Lecture 2: Regression metrics

Contents

Imports

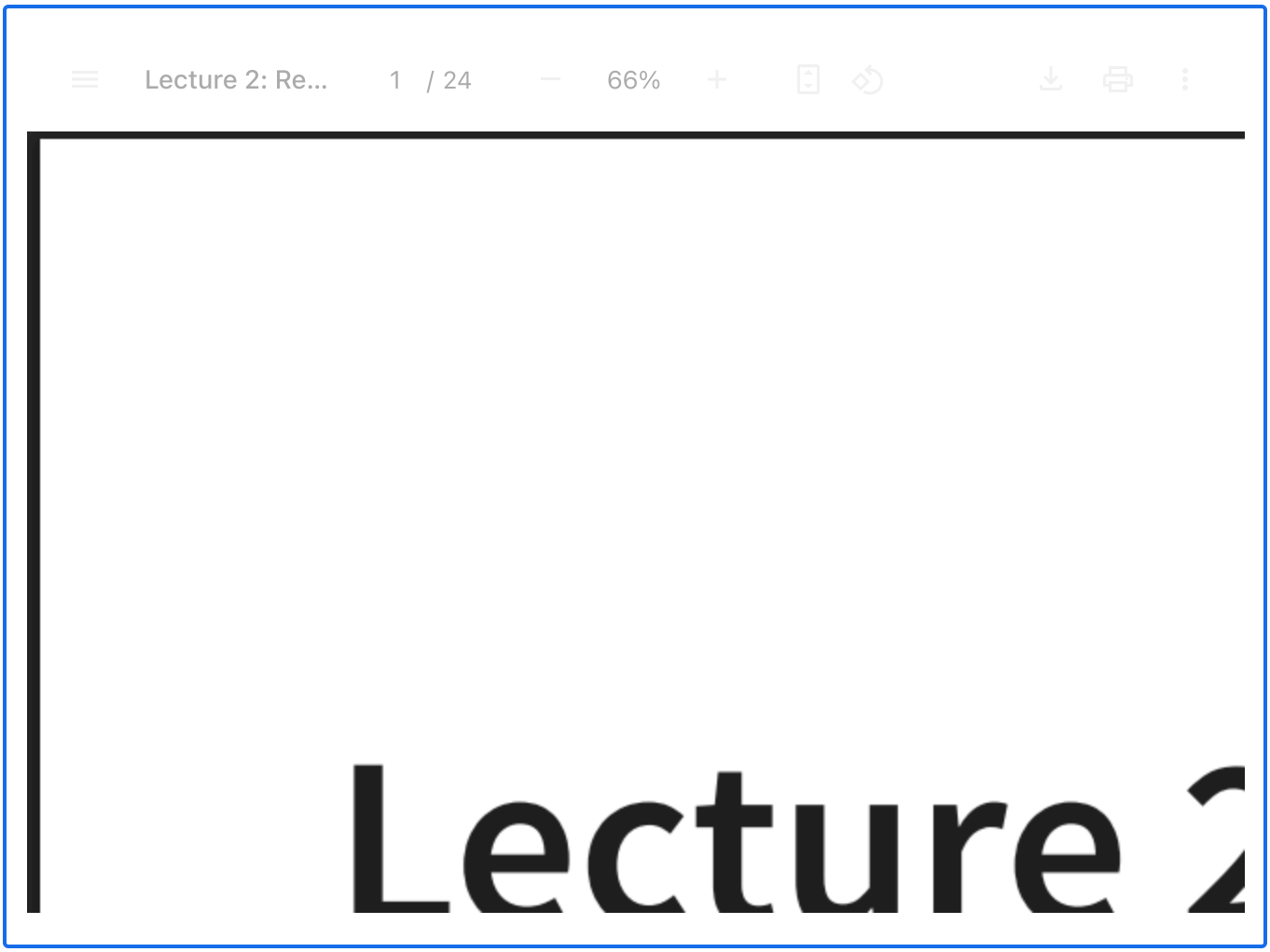
Learning outcomes

From this lecture, students are expected to be able to:

- Carry out feature transformations on somewhat complicated dataset.
- Visualize transformed features as a dataframe.
- Use `Ridge` and `RidgeCV`.
- Explain how `alpha` hyperparameter of `Ridge` relates to the fundamental tradeoff.
- Explain the effect of `alpha` on the magnitude of the learned coefficients.
- Examine coefficients of transformed features.
- Appropriately select a scoring metric given a regression problem.
- Interpret and communicate the meanings of different scoring metrics on regression problems.
  - MSE, RMSE, R^2, MAE, MAPE
- Apply log-transform on the target values in a regression problem with `TransformedTargetRegressor`.

Lecture slides

Section 1 slides **Section 2 slides**


Lecture 2: Re...
1 / 24
66%
Lecture 2

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.compose import (
    ColumnTransformer,
    TransformedTargetRegressor,
    make_column_transformer,
)
from sklearn.dummy import DummyRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge, RidgeCV
from sklearn.metrics import make_scorer, mean_squared_error, root_mean_squared
from sklearn.model_selection import (
    GridSearchCV,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScale
from sklearn.tree import DecisionTreeRegressor

# Ignore future deprecation warnings from sklearn (using `os` instead of `warn
import os
os.environ['PYTHONWARNINGS']='ignore::FutureWarning'
```

❓❓ Questions for you

iClicker Exercise 2.1

Select all of the following statements which are TRUE.

- (A) Average precision score at threshold 0.5 is the same as F1-score.
- (B) Using `class_weight="balanced"` is likely to decrease false negatives but increase false positives.
- (C) In ROC curve, the threshold goes from lower left (1.0) to upper right (0.0).
- (D) It's possible that automated tools such as `pandas-profiling` classify a categorical feature as a numeric feature.

▶ Click to see answers

After carrying out preprocessing, why it's useful to get feature names for transformed features?

Dataset [video]

In this lecture, we'll be using Kaggle House Prices dataset. As usual, to run this notebook you'll need to download the data. For this dataset, train and test have already been separated. We'll be working with the train portion in this lecture.

```
df = pd.read_csv("data/housing-kaggle-train.csv").drop(columns=['PoolQC'])
train_df, test_df = train_test_split(df, test_size=0.10, random_state=123)
train_df
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotSl
302	303	20	RL	118.0	13704	Pave	NaN	
767	768	50	RL	75.0	12508	Pave	NaN	
429	430	20	RL	130.0	11457	Pave	NaN	
1139	1140	30	RL	98.0	8731	Pave	NaN	
558	559	60	RL	57.0	21872	Pave	NaN	
...	...	...	...	...	...	...	...	
1041	1042	60	RL	NaN	9130	Pave	NaN	
1122	1123	20	RL	NaN	8926	Pave	NaN	
1346	1347	20	RL	NaN	20781	Pave	NaN	
1406	1407	85	RL	70.0	8445	Pave	NaN	
1389	1390	50	RM	60.0	6000	Pave	NaN	

1314 rows × 80 columns

- The supervised machine learning problem is predicting housing price given features associated with properties.
- Here, the target is `SalePrice`, which is continuous. So it's a **regression problem** (as opposed to classification).

Let's separate `X` and `y`

```
X_train = train_df.drop(columns=["SalePrice"])
y_train = train_df["SalePrice"]

X_test = test_df.drop(columns=["SalePrice"])
y_test = test_df["SalePrice"]
```

EDA

```
train_df.describe()
```

	Id	MSSubClass	LotFrontage	LotArea	OverallQual	Ov
count	1314.000000	1314.000000	1089.000000	1314.000000	1314.000000	13
mean	734.182648	56.472603	69.641873	10273.261035	6.076104	
std	422.224662	42.036646	23.031794	8997.895541	1.392612	
min	1.000000	20.000000	21.000000	1300.000000	1.000000	
25%	369.250000	20.000000	59.000000	7500.000000	5.000000	
50%	735.500000	50.000000	69.000000	9391.000000	6.000000	
75%	1099.750000	70.000000	80.000000	11509.000000	7.000000	
max	1460.000000	190.000000	313.000000	215245.000000	10.000000	

8 rows × 38 columns

```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1314 entries, 302 to 1389
Data columns (total 80 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Id             1314 non-null   int64  
 1   MSSubClass     1314 non-null   int64  
 2   MSZoning       1314 non-null   object 
 3   LotFrontage    1089 non-null   float64
 4   LotArea        1314 non-null   int64  
 5   Street         1314 non-null   object 
 6   Alley          81 non-null     object 
 7   LotShape       1314 non-null   object 
 8   LandContour    1314 non-null   object 
 9   Utilities      1314 non-null   object 
 10  LotConfig      1314 non-null   object 
 11  LandSlope      1314 non-null   object 
 12  Neighborhood   1314 non-null   object 
 13  Condition1     1314 non-null   object 
 14  Condition2     1314 non-null   object 
 15  BldgType       1314 non-null   object 
 16  HouseStyle     1314 non-null   object 
 17  OverallQual    1314 non-null   int64  
 18  OverallCond    1314 non-null   int64  
 19  YearBuilt      1314 non-null   int64  
 20  YearRemodAdd   1314 non-null   int64  
 21  RoofStyle      1314 non-null   object 
 22  RoofMatl       1314 non-null   object 
 23  Exterior1st    1314 non-null   object 
 24  Exterior2nd    1314 non-null   object 
 25  MasVnrType     528 non-null    object 
 26  MasVnrArea     1307 non-null   float64
 27  ExterQual      1314 non-null   object 
 28  ExterCond      1314 non-null   object 
 29  Foundation     1314 non-null   object 
 30  BsmtQual       1280 non-null   object 
 31  BsmtCond       1280 non-null   object 
 32  BsmtExposure   1279 non-null   object 
 33  BsmtFinType1   1280 non-null   object 
 34  BsmtFinSF1     1314 non-null   int64  
 35  BsmtFinType2   1280 non-null   object 
 36  BsmtFinSF2     1314 non-null   int64  
 37  BsmtUnfSF      1314 non-null   int64  
 38  TotalBsmtSF    1314 non-null   int64  
 39  Heating        1314 non-null   object 
 40  HeatingQC      1314 non-null   object 
 41  CentralAir     1314 non-null   object 
 42  Electrical     1313 non-null   object 
 43  1stFlrSF       1314 non-null   int64  
 44  2ndFlrSF       1314 non-null   int64  
 45  LowQualFinSF   1314 non-null   int64  
 46  GrLivArea      1314 non-null   int64  
 47  BsmtFullBath   1314 non-null   int64  
 48  BsmtHalfBath   1314 non-null   int64  
 49  FullBath       1314 non-null   int64  
```

```
 50  HalfBath       1314 non-null   int64
 51  BedroomAbvGr   1314 non-null   int64
 52  KitchenAbvGr   1314 non-null   int64
 53  KitchenQual    1314 non-null   object
 54  TotRmsAbvGrd   1314 non-null   int64
 55  Functional     1314 non-null   object
 56  Fireplaces     1314 non-null   int64
 57  FireplaceQu    687 non-null    object
 58  GarageType     1241 non-null   object
 59  GarageYrBlt    1241 non-null   float64
 60  GarageFinish   1241 non-null   object
 61  GarageCars     1314 non-null   int64
 62  GarageArea     1314 non-null   int64
 63  GarageQual     1241 non-null   object
 64  GarageCond     1241 non-null   object
 65  PavedDrive     1314 non-null   object
 66  WoodDeckSF     1314 non-null   int64
 67  OpenPorchSF    1314 non-null   int64
 68  EnclosedPorch  1314 non-null   int64
 69  3SsnPorch      1314 non-null   int64
 70  ScreenPorch    1314 non-null   int64
 71  PoolArea       1314 non-null   int64
 72  Fence          259 non-null    object
 73  MiscFeature    50 non-null     object
 74  MiscVal        1314 non-null   int64
 75  MoSold         1314 non-null   int64
 76  YrSold         1314 non-null   int64
 77  SaleType       1314 non-null   object
 78  SaleCondition  1314 non-null   object
 79  SalePrice      1314 non-null   int64
dtypes: float64(3), int64(35), object(42)
memory usage: 831.5+ KB
```

`pandas_profiler`

We do not have `pandas_profiling` in our course environment. You will have to install it in the environment on your own if you want to run the code below.

`conda install -c conda-forge pandas-profiling`

```
# from pandas_profiling import ProfileReport

# profile = ProfileReport(train_df, title="Pandas Profiling Report")  # , mini
# profile.to_notebook_iframe()
```

Feature types

- Do not blindly trust all the info given to you by automated tools.
- How does pandas profiling figure out the data type?
  - You can look at the Python data type and say floats are numeric, strings are categorical.
  - However, in doing so you would miss out on various subtleties such as some of the string features being ordinal rather than truly categorical.
  - Also, it will think free text is categorical.
- In addition to tools such as above, it's important to go through data description to understand the data.
- The data description for our dataset is available here.

Feature types

- We have mixed feature types and a bunch of missing values.
- Now, let's identify feature types and transformations, starting with the numerical columns

```
numeric_looking_columns = X_train.select_dtypes(include='number').columns.toli
print(numeric_looking_columns)
```

```
['Id', 'MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond', '
```

Not all columns with a numerical data type hold quantitative data, some of them are categories encoded as numbers:

```
train_df["MSSubClass"].unique()
```

```
array([ 20,  50,  30,  60, 160,  85,  90, 120, 180,  80,  70,  75, 190,
        45,  40])
```

MSSubClass: Identifies the type of dwelling involved in the sale.

```
 20  1-STORY 1946 & NEWER ALL STYLES
 30  1-STORY 1945 & OLDER
 40  1-STORY W/FINISHED ATTIC ALL AGES
 45  1-1/2 STORY - UNFINISHED ALL AGES
 50  1-1/2 STORY FINISHED ALL AGES
 60  2-STORY 1946 & NEWER
 70  2-STORY 1945 & OLDER
 75  2-1/2 STORY ALL AGES
 80  SPLIT OR MULTI-LEVEL
 85  SPLIT FOYER
 90  DUPLEX - ALL STYLES AND AGES
120  1-STORY PUD (Planned Unit Development) - 1946 & NEWER
150  1-1/2 STORY PUD - ALL AGES
160  2-STORY PUD - 1946 & NEWER
180  PUD - MULTILEVEL - INCL SPLIT LEV/FOYER
190  2 FAMILY CONVERSION - ALL STYLES AND AGES
```

Also, month sold is more of a categorical feature than a numeric feature.

```
train_df["MoSold"].unique()  # Month Sold
```

```
array([ 1,  7,  3,  5,  8, 10,  6,  9, 12,  2,  4, 11])
```

```
drop_features = ["Id"]
numeric_features = [
    "BedroomAbvGr",
    "KitchenAbvGr",
    "LotFrontage",
    "LotArea",
    "OverallQual",
    "OverallCond",
    "YearBuilt",
    "YearRemodAdd",
    "MasVnrArea",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtUnfSF",
    "TotalBsmtSF",
    "1stFlrSF",
    "2ndFlrSF",
    "LowQualFinSF",
    "GrLivArea",
    "BsmtFullBath",
    "BsmtHalfBath",
    "FullBath",
    "HalfBath",
    "TotRmsAbvGrd",
    "Fireplaces",
    "GarageYrBlt",
    "GarageCars",
    "GarageArea",
    "WoodDeckSF",
    "OpenPorchSF",
    "EnclosedPorch",
    "3SsnPorch",
    "ScreenPorch",
    "PoolArea",
    "MiscVal",
    "YrSold",
]
```

Note

I've not looked at all the features carefully. It might be appropriate to apply some other encoding on some of the numeric features above.

```
set(numeric_looking_columns) - set(numeric_features) - set(drop_features)
```

```
{'MSSubClass', 'MoSold'}
```

We'll treat the above numeric-looking features as categorical features.

- There are a bunch of ordinal features in this dataset.
- Ordinal features with the same scale
  - Poor (Po), Fair (Fa), Typical (TA), Good (Gd), Excellent (Ex)
  - These we'll be calling `ordinal_features_reg`.
- Ordinal features with different scales
  - These we'll be calling `ordinal_features_oth`.

```
ordinal_features_reg = [
    "ExterQual",
    "ExterCond",
    "BsmtQual",
    "BsmtCond",
    "HeatingQC",
    "KitchenQual",
    "FireplaceQu",
    "GarageQual",
    "GarageCond",
    # "PoolQC",
]
ordering = [
    "Po",
    "Fa",
    "TA",
    "Gd",
    "Ex",
]  # if N/A it will just impute something, per below
ordering_ordinal_reg = [ordering] * len(ordinal_features_reg)
ordering_ordinal_reg
```

```
[['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex']]
```

We'll pass the above as categories in our `OrdinalEncoder`.

- There are a bunch more ordinal features using different scales.
  - These we'll be calling `ordinal_features_oth`.

- We are encoding them separately, to facilitate setting up a column transformer later on.

```
ordinal_features_oth = [
    "BsmtExposure",
    "BsmtFinType1",
    "BsmtFinType2",
    "Functional",
    "Fence",
]
ordering_ordinal_oth = [
    ["NA", "No", "Mn", "Av", "Gd"],
    ["NA", "Unf", "LwQ", "Rec", "BLQ", "ALQ", "GLQ"],
    ["NA", "Unf", "LwQ", "Rec", "BLQ", "ALQ", "GLQ"],
    ["Sal", "Sev", "Maj2", "Maj1", "Mod", "Min2", "Min1", "Typ"],
    ["NA", "MnWw", "GdWo", "MnPrv", "GdPrv"],
]
```

The remaining features are categorical features.

```
categorical_features = list(
    set(X_train.columns)
    - set(numeric_features)
    - set(ordinal_features_reg)
    - set(ordinal_features_oth)
    - set(drop_features)
)
categorical_features
```

```
['RoofStyle',
 'RoofMatl',
 'Exterior2nd',
 'Utilities',
 'Condition2',
 'MSSubClass',
 'LotShape',
 'Alley',
 'SaleCondition',
 'CentralAir',
 'MoSold',
 'HouseStyle',
 'SaleType',
 'MiscFeature',
 'LotConfig',
 'PavedDrive',
 'Electrical',
 'Exterior1st',
 'Heating',
 'BldgType',
 'Neighborhood',
 'LandContour',
 'GarageType',
 'GarageFinish',
 'Condition1',
 'LandSlope',
 'Foundation',
 'MSZoning',
 'Street',
 'MasVnrType']
```

- We are not doing it here but we can engineer our own features too.
- Would price per square foot be a good feature to add in here?

▶ Click to view the answer

Applying feature transformations

- Since we have mixed feature types, let's use `ColumnTransformer` to apply different transformations on different features types.

```
from sklearn.compose import ColumnTransformer, make_column_transformer

numeric_transformer = make_pipeline(SimpleImputer(strategy="median"), Standard
ordinal_transformer_reg = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OrdinalEncoder(categories=ordering_ordinal_reg),
)

ordinal_transformer_oth = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=np.nan, c
)

categorical_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(handle_unknown="ignore", sparse_output=False),
)

preprocessor = make_column_transformer(
    ("drop", drop_features),
    (numeric_transformer, numeric_features),
    (ordinal_transformer_reg, ordinal_features_reg),
    (ordinal_transformer_oth, ordinal_features_oth),
    (categorical_transformer, categorical_features),
)
```

Examining the preprocessed data

```
preprocessor.fit(X_train)  # Calling fit to examine all the transformers.
preprocessor.named_transformers_
```

```
{'drop': 'drop',
 'pipeline-1': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='median
                ('standardscaler', StandardScaler())]),
 'pipeline-2': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='most_
                ('ordinalencoder',
                 OrdinalEncoder(categories=[['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                            ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                            ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                            ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                            ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                            ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                            ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                            ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
                                            ['Po', 'Fa', 'TA', 'Gd', 'Ex']]))]
 'pipeline-3': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='most_
                ('ordinalencoder',
                 OrdinalEncoder(categories=[['NA', 'No', 'Mn', 'Av', 'Gd'],
                                            ['NA', 'Unf', 'LwQ', 'Rec', 'BLQ',
                                             'ALQ', 'GLQ'],
                                            ['NA', 'Unf', 'LwQ', 'Rec', 'BLQ',
                                             'ALQ', 'GLQ'],
                                            ['Sal', 'Sev', 'Maj2', 'Maj1',
                                             'Mod', 'Min2', 'Min1', 'Typ'],
                                            ['NA', 'MnWw', 'GdWo', 'MnPrv',
                                             'GdPrv']],
                                handle_unknown='use_encoded_value',
                                unknown_value=nan))]),
 'pipeline-4': Pipeline(steps=[('simpleimputer',
                 SimpleImputer(fill_value='missing', strategy='constant')),
                ('onehotencoder',
                 OneHotEncoder(handle_unknown='ignore', sparse_output=False))]
```

To create a new df with the transformed feature values, we need to make sure we get all the correct column names from the pipeline:

```
ohe_columns = list(
    preprocessor.named_transformers_["pipeline-4"]
    .named_steps["onehotencoder"]
    .get_feature_names_out(categorical_features)
)
new_columns = (
    numeric_features + ordinal_features_reg + ordinal_features_oth + ohe_colum
)
```

```
X_train_enc = pd.DataFrame(
    preprocessor.transform(X_train), index=X_train.index, columns=new_columns
)
X_train_enc.head()
```

	BedroomAbvGr	KitchenAbvGr	LotFrontage	LotArea	OverallQual	Overa
302	0.154795	-0.222647	2.312501	0.381428	0.663680	-0.5
767	1.372763	-0.222647	0.260890	0.248457	-0.054669	1.2
429	0.154795	-0.222647	2.885044	0.131607	-0.054669	-0.5
1139	0.154795	-0.222647	1.358264	-0.171468	-0.773017	-0.5
558	0.154795	-0.222647	-0.597924	1.289541	0.663680	-0.5

5 rows × 261 columns

Remember that OHE creates additional columns for each categorical feature

```
X_train.shape
```

```
(1314, 79)
```

```
X_train_enc.shape
```

```
(1314, 261)
```

We went from 80 features to 263 features!!

Other possible preprocessing?

- There is a lot of room for improvement ...
- We're just using `SimpleImputer`.
  - In reality we'd want to go through this more carefully.
  - We may also want to drop some columns that are almost entirely missing.
- We could also check for outliers, and do other exploratory data analysis (EDA).
- But for now this is good enough ...

Model building

`DummyRegressor`

```
dummy = DummyRegressor()
pd.DataFrame(cross_validate(dummy, X_train, y_train, cv=10, return_train_score
```

	fit_time	score_time	test_score	train_score
0	0.001944	0.000748	-0.003547	0.0
1	0.001749	0.000592	-0.001266	0.0
2	0.000970	0.000329	-0.011767	0.0
3	0.000956	0.000330	-0.006744	0.0
4	0.000871	0.000307	-0.076533	0.0
5	0.000846	0.000341	-0.003133	0.0
6	0.000853	0.000324	-0.000397	0.0
7	0.001251	0.000411	-0.003785	0.0
8	0.001153	0.000502	-0.001740	0.0
9	0.001318	0.000449	-0.000117	0.0

Let's try a linear model: `Ridge`

- We are going to use `Ridge()` instead of `LinearRegression()` in this course.
- Similar to linear regression, ridge regression is also a linear model for regression.
- So the formula it uses to make predictions is the same one used for ordinary least squares (which you'll be learning in DSCI 561).
- But it has a hyperparameter `alpha` which controls the fundamental tradeoff between under and overfitting.

```
lr = make_pipeline(preprocessor, Ridge())
lr.fit(X_train, y_train);
```

Let's check some predictions to see if they are reasonable.

```
lr_preds = lr.predict(X_test)
lr_preds[:10]
```

```
array([225258.22017611,  64999.23854067, 133730.3020094 , 251777.05153863,
       128896.90816884, 207444.93617537, 336055.86532628, 162794.57630587,
       148210.32262533, 129568.33382024])
```

```
lr_preds.max(), lr_preds.min()
```

```
(409412.7810045788, 39946.12753197555)
```

```
print("Smallest coefficient: ", lr.named_steps["ridge"].coef_.min())
print("Largest coefficient:", lr.named_steps["ridge"].coef_.max())
```

```
Smallest coefficient:  -191372.77884307227
Largest coefficient: 83520.96125095767
```

Let's carry out cross-validation with `Ridge`.

```
lr_pipe = make_pipeline(preprocessor, Ridge())
pd.DataFrame(cross_validate(lr_pipe, X_train, y_train, cv=10, return_train_sco
```

	fit_time	score_time	test_score	train_score
0	0.043036	0.009936	0.861136	0.910880
1	0.040699	0.009924	0.828307	0.911911
2	0.027216	0.007182	0.769388	0.915257
3	0.027211	0.007261	0.874678	0.909382
4	0.027433	0.007210	0.855011	0.910527
5	0.027028	0.007760	0.836293	0.910054
6	0.040869	0.010217	0.827008	0.912565
7	0.040023	0.010410	0.878256	0.908701
8	0.040362	0.010487	0.295876	0.919158
9	0.040290	0.010472	0.893292	0.906986

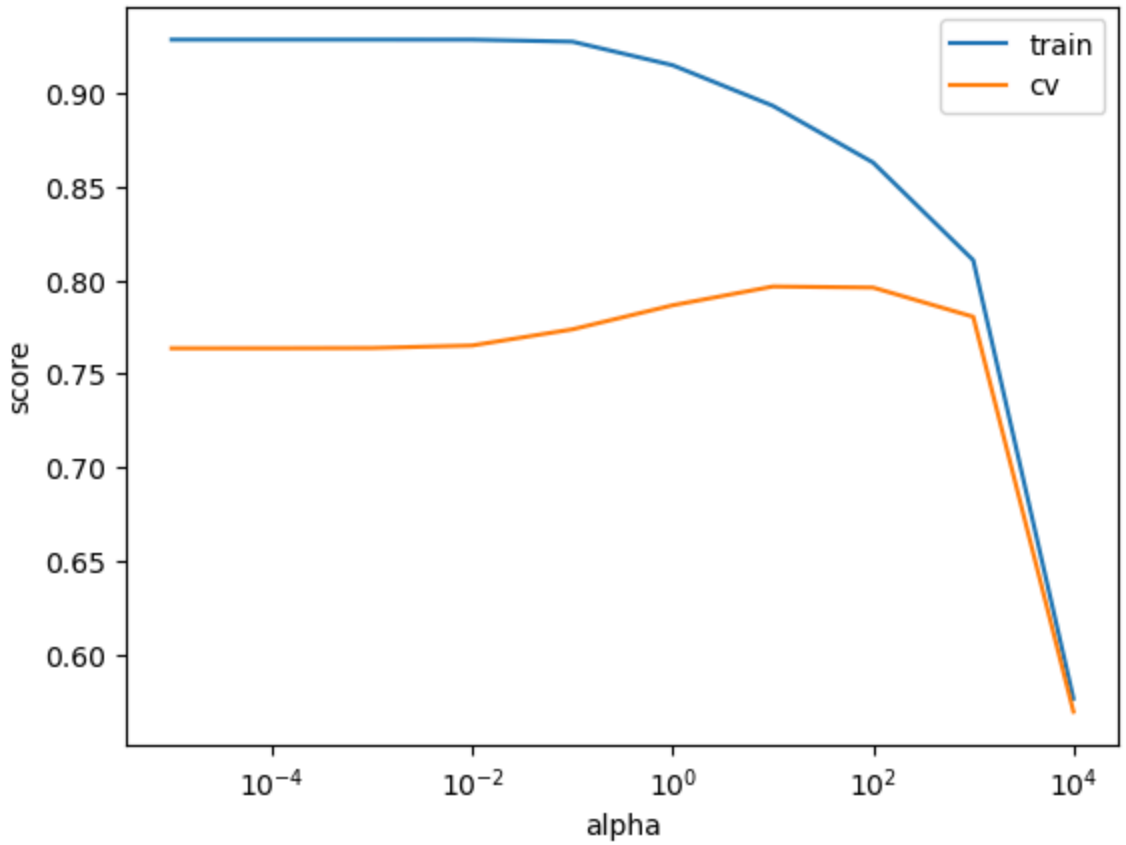
- Quite a bit of variation in the test scores.
- Performing poorly in fold 8. Not sure why.
    - Probably it contains the outliers in the data which we kind of ignored.

Tuning `alpha` hyperparameter of `Ridge`

- Recall that `Ridge` has a hyperparameter `alpha` that controls the fundamental tradeoff.
- This is like `C` in `LogisticRegression` but, annoyingly, `alpha` is the inverse of `C`.
- That is, large `C` is like small `alpha` and vice versa.
- Smaller `alpha`: lower training error (overfitting)

```
param_grid = {"ridge__alpha": 10.0 ** np.arange(-5, 5, 1)}

pipe_ridge = make_pipeline(preprocessor, Ridge())

search = GridSearchCV(pipe_ridge, param_grid, return_train_score=True, n_jobs=
search.fit(X_train, y_train)
train_scores = search.cv_results_["mean_train_score"]
cv_scores = search.cv_results_["mean_test_score"]
```

```
plt.semilogx(param_grid["ridge__alpha"], train_scores.tolist(), label="train")
plt.semilogx(param_grid["ridge__alpha"], cv_scores.tolist(), label="cv")
plt.legend()
plt.xlabel("alpha")
plt.ylabel("score");
```




```
best_alpha = search.best_params_
best_alpha
```

```
{'ridge__alpha': 10.0}
```

```
search.best_score_
```

```
0.7966886306743555
```

- It seems alpha=100 is the best choice here.
- General intuition: larger `alpha` leads to smaller coefficients.

- Smaller coefficients mean the predictions are less sensitive to changes in the data. Hence less chance of overfitting.

```
pipe_bigalpha = make_pipeline(preprocessor, Ridge(alpha=1000))
pipe_bigalpha.fit(X_train, y_train)
bigalpha_coeffs = pipe_bigalpha.named_steps["ridge"].coef_
pd.DataFrame(
    data=bigalpha_coeffs, index=new_columns, columns=["Coefficients"]
).sort_values(by="Coefficients", ascending=False)
```

	Coefficients
OverallQual	9691.749347
GrLivArea	7817.545509
1stFlrSF	5931.671138
TotRmsAbvGrd	5208.478392
GarageCars	5057.045031
...	...
SaleType_WD	-1214.419802
GarageFinish_Unf	-1281.080501
Foundation_CBlock	-1766.317670
RoofStyle_Gable	-1985.220938
KitchenAbvGr	-2623.501376

261 rows × 1 columns

- Smaller `alpha` leads to bigger coefficients.

```
pipe_smallalpha = make_pipeline(preprocessor, Ridge(alpha=0.01))
pipe_smallalpha.fit(X_train, y_train)
smallalpha_coeffs = pipe_smallalpha.named_steps["ridge"].coef_
pd.DataFrame(
    data=smallalpha_coeffs, index=new_columns, columns=["Coefficients"]
).sort_values(by="Coefficients", ascending=False)
```

	Coefficients
RoofMatl_WdShngl	128931.535524
RoofMatl_Membran	128335.817759
RoofMatl_Metal	105899.461747
Condition2_PosA	81093.444644
RoofMatl_CompShg	71474.560080
...	...
Exterior1st_ImStucc	-39136.738405
Condition2_RRAe	-66145.881890
MiscFeature_TenC	-69927.813893
Condition2_PosN	-195664.299490
RoofMatl_ClyTile	-584070.715187

261 rows × 1 columns

With the best alpha found by the grid search, the coefficients are somewhere in between.

```
pipe_bestalpha = make_pipeline(
    preprocessor, Ridge(alpha=search.best_params_["ridge__alpha"])
)
pipe_bestalpha.fit(X_train, y_train)
bestalpha_coeffs = pipe_bestalpha.named_steps["ridge"].coef_
pd.DataFrame(
    data=bestalpha_coeffs, index=new_columns, columns=["Coefficients"]
).sort_values(by="Coefficients", ascending=False)
```

	Coefficients
Neighborhood_NridgHt	28621.181558
Neighborhood_StoneBr	26213.724441
Neighborhood_NoRidge	26092.301405
RoofMatl_WdShngl	23438.239383
GrLivArea	14801.592011
...	...
Neighborhood_Gilbert	-13724.472165
LotShape_IR3	-13805.240950
Neighborhood_Edwards	-14272.663320
Condition2_PosN	-23876.331528
RoofMatl_ClyTile	-30652.823854

261 rows × 1 columns

To summarize:

- Higher values of `alpha` means a more restricted model.
- The values of coefficients are likely to be smaller for higher values of `alpha` compared to lower values of alpha.

`RidgeCV`

Because it's so common to want to tune `alpha` with `Ridge`, sklearn provides a class called `RidgeCV`, which automatically tunes `alpha` based on cross-validation, as a convenience instead of typing out the full gridsearch syntax each time.

```
alphas = 10.0 ** np.arange(-6, 6, 1)
ridgecv_pipe = make_pipeline(preprocessor, RidgeCV(alphas=alphas, cv=10))
ridgecv_pipe.fit(X_train, y_train);
```

```
best_alpha = ridgecv_pipe.named_steps["ridgecv"].alpha_

best_alpha
```

```
10.0
```

Let's examine the tuned model.

```
ridge_tuned = make_pipeline(preprocessor, Ridge(alpha=best_alpha))
ridge_tuned.fit(X_train, y_train)
ridge_preds = ridge_tuned.predict(X_test)
ridge_preds[:10]
```

```
array([226852.0984204 ,  85841.21323794, 138493.79254747, 247996.69427623,
       124356.62249424, 213958.69350431, 327547.42399962, 148013.15409263,
       152934.27365947, 127411.86088092])
```

```
df = pd.DataFrame(
    data={"coefficients": ridge_tuned.named_steps["ridge"].coef_}, index=new_c
)
```

```
df.sort_values("coefficients", ascending=False)
```

	coefficients
Neighborhood_NridgHt	28621.181558
Neighborhood_StoneBr	26213.724441
Neighborhood_NoRidge	26092.301405
RoofMatl_WdShngl	23438.239383
GrLivArea	14801.592011
...	...
Neighborhood_Gilbert	-13724.472165
LotShape_IR3	-13805.240950
Neighborhood_Edwards	-14272.663320
Condition2_PosN	-23876.331528
RoofMatl_ClyTile	-30652.823854

261 rows × 1 columns

So according to this model:

- As `OverallQual` feature gets bigger the housing price will get bigger.
- `Neighborhood_Edwards` is associated with reducing the housing price.
    - We'll talk more about interpretation of different kinds of features next week.

❓❓ Questions for you

Select all of the following statements which are TRUE.

- (A) Price per square foot would be a good feature to add in our `X`.
- (B) The `alpha` hyperparameter of `Ridge` has similar interpretation as the `C` hyperparameter of `LogisticRegression`; higher `alpha` means more complex model.
- (C) In `Ridge`, smaller alpha means bigger coefficients whereas bigger alpha means smaller coefficients.

▶ Click to view the answers

Regression scoring functions

When we are interested in evaluating how our regression model is doing, we can no longer use the metrics of classification, such as accuracy, precision, recall, etc. Why? All these metrics check for an exact match to one of the categories, and there is no notion of being "close" or "far" away from the correct answer. If we were to check for exact equality of the numerical values of regression, we would get no exact matches:

```
(ridge_tuned.predict(X_train) == y_train).mean()
```

```
0.0
```

```
y_train.values
```

```
array([205000, 160000, 175000, ..., 262500, 133000, 131000])
```

```
ridge_tuned.predict(X_train)
```

```
array([201430.35073678, 172963.41072382, 185110.38205626, ...,
       255111.09578856, 124678.47890063, 137977.01381528])
```

Instead, we need a score that reflects **how** right/wrong each prediction is. These all measure the distance of the prediction from the observed value, with slight variations in how the distance is measured or how they are aggregated together into a single number indicating the model performance.

A few of the most popular scoring functions for regression:

- Mean squared error (MSE)
- Root mean squared error (RMSE)
- Mean absolute error (MAE)
- Mean absolute percentage error (MAPE)
- R^2

See sklearn documentation for more details.

Mean squared error (MSE)

- A common metric is mean squared error, which squares the distance from the prediction to the observed value and reports an average for all the predictions as a final single number metric for how well the model performs:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

```
preds = ridge_tuned.predict(X_train)
```

```
np.mean((y_train - preds) ** 2)
```

```
703801945.0912743
```

Perfect predictions would have MSE=0:

```
np.mean((y_train - y_train) ** 2)
```

```
0.0
```

This score is also implemented in sklearn:

```
from sklearn.metrics import mean_squared_error

mean_squared_error(y_train, preds)
```

```
703801945.0912743
```

- MSE looks huge and unreasonable. There is an error of ~$1 Billion!
- How do we know if this score is good or bad?
- Unlike in classification, with regression **our target has units**.
- The target is in dollars, the mean squared error is in $dollars^2$, also hard to interpret!
- The score also depends on the scale of the targets.
- If we were working in cents instead of dollars, our MSE would be $10,000 \times (100^2)$ higher!

```
np.mean((y_train * 100 - preds * 100) ** 2)
```

```
7038019450912.743
```

Root mean squared error (RMSE)

- To convert the MSE from $dollars^2$ into dollars (and make it more interpretable) we can find it's square root.
- This type of conversion is so common that the metric has its own name: The root mean squared error, or RMSE for short.
- Scoring multiple models with either MSE or RMSE would yield the same ranking of the models, but communicating erros in RMSE are often easier to understand.

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

```
np.sqrt(mean_squared_error(y_train, ridge_tuned.predict(X_train)))
```

```
26529.265822696154
```

```
# RMSE also has a separate function in sklearn
root_mean_squared_error(y_train, preds)
```

```
26529.265822696154
```

- An error of ~$30,000 makes more sense than 800 million square dollars.

Mean absolute error (MAE)

- Instead of squaring the error distances, we could average their absolute value (their magnitude).
- Less sensitive to outliers than MSE (similar to how the median is less sensitive to ouliers than the average).
- Same units as the target so no need to take the square root.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

```
from sklearn.metrics import mean_absolute_error

mean_absolute_error(y_train, preds)
```

```
16362.067003375976
```

R^2

Another common score is the R^2

- This is the score that `sklearn` uses by default when you call score()
- R^2 measures the proportion of variation in y that can be explained using X.

- Another way of thinking of this is that we compare our model's errors to the errors in a baseline model which always predicts the mean.
- Independent of the scale of y. So the max is 1 for perfect description, and it can be negative if it is worse than predicting the mean.
- You will learn about it in more detail in DSCI 561.

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

MAPE

- We got an RMSE of ~$30,000 before.
  - Question: Is an error of $30,000 acceptable?
    - For a house worth $600k, it seems reasonable! That's 5% error.
    - For a house worth $60k, that is terrible. It's 50% error.
- The idea that the severity of the error depends not only on its absolute magnitude, but also on the magnitude relative the actual value, motivates a score that frames the error in terms of a proportion/percentage instead of an absolute number.
- One such error metric is the mean absolute percentage error, or MAPE.
- The equation is similar to that of MAE, but we are also dividing by the oobserved value that we are trying to predict.

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{y_i}$$

```
def my_mape(true, pred):
    return np.mean(np.abs((pred - true)) / true)

my_mape(y_train, preds)
```

```
0.09789380094473478
```

Let's use `sklearn` to calculate MAPE.

```
from sklearn.metrics import mean_absolute_percentage_error

mean_absolute_percentage_error(y_train, preds)
```

```
0.09789380094473478
```

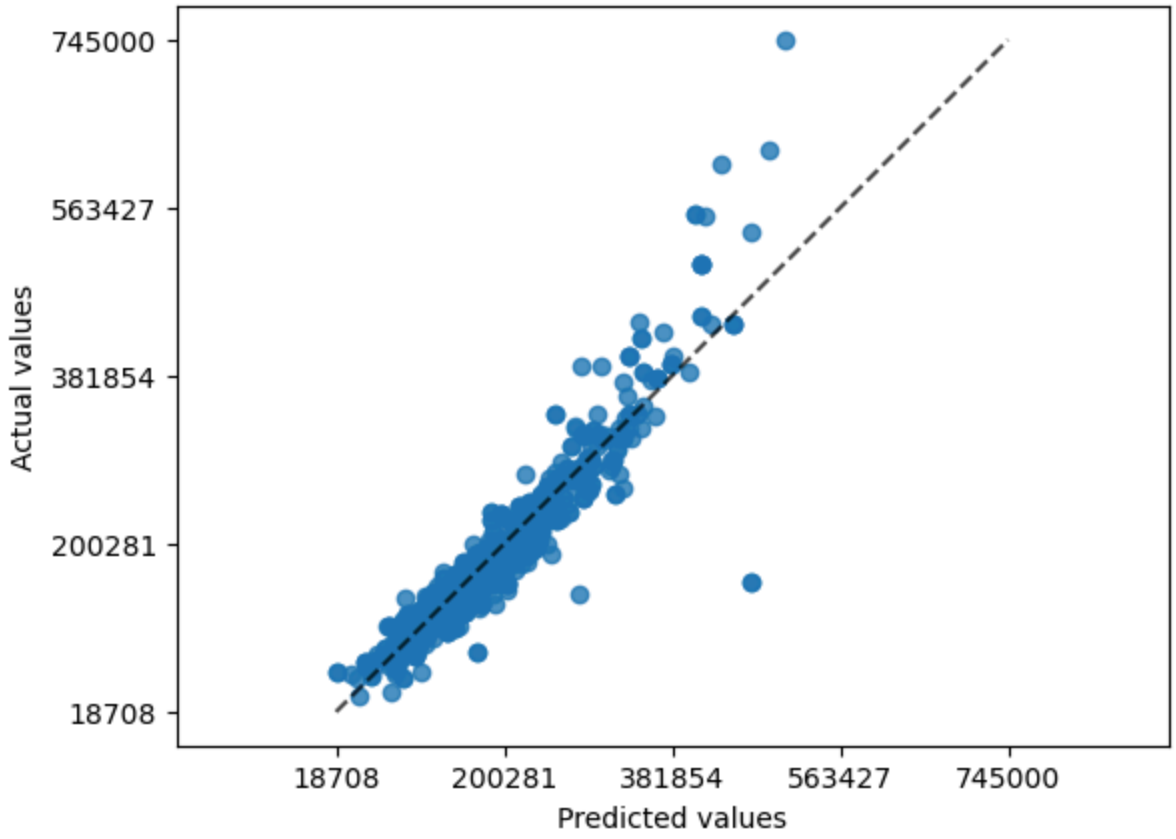
- On average, we have around 10% error.
- That is quite interpretable!

Inspecting the error of the model

- Regardless of which error metric we use, it is always a good idea to inspect what type of errors are happening during the prediciton. Maybe the model is really good at houses with high prices and bad at those with lower prices?
- We can do this in several ways, for example, for a time series it would be interesting to plot the actual values and the predicted ones over time.
    - We could of course extend this and plot the error against any of the variables in our dataset to see if there are specific ranges that are problematic
- Another common way is to plot the predicted values (or the error of the predicted values) versus the actual values. This is a similar idea to what what we did in classification when we compared the actual vs predicted values in a confusion matrix.

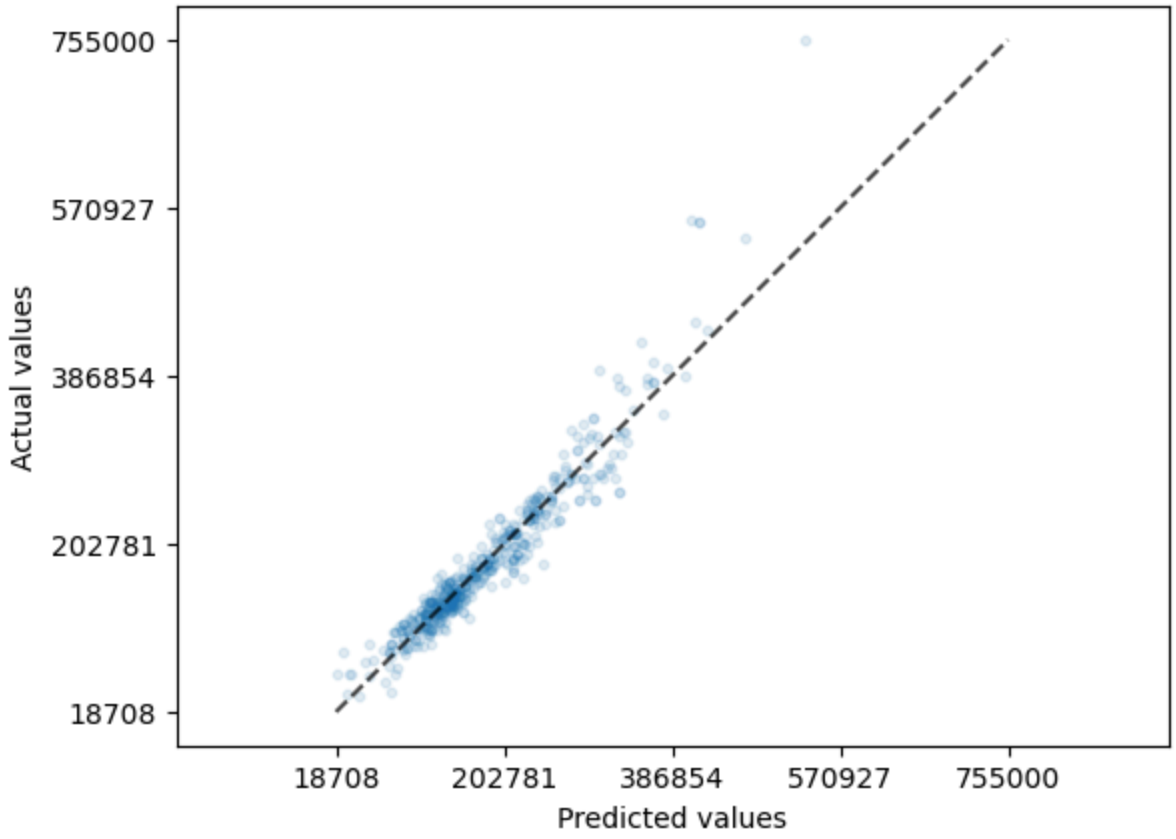
```
from sklearn.metrics import PredictionErrorDisplay

PredictionErrorDisplay.from_estimator(
    ridge_tuned,
    X_train,
    y_train,
    kind='actual_vs_predicted',
)
```

```
<sklearn.metrics._plot.regression.PredictionErrorDisplay at 0x70a4bad1aea0>
```



As you can see above, there is notable saturation/overplotting in the chart. We can't really say anything about the relative counts of points for different regions within the fully opaque area. A 2D histogram/heatmap would be the ideal solution here, but there is no easy way to achieve this with `PredictionErrorDisplay`, so instead we can sample to reduce the number of errors displayed, and adjust the transparency and size of the points.

```
PredictionErrorDisplay.from_estimator(
    ridge_tuned,
    X_train,
    y_train,
    kind='actual_vs_predicted',
    # The default is to show 1000 error points
    subsample=500,
    scatter_kwargs={'alpha': 0.12, 's': 10},
);
```

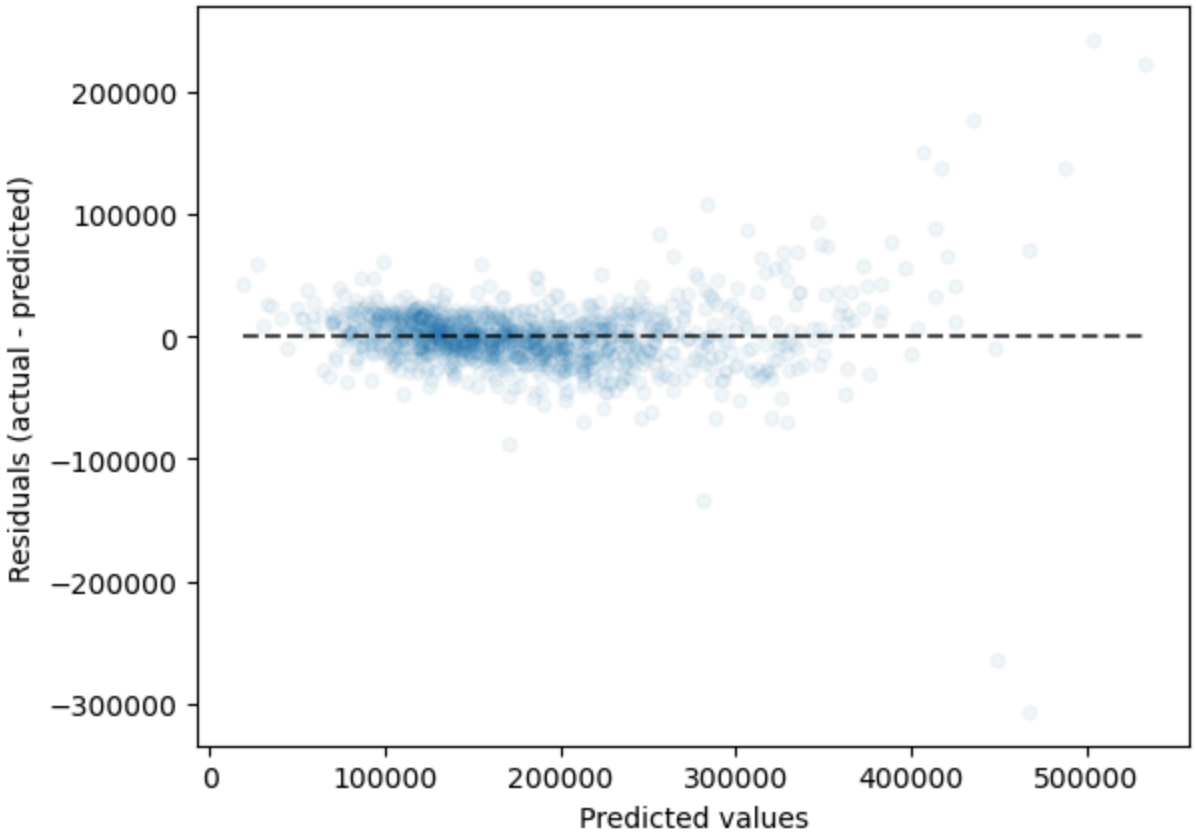


It can be easier to see the pattern of prediction error, if we plot the error vs the actual value instead of the prediction versus the actual value. In this type of plot, the ideal case of zero error is a flat line instead of a diagonal one, which can make it easier to see patterns in the data. You can get an idea of this by observing the two most severe over-predictions (those far under the dotted line). In the chart above, they seem to be closer to the line and the rest of the points than some of the under-predictions (those over the dottedline), although they are actually further away as we can see below.

> Note that in statistics the difference between an observed value and its predicted value is called a residual. The term “error” is reserved for “true” values (e.g. a population mean which can’t be measured since we don’t have access to the whole population). In machine learning, the terms error and residuals are used more interchangeably.

```
PredictionErrorDisplay.from_estimator(
    ridge_tuned,
    X_train,
    y_train,
    scatter_kwargs={'alpha': 0.05, 's': 20},
    subsample=None  # show all predictions
)
```

```
<sklearn.metrics._plot.regression.PredictionErrorDisplay at 0x70a52e74d2e0>
```



- Here we can see a few cases where our prediction is way off.
- Is there something weird about those houses, perhaps? Outliers in terms of the input feautres? Or maybe we just don't have that much data for these types of houses which throws off the predictions.
- Under the line means we're over-predicting, over the line means we're under-predicting.
- Note that here we inspected the errors on the model fit on all the data. For a more realistic assessment of the model performance, see the section below for how to create these charts during crossvalidation.

Different scoring functions with `cross_validate`

So far we have seen how to use different metrics to evaluate the errors in our model, but how can we use them during cross-validation to influence which the best hyperparameters are? For example, how can we use MSE instead of the default R^2 score? To control the scoring behavior during CV, we can use the `scoring` parameter and set it to either a string of an existing metric, or a scoring function that we have made ourselves.

The default in sklearn's CV is to try to make the scoring metric as big as possible, which makes sense for classification metrics such as accuracy, precision, recall, etc (and R2). However, for many our regression metrics, smaller is better. Because of this, the version used for CV needs to be inverted to be negative, which is reflected in the name of these CV scoring methods, such as `neg_mean_squared_error`, and `neg_root_mean_squared_error`.

```
pd.DataFrame(
    cross_validate(
        ridge_tuned,
        X_train,
        y_train,
        return_train_score=True,
        scoring="neg_mean_squared_error",
    )
)
```

	fit_time	score_time	test_score	train_score
0	0.029426	0.011819	-7.425590e+08	-7.149865e+08
1	0.040417	0.012570	-1.248247e+09	-6.342779e+08
2	0.040979	0.012555	-1.065618e+09	-6.913802e+08
3	0.040008	0.012170	-9.952110e+08	-6.816769e+08
4	0.041397	0.012455	-2.215795e+09	-6.063768e+08

To make our own scoring metric, we can use the `make_scorer` wrapper function. This also allows us to specificy if greater or lesser scores indicate better performance.

```
# make a scorer function that we can pass into cross-validation
mape_scorer = make_scorer(my_mape, greater_is_better=False)

pd.DataFrame(
    cross_validate(
        ridge_tuned, X_train, y_train, return_train_score=True, scoring=mape_s
    )
)
```

	fit_time	score_time	test_score	train_score
0	0.042820	0.012384	-0.104175	-0.099270
1	0.041596	0.012867	-0.116258	-0.095073
2	0.042752	0.012727	-0.121141	-0.097903
3	0.041385	0.012605	-0.119933	-0.098088
4	0.041854	0.012387	-0.121586	-0.093236

If you are finding `greater_is_better=False` argument confusing, here is the documentation:

> greater_is_better(bool), default=True Whether score_func is a score function (default), meaning high is good, or a loss function, meaning low is good. In the latter case, the scorer object will sign-flip the outcome of the score_func.

Since our custom scorer `mape` gives an error and not a score, I'm passing `False` to it and it'll sign flip so that we can interpret bigger numbers as better performance. We can also pass multiple metrics together as a dictionary to CV:

```
scoring = {
    "r2": "r2",
    "mape_scorer": mape_scorer, # just for demonstration for a custom scorer
    "sklearn MAPE": "neg_mean_absolute_percentage_error",
    "neg_root_mean_square_error": "neg_root_mean_squared_error",
    "neg_mean_squared_error": "neg_mean_squared_error",
}

pd.DataFrame(
    cross_validate(
        ridge_tuned, X_train, y_train, return_train_score=True, scoring=scorin
    )
).T
```

	0	1	
fit_time	4.275608e-02	4.208136e-02	4.191828e-0
score_time	1.466393e-02	1.478696e-02	1.451612e-0
test_r2	8.600113e-01	8.188275e-01	8.354531e-0
train_r2	8.896149e-01	8.953791e-01	8.879447e-0
test_mape_scorer	-1.041750e-01	-1.162580e-01	-1.211412e-0
train_mape_scorer	-9.927005e-02	-9.507336e-02	-9.790265e-0
test_sklearn MAPE	-1.041750e-01	-1.162580e-01	-1.211412e-0
train_sklearn MAPE	-9.927005e-02	-9.507336e-02	-9.790265e-0
test_neg_root_mean_square_error	-2.724994e+04	-3.533054e+04	-3.264380e+0
train_neg_root_mean_square_error	-2.673923e+04	-2.518487e+04	-2.629411e+0
test_neg_mean_squared_error	-7.425590e+08	-1.248247e+09	-1.065618e+0
train_neg_mean_squared_error	-7.149865e+08	-6.342779e+08	-6.913802e+0

Are we getting the same `alpha` with mape?

```
param_grid = {"ridge__alpha": 10.0 ** np.arange(-6, 6, 1)}

pipe_ridge = make_pipeline(preprocessor, Ridge())

search = GridSearchCV(
    pipe_ridge,
    param_grid,
    return_train_score=True,
    n_jobs=-1,
    scoring=mape_scorer
)
search.fit(X_train, y_train);
print("Best hyperparameter values: ", search.best_params_)
print("Best score: %0.3f" % (search.best_score_))
```

```
Best hyperparameter values:  {'ridge__alpha': 100.0}
Best score: -0.111
```

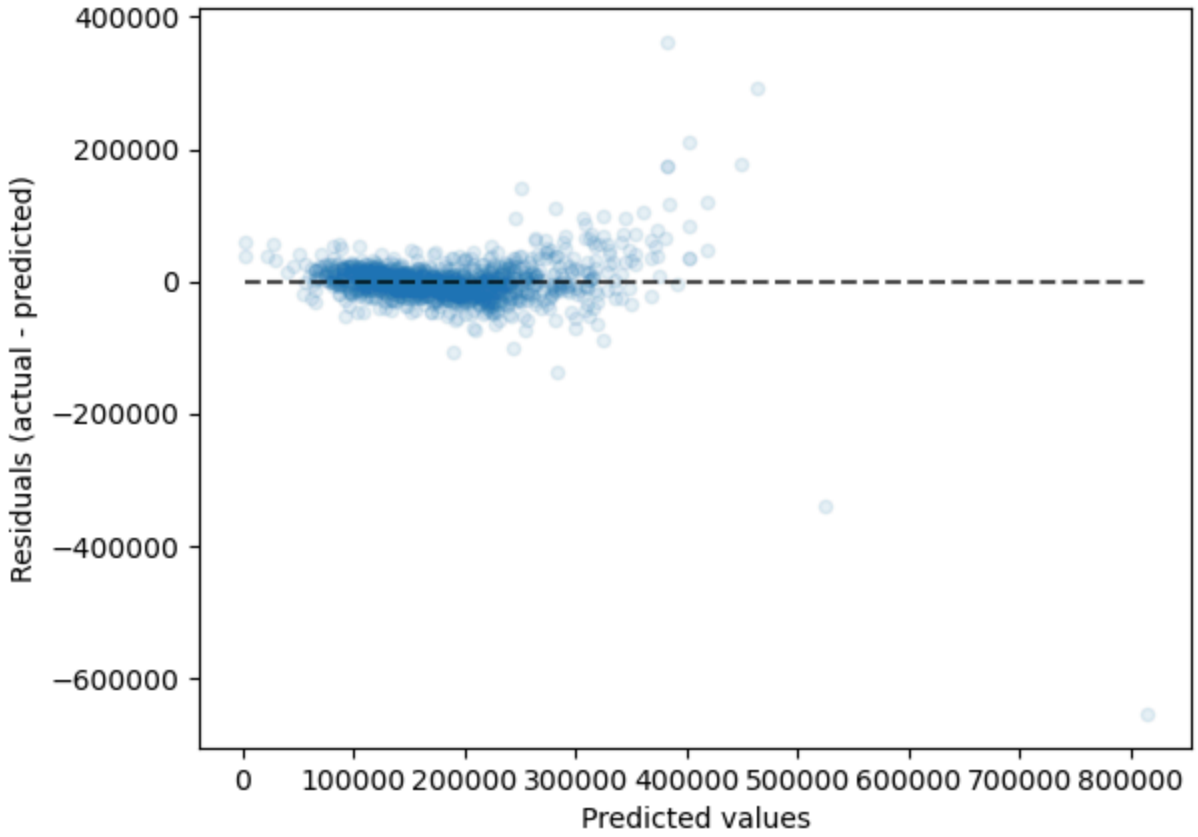
```
pd.DataFrame(search.cv_results_)[[
    "mean_train_score",
    "mean_test_score",
    "param_ridge__alpha",
    "mean_fit_time",
    "rank_test_score",
]].sort_values("rank_test_score")
```

	mean_train_score	mean_test_score	param_ridge__alpha	mean_fit_time	ra
8	-0.101282	-0.110627	100.000000	0.073630	
9	-0.111543	-0.115387	1000.000000	0.062149	
7	-0.096714	-0.116619	10.000000	0.071083	
6	-0.091244	-0.122433	1.000000	0.085649	
5	-0.086182	-0.126190	0.100000	0.075532	
4	-0.085975	-0.128442	0.010000	0.083047	
3	-0.086015	-0.128815	0.001000	0.073010	
2	-0.086020	-0.128855	0.000100	0.081070	
1	-0.086020	-0.128859	0.000010	0.066091	
0	-0.086020	-0.128859	0.000001	0.057801	
10	-0.203421	-0.204692	10000.000000	0.060869	
11	-0.332130	-0.332710	100000.000000	0.062444	

Like we did in classification, we can still use our diagnostic plots with cross-validation:

```
from sklearn.model_selection import cross_val_predict

PredictionErrorDisplay.from_predictions(
    y_train,
    cross_val_predict(search, X_train, y_train),
    # Here we show all the errors and set the transparency a bit lower
    # to draw attention to the large errors, since we discuss them in the
    # paragraph below (this causes a saturation in the rest of the chart).
    subsample=None,
    scatter_kwargs={'alpha': 0.1, 's': 20},
)
```

```
<sklearn.metrics._plot.regression.PredictionErrorDisplay at 0x70a52e86b020>
```



Woa! The error is even bigger now. Why is that?

▶ Click to see the answer

Using multiple metrics in `GridSearchCV` or `RandomizedSearchCV`

- We could use multiple metrics with `GridSearchCV` or `RandomizedSearchCV`.
- But if you do so, you need to set `refit` to the metric (string) for which the `best_params_` will be found and used to build the `best_estimator_` on the whole dataset (remember that the grid search refits the best model on the whole dataset at the end).

```
search_multi = GridSearchCV(
    pipe_ridge,
    param_grid,
    return_train_score=True,
    n_jobs=-1,
    scoring=scoring,
    refit="mape_scorer",
)
search_multi.fit(X_train, y_train);
```

```
print("Best hyperparameter values: ", search_multi.best_params_)
print("Best score: %0.3f" % (search_multi.best_score_))
```

```
Best hyperparameter values:  {'ridge__alpha': 100.0}
Best score: -0.111
```

Although the model was picked based on one metric, we now have easy access to review how the top models performed across all the metrics we specified.

```
pd.DataFrame(
    search_multi.cv_results_
).filter(
    regex='param|mean_test|rank_test'  # Filter the column to display
).sort_values(
    'rank_test_mape_scorer'
)
```

	param_ridge__alpha	params	mean_test_r2	rank_test_r2	n
8	100.000000	{'ridge__alpha': 100.0}	0.796141	2	
9	1000.000000	{'ridge__alpha': 1000.0}	0.780511	4	
7	10.000000	{'ridge__alpha': 10.0}	0.796689	1	
6	1.000000	{'ridge__alpha': 1.0}	0.786662	3	
5	0.100000	{'ridge__alpha': 0.09999999999999999}	0.773797	5	
4	0.010000	{'ridge__alpha': 0.01}	0.765168	6	
3	0.001000	{'ridge__alpha': 0.001}	0.763811	7	
2	0.000100	{'ridge__alpha': 9.999999999999999e-05}	0.763666	8	
1	0.000010	{'ridge__alpha': 9.999999999999999e-06}	0.763652	9	
0	0.000001	{'ridge__alpha': 1e-06}	0.763650	10	
10	10000.000000	{'ridge__alpha': 10000.0}	0.569424	11	
11	100000.000000	{'ridge__alpha': 100000.0}	0.125775	12	

We can see that the same model (choice of hyperparameters) was ranked first across all metrics.

> (Optional) Note that when we fit a linear regression model scikit-learn needs to figure out which line is the best fit to the data (even without optimizing hyperparameters). For a Ridge regression, this is done by using so called ordinary least squares (OLS) regression, which means that the line that is considered "the best fit", is the one that minimzed the sum of squared errors/residuals to the data. So although we can choose which scoring metric to use to find the best hyperparameters in our grid search, the fit step will always include this squared distance notion of what the best line is. OLS is not the only way to find the best line, and there are many other types ot linear regressors that all find optimal lines in different ways (e.g. minimizing the absolute distances).

What's the score on the test dataset?

```
search_multi.score(X_test, y_test)
```

```
-0.09492325020968066
```

```
mean_absolute_percentage_error(y_test, ridge_tuned.predict(X_test))
```

```
0.0956641017539453
```

When reporting the error of our model, we should ideally show multiple metrics to make it easier to interpret.

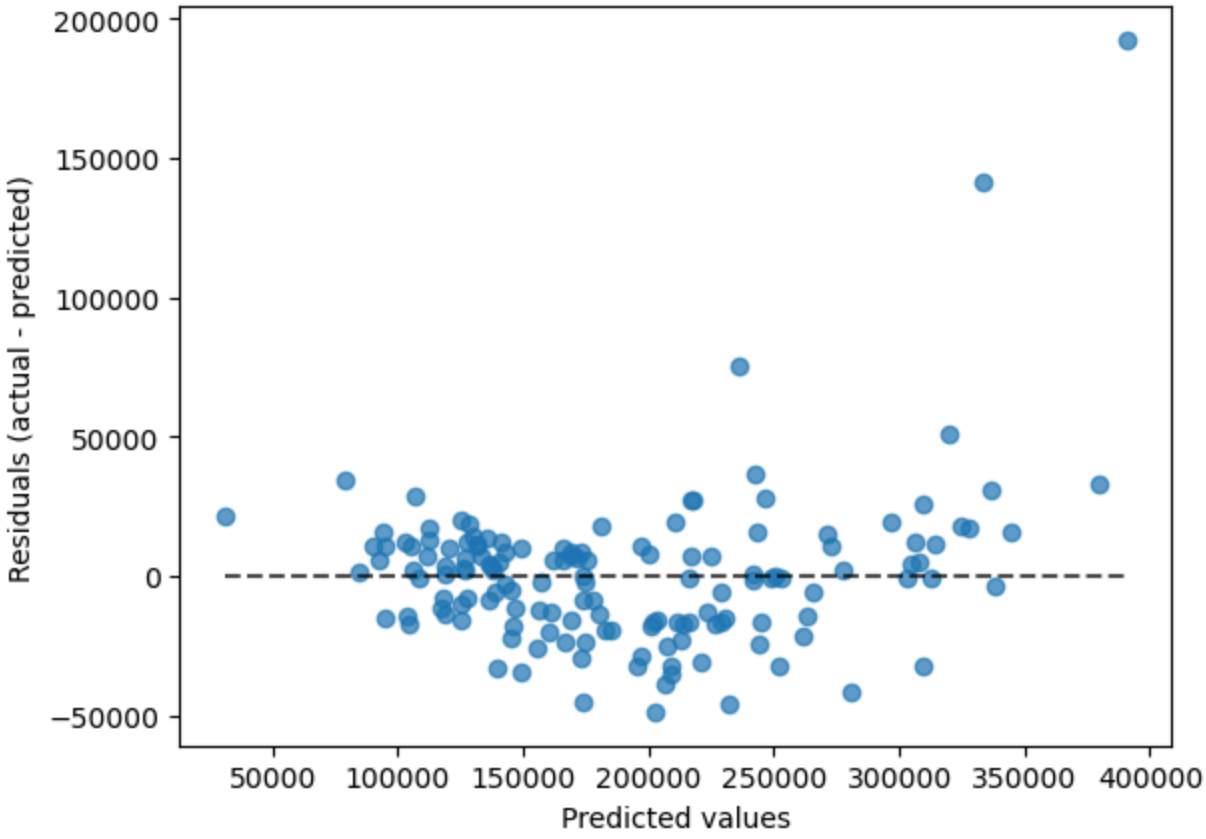
```
pd.DataFrame({
    'MAPE': [mean_absolute_percentage_error(y_test, ridge_tuned.predict(X_test
    'RMSE': [root_mean_squared_error(y_test, ridge_tuned.predict(X_test))],
    'R2': [r2_score(y_test, ridge_tuned.predict(X_test))],
})
```

	MAPE	RMSE	R2
0	0.095664	26559.127006	0.895811

It is also a good idea to communicate how the model prediction look compare to the predicted values, using the same charts as before, but now on the predictions from the test data.

```
PredictionErrorDisplay.from_predictions(
    y_test,
    search_multi.predict(X_test),
    subsample=None,  # show all predictions
    scatter_kwargs={'alpha': 0.7}
)
```

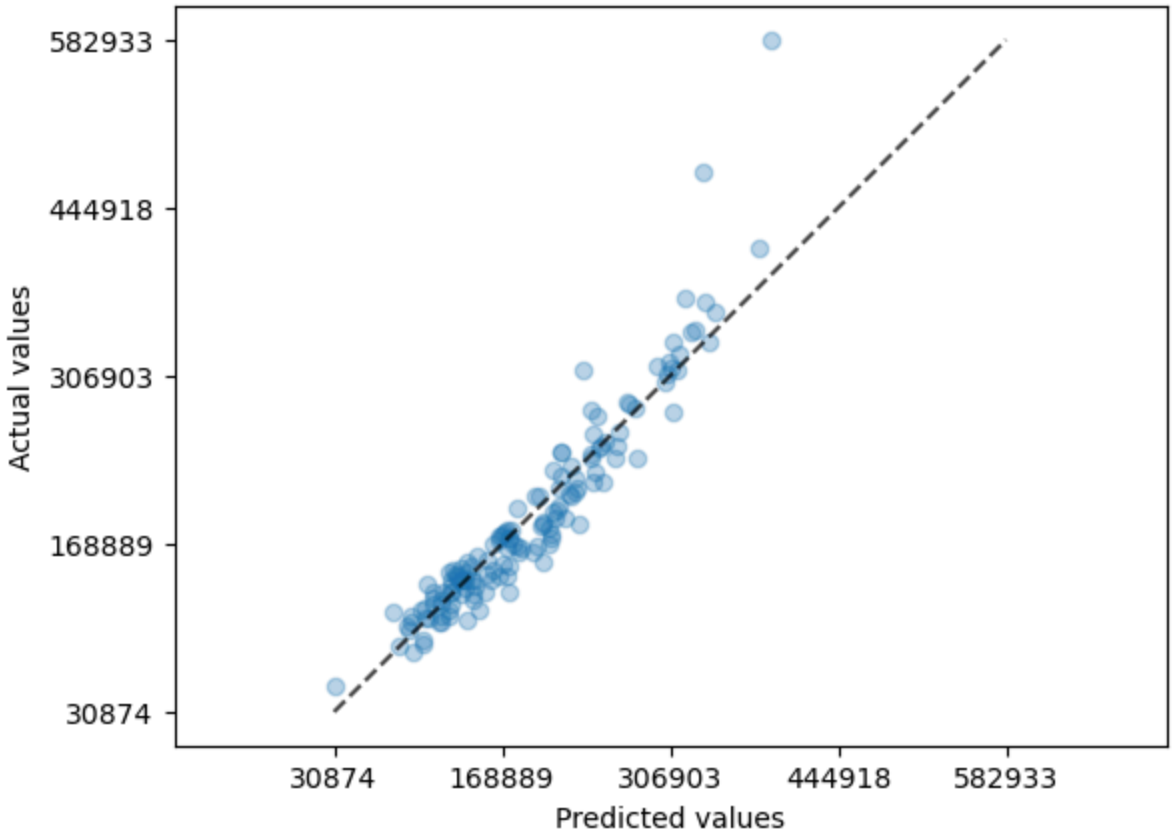
```
<sklearn.metrics._plot.regression.PredictionErrorDisplay at 0x70a4bb0aac60>
```



Depending of the target audience for the communication, it can be helpful to show the actual vs predicted chart, since this is easier to understand than looking at the errors/predictions (but note the caveat mentioned previously).

```
PredictionErrorDisplay.from_predictions(
    y_test,
    search_multi.predict(X_test),
    kind='actual_vs_predicted',
    subsample=None,  # show all predictions
    scatter_kwargs={'alpha': 0.3}
)
```

```
<sklearn.metrics._plot.regression.PredictionErrorDisplay at 0x70a4ba872270>
```



Note

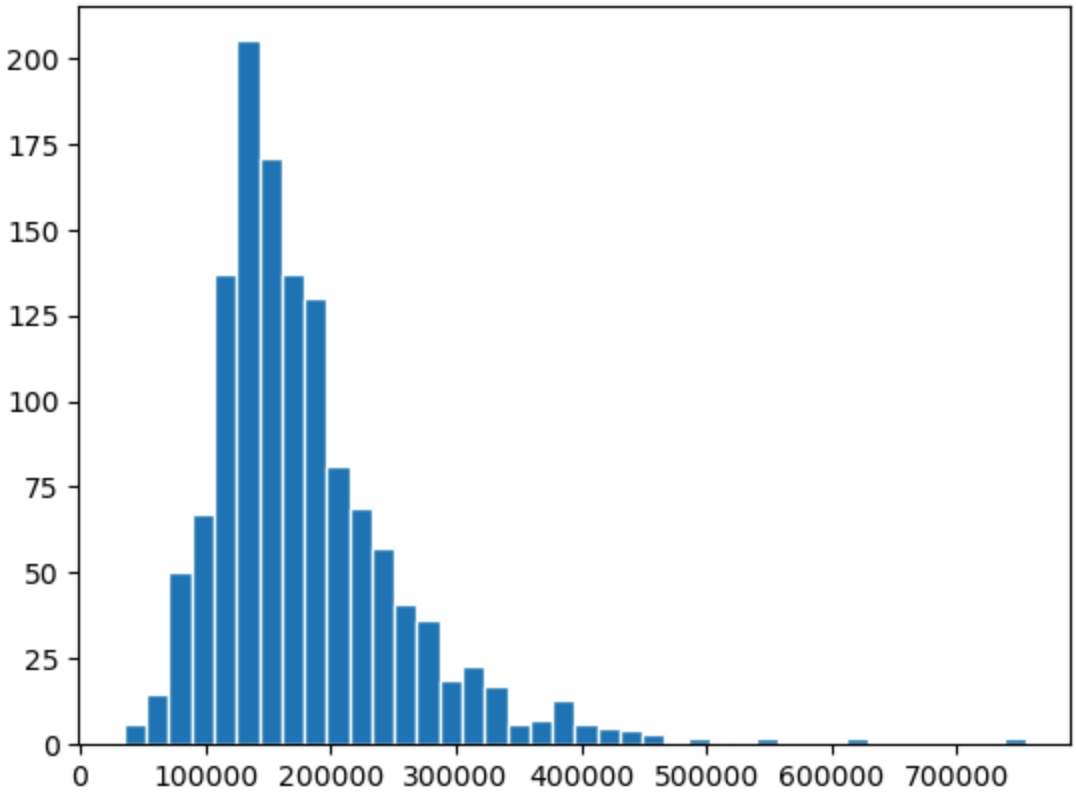
Note that we are not allowed to make any modifications to our model, based on the scores or visualizations of the test data errors! That would break the golden rule and lead to an overly optimistic model since we are using info from the the test data to improve the model. Any information in the test data must be completely ignored for it to be an valid approximation of the model's generalization performance on unseen data.

Transforming the targets

We have talked about transforming features, but what about transforming targets?

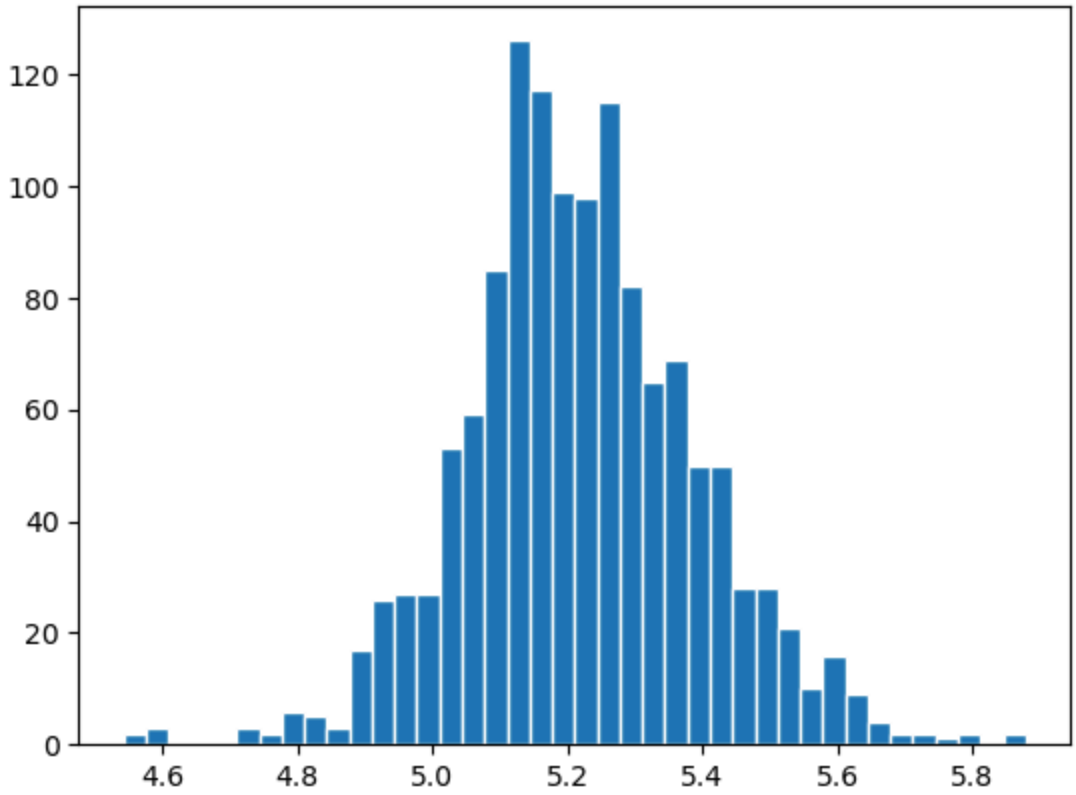
- When you have prices or count data, the target values are often skewed.
- Let's look at our target column.

```
plt.hist(y_train, bins=40, edgecolor='white');
```



- A common trick in such cases is applying a log transform on the target column to make it more normal and less skewed.
- That is, transform $y \rightarrow \log(y)$.
- Linear regression will usually have better performance on data that is more normal.

```
plt.hist(np.log10(y_train), bins=40, edgecolor='white');
```



We can incorporate this in our pipeline using `sklearn` using a TransformedTargetRegressor.

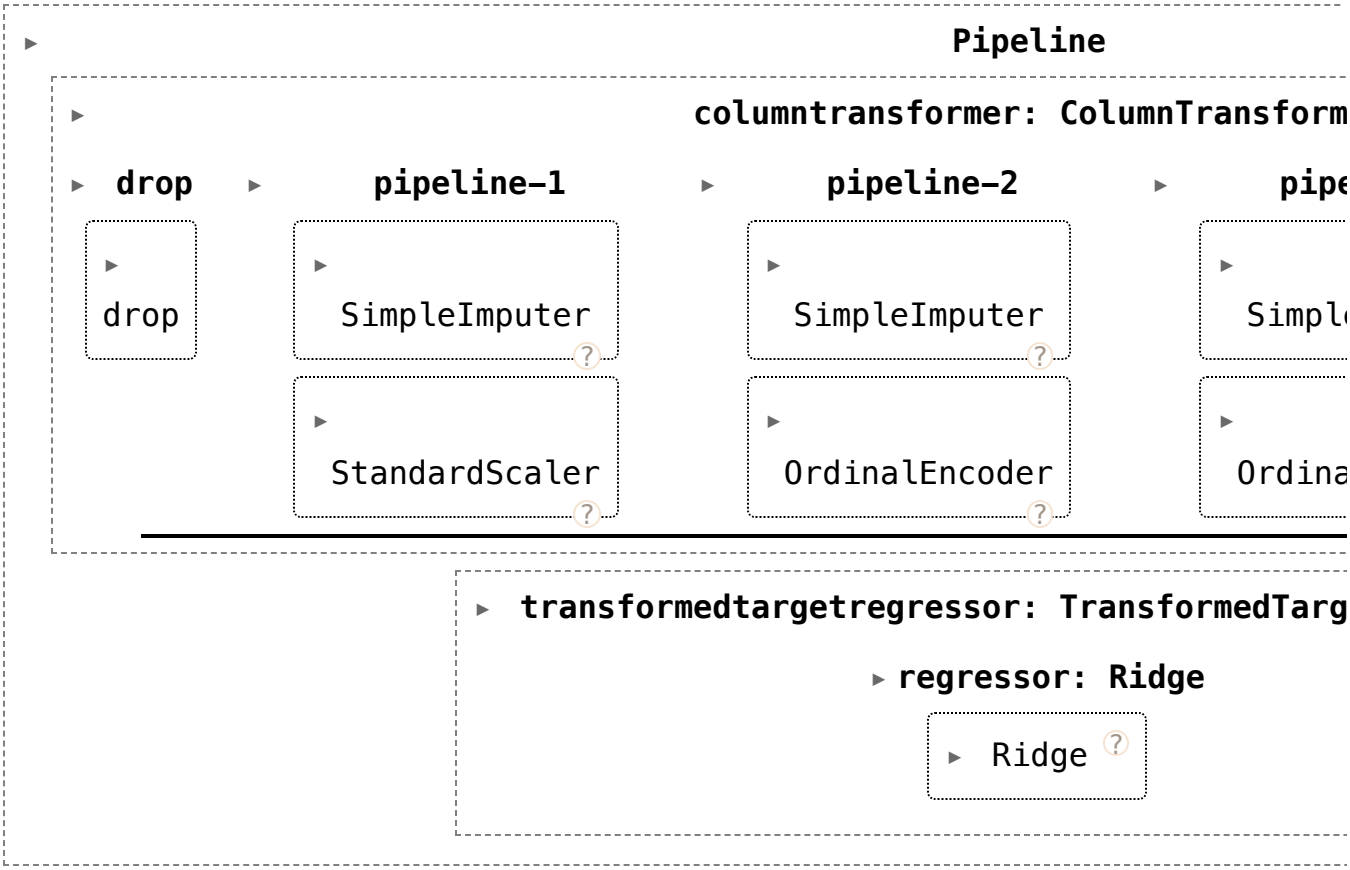
Why don't we incorporate preprocessing targets in our column transformer instead of this separate TransformedTargetRegressor?

Because we want our predictions to be in the original scale of the housing prices rather than their log scale. So when we fit with the TransformedTargetRegressor, the y values will be automatically log transformed, but when we predict, the model knows to predict in the original scale instead of returning the log transformed values:

```
from sklearn.compose import TransformedTargetRegressor
```

```
ttr = TransformedTargetRegressor(
    Ridge(alpha=best_alpha),
    func=np.log1p,  # The transformation to use when fitting (to get the origi
    inverse_func=np.expm1  # The transformation to use when predicting (to get
)
ttr_pipe = make_pipeline(preprocessor, ttr)
```

```
ttr_pipe
```



Now we can fit and predict using this transformed regresion model.

```
ttr_pipe.fit(X_train, y_train); # y_train automatically transformed
```

```
ttr_pipe.predict(X_train)  # predictions automatically un-transformed
```

```
array([212336.52875482, 165605.29337552, 181642.35750539, ...,
       265632.62249693, 131235.40231658, 135954.43468708])
```

```
mean_absolute_percentage_error(y_test, ttr_pipe.predict(X_test))
```

```
0.07740212368464579
```

We reduced MAPE from ~10% to ~8% with this trick! It seems like it had a bigger effect than tuning the hyperparameters for this particular problem.

Brier score - MSE for classification

We could extend the notion of how wrong an individual prediction is to also apply to classification. Specifically, classifiers that output a probability/confidence, can be scored on how far that probability is from the actual outcome in a similar way that a regression prediction can be scored on how far it is from the observed value. Note that the observed value in a classification is always 0 or 1, but the predicted probability can be anything between 0 and 1. The main notion here is that we think it is a bigger mistake if the model makes an error when it is really certain on its prediction versus when it says it is says it doesn't really know and gives maybe just a 0.51/0.49 soft prediction. Drawing a hard classification boundary treats all incorrect predictions equally, no matter how certain the model is.

This is a slighlty different approach to classification scoring, and it has the advantage that you can now also ommunicate the outputted probabilities, and give an inclination of how certain the model is, which can then be used as one piece of infomration by a domain expert to decide the next action, rather than giving a certain yes/no answer. If a hard prediction is needed, you can still use a PR or ROC curve to decide on a threshold for hard predictions, after you have used a probabilistic socring method to find the model with the mode accurate probabilistic predictions.

One of the most common scoring metrics for probabilistic loss is Brier score, which is the mean squared error of the predictions versus the observed value. Another commonly used one is log-loss, which penalizes extreme predictions more if they are wrong. You can read more about Brier score in the documentation and see the scikit-learn implementation together with an example.

$$BS = \frac{1}{n}\sum_{i=1}^{n}(y_i - p_i)^2$$

As you can see, the formula for BS is the same as for MSE, but we use y for the actual class label (0 or 1) and p denotes the probability of $y = 1$ (ie `predict_proba`). Note that you often see Brier score written with f (forecasted) instead of p and o (observed) instead of y.

❓❓ Questions for you

Select all of the following statements which are TRUE.

- (A) We can use still use precision and recall for regression problems but now we have other metrics we can use as well.
- (B) In `sklearn` for regression problems, using `r2_score()` and `.score()` (with default values) will produce the same results.
- (C) RMSE is always going to be non-negative.
- (D) MSE does not directly provide the information about whether the model is underpredicting or overpredicting.
- (E) We can pass multiple scoring metrics to `GridSearchCV` or `RandomizedSearchCV` for regression as well as classification problems.

▶ Click to view the answers

What did we learn today?

- House prices dataset target is price, which is numeric -> regression rather than classification
- There are corresponding versions of all the tools we used:
  - `DummyClassifier` -> `DummyRegressor`
  - `LogisticRegression` -> `Ridge`
- `Ridge` hyperparameter `alpha` is like `LogisticRegression` hyperparameter `C`, but opposite meaning
- We'll avoid `LinearRegression` in this course in favor of using `Ridge`.
- Scoring metrics
- R^2 is the default .score(), it is unitless, 0 is bad, 1 is best
- MSE (mean squared error) is in units of target squared, hard to interpret; 0 is best
- RMSE (root mean squared error) is in the same units as the target; 0 is best

- MAE (mean absolute error) is in the same units as the target; 0 is best
- MAPE (mean absolute percent error) is unitless; 0 is best, 1 is bad