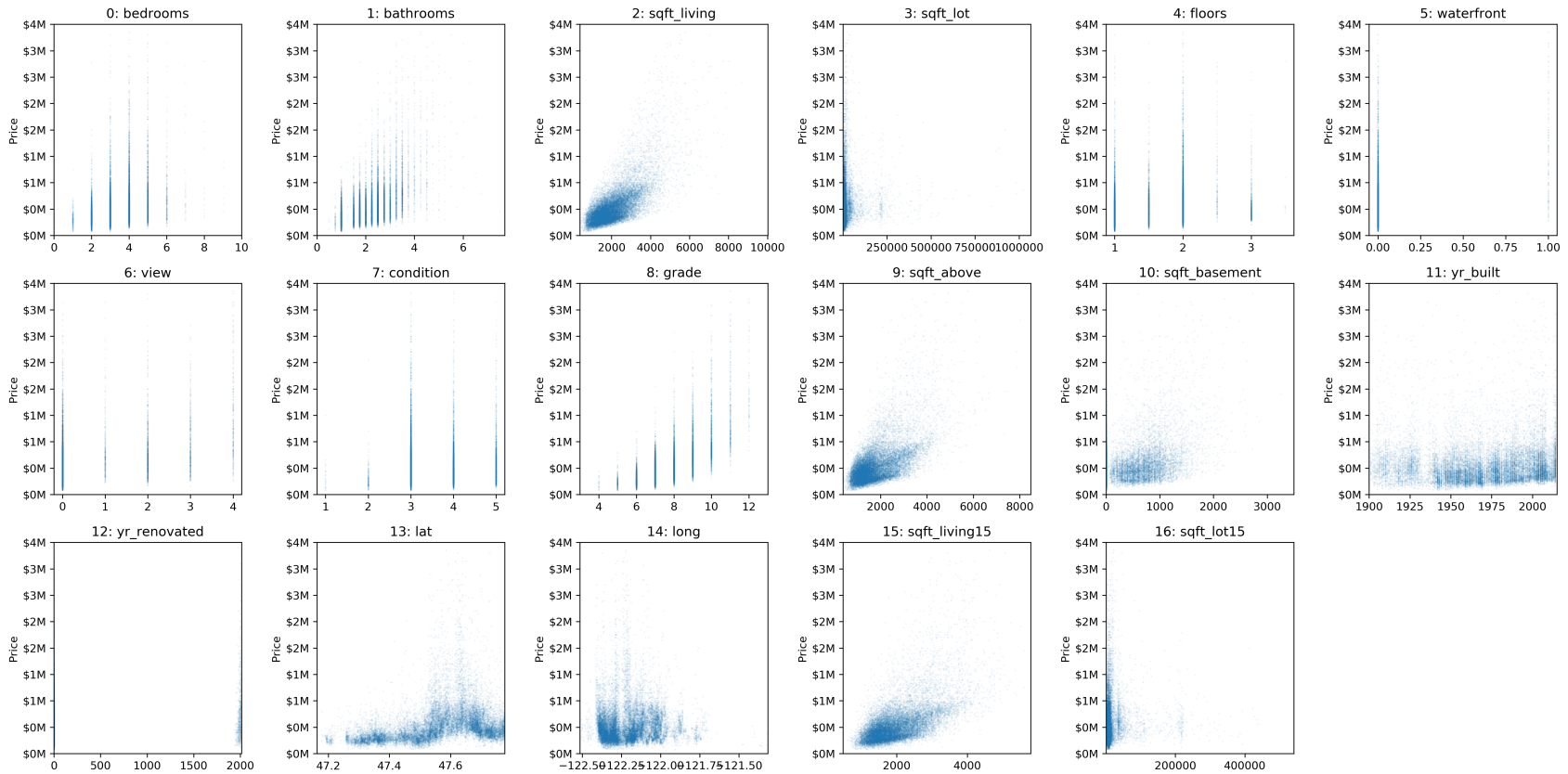
Preprocessing

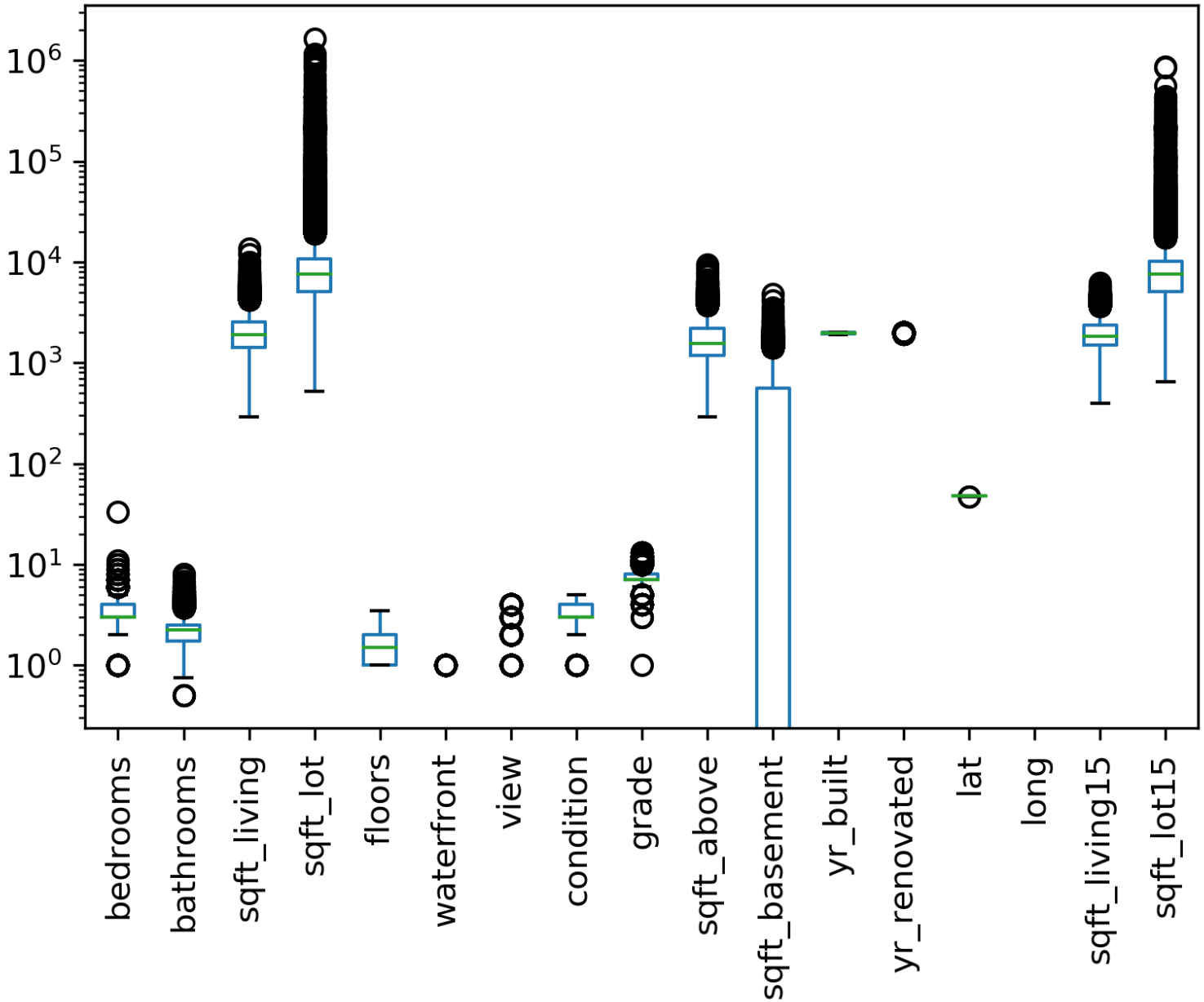
Presented by Yasin Ceran

Table of Contents

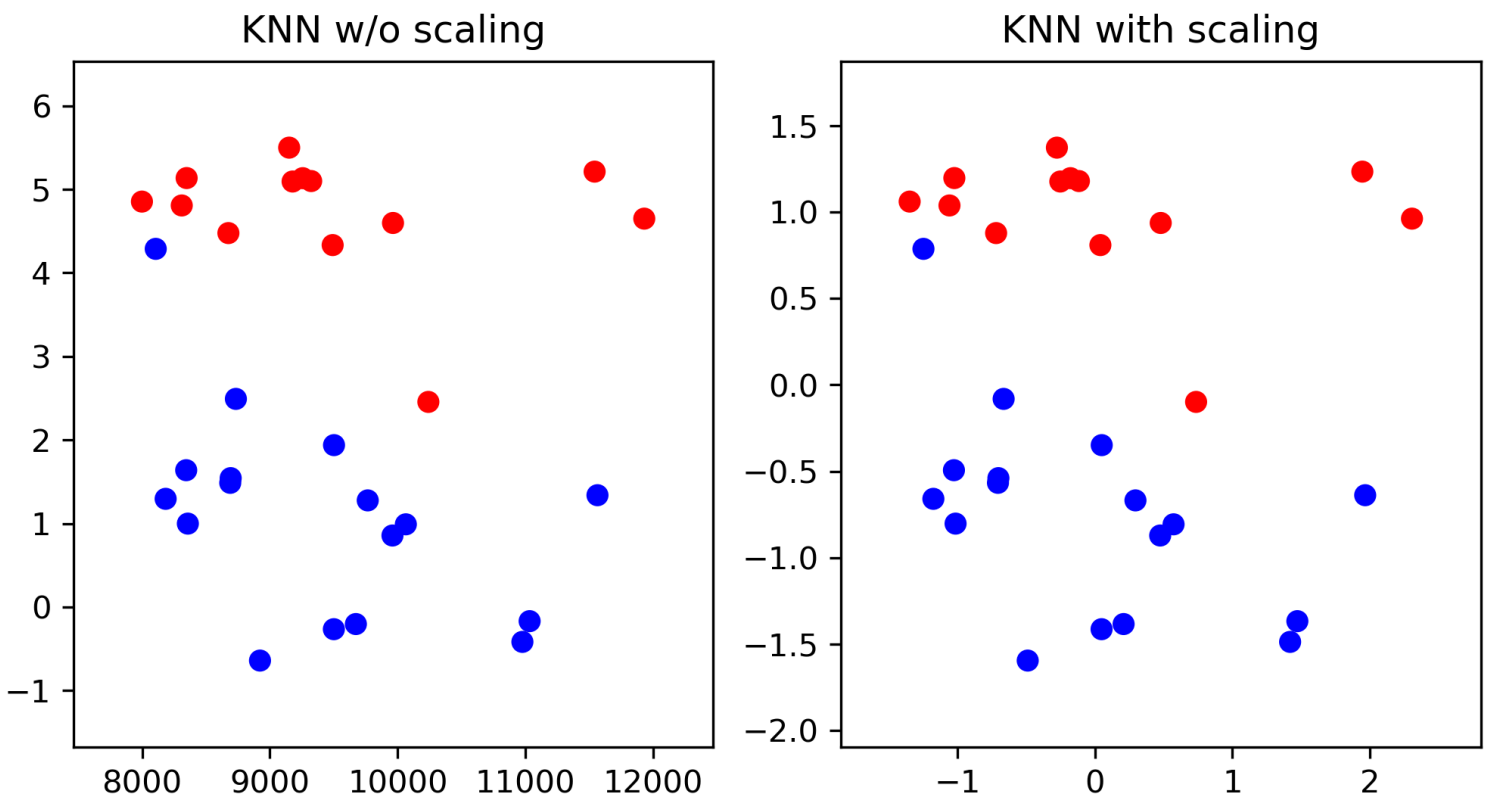
Scatter Plot of King County House Data



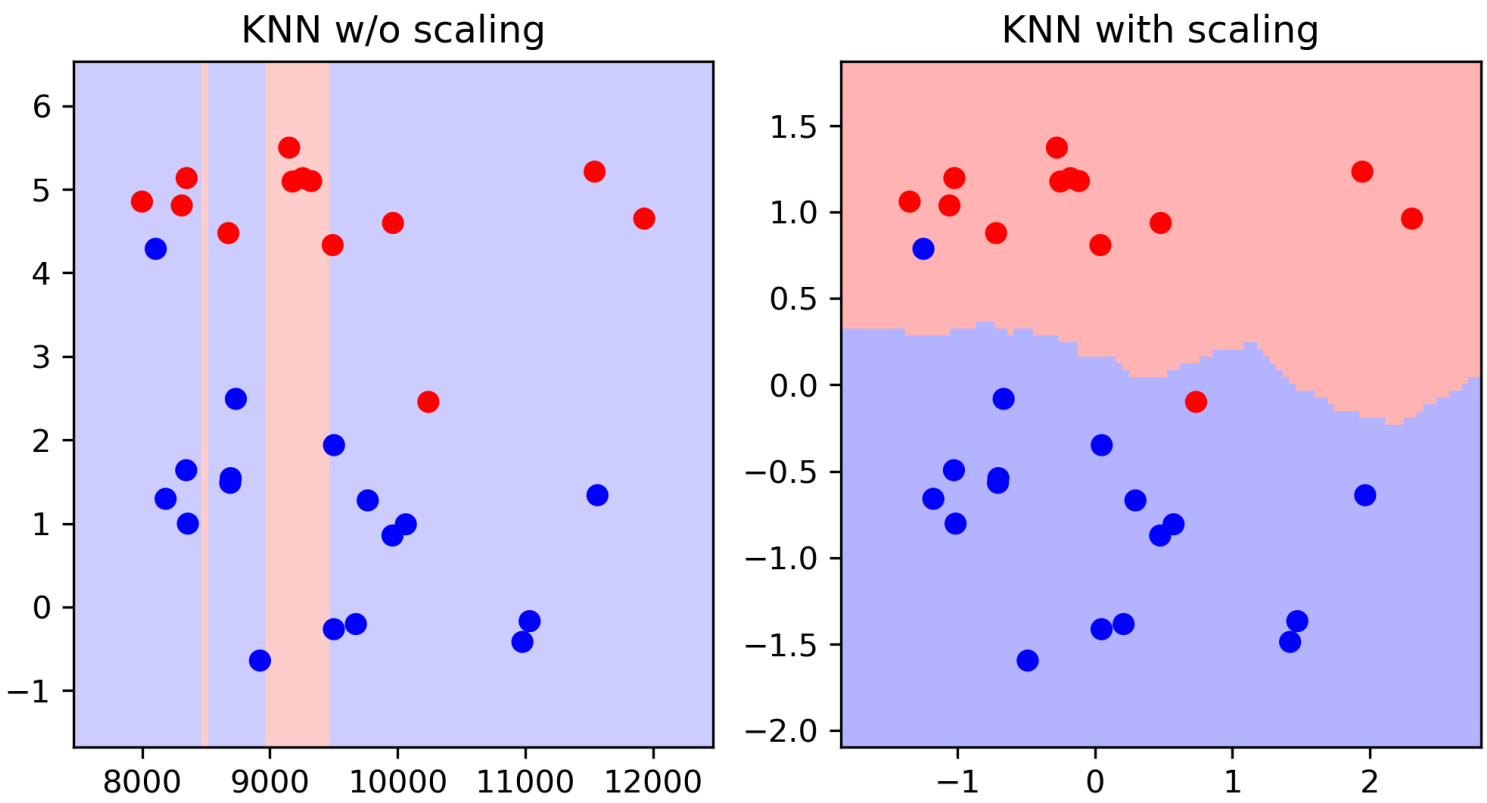
Box Plot of King County House Data



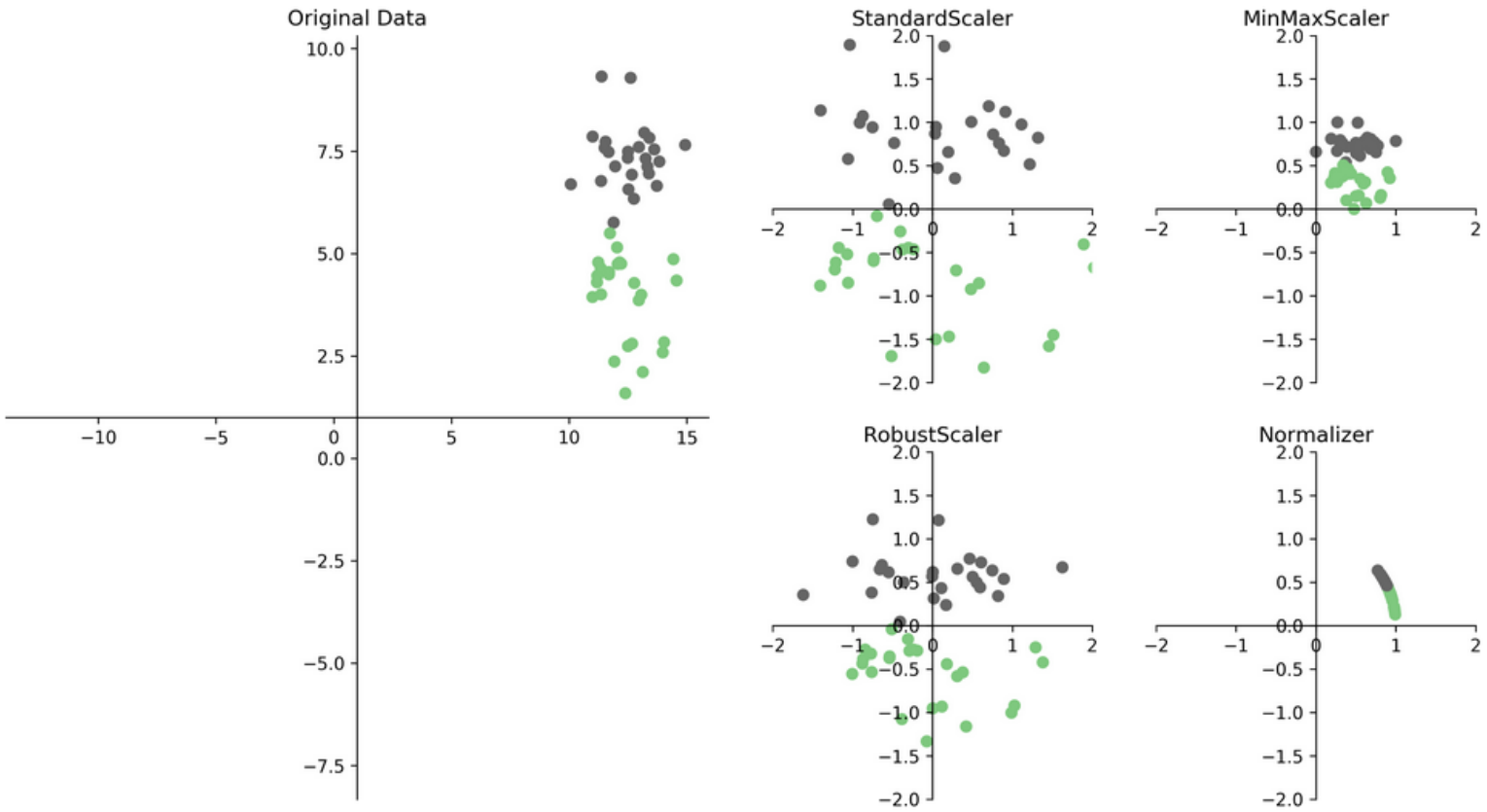
Scaling and Distances



Scaling and Distances



Ways to Scale



Sparse Data

- Data with many zeros – only store non-zero entries.
- Subtracting anything will make the data "dense" (no more zeros) and blow the RAM.
- Only scale, don't center (use MaxAbsScaler)

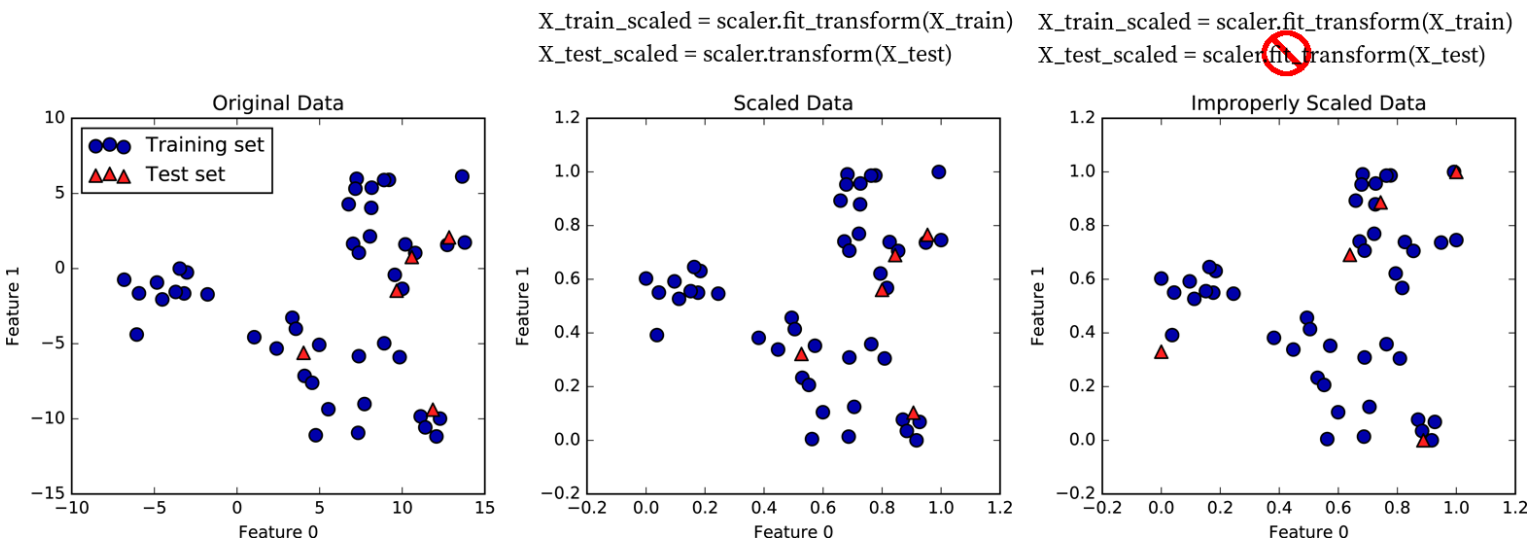
Standard Scaler Example

```
# Back to King Country house prices
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=0)

scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)

ridge = Ridge().fit(X_train_scaled, y_train)
X_test_scaled = scaler.transform(X_test)
ridge.score(X_test_scaled, y_test)

0.684
```



Sckit-Learn API Summary

estimator.fit(X, [y])

estimator.predict	estimator.transform
Classification	Preprocessing
Regression	Dimensionality reduction
Clustering	Feature selection
	Feature extraction

```
1
2 # Efficient Shortcuts:
3
4 est.fit_transform(X) == est.fit(X).transform(X)  # mostly
5 est.fit_predict(X) == est.fit(X).predict(X)   # mostly
```

Scaling In Action

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import RidgeCV
scores = cross_val_score(RidgeCV(), X_train, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.694, 0.027)
```

```
scores = cross_val_score(RidgeCV(), X_train_scaled, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.694, 0.027)
```

```
scores = cross_val_score(KNeighborsRegressor(), X_train, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.500, 0.039)
```

```
scores = cross_val_score(KNeighborsRegressor(), X_train_scaled, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.786, 0.030)
```

Table of Contents

A common Error

```
print(X.shape)

(100, 10000)
```

```
# select most informative 5% of features
from sklearn.feature_selection import SelectPercentile, f_regression
select = SelectPercentile(score_func=f_regression, percentile=5)
select.fit(X, y)
X_selected = select.transform(X)
print(X_selected.shape)

(100, 500)
```

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge
np.mean(cross_val_score(Ridge(), X_selected, y))

0.90
```

```
ridge = Ridge().fit(X_selected, y)
X_test_selected = select.transform(X_test)
ridge.score(X_test_selected, y_test)

-0.18
```

A Common Error: Leaking Information

```python
# BAD!
select.fit(X, y)  # includes the cv test parts!
X_sel = select.transform(X)
scores = []
for train, test in cv.split(X, y):
    ridge = Ridge().fit(X_sel[train], y[train])
    score = ridge.score(X_sel[test], y[test])
    scores.append(score)
```

```python
# GOOD!
scores = []
for train, test in cv.split(X, y):
    select.fit(X[train], y[train])
    X_sel_train = select.transform(X[train])
    ridge = Ridge().fit(X_sel_train, y[train])
    X_sel_test = select.transform(X[test])
    score = ridge.score(X_sel_test, y[test])
    scores.append(score)
```

Need to include preprocessing in cross-validation !

Scaling with Pipeline

```
# Housing data example
from sklearn.linear_model import Ridge
X, y = df, target

scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
ridge = Ridge().fit(X_train_scaled, y_train)

X_test_scaled = scaler.transform(X_test)
ridge.score(X_test_scaled, y_test)

0.684
```

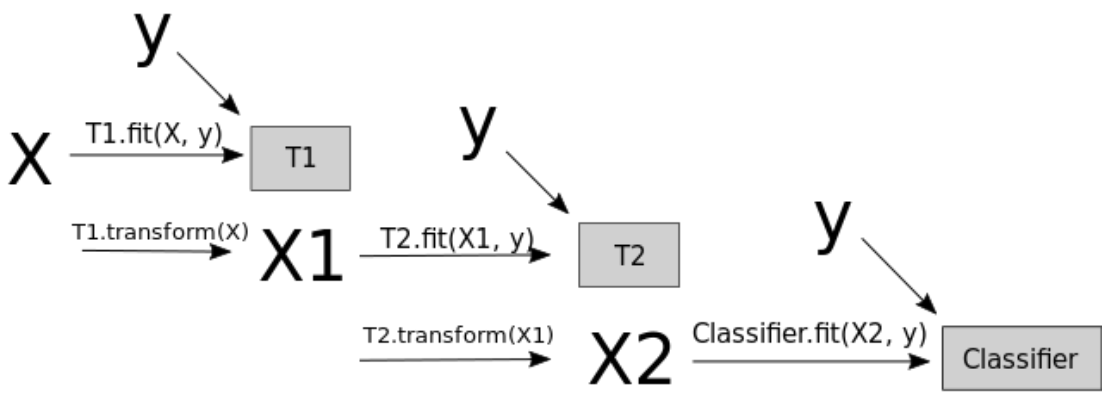
```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(StandardScaler(), Ridge())
pipe.fit(X_train, y_train)
pipe.score(X_test, y_test)

0.684
```

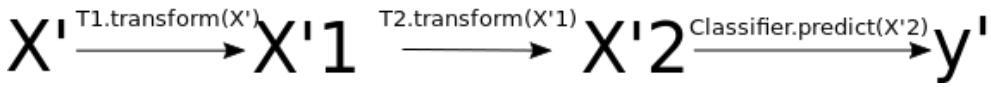
Pipeline

```
pipe = make_pipeline(T1(), T2(), Classifier())
```


```
pipe.fit(X, y)
```




```
pipe.predict(X')
```



Correcting Our Previous Error

```
# BAD!
select.fit(X, y)  # includes the cv test parts!
X_sel = select.transform(X)
scores = []
for train, test in cv.split(X, y):
    ridge = Ridge().fit(X_sel[train], y[train])
    score = ridge.score(X_sel[test], y[test])
    scores.append(score)
```

Same as:

```
select.fit(X, y)
X_selected = select.transform(X, y)
np.mean(cross_val_score(Ridge(), X_selected, y))

0.90
```

```
# GOOD!
scores = []
for train, test in cv.split(X, y):
    select.fit(X[train], y[train])
    X_sel_train = select.transform(X[train])
    ridge = Ridge().fit(X_sel_train, y[train])
    X_sel_test = select.transform(X[test])
    score = ridge.score(X_sel_test, y[test])
    scores.append(score)
```

Same as:

```
pipe = make_pipeline(select, Ridge())
np.mean(cross_val_score(pipe, X, y))

-0.079
```

Naming Steps

```

from sklearn.pipeline import make_pipeline
knn_pipe = make_pipeline(StandardScaler(), KNeighborsRegressor())
print(knn_pipe.steps)

[('standardscaler', StandardScaler()),
 ('kneighborsregressor', KNeighborsRegressor())]
```

```

from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", StandardScaler()),
                 ("regressor", KNeighborsRegressor())])
```

Pipeline and GridSearchCV

```
1
2
3  from sklearn.model_selection import GridSearchCV
4
5  knn_pipe = make_pipeline(StandardScaler(), KNeighborsRegressor())
6  param_grid = {'kneighborsregressor__n_neighbors': range(1, 10)}
7  grid = GridSearchCV(knn_pipe, param_grid, cv=10)
8  grid.fit(X_train, y_train)
9  print(grid.best_params_)
10 print(grid.score(X_test, y_test))
11
12 {'kneighborsregressor__n_neighbors': 7}
13 0.60
```

Going Wild with Pipeline

```
1
2
3  from sklearn.datasets import load_diabetes
4  diabetes = load_diabetes()
5  X_train, X_test, y_train, y_test = train_test_split(
6      diabetes.data, diabetes.target, random_state=0)
7
8  from sklearn.preprocessing import PolynomialFeatures
9  pipe = make_pipeline(
10     StandardScaler(),
11     PolynomialFeatures(),
12     Ridge())
13
14 param_grid = {'polynomialfeatures__degree': [1, 2, 3],
15                'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
16 grid = GridSearchCV(pipe, param_grid=param_grid,
17                     n_jobs=-1, return_train_score=True)
18 grid.fit(X_train, y_train)
```

Going Wilder with Pipeline

```
1
2
3  pipe = Pipeline([('scaler', StandardScaler()),
4                   ('regressor', Ridge())])
5
6  param_grid = {'scaler': [StandardScaler(), MinMaxScaler(),
7                           'passthrough'],
8                'regressor': [Ridge(), Lasso()],
9                'regressor__alpha': np.logspace(-3, 3, 7)}
10
11
12 grid = GridSearchCV(pipe, param_grid)
13 grid.fit(X_train, y_train)
14 grid.score(X_test, y_test)
```

Going Wildest with Pipeline

```
1
2
3  from sklearn.tree import DecisionTreeRegressor
4  pipe = Pipeline([('scaler', StandardScaler()),
5                   ('regressor', Ridge())])
6
7  # check out searchgrid for more convenience
8  param_grid = [{'regressor': [DecisionTreeRegressor()],
9                 'regressor__max_depth': [2, 3, 4],
10                'scaler': ['passthrough']},
11               {'regressor': [Ridge()],
12                'regressor__alpha': [0.1, 1],
13                'scaler': [StandardScaler(), MinMaxScaler(),
14                           'passthrough']}
15              ]
16 grid = GridSearchCV(pipe, param_grid)
17 grid.fit(X_train, y_train)
18 grid.score(X_test, y_test)
```

Table of Contents

Dealing with missing values

- Missing values can be encoded in many ways
- Numpy has no standard format for it (often np.NaN) - pandas does
- Sometimes: 999, ???, ?, np.inf, “N/A”, “Unknown”...
- Not discussing “missing output” - that’s semi-supervised learning.
- Often missingness is informative (Use ‘MissingIndicator‘)

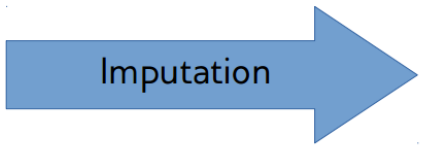
How Do Missing Values Look Like?

```
array([[ nan,  3.2,  5.7,  2.3],
       [ nan,  2.8,  4.9,  2. ],
       [ nan,  2.8,  6.7,  2. ],
       [ nan,  2.7,  4.9,  1.8],
       [ 6.7,  3.3,  5.7,  2.1],
       [ nan,  3.2,  6. ,  1.8],
       [ nan,  2.8,  4.8,  1.8],
       [ nan,  3. ,  4.9,  1.8],
       [ nan,  2.8,  5.6,  2.1],
       [ nan,  3. ,  5.8,  1.6],
       [ 7.4,  2.8,  6.1,  1.9],
       [ nan,  3.8,  6.4,  2. ],
       [ 6.4,  2.8,  5.6,  2.2],
       [ nan,  2.8,  5.1,  1.5],
       [ nan,  2.6,  5.6,  1.4],
       [ nan,  3. ,  6.1,  2.3],
       [ nan,  3.4,  5.6,  2.4],
       [ nan,  3.1,  5.5,  1.8],
       [ nan,  3. ,  4.8,  1.8],
       [ 6.9,  3.1,  5.4,  2.1],
       [ 6.7,  3.1,  5.6,  2.4],
       [ nan,  3.1,  5.1,  2.3],
       [ nan,  2.7,  5.1,  1.9],
       [ nan,  3.2,  5.9,  2.3],
       [ nan,  3.3,  5.7,  2.5],
       [ nan,  3. ,  5.2,  2.3],
       [ nan,  2.5,  5. ,  1.9],
       [ nan,  3. ,  5.2,  2. ],
       [ 6.2,  3.4,  5.4,  2.3],
       [ nan,  3. ,  5.1,  1.8]])
```

```
array([[ 5.1,  3.5,  1.4,  0.2],
       [ nan,  nan,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2],
       [ nan,  nan,  nan,  nan],
       [ 4.6,  3.4,  1.4,  0.3],
       [ 5. ,  3.4,  1.5,  0.2],
       [ 4.4,  2.9,  1.4,  0.2],
       [ 4.9,  3.1,  1.5,  0.1],
       [ 5.4,  3.7,  1.5,  0.2],
       [ 4.8,  3.4,  1.6,  0.2],
       [ 4.8,  3. ,  1.4,  0.1],
       [ 4.3,  3. ,  1.1,  0.1],
       [ nan,  nan,  nan,  nan],
       [ 5.7,  4.4,  1.5,  0.4],
       [ 5.4,  3.9,  1.3,  0.4],
       [ 5.1,  3.5,  1.4,  0.3],
       [ 5.7,  3.8,  1.7,  0.3],
       [ 5.1,  3.8,  1.5,  0.3],
       [ 5.4,  3.4,  1.7,  0.2],
       [ 5.1,  3.7,  1.5,  0.4],
       [ 4.6,  3.6,  1. ,  0.2],
       [ 5.1,  nan,  nan,  nan],
       [ 4.8,  3.4,  1.9,  0.2],
       [ 5. ,  3. ,  1.6,  0.2],
       [ nan,  nan,  nan,  0.4],
       [ 5.2,  3.5,  1.5,  0.2],
       [ 5.2,  3.4,  1.4,  0.2],
       [ 4.7,  3.2,  1.6,  0.2]])
```

How Do Missing Values Look Like?

```
array([[ 6. ,  3.4,  4.5,  nan],
       [ 6.9,  3.1,  5.1,  2.3],
       [ 4.6,  3.2,  1.4,  nan],
       [ 5.1,  3.8,  1.5,  nan],
       [ 4.4,  2.9,  nan,  nan],
       [ 6.6,  2.9,  4.6,  1.3],
       [ 6.7,  3. ,  5.2,  2.3],
       [ 6.3,  3.3,  6. ,  2.5],
       [ 7.2,  3. ,  5.8,  1.6],
       [ 4.6,  3.4,  1.4,  nan],
       [ 5.2,  3.5,  1.5,  nan],
       [ 5.4,  3.4,  nan,  nan],
       [ 5.9,  3.2,  4.8,  1.8],
       [ 4.9,  3.1,  nan,  nan],
       [ 6.9,  3.2,  5.7,  2.3],
       [ 5.7,  3.8,  nan,  0.3],
       [ 5.3,  3.7,  1.5,  nan],
       [ 4.5,  2.3,  1.3,  nan],
       [ 6.5,  3. ,  5.5,  1.8],
       [ 6.2,  2.9,  4.3,  1.3],
       [ 6.4,  2.8,  5.6,  2.2],
       [ 6.1,  3. ,  4.6,  1.4],
       [ 6.2,  2.8,  4.8,  1.8],
       [ 4.9,  2.5,  4.5,  1.7],
       [ 6. ,  2.7,  5.1,  nan],
       [ 6.8,  3.2,  5.9,  2.3],
       [ 6. ,  2.9,  4.5,  1.5],
       [ 4.9,  2.4,  3.3,  1. ],
       [ 5.8,  2.7,  5.1,  nan],
       [ 5.5,  2.4,  3.8,  nan]])
```




```
array([[ 6. ,  3.4,  4.5,  1.6],
       [ 6.9,  3.1,  5.1,  2.3],
       [ 4.6,  3.2,  1.4,  0.2],
       [ 5.1,  3.8,  1.5,  0.3],
       [ 4.4,  2.9,  1.4,  0.2],
       [ 6.6,  2.9,  4.6,  1.3],
       [ 6.7,  3. ,  5.2,  2.3],
       [ 6.3,  3.3,  6. ,  2.5],
       [ 7.2,  3. ,  5.8,  1.6],
       [ 4.6,  3.4,  1.4,  0.3],
       [ 5.2,  3.5,  1.5,  0.2],
       [ 5.4,  3.4,  1.5,  0.4],
       [ 5.9,  3.2,  4.8,  1.8],
       [ 4.9,  3.1,  1.5,  0.1],
       [ 6.9,  3.2,  5.7,  2.3],
       [ 5.7,  3.8,  1.7,  0.3],
       [ 5.3,  3.7,  1.5,  0.2],
       [ 4.5,  2.3,  1.3,  0.3],
       [ 6.5,  3. ,  5.5,  1.8],
       [ 6.2,  2.9,  4.3,  1.3],
       [ 6.4,  2.8,  5.6,  2.2],
       [ 6.1,  3. ,  4.6,  1.4],
       [ 6.2,  2.8,  4.8,  1.8],
       [ 4.9,  2.5,  4.5,  1.7],
       [ 6. ,  2.7,  5.1,  1.6],
       [ 6.8,  3.2,  5.9,  2.3],
       [ 6. ,  2.9,  4.5,  1.5],
       [ 4.9,  2.4,  3.3,  1. ],
       [ 5.8,  2.7,  5.1,  1.9],
       [ 5.5,  2.4,  3.8,  1.1]])
```

Imputation Methods

- Mean / Median
- kNN
- Regression models

Baseline: Dropping Columns

```
24 from sklearn.model_selection import train_test_split, cross_val_score
25 X_train, X_test, y_train, y_test = train_test_split(X_, y, stratify=y)
26
27 nan_columns = np.any(np.isnan(X_train), axis=0)
28 X_drop_columns = X_train[:, ~nan_columns]
29 scores = cross_val_score(LogisticRegressionCV(v=5), X_drop_columns, y_train, cv=10)
30 np.mean(scores)
31
32 0.772
```

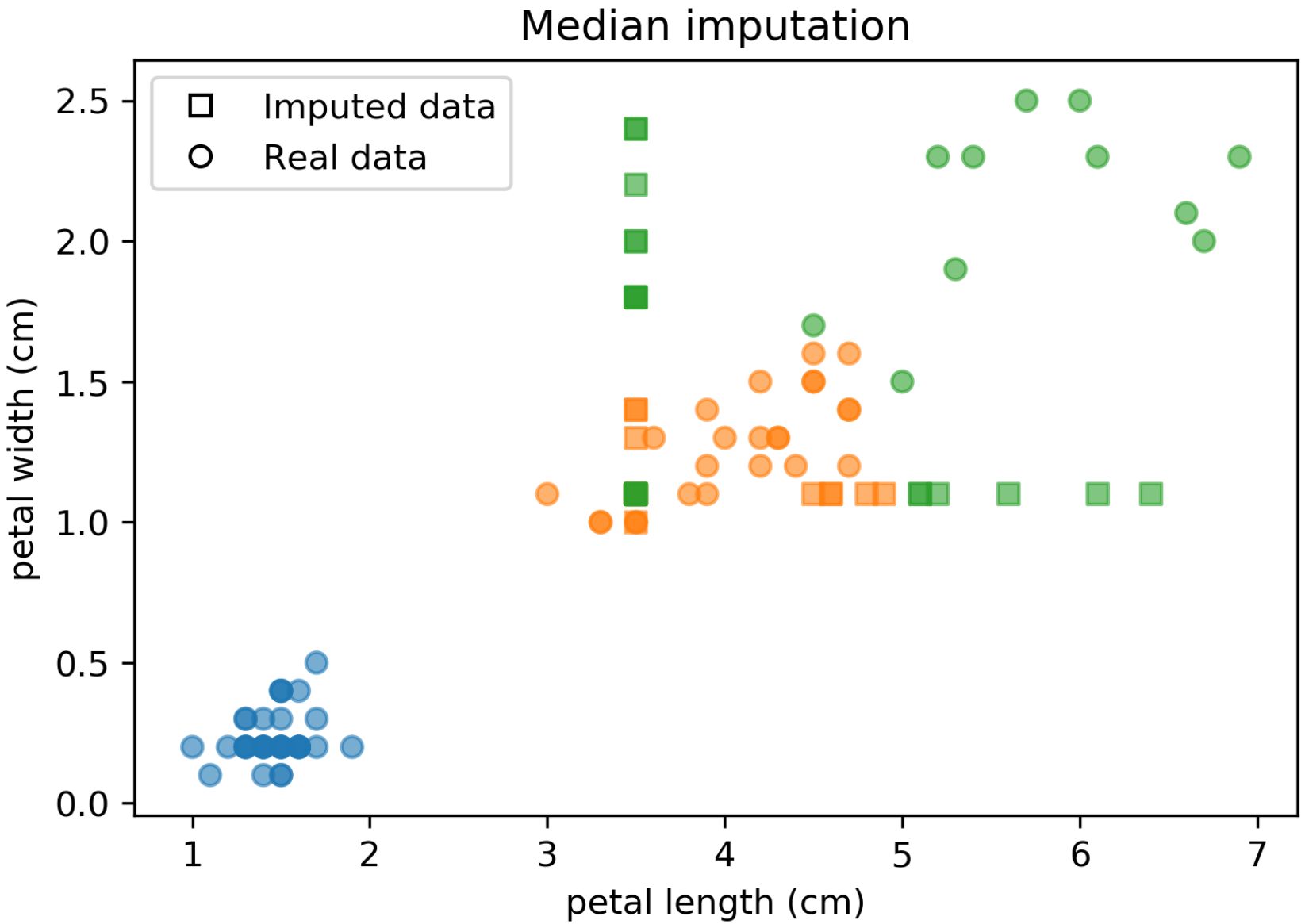
Mean and Median

```
[[ 6.   2.9  4.5  1.5]
 [ 5.9  3.   5.1  1.8]
 [ 4.4  3.   1.3  0.2]
 [ 5.1  3.3  nan  nan]
 [ 5.   3.5  1.6  0.6]
 [ 5.4  3.4  nan  nan]
 [ 5.7  3.8  nan  0.3]
 [ 5.6  2.5  3.9  nan]
 [ 7.7  2.6  6.9  2.3]
 [ 5.8  2.7  5.1  1.9]
 [ 6.7  3.1  5.6  2.4]
 [ 4.8  3.4  1.9  nan]
 [ 7.2  3.2  6.   1.8]
 [ 4.4  2.9  nan  nan]
 [ 6.9  3.2  5.7  2.3]
 [ 5.5  4.2  1.4  nan]
 [ 6.3  2.3  4.4  1.3]
 [ 7.   3.2  4.7  1.4]
 [ 5.8  2.7  nan  nan]
 [ 6.8  2.8  4.8  1.4]
 [ 5.4  3.9  1.7  nan]
 [ 7.6  3.   6.6  2.1]
 [ 7.7  2.8  6.7  2. ]
 [ 5.   3.3  nan  0.2]
 [ 5.9  3.   4.2  1.5]
 [ 6.1  2.8  4.   1.3]
 [ 5.   3.6  1.4  0.2]
 [ 7.4  2.8  6.1  1.9]
 [ 6.3  2.5  5.   1.9]
 [ 6.7  3.3  5.7  2.5]]
```

```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(strategy="median").fit(X_train)
X_median_imp = imp.transform(X_train)
```


```
array([[ 6.   ,  2.9  ,  4.5  ,  1.5  ],
       [ 5.9  ,  3.   ,  5.1  ,  1.8  ],
       [ 4.4  ,  3.   ,  1.3  ,  0.2  ],
       [ 5.1  ,  3.3  ,  4.116,  1.462],
       [ 5.   ,  3.5  ,  1.6  ,  0.6  ],
       [ 5.4  ,  3.4  ,  4.116,  1.462],
       [ 5.7  ,  3.8  ,  4.116,  0.3  ],
       [ 5.6  ,  2.5  ,  3.9  ,  1.462],
       [ 7.7  ,  2.6  ,  6.9  ,  2.3  ],
       [ 5.8  ,  2.7  ,  5.1  ,  1.9  ],
       [ 6.7  ,  3.1  ,  5.6  ,  2.4  ],
       [ 4.8  ,  3.4  ,  1.9  ,  1.462],
       [ 7.2  ,  3.2  ,  6.   ,  1.8  ],
       [ 4.4  ,  2.9  ,  4.116,  1.462],
       [ 6.9  ,  3.2  ,  5.7  ,  2.3  ],
       [ 5.5  ,  4.2  ,  1.4  ,  1.462],
       [ 6.3  ,  2.3  ,  4.4  ,  1.3  ],
       [ 7.   ,  3.2  ,  4.7  ,  1.4  ],
       [ 5.8  ,  2.7  ,  4.116,  1.462],
       [ 6.8  ,  2.8  ,  4.8  ,  1.4  ],
       [ 5.4  ,  3.9  ,  1.7  ,  1.462],
       [ 7.6  ,  3.   ,  6.6  ,  2.1  ],
       [ 7.7  ,  2.8  ,  6.7  ,  2.   ],
       [ 5.   ,  3.3  ,  4.116,  0.2  ],
       [ 5.9  ,  3.   ,  4.2  ,  1.5  ],
       [ 6.1  ,  2.8  ,  4.   ,  1.3  ],
       [ 5.   ,  3.6  ,  1.4  ,  0.2  ],
       [ 7.4  ,  2.8  ,  6.1  ,  1.9  ],
       [ 6.3  ,  2.5  ,  5.   ,  1.9  ],
       [ 6.7  ,  3.3  ,  5.7  ,  2.5  ]])
```

Mean and Median-Graph



Baseline: Dropping Columns

```
33 from sklearn.preprocessing import StandardScalar
34
35 nan_columns = np.any(np.isnan(X_train), axis = 0)
36 X_drop_columns = X_train[:,~nan_columns]
37 logreg = make_pipeline(StandardScalar(), LogisticRegression())
38 scores = cross_val_score(logreg, X_drop_columns, y_train, cv=10)
39 np.mean(scores)
40
41 0.794
42
43 mean_pipe = make_pipeline(SimpleImputer(strategy='median'),
44                           StandardScalar(),
45                           LogisticRegression())
46 scores = cross_val_score(mean_pipe, X_train, y_train, cv=10)
47 np.mean(scores)
48
49 0.729
```

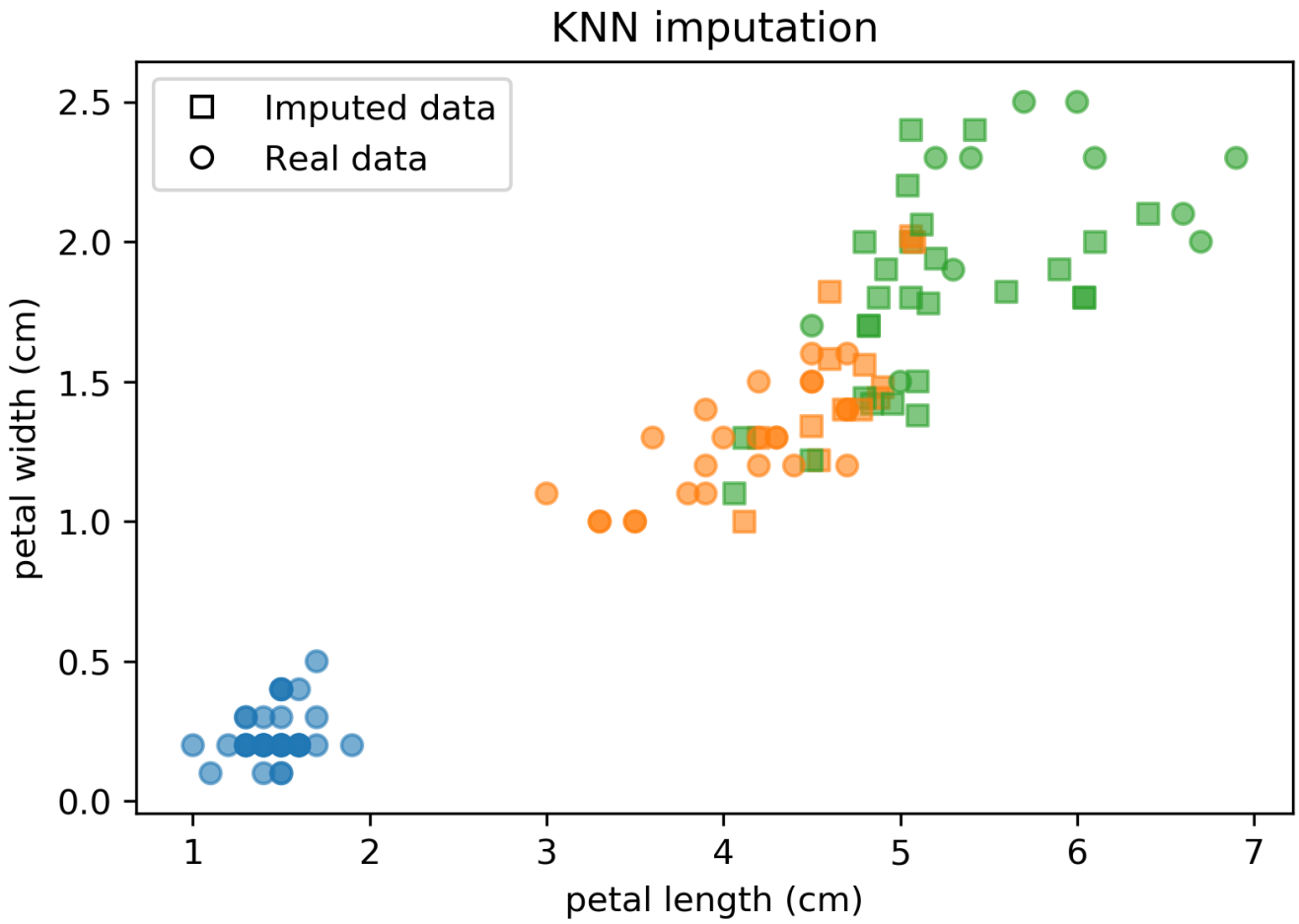
KNN Imputation

- Find k nearest neighbors that have non-missing values.
- Fill in all missing values using the average of the neighbors.

KNN Imputation-Code-Graph

```
knn_pipe = make_pipeline(KNNImputer(), StandardScaler(), LogisticRegression())
np.mean(scores)

0.849
```



Model-Driven Imputation

- Train regression model for missing values
- Iterate: retrain after filling in

```
rf_imp = IterativeImputer(predictor=RandomForestRegressor())
rf_pipe = make_pipeline(rf_imp, StandardScaler(), LogisticRegression())

scores = cross_val_score(rf_pipe, X_train, y_train, cv=10)
np.mean(scores)

0.845
```

Comparison of Imputation Methods

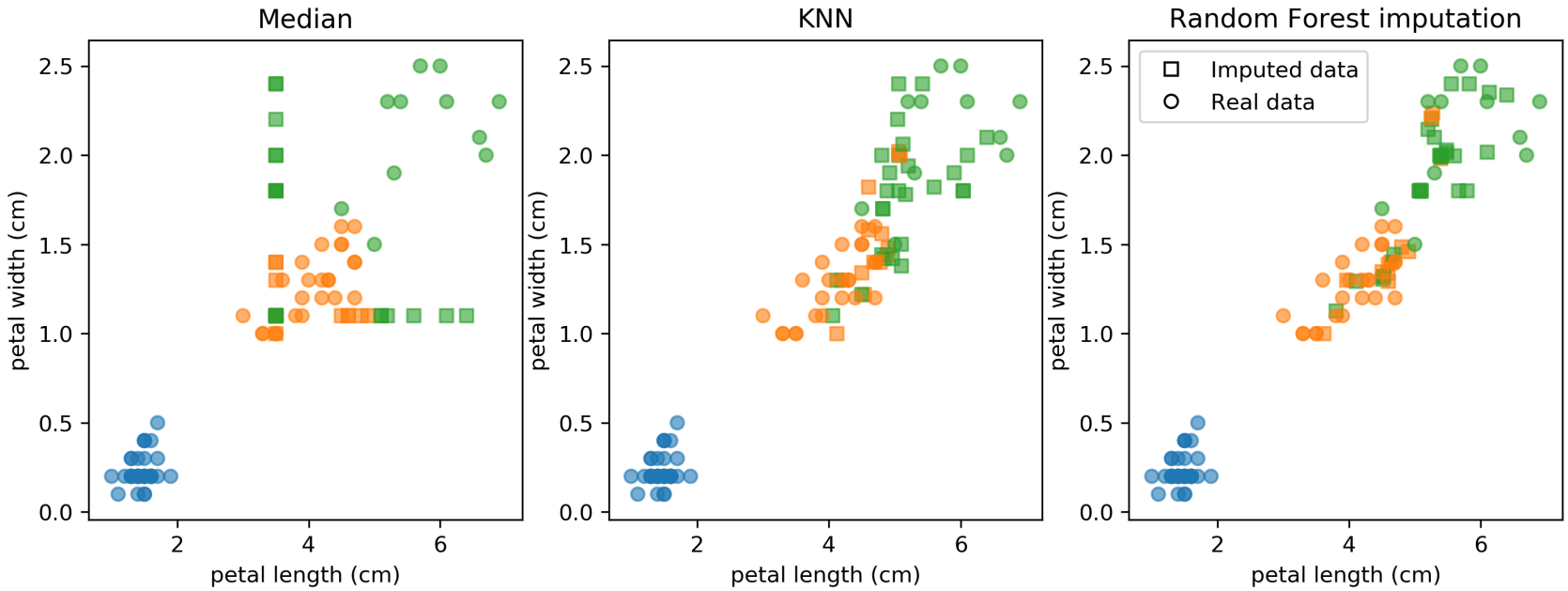


Table of Contents

Categorical Variables

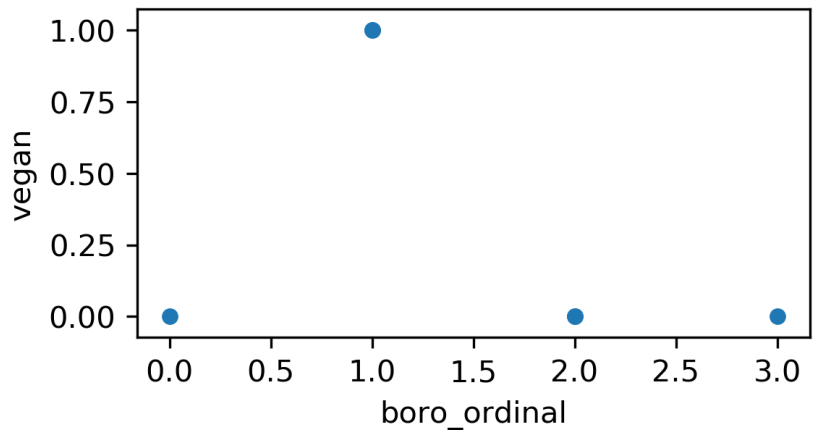
```
import pandas as pd
df = pd.DataFrame({
  'boro': ['Manhattan', 'Queens', 'Manhattan', 'Brooklyn', 'Brooklyn', 'Bronx'],
  'salary': [103, 89, 142, 54, 63, 219],
     'vegan': ['No', 'No','No','Yes', 'Yes', 'No']})
```

id	Boro	Salary	Vegan
0	Manhattan	103	No
1	Queens	89	No
2	Manhattan	142	No
3	Brooklyn	54	Yes
4	Brooklyn	63	Yes
5	Bronx	219	No

Ordinal Encoding

```
df['boro_ordinal'] = df.boro.astype("category").cat.codes
df
```

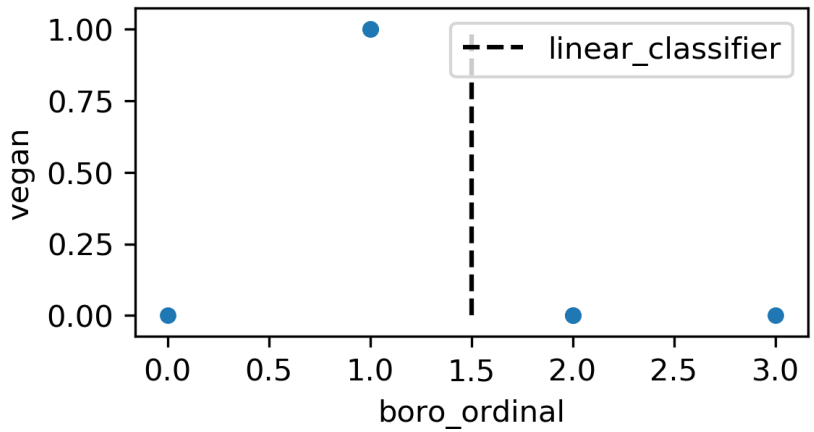
id	Boro	Salary	Vegan
0	2	103	No
1	3	89	No
2	2	142	No
3	1	54	Yes
4	1	63	Yes
5	0	219	No



Ordinal Encoding

```
df['boro_ordinal'] = df.boro.astype("category").cat.codes
df
```

id	Boro	Salary	Vegan
0	2	103	No
1	3	89	No
2	2	142	No
3	1	54	Yes
4	1	63	Yes
5	0	219	No



One-Hot (Dummy) Encoding

id	Boro	Salary	Vegan
0	Manhattan	103	No
1	Queens	89	No
2	Manhattan	142	No
3	Brooklyn	54	Yes
4	Brooklyn	63	Yes
5	Bronx	219	No

```
1 import pandas as pd
2 pd.get_dummies(df)
```

	salary	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens	vegan_No	vegan_Yes
0	103	0	0	1	0	1	0
1	89	0	0	0	1	1	0
2	142	0	0	1	0	1	0
3	54	0	1	0	0	0	1
4	63	0	1	0	0	0	1
5	219	1	0	0	0	1	0

One-Hot (Dummy) Encoding

id	Boro	Salary	Vegan
0	Manhattan	103	No
1	Queens	89	No
2	Manhattan	142	No
3	Brooklyn	54	Yes
4	Brooklyn	63	Yes
5	Bronx	219	No

```
1 import pandas as pd
2 pd.get_dummies(df, columns=['boro'])
```

	salary	vegan	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens
0	103	No	0	0	1	0
1	89	No	0	0	0	1
2	142	No	0	0	1	0
3	54	Yes	0	1	0	0
4	63	Yes	0	1	0	0
5	219	No	1	0	0	0

One-Hot (Dummy) Encoding

	Boro	Salary	Vegan
0	2	103	No
1	3	89	No
2	2	142	No
3	1	54	Yes
4	1	63	Yes
5	0	219	No

```
1 import pandas as pd
2 pd.get_dummies(df_ordinal, columns=['boro'])
```

	salary	vegan	boro_0	boro_1	boro_2	boro_3
0	103	No	0	0	1	0
1	89	No	0	0	0	1
2	142	No	0	0	1	0
3	54	Yes	0	1	0	0
4	63	Yes	0	1	0	0
5	219	No	1	0	0	0

One-Hot (Dummy) Encoding

```
df = pd.DataFrame({
                    'boro': ['Manhattan', 'Queens
    ', 'Manhattan',
             'Brooklyn', 'Brooklyn', 'Bronx'],
  'salary': [103, 89, 142, 54, 63, 219],
  'vegan': ['No', 'No','No','Yes', 'Yes', 'No'
      ]})
df_dummies = pd.get_dummies(df, columns=['boro']
```

	salary	vegan	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens
0	103	No	0	0	1	0
1	89	No	0	0	0	1
2	142	No	0	0	1	0
3	54	Yes	0	1	0	0
4	63	Yes	0	1	0	0
5	219	No	1	0	0	0

```
df = pd.DataFrame({
  'boro': ['Brooklyn', 'Manhattan', 'Brooklyn',
           'Queens', 'Brooklyn', 'Staten Island'
      ],
  'salary': [61, 146, 142, 212, 98, 47],
  'vegan': ['Yes', 'No','Yes','No', 'Yes', 'No'
      ]})
df_dummies = pd.get_dummies(df, columns=['boro'
      ])
```

	salary	vegan	boro_Brooklyn	boro_Manhattan	boro_Queens	boro_Staten Island
0	61	Yes	1	0	0	0
1	146	No	0	1	0	0
2	142	Yes	1	0	0	0
3	212	No	0	0	1	0
4	98	Yes	1	0	0	0
5	47	No	0	0	0	1

Pandas Categorical Columns

```
df = pd.DataFrame({
    'boro': ['Manhattan', 'Queens', 'Manhattan', 'Brooklyn', 'Brooklyn', 'Bronx'],
    'salary': [103, 89, 142, 54, 63, 219],
    'vegan': ['No', 'No','No','Yes', 'Yes', 'No']})

df['boro'] = pd.Categorical(df.boro,
                            categories=['Manhattan', 'Queens', 'Brooklyn',
                                        'Bronx', 'Staten Island'])
pd.get_dummies(df, columns=['boro'])
```

	salary	vegan	boro_Manhattan	boro_Queens	boro_Brooklyn	boro_Bronx	boro_Staten Island
0	103	No	1	0	0	0	0
1	89	No	0	1	0	0	0
2	142	No	1	0	0	0	0
3	54	Yes	0	0	1	0	0
4	63	Yes	0	0	1	0	0
5	219	No	0	0	0	1	0

OneHotEncoder

```
import pandas as pd
df = pd.DataFrame({'salary': [103, 89, 142, 54, 63, 219],
                   'boro': ['Manhattan', 'Queens', 'Manhattan',
                            'Brooklyn', 'Brooklyn', 'Bronx']})

ce = OneHotEncoder().fit(df)
ce.transform(df).toarray()
```

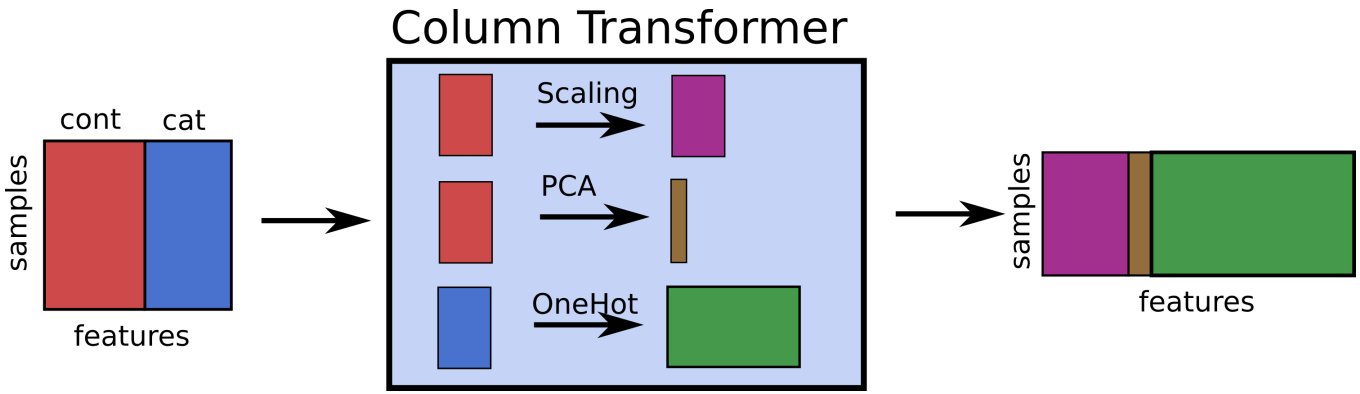
```
array([[ 0.,  0.,  1.,  0.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  1.,  0.],
       [ 0.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.]])
```

OneHotEncoder + ColumnTransformer

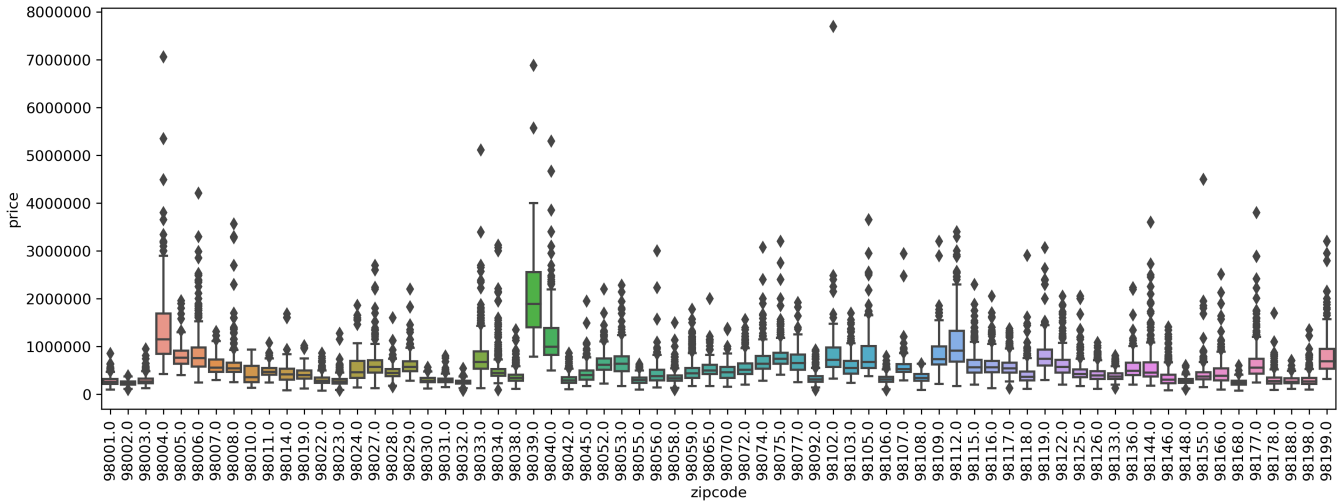
```
categorical = df.dtypes == object

preprocess = make_column_transformer(
    (StandardScaler(), ~categorical),
    (OneHotEncoder(), categorical))

model = make_pipeline(preprocess, LogisticRegression())
```



Target Encoding



- For high cardinality categorical features
- Instead of 70 one-hot variables, one "response encoded" variable.
- For regression- "average price in zip code"
- Binary classification– "building in this zip code have a likelihood p for class 1"
- Multiclass– One feature per class: probability distribution

Load Data, Include ZipCode

```
data = pd.read_csv("kc_house_data.csv", index_col=0)
X = data.frame.drop(['date', 'price'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, target)
X_train.columns
```

```
Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',
       'waterfront', 'view', 'condition', 'grade', 'sqft_above',
       'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat',
       'long', 'sqft_living15', 'sqft_lot15'],
      dtype='object')
```

Explore House Data Set

```
1 X_train.head()
```

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	...	zipcode	lat	long	sqft_living15	sqft_lot15
10666	4.0	2.50	2160.0	7000.0	2.0	...	98029.0	47.566	-122.013	2300.0	7440.0
19108	4.0	4.25	3250.0	11780.0	2.0	...	98004.0	47.632	-122.203	1800.0	9000.0
20132	3.0	2.50	1280.0	1920.0	3.0	...	98105.0	47.662	-122.324	1450.0	1900.0
16169	4.0	1.50	1220.0	9600.0	1.0	...	98014.0	47.646	-121.909	1180.0	9000.0
16890	3.0	1.50	2120.0	6290.0	1.0	...	98108.0	47.566	-122.318	1620.0	5400.0

```
1 te = TargetEncoder(cols='zipcode').fit(X_train, y_train)
2 te.transform(X_train).head()
```

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	...	zipcode	lat	long	sqft_living15	sqft_lot15
10666	4.0	2.50	2160.0	7000.0	2.0	...	6.164e5	47.566	-122.013	2300.0	7440.0
19108	4.0	4.25	3250.0	11780.0	2.0	...	1.357e7	47.632	-122.203	1800.0	9000.0
20132	3.0	2.50	1280.0	1920.0	3.0	...	8.503e7	47.662	-122.324	1450.0	1900.0
16169	4.0	1.50	1220.0	9600.0	1.0	...	4.464e8	47.646	-121.909	1180.0	9000.0
16890	3.0	1.50	2120.0	6290.0	1.0	...	3.604e9	47.566	-122.318	1620.0	5400.0

```
1 y_train.groupby(X_train.zipcode).mean()[X_train.head().zipcode])
```

zipcode	98029.0	98004.0	98105.0	98014.0	98108.0
price	616356.941	1.357e6	850306.816	446448.065	360416.811

Explore House Data Set

```
X = data.frame.drop(['date', 'price', 'zipcode'], axis=1)
scores = cross_val_score(Ridge(), X, target)
np.mean(scores)
0.69
```

```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import OneHotEncoder
X = data.frame.drop(['date', 'price'], axis=1)
ct = make_column_transformer((OneHotEncoder(), ['zipcode']), remainder='passthrough')
pipe_ohe = make_pipeline(ct, Ridge())
scores = cross_val_score(pipe_ohe, X, target)
np.mean(scores)
0.52
```

```
from category_encoders import TargetEncoder
X = data.frame.drop(['date', 'price'], axis=1)
pipe_target = make_pipeline(TargetEncoder(cols='zipcode'), Ridge())
scores = cross_val_score(pipe_target, X, target)
np.mean(scores)
0.78
```