.783993        -0.712234        0.167674              126
3          -0.781380        -0.711921        0.168215              123
4          -0.778793        -0.711622        0.168750              119
```

In particular, let's examining the `mean_test_score` and the `split#_test_score` keys since these are used to determine the optimal α.

In the code below we extract these data into a data frame by selecting our columns of interest along with the α values used (and transform negative MSE values into positive values).

[36]:
```
# Extract only mean and split scores
cv_mse = pd.DataFrame(
    data = gs.cv_results_
).filter(
    # Extract the split#_test_score and mean_test_score columns
    regex = '(split[0-9]+|mean)_test_score'
).assign(
    # Add the alphas as a column
    alpha = alphas
)

cv_mse.update(
    # Convert negative mses to positive
    -1 * cv_mse.filter(regex = '_test_score')
)
```

[37]:
```
# Plot CV MSE
plt.figure(figsize=(10,6))
ax = sns.lineplot(x='alpha', y='mean_test_score', data=cv_mse)
ax.set_ylabel('CV MSE')
plt.show()
```

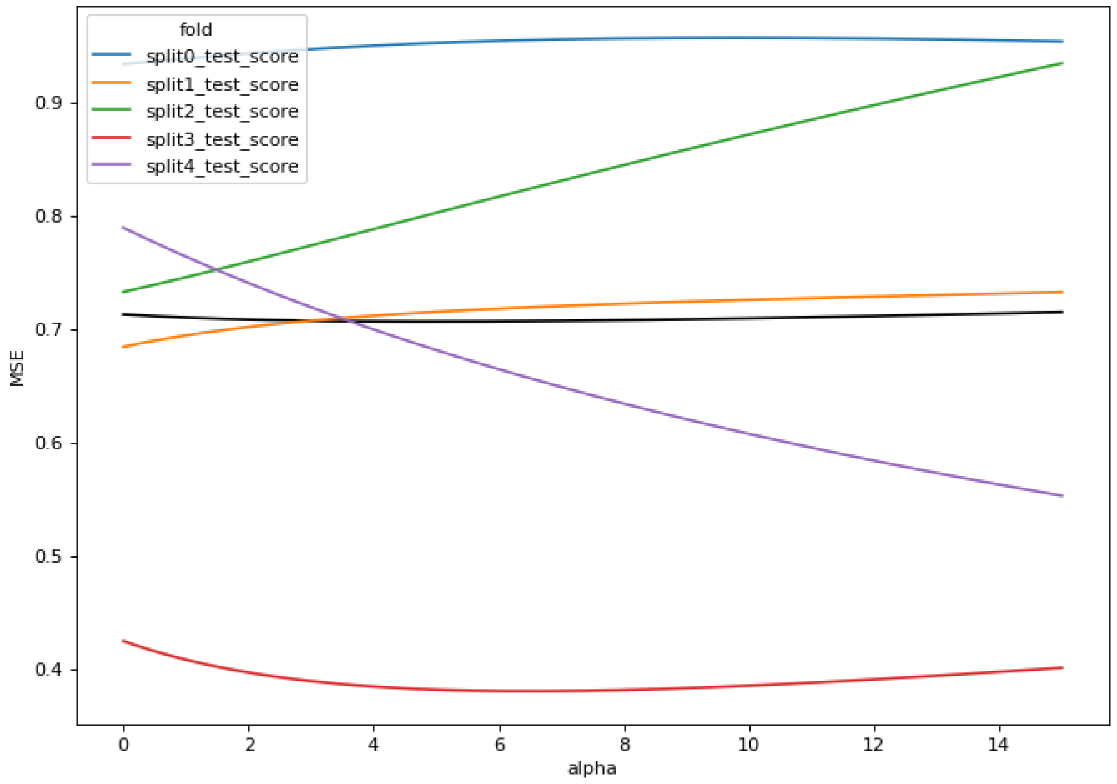


This plot shows that the value of $\alpha = 4.9$ corresponds to the minimum of this curve. However, this plot gives us an overly confident view of this particular value of α. Specifically, if instead of just plotting the mean MSE across all of the validation sets, we also examine the MSE for each fold individually and the corresponding optimal value of α, we see that there is a lot of noise in the MSE and we should take the value $\alpha = 4.9$ with a grain of salt.

5.2.2 Exercise 9 (CORE)

Run the code below to plot the MSE for each validation set in the 5-fold cross validation. Why do you think that our cross validation results are unstable?

[38]:
```
# Reshape the data frame for plotting
d = cv_mse.melt(
    id_vars=('alpha','mean_test_score'),
    var_name='fold',
    value_name='MSE'
)

# Plot the validation scores across folds
plt.figure(figsize=(10,7))
sns.lineplot(x='alpha', y='MSE', color='black', errorbar=None, data = d)  #␣
  ↪Plot the mean MSE in black.
sns.lineplot(x='alpha', y='MSE', hue='fold', data = d) # Plot the curves for␣
  ↪each fold in different colors
plt.show()
```



Unstability occurs due to several factors, in particular the small dataset size and potential class imbalance.

Note: Due to the importance of tuning the value of α in ridge regression, sklearn provides a function called `RidgeCV` which combines `Ridge` with `GridSearchCV`. However, we will avoid using this function for two reasons:

- it does not allow us to account for additional steps in our pipeline such as standardization when carrying out cross validation, resulting in data leakage
- it only allows storing all results of the cross-validation in the attribute `.cv_results_` in the case of the default leave-one-out cross validation, with option `store_cv_results=True`. So, if you want to access all results and use a cross-validation strategy other than leave-one-out, you will need to use `GridSearchCV`.

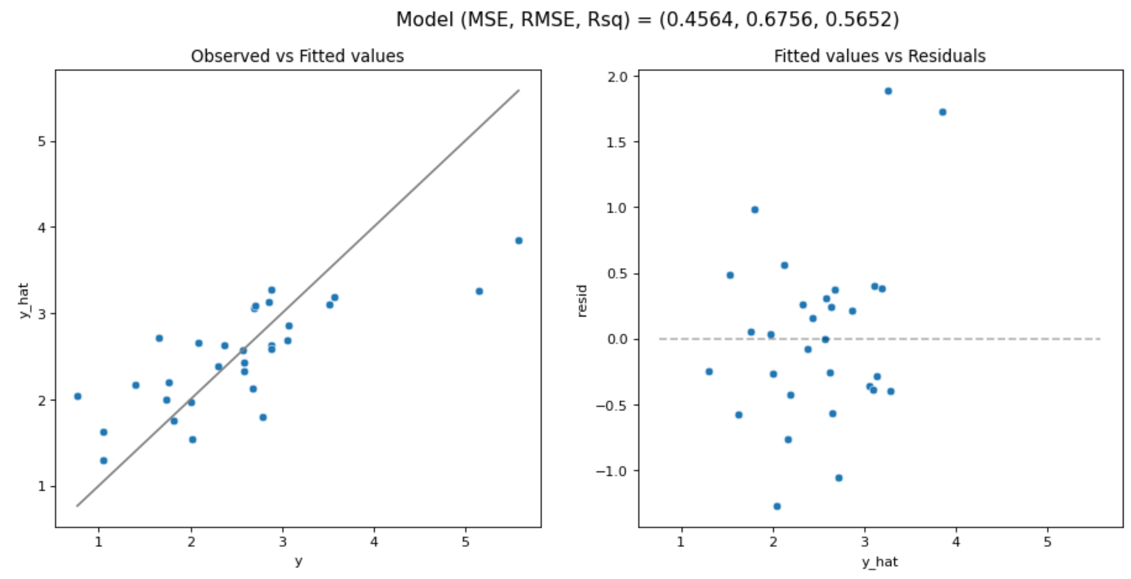
6 Lasso Regression

We saw that ridge regression with a wise choice of α can outperform our baseline linear regression. We can now investigate if lasso can yield a more accurate or interpretable solution. Recall that lasso uses an ℓ_1 penalty on the coefficients, as opposed to the ℓ_2 penalty of ridge.

The `Lasso` model is also provided by the `linear_model` submodule and similarly requires the choice of the tuning parameter `alpha` to determine the weight of the ℓ_1 penalty.

Try running the code below with different values of α to see how it effects sparsity in the coefficients and model performance.

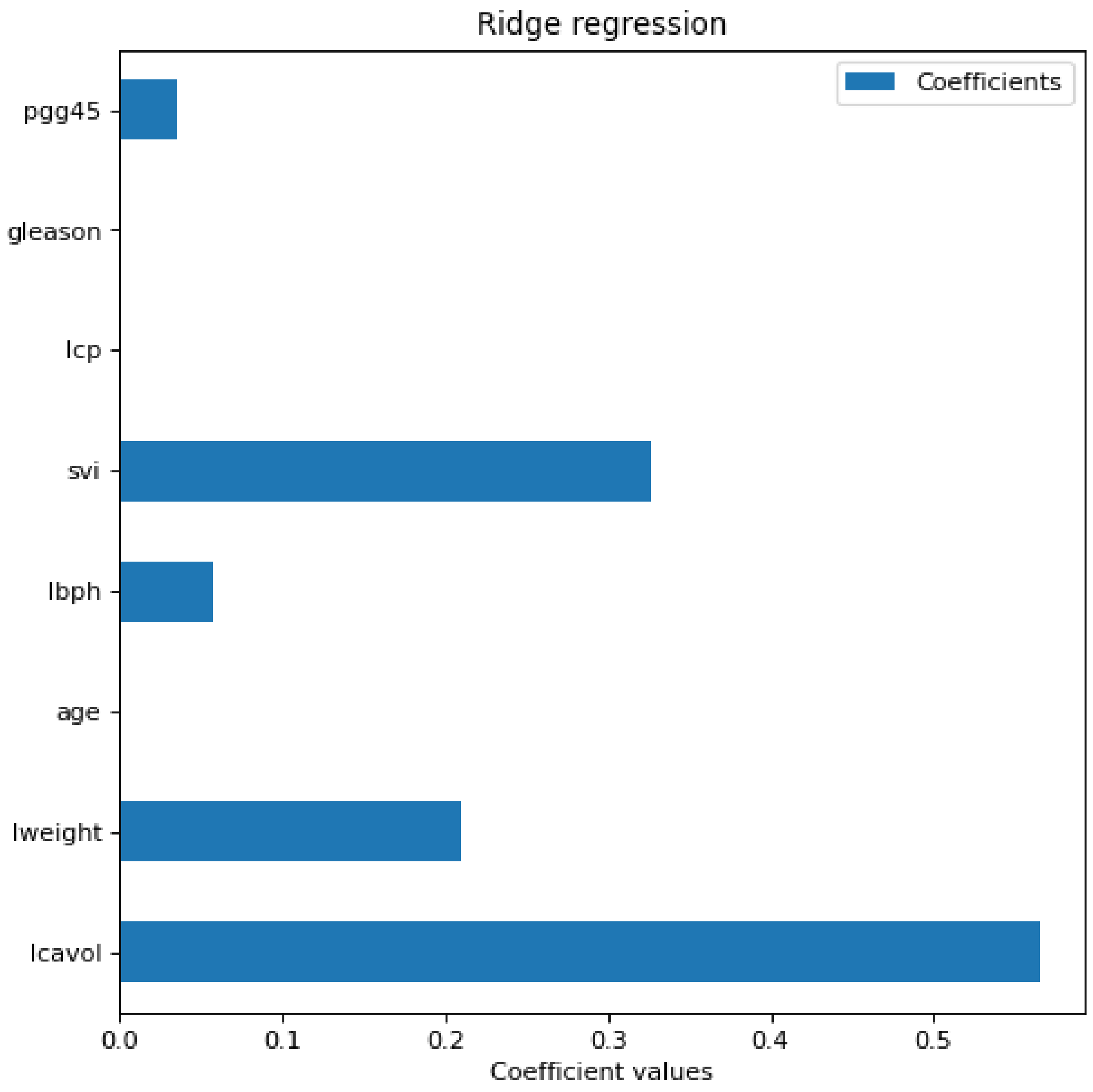
```
[39]: from sklearn.linear_model import Lasso

      # Selected alpha value
      alpha_val = 0.15

      # Lasso pipeline
      l = make_pipeline(
          StandardScaler(),
          Lasso(alpha = alpha_val)
      ).fit(X_train, y_train)

      model_fit(l, X_test, y_test, plot = True)
```



```
[39]: (0.4564332868640818, 0.6755984657058376, 0.565153902009766)
```

```
[40]: # Create dataframe with coefficents, and unstandardize the binary coeffcients
      lcoefs = np.copy(l[-1].coef_)
      lcoefs[[4,6]] = lcoefs[[4,6]]/l[0].scale_[[4,6]]

      lcoefs_ = pd.DataFrame(
          lcoefs,
          columns=["Coefficients"],
          index=r.feature_names_in_,
      )

      # Plot of the coefficients
```

```
lcoefs_.plot.barh(figsize=(9, 7))
plt.title("Ridge regression")
plt.axvline(x=0, color=".5")
plt.xlabel("Coefficient values")
plt.subplots_adjust(left=0.3)
plt.show()
```



6.0.1 Exercise 10 (CORE)

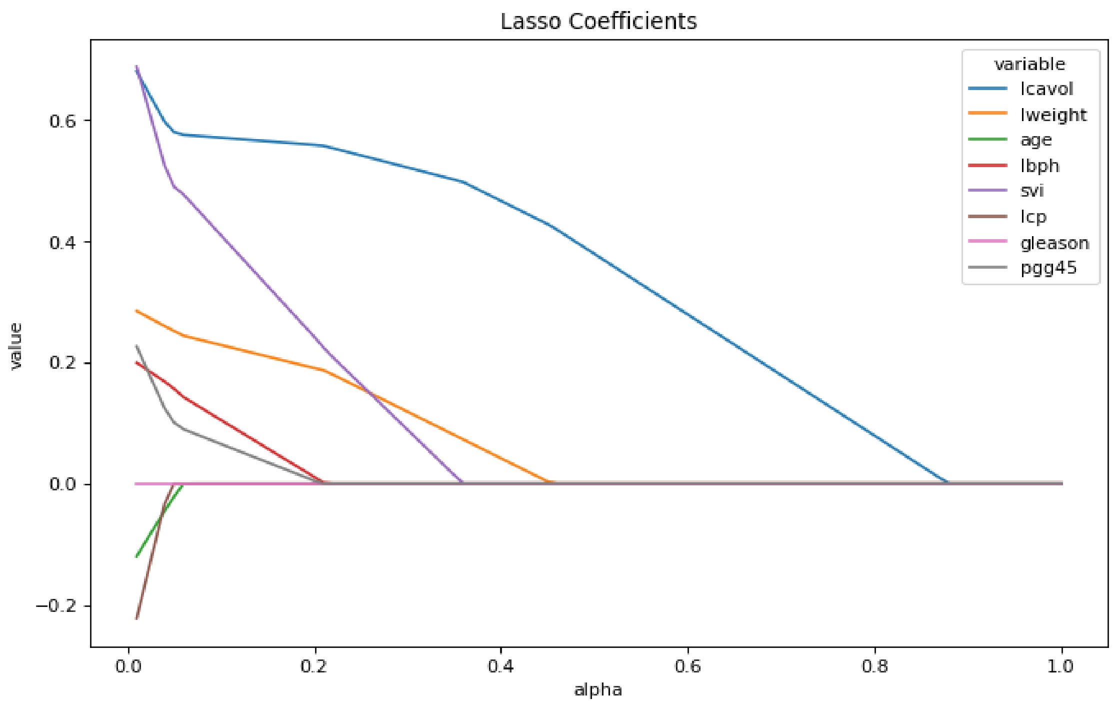
a) Plot the solution path of the coefficients as a function of α.

b) How does this differ between the solution path for Ridge for large α? for small α?

c) Which variable seems to be the most important for predicting `lpsa`?

Note that $\alpha = 0$ causes a warning due to the fitting method (coordinate descent) not converging well without regularization (the ℓ_1 penalty here). So, the grid of α values needs to start at some small positive constant.

[43]:
```
# Part a: Compute and plot the solution path
alphas = np.linspace(0.01, 1, num=100) #We need smaller values of alpha in the␣
  ↪grid

ws = [] # Store coefficients
mses_train = [] # Store training mses
mses_test = [] # Store test mses

for a in alphas:
    m = make_pipeline(
        StandardScaler(),
        Lasso(alpha=a)
    ).fit(X_train, y_train)

    w_temp = np.copy(m[1].coef_)
    w_temp[[4,6]] = w_temp[[4,6]]/m[0].scale_[[4,6]]
    ws.append(w_temp)
    mses_train.append(mean_squared_error(y_train, m.predict(X_train)))
    mses_test.append(mean_squared_error(y_test, m.predict(X_test)))

# Create a data frame for plotting
sol_path = pd.DataFrame(
    data = ws,
    columns = X_train.columns # Label columns w/ feature names
).assign(
    alpha = alphas,
).melt(
    id_vars = ('alpha')
)

# Plot solution path of the weights
plt.figure(figsize=(10,6))
ax = sns.lineplot(x='alpha', y='value', hue='variable', data=sol_path)
ax.set_title("Lasso Coefficients")
plt.show()
```



This differs significantly from the solution path from Ridge as the coefficients converge to zero much faster than in the Ridge case. In fact all coefficients converge to zero before the value of alpha hits 1. We again note that the most significant predictors are lcavol, svi and lweight, with lcavol being the most significant in the Lasso case.

6.1 Tuning the Lasso penalty parameter

Again, we can use the `GridSearchCV` function to tune our Lasso model and optimize the α hyperparameter (or use LassoCV, which combines `Lasso` and `GridSearchCV` but we will focus on the former).

6.1.1 Exercise 11 (CORE)

a) Use `GridSearchCV` to find the optimal value of α.

b) Plot the CV MSE and MSE for each fold. Comment on the stability and uncertainty of α across the different folds.

c) Which variables are included with this optimal value of α?

[45]:
```
# Part a: optimal alpha

# Grid of tuning parameters
alphas = np.linspace(0.01, 1, num=100)

#Pipeline
```

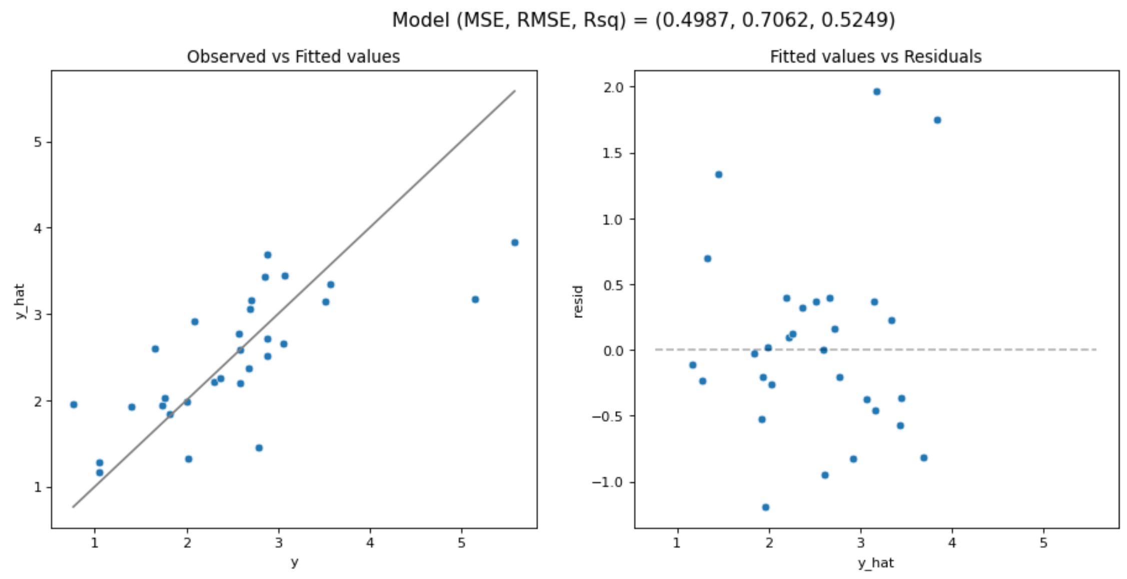
```
m = make_pipeline(
        StandardScaler(),
        Lasso())
# To get the parameter name for grid search
# m.get_params()

# CV strategy
cv = KFold(5, shuffle=True, random_state=1234)

# Grid search
gs = GridSearchCV(m,
    param_grid={'lasso__alpha': alphas},
    cv=cv,
    scoring="neg_mean_squared_error")
gs.fit(X_train, y_train)
```

[45]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=1234, shuffle=True),
             estimator=Pipeline(steps=[('standardscaler', StandardScaler()),
                                        ('lasso', Lasso())]),
             param_grid={'lasso__alpha': array([0.01, 0.02, 0.03, 0.04, 0.05,
0.06, 0.07, 0.08, 0.09, 0.1 , 0.11,
       0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2 , 0.21, 0.22,
       0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.3 , 0.31, 0.32, 0.33,
       0.34, 0…6, 0.37, 0.38, 0.39, 0.4 , 0.41, 0.42, 0.43, 0.44,
       0.45, 0.46, 0.47, 0.48, 0.49, 0.5 , 0.51, 0.52, 0.53, 0.54, 0.55,
       0.56, 0.57, 0.58, 0.59, 0.6 , 0.61, 0.62, 0.63, 0.64, 0.65, 0.66,
       0.67, 0.68, 0.69, 0.7 , 0.71, 0.72, 0.73, 0.74, 0.75, 0.76, 0.77,
       0.78, 0.79, 0.8 , 0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87, 0.88,
       0.89, 0.9 , 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99,
       1.  ])},
             scoring='neg_mean_squared_error')
```

[46]:
```
print(gs.best_params_)
print(-gs.best_score_)
```

```
{'lasso__alpha': 0.01}
0.7200836927253956
```

[47]:
```
model_fit(gs.best_estimator_, X_test, y_test, plot=True)
```



```
[47]: (0.49871501409473484, 0.7061975744044544, 0.5248719054252093)
```

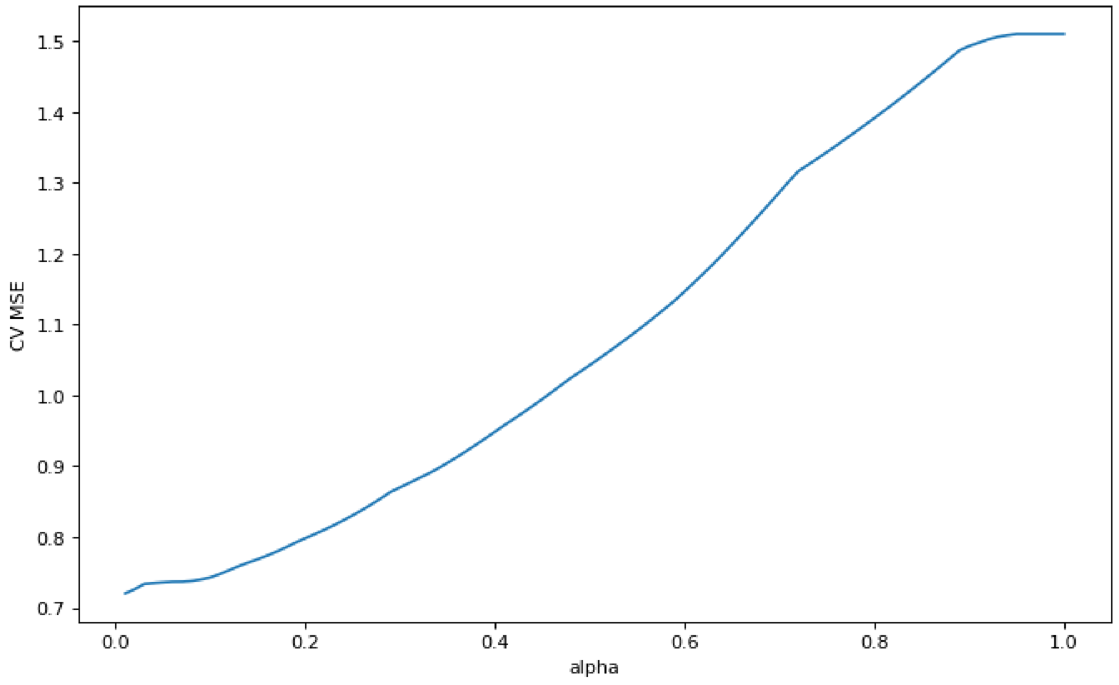
```
[48]: cv_results = pd.DataFrame(gs.cv_results_)
      cv_results.head()
```

```
[48]:    mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
      0       0.003688      0.000692         0.001650        0.000227
      1       0.003089      0.000174         0.001371        0.000048
      2       0.003064      0.000224         0.001415        0.000026
      3       0.003031      0.000119         0.001381        0.000042
      4       0.002938      0.000081         0.001400        0.000069

         param_lasso__alpha                  params  split0_test_score  \
      0               0.01  {'lasso__alpha': 0.01}          -0.940161
      1               0.02  {'lasso__alpha': 0.02}          -0.953103
      2               0.03  {'lasso__alpha': 0.03}          -0.958974
      3               0.04  {'lasso__alpha': 0.04}          -0.937995
      4               0.05  {'lasso__alpha': 0.05}          -0.917877

         split1_test_score  split2_test_score  split3_test_score  split4_test_score  \
      0          -0.740926          -0.749231          -0.415599          -0.754501
      1          -0.770968          -0.772592          -0.413556          -0.720210
      2          -0.793434          -0.808020          -0.419270          -0.687614
      3          -0.810282          -0.847646          -0.420746          -0.656808
      4          -0.820705          -0.891464          -0.420835          -0.627757

         mean_test_score  std_test_score  rank_test_score
      0        -0.720084        0.169478                1
```

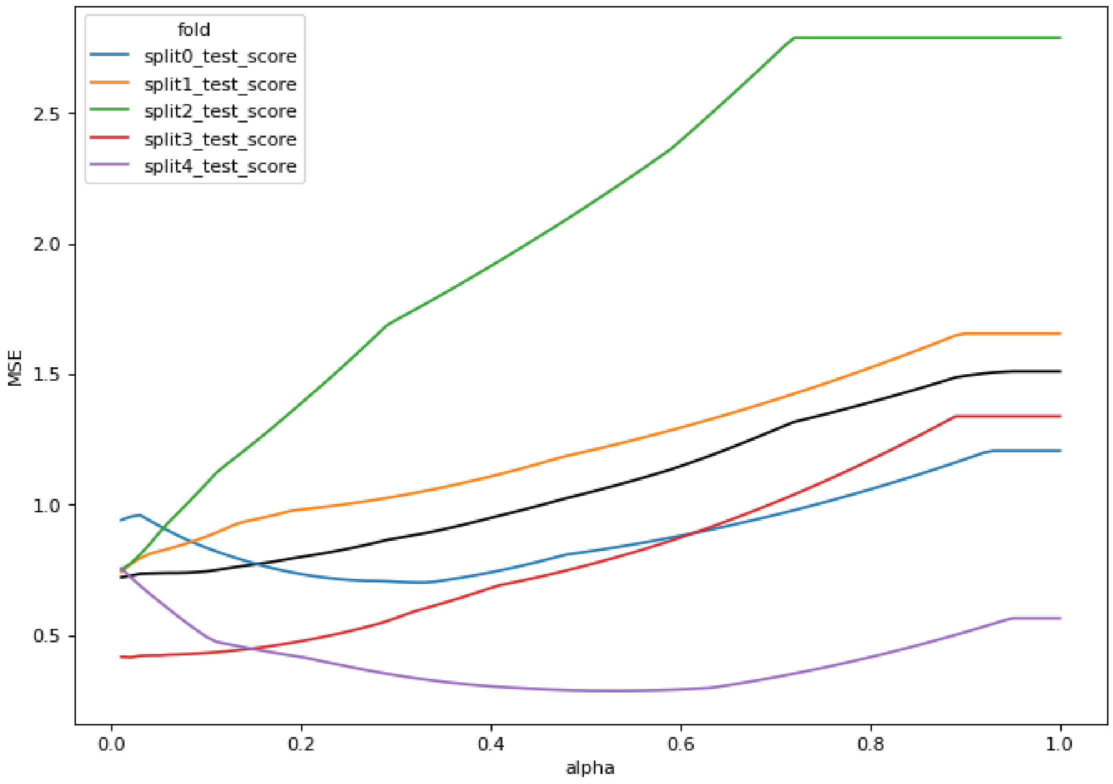
```
1        -0.726086        0.175171              2
2        -0.733463        0.179349              3
3        -0.734695        0.181346              4
4        -0.735728        0.187333              5
```

[49]:
```
# Extract only mean and split scores
cv_mse = pd.DataFrame(
    data = gs.cv_results_
).filter(
    # Extract the split#_test_score and mean_test_score columns
    regex = '(split[0-9]+|mean)_test_score'
).assign(
    # Add the alphas as a column
    alpha = alphas
)

cv_mse.update(
    # Convert negative mses to positive
    -1 * cv_mse.filter(regex = '_test_score')
)
```

[51]:
```
# Part b: plot the CV MSE and MSE for each fold as a function of alpha
# Plot CV MSE
plt.figure(figsize=(10,6))
ax = sns.lineplot(x='alpha', y='mean_test_score', data=cv_mse)
ax.set_ylabel('CV MSE')
plt.show()
```



[52]:
```
# Reshape the data frame for plotting
d = cv_mse.melt(
    id_vars=('alpha','mean_test_score'),
    var_name='fold',
    value_name='MSE'
)

# Plot the validation scores across folds
plt.figure(figsize=(10,7))
sns.lineplot(x='alpha', y='MSE', color='black', errorbar=None, data = d)  #␣
  ↪Plot the mean MSE in black.
sns.lineplot(x='alpha', y='MSE', hue='fold', data = d) # Plot the curves for␣
  ↪each fold in different colors
plt.show()
```



Note that CV MSE is increasing with alpha, while the MSEs for each fold are again unstable for the reasons previously mentioned. Additionally, MSE for each CV is in general increasing for increasing alpha.

[54]:
```
# Part c: extract the coefficients
lasso_coefs = pd.DataFrame(
    np.copy(get_coefs(gs.best_estimator_)),
    columns=["Coefficients"],
    index=fe_names,
)

lasso_coefs
```

[54]:
```
           Coefficients
intercept      2.452345
lcavol         0.680148
lweight        0.284639
age           -0.120078
lbph           0.199373
svi            0.286585
lcp           -0.222663
gleason       -0.000000
pgg45          0.226124
```

Included variables: lcavol, lweight, age, lbph, svi, lcp, pgg45

6.1.2 Exercise 12 (CORE)

Run the following code to compute the CV MSE for the linear model and compare with the CV MSE of the lasso model to suggest an optimal value of α.

[55]:
```
# Lasso doesn't allow for alpha=0, so compute CV MSE for linear regression␣
↪model to compare with Lasso
gs_l = GridSearchCV(
    make_pipeline(
        StandardScaler(),
        LinearRegression()
    ),
    param_grid = {},
    cv=KFold(5, shuffle=True, random_state=1234),
    scoring="neg_mean_squared_error"
).fit(X_train, y_train)
```

[56]:
```
print('CV MSE for baseline linear model', round(gs_l.best_score_ * -1,4))
```

```
CV MSE for baseline linear model 0.7129
```

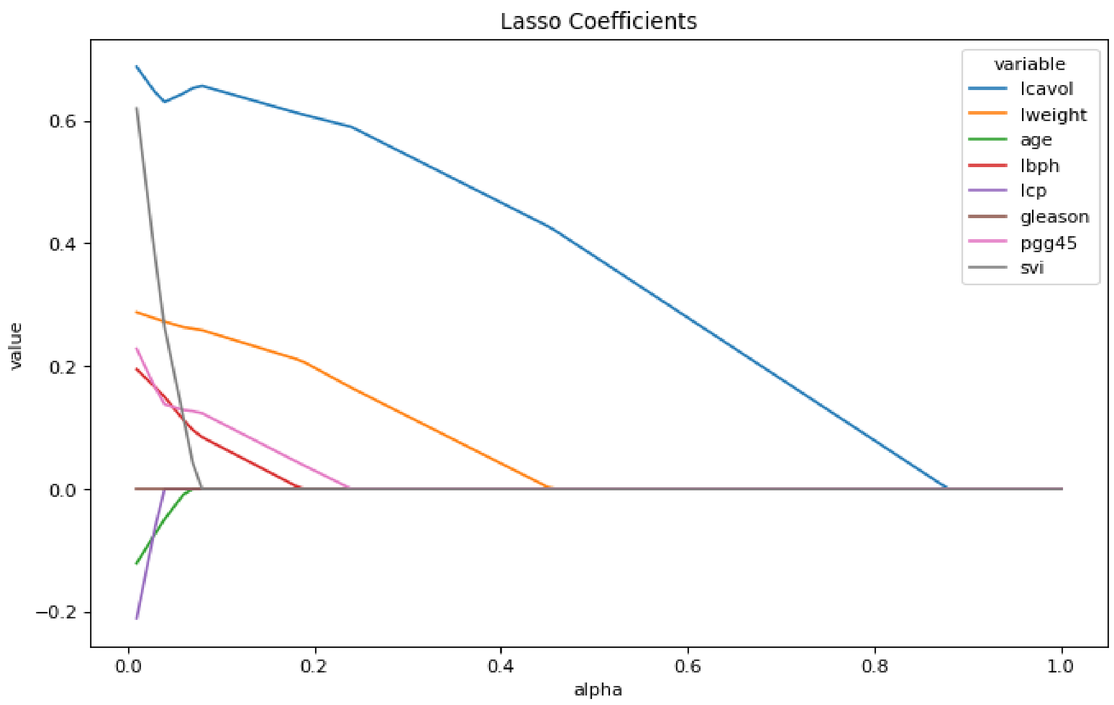
The CV MSE of the lasso model was 0.72 for a value of a lasso alpha = 0.01 (versus the CV MSE for the baseline linear model of 0.7129). This indicates that unlike Ridge regression, Lasso regression may be unable to provide us with a better model compared to the baseline.

6.1.3 Exercise 13 (EXTRA)

In the following code, use `ColumnTransfomer` to apply standarization to all variables except the binary variable `svi`. How does the affect the lasso solution path and the importance of `svi` relative to the other variables?

[57]:
```python
from sklearn.compose import ColumnTransformer
alphas = np.linspace(0.01, 1, num=100)

ws = [] # Store coefficients
mses_train = [] # Store training mses
mses_test = [] # Store test mses

for a in alphas:
    m = make_pipeline(
        ColumnTransformer([
            ('num',StandardScaler(),[0,1,2,3,5,6,7]), # all variables except svi
            ('binary','passthrough',[4])]), # binary variable
        Lasso(alpha=a)
    ).fit(X_train, y_train)

    ws.append(m[1].coef_)
    mses_train.append(mean_squared_error(y_train, m.predict(X_train)))
    mses_test.append(mean_squared_error(y_test, m.predict(X_test)))
```

[58]:
```python
sol_path = pd.DataFrame(
    data = ws,
    columns = X_train.columns[np.array([0,1,2,3,5,6,7,4])] # Label columns w/␣
 ↪feature names
).assign(
    alpha = alphas,
).melt(
    id_vars = ('alpha')
)

# Plot the solution path of the weights
plt.figure(figsize=[10,6])
ax = sns.lineplot(x='alpha', y='value', hue='variable', data=sol_path)
ax.set_title("Lasso Coefficients")
plt.show()
```



7 ElasticNet Regression

Lastly, we can use elastic net regression, which is hybrid between lasso and ridge, including both an ℓ_1 and ℓ_2 penalty. The `ElasticNet` model is again provided by the `linear_model` submodule and minimizes the objective:

$$\frac{1}{2N}||\mathbf{y}-\mathbf{X}\mathbf{w}||_2^2 + \alpha\rho||\mathbf{w}||_1 + 0.5\alpha(1-\rho)||\mathbf{w}||_2^2.$$

In this parameterization, ρ determines relative strength of the ℓ_1 penalty compared to the ℓ_2 and is referred to as `l1_ratio` in `ElasticNet`. Thus, we can also fit ridge and lasso regression models with `ElasticNet` through appropriate choice of `l1_ratio`: - ridge corresponds to `l1_ratio=0` - lasso corresponds to `l1_ratio=1`

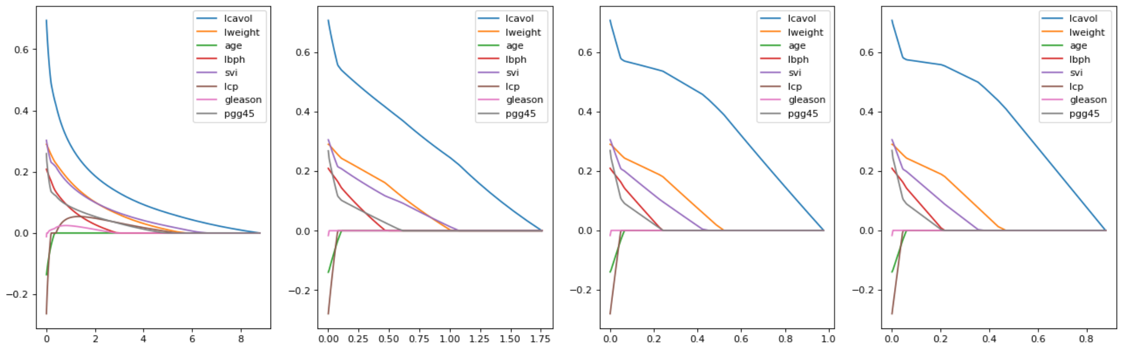
The parameter α is referred to as `alpha` in `ElasticNet` and controls the overall penalty relative the residual sum of squares.

The general `ElasticNet` requires tuning of both `alpha` and `l1_ratio`.

The following code plots the solution path for different choices of `l1_ratio` using the `.path()` method of `ElasticNet`. Notice how the solution paths resemble ridge and lasso for small and large values of `l1_ratio` respectively.

In this case, `.path()` by default automatically selects a range of `alpha` values, except for the case when `l1_ratio = 0`, i.e. ridge regression. For ridge, you need to supply your own grid of `alpha` values through the option `path(...,alphas=myalphas)`.

[59]:
```
from sklearn.linear_model import ElasticNet

Xs = StandardScaler().fit_transform(X_train)
l1r = [.1, .5, .9, 1]
fig, ax = plt.subplots(1,4,figsize= (20,6))
for i, l in enumerate(l1r):
    sol_path = ElasticNet.path(Xs, y_train, l1_ratio=l)
    d = pd.DataFrame( data = sol_path[1].T, columns = X_train.columns, index =␣
 ↪sol_path[0])
    d.plot(ax=ax[i])
```



Again, we can use GridSearchCV (or ElasticNetCV) to tune the parameters. In the following code, we use GridSearchCV to tune both alpha and l1_ratio.

[60]:
```
from sklearn.linear_model import ElasticNetCV

# Grid of tuning parameters
alphas = np.linspace(0.01, 10, num=50)
l1r = [0.01, .1, .5, .7, .9, .95, 1]

# CV strategy
cv = KFold(5, shuffle=True, random_state=1234)

# Pipeline
m = make_pipeline(
        StandardScaler(),
        ElasticNet())

# Grid search
gs_enet = GridSearchCV(m,
                        param_grid={'elasticnet__alpha': alphas,␣
 ↪'elasticnet__l1_ratio': l1r},
                        cv = cv,
                        scoring="neg_mean_squared_error")
```

```
gs_enet.fit(X_train, y_train)

gs_enet.best_params_
```

```
[60]: {'elasticnet__alpha': 0.01, 'elasticnet__l1_ratio': 0.01}
```

```
[61]: print('CV MSE for elasticnet model', round(-gs_enet.best_score_,4))
      print('CV MSE for ridge model',round(-gs.best_score_,4))
```

```
CV MSE for elasticnet model 0.7113
CV MSE for ridge model 0.7201
```

7.0.1 Exercise 15 (EXTRA)

Comment on the optimal values of ElasticNet compared with our basineline, ridge, and lasso models. How does the performance of the models compare on the test data?

```
[ ]:
```

8 Competing the Worksheet

At this point you have hopefully been able to complete all the CORE exercises and attempted the EXTRA ones. Now is a good time to check the reproducibility of this document by restarting the notebook's kernel and rerunning all cells in order.

Before generating the PDF, please go to Edit -> Edit Notebook Metadata and change 'Student 1' and 'Student 2' in the **name** attribute to include your name. If you are unable to edit the Notebook Metadata, please add a Markdown cell at the top of the notebook with your name(s).

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF. Once generated, please submit this PDF on Learn page by 16:00 PM on the Friday of the week the workshop was given.

```
[ ]: !jupyter nbconvert --to pdf mlp_week05.ipynb
```

```
[ ]:
```