Preprocessing

Chantri Polprasert

CPDSAI

Preprocessing

- Process raw feature vector into suitable form
- There are several common preprocessing methods
  - Standardization and scaling
  - Encoding categorical features
  - Discretization
  - Imputation
  - Pipeline
    - GridSearchCV
    - Column Transformer

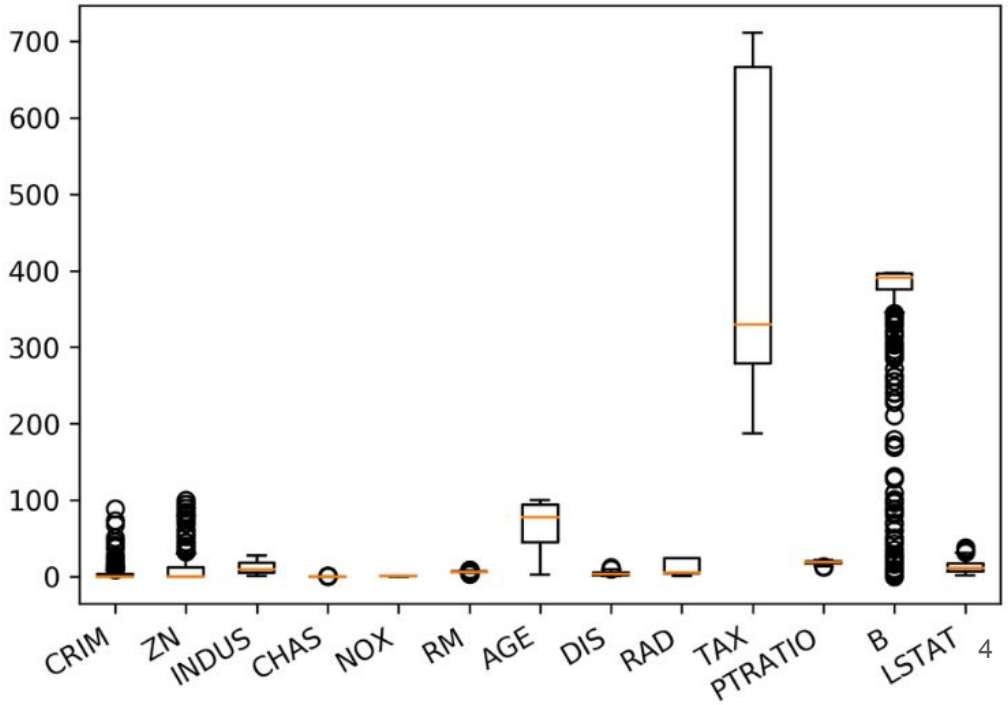
Standardization and scaling

Standardization and scaling

- Each feature comes in different scale e.g.
  - Tax ranges (300-700)
  - Age ranges (20-100)
- Difficulties to visualize the data
- Degrade the predictive performance of many machine learning algorithms
- Unscaled data can also slow down or even prevent the convergence of many gradient-based estimators.

```
plt.boxplot(X)
plt.xticks(np.arange(1, X.shape[1] + 1), boston.feature_names, rotation=30, ha="right")
```

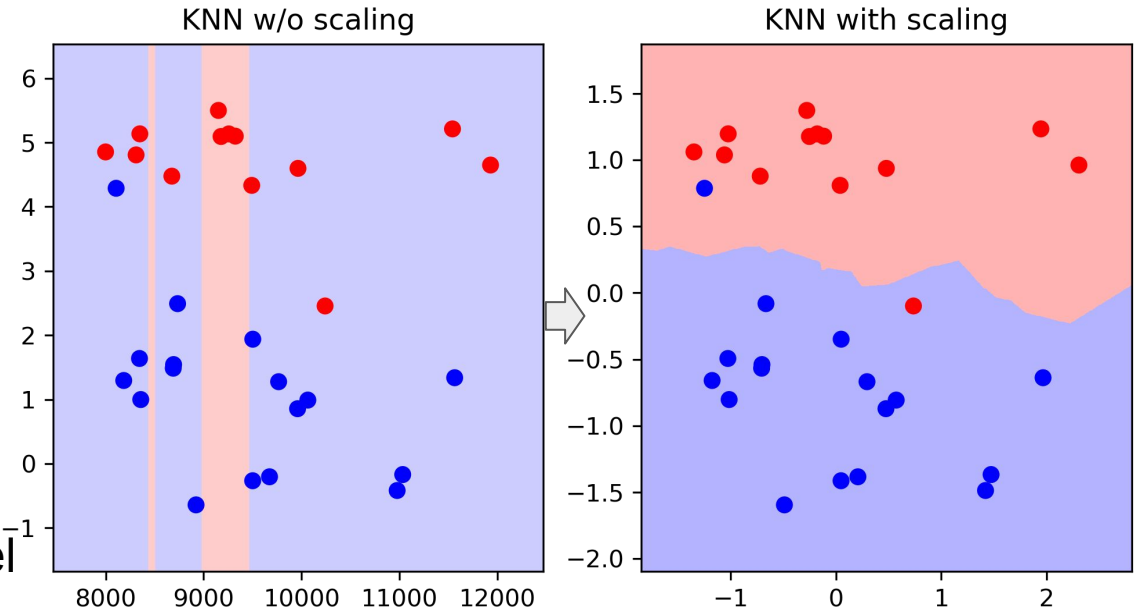
```
<matplotlib.text.Text at 0x7f580303eac8>
```



Aim of standardization and scaling

We want the values of all features to be in the same scale

- Benefits:
  - There are some models that need the features in the same scale; for example, the KNN model



  - In determining the importance of features in a linear model by comparing the coefficient of the features, we need the features to be in the same scale

Sklearn instructions

Same as modelling process

- Define the operation: specify the model
- Fit the data: standard scaler find mean and variance of X
- Transform the data: calculate the Z score of each feature

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()                          (Define the operation)
X_train_scaled = scaler.fit_transform(X_train)     (fit+transform)
```

StandardScaler in sklearn

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> scaler
StandardScaler()

>>> scaler.mean_
array([1. ..., 0. ..., 0.33...])

>>> scaler.scale_
array([0.81..., 0.81..., 1.24...])

>>> X_scaled = scaler.transform(X_train)
>>> X_scaled
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

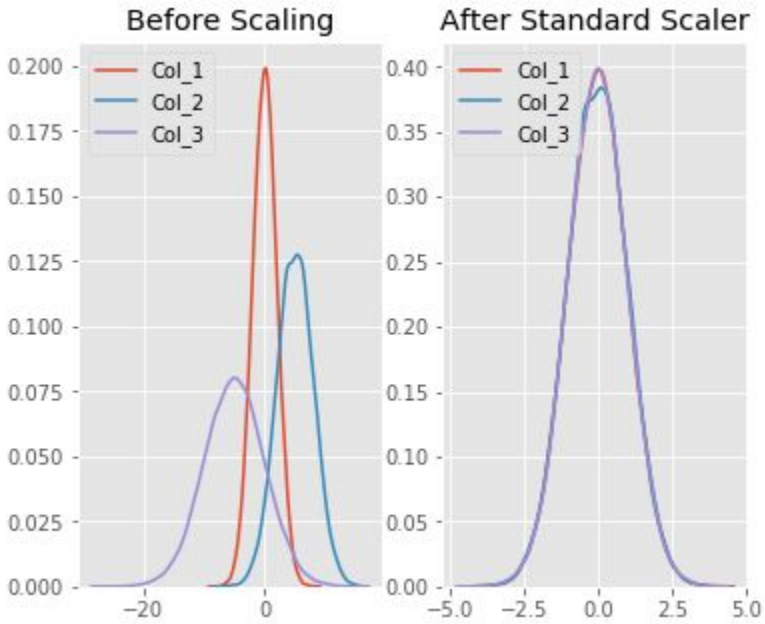
Standard scaler

calculating z-score: subtract mean, divided by standard deviation

$$z_f^{(i)} = \frac{x_f^{(i)} - \mu_f}{\sigma_f^2}$$, where $x^{(i)}_f$ is

the f[th] feature of the i[th] instance

- Usually good, but doesn't guarantee particular min and max values
- Can be influenced by outliers



https://michael-fuchs-python.netlify.app/2019/08/31/feature-scaling-with-scikit-learn/

California Housing dataset

Number of Instances: 20640

Number of Attributes: 8 numeric, predictive attributes and the target

Attribute Information:

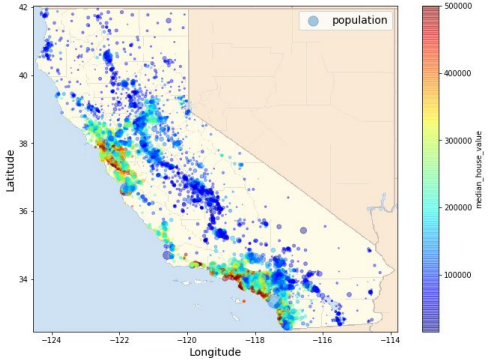
- MedInc median income in block group
- HouseAge median house age in block group
- AveRooms average number of rooms per household
- AveBedrms average number of bedrooms per household
- Population block group population
- AveOccup average number of household members
- Latitude block group latitude
- Longitude block group longitude

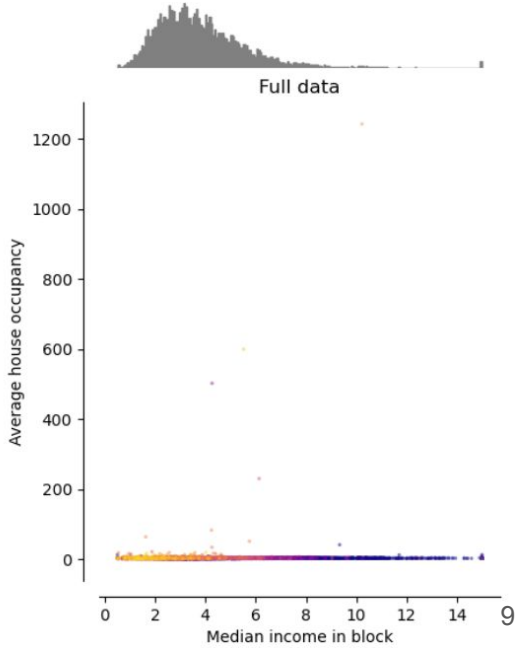
Target: median house value

- A large majority of the median income are compacted to a specific range, [0, 10]
- The average house occupancy is mostly in the range [0,6] with outliers more than 1,200

https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

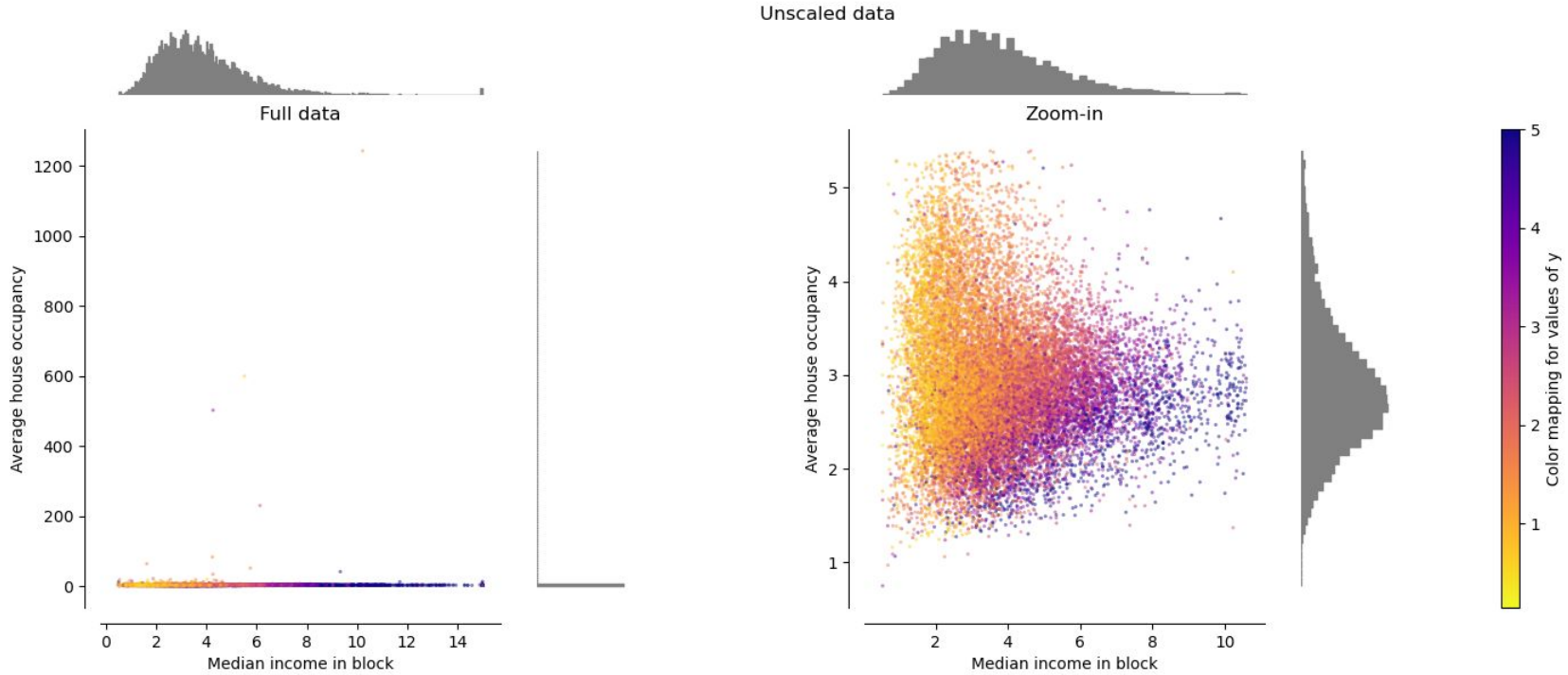
https://github.com/amansingh9097/California-housing-price-prediction?tab=readme-ov-file





StandardScaler

- Is it good?
- What's wrong with the standard scaler?



https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html

Other kinds of scaling

- Minmax scaling: keep the transform values between 0 and 1
  - Fit: find min and max values of each f feature $\hat{x}^{(i)} = \frac{x^{(i)} - \min_f}{\max_f - \min_f}$
  - Transform: calculate the normalized value
- Robust scaling: using median and quartile instead of mean and variance to reduce effect of outliers
  - Fit: find median and quartile of each f feature $z_f^{(i)} = \frac{x_f^{(i)} - m_f}{IQR}$
  - Transform: calculate the scaled value as in the z score
    - IQR: the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile); m_f: the median

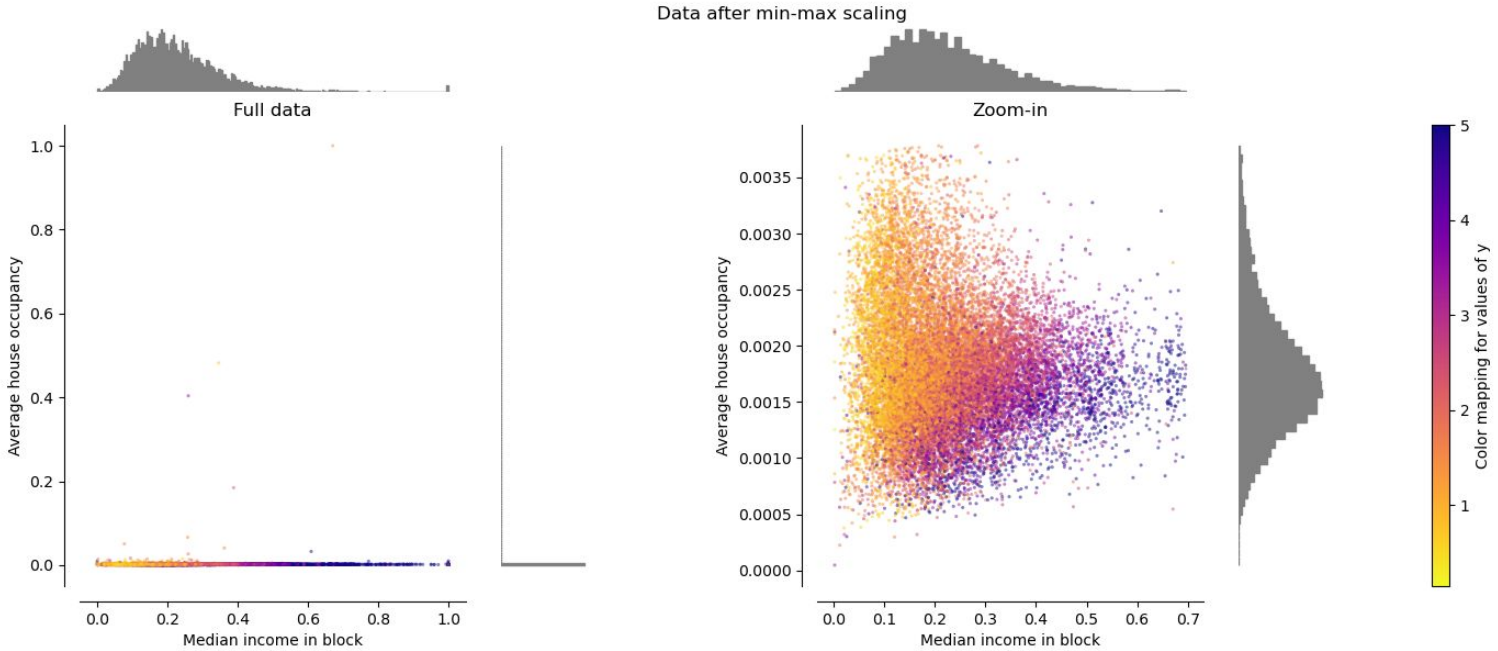
MinMaxScaler

```
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
...
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[0.5       , 0.        , 1.        ],
       [1.        , 0.5       , 0.33333333],
       [0.        , 1.        , 0.        ]])
```

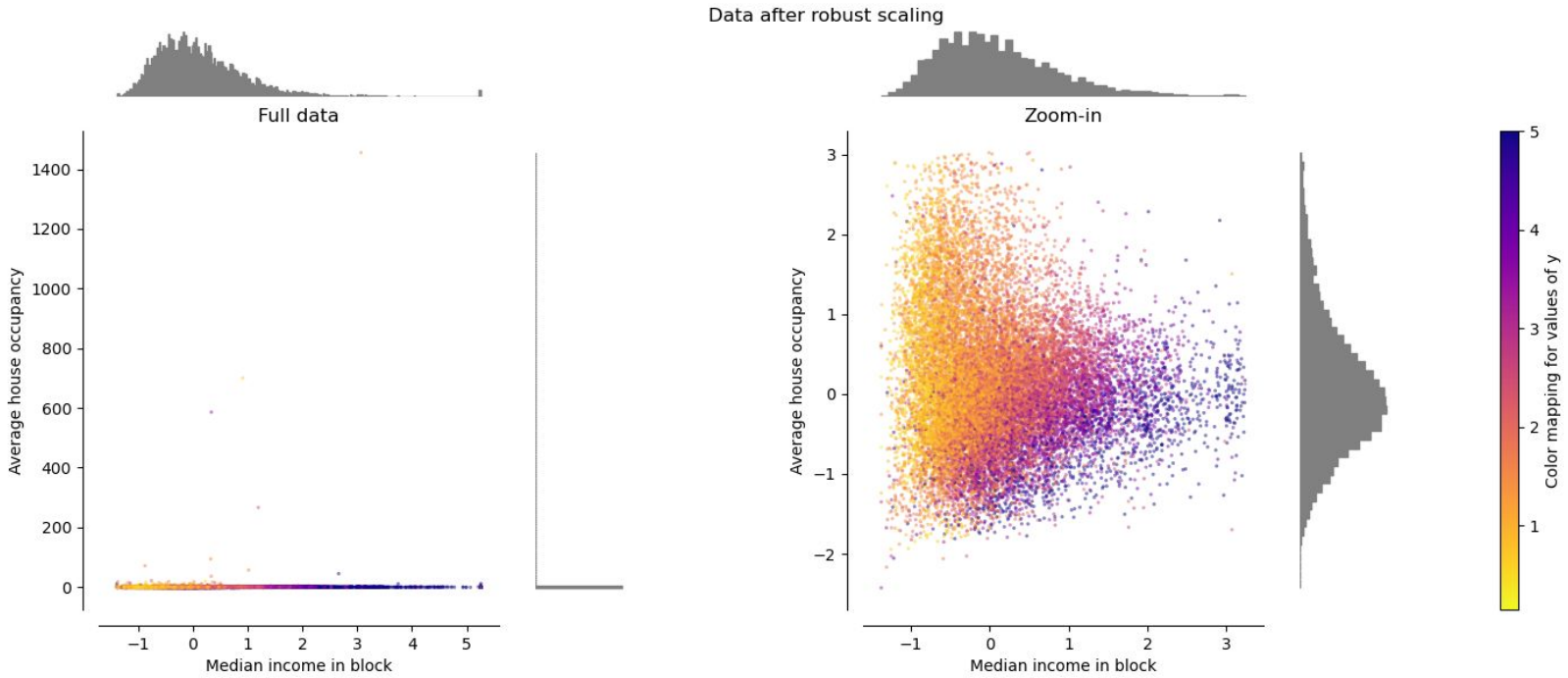
- The same instance of the transformer can then be applied to some new test data unseen during the fit call

```
>>> X_test = np.array([[-3., -1.,  4.]])
>>> X_test_minmax = min_max_scaler.transform(X_test)
>>> X_test_minmax
array([[-1.5       ,  0.        ,  1.66666667]])
```

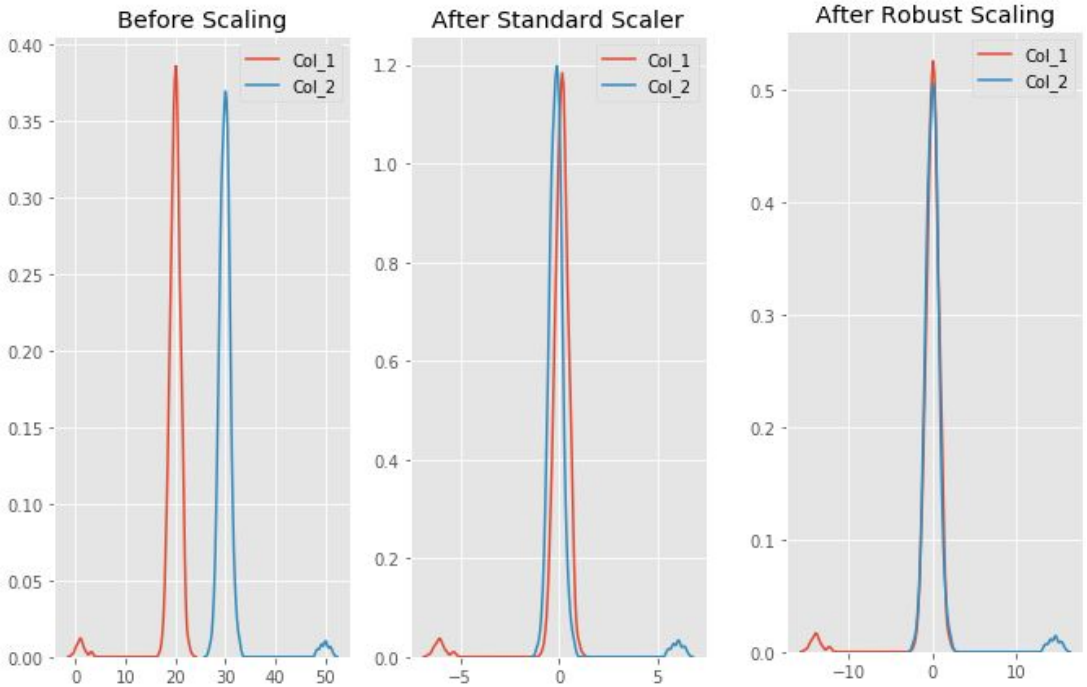
MinMaxScaler



RobustScaler



StandardScaler
vs RobustScaler



https://michael-fuchs-python.netlify.app/2019/08/31/feature-scaling-with-scikit-learn/

Data leakage during pre-processing

- What's wrong with this?

```
# Load your dataset
X, y = datasets.load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Initialize the scaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.fit_transform(X_test)
# Initialize the KNN classifier with a specified number of neighbors
knn = KNeighborsClassifier(n_neighbors=7)
#Prediction
y_pred = knn.fit(X_train_scaled, y_train).predict(X_test_scaled)
print(y_pred)
#Evaluate the model using accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

Encoding

Categorical encoding

- Mapping categorical values into numerical values
- Ordinal encoding: mapped to integer value directly
- One-hot encoding: mapped each value into each new column

Ordinal encoding

- Same as modelling process
  - Define the operation: specify the model
  - Fit the data: find all possible values in feature f
  - Transform the data: mapping integers to the values
- Let the model determine possible values from the training set
- Predefine the possible values (we must predefine all features)
- Features such as: ["male", "female"] , ["from Europe", "from US", "from Asia"] ["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]
- Such features can be efficiently coded as integers, for instance ["male", "from US", "uses Internet Explorer"] could be expressed as [0, 1, 3]
- A feature vector ["female", "from Asia", "uses Chrome"] can be represented as [1, 2, 1]

sklearn.preprocessing.OrdinalEncoder()

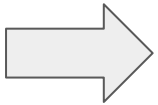
```
>>> enc = preprocessing.OrdinalEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OrdinalEncoder()
>>> enc.transform([['female', 'from US', 'uses Safari']])
array([[0., 1., 1.]])
```

Warning:

- ‘using Safari’ and ‘using Firefox’ have no order relationship. Suppose that we assign the value 0 and 1 to ‘using Safari’ and ‘using Firefox’ respectively, the model could interpret as they are close to each other.
- Encode ‘using Chrome’ as 2, some models would interpret that ‘using Chrome’ is closer to ‘using Safari than ‘using Firefox, because 2 is closer to 1 than 0.

One-hot Encoder

	boro	salary
0	Manhatten	103
1	Queens	89
2	Manhatten	142
3	Brooklyn	54
4	Brooklyn	63
5	Bronx	219



	salary	boro_Bronx	boro_Brooklyn	boro_Manhatten	boro_Queens
0	103	0.0	0.0	1.0	0.0
1	89	0.0	0.0	0.0	1.0
2	142	0.0	0.0	1.0	0.0
3	54	0.0	1.0	0.0	0.0
4	63	0.0	1.0	0.0	0.0
5	219	1.0	0.0	0.0	0.0

- Transform each categorical feature with n_categories possible values into n_categories binary features, with one of them 1, and all others 0.
- Pros: No false proximity issue as in ordinal encoding
- Cons: Could cause a lot more features, if the feature contains many categorical values

sklearn.preprocessing.OneHotEncoder()

```
>>> enc = preprocessing.OneHotEncoder()
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OneHotEncoder()
>>> enc.transform([['female', 'from US', 'uses Safari'],
...                ['male', 'from Europe', 'uses Safari']]).toarray()
array([[1., 0., 0., 1., 0., 1.],
       [0., 1., 1., 0., 0., 1.]])
```

Handle unknown value in OneHotEncoder()

What will happen?

```
enc = preprocessing.OneHotEncoder()
X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
enc.fit(X)

enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
```

How to solve this?

Handle unknown value in OneHotEncoder()

What will happen?

```
>>> enc = preprocessing.OneHotEncoder(handle_unknown='infrequent_if_exist')
>>> X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
>>> enc.fit(X)
OneHotEncoder(handle_unknown='infrequent_if_exist')
>>> enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
array([[1., 0., 0., 0., 0., 0.]])
```

With handle_unknown='infrequent_if_exist' specified and unknown categories are encountered during transform, no error will be raised but the resulting one-hot encoded columns for this feature will be all zeros or considered as an infrequent category if enabled

Missing values as an additional category (only in sklearn Version 1.0)

```
X = [['male', 'Safari'],['female',None],[np.nan, 'Firefox']]
enc = preprocessing.OneHotEncoder(handle_unknown='error').fit(X)
enc.categories_
```

```
[array(['female', 'male', nan], dtype=object),
 array(['Firefox', 'Safari', None], dtype=object)]
```

```
enc.transform(X).toarray()
```

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

One-hot encoding: sparse representation

Normally, most entries from one-hot encoding would be zeros

Sklearn uses sparse representation to encode only entries with value 1 to save resources.

If we want to see the actual data we need to convert it to ordinary array, using toarray( )

```
one hot mapped value:
   (0, 1)        1.0
  (0, 2)         1.0
  (0, 4)         1.0
  (1, 0)         1.0
  (1, 2)         1.0
  (1, 3)         1.0
```

```
print(enc.transform([['male', 'from America', 'uses Safari']]).toarray())
```

```
[[0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 1.]]
```

Discretization

Discretization

- Quantization or binning
- partitioning continuous features into discrete values
- In sklearn, use **k-bin discretizer**: dividing each feature into k bins and assigning the values of the feature into corresponding bins.
- `sklearn.preprocessing.KBinsDiscretizer(n_bins=5, *, encode='onehot', strategy='quantile', dtype=None)`

sklearn.preprocessing.KBinsDiscretizer

```
sklearn.preprocessing.KBinsDiscretizer(n_bins=5, *,
encode='onehot', strategy='quantile', dtype=None)
```

encode : *{'onehot', 'onehot-dense', 'ordinal'}, default='onehot'*

Method used to encode the transformed result.

onehot

Encode the transformed result with one-hot encoding and return a sparse matrix. Ignored features are always stacked to the right.

onehot-dense

Encode the transformed result with one-hot encoding and return a dense array. Ignored features are always stacked to the right.

ordinal

Return the bin identifier encoded as an integer value.

sklearn.preprocessing.KBinsDiscretizer

```
sklearn.preprocessing.KBinsDiscretizer(n_bins=5, *,
encode='onehot', strategy='quantile', dtype=None)
```

strategy : *{'uniform', 'quantile', 'kmeans'}, default='quantile'*

Strategy used to define the widths of the bins.

uniform

All bins in each feature have identical widths.

quantile

All bins in each feature have the same number of points.

kmeans

Values in each bin have the same nearest center of a 1D k-means cluster.

sklearn.preprocessing.Binarizer()

- Feature Binarization
- Setting the threshold and divide the value into two groups
- This method is equivalent to KBinsDiscretizer when k=2.

```
class sklearn.preprocessing.Binarizer(*, threshold=0.0, copy=True)
```

- Feature values below or equal to this are replaced by 0, above it by 1. Threshold may not be less than 0 for operations on sparse matrices.

sklearn.preprocessing.Binarizer()

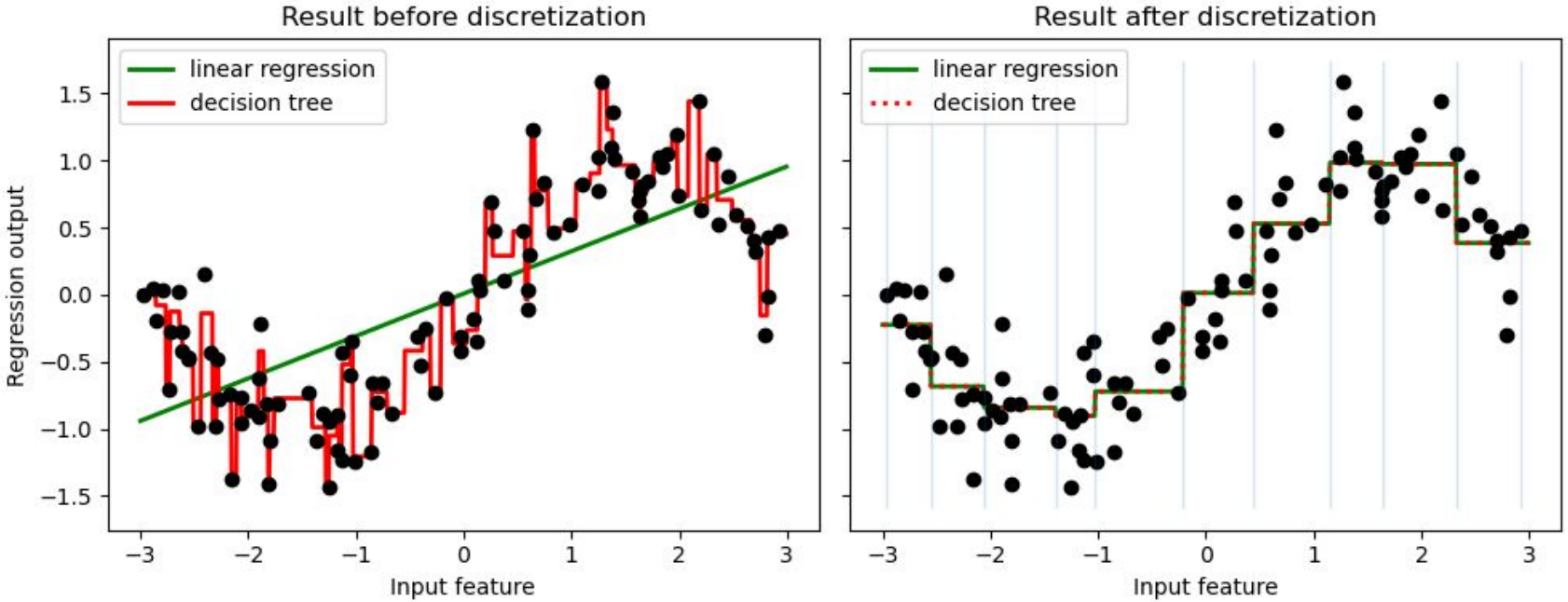
```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]

>>> binarizer = preprocessing.Binarizer().fit(X)  # fit does nothing
>>> binarizer
Binarizer()

>>> binarizer.transform(X)
array([[1., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.]])
```

Anything less than or equal to zero is designated ‘0’

```
>>> binarizer = preprocessing.Binarizer(threshold=1.1)
>>> binarizer.transform(X)
array([[0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 0.]])
```

Instead of using the threshold value at 0, we can set the value freely using threshold argument.

Benefits of discretization: Using KBinsDiscretizer to discretize continuous features



- After discretization, linear regression and decision tree make exactly the same prediction.

Transformation summary

Feature type	Transformation
Continuous: numerical values	● **Standard Scaling**, **Min-Max Scaling** ● **Discretization**
Nominal: categorical, unordered features (*True* or *False*)	● **One-hot encoding** (0, 1)
Ordinal: categorical, ordered features (movie ratings)	● **Ordinal encoding** (0, 1, 2, 3)

Data imputation

Univariate feature imputation

- Univariate feature imputation by using the statistics, such as mean or median, of the missing feature alone.
- `sklearn.impute.SimpleImputer(*, missing_values=nan, strategy='mean', fill_value=None, verbose=0, copy=True, add_indicator=False)`

```
>>> import numpy as np
>>> from sklearn.impute import SimpleImputer
>>> imp = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
SimpleImputer()
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[4.          2.        ]
 [6.          3.666...]
 [7.          6.        ]]
```

Strategy can be mean, mode, median or constant

Multivariate feature imputation

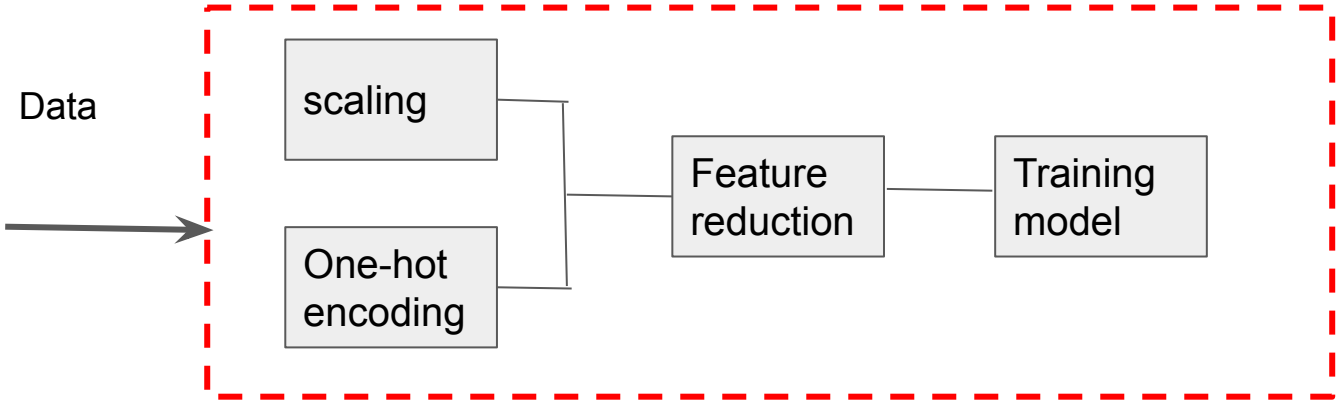
- Imputes missing values by considering the values of other feature together with the missing feature.
- Operation:
  - y: feature with missing values treated as target vector
  - X: the other feature columns treated as feature vector
- A regressor is fit on (X, y) for known y. Then, the regressor is used to predict the missing values of y.

```
>>> import numpy as np
>>> from sklearn.experimental import enable_iterative_imputer
>>> from sklearn.impute import IterativeImputer
>>> imp_mean = IterativeImputer(random_state=0)
>>> imp_mean.fit([[7, 2, 3], [4, np.nan, 6], [10, 5, 9]])
IterativeImputer(random_state=0)
>>> X = [[np.nan, 2, 3], [4, np.nan, 6], [10, np.nan, 9]]
>>> imp_mean.transform(X)
array([[ 6.9584...,  2.       ,  3.        ],
       [ 4.       ,  2.6000...,  6.        ],
       [10.       ,  4.9999...,  9.        ]])
```

Data pipelining

Pipeline

- Pipelining is the operation that allows us to chain multiple operations into a single instruction
- Very beneficial for streaming data

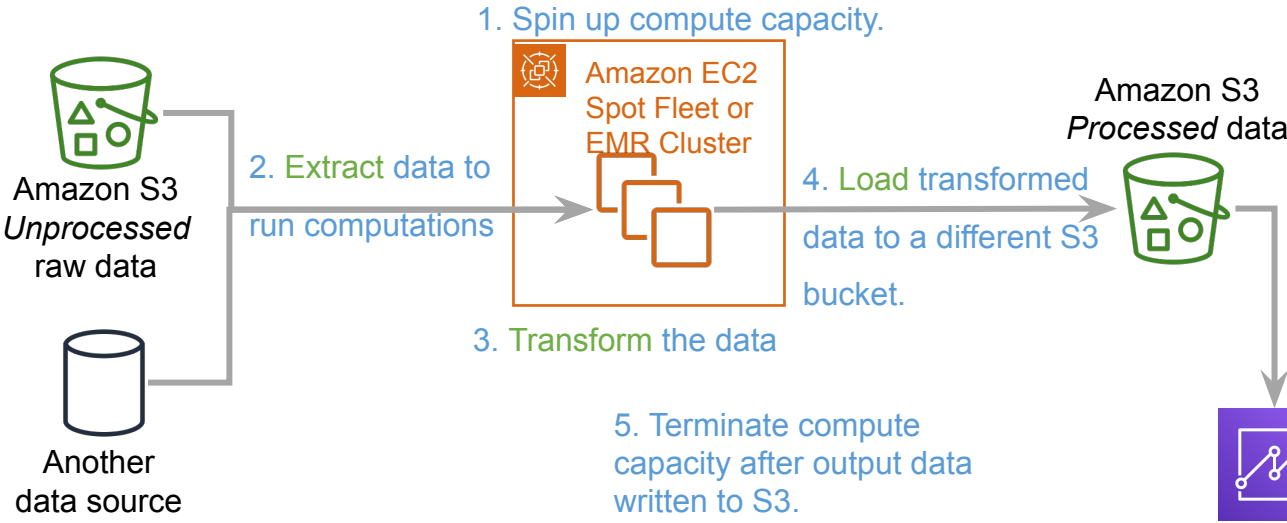


Pipeline of the data

Amazon S3 use case 3: Data store for computation and analytics

Data store for computation and large-scale analytics

Example data integration and preparation pattern



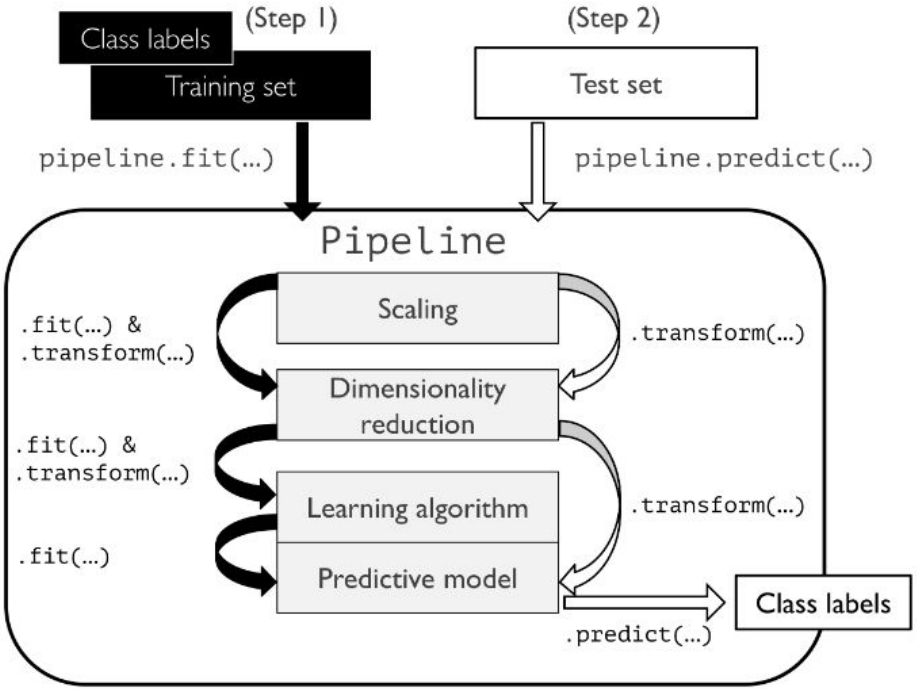
Financial transaction analysis

Clickstream analytics

Media transcoding

sklearn.pipeline.make_pipeline

- Construct a Pipeline from the given estimators.
- Sequentially apply a list of transforms and a final estimator.
- Intermediate steps of the pipeline must be ‘transforms’, that is, they must implement fit and transform methods.
- The final estimator only needs to implement fit.



https://vitalflux.com/wp-content/uploads/2022/09/Machine-learning-pipeline-Sklearn.png

sklearn.pipeline.make_pipeline

```
from sklearn.linear_model import Ridge
X, y = boston.data, boston.target
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)
```

```
0.634
```

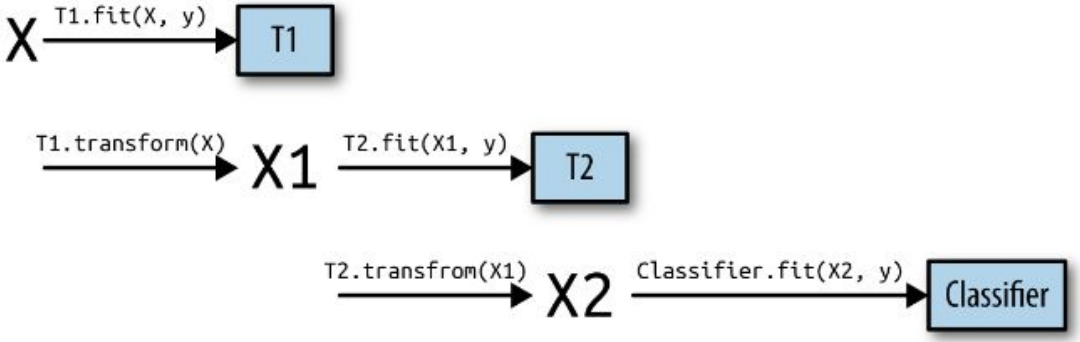
```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(StandardScaler(), Ridge())
pipe.fit(X_train, y_train)
pipe.score(X_test, y_test)
```

```
0.634
```

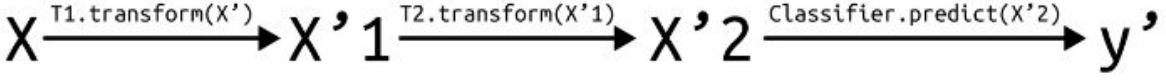
sklearn.pipeline.Pipeline

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


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




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



sklearn.pipeline.Pipeline

- Must assign the name of each process whereas make_pipeline assign the name automatically.
- The steps in this command is a list of (key, value) pairs, where the key is a string containing the name you want to give for this step and value is an estimator object

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
X, y = make_classification(random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state=0)
pipe = Pipeline([('scaler', StandardScaler()), ('knn3', KNeighborsClassifier(n_neighbors=3))])
# The pipeline can be used as any other estimator
# and avoids leaking the test set into the train set
pipe.fit(X_train, y_train).score(X_test, y_test)
```

```
0.72
```

sklearn.pipeline.Pipeline

- An estimator's parameter can be set using '__' syntax (double underscore)
- `pipe.set_params(knn3__n_neighbors=5).fit(X_train, y_train).score(X_test, y_test)`
- can access each step by indexing the list of the pipeline
- What is the output of pipe[0]?

```
pipe.steps[1]
```

```
('knn3', KNeighborsClassifier())
```

```
pipe[1]
```

```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

```
pipe['knn3']
```

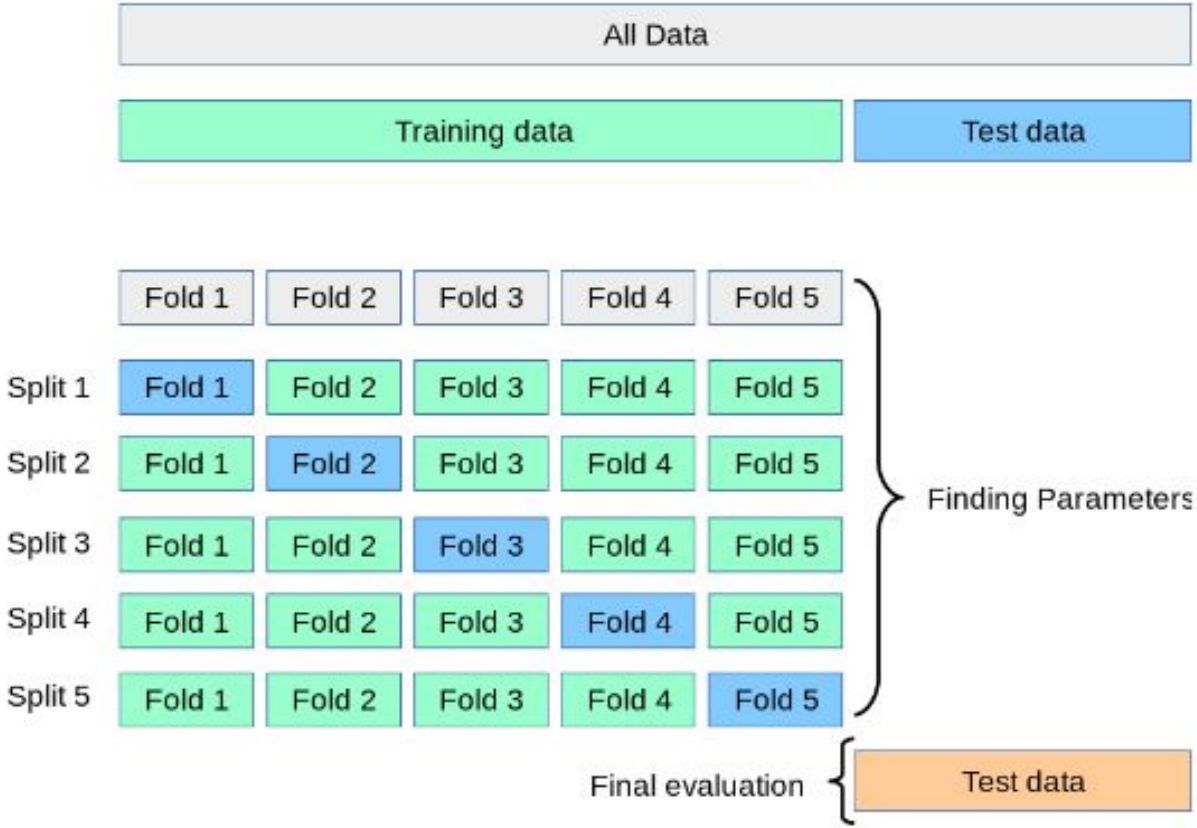
```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

Data leakage

- Scaling the whole dataset could cause information leakage
- We must split the dataset into training and test datasets, then, we fit training set and use the fitted model to scale the test set
- In case performing cross validation, we cannot split training set and validation set

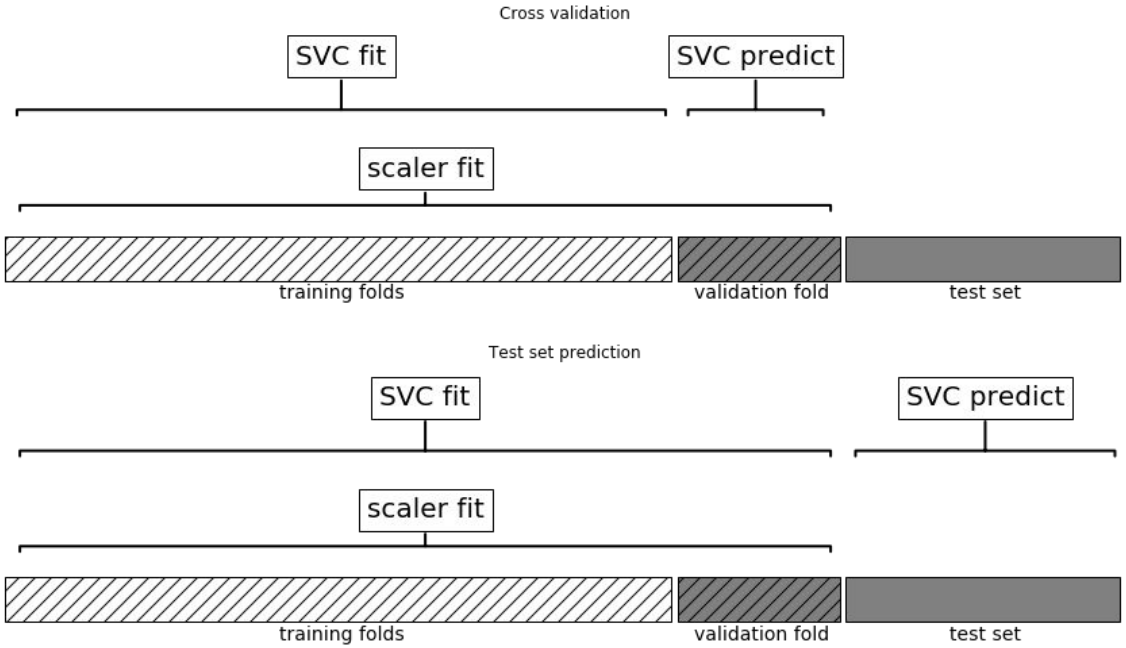
We have to use data pipeline

N-fold Cross-validation revisit

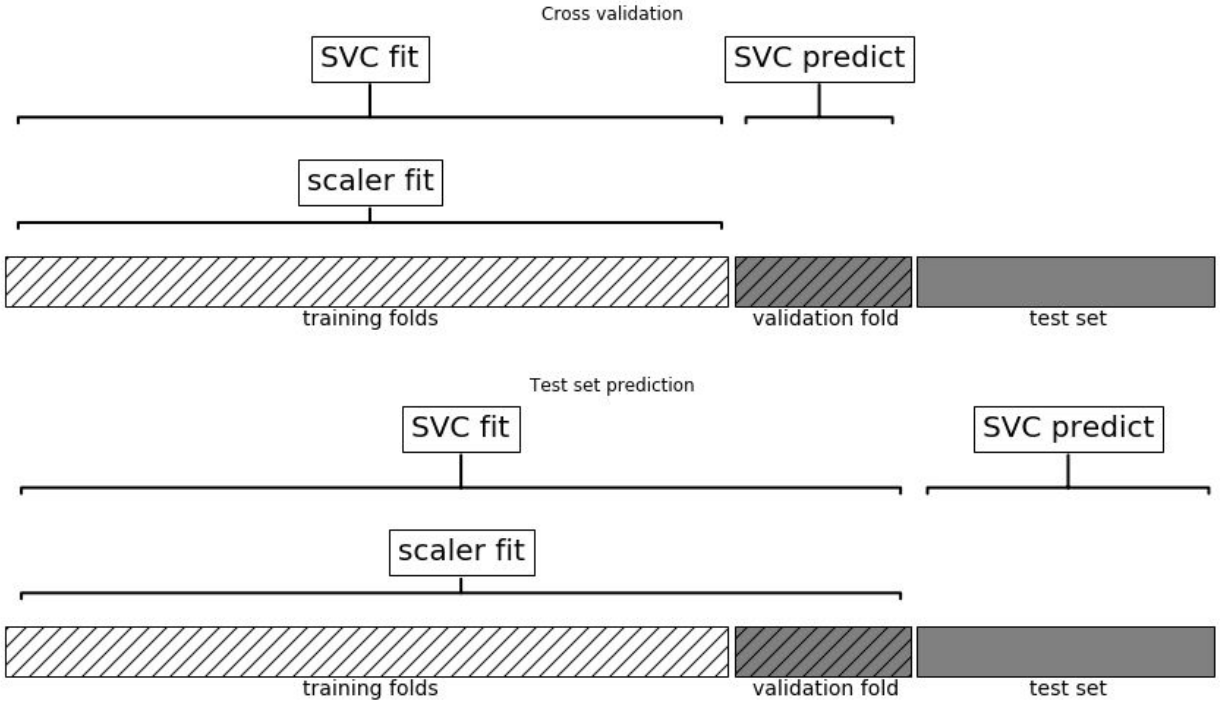


Information leakage

When we perform scaling, it must be ensured that the validation dataset is excluded from scaling process. Otherwise the information leakage could occur.



No information leakage



.fit(Train_fold)
.transform(Train_fold)
.transform(Valid_fold)

How to implement in CV?

Example of Information leakage

Where does the leakage occur?

```
#information leakage
b_cancer=load_breast_cancer()
X, y = b_cancer.data, b_cancer.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
est = LogisticRegression(max_iter=400)
#Scaling
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score
scale = StandardScaler()
X_train_scaled=scale.fit_transform(X_train)
scores = cross_val_score(est, X_train_scaled, y_train, cv=5)
print(f'{scores.mean():.4f}')
```

Use pipeline to prevent information leakage

- Earlier, the whole training set (train fold + validate fold) is used for scaling -> Leakage

Best Practice

```
#No information leakage
#Scaling
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(StandardScaler(),est)
scores = cross_val_score(pipe, X_train, y_train, cv=5)
```

Use pipeline to prevent information leakage

- For KNN

Best Practice

```
from sklearn.neighbors import KNeighborsRegressor
knn_pipe = make_pipeline(StandardScaler(), KNeighborsRegressor())
scores = cross_val_score(knn_pipe, X_train, y_train, cv=10)
np.mean(scores), np.std(scores)
```

```
(0.745, 0.106)
```

Grid-Search with Cross-Validation

```
from sklearn.model_selection import cross_val_score

X_train, X_test, y_train, y_test = train_test_split(X, y)

cross_val_scores = []

for i in neighbors:
    knn = KNeighborsClassifier(n_neighbors=i)
    scores = cross_val_score(knn, X_train, y_train, cv=10)
    cross_val_scores.append(np.mean(scores))

print("best cross-validation score: {:.3f}".format(np.max(cross_val_scores)))
best_n_neighbors = neighbors[np.argmax(cross_val_scores)]
print("best n_neighbors: {}".format(best_n_neighbors))

knn = KNeighborsClassifier(n_neighbors=best_n_neighbors)
knn.fit(X_train, y_train)
print("test-set score: {:.3f}".format(knn.score(X_test, y_test)))
```

```
best cross-validation score: 0.969
best n_neighbors: 9
test-set score: 0.944
```

GridSearchCV

- Called Meta-estimator
- Can specify the estimator, parameters
- Retrain the model with the best parameter settings.

Grid search is the process of performing hyper parameter tuning in order to determine the optimal values for a given model.

```
from sklearn.model_selection import GridSearchCV

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y)


param_grid = {'n_neighbors':  np.arange(1, 30, 2)}
grid = GridSearchCV(KNeighborsClassifier(), param_grid=param_grid, cv=10, return_train_score=True)
grid.fit(X_train, y_train)
print("best mean cross-validation score: {:.3f}".format(grid.best_score_))
print("best parameters: {}".format(grid.best_params_))

print("test-set score: {:.3f}".format(grid.score(X_test, y_test)))
```

```
best mean cross-validation score: 0.972
best parameters: {'n_neighbors': 3}
test-set score: 0.958
```

Naming steps

```
from sklearn.neighbors import KNeighborsRegressor
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

```
from sklearn.model_selection import GridSearchCV
knn_pipe = make_pipeline(StandardScaler(), KNeighborsRegressor())
param_grid = {'kneighborsregressor__n_neighbors':range(1, 10, 2)}
grid = GridSearchCV(knn_pipe, param_grid, cv = 10)
grid.fit(X_train, y_train)
print(grid.best_params_)
print(grid.score(X_test, y_test))
```

```
{'kneighborsregressor__n_neighbors': 7}
0.8409275659949514
```

- Use Pipeline inside GridSearchCV
- Use keys as stepname__parametername since multiple steps could have parameters with same name

Column transformer

- Different columns need different preprocessing treatments e.g. some columns need one-hot processing, while some need standardization.
- Cannot apply OneHotEncoder() or StandardScaler() directly because all columns would receive these treatments together.
- In column transformer, we can specify different columns for different treatments
- If we want more than two or more stages process, we might have to use pipeline together with columns transformer

Column transformer

```
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression())])
```

- Apply Median Imputer and Standard Scaler for numeric features (age, fare)
- Apply One-hot Encoder for categorical features (embarked, sex, pclass)
- Pipeline can simplify this implementation

Summary

- Pipeline: to specify all ML steps with one command e.g. scaling -> define model -> cross validation
- GridSearchCV is used to find hyper-parameters using brute-force approaches. Set parameters by dictionary: {‘parametername’:search range}
- With make_pipeline or Pipeline, we can reference every step used in GridSearchCV in Pipeline
- GridSearchCV inside pipeline: set keys as stepname__parametername

References

- Andreas C. Müller and Sarah Guido. Introduction to Machine
- Learning with Python: A Guide for Data Scientists. O'Reilly Media; 1st edition.
- Andreas C. Müller, COMS W4995 Applied Machine Learning, Columbia University, Spring 2019.
- https://scikit-learn.org/stable/auto_examples/preprocessing/index.html
- https://scikit-learn.org/1.5/modules/generated/sklearn.pipeline.Pipeline.html