10_regression-metrics

June 23, 2022

CPSC 330
Applied Machine Learning

1 Lecture 10: Regression Evaluation Metrics

UBC 2022 Summer

Instructor: Mehrdad Oveisi

1.1 Imports

[1]:
```
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 LinearRegression, Ridge, RidgeCV
from sklearn.metrics import make_scorer, mean_squared_error, r2_score
from sklearn.model_selection import cross_val_score, cross_validate,␣
 ↪train_test_split
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler
from sklearn.tree import DecisionTreeRegressor

%matplotlib inline
```

[2]:
```
import warnings
```

```
warnings.simplefilter(action="ignore", category=FutureWarning)
```

1.2 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.
- 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, MAPE
- Apply log-transform on the target values in a regression problem with `TransformedTargetRegressor`.

1.3 Dataset

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.

[3]:
```
df = pd.read_csv("data/housing-kaggle/train.csv")
train_df, test_df = train_test_split(df, test_size=0.10, random_state=123)
train_df.head()
```

[3]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	\
302	303	20	RL	118.0	13704	Pave	NaN	IR1	
767	768	50	RL	75.0	12508	Pave	NaN	IR1	
429	430	20	RL	130.0	11457	Pave	NaN	IR1	
1139	1140	30	RL	98.0	8731	Pave	NaN	IR1	
558	559	60	RL	57.0	21872	Pave	NaN	IR2	

	LandContour	Utilities	…	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	\
302	Lvl	AllPub	…	0	NaN	NaN	NaN	0	
767	Lvl	AllPub	…	0	NaN	NaN	Shed	1300	
429	Lvl	AllPub	…	0	NaN	NaN	NaN	0	
1139	Lvl	AllPub	…	0	NaN	NaN	NaN	0	
558	HLS	AllPub	…	0	NaN	NaN	NaN	0	

	MoSold	YrSold	SaleType	SaleCondition	SalePrice
302	1	2006	WD	Normal	205000
767	7	2008	WD	Normal	160000
429	3	2009	WD	Normal	175000
1139	5	2007	WD	Normal	144000
558	8	2008	WD	Normal	175000

```
[5 rows x 81 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).

[4]:
```
train_df.shape
```

[4]:
```
(1314, 81)
```

1.3.1 Let's separate `X` and `y`

[5]:
```
X_train = train_df.drop(columns=["SalePrice"])
y_train = train_df["SalePrice"]

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

1.3.2 Exploratory Data Analysis (EDA)

[6]:
```
train_df.describe()
```

[6]:
```
                Id   MSSubClass  LotFrontage        LotArea  OverallQual  \
count  1314.000000  1314.000000  1089.000000    1314.000000  1314.000000
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

       OverallCond    YearBuilt  YearRemodAdd   MasVnrArea   BsmtFinSF1  …  \
count  1314.000000  1314.000000   1314.000000  1307.000000  1314.000000  …
mean      5.570015  1970.995434   1984.659056   102.514155   441.425419  …
std       1.112848    30.198127     20.639754   178.301563   459.276687  …
min       1.000000  1872.000000   1950.000000     0.000000     0.000000  …
25%       5.000000  1953.000000   1966.250000     0.000000     0.000000  …
50%       5.000000  1972.000000   1993.000000     0.000000   376.000000  …
75%       6.000000  2000.000000   2004.000000   165.500000   704.750000  …
max       9.000000  2010.000000   2010.000000  1378.000000  5644.000000  …

        WoodDeckSF  OpenPorchSF  EnclosedPorch    3SsnPorch  ScreenPorch  \
count  1314.000000  1314.000000    1314.000000  1314.000000  1314.000000
mean     94.281583    45.765601      21.726788     3.624049    13.987062
std     125.436492    65.757545      60.766423    30.320430    53.854129
```

```
min       0.000000     0.000000       0.000000     0.000000     0.000000
25%       0.000000     0.000000       0.000000     0.000000     0.000000
50%       0.000000    24.000000       0.000000     0.000000     0.000000
75%     168.000000    66.750000       0.000000     0.000000     0.000000
max     857.000000   547.000000     552.000000   508.000000   480.000000

          PoolArea       MiscVal       MoSold       YrSold      SalePrice
count  1314.000000   1314.000000  1314.000000  1314.000000    1314.000000
mean      3.065449     46.951294     6.302131  2007.840183  179802.147641
std      42.341109    522.283421     2.698206     1.332824   79041.260572
min       0.000000      0.000000     1.000000  2006.000000   34900.000000
25%       0.000000      0.000000     5.000000  2007.000000  129600.000000
50%       0.000000      0.000000     6.000000  2008.000000  162000.000000
75%       0.000000      0.000000     8.000000  2009.000000  212975.000000
max     738.000000  15500.000000    12.000000  2010.000000  755000.000000

[8 rows x 38 columns]
```

[7]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1314 entries, 302 to 1389
Data columns (total 81 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     1307 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  PoolQC         7 non-null      object
 73  Fence          259 non-null    object
 74  MiscFeature    50 non-null     object
 75  MiscVal        1314 non-null   int64
 76  MoSold         1314 non-null   int64
 77  YrSold         1314 non-null   int64
 78  SaleType       1314 non-null   object
 79  SaleCondition  1314 non-null   object
 80  SalePrice      1314 non-null   int64
dtypes: float64(3), int64(35), object(43)
memory usage: 841.8+ KB
```

1.3.3 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 -n cpsc330 -c conda-forge pandas-profiling`

[8]:
```
# from pandas_profiling import ProfileReport

# profile = ProfileReport(train_df, title="Pandas Profiling Report")  # ,␣
  ↪minimal=True)
# profile.to_notebook_iframe()
```

1.3.4 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.

1.3.5 Feature types

- We have mixed feature types and a bunch of missing values.
- Now, let's identify feature types and transformations.
- Let's get the numeric-looking columns.

[9]:
```
numeric_looking_columns = X_train.select_dtypes(include=np.number).columns.
  ↪tolist()
print(numeric_looking_columns)
```

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

Not all numeric looking columns are necessarily numeric.

[10]:
```
train_df["MSSubClass"].unique()
```

[10]:
```
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.

[11]:
```
train_df["MoSold"].unique() # Month Sold
```

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

[12]:
```
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.

[13]:
```
set(numeric_looking_columns) - set(numeric_features) - set(drop_features)
```

[13]:
```
{'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`.

[14]:
```
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
```

[14]:
```
[['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'],
 ['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.

[15]:
```
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.

[16]:
```
categorical_features = list(
    set(X_train.columns)
    - set(numeric_features)
    - set(ordinal_features_reg)
    - set(ordinal_features_oth)
    - set(drop_features)
)
categorical_features
```

[16]:
```
['SaleCondition',
 'Alley',
 'Condition2',
 'Street',
 'BldgType',
 'GarageType',
 'Electrical',
 'Exterior2nd',
 'GarageFinish',
 'LotConfig',
 'PavedDrive',
 'MSSubClass',
 'Neighborhood',
 'RoofMatl',
 'MiscFeature',
 'CentralAir',
 'MSZoning',
 'HouseStyle',
 'Exterior1st',
 'RoofStyle',
 'Foundation',
 'MoSold',
 'LotShape',
 'SaleType',
 'LandContour',
 'Utilities',
 'Heating',
 'LandSlope',
 'Condition1',
 'MasVnrType']
```

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

1.3.6 Applying feature transformations

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

[17]:
```
from sklearn.compose import ColumnTransformer, make_column_transformer

numeric_transformer = make_pipeline(SimpleImputer(strategy="median"),␣
 ↪StandardScaler())

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

ordinal_transformer_oth = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OrdinalEncoder(categories=ordering_ordinal_oth),
)

categorical_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(handle_unknown="ignore", sparse=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),
)
```

1.3.7 Examining the preprocessed data

[18]:
```
preprocessor.fit(X_train) # Calling fit to examine all the transformers.
preprocessor.named_transformers_
```

[18]:
```
{'drop': 'drop',
 'pipeline-1': Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='median')),
                ('standardscaler', StandardScaler())]),
 'pipeline-2': Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='most_frequent')),
                ('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'],
                                                ['Po', 'Fa', 'TA', 'Gd',
'Ex']]))]),
 'pipeline-3': Pipeline(steps=[('simpleimputer',
SimpleImputer(strategy='most_frequent')),
                ('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']]))]),
 'pipeline-4': Pipeline(steps=[('simpleimputer',
                 SimpleImputer(fill_value='missing', strategy='constant')),
                ('onehotencoder',
                 OneHotEncoder(handle_unknown='ignore', sparse=False))])}
```

[19]:
```
ohe_columns = list(
    preprocessor.named_transformers_["pipeline-4"]
    .named_steps["onehotencoder"]
    .get_feature_names(categorical_features)
)
new_columns = numeric_features + ordinal_features_reg + ordinal_features_oth +␣
 ↪ohe_columns
```

[20]:
```
X_train_enc = pd.DataFrame(
    preprocessor.transform(X_train), index=X_train.index, columns=new_columns
)
X_train_enc.head()
```

[20]:

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

	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	…	\
302	-0.512408	0.993969	0.840492	0.269972	-0.961498	…	
767	1.285467	-1.026793	0.016525	-0.573129	0.476092	…	
429	-0.512408	0.563314	0.161931	-0.573129	1.227559	…	
1139	-0.512408	-1.689338	-1.679877	-0.573129	0.443419	…	
558	-0.512408	0.828332	0.598149	-0.573129	0.354114	…	

```
      Condition1_PosN  Condition1_RRAe  Condition1_RRAn  Condition1_RRNe  \
302               0.0              0.0              0.0              0.0
767               0.0              0.0              0.0              0.0
429               0.0              0.0              0.0              0.0
1139              0.0              0.0              0.0              0.0
558               0.0              0.0              0.0              0.0

      Condition1_RRNn  MasVnrType_BrkCmn  MasVnrType_BrkFace  MasVnrType_None  \
302               0.0                0.0                 1.0              0.0
767               0.0                0.0                 0.0              1.0
429               0.0                0.0                 0.0              1.0
1139              0.0                0.0                 0.0              1.0
558               0.0                0.0                 0.0              1.0

      MasVnrType_Stone  MasVnrType_missing
302                0.0                 0.0
767                0.0                 0.0
429                0.0                 0.0
1139               0.0                 0.0
558                0.0                 0.0

[5 rows x 263 columns]
```

[21]:
```
X_train.shape
```

[21]:
```
(1314, 80)
```

[22]:
```
X_train_enc.shape
```

[22]:
```
(1314, 263)
```

We went from 80 features to 263 features!!

1.3.8 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 …

1.4 Model building

1.4.1 DummyRegressor

[23]:
```
dummy = DummyRegressor()
pd.DataFrame(cross_validate(dummy, X_train, y_train, cv=10,␣
 ↪return_train_score=True))
```

[23]:

	fit_time	score_time	test_score	train_score
0	0.002102	0.000795	-0.003547	0.0
1	0.001311	0.000332	-0.001266	0.0
2	0.001037	0.000849	-0.011767	0.0
3	0.001410	0.000469	-0.006744	0.0
4	0.000971	0.000274	-0.076533	0.0
5	0.000752	0.000273	-0.003133	0.0
6	0.000883	0.000303	-0.000397	0.0
7	0.001487	0.000434	-0.003785	0.0
8	0.001281	0.000291	-0.001740	0.0
9	0.003163	0.000802	-0.000117	0.0

1.4.2 Apply Ridge

- Recall that we are going to use `Ridge()` instead of `LinearRegression()` in this course.
  - It has a hyperparameter `alpha` which controls the fundamental tradeoff.

[24]:
```
lr_pipe = make_pipeline(preprocessor, Ridge())
pd.DataFrame(cross_validate(lr_pipe, X_train, y_train, cv=10,␣
 ↪return_train_score=True))
```

[24]:

	fit_time	score_time	test_score	train_score
0	0.082690	0.019475	0.861355	0.911906
1	0.066603	0.025994	0.812301	0.913861
2	0.068434	0.020164	0.775283	0.915963
3	0.043042	0.009863	0.874519	0.910849
4	0.040309	0.015504	0.851969	0.911622
5	0.040240	0.010918	0.826198	0.910176
6	0.055487	0.014583	0.825533	0.913781
7	0.044739	0.014253	0.872238	0.910071
8	0.034239	0.015367	0.196663	0.921448
9	0.043370	0.012902	0.890474	0.908221

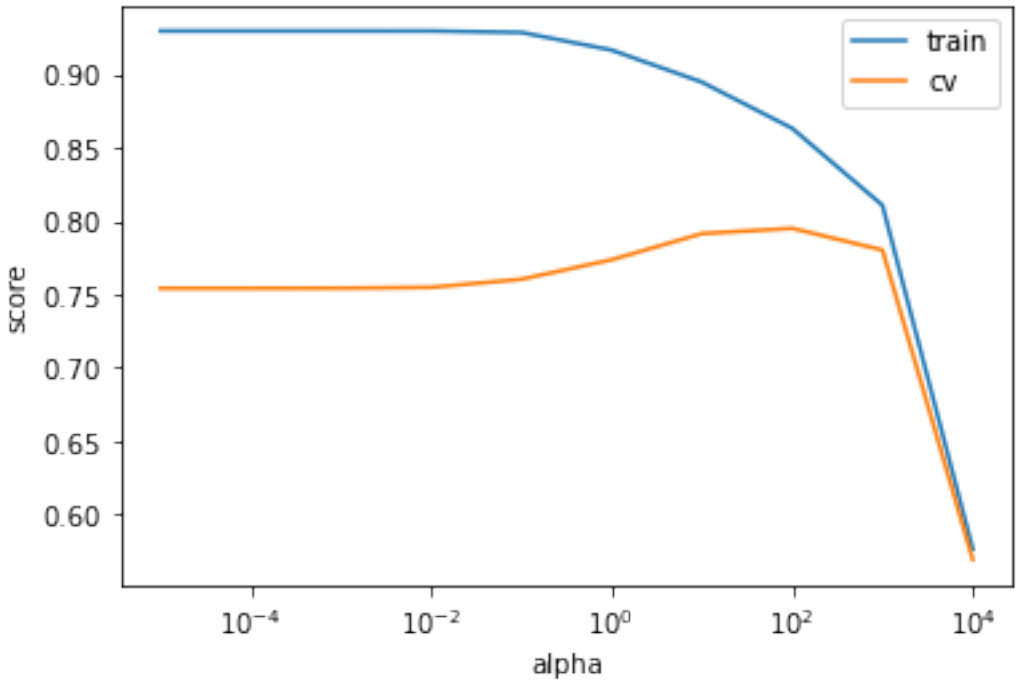
- Quite a bit of variation in the test scores.
- Performing poorly in fold 8. Not sure why.

1.4.3 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)

[25]:
```
alphas = 10.0 ** np.arange(-5, 5, 1)
train_scores = []
cv_scores = []
for alpha in alphas:
    lr = make_pipeline(preprocessor, Ridge(alpha=alpha))
    results = cross_validate(lr, X_train, y_train, return_train_score=True)
    train_scores.append(np.mean(results["train_score"]))
    cv_scores.append(np.mean(results["test_score"]))
```

[26]:
```
plt.semilogx(alphas, train_scores, label="train")
plt.semilogx(alphas, cv_scores, label="cv")
plt.legend()
plt.xlabel("alpha")
plt.ylabel("score");
```



[27]:
```
best_alpha = alphas[np.argmax(cv_scores)]
best_alpha
```

[27]:
```
100.0
```

- 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 (seeing big dependencies when you shouldn't).

1.4.4 RidgeCV

BTW, 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.

[28]:
```
ridgecv_pipe = make_pipeline(preprocessor, RidgeCV(alphas=alphas, cv=10))
ridgecv_pipe.fit(X_train, y_train);
```

[29]:
```
best_alpha = ridgecv_pipe.named_steps['ridgecv'].alpha_
best_alpha
```

[29]:
```
100.0
```

1.4.5 Let's examine the coefficients

[30]:
```
alphas = 10.0 ** np.arange(-5, 5, 1)
lr_tuned = make_pipeline(preprocessor, Ridge(alpha=best_alpha))
lr_tuned.fit(X_train, y_train)
lr_preds = lr_tuned.predict(X_test)
lr_preds[:10]
```

[30]:
```
array([228728.1963872 , 104718.39905565, 155778.96723311, 246316.71119031,
       127633.10676873, 243207.19441128, 304930.24461291, 145374.59435295,
       157059.38983893, 128487.51979632])
```

[31]:
```
lr_preds.max(), lr_preds.min()
```

[31]:
```
(390726.1064742326, 30791.092505420827)
```

Let's get the feature names of the transformed data.

[32]:
```
ohe_columns = list(
    preprocessor.named_transformers_["pipeline-4"]
    .named_steps["onehotencoder"]
    .get_feature_names(categorical_features)
)
new_columns = numeric_features + ordinal_features_reg + ordinal_features_oth +␣
 ↪ohe_columns
```

[33]:
```
df = pd.DataFrame(
    data={
        "features": new_columns,
        "coefficients": lr_tuned.named_steps["ridge"].coef_,
    }
)
```

[34]:
```
df.sort_values("coefficients",ascending=False)
```

[34]:
```
                  features  coefficients
4             OverallQual  14484.902165
16              GrLivArea  11704.053037
145  Neighborhood_NridgHt   9662.969631
144  Neighborhood_NoRidge   9497.598615
36               BsmtQual   8073.088562
..                    …           …
154      RoofMatl_ClyTile  -3992.399179
234       LandContour_Bnk  -5001.996997
137  Neighborhood_Gilbert  -5197.585536
134  Neighborhood_CollgCr  -5467.463086
136  Neighborhood_Edwards  -5796.508529

[263 rows x 2 columns]
```

So according to this model:

- As `OverallQual` feature gets bigger the housing price will get bigger.
- Presence of `Neighborhood_Edwards` will result in smaller median house value.

[35]:
```
X_train_enc['Neighborhood_Edwards']
```

[35]:
```
302     0.0
767     0.0
429     0.0
1139    0.0
558     0.0
       …
1041    0.0
1122    1.0
1346    0.0
1406    0.0
1389    0.0
Name: Neighborhood_Edwards, Length: 1314, dtype: float64
```

1.5 Regression score functions

- We aren't doing classification anymore, so we can't just check for equality:

[36]:
```
lr_tuned.predict(X_train) == y_train
```

[36]:
```
302     False
767     False
429     False
1139    False
558     False
       …
1041    False
1122    False
```

```
1346    False
1406    False
1389    False
Name: SalePrice, Length: 1314, dtype: bool
```

[37]:
```
y_train.values
```

[37]:
```
array([205000, 160000, 175000, …, 262500, 133000, 131000])
```

[38]:
```
lr_tuned.predict(X_train)
```

[38]:
```
array([212894.62756285, 178502.78223444, 189937.18327372, …,
       245233.6751565 , 129863.13373552, 135439.89186716])
```

We need a **score** that reflects **how right/wrong** each prediction is.

A number of popular scoring functions for regression. We are going to look at some common metrics:

- mean squared error (MSE)
- R^2
- root mean squared error (RMSE)
- MAPE

See sklearn documentation for more details.

1.5.1 Mean squared error (MSE)

- A common metric is mean squared error:

[39]:
```
preds = lr_tuned.predict(X_train)
```

[40]:
```
np.mean((y_train - preds) ** 2)
```

[40]:
```
873230473.3636096
```

Perfect predictions would have MSE=0.

[41]:
```
np.mean((y_train - y_train) ** 2)
```

[41]:
```
0.0
```

This is also implemented in sklearn:

[42]:
```
from sklearn.metrics import mean_squared_error

mean_squared_error(y_train, preds)
```

[42]:
```
873230473.3636096
```

- MSE looks huge and unreasonable. There is an error of ~$1 Billion!
- Is this score good or bad?

- Unlike classification, with regression **our target has units**.
- The target is in dollars, the mean squared error is in $dollars^2$
- 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!

[43]:
```
np.mean((y_train * 100 - preds * 100) ** 2)
```

[43]:
```
8732304733636.096
```

1.5.2 Root mean squared error or RMSE

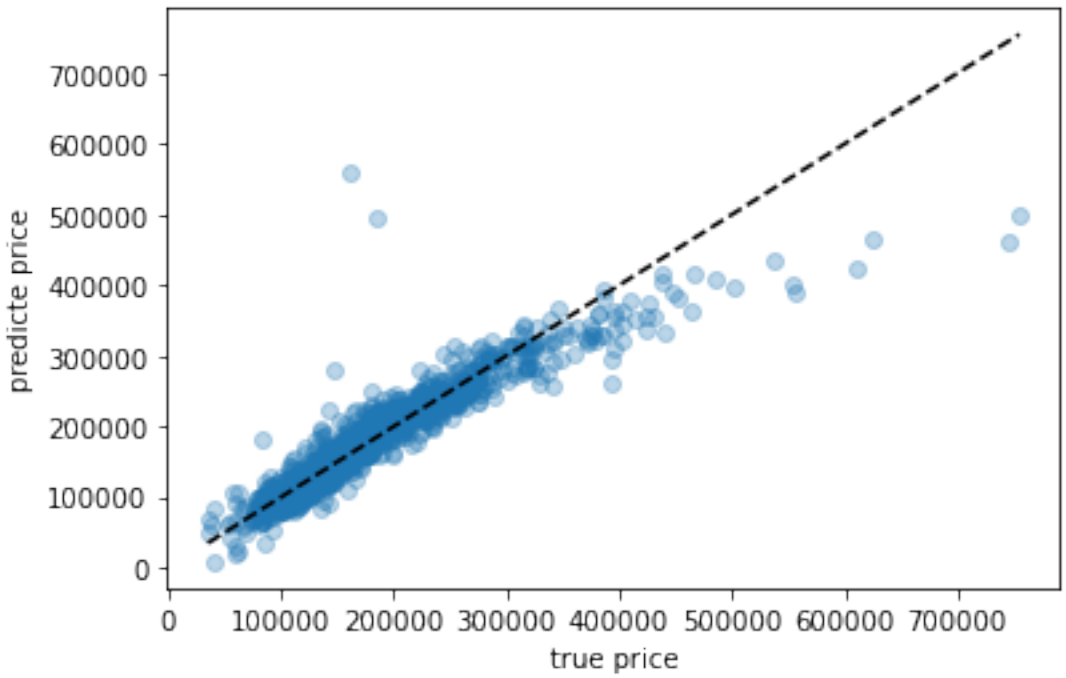
- The MSE above is in $dollars^2$.
- A more relatable metric would be the root mean squared error, or RMSE

[44]:
```
np.sqrt(mean_squared_error(y_train, lr_tuned.predict(X_train)))
```

[44]:
```
29550.4733187746
```

- Error of $30,000 makes more sense.
- Can we dig deeper?

[45]:
```
plt.scatter(y_train, lr_tuned.predict(X_train), alpha=0.3)
grid = np.linspace(y_train.min(), y_train.max(), 1000)
plt.plot(grid, grid, "--k")
plt.xlabel("true price")
plt.ylabel("predicte price");
```



- Here we can see a few cases where our prediction is way off.
- Is there something weird about those houses, perhaps? Outliers?
- Under the line means we're **under-predicting**, over the line means we're **over-predicting**.

1.5.3 R^2 (not in detail)

A common score is the R^2

- This is the score that `sklearn` uses by default when you call score():
- You can read about it if interested.
- **Intuition**: similar to mean squared error, but flipped (higher is better), and normalized so the max is 1.

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

Key points: - The maximum is 1 for perfect predictions - Negative values are very bad: "worse than DummyRegressor" (very bad)

(optional) Warning: MSE is "reversible" but R^2 is not:

[46]:
```
mean_squared_error(y_train, preds)
```

[46]:
```
873230473.3636096
```

[47]:
```
mean_squared_error(preds, y_train)
```

[47]:
```
873230473.3636096
```

[48]:
```
r2_score(y_train, preds)
```

[48]:
```
0.8601212294857903
```

[49]:
```
r2_score(preds, y_train)
```

[49]:
```
0.8279622258827071
```

- When you call `fit` it minimizes MSE / maximizes R^2 (or something like that) by default.
- Just like in classification, this isn't always what you want!!

1.5.4 MAPE

- We got an RMSE of ~$30,000 before.

Question: Is an error of $30,000 **acceptable**?

[50]:
```
np.sqrt(mean_squared_error(y_train, lr_tuned.predict(X_train)))
```

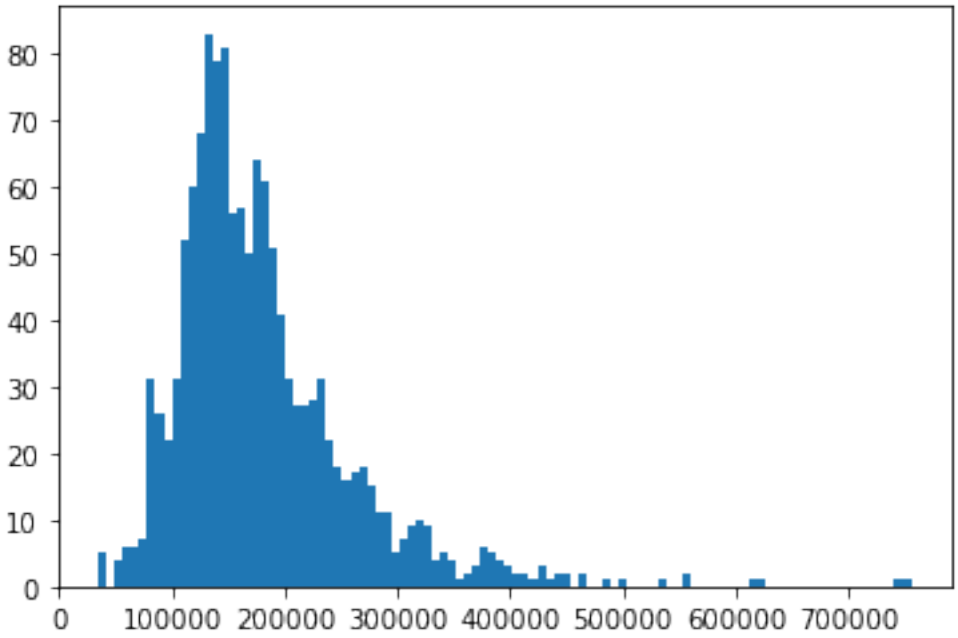
[50]:
```
29550.4733187746
```

- For a house worth $600k, it seems reasonable! That's 5% error.

- For a house worth $60k, that is terrible. It's 50% error.

We have both of these cases in our dataset.

[51]:
```
plt.hist(y_train, bins=100);
```



How about looking at percent error?

[52]:
```
pred_train = lr_tuned.predict(X_train)
percent_errors = (pred_train - y_train) / y_train * 100.0
percent_errors
```

[52]:
```
302      3.851038
767     11.564239
429      8.535533
1139   -16.371069
558     17.177968
           ...
1041    -0.496571
1122   -28.696351
1346    -6.577648
1406    -2.358546
1389     3.389230
Name: SalePrice, Length: 1314, dtype: float64
```

These are both positive (predict too high) and negative (predict too low).

We can look at the absolute percent error:

[53]:
```
np.abs(percent_errors)
```

[53]:
```
302      3.851038
767     11.564239
429      8.535533
1139    16.371069
558     17.177968
           …
1041     0.496571
1122    28.696351
1346     6.577648
1406     2.358546
1389     3.389230
Name: SalePrice, Length: 1314, dtype: float64
```

And, like MSE, we can take the average over examples. This is called mean absolute percent error (MAPE).

[54]:
```
def mape(true, pred):
    return 100.0 * np.mean(np.abs((pred - true) / true))
```

[55]:
```
mape(y_train, pred_train)
```

[55]:
```
10.093121294225256
```

- Ok, this is quite interpretable.
- On average, we have around 10% error.

1.5.5 Transforming the targets

- Does `.fit()` know we care about MAPE?
- No, it doesn't. Why are we minimizing MSE (or something similar) if we care about MAPE??
- When minimizing MSE, the **expensive houses** will *dominate* because they have the **biggest error**.
- Which is better for RMSE?

Model A

- Example 1: Truth: \$50k, Prediction: \$100k
- Example 2: Truth: \$500k, Prediction: \$550k
- RMSE: \$50k
- MAPE: 45%

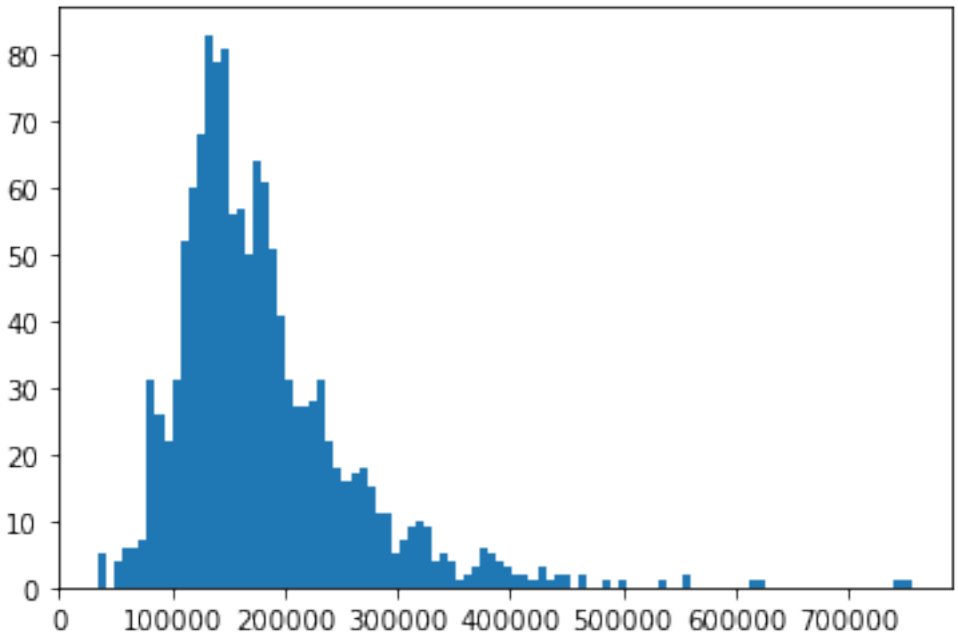
Model B

- Example 1: Truth: \$50k, Prediction: \$60k
- Example 2: Truth: \$500k, Prediction: \$600k
- RMSE: \$71k
- MAPE: 20%

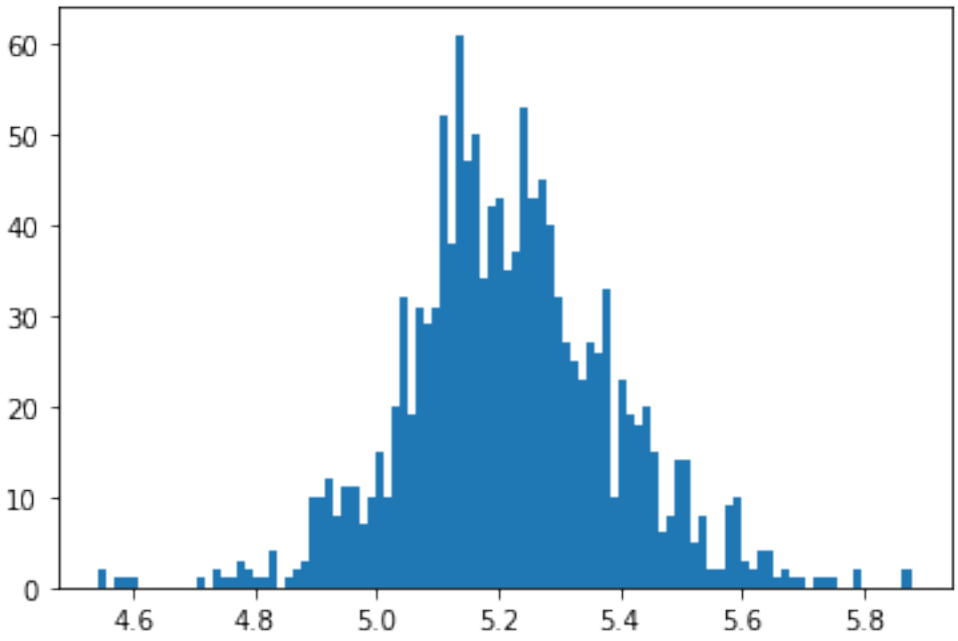
- How can we get `.fit()` to think about MAPE?

- A common practice which tends to work is **log transforming the targets**.
- That is, transform $y \rightarrow \log(y)$.

[56]:
```
plt.hist(y_train, bins=100);
```



[57]:
```
plt.hist(np.log10(y_train), bins=100);
```



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

[58]:
```
from sklearn.compose import TransformedTargetRegressor
```

[59]:
```
ttr = TransformedTargetRegressor(
    Ridge(alpha=best_alpha), func=np.log1p, inverse_func=np.expm1
) # transformer for log transforming the target
ttr_pipe = make_pipeline(preprocessor, ttr)
```

[60]:
```
ttr_pipe.fit(X_train, y_train); # y_train automatically transformed
```

[61]:
```
ttr_pipe.predict(X_train)  # predictions automatically un-transformed
```

[61]:
```
array([221355.29528077, 170663.43286226, 182608.09768702, …,
       248575.94877669, 132148.9047652 , 133262.17638244])
```

[62]:
```
mape(y_test, ttr_pipe.predict(X_test))
```

[62]:
```
7.8086009242408565
```

We reduced MAPE from ~10% to ~8% with this trick!

1.5.6 Different scoring functions with `cross_validate`

- Let's try using MSE instead of the default R^2 score.

[63]:
```
pd.DataFrame(
    cross_validate(
        lr_tuned,
        X_train,
        y_train,
        return_train_score=True,
        scoring="neg_mean_squared_error",
    )
)
```

[63]:

	fit_time	score_time	test_score	train_score
0	0.070695	0.022839	-7.060346e+08	-9.383069e+08
1	0.050639	0.014701	-1.239851e+09	-8.267971e+08
2	0.037472	0.016466	-1.125125e+09	-8.763019e+08
3	0.044787	0.015851	-9.819320e+08	-8.847908e+08
4	0.037190	0.012614	-2.268434e+09	-7.397199e+08

[64]:
```
def mape(true, pred):
    return 100.0 * np.mean(np.abs((pred - true) / true))
```

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

pd.DataFrame(
    cross_validate(
        lr_tuned, X_train, y_train, return_train_score=True, scoring=mape_scorer
    )
)
```

[64]:

	fit_time	score_time	test_score	train_score
0	0.062661	0.018162	9.699277	10.407124
1	0.055550	0.022333	10.803043	9.966190
2	0.056732	0.017879	11.836195	10.180734
3	0.065101	0.024593	10.784686	10.247198
4	0.049136	0.024848	12.196718	9.828607

[65]:
```
scoring = {
    "r2": "r2",
    "mape_scorer": mape_scorer,
    "neg_root_mean_square_error": "neg_root_mean_squared_error",
    "neg_mean_squared_error": "neg_mean_squared_error",
}

pd.DataFrame(
    cross_validate(lr_tuned, X_train, y_train, return_train_score=True,␣
 ↪scoring=scoring)
).T
```

[65]:

	0	1	2	3	4
fit_time	7.441640e-02	7.050776e-02	7.121372e-02	4.622102e-02	6.876326e-02
score_time	2.501154e-02	2.491736e-02	1.776671e-02	2.255344e-02	2.450657e-02
test_r2	8.668969e-01	8.200460e-01	8.262644e-01	8.511854e-01	6.109505e-01
train_r2	8.551369e-01	8.636241e-01	8.579735e-01	8.561893e-01	8.834054e-01
test_mape_scorer	9.699277e+00	1.080304e+01	1.183620e+01	1.078469e+01	1.219672e+01
train_mape_scorer	1.040712e+01	9.966190e+00	1.018073e+01		
test_neg_root_mean_square_error	-2.657131e+04	-3.521152e+04	-3.354288e+04		
train_neg_root_mean_square_error	-3.063179e+04	-2.875408e+04	-2.960240e+04		
test_neg_mean_squared_error	-7.060346e+08	-1.239851e+09	-1.125125e+09		
train_neg_mean_squared_error	-9.383069e+08	-8.267971e+08	-8.763019e+08		

```
train_mape_scorer                 1.024720e+01  9.828607e+00
test_neg_root_mean_square_error  -3.133579e+04 -4.762808e+04
train_neg_root_mean_square_error -2.974543e+04 -2.719779e+04
test_neg_mean_squared_error      -9.819320e+08 -2.268434e+09
train_neg_mean_squared_error     -8.847908e+08 -7.397199e+08
```

[66]:
```
mape(y_test, lr_tuned.predict(X_test))
```

[66]:
```
9.496387589495999
```

1.5.7 Using regression metrics with `scikit-learn`

- In `sklearn` you will notice that it has negative version of the metrics above (e.g., `neg_mean_squared_error`, `neg_root_mean_squared_error`).
- The reason for this is that scores return a value to maximize, **the higher the better**.
- If you define your own scorer function and if you do not want this interpretation, you can set the **`greater_is_better` parameter to False**

1.6 Questions for class discussion

1.6.1 True/False

1. Price per square foot would be a good feature to add in our `X`. **TRUE**
2. The `alpha` hyperparameter of `Ridge` has similar interpretation of `C` hyperparameter of `LogisticRegression`; higher `alpha` means more complex model. **FALSE**
3. In regression, one should use MAPE instead of MSE when relative (percent) error matters more than absolute error. **TRUE**
4. A lower RMSE value indicates a better model. **FALSE** not always
5. We can use still use precision and recall for regression problems but now we have other metrics we can use as well. **FALSE** Precision and recall are classification metrics.

1.7 Summary

- 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.

- 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
- MAPE (mean average percent error) is unitless; 0 is best, 100 is bad