Machine Learning Practice

Objective: After this assignment, you can build a pipeline that

1. Preprocesses realistic data (multiple variable types) in a pipeline that handles each variable type
2. Estimates a model using CV
3. Hypertunes a model on a CV folds within training sample
4. Finally, evaluate its performance in the test sample

Let's start by loading the data

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from df_after_transform import df_after_transform
from sklearn import set_config
from sklearn.calibration import CalibrationDisplay
from sklearn.compose import (
    ColumnTransformer,
    make_column_selector,
    make_column_transformer,
)
from sklearn.decomposition import PCA
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.feature_selection import (
    RFECV,
    SelectFromModel,
    SelectKBest,
    SequentialFeatureSelector,
    f_classif,
)
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Lasso, LassoCV, LogisticRegression,Ridge
from sklearn.metrics import (
    ConfusionMatrixDisplay,
    DetCurveDisplay,
    PrecisionRecallDisplay,
    RocCurveDisplay,
    classification_report,
    make_scorer,
)
from sklearn.model_selection import (
    GridSearchCV,
    KFold,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import (
    OneHotEncoder,
    OrdinalEncoder,
    PolynomialFeatures,
    StandardScaler,
)
from sklearn.svm import LinearSVC
from sklearn.feature_selection import RFE
from sklearn.metrics import r2_score

# load data and split off X and y
housing = pd.read_csv('input_data2/housing_train.csv')
```

```
y = np.log(housing.v_SalePrice)
housing = housing.drop('v_SalePrice',axis=1)
```

To ensure you can be graded accurately, we need to make the "randomness" predictable. (I.e. you should get the exact same answers every single time we run this.)

Per the recommendations in the sk-learn documentation, what that means is we need to put `random_state=rng` inside every function in this file that accepts "random_state" as an argument.

In [3]:
```
# create test set for use later - notice the (random_state=rng)
rng = np.random.RandomState(0)
X_train, X_test, y_train, y_test = train_test_split(housing, y, random_state=rng)
```

In [6]:
```
y_train
```

Out[6]:
```
529     12.305918
1203    12.744444
378     12.138864
568     11.957611
1494    12.007622
          ...
835     11.744037
1216    12.429216
1653    11.890677
559     11.838626
684     11.964001
Name: v_SalePrice, Length: 1455, dtype: float64
```

In [5]:
```
X_train.columns
```

Out[5]:
```
Index(['parcel', 'v_MS_SubClass', 'v_MS_Zoning', 'v_Lot_Frontage',
       'v_Lot_Area', 'v_Street', 'v_Alley', 'v_Lot_Shape', 'v_Land_Contour',
       'v_Utilities', 'v_Lot_Config', 'v_Land_Slope', 'v_Neighborhood',
       'v_Condition_1', 'v_Condition_2', 'v_Bldg_Type', 'v_House_Style',
       'v_Overall_Qual', 'v_Overall_Cond', 'v_Year_Built', 'v_Year_Remod/Add',
       'v_Roof_Style', 'v_Roof_Matl', 'v_Exterior_1st', 'v_Exterior_2nd',
       'v_Mas_Vnr_Type', 'v_Mas_Vnr_Area', 'v_Exter_Qual', 'v_Exter_Cond',
       'v_Foundation', 'v_Bsmt_Qual', 'v_Bsmt_Cond', 'v_Bsmt_Exposure',
       'v_BsmtFin_Type_1', 'v_BsmtFin_SF_1', 'v_BsmtFin_Type_2',
       'v_BsmtFin_SF_2', 'v_Bsmt_Unf_SF', 'v_Total_Bsmt_SF', 'v_Heating',
       'v_Heating_QC', 'v_Central_Air', 'v_Electrical', 'v_1st_Flr_SF',
       'v_2nd_Flr_SF', 'v_Low_Qual_Fin_SF', 'v_Gr_Liv_Area',
       'v_Bsmt_Full_Bath', 'v_Bsmt_Half_Bath', 'v_Full_Bath', 'v_Half_Bath',
       'v_Bedroom_AbvGr', 'v_Kitchen_AbvGr', 'v_Kitchen_Qual',
       'v_TotRms_AbvGrd', 'v_Functional', 'v_Fireplaces', 'v_Fireplace_Qu',
       'v_Garage_Type', 'v_Garage_Yr_Blt', 'v_Garage_Finish', 'v_Garage_Cars',
       'v_Garage_Area', 'v_Garage_Qual', 'v_Garage_Cond', 'v_Paved_Drive',
       'v_Wood_Deck_SF', 'v_Open_Porch_SF', 'v_Enclosed_Porch', 'v_3Ssn_Porch',
       'v_Screen_Porch', 'v_Pool_Area', 'v_Pool_QC', 'v_Fence',
       'v_Misc_Feature', 'v_Misc_Val', 'v_Mo_Sold', 'v_Yr_Sold', 'v_Sale_Type',
       'v_Sale_Condition'],
      dtype='object')
```

Part 1: Preprocessing the data

1. Set up a single pipeline called `preproc_pipe` to preprocess the data.
   A. For **all** numerical variables, impute missing values with SimpleImputer and scale them with StandardScaler
   B. `v_Lot_Config` : Use OneHotEncoder on it
   C. Drop any other variables (handle this **inside** the pipeline)
2. Use this pipeline to preprocess X_train.

A. Describe the resulting data **with two digits.**

B. How many columns are in this object?

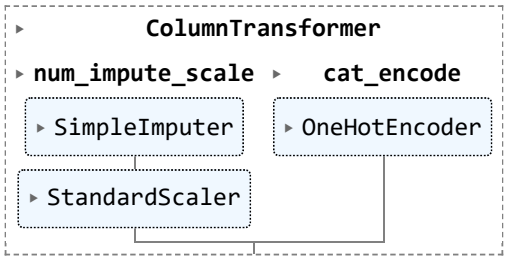
HINTS:

- *You do NOT need to type the names of all variables. There is a lil trick to catch all the variables.*
- *The first few rows of my print out look like this:*

	count	mean	std	min	25%	50%	75%	max
v_MS_SubClass	1455	0	1	-0.89	-0.89	-0.2	0.26	3.03
v_Lot_Frontage	1455	0	1	-2.2	-0.43	0	0.39	11.07
v_Lot_Area	1455	0	1	-1.17	-0.39	-0.11	0.19	20.68
v_Overall_Qual	1455	0	1	-3.7	-0.81	-0.09	0.64	2.8

In [4]:
```
# answering Part1 q1
numerical_cols = X_train.select_dtypes(include=['int64', 'float64']).columns.tolist()
numer_pipe = make_pipeline(SimpleImputer(strategy='mean'), StandardScaler())
cat_pipe = make_pipeline(OneHotEncoder(drop='first'))

preproc_pipe = ColumnTransformer([("num_impute_scale", numer_pipe, numerical_cols),("cat_encode",
preproc_pipe
```

Out[4]:



In [5]:
```
from df_after_transform import df_after_transform

preproc_df = df_after_transform(preproc_pipe,X_train)
print(f'There are {preproc_df.shape[1]} columns in the preprocessed data.')
preproc_df.describe().T.round(2).iloc[:4,:]
```

```
There are 40 columns in the preprocessed data.
```

Out[5]:

	count	mean	std	min	25%	50%	75%	max
v_MS_SubClass	1455.0	0.0	1.0	-0.89	-0.89	-0.20	0.26	3.03
v_Lot_Frontage	1455.0	0.0	1.0	-2.20	-0.43	0.00	0.39	11.07
v_Lot_Area	1455.0	0.0	1.0	-1.17	-0.39	-0.11	0.19	20.68
v_Overall_Qual	1455.0	0.0	1.0	-3.70	-0.81	-0.09	0.64	2.80

Part 2: Estimating one model

Note: A Lasso model is basically OLS, but it pushes some coefficients to zero. Read more in the `skLearn` User Guide.

1. Report the mean test score (**show 5 digits**) when you use cross validation on a Lasso Model (after using the preprocessor from Part 1) with
   - alpha = 0.3,
   - CV uses 10 `KFold` s

- R^2 scoring

2. Now, still using CV with 10 `KFold`s and R^2 scoring, let's find the optimal alpha for the lasso model. You should optimize the alpha out to the exact fifth digit that yields the highest R2.
   A. According to the CV function, what alpha leads to the highest *average* R2 across the validation/test folds? (**Show 5 digits.**)
   B. What is the mean test score in the CV output for that alpha? (**Show 5 digits.**)
   C. After fitting your optimal model on **all** of X_train, how many of the variables did it select? (Meaning: How many coefficients aren't zero?)
   D. After fitting your optimal model on **all** of X_train, report the 5 highest *non-zero* coefficients (Show the names of the variables and the value of the coefficients.)
   E. After fitting your optimal model on **all** of X_train, report the 5 lowest *non-zero* coefficients (Show the names of the variables and the value of the coefficients.)
   F. After fitting your optimal model on **all** of X_train, now use your predicted coefficients on the test ("holdout") set! What's the R2?

In [6]:
```
# answering Part2 q1
lasso_pipe = make_pipeline(preproc_pipe, Lasso(alpha=0.3))
# lasso_pipeline = Pipeline(steps=[('preprocessor', preproc_pipe),('model', Lasso(alpha=0.3))])
# this is another way to add model into the pipeline
cross_validate(lasso_pipe,X_train,y_train,
               cv=KFold(10), scoring='r2')['test_score'].mean().round(5)
```

Out[6]:
```
0.08666
```

In [7]:
```
# answering Part2 q2
# A
param_grid = {'lasso__alpha': np.linspace(0.0077,0.0079,20)}
grid_search = GridSearchCV(lasso_pipe, param_grid, cv=10, scoring='r2', verbose=1)
grid_search.fit(X_train, y_train)
best_alpha = grid_search.best_params_['lasso__alpha']
best_score = grid_search.best_score_
print(f"Best alpha: {best_alpha:.5f}")
print(f"Best R^2 score: {best_score:.5f}")
```

```
Fitting 10 folds for each of 20 candidates, totalling 200 fits
Best alpha: 0.00771
Best R^2 score: 0.83108
```

In [8]:
```
# print(np.linspace(0.0077,0.0079,20))
```

In [9]:
```
# B
lasso_pipe = make_pipeline(preproc_pipe, Lasso(alpha=best_alpha))
cross_validate(lasso_pipe,X_train,y_train,
               cv=KFold(10), scoring='r2')['test_score'].mean().round(5)
```

Out[9]:
```
0.83108
```

In [10]:
```
# C
# best_alpha = grid_search.best_params_['lasso__alpha']
# redundant step
final_lasso_pipe = make_pipeline(preproc_pipe, Lasso(alpha=best_alpha))
final_lasso_pipe.fit(X_train, y_train)

coef = final_lasso_pipe.named_steps['lasso'].coef_

non_zero_coefs = np.sum(coef != 0)

print(f"Number of non-zero coefficients (selected variables): {non_zero_coefs}")
```

```
Number of non-zero coefficients (selected variables): 21
```

In [11]:
```
# D&E
feature_names = []
for transformer_name, transformer, column in preproc_pipe.transformers_:
    if transformer_name == 'num_impute_scale':

        feature_names.extend(column)
    elif transformer_name == 'cat_encode':

        original_feature_names = transformer.named_steps['onehotencoder'].get_feature_names_out(c

        new_feature_names = [f"{column[0]}__{name}" for name in original_feature_names]
        feature_names.extend(new_feature_names)
lasso_model = final_lasso_pipe.named_steps['lasso']
coefficients = lasso_model.coef_
df = pd.DataFrame({
    'Feature': feature_names,
    'Coefficient': coefficients
})
coef_df = df[df['Coefficient'] != 0]
highest_coefs = coef_df.reindex(coef_df.Coefficient.abs().sort_values(ascending=False).index).hea
lowest_coefs = coef_df.reindex(coef_df.Coefficient.abs().sort_values(ascending=False).index).tail
print(highest_coefs)
print('--------------------------------')
print(lowest_coefs)
```

```
           Feature  Coefficient
3   v_Overall_Qual     0.134463
15   v_Gr_Liv_Area     0.098280
5     v_Year_Built     0.066264
25   v_Garage_Cars     0.047613
4   v_Overall_Cond     0.035726
--------------------------------
            Feature  Coefficient
12     v_1st_Flr_SF     0.005065
8    v_BsmtFin_SF_1     0.004263
21  v_Kitchen_AbvGr    -0.004015
20  v_Bedroom_AbvGr     0.003921
32      v_Pool_Area    -0.002427
```

My understanding for the part above

- When comparing coefficients, we typically compare their absolute values.
- This allows us to focus on the magnitude of the effect rather than the direction.

In [12]:
```
# F
y_pred = final_lasso_pipe.predict(X_test)
r2 = r2_score(y_test, y_pred)
print(r2)
```

```
0.8654542152856219
```

Part 3: Optimizing and estimating your own model

You can walk! Let's try to run! The next skill level is trying more models and picking your favorite.

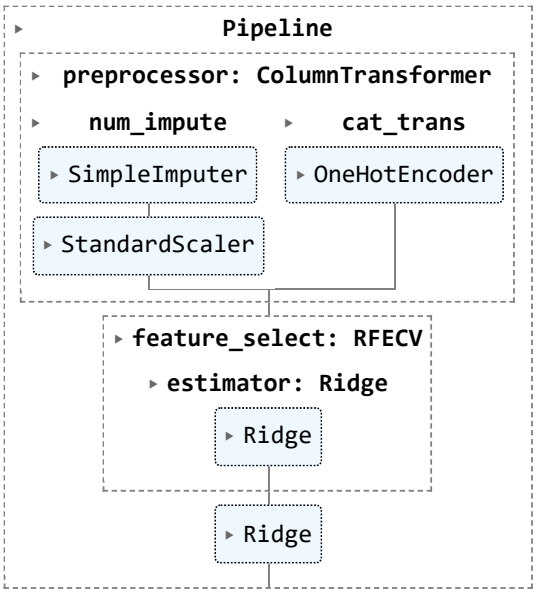
Read this whole section before starting!

1. Output 1: Build a pipeline with these 3 steps and **display the pipeline**
    A. step 1: preprocessing: possible preprocessing things you can try include imputation, scaling numerics, outlier handling, encoding categoricals, and feature creation (polynomial transformations / interactions)

    B. step 2: feature "selection": Either selectKbest, RFEcv, or PCA
    C. step 3: model estimation: e.g. a linear model or an ensemble model like HistGradientBoostingRegressor
2. Pick two hyperparameters to optimize: Both of the hyperparameters you optimize should be **numeric** in nature (i.e. not median vs mean in the imputation step). Pick parameters you think will matter the most to improve predictions.
    - Put those parameters into a grid search and run it.
3. Output 2: Describe what each of the two parameters you chose is/does, and why you thought it was important/useful to optimize it.
4. Output 3: Plot the average (on the y-axis) and STD (on the x-axis) of the CV test scores from your grid search for 25+ models you've considered. Highlight in red the dot corresponding to the model **you** prefer from this set of options, and **in the figure somewhere,** list the parameters that red dot's model uses.
    - Your plot should show at least 25 ***total*** combinations.
    - You'll try far more than 25 combinations to find your preferred model. You don't need to report them all.
5. Output 4: Tell us the set of possible values for each parameter that were reported in the last figure.
    - For example: "Param 1 could be 0.1, 0.2, 0.3, 0.4, and 0.5. Param 2 could be 0.1, 0.2, 0.3, 0.4, and 0.5." Note: Use the name of the parameter in your write up, don't call it "Param 1".
    - Adjust your gridsearch as needed so that your preferred model doesn't use a hyperparameter whose value is the lowest or highest possible value for that parameter. Meaning: If the optimal is at the high or low end for a parameter, you've *probably* not optimized it!
6. Output 5: Fit your pipeline on all of X_train using the optimal parameters you just found. Now use your predicted coefficients on the test ("holdout") set! **What's the R2 in the holdout sample with your optimized pipeline?**

In [13]:
```
# output 1
numer_pipe = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
cat_pipe = make_pipeline(OneHotEncoder(drop='first'))
preproc_pipe = ColumnTransformer([("num_impute", numer_pipe, numerical_cols), ("cat_trans", cat_p

pipe = Pipeline([('preprocessor', preproc_pipe),
    # ('feature_create', PolynomialFeatures(degree=2, interaction_only=True, include_bias=False))
    ('feature_select', RFECV(estimator=Ridge(), step=1, cv=5, scoring='neg_mean_squared_error')),
    ('reg',Ridge())
])

pipe
```

Out[13]:



In [14]:
```
# pipe.get_params()
```

output 2

1. reg__alpha (Alpha for Ridge Regression in the Final Estimator)

- Objective of Alpha: The alpha term controls the strength of regularization in ridge regression. Regularization is a method that reduces the complexity of the model. It puts penalty on coefficients in a regression equation proportional to their sizes or magnitudes. Further, the higher the value of α, stronger will be the regularization and hence more robust will be the model.
- Impact on Model: Increasing alpha increases shrinkage which can help mitigate variance and prevent overfitting by reducing model complexity but it may lead to underfitting if too large whereby it becomes difficult for it to capture pattern behind data.
- Optimization Goal: Optimization of alpha achieves balance between bias (error due to errors made during learning algorithm's assumptions) and variance (error due to variation in training data), leading to improved generalization on unseen data.

2. feature_select**estimator**alpha (Alpha for Ridge Regression within RFECV)

- RFECV or Recursive Feature Elimination with Cross Validation is a feature selection method using an external estimator among other things that estimate importance of features. The least important features are recursively eliminated based on coefficients magnitude as determined from each iteration driven by RFECV.
- Role of Alpha in RFECV: When employing Ridge regression as an estimator, α defines how much importance is placed on shrinking down coefficients as a way of identifying predictive variables. Any slight change in alpha would determine what remains significant at each step towards eliminating some predictors when using Ridge regression as an estimator within FECEH.
- Strategic Tuning: By tuning α separately for this Ridge estimators used within RFECV than those used within final ridge regression models allows one not only to potentially prioritize different variables than

might be done by those last models but also more importantly discover set(s) that either provide greater stability or better performance in modeling.

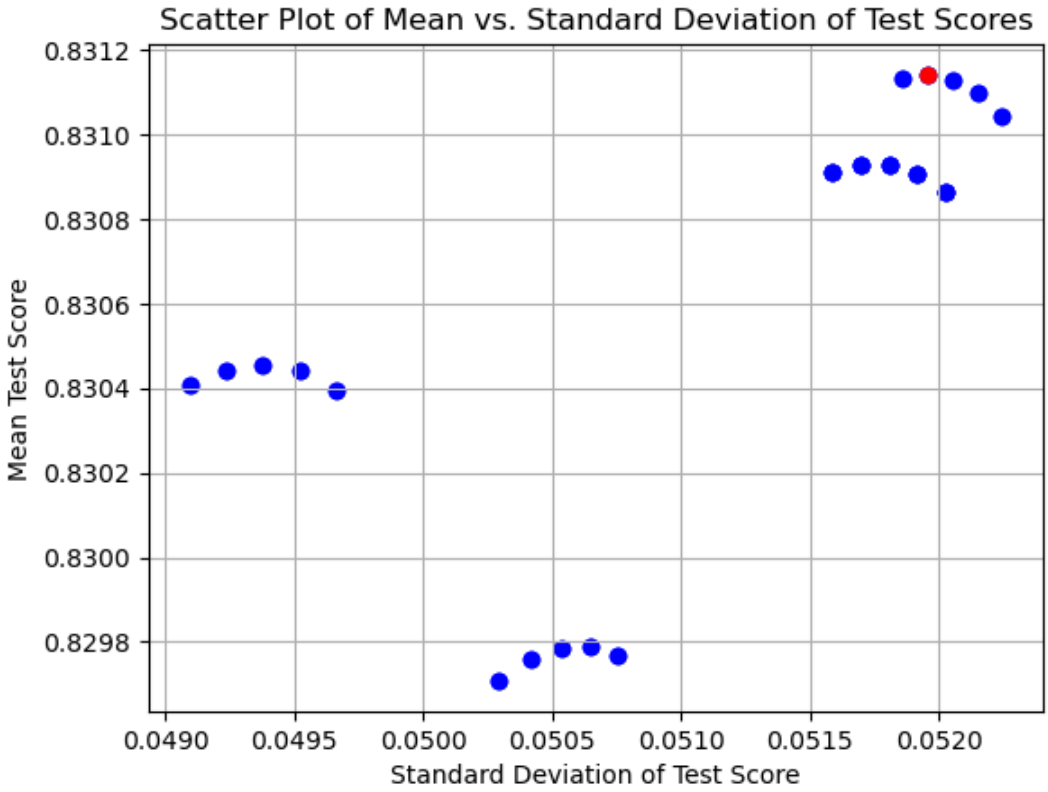
Practical Implementation

- The practical application mediated through both reg**alpha and feature_select**estimator__alpha is to try a range for each and see how different strengths of regularization at different stages of the modeling process (feature selection versus final model fitting) influence the overall performance of the pipeline.

In [15]:
```
param_grid = {
     'reg__alpha': np.linspace(50,100,5),
      'feature_select__estimator__alpha': np.linspace(1,2,5)
 }

grid_search = GridSearchCV(pipe, param_grid=param_grid, cv=5, scoring='r2',verbose=1)

results = grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_

best_score = grid_search.best_score_

best_pipeline = grid_search.best_estimator_

results_df=pd.DataFrame(results.cv_results_)
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits
```

In [16]:
```
# output 3
mean_test_scores = results_df['mean_test_score']
std_test_scores = results_df['std_test_score']
plt.scatter(y=mean_test_scores, x=std_test_scores, color='blue')
highlight_index = 13
plt.scatter(results_df['std_test_score'], results_df['mean_test_score'], color='blue')
plt.scatter(results_df.loc[highlight_index, 'std_test_score'], results_df.loc[highlight_index, 'm
plt.title('Scatter Plot of Mean vs. Standard Deviation of Test Scores')
plt.xlabel('Standard Deviation of Test Score')
plt.ylabel('Mean Test Score')
plt.grid(True)
plt.show()
```



output 4

Values of parameters from Grid Search Set up

reg_alpha:

- This parameter is responsible for controlling the regularization strength of Ridge regressor which is used in the last step of your pipeline.
- Possible Values: np.linspace(50, 100, 5) gives you an array with 5 numbers equally spaced between 50 and 100. In this particular case these would be [50.0, 62.5, 75.0, 87.5, 100.0].

feature_select_estimator_alpha:

- This parameter controls how much shrinkage will take place during fitting on training data by penalizing larger coefficients more heavily than smaller ones; it also allows us to choose variables automatically or manually based on their importance rankings according to different methods such as L1 norm etcetera; henceforth we can see that this technique uses Ridge Regression as a base model.
- Possible Values: np.linspace(1, 2, 5) generates five numbers evenly spaced from one through two inclusive – so they would be [1.0, 1.25, 1.5, 1.75, 2.0] in this case too.

Even the range of the parameters seems enormous and vague, I did find the approximate optimal value for both for the sake of experiment. For reg_alpha, the value is around 75, and for estimator_alpha, the value is around 1.7. The problem is when I'm using the "optimal range", the plot didn't display the result I expected. Thus, I left the range wide.

In [18]:
```
# output 5
best_pipeline.fit(X_train, y_train)
```

```
print(r2_score(y_test,best_pipeline.predict(X_test)))
```

```
0.8639396091000091
```