Lec 15 - scikit-learn
Cross-validation

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

Column Transformers

Are a tool for selectively applying transformer(s) to the columns of an array or DataFrame, they function in a way that is similar to a pipeline and similarly have a helper function `make_column_transformer()`.

```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

ct = make_column_transformer(
  (StandardScaler(), ["volume"]),
  (OneHotEncoder(), ["cover"]),
).fit(
  books
)
```

```
ct.get_feature_names_out()
```

```
## array(['standardscaler__volume', 'onehotencoder__cover_hb', 'onehotencoder__cover_pb'], dtype=object)
```

```
ct.transform(books)
```

```
## array([[ 0.12101,  1.     ,  0.     ],
##        [ 0.51997,  1.     ,  0.     ],
##        [ 0.85192,  1.     ,  0.     ],
##        [-1.84637,  1.     ,  0.     ],
##        [-0.43936,  1.     ,  0.     ],
##        [-0.62209,  1.     ,  0.     ],
##        [ 1.16561,  1.     ,  0.     ],
```

Keeping or dropping other columns

One addition important argument is `remainder` which determines what happens to not specified columns. The default is `"drop"` which is why `weight` was removed, the alternative is `"passthrough"` which then retains untransformed columns.

```
ct = make_column_transformer(
  (StandardScaler(), ["volume"]),
  (OneHotEncoder(), ["cover"]),
  remainder = "passthrough"
).fit(
  books
)
```

```
ct.get_feature_names_out()
```

```
## array(['standardscaler__volume', 'onehotencoder__cover_hb', 'onehotencoder__cover_pb', 'remainder__weight'], dtype=object)
```

```
ct.transform(books)
```

```
## array([[   0.12101,    1.     ,    0.     ,  800.     ],
##        [   0.51997,    1.     ,    0.     ,  950.     ],
##        [   0.85192,    1.     ,    0.     , 1050.     ],
##        [  -1.84637,    1.     ,    0.     ,  350.     ],
##        [  -0.43936,    1.     ,    0.     ,  750.     ],
##        [  -0.62209,    1.     ,    0.     ,  600.     ],
##        [   1.16561,    1.     ,    0.     , 1075.     ],
##        [  -1.31951,    0.     ,    1.     ,  250.     ],
##        [   0.3281 ,    0.     ,    1.     ,  700.     ],
```

Column selection

One lingering issue with the above approach is that we've had to hard code the column names (can also use indexes). Often we want to select columns based on their dtype (e.g. categorical vs numerical) this can be done via pandas or sklearn,

```
from sklearn.compose import make_column_selector
```

```
ct = make_column_transformer(
  ( StandardScaler(),
    make_column_selector(dtype_include=np.number)),
  ( OneHotEncoder(),
    make_column_selector(dtype_include=[object, bool]))
)

ct.fit_transform(books)
```

```
## array([[ 0.12101,  0.35936,  1.     ,  0.     ],
##        [ 0.51997,  0.9369 ,  1.     ,  0.     ],
##        [ 0.85192,  1.32193,  1.     ,  0.     ],
##        [-1.84637, -1.37326,  1.     ,  0.     ],
##        [-0.43936,  0.16685,  1.     ,  0.     ],
##        [-0.62209, -0.4107 ,  1.     ,  0.     ],
##        [ 1.16561,  1.41818,  1.     ,  0.     ],
##        [-1.31951, -1.75829,  0.     ,  1.     ],
##        [ 0.3281 , -0.02567,  0.     ,  1.     ],
##        [ 0.25501, -0.21818,  0.     ,  1.     ],
##        [ 1.96962,  1.03316,  0.     ,  1.     ],
```

```
ct = make_column_transformer(
  ( StandardScaler(),
    books.select_dtypes(include=['number']).columns ),
  ( OneHotEncoder(),
    books.select_dtypes(include=['object']).columns )
)

ct.fit_transform(books)
```

```
## array([[ 0.12101,  0.35936,  1.     ,  0.     ],
##        [ 0.51997,  0.9369 ,  1.     ,  0.     ],
##        [ 0.85192,  1.32193,  1.     ,  0.     ],
##        [-1.84637, -1.37326,  1.     ,  0.     ],
##        [-0.43936,  0.16685,  1.     ,  0.     ],
##        [-0.62209, -0.4107 ,  1.     ,  0.     ],
##        [ 1.16561,  1.41818,  1.     ,  0.     ],
##        [-1.31951, -1.75829,  0.     ,  1.     ],
##        [ 0.3281 , -0.02567,  0.     ,  1.     ],
##        [ 0.25501, -0.21818,  0.     ,  1.     ],
##        [ 1.96962,  1.03316,  0.     ,  1.     ],
```

`make_column_selector()` also supports selecting via `pattern` or excluding via `dtype_exclude`

Demo 1 - Putting it together
Interaction model

Cross validation & hyper parameter tuning

hw2 ridge regression data

```
d = pd.read_csv("data/ridge.csv")
d
```

```
##             y        x1        x2        x3        x4 x5
## 0   -0.151710  0.353658  1.633932  0.553257  1.415731  A
## 1    3.579895  1.311354  1.457500  0.072879  0.330330  B
## 2    0.768329 -0.744034  0.710362 -0.246941  0.008825  B
## 3    7.788646  0.806624 -0.228695  0.408348 -2.481624  B
## 4    1.394327  0.837430 -1.091535 -0.860979 -0.810492  A
## ..        ...       ...       ...       ...       ... ..
## 495 -0.204932 -0.385814 -0.130371 -0.046242  0.004914  A
## 496  0.541988  0.845885  0.045291  0.171596  0.332869  A
## 497 -1.402627 -1.071672 -1.716487 -0.319496 -1.163740  C
## 498 -0.043645  1.744800 -0.010161  0.422594  0.772606  A
## 499 -1.550276  0.910775 -1.675396  1.921238 -0.232189  B
##
## [500 rows x 6 columns]
```

```
d = pd.get_dummies(d)
d
```

```
##             y        x1        x2        x3        x4  x5_A  x5_B  x5_C  x5_D
## 0   -0.151710  0.353658  1.633932  0.553257  1.415731     1     0     0     0
## 1    3.579895  1.311354  1.457500  0.072879  0.330330     0     1     0     0
## 2    0.768329 -0.744034  0.710362 -0.246941  0.008825     0     1     0     0
## 3    7.788646  0.806624 -0.228695  0.408348 -2.481624     0     1     0     0
## 4    1.394327  0.837430 -1.091535 -0.860979 -0.810492     1     0     0     0
## ..        ...       ...       ...       ...       ...   ...   ...   ...   ...
## 495 -0.204932 -0.385814 -0.130371 -0.046242  0.004914     1     0     0     0
## 496  0.541988  0.845885  0.045291  0.171596  0.332869     1     0     0     0
```

Fitting a ridge regession model

The `linear_model` submodule also contains the `Ridge` model which can be used to fit a ridge regression, usage is identical other than `Ridge()` takes the parameter `alpha` to specify the regularization strength.

```
from sklearn.linear_model import Ridge, LinearRegression

X, y = d.drop(["y"], axis=1), d.y

rg = Ridge(fit_intercept=False, alpha=10).fit(X, y)
lm = LinearRegression(fit_intercept=False).fit(X, y)
```

```
rg.coef_
```

```
## array([ 0.97809,  1.96215,  0.00172, -2.94457,  0.45558,  0.09001, -0.28193,  0.79781])
```

```
lm.coef_
```

```
## array([ 0.99505,  2.00762,  0.00232, -3.00088,  0.49329,  0.10193, -0.29413,  1.00856])
```

```
mean_squared_error(y, rg.predict(X))
```

```
## 0.01910143134988385
```

Generally for a Ridge (or Lasso) model it is important to scale the features before fitting - in this case this is not necessary as $(x_1,\ldots,x_4)$ all have mean of ~0 and std dev of ~1

Test-Train split

The most basic form of CV is to split the data into a testing and training set, this can be achieved using `train_test_split` from the `model_selection` submodule.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
```

```
X.shape
```

```
## (500, 8)
```

```
y.shape
```

```
## (500,)
```

```
X_train.shape
```

```
## (400, 8)
```

```
y_train.shape
```

```
## (400,)
```

```
X_test.shape
```

```
## (100, 8)
```

```
y_test.shape
```

```
## (100,)
```

```
X_train
```

```
##            x1        x2        x3        x4  x5_A  x5_B  x5_C
## 296 -0.261142 -0.887193 -0.441300  0.053902     0     0     1
## 220  0.155596  0.551363  0.749117  0.875181     0     0     1
## 0    0.353658  1.633932  0.553257  1.415731     1     0     0
## 255 -1.206309 -0.073534 -1.920777 -0.554861     1     0     0
## 335 -0.380790 -0.117404 -0.037709  0.202757     0     1     0
## ..        ...       ...       ...       ...   ...   ...   ...
## 204 -2.646094  1.170804 -0.185098  0.165830     0     1     0
## 53  -0.483511  0.452531  0.223226 -0.753872     0     1     0
## 294 -1.424818 -0.396870 -0.595927 -1.114747     1     0     0
## 211 -1.000845 -0.842665  0.407765  0.375650     0     1     0
## 303  1.037404 -0.961266  0.433180  0.890055     0     1     0
##
## [400 rows x 8 columns]
```

```
y_train
```

```
## 296   -2.462944
## 220   -1.760134
## 0     -0.151710
## 255    0.668016
## 335   -1.178652
##           ...
## 204   -0.657622
## 53     2.831201
## 294    1.566109
## 211   -3.711740
## 303   -3.552971
## Name: y, Length: 400, dtype: float64
```

Train vs Test rmse

```
alpha = np.logspace(-2,1, 100)
train_rmse = []
test_rmse = []

for a in alpha:
    rg = Ridge(alpha=a).fit(X_train, y_train)

    train_rmse.append(
      mean_squared_error(y_train, rg.predict(X_train), squared=False)
    )
    test_rmse.append(
      mean_squared_error(y_test, rg.predict(X_test), squared=False)
    )

res = pd.DataFrame(data = {"alpha": alpha, "train_rmse": train_rmse, "test_rmse": test_rmse})
res
```

```
##         alpha  train_rmse  test_rmse
## 0    0.010000    0.097568   0.106985
## 1    0.010723    0.097568   0.106984
## 2    0.011498    0.097568   0.106984
## 3    0.012328    0.097568   0.106983
## 4    0.013219    0.097568   0.106983
## ..        ...         ...        ...
```

```
g = sns.relplot(x="alpha", y="value", hue="variable", data = pd.melt(res, id_vars=["alpha"]))
g.set(xscale="log")
```

Best alpha?

```
min_i = np.argmin(res.train_rmse)
min_i
```

```
## 0
```

```
res.iloc[[min_i],:]
```

```
##    alpha  train_rmse  test_rmse
## 0   0.01    0.097568   0.106985
```

```
min_i = np.argmin(res.test_rmse)
min_i
```

```
## 58
```

```
res.iloc[[min_i],:]
```

```
##        alpha  train_rmse  test_rmse
## 58  0.572237    0.097787     0.1068
```

k-fold cross validation

The previous approach was relatively straight forward, but it required a fair bit of book keeping code to implement and we only examined a single test train split. If we would like to perform k-fold cross validation we can use `cross_val_score` from the `model_selection` submodule.

```
from sklearn.model_selection import cross_val_score

cross_val_score(
  Ridge(alpha=0.59, fit_intercept=False),
  X, y,
  cv=5,
  scoring="neg_root_mean_squared_error"
)
```

```
## array([-0.09364, -0.09995, -0.10474, -0.10273, -0.10597])
```

Note that the default k-fold cross validation used here does not shuffle your data which can be massively problematic if your data is ordered

Controling k-fold behavior

Rather than providing `cv` as an integer, it is better to specify a cross-validation scheme directly (with additional options). Here we will use the `KFold` class from the `model_selection` submodule.

```
from sklearn.model_selection import KFold

cross_val_score(
  Ridge(alpha=0.59, fit_intercept=False),
  X, y,
  cv = KFold(n_splits=5, shuffle=True, random_state=1234),
  scoring="neg_root_mean_squared_error"
)
```

```
## array([-0.10658, -0.104  , -0.1037 , -0.10125, -0.09228])
```

KFold object

`KFold()` returns a class object which provides the method `split()` which in turn is a generator that returns a tuple with the indexes of the training and testing selects for each fold given a model matrix `X`,

```
ex = pd.DataFrame(data = list(range(10)), columns=["x"])

cv = KFold(5)
for train, test in cv.split(ex):
  print(f'Train: {train} | test: {test}')
```

```
## Train: [2 3 4 5 6 7 8 9] | test: [0 1]
## Train: [0 1 4 5 6 7 8 9] | test: [2 3]
## Train: [0 1 2 3 6 7 8 9] | test: [4 5]
## Train: [0 1 2 3 4 5 8 9] | test: [6 7]
## Train: [0 1 2 3 4 5 6 7] | test: [8 9]
```

```
cv = KFold(5, shuffle=True, random_state=1234)
for train, test in cv.split(ex):
  print(f'Train: {train} | test: {test}')
```

```
## Train: [0 1 3 4 5 6 8 9] | test: [2 7]
## Train: [0 2 3 4 5 6 7 8] | test: [1 9]
## Train: [1 2 3 4 5 6 7 9] | test: [0 8]
```

Train vs Test rmse (again)

```
alpha = np.logspace(-2,1, 30)
test_mean_rmse = []
test_rmse = []
cv = KFold(n_splits=5, shuffle=True, random_state=1234)

for a in alpha:
    rg = Ridge(fit_intercept=False, alpha=a).fit(X_train, y_train)

    scores = -1 * cross_val_score(
      rg, X, y,
      cv = cv,
      scoring="neg_root_mean_squared_error"
    )
    test_mean_rmse.append(np.mean(scores))
    test_rmse.append(scores)

res = pd.DataFrame(
    data = np.c_[alpha, test_mean_rmse, test_rmse],
    columns = ["alpha", "mean_rmse"] + ["fold" + str(i) for i in range(1,6) ]
)
res
```

```
##         alpha  mean_rmse     fold1     fold2     fold3     fold4     fold5
## 0    0.010000   0.101257  0.106979  0.103691  0.102288  0.101130  0.092195
```

```
g = sns.relplot(x="alpha", y="value", hue="variable", data=res.melt(id_vars=["alpha"]), marker="o", kind=
g.set(xscale="log")
```

Best alpha? (again)

```
i = res.drop(
  ["alpha"], axis=1
).agg(
  np.argmin
).to_numpy()

i = np.sort(np.unique(i))

res.iloc[ i, : ]
```

```
##        alpha  mean_rmse     fold1     fold2     fold3     fold4     fold5
## 0   0.010000   0.101257  0.106979  0.103691  0.102288  0.101130  0.092195
## 5   0.032903   0.101256  0.106951  0.103696  0.102328  0.101116  0.092190
## 12  0.174333   0.101276  0.106800  0.103739  0.102607  0.101060  0.092174
## 13  0.221222   0.101291  0.106758  0.103758  0.102710  0.101055  0.092175
## 18  0.727895   0.101729  0.106580  0.104128  0.104149  0.101420  0.092367
```

Aside - Available metrics

For most of the cross validation functions we pass in a string instead of a scoring function from the metrics submodule - if you are interested in seeing the names of the possible metrics, these are available via the `sklearn.metrics.SCORERS` dictionary,

```
np.array( sorted(
  sklearn.metrics.SCORERS.keys()
) )
```

```
## array(['accuracy', 'adjusted_mutual_info_score', 'adjusted_rand_score', 'average_precision', 'balanced_accura
##        'f1_samples', 'f1_weighted', 'fowlkes_mallows_score', 'homogeneity_score', 'jaccard', 'jaccard_macro'
##        'neg_brier_score', 'neg_log_loss', 'neg_mean_absolute_error', 'neg_mean_absolute_percentage_error', 'n
##        'neg_mean_squared_log_error', 'neg_median_absolute_error', 'neg_root_mean_squared_error', 'normalized_
##        'precision_samples', 'precision_weighted', 'r2', 'rand_score', 'recall', 'recall_macro', 'recall_micro
##        'roc_auc_ovr', 'roc_auc_ovr_weighted', 'top_k_accuracy', 'v_measure_score'], dtype='<U34')
```

Grid Search

We can further reduce the amount of code needed if there is a specific set of parameter values we would like to explore using cross validation. This is done using the `GridSearchCV` function from the `model_selection` submodule.

```
from sklearn.model_selection import GridSearchCV

gs = GridSearchCV(
  Ridge(fit_intercept=False),
  {"alpha": np.logspace(-2, 1, 30)},
  cv = KFold(5, shuffle=True, random_state=1234),
  scoring = "neg_root_mean_squared_error"
).fit(
  X, y
)
```

```
gs.best_index_
```

```
## 5
```

```
gs.best_params_
```

```
## {'alpha': 0.03290344562312668}
```

`best_estimator_` attribute

If `refit = True` (the default) with `GridSearchCV()` then the `best_estimator_` attribute will be available which gives direct access to the "best" model or pipeline object. This model is constructed by using the parameter(s) that achieved the maximum score and refitting the model to the complete data set.

```
gs.best_estimator_
```

```
## Ridge(alpha=0.03290344562312668, fit_intercept=False)
```

```
gs.best_estimator_.coef_
```

```
## array([ 0.99499,  2.00747,  0.00231, -3.0007 ,  0.49316,  0.10189, -0.29408,  1.00767])
```

```
gs.best_estimator_.predict(X)
```

```
## array([ -0.12179,   3.34151,   0.76055,   7.89292,   1.56523,  -5.33575,  -4.37469,   3.13003,  -0.16859,  -
##         -1.96548,   2.99039,   0.56796,  -5.26672,   5.4966 ,   3.47247,  -2.66117,   3.35011,   0.64221,  -
##          0.76008,   5.49779,   2.6521 ,  -0.83127,   0.04167,  -1.92585,  -2.48865,   2.29127,   3.62514,  -
##         -2.78598, -12.55143,   2.79189,  -1.89763,  -5.1769 ,   1.87484,   2.18345,  -6.45358,   0.91006,
##          1.04564,  -1.54843,   0.76161,  -1.65495,   0.22378,  -0.68221,   0.12976,   2.58875,   2.54421,  -
##          0.36935,   0.87397,   9.22348,  -1.29078,   1.74347,  -1.55169,  -0.69398,  -1.40445,   0.23072,
##          1.70208,   7.15821,   3.96172,   5.75363,  -4.50718,  -5.81785,  -2.47424,   1.19276,   2.57431,  -
```

`cv_results_` attribute

Other useful details about the grid search process are stored in the dictionary `cv_results_` attribute which includes things like average test scores, fold level test scores, test ranks, test runtimes, etc.

```
gs.cv_results_.keys()
```

```
## dict_keys(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time', 'param_alpha', 'params', 'sp
```

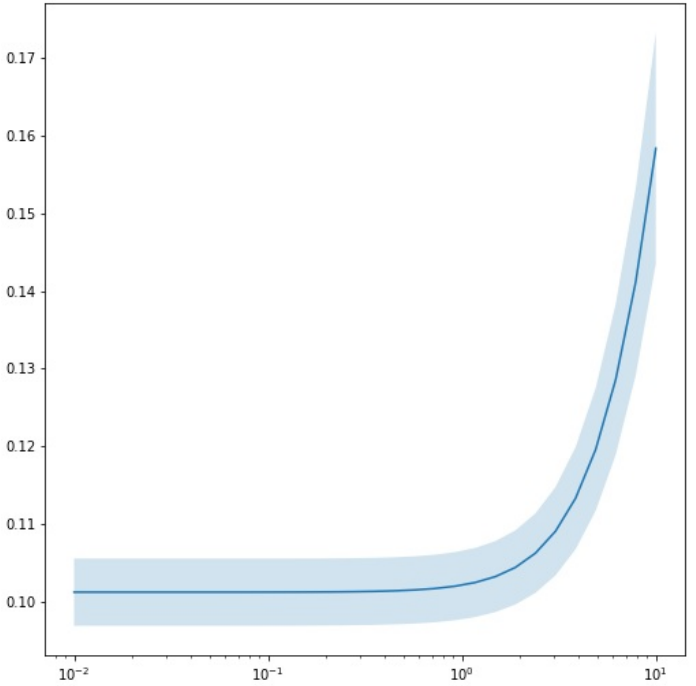
```
gs.cv_results_["param_alpha"]
```

```
## masked_array(data=[0.01, 0.01268961003167922, 0.01610262027560939, 0.020433597178569417, 0.02592943797404667
##                    0.08531678524172806, 0.10826367338740546, 0.1373823795883263, 0.17433288221999882, 0.2212
##                    0.5736152510448679, 0.727895384398315, 0.9236708571873861, 1.1721022975334805, 1.48735210
##                    4.893900918477494, 6.2101694189156165, 7.880462815669913, 10.0],
##              mask=[False, False, False, False, False, False, False, False, False, False, False, False, False
##                    False, False, False, False, False],
##        fill_value='?',
##             dtype=object)
```

```
gs.cv_results_["mean_test_score"]
```

```
## array([-0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0
```

```
alpha = np.array(gs.cv_results_["param_alpha"],dtype="float64")
score = -gs.cv_results_["mean_test_score"]
score_std = gs.cv_results_["std_test_score"]
n_folds = gs.cv.get_n_splits()

plt.figure(layout="constrained")

ax = sns.lineplot(x=alpha, y=score)
ax.set_xscale("log")

plt.fill_between(
  x = alpha,
  y1 = score + 1.96*score_std / np.sqrt(n_folds),
  y2 = score - 1.96*score_std / np.sqrt(n_folds),
  alpha = 0.2
)

plt.show()
```

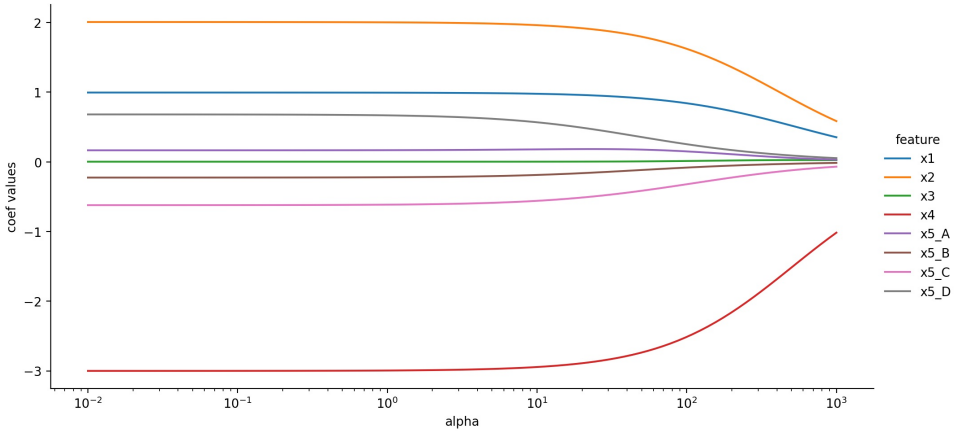


Ridge traceplot

```
alpha = np.logspace(-2,3, 100)
betas = []

for a in alpha:
    rg = Ridge(alpha=a).fit(X, y)

    betas.append(rg.coef_)

res = pd.DataFrame(
  data = betas, columns = rg.feature_names_in_
).assign(
  alpha = alpha
)

res
```

```
##           x1        x2        x3  ...      x5_C      x5_D        alpha
## 0   0.995032  2.007574  0.002317  ... -0.621467  0.681007     0.010000
## 1   0.995030  2.007568  0.002317  ... -0.621458  0.680990     0.011233
## 2   0.995027  2.007562  0.002317  ... -0.621448  0.680971     0.012619
## 3   0.995024  2.007555  0.002317  ... -0.621437  0.680949     0.014175
## 4   0.995021  2.007547  0.002317  ... -0.621424  0.680925     0.015923
## ..       ...       ...       ...  ...       ...       ...          ...
## 95  0.464132  0.796323  0.025934  ... -0.101339  0.077138   628.029144
```

```
g = sns.relplot(
  data = res.melt(id_vars="alpha", value_name="coef values", var_name="feature"),
  x = "alpha", y = "coef values", hue = "feature",
  kind = "line", aspect=2
)
g.set(xscale="log")
```



Exercise 1

Obtain the diabetes dataset from sklearn using the following code,

```
from sklearn import datasets
X, y = datasets.load_diabetes(return_X_y=True)
```

Our goal is to fit a Lasso model to these data and determine an optimal value of `alpha` using cross validation. Make sure to perform each of the following:

- Verify whether scaling is necessary for these data
- Even if scaling is not necessary, implement a pipeline that integrates `StandardScaler()` and `Lasso()`
- Find the "optimal" value of `alpha` using `GridSearchCV()` and an appropriate metric, how robust does this result appear to be?
- Time permitting, construct a traceplot of coefficients from the lasso models as a function of `alpha`

Dataset details

```
datasets.load_diabetes()["feature_names"]
```

```
## ['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
```

```
print(datasets.load_diabetes()["DESCR"])
```

```
## .. _diabetes_dataset:
##
## Diabetes dataset
## ----------------
##
## Ten baseline variables, age, sex, body mass index, average blood
## pressure, and six blood serum measurements were obtained for each of n =
## 442 diabetes patients, as well as the response of interest, a
## quantitative measure of disease progression one year after baseline.
##
## **Data Set Characteristics:**
##
##   :Number of Instances: 442
##
##   :Number of Attributes: First 10 columns are numeric predictive values
##
##   :Target: Column 11 is a quantitative measure of disease progression one year after baseline
##
##   :Attribute Information:
##       - age     age in years
##       - sex
##       - bmi     body mass index
##       - bp      average blood pressure
##       - s1      tc, total serum cholesterol
```