mlp-week05

February 24, 2021

1 Machine Learning in Python - Workshop 5

2 1. Setup

2.1 1.1 Packages

In the cell below we will load the core libraries we will be using for this workshop and setting some sensible defaults for our plot size and resolution.

[1]:
```
# Display plots inline
%matplotlib inline

# 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'] = (8,5)
plt.rcParams['figure.dpi'] = 80

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

2.2 1.2 Helper Functions

Below are two helper functions we will be using in this workshop.

[2]:
```
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_])

def model_fit(m, X, y, plot = False):
    """Returns the root mean squared error 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)
    rmse = mean_squared_error(y, y_hat, squared=False)

    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("Fit plot")

        plt.subplot(122)
        sns.scatterplot(x='y', y='resid', data=res).set_title("Residual plot")

        plt.subplots_adjust(left=0.0)
```

```
        plt.suptitle("Model rmse = " + str(round(rmse, 4)), fontsize=16)
        plt.show()

    return rmse
```

2.3 1.3 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 provided with this worksheet.

[3]:
```
prostate = pd.read_csv('prostate.csv')
prostate
```

[3]:

	lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45	\
0	-0.579818	2.769459	50	-1.386294	0	-1.386294	6	0	
1	-0.994252	3.319626	58	-1.386294	0	-1.386294	6	0	
2	-0.510826	2.691243	74	-1.386294	0	-1.386294	7	20	
3	-1.203973	3.282789	58	-1.386294	0	-1.386294	6	0	
4	0.751416	3.432373	62	-1.386294	0	-1.386294	6	0	
..	…	…	…	…	…	…	…	…	
92	2.830268	3.876396	68	-1.386294	1	1.321756	7	60	
93	3.821004	3.896909	44	-1.386294	1	2.169054	7	40	
94	2.907447	3.396185	52	-1.386294	1	2.463853	7	10	
95	2.882564	3.773910	68	1.558145	1	1.558145	7	80	
96	3.471966	3.974998	68	0.438255	1	2.904165	7	20	

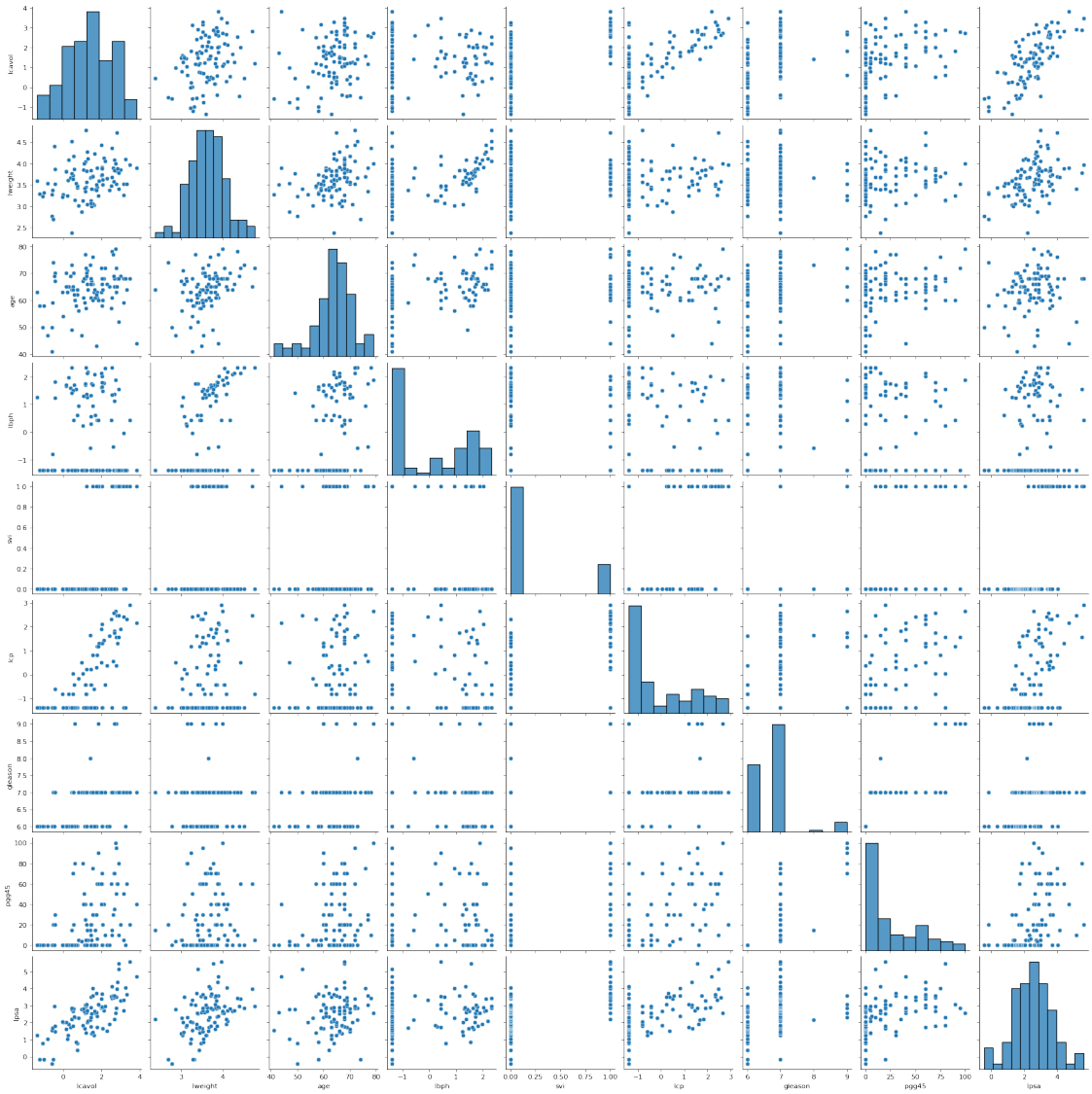
	lpsa	train
0	-0.430783	T
1	-0.162519	T
2	-0.162519	T
3	-0.162519	T

```
4   0.371564     T
..       …   …
92  4.385147     T
93  4.684443     T
94  5.143124     F
95  5.477509     T
96  5.582932     F

[97 rows x 10 columns]
```

As before we will begin by constructing a pairs plot of our data and examining the relationships between our variables.

[4]:
```
sns.pairplot(data=prostate)
```

[4]:
```
<seaborn.axisgrid.PairGrid at 0x7fc77ee20f50>
```



2.3.1 Exercise 1

Are there any interesting patterns in these data? Specifically, your answer should address, * Do any of our variables appear to be categorical / ordinal rather than numeric? * Which variable appears likely to have the strongest relationship with `lpsa`?

The variables svi, pgg45 and gleason appear to be categorical. pgg45 and gleason are ordinal, while svi is nominal.

It appears that lcavol has the strongest relationship with lpsa. Also likely to have a relationship are lweight, svi, lcp, pgg45.

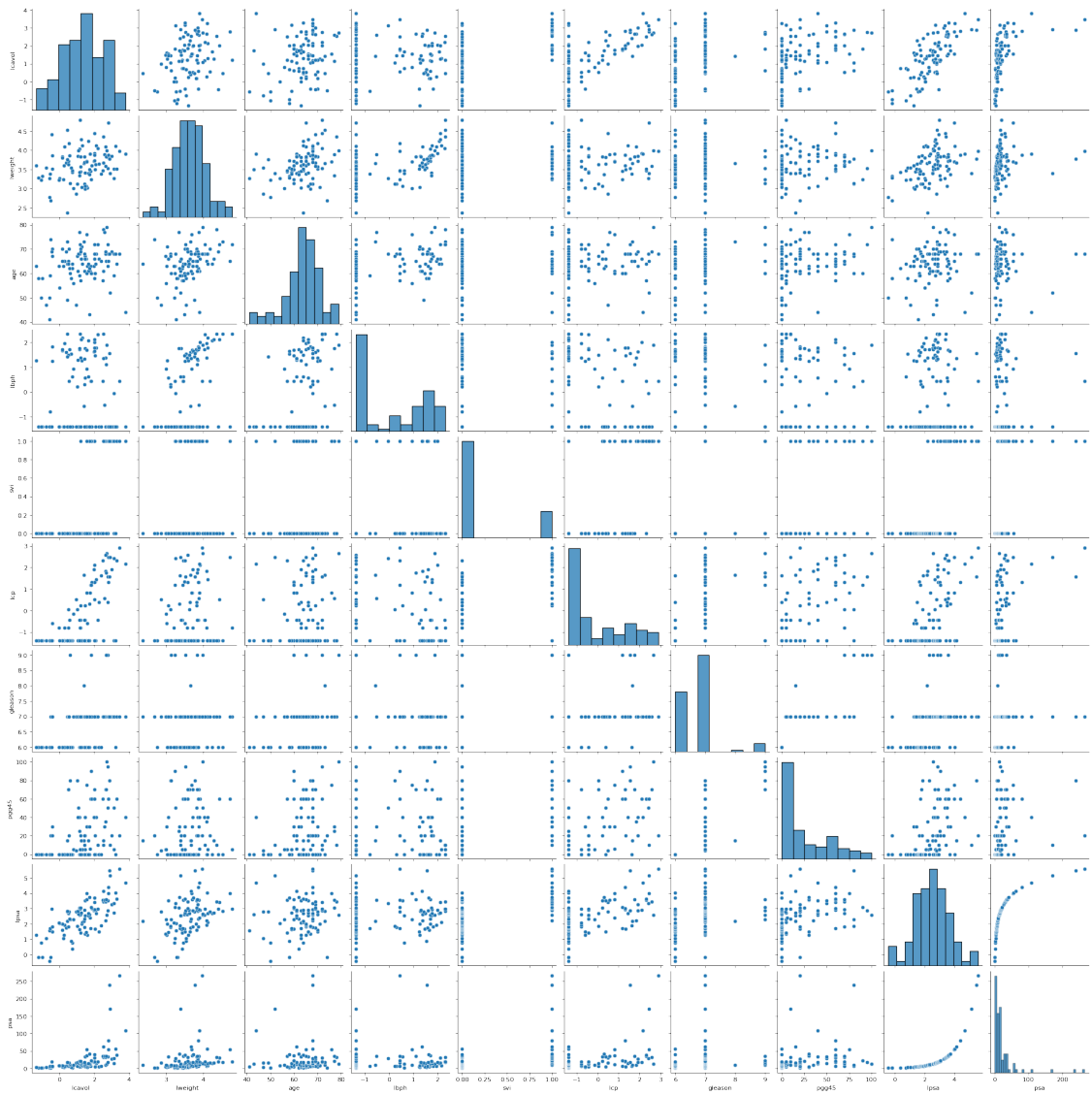
2.3.2 Exercise 2

Why do you think we are exploring the relationship between these variables and `lpsa` (log of psa) rather than just psa?

Let's try to plot using psa and identify potential issues,

[5]:
```
prostate2 = pd.read_csv('prostate.csv')
prostate2["psa"]=np.exp(prostate2["lpsa"])
sns.pairplot(data=prostate2)
```

[5]:
```
<seaborn.axisgrid.PairGrid at 0x7fc772234e10>
```



We can see that, without the log, the data points in the psa vs x (where x is another variable) are concentrated close to the x axis and it is hard to determine if any relationships exist. When we take the log in this case, we can more easily identify the relationships.

2.4 1.4 Validation 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 validation 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`. As we will also need the complete data set we will also construct `X` and `y`, which contain all 97 observations but without the `train` column.

[6]:
```
# Create train and validate data frames
train = prostate.query("train == 'T'").drop('train', axis=1)    # Takes all␣
 ↪rows with T in the train column
validate = prostate.query("train == 'F'").drop('train', axis=1) # Takes all␣
 ↪rows with F in the train column

#train
print(validate.shape)
```

```
(30, 9)
```

[7]:
```
# Training data
X_train = train.drop(['lpsa'], axis=1) # Creates the input matrix X by dropping␣
 ↪the y column
y_train = train.lpsa                   # Creates the y vector of outputs by␣
 ↪taking the y column

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

X_train
#y_train
```

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

[7]:

	lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45
0	-0.579818	2.769459	50	-1.386294	0	-1.386294	6	0
1	-0.994252	3.319626	58	-1.386294	0	-1.386294	6	0
2	-0.510826	2.691243	74	-1.386294	0	-1.386294	7	20
3	-1.203973	3.282789	58	-1.386294	0	-1.386294	6	0
4	0.751416	3.432373	62	-1.386294	0	-1.386294	6	0
..	…	…	…	…	…	…	…	…
90	3.246491	4.101817	68	-1.386294	0	-1.386294	6	0
91	2.532903	3.677566	61	1.348073	1	-1.386294	7	15
92	2.830268	3.876396	68	-1.386294	1	1.321756	7	60
93	3.821004	3.896909	44	-1.386294	1	2.169054	7	40
95	2.882564	3.773910	68	1.558145	1	1.558145	7	80

[67 rows x 8 columns]

[8]:
```
# Validation data
X_test = validate.drop('lpsa', axis=1)  # Drop the y column from the validation␣
 ↪data - gives the validation input matrix
y_test = validate.lpsa                  # Take the y column from the validation␣
 ↪data
```

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

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

[9]:
```
# Complete data
X = prostate.drop(['lpsa','train'], axis=1) # Drop the y and train columns to␣
 ↪obtain the complete input matrix X
y = prostate.lpsa                           # Take out just the lpsa column as␣
 ↪our complete output

print("X:", X.shape)
print("y:", y.shape)
```

```
X: (97, 8)
y: (97,)
```

2.5 1.5 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]:
```
from sklearn.linear_model import LinearRegression
lm = LinearRegression().fit(X_train, y_train)
```

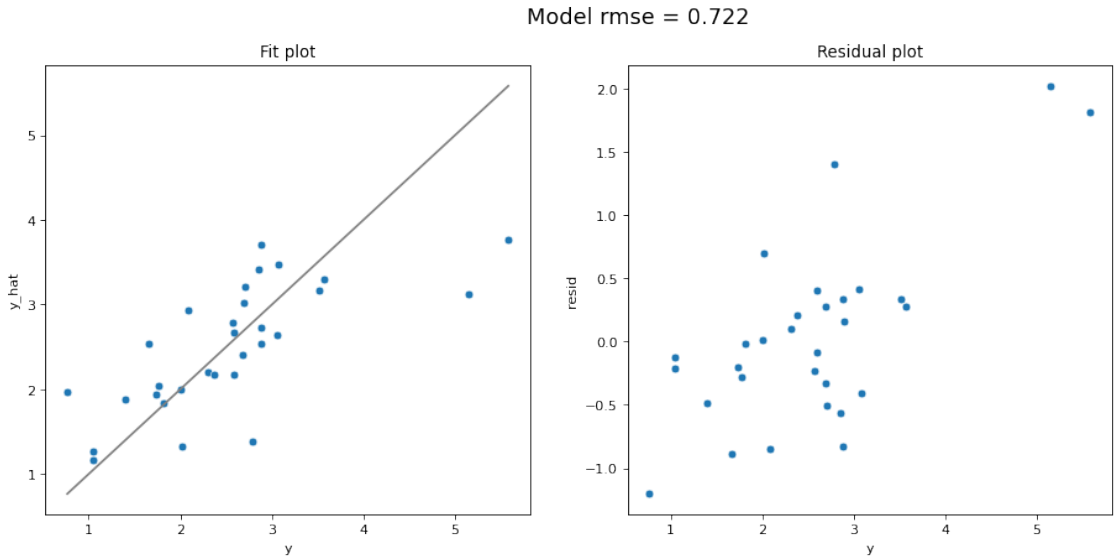
We can extract the coefficients for the model, which correspond to the variables: `intercept`, `lcavol`, `lweight`, `age`, `lbph`, `svi`, `lcp`, `gleason`, and `pgg45` respectively.

[11]:
```
get_coefs(lm)
```

[11]:
```
array([ 0.42917013,  0.57654319,  0.61402   , -0.01900102,  0.14484808,
        0.73720864, -0.20632423, -0.02950288,  0.00946516])
```

These coefficients have the typical regression interpretation, e.g. for each unit increase in `lcavol` we expect `lpsa` to increase by 0.577 on average. These values are not of particular interest for us for this particular problem as we are more interested in the predictive properties of our model(s). To evaluate this we will use the `model_fit` helper function defined above.

[12]:
```
model_fit(lm, X_test, y_test, plot=True)
```



[12]: 0.7219930785731963

Primarily we will use this function to obtain the rmse of our model using the validation data (`X_test` and `y_test`). Note that we fit the model using the training data (`X_train` and `y_train`). We have also included a fit (y vs $\hat{y}$) and resid (y vs $y - \hat{y}$) plot of these results.

2.5.1 Exercise 3

Based on these plots do you see anything in the fit or residual plot that is potentially concerning?

There are two outliers on the Fit plot on the far right.

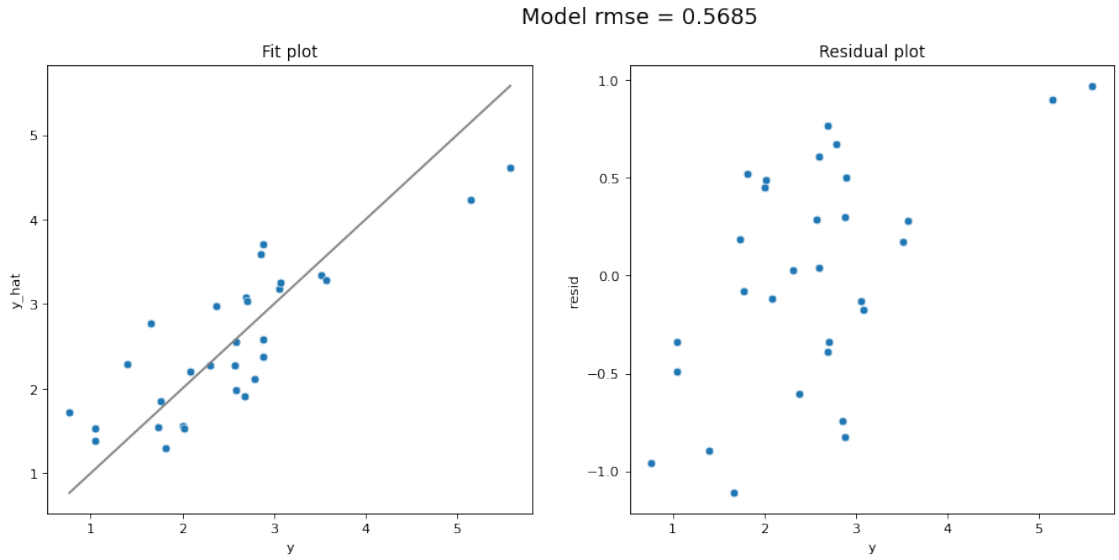
For larger values of y, the model has large residuals.

2.5.2 Exercise 4

Would you expect the rmse of the model to be better or worse when using the training data (compared to the validation data)? Check your answer using the `model_fit` function.

[13]:
```
# We would expect a lower rmse when using the validation data, as the model␣
 ↪will closer fit the data. However this is bad
# practise and should be avoided.

lm2 = LinearRegression().fit(X_test, y_test)
model_fit(lm2, X_test, y_test, plot=True)
```

```
get_coefs(lm2)
```



[13]:
```
array([ 0.42841554,  0.45570429,  0.5553612 , -0.00894243, -0.0810177 ,
        0.65973603,  0.16970839, -0.01843756,  0.0007157 ])
```

2.6 1.6 Standardization

In subsequent sections we will be exploring the use of the Ridge and Lasso regression models which both penalize larger values of β. While not particularly bad, our baseline model had βs 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 center and scale *all* features to a common scale before fitting one of these models. The typical scaling approach is to subtract the mean of each feature and then divide by its standard deviation - this results in all feature columns having a mean of 0 and a variance of 1. 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.

[14]:
```
from sklearn.preprocessing import StandardScaler
```

[15]:
```
S = StandardScaler().fit(X)

X
```

[15]:

	lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45
0	-0.579818	2.769459	50	-1.386294	0	-1.386294	6	0
1	-0.994252	3.319626	58	-1.386294	0	-1.386294	6	0
2	-0.510826	2.691243	74	-1.386294	0	-1.386294	7	20
3	-1.203973	3.282789	58	-1.386294	0	-1.386294	6	0
4	0.751416	3.432373	62	-1.386294	0	-1.386294	6	0
..	…	…	…	…	…	…	…	…
92	2.830268	3.876396	68	-1.386294	1	1.321756	7	60
93	3.821004	3.896909	44	-1.386294	1	2.169054	7	40
94	2.907447	3.396185	52	-1.386294	1	2.463853	7	10
95	2.882564	3.773910	68	1.558145	1	1.558145	7	80
96	3.471966	3.974998	68	0.438255	1	2.904165	7	20

[97 rows x 8 columns]

Once fit, we can examine the values used for the scaling by checking the `mean_` and `var_` attributes of the transformer.

[16]:
```
S.mean_
```

[16]:
```
array([ 1.35000958,  3.62894266, 63.86597938,  0.10035561,  0.21649485,
       -0.17936558,  6.75257732, 24.3814433 ])
```

[17]:
```
S.var_
```

[17]:
```
array([1.37483540e+00, 1.81644057e-01, 5.48583271e+01, 2.08314048e+00,
       1.69624827e-01, 1.93494631e+00, 5.16101605e-01, 7.87266872e+02])
```

Keep in mind, that the training, testing, and validation sets will not necessarily have the same feature column means and standard deviations - as such it is important that we choose a consistent set of values that are used for all of the data. In other words, be careful to not expect that `StandardScaler().fit_transform(X_train)` and `StandardScaler().fit(X).transform(X_train)` will give the same answer. The best way to avoid this issue is to include the `StandardScaler` in a modeling pipeline for your data.

2.6.1 Exercise 5

Explain why scaling `y`, `y_train`, or `y_test` is not necessary.

We perform standardisation so that none of our coefficients dominate the penalty function in ridge and lasso regression. The values `y`, `y_train` and `y_test` are the objective of each of our models

and so do not have coefficients and we do not need to worry about them dominating the penalty term.

2.6.2 Exercise 6

What are the units of the transformed features in `StandardScaler().fit_transform(X_train)`?

The transformed features have no units

[18]:
```
S_transform = S.transform(X_train)

S2 = StandardScaler().fit_transform(X_train)

#print(S2 - S_transform)
```

2.7 1.7 Scaled Linear Regression

Now that we have scaled the data, let us fit another simple linear regression model using these scaled features.

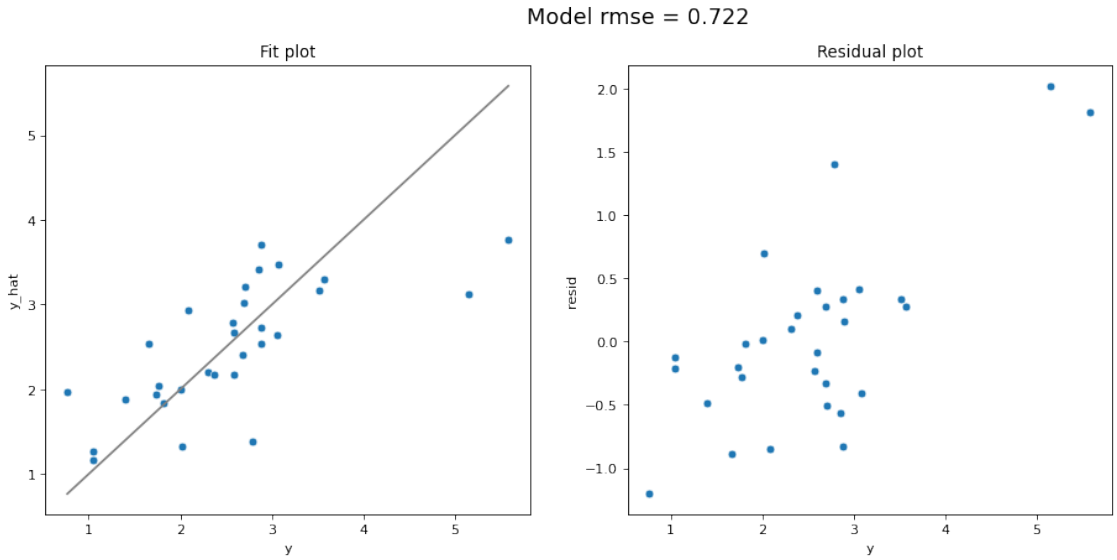
[19]:
```
lm_scaled = make_pipeline(
    StandardScaler(),
    LinearRegression()
).fit(X_train, y_train)
```

Once fit we can extract the model coefficients and calculate the validation rmse,

[20]:
```
get_coefs(lm_scaled)
```

[20]:
```
array([ 2.45234509,  0.71104059,  0.29045029, -0.14148182,  0.21041951,
        0.30730025, -0.28684075, -0.02075686,  0.27526843])
```

[21]:
```
model_fit(lm_scaled, X_test, y_test, plot=True)
```



[21]: 0.7219930785731952

2.7.1 Exercise 7

Using this new model what has changed about our model results? Comment on both the model's coefficients as well as its predictive performance.

We can see that the coefficients are substantially different for most of the terms

[22]:
```
print(get_coefs(lm)-get_coefs(lm_scaled))
```

```
[-2.02317495 -0.13449741  0.32356971  0.1224808  -0.06557143  0.42990839
  0.08051652 -0.00874602 -0.26580326]
```

We note that the model rmse is the same, we would expect this as scaling the data should have no impact on the model's fit - consequentially the fit and residual plots are identical.

---

3 2. Ridge Regression

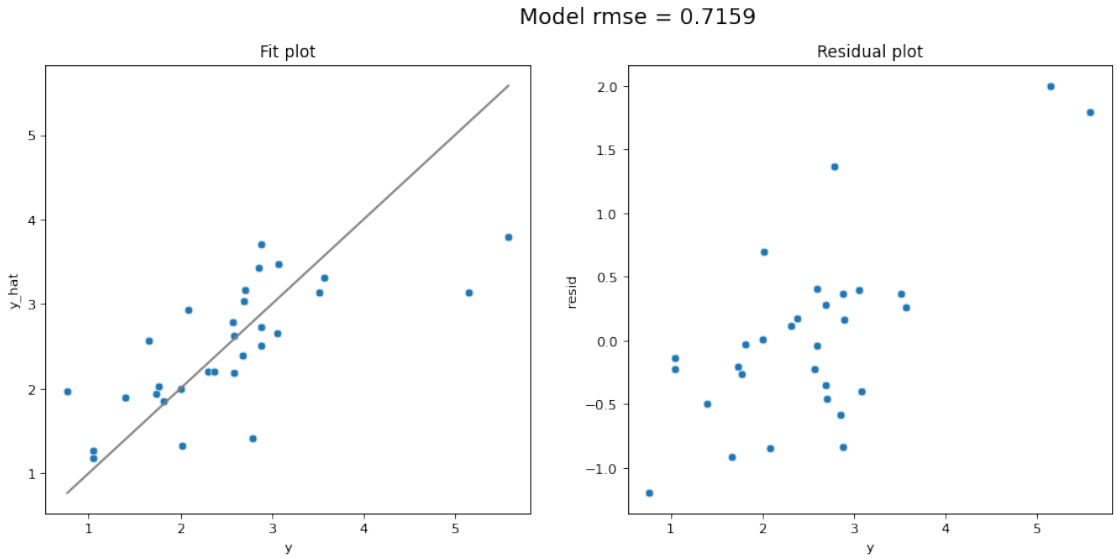
Ridge regression is a natural extension to linear regression which introduces an ℓ_2 penalty on the coefficients to a standard least squares problem. Mathematically, we can express this as the following optimization problem,

$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \alpha(\boldsymbol{\beta}^T\boldsymbol{\beta})$$

The `Ridge` model is provided by the `linear_model` submodule and requires a single parameter `alpha` which determines the tuning parameter that adjusts the weight of the ℓ_2 penalty.

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

[24]:
```
r = make_pipeline(
    StandardScaler(),
    Ridge(alpha=1)
).fit(X_train, y_train)

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



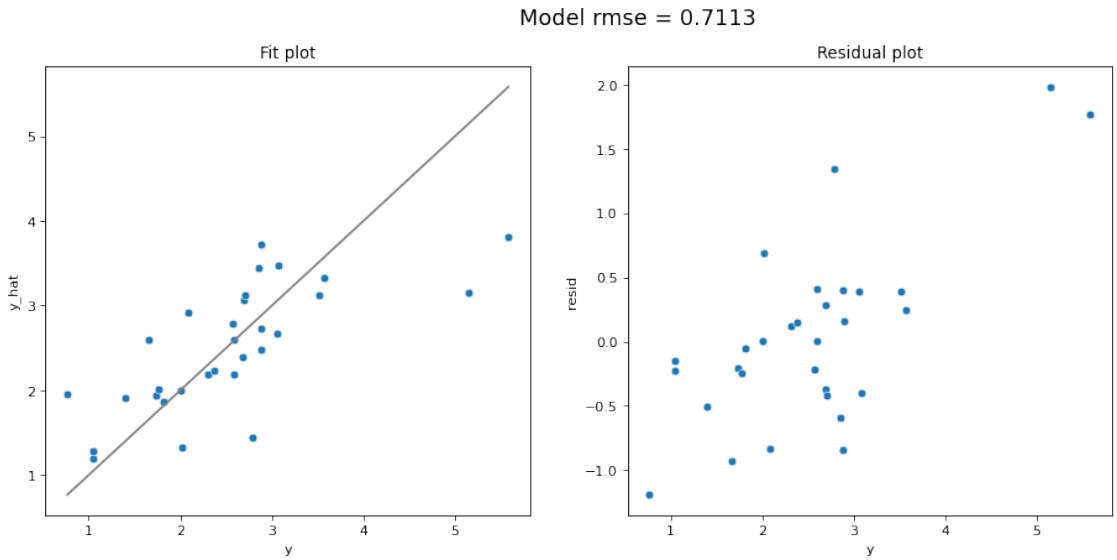
[24]:
```
0.715903222061737
```

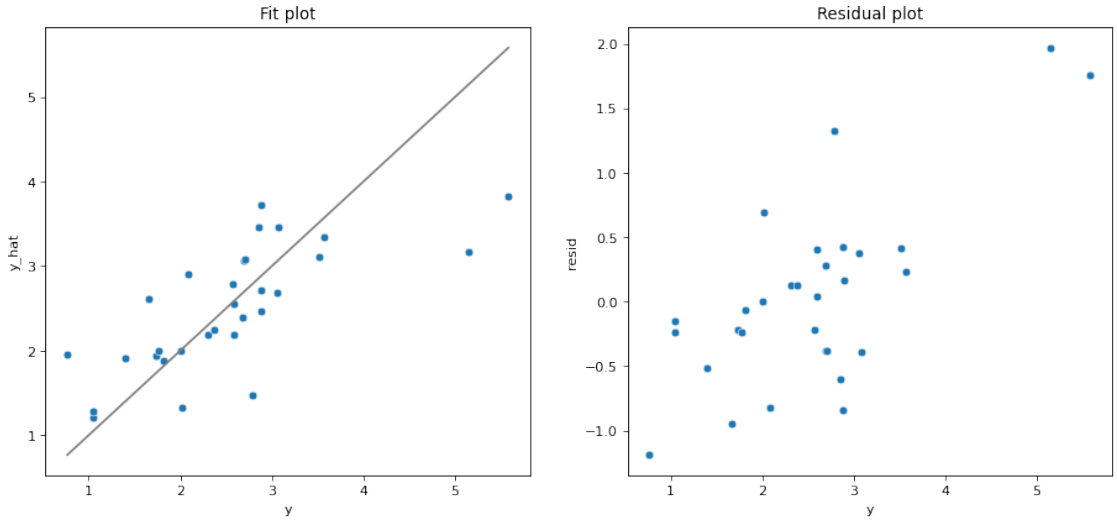
3.0.1 Exercise 8

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 rmse change?

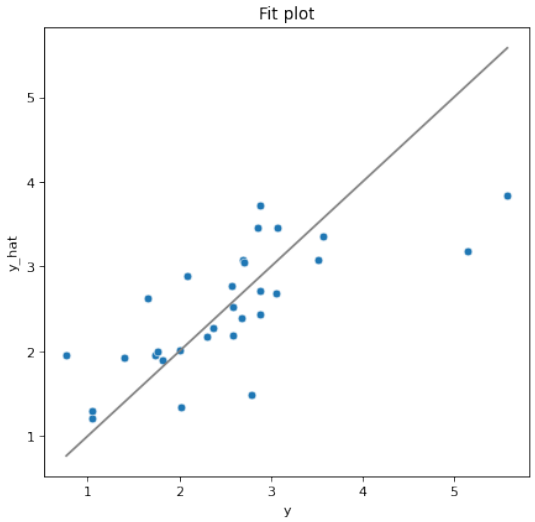
[25]:
```
for i in range(2,7):
    r = make_pipeline(
        StandardScaler(),
        Ridge(alpha=i)
    ).fit(X_train, y_train)

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

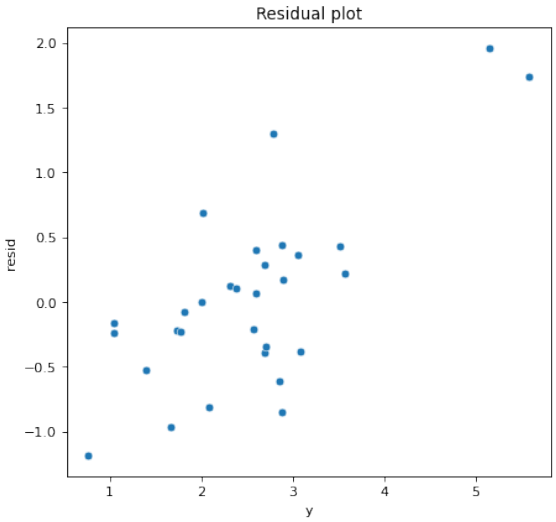
```
r = make_pipeline(
    StandardScaler(),
    Ridge(alpha=50)
).fit(X_train, y_train)

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



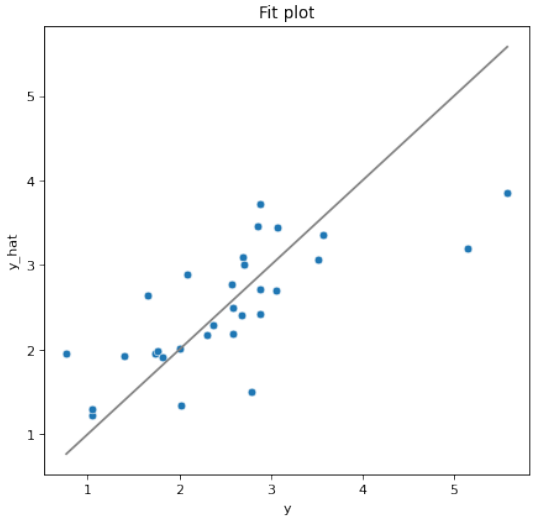


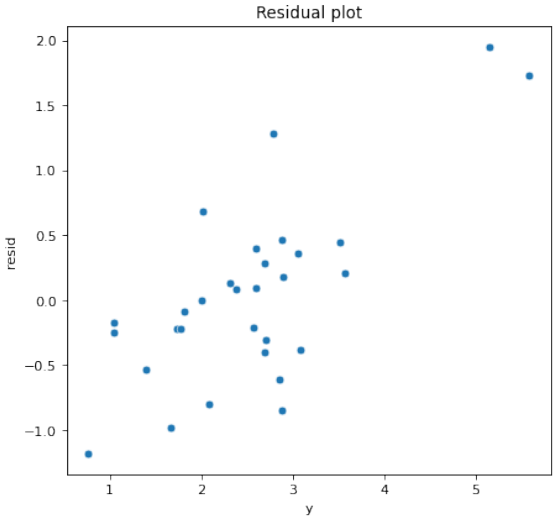
Model rmse = 0.705

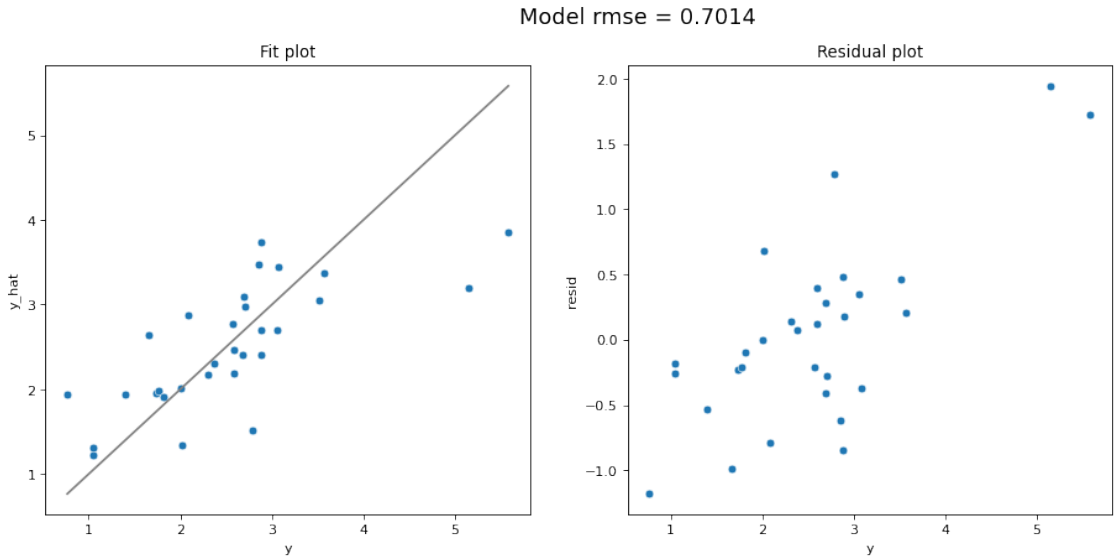


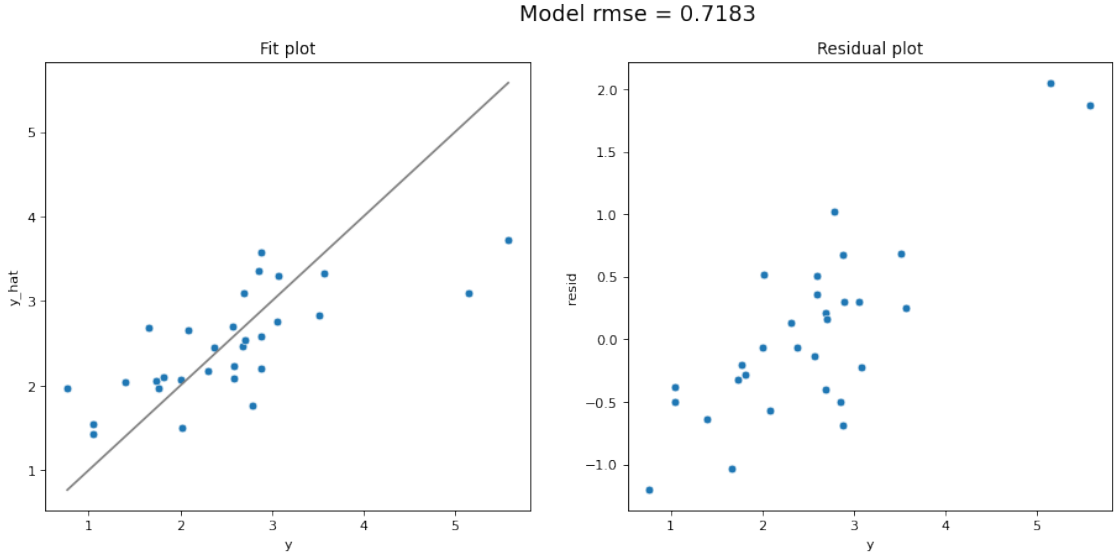


Model rmse = 0.703









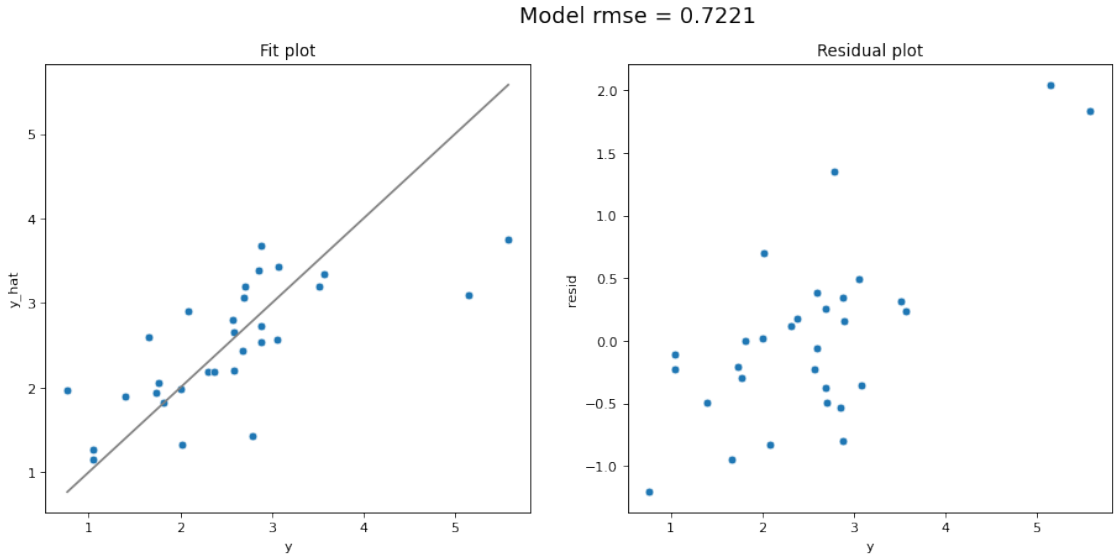
[25]: 0.718268379341787

As we increase alpha, the model rmse decreases. (Note that if we use alpha = 50, then the rmse is worse). Qualitatively, as we increase alpha, the fit and residual plots don't appear to change much.

3.0.2 Exercise 9

In Section 1.4 we mentioned the importance of scaling features before fitting a Ridge regression model. The code below fits the Ridge model to the untransformed training data - repeat Exercise 8 using these data. How does the model fit change as alpha changes? How does the rmse change? How does the models performance compare to the previous model?

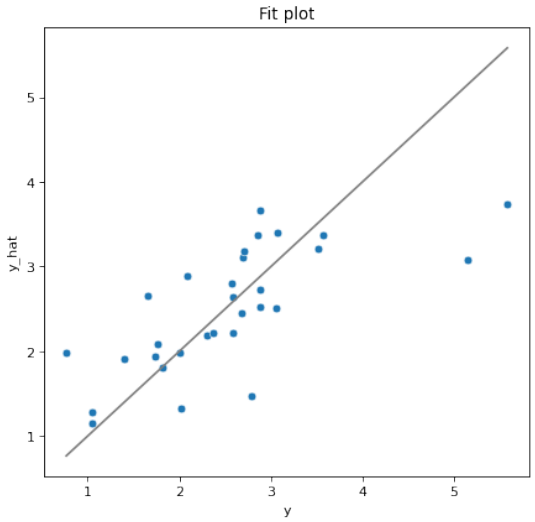
[26]:
```
r_wo_scale = make_pipeline(
    Ridge(alpha=1)
).fit(X_train, y_train)

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

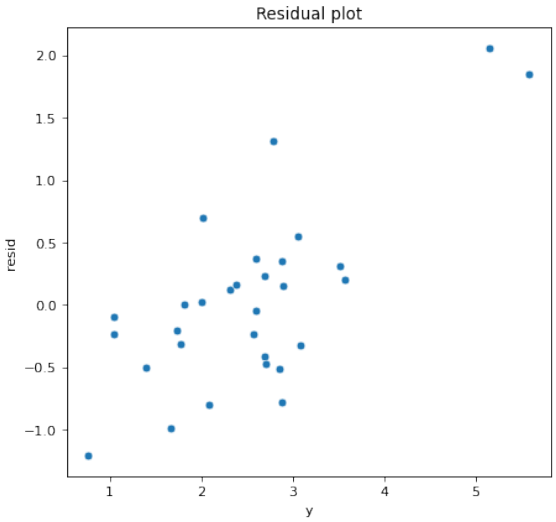


[26]:
```
0.7220807306906396
```

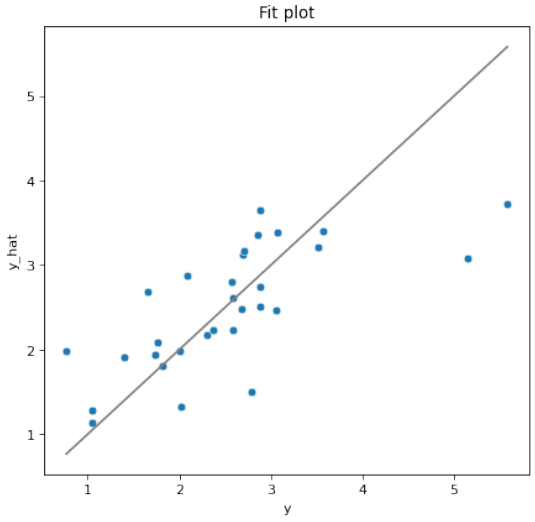
[27]:
```
for i in range(2,7):
    r_wo_scale = make_pipeline(
        Ridge(alpha=i)
    ).fit(X_train, y_train)

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

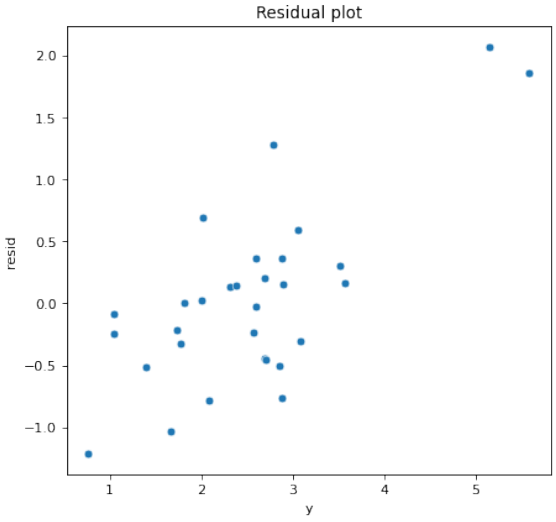
Model rmse = 0.7227



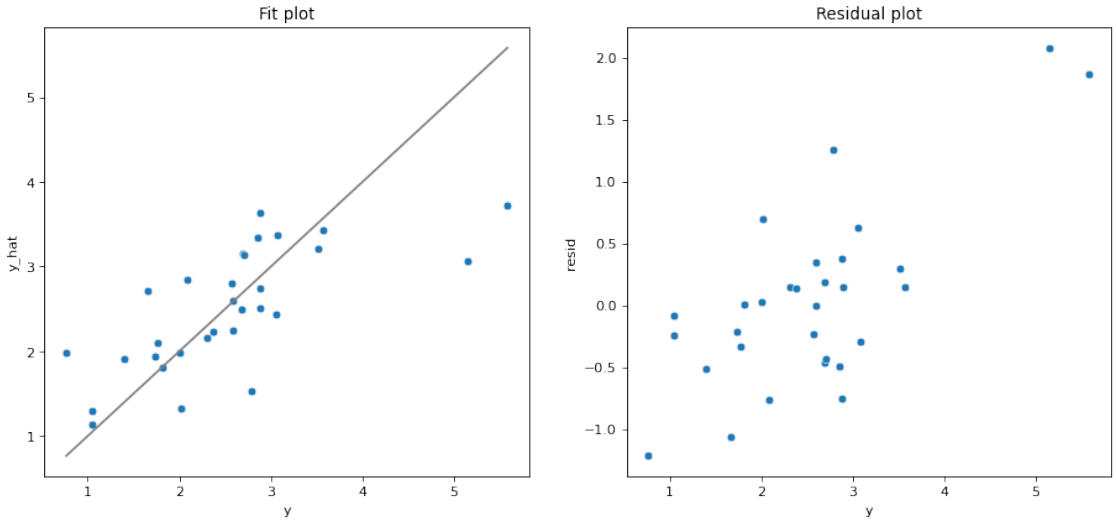


Model rmse = 0.7234

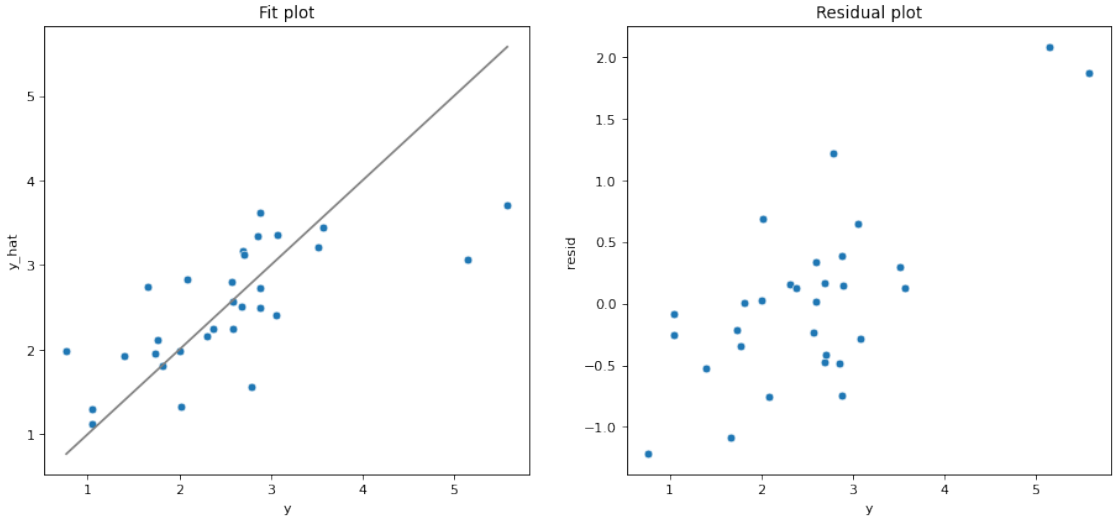


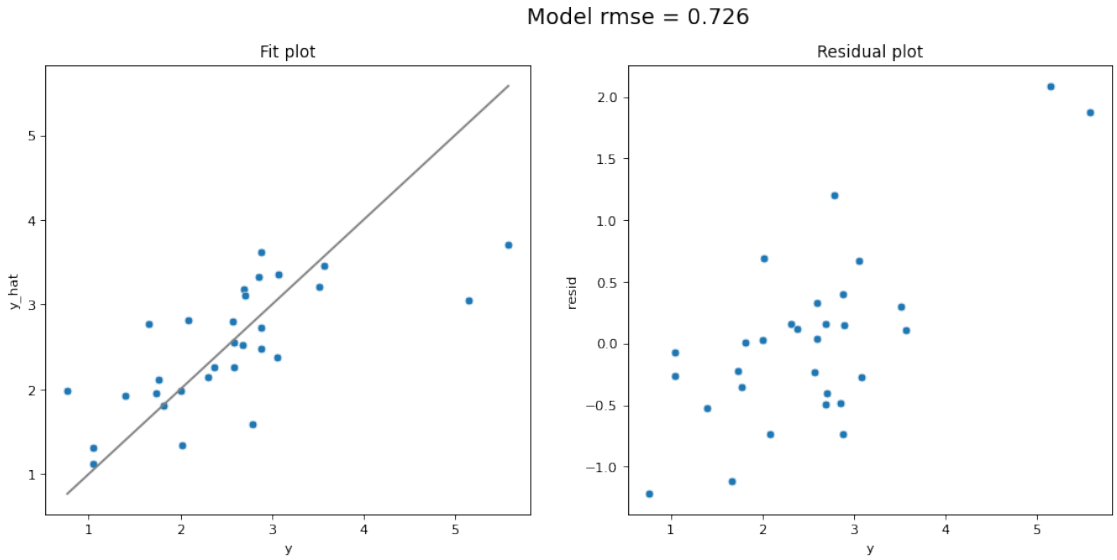


Model rmse = 0.7243



Model rmse = 0.7251





When using the scaled data, the rmse is worse than the previous model. However the fit and residual plots appear to be very similar.

3.1 2.1 Ridge βs as a function of α

Finally, one of the useful ways of thinking about the behavior of Ridge regression models is to examine the relationship between our choice of α and the resulting βs relative to the results we would have obtained from the linear regression model. 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 βs towards 0 asymptotically.

[28]:
```
alphas = np.logspace(-2, 3, num=200) # from 10^-2 to 10^3

betas = [] # Store coefficients
rmses = [] # Store validation rmses

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

    # We drop the intercept as it is not included in Ridge's l2 penalty and␣
↪hence not shrunk
    betas.append(get_coefs(m)[1:])
    rmses.append(model_fit(m, X_test, y_test))
```

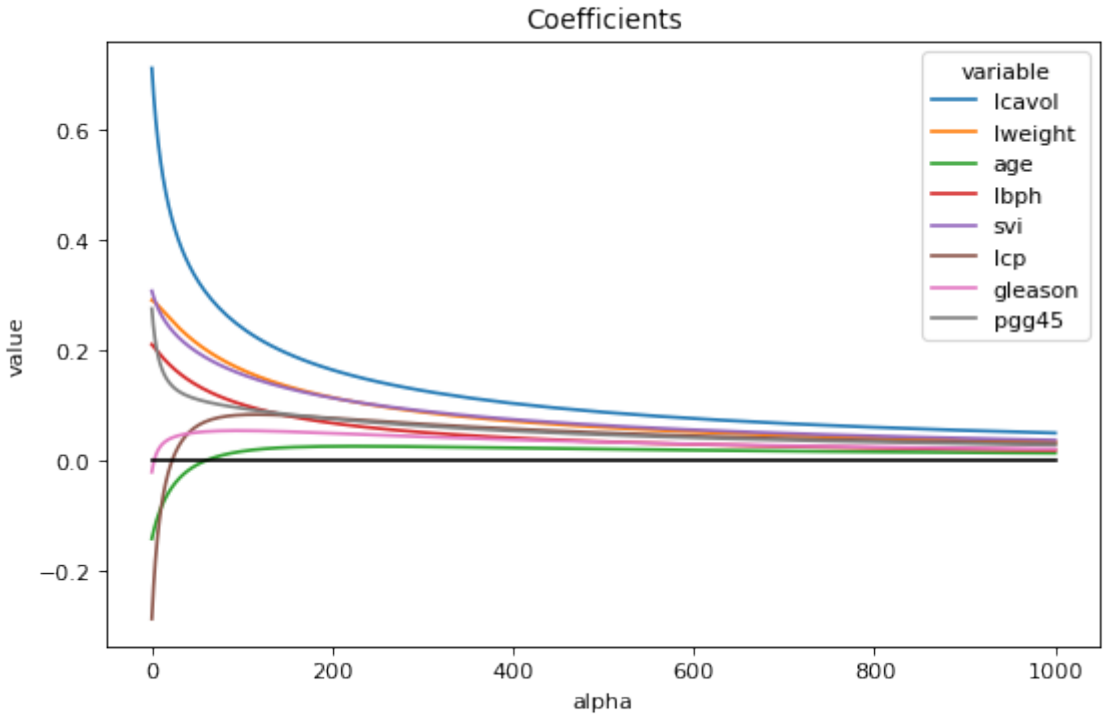
[29]:
```
res = pd.DataFrame(
    data = betas,
    columns = X.columns # Label columns w/ feature names
).assign(
    alpha = alphas,
    rmse = rmses
).melt(
    id_vars = ('alpha', 'rmse')
)

res
```

[29]:

	alpha	rmse	variable	value
0	0.010000	0.721923	lcavol	0.710769
1	0.010596	0.721919	lcavol	0.710753
2	0.011227	0.721915	lcavol	0.710735
3	0.011895	0.721910	lcavol	0.710717
4	0.012604	0.721905	lcavol	0.710698
…	…	…	…	…
1595	793.409667	0.918990	pgg45	0.033958
1596	840.665289	0.923488	pgg45	0.032517
1597	890.735464	0.927870	pgg45	0.031116
1598	943.787828	0.932133	pgg45	0.029757
1599	1000.000000	0.936274	pgg45	0.028441

[1600 rows x 4 columns]

[30]:
```
sns.lineplot(x='alpha', y='value', hue='variable', data=res).
 ↪set_title("Coefficients")
plt.plot(np.linspace(0,1000,200),np.zeros(200),'k')
plt.show()
```



3.1.1 Exercise 10

Based on this plot, which variable(s) seem to be the most important for predicting `lpsa`. *Hint* - think about what the degree of shrinkage towards 0 means in this context.

lcavol appears to be the most influencial variable for predisicting `lpsa` as it remains the largest for all chosen values of alpha.

We could also say that lweight shrinks to zero the slowest and has influence over all values of alpha.

3.2 2.2 Tuning with GridSearchCV

The α in the Ridge regression model is another example of a hyperparameter, and just like the degree in a polynomial model we can use cross validation to attempt to identify the optimal value for our data. As with the polynomial models from last week we will start by using `GridSearchCV` to employ 5-fold cross validation to determine an optimal α.

[31]:
```
alphas = np.linspace(0, 15, num=151)
```

[32]:
```
gs = GridSearchCV(
    make_pipeline(
        StandardScaler(),
        Ridge()
    ),
    param_grid={'ridge__alpha': alphas},
    cv=KFold(5, shuffle=True, random_state=1234),
    scoring="neg_root_mean_squared_error"
).fit(X_train, y_train)
```

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.

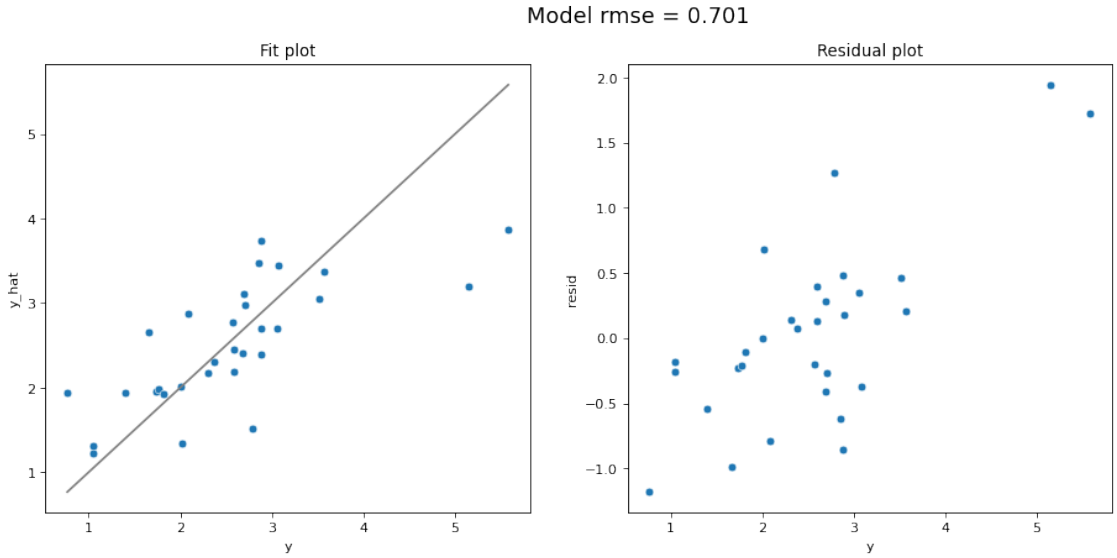
Once fit, we can examine the results to determine what value of α was chosen as well as examine the calculated mean of the rmses.

[33]:
```
print(gs.best_params_)
```

```
{'ridge__alpha': 6.300000000000001}
```

To evaluate this model we can access the `best_estimator_` model object and use it to obtain an rmse for our validation data.

[34]:
```
model_fit(gs.best_estimator_, X_test, y_test, plot=True)
```



[34]:
```
0.7010036592591791
```

3.2.1 Exercise 11

How does this model compare to the performance of our baseline model? Is it better or worse?

This model performs much better than the baseline model. The rmse in this case is 0.701 compared to 0.722 for the baseline model. We can see the fit plot here has the data more closely centred around y=x and the residuals are all more centred around zero.

3.2.2 Exercise 12

How do the model coefficient for this model compare to the base line model? *Hint* - be careful about which baseline model you compare with.

[35]:
```
baseline_coefficients = get_coefs(lm)
#new_coefficients2 = gs.best_estimator_.named_steps['ridge'].coef_
new_coefficients = get_coefs(gs.best_estimator_)

print('difference is ',new_coefficients - baseline_coefficients)
#print(new_coefficients)
#print(new_coefficients2)
```

```
difference is  [ 2.02317495  0.00939586 -0.33198247 -0.08406354  0.05285496
-0.45937008
  0.06807395  0.04674712  0.18460702]
```

The coefficients are somewhat similar, however we notice a large difference in the first coefficient

---

To further explore this choice of α, we can collect relevant data about the folds and their performance from the `cv_results_` attribute. In this case we are particularly interested in 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 rmse values into positive values).

[36]:
```
cv_res = 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_res.update(
```

```
    # Convert negative rmses to positive
    -1 * cv_res.filter(regex = '_test_score')
)

#print(gs.cv_results_)
cv_res
```

[36]:

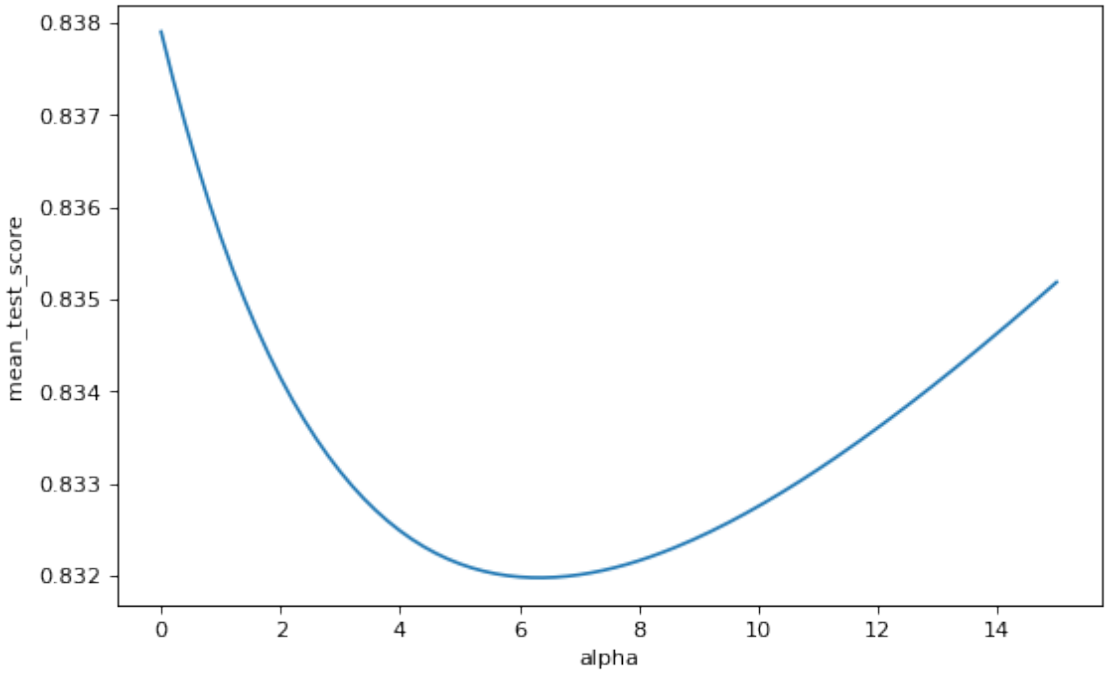
	split0_test_score	split1_test_score	split2_test_score	\
0	0.966190	0.827171	0.855973	
1	0.966449	0.827878	0.856687	
2	0.966707	0.828562	0.857410	
3	0.966965	0.829222	0.858141	
4	0.967222	0.829860	0.858880	
..	…	…	…	
146	0.976708	0.855542	0.964078	
147	0.976656	0.855616	0.964700	
148	0.976604	0.855690	0.965321	
149	0.976550	0.855764	0.965939	
150	0.976496	0.855838	0.966556	

	split3_test_score	split4_test_score	mean_test_score	alpha
0	0.651764	0.888423	0.837904	0.0
1	0.650275	0.886923	0.837642	0.1
2	0.648836	0.885434	0.837390	0.2
3	0.647447	0.883957	0.837147	0.3
4	0.646106	0.882492	0.836912	0.4
..	…	…	… …	
146	0.632215	0.746262	0.834961	14.6
147	0.632499	0.745614	0.835017	14.7
148	0.632785	0.744970	0.835074	14.8
149	0.633071	0.744328	0.835131	14.9
150	0.633358	0.743689	0.835187	15.0

[151 rows x 7 columns]

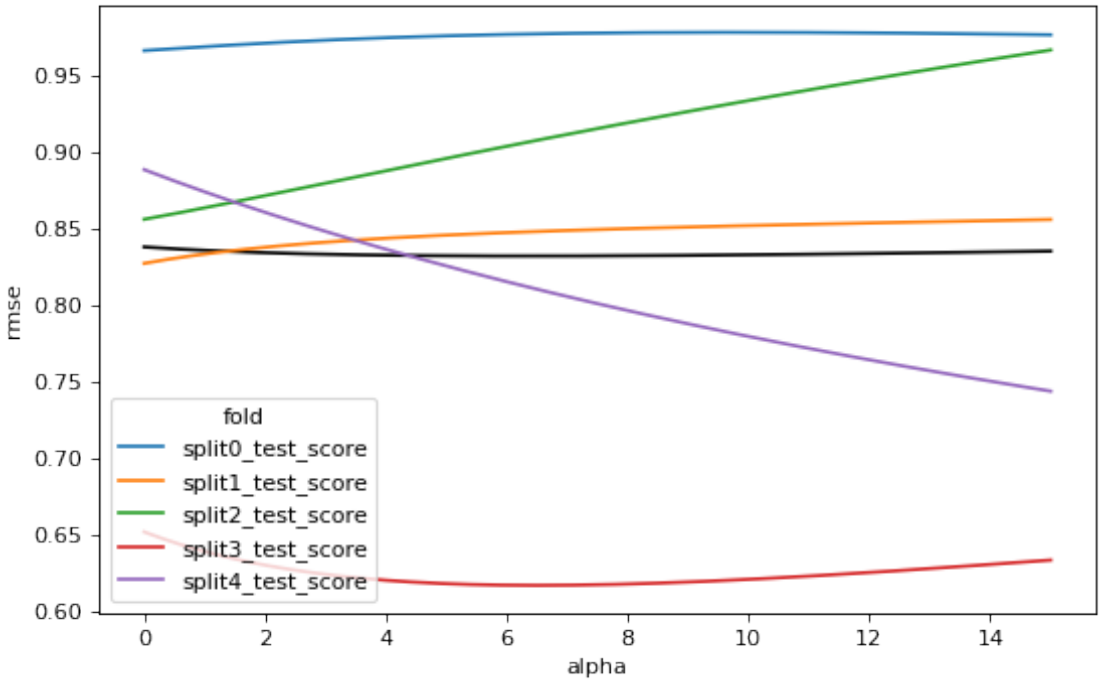
This data frame can then be used to plot α against the mean root mean squared value over the 5 folds, to produce the following plot.

[37]:
```
sns.lineplot(x='alpha', y='mean_test_score', data=cv_res)
plt.show()
```



This plot shows that the value 6.4 is obtained as the minimum of this curve. However, this plot gives us an overly confident view of this choice of this particular value of α. Specifically, if instead of just plotting the mean rmse, we also examine the variablility of that estimate as well as examine the α vs rmse curve of each fold we see that these estimates are far noisier than they first appeared and we should take the value $\alpha = 6.4$ with a grain of salt.

[38]:
```
d = cv_res.melt(
    id_vars=('alpha','mean_test_score'),
    var_name='fold',
    value_name='rmse'
)

sns.lineplot(x='alpha', y='rmse', color='black', ci = None, data = d)  # Plot␣
 ↪the mean rmse +/- the std dev of the rmse.
sns.lineplot(x='alpha', y='rmse', hue='fold', data = d) # Plot the curves for␣
 ↪each fold
plt.show()

d
```



[38]:

	alpha	mean_test_score	fold	rmse
0	0.0	0.837904	split0_test_score	0.966190
1	0.1	0.837642	split0_test_score	0.966449
2	0.2	0.837390	split0_test_score	0.966707
3	0.3	0.837147	split0_test_score	0.966965
4	0.4	0.836912	split0_test_score	0.967222
..	…	…	…	…
750	14.6	0.834961	split4_test_score	0.746262
751	14.7	0.835017	split4_test_score	0.745614
752	14.8	0.835074	split4_test_score	0.744970
753	14.9	0.835131	split4_test_score	0.744328
754	15.0	0.835187	split4_test_score	0.743689

[755 rows x 4 columns]

In the plot above the black line shows the mean rmse across the folds (this is the same curve as shown in the previous plot) and the gray interval indicates + and - 1 standard deviation of the rmses. The other colored cuves shows the rmse curve for each of the different folds.

3.2.3 Exercise 13

Why do you think that our cross validation results are unstable?

The cross validation results are different for each fold, this is to be expected as each split is of a different section of the data and we expect variability across the data.

3.3 2.3 Tuning with RidgeCV

Because the process of identifying the value of α is critical to most uses of Ridge regression, sklearn provides a helpful function called `RidgeCV` which combines `Ridge` with `GridSearchCV`. We import this function from `linear_model` and fit it in the same way.

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

[40]:
```
r_cv = RidgeCV(
    alphas = np.linspace(0.1, 15, num=150), # RidgeCV does not allow alpha=0␣
 ↪for some reason
    scoring = "neg_root_mean_squared_error"
).fit(X_train, y_train)
```

[41]:
```
r_cv = make_pipeline(
    StandardScaler(),
    RidgeCV(
        alphas = np.linspace(0.1, 15, num=150), # RidgeCV does not allow␣
 ↪alpha=0 for some reason
        scoring = "neg_root_mean_squared_error"
    )
).fit(X_train, y_train)
```

The resulting model object now has the "optimal" value of α stored in the `alpha_` attribute which we can access directly.

[42]:
```
r_cv.named_steps["ridgecv"].alpha_
```

[42]:
```
3.6
```

Additionally, the returned object can be used like any other model object to obtain predictions for the fitted model using this value of α.

[43]:
```
model_fit(r_cv, X_test, y_test)
```

[43]:
```
0.7060258433830947
```

3.3.1 Exercise 14

This model seems to have arrived at a different optimal value for α compared to using `GridSearchCV` and it also has a different (slightly worse) rmse. Review the documentation for `RidgeCV`. Can you

explain this discrepancy?

GirdSearchCV is using 5-fold cross fold validation while RidgeCV is using Leave-One-Out Cross-Validation. Since they are using different cross-validation techniques, we expect different answers.

3.3.2 Exercise 15

Refit the model using the RidgeCV in such a way that you obtain a result similar to what was obtained by `GridSearchCV` (in terms of the optimal α and validation rmse).

[44]:
```
r_cv2 = RidgeCV(
    alphas = np.linspace(0.1, 15, num=150), # RidgeCV does not allow alpha=0␣
 ↪for some reason
    scoring = "neg_root_mean_squared_error"
).fit(X_train, y_train)

r_cv2 = make_pipeline(
    StandardScaler(),
    RidgeCV(
        alphas = np.linspace(0.1, 15, num=150), # RidgeCV does not allow␣
 ↪alpha=0 for some reason
        scoring = "neg_root_mean_squared_error",
        cv=KFold(5, shuffle=True, random_state=1234)
    )
).fit(X_train, y_train)

print(model_fit(r_cv2, X_test, y_test), r_cv2.named_steps["ridgecv"].alpha_)
```

```
0.7016718631100777 5.8
```

---

4 3. The Lasso

The Lasso is a related modeling approach to Ridge regression, but instead uses an ℓ_1 penalty on the coefficients. Mathematically, we can express this model as the solution of the following optimization problem,

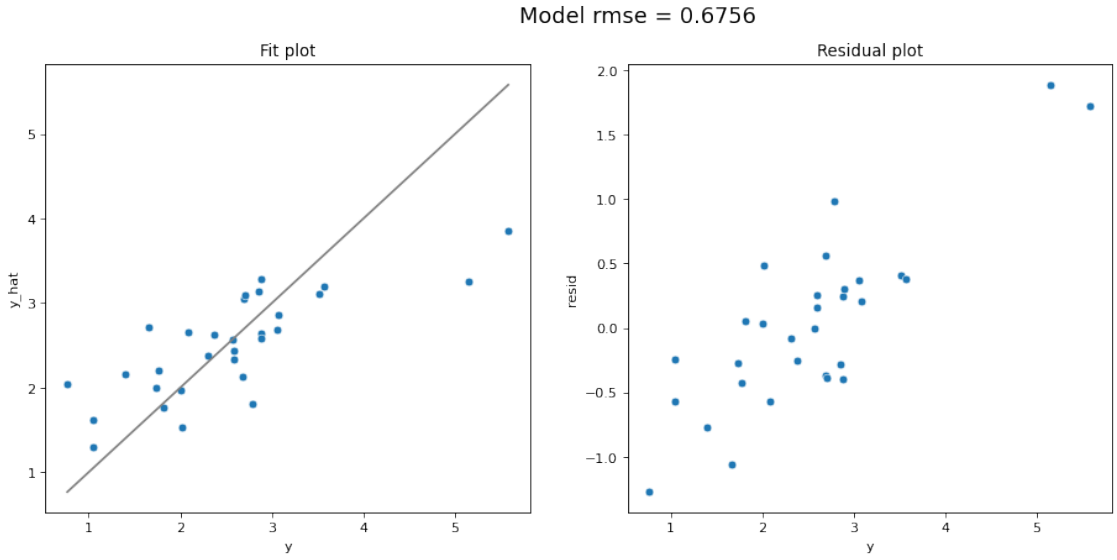
$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|_2^2 + \alpha\|\boldsymbol{\beta}\|_1$$

As with the other models from this worksheet, 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.

[45]:
```
from sklearn.linear_model import Lasso
```

[46]:
```
l = make_pipeline(
    StandardScaler(),
    Lasso(alpha=0.15)
).fit(X_train, y_train)
```

[47]:
```
print("lasso rmse:", model_fit(l, X_test, y_test, plot=True))
print("lasso coefs:", get_coefs(l))
#print(l.get_params)
```




```
lasso rmse: 0.6755984657058377
lasso coefs: [2.45234509 0.56476334 0.20956522 0.         0.05792154 0.13596345
 0.         0.         0.03500349]
```

4.0.1 Exercise 16

Adjust the value of `alpha` in the cell above and rerun it. How does the model fit change as alpha changes? How does the validation rmse change?

[48]:
```
for alpha2 in np.linspace(0.001,0.5,30):
    l = make_pipeline(
        StandardScaler(),
        Lasso(alpha=alpha2)
    ).fit(X_train, y_train)
```

```
    print("alpha: ", round(alpha2,3), "lasso rmse:", model_fit(l, X_test,␣
↪y_test, plot=False))
```

```
alpha:  0.001 lasso rmse: 0.7199741622185774
alpha:  0.018 lasso rmse: 0.6971249207284915
alpha:  0.035 lasso rmse: 0.6817564869585512
alpha:  0.053 lasso rmse: 0.6762309249890449
alpha:  0.07 lasso rmse: 0.6759561315289178
alpha:  0.087 lasso rmse: 0.6736878692257212
alpha:  0.104 lasso rmse: 0.6726146539189751
alpha:  0.121 lasso rmse: 0.6727422049533461
alpha:  0.139 lasso rmse: 0.6740698409451681
alpha:  0.156 lasso rmse: 0.6765904920944367
alpha:  0.173 lasso rmse: 0.6802933102864899
alpha:  0.19 lasso rmse: 0.6851534081831185
alpha:  0.207 lasso rmse: 0.6911473540970788
alpha:  0.225 lasso rmse: 0.698381983413937
alpha:  0.242 lasso rmse: 0.7059893167722748
alpha:  0.259 lasso rmse: 0.7139147425639472
alpha:  0.276 lasso rmse: 0.7221477878646968
alpha:  0.294 lasso rmse: 0.7306780542754572
alpha:  0.311 lasso rmse: 0.7394952563038019
alpha:  0.328 lasso rmse: 0.7485887440313159
alpha:  0.345 lasso rmse: 0.7579498695001146
alpha:  0.362 lasso rmse: 0.7669039838489673
alpha:  0.38 lasso rmse: 0.7731091736264653
alpha:  0.397 lasso rmse: 0.7795565596520232
alpha:  0.414 lasso rmse: 0.7862401837204228
alpha:  0.431 lasso rmse: 0.7931527584586315
alpha:  0.448 lasso rmse: 0.800291925527267
alpha:  0.466 lasso rmse: 0.8075985481121265
alpha:  0.483 lasso rmse: 0.8150703280004103
alpha:  0.5 lasso rmse: 0.8227233569617336
```

As the value of alpha increases from (almost) zero, the rmse decreases until alpha = 0.104 after which it increases as the value of alpha increases. This implies that the best model fit is around alpha = 0.104

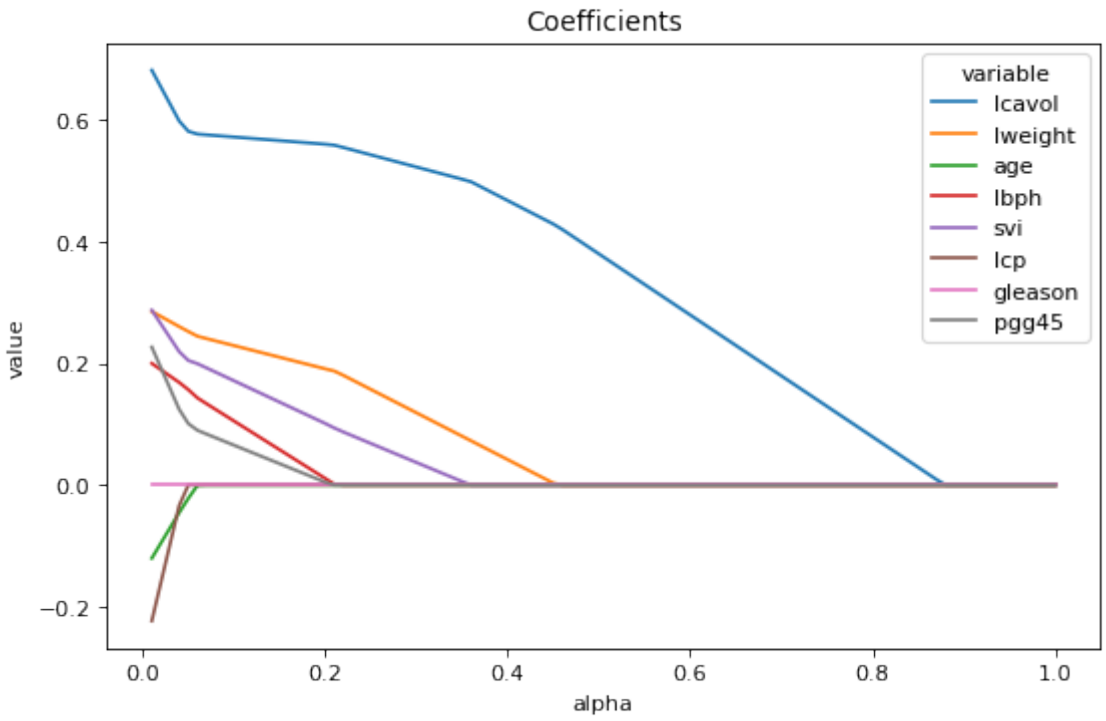
4.1 3.1 Lasso βs as a function of α

As with Ridge regression we can examine the values of β we obtain as tuning parameter α is adjusted.

[49]:
```
alphas = np.linspace(0.01, 1, num=100)
betas = [] # Store coefficients
rmses = [] # Store validation rmses
```

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

    # Again ignore the intercept since it isn't included in the penalty
    betas.append(get_coefs(m)[1:])
    rmses.append(model_fit(m, X_test, y_test))

res = pd.DataFrame(
    data = betas,        # Coefficients
    columns = X.columns # Coefficient names
).assign(
    alpha = alphas,      # Add alpahs
    rmse = rmses         # Add validation rmses
).melt(
    id_vars = ('alpha', 'rmse') # Move columns into the rows
)
```

[50]:
```
sns.lineplot(x='alpha', y='value', hue='variable', data=res).
 ↪set_title("Coefficients")
plt.show()
```



[51]:
```
res
```

[51]:

	alpha	rmse	variable	value
0	0.01	0.706198	lcavol	0.680148
1	0.02	0.695289	lcavol	0.652391
2	0.03	0.686044	lcavol	0.624639
3	0.04	0.678529	lcavol	0.596890
4	0.05	0.675825	lcavol	0.579971
..	…	…	…	…
795	0.96	1.027975	pgg45	0.000000
796	0.97	1.027975	pgg45	0.000000
797	0.98	1.027975	pgg45	0.000000
798	0.99	1.027975	pgg45	0.000000
799	1.00	1.027975	pgg45	0.000000

[800 rows x 4 columns]

4.1.1 Exercise 17

How does the relationship between the βs and α differ from what we saw with the Ridge regression results.

In the Ridge regression model, we require much larger values of alpha to see each coefficient tend to zero (in the range 0-1000). We also note that the coefficients in Ridge regression never equal zero, they just tend to zero as we increase alpha to infinity. In Lasso, above alpha = 0.9 all coefficients are zero.

4.1.2 Exercise 18

Based on this plot, which variable(s) seem to be the most important for predicting `lpsa`. *Hint* - think about what the degree of shrinkage towards 0 means in this context. How does this compare to your answer from the ridge regression model?

It appears again that `lcavol` is the most important variable, it has the largest value for all chosen values of alpha and so the greatest influence on our model. `lweight` has the smallest degree of shrinkage towards zero in this case, indicating that the variable is important in the prediction.

4.2 3.2 Tuning Lasso

We can again use the `GridSearchCV` function to tune our Lasso model and optimize the α hyper-parameter. We avoid using $\alpha = 0$ as this causes a warning due to the fitting method (coordinate descent) not converging well without regularization (the ℓ_1 penalty here).

[ ]:
```
alphas = np.linspace(0.01, 1, num=100)

l_gs = GridSearchCV(
    make_pipeline(
        StandardScaler(),
        Lasso()
    ),
    param_grid={'lasso__alpha': alphas},
    cv=KFold(5, shuffle=True, random_state=1234),
    scoring="neg_root_mean_squared_error"
).fit(X_train, y_train)
```

[ ]:
```
print( "best alpha:", l_gs.best_params_['lasso__alpha'])
print( "best rmse :", l_gs.best_score_ * -1)
print( "validation rmse:", model_fit(l_gs.best_estimator_, X_test, y_test) )
```

Worryingly, the chosen alpha is the smallest value provided to our grid search, and hence it has selected the model closest to a linear regression model. We can investigate this further by plotting α versus the `mean_test_score` values from the `cv_results_` attribute.

[ ]:
```
l_cv_res = pd.DataFrame().assign(
    alpha = alphas,
    rmse = -1 * l_gs.cv_results_['mean_test_score'],           # mean of the␣
 ↪rmse over the folds
    rmse_se = l_gs.cv_results_['std_test_score'] / np.sqrt(l_gs.n_splits_), #␣
 ↪standard error of the rmse
)
```

[ ]:
```
plt.figure(figsize=(14, 6))

plt.subplot(121)
sns.lineplot(x='alpha', y='rmse', data=l_cv_res)

plt.subplot(122)
sns.lineplot(x='alpha', y='rmse', data=l_cv_res).set_xscale('log')
```

In this case it appears that the model's rmses nearly monotonically increase as α increases. This indicates that the CV proceedure is exhibiting a preference for the linear regression mode, i.e. a lasso model with no shrinkage. We can check this explicitly by fitting the `LinearRegression` with `GridSearchCV` and comparing the cross validation rmse, this is necessary because our previous modeling did not use any CV to calculate the rmse.

[ ]:
```
GridSearchCV(
    make_pipeline(
        StandardScaler(),
        LinearRegression()
    ),
    param_grid = {},
    cv=KFold(5, shuffle=True, random_state=1234),
    scoring="neg_root_mean_squared_error"
).fit(
    X_train, y_train
).best_score_ * -1
```

In this case the linear regression model does produce a smaller mean rmse than any of the Lasso models. This suggests our choice of model should just be the original linear regression model.

Aside - However, if we examine these plots closesly, values of α between 0.01 and 0.1 have very similar mean rmses and there is uncertainty in our estimates of these rmses based on the cross validation folds and the size of our data. One of the suggestions employed by Hastie, et al. in their `glmnet` R package is instead of using the α with the smallest mean rmse to instead use the largest value of α that has an error metric (rmse) that is within 1 standard *error* of the minimum value of the error metric (rmse). We can find this value using the `l_cv_res` data frame we previously constructed.

[ ]:
```
i = l_cv_res.rmse.idxmin()

min_rmse = l_cv_res.rmse[i]       # Smallest rmse
min_rmse_se = l_cv_res.rmse_se[i] # Std error of the smallest rmse

sub = l_cv_res.rmse <= min_rmse + min_rmse_se # Find rmses w/in 1se of the min␣
↪+ se

alpha_1se = l_cv_res.alpha[ sub ].max() # Find the largest alpha

alpha_1se
```

While this approach seems plausible / practical, it should be treated as at best a heuristic as I have not been able to track down any theoretical support for it. Note that we could also employ this strategy even if the minimal α had not been approximately 0 and it is still likely to be helpful as any increase in α is likely to reduce the number of coefficients in the final model.

4.2.1 Exercise 19

If you were to use the $\alpha_{1\text{se}} = 0.26$ which variables would be excluded from the model?

[ ]:
```
l = make_pipeline(
    StandardScaler(),
```

```
    Lasso(alpha=0.26)
).fit(X_train, y_train)

get_coefs(l)
```

We can see that the only coefficients included are those for `lcavol`,`lweight` and `svi`. So all others are excluded

---

4.2.2 Exercise 20

If you were to use the $\alpha_{1se} = 0.26$ what would the validation rmse be for this model? How does this compare to the other models we've examined so far? Why might we still prefer this model over the linear regression or Ridge model?

[ ]:
```
model_fit(l, X_test, y_test, plot=True)
```

The validation rmse is 0.7143 which is higher than our previous models, however it is more simple for this value of alpha as we are only considering three coefficients. So we would prefer this model.

4.3 4 Concluding Remarks

It is important to notice that throughout the previous two sections we have taken great pains to avoid using our test data to in any way inform our choice of the tuning parameter. Instead, we have always used `KFold` with our training data to obtain the necessary metrics for optimizing the α hyperparameter. This would also have been possible using the complete data `X` and would have slightly improved our rmse estimates due to the slightly larger sample sizes in the test train splits but it would then mean were repeated using the validation data in the process of determining α which then puts us at risk for overfitting and therefore having an overly optimistic view of our model's uncertainty.

4.4 5. Competing the worksheet

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

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF and turn it in on gradescope under the `mlp-week05` assignment.

[ ]:
```
!jupyter nbconvert --to pdf mlp-week05.ipynb
```

Created in Deepnote