CPSC429/529: Machine Learning

Lecture 4: Data Preparation, Transformation and Pipeline

Dongsheng Che
Computer Science Department
East Stroudsburg University

`sklearn.preprocessing`: Preprocessing and Normalization ¶

The `sklearn.preprocessing` module includes scaling, centering, normalization, binarization methods.

User guide: See the Preprocessing data section for further details.

`preprocessing.Binarizer`(*[, threshold, copy])	Binarize data (set feature values to 0 or 1) according to a threshold.
`preprocessing.FunctionTransformer`([func, ...])	Constructs a transformer from an arbitrary callable.
`preprocessing.KBinsDiscretizer`([n_bins, ...])	Bin continuous data into intervals.
`preprocessing.KernelCenterer`()	Center an arbitrary kernel matrix K.
`preprocessing.LabelBinarizer`(*[, neg_label, ...])	Binarize labels in a one-vs-all fashion.
`preprocessing.LabelEncoder`()	Encode target labels with value between 0 and n_classes-1.
`preprocessing.MultiLabelBinarizer`(*[, ...])	Transform between iterable of iterables and a multilabel format.
`preprocessing.MaxAbsScaler`(*[, copy])	Scale each feature by its maximum absolute value.
`preprocessing.MinMaxScaler`([feature_range, ...])	Transform features by scaling each feature to a given range.
`preprocessing.Normalizer`([norm, copy])	Normalize samples individually to unit norm.
`preprocessing.OneHotEncoder`(*[, categories, ...])	Encode categorical features as a one-hot numeric array.
`preprocessing.OrdinalEncoder`(*[, ...])	Encode categorical features as an integer array.
`preprocessing.PolynomialFeatures`([degree, ...])	Generate polynomial and interaction features.
`preprocessing.PowerTransformer`([method, ...])	Apply a power transform featurewise to make data more Gaussian-like.
`preprocessing.QuantileTransformer`(*[, ...])	Transform features using quantiles information.
`preprocessing.RobustScaler`(*[, ...])	Scale features using statistics that are robust to outliers.
`preprocessing.SplineTransformer`([n_knots, ...])	Generate univariate B-spline bases for features.
`preprocessing.StandardScaler`(*[, copy, ...])	Standardize features by removing the mean and scaling to unit variance.

Data Preparation

Standardization

Why Scaling Features?

- Models that rely on the distance between a pair of samples, for instance k-nearest neighbors, should be trained on normalized features to make **each feature contribute approximately equally to the distance computations**.

- Many models such as logistic regression use a numerical solver (based on gradient descent) to find their optimal parameters. This solver **converges faster when the features are scaled**.

The result of normalising a small sample of the HEIGHT and SPONSORSHIP EARNINGS features from the professional basketball squad dataset.

	HEIGHT			SPONSORSHIP EARNINGS		
	Values	Range	Standard	Values	Range	Standard
	192	0.500	-0.073	561	0.315	-0.649
	197	0.679	0.533	1,312	0.776	0.762
	192	0.500	-0.073	1,359	0.804	0.850
	182	0.143	-1.283	1,678	1.000	1.449
	206	1.000	1.622	314	0.164	-1.114
	192	0.500	-0.073	427	0.233	-0.901
	190	0.429	-0.315	1,179	0.694	0.512
	178	0.000	-1.767	1,078	0.632	0.322
	196	0.643	0.412	47	0.000	-1.615
	201	0.821	1.017	1111	0.652	0.384
Max	206			1,678		
Min	178			47		
Mean	193			907		
Std Dev	8.26			532.18		

range normalization

- We use **range normalization** to convert a feature value into the range $[low, high]$ as follows:

$$a_i' = \frac{a_i - min(a)}{max(a) - min(a)} \times (high - low) + low \qquad (5)$$

Example:
range [0,1]

a'= (190-178)/(206-178)*(1-0)+0=0.429

Using MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, 1))
rescaledX = scaler.fit_transform(X)
```

```
[[0.5  ]
 [0.679]
 [0.5  ]
 [0.143]
 [1.   ]]
```

```
scaler = MinMaxScaler(feature_range=(-1, 1))
rescaledX = scaler.fit_transform(X)
```

```
[[ 0.   ]
 [ 0.357]
 [ 0.   ]
 [-0.714]
 [ 1.   ]]
```

standardization

- Another way to normalize data is to **standardize** it into **standard scores**.
- A standard score measures how many standard deviations a feature value is from the mean for that feature.
- We calculate a standard score as follows:

$$a_i' = \frac{a_i - \overline{a}}{sd(a)} \qquad (6)$$

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

Example:

a'= (190-193)/8.26 = -0.315

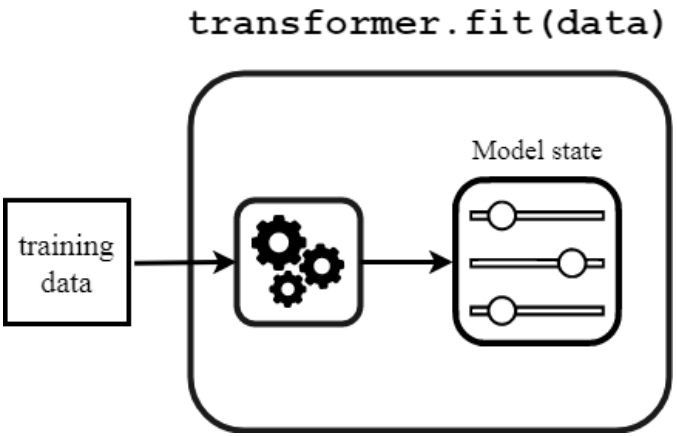
Using StandardScaler

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(X)
rescaledX = scaler.transform(X)
```

```
[[-0.077]
 [ 0.561]
 [-0.077]
 [-1.352]
 [ 1.71 ]]
```

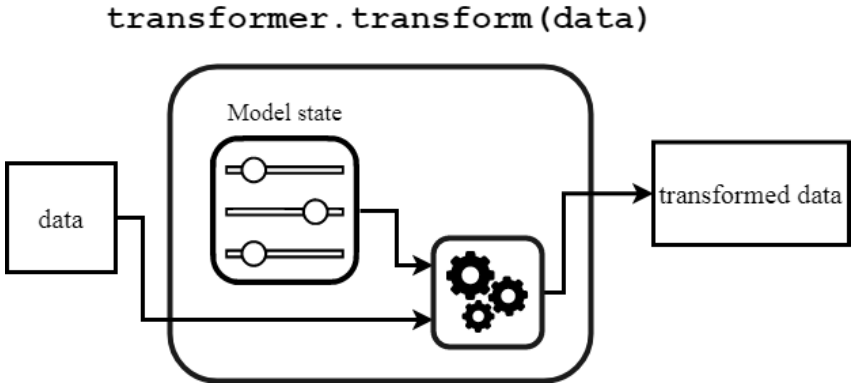
Key methods: fit()

- The fit method for this transformers is similar to the fit method for predictors. The main difference is that this one has a single argument (the data matrix), whereas the predictor has two arguments (the data matrix and the target).



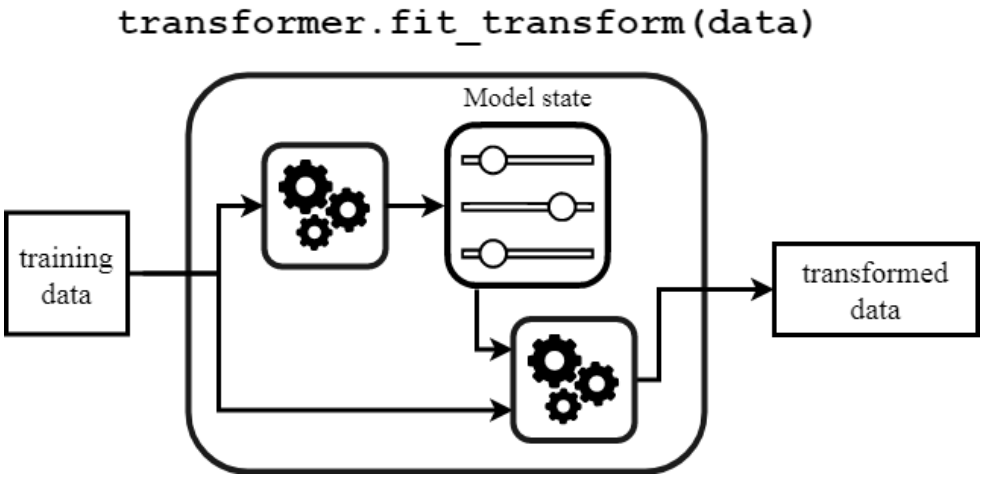
Key methods: transform()

- The transform method for transformers is similar to the predict method for predictors.
- It uses a predefined function, called a transformation function, and uses the model states and the input data to output a transformed version of the input data.



Key methods: fit_transform()

- The method fit_transform is a shorthand method to call successively fit and then transform.



Data Preparation

Standardization

Data Preparation

Discretization

Discretization

- Data discretization is a process of converting continues numerical data into discrete bins.
- Three types of Data discretization methods –
  1. **Quantile Transformation**: Each bin has an equal number of values based on the percentiles.
  2. **Uniform Transformation**: Each bin has equal or the same width with the possible values in the attribute.
  3. **Kmeans Transformation**: clusters are defined and values are assigned to them.

Quantile Transformation

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

In [6]:
```
#Import the class
from sklearn.preprocessing import KBinsDiscretizer

#Discrete the data
transf = KBinsDiscretizer(n_bins = 3, encode = 'ordinal', strategy = 'quantile')

#fit transform
X_q = transf.fit_transform(X)
print("Original: ", X.astype(int).tolist())
print("Quantile: ", X_q.astype(int).tolist())
```

```
Original:  [[192], [197], [192], [182], [206], [192], [190], [178], [196], [201]]
Quantile:  [[1], [2], [1], [0], [2], [1], [0], [0], [2], [2]]
```

Uniform Transformation

‘uniform’: All bins in each feature have identical widths.

In [7]:
```
#Import the class
from sklearn.preprocessing import KBinsDiscretizer

#Discrete the data
transf = KBinsDiscretizer(n_bins = 3, encode = 'ordinal', strategy = 'uniform')

#fit transform
X_u = transf.fit_transform(X)
print("Original: ", X.astype(int).tolist())
print("Quantile: ", X_q.astype(int).tolist())
print("Uniform:  ", X_u.astype(int).tolist())
```

```
Original:  [[192], [197], [192], [182], [206], [192], [190], [178], [196], [201]]
Quantile:  [[1], [2], [1], [0], [2], [1], [0], [0], [2], [2]]
Uniform:   [[1], [2], [1], [0], [2], [1], [1], [0], [1], [2]]
```

Kmeans Transformation

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

In [8]:
```
#Import the class
from sklearn.preprocessing import KBinsDiscretizer

#Discrete the data
transf = KBinsDiscretizer(n_bins = 3, encode = 'ordinal', strategy = 'kmeans')

#fit transform
X_kmeans = transf.fit_transform(X)
print("Original: ", X.astype(int).tolist())
print("Quantile: ", X_q.astype(int).tolist())
print("Uniform:  ", X_u.astype(int).tolist())
print("Kmeans:   ",  X_kmeans.astype(int).tolist())
```

```
Original:  [[192], [197], [192], [182], [206], [192], [190], [178], [196], [201]]
Quantile:  [[1], [2], [1], [0], [2], [1], [0], [0], [2], [2]]
Uniform:   [[1], [2], [1], [0], [2], [1], [1], [0], [1], [2]]
Kmeans:    [[1], [2], [1], [0], [2], [1], [1], [0], [1], [2]]
```

Data Preparation

Discretization

Data Preparation

Encoding categorical features

Why categorical data encoding?

- Machine learning models require all input and output variables to be numeric. This means that if your data contains categorical data, you must encode it to numbers before you can fit and evaluate a model. ... Encoding is a required pre-processing step when working with categorical data for machine learning algorithms.

Oridinal (Label) Encoding

- In Ordinal (label encoding, each distinct value of the feature is assigned numeric values starting from 0 to N-1 where N is the total number of distinct values.

Label	Encoded Label
Africa	1
Asia	2
Europe	3
South America	4
North America	5
Other	6

https://machinelearningknowledge.ai/categorical-data-encoding-with-sklearn-labelencoder-and-onehotencoder/

Dataset

In [105]:
```
#importing the necassary libraries
import pandas as pd
import numpy as np
from sklearn.preprocessing import OrdinalEncoder,OneHotEncoder
#reading the dataset
df=pd.read_csv("50_Startups.csv")
df.head()
```

Out[105]:

	R&D Spend	Administration	Marketing Spend	State	Profit
0	165349.20	136897.80	471784.10	New York	192261.83
1	162597.70	151377.59	443898.53	California	191792.06
2	153441.51	101145.55	407934.54	Florida	191050.39
3	144372.41	118671.85	383199.62	New York	182901.99
4	142107.34	91391.77	366168.42	Florida	166187.94

OrdinalEncoder Code

In [108]:
```
# Ordinal_encoder object
ordinal_encoder =OrdinalEncoder()

# Encode labels in column.
df['State_Label'] = ordinal_encoder.fit_transform(df[['State']])
df.head()
```

Out[108]:

	R&D Spend	Administration	Marketing Spend	State	Profit	State_Label
0	165349.20	136897.80	471784.10	New York	192261.83	2.0
1	162597.70	151377.59	443898.53	California	191792.06	0.0
2	153441.51	101145.55	407934.54	Florida	191050.39	1.0
3	144372.41	118671.85	383199.62	New York	182901.99	2.0
4	142107.34	91391.77	366168.42	Florida	166187.94	1.0

Common Error !!

```
# Ordinal_encoder object
ordinal_encoder =OrdinalEncoder()

# Encode labels in column.
df['State_Label'] = ordinal_encoder.fit_transform(df['State'])
df.head()
```

```
ValueError: Expected 2D array, got 1D array instead:
array=['New York' 'California' 'Florida' 'New York' 'Florida' 'New York'
 'California' 'Florida' 'New York' 'California' 'Florida' 'California'
 'Florida' 'California' 'Florida' 'New York' 'California' 'New York'
 'Florida' 'New York' 'California' 'New York' 'Florida' 'Florida'
 'New York' 'California' 'Florida' 'New York' 'Florida' 'New York'
 'Florida' 'New York' 'California' 'Florida' 'California' 'New York'
 'Florida' 'California' 'New York' 'California' 'California' 'Florida'
 'California' 'New York' 'California' 'New York' 'Florida' 'California'
 'New York' 'California'].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.resha
pe(1, -1) if it contains a single sample.
```

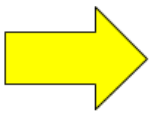
OrdinalEncoder and LabelEncoder

- Both have the same functionality.
- OrdinalEncoder is for 2D data with the shape (n_samples, n_features)
  - OrdinalEncoder is for the "features" (often a 2D array)
- LabelEncoder is for 1D data with the shape (n_samples,))
  - LabelEncoder is for the "target variable" (often a 1D array)

One Hot Encoding

- In this technique, first of all, for each distinct value of the feature, new columns are created.
- Then in these new columns, the absence of that value in the row is denoted by 1, and absence is denoted by 0.

Color
Red
Red
Yellow
Green
Yellow



Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1

https://machinelearningknowledge.ai/categorical-data-encoding-with-sklearn-labelencoder-and-onehotencoder/

One Hot Encoding Code (1)

In [36]:
```
# creating instance of one-hot-encoder ()
# The output will be a Numpy array if specifying sparse=False.
encoder = OneHotEncoder(sparse=False)
state_encoded = encoder.fit_transform(df[['State']])
print (encoder.categories_)

feature_names = encoder.get_feature_names_out(input_features=["State"])
state_encoded = pd.DataFrame(state_encoded, columns=feature_names)

# # merge with main df bridge_df on key values
df.join(state_encoded).head()
```

```
[array(['California', 'Florida', 'New York'], dtype=object)]
```

Out[36]:

	R&D Spend	Administration	Marketing Spend	State	Profit	State_Label	State_California	State_Florida	State_New York
0	165349.20	136897.80	471784.10	New York	192261.83	2	0.0	0.0	1.0
1	162597.70	151377.59	443898.53	California	191792.06	0	1.0	0.0	0.0
2	153441.51	101145.55	407934.54	Florida	191050.39	1	0.0	1.0	0.0
3	144372.41	118671.85	383199.62	New York	182901.99	2	0.0	0.0	1.0
4	142107.34	91391.77	366168.42	Florida	166187.94	1	0.0	1.0	0.0

One Hot Encoding Code (2)

In [37]:
```
# creating instance of one-hot-encoder
# The output is SciPy sparse matrix, instead of a Numpy array.
# The sparse matrix can be converted into Numpy array using toarray() method
encoder2 = OneHotEncoder()
state_encoded2 = encoder2.fit_transform(df[['State']]).toarray()

feature_names2 = encoder2.get_feature_names_out(input_features=["State"])
state_encoded2 = pd.DataFrame(state_encoded2, columns=feature_names2)

# # merge with main df bridge_df on key values
df.join(state_encoded2).head()
```

Out[37]:

	R&D Spend	Administration	Marketing Spend	State	Profit	State_Label	State_California	State_Florida	State_New York
0	165349.20	136897.80	471784.10	New York	192261.83	2	0.0	0.0	1.0
1	162597.70	151377.59	443898.53	California	191792.06	0	1.0	0.0	0.0
2	153441.51	101145.55	407934.54	Florida	191050.39	1	0.0	1.0	0.0
3	144372.41	118671.85	383199.62	New York	182901.99	2	0.0	0.0	1.0
4	142107.34	91391.77	366168.42	Florida	166187.94	1	0.0	1.0	0.0

Label Encoding vs One Hot Encoding

- Label encoding can misinterpret it by assuming they have an ordinal ranking.
  - In the dataset example, California has an encoding of 0 and Florida has encoding 1. But it does not mean anything about the numbers.

- One Hot Encoding is much suited to overcome the shortcoming of Label Encoding and is commonly used with machine learning algorithms.
  - When the cardinality of the categorical variable is high i.e. there are too many distinct values of the categorical column it may produce a very big encoding with a high number of additional columns that may not even fit the memory or produce not so great results

Data Preparation

Encoding categorical features

Data Preparation

Imputation of Missing Values

Imputer Strategies

In [ ]:
```
# Imputing with mean value
imputer = SimpleImputer(missing_values=np.NaN, strategy='mean')

# Imputing with median value
imputer = SimpleImputer(missing_values=np.NaN, strategy='median')

# Imputing with most frequent / mode value
imputer = SimpleImputer(missing_values=np.NaN, strategy='most_frequent')

# Imputing with constant value; The command below replaces the missing
#value with constant value such as 80
imputer = SimpleImputer(missing_values=np.NaN, strategy='constant', fill_value=80)
```

Imputer Example

In [25]:
```
from sklearn.impute import SimpleImputer
df2=df1.copy()

imputer = SimpleImputer(missing_values=None, strategy='most_frequent')
df2.gender = imputer.fit_transform(df2['gender'].values.reshape(-1,1))[:,0]

imputer2 = SimpleImputer(missing_values=np.NaN, strategy='mean')
df2.marks = imputer2.fit_transform(df2['marks'].values.reshape(-1,1))[:,0]

display('df1', 'df2')
```

Out[25]:

df1

	marks	gender	result
0	85.0	M	verygood
1	95.0	F	excellent
2	75.0	None	good
3	NaN	M	average
4	70.0	M	good
5	NaN	None	verygood

df2

	marks	gender	result
0	85.00	M	verygood
1	95.00	F	excellent
2	75.00	M	good
3	81.25	M	average
4	70.00	M	good
5	81.25	M	verygood

Data Preparation

Imputation of Missing Values

Data Preparation

Generating Polynomial Features

Polynomial Feature Example

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

df_data = pd.DataFrame({
    'x': np.random.randint(low=1, high=10, size=5),
    'y': np.random.randint(low=-1, high=1, size=5)})

PolyFeats = PolynomialFeatures(degree=2,include_bias=False)
poly_data = PolyFeats.fit_transform(df_data)
poly_names= PolyFeats.get_feature_names_out(df_data.columns)
df_poly_data = pd.DataFrame(poly_data, columns=poly_names)
```

df_data

	x	y
0	5	-1
1	6	-1
2	9	-1
3	7	0
4	1	0

df_poly_data

	x	y	x^2	x y	y^2
0	5.0	-1.0	25.0	-5.0	1.0
1	6.0	-1.0	36.0	-6.0	1.0
2	9.0	-1.0	81.0	-9.0	1.0
3	7.0	0.0	49.0	0.0	0.0
4	1.0	0.0	1.0	0.0	0.0

Data Preparation

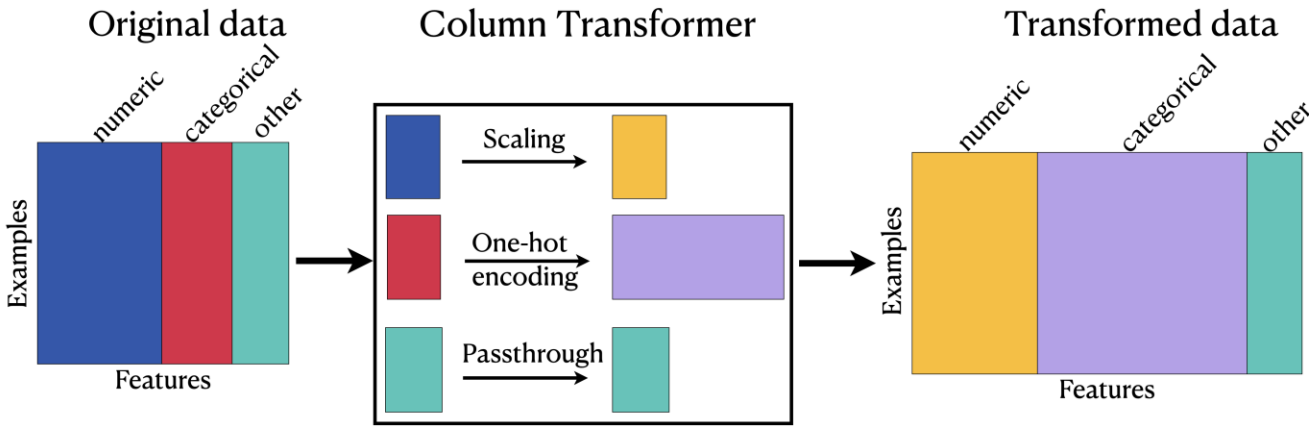
Generating Polynomial Features

Column Transformer

Column Transformer

Column Transformer

- Column Transformer is a sciket-learn class used to create and apply separate transformers for numerical and categorical data. You may do the following transformation:
  - Missing Value Imputation using Simple Imputer class
  - Ordinal encoding using Ordinal Encoder
  - Nominal encoding on countries using One Hot encoder



https://ubc-cs.github.io/cpsc330/lectures/06_column-transformer-text-feats.html

Create a column transformer

- Each transformation is specified by a **name**, a **transformer** object, and **the columns** this transformer should be applied to.

```
numeric_feats = ["university_years", "lab1", "lab3", "lab4", "quiz1"]  # apply scaling
categorical_feats = ["major"]  # apply one-hot encoding
passthrough_feats = ["ml_experience"]  # do not apply any transformation
drop_feats = [
    "lab2",
    "class_attendance",
    "enjoy_course",
]  # do not include these features in modeling
```

```
from sklearn.compose import ColumnTransformer
```

```
ct = ColumnTransformer(
    [
        ("scaling", StandardScaler(), numeric_feats),
        ("onehot", OneHotEncoder(sparse=False), categorical_feats),
    ]
)
```

make_column_transformer syntax

- The syntax automatically names each step based on its class

```
from sklearn.compose import make_column_transformer

ct = make_column_transformer(
    (StandardScaler(), numeric_feats),  # scaling on numeric features
    ("passthrough", passthrough_feats),  # no transformations on the binary features
    (OneHotEncoder(), categorical_feats),  # OHE on categorical features
    ("drop", drop_feats),  # drop the drop features
)
```

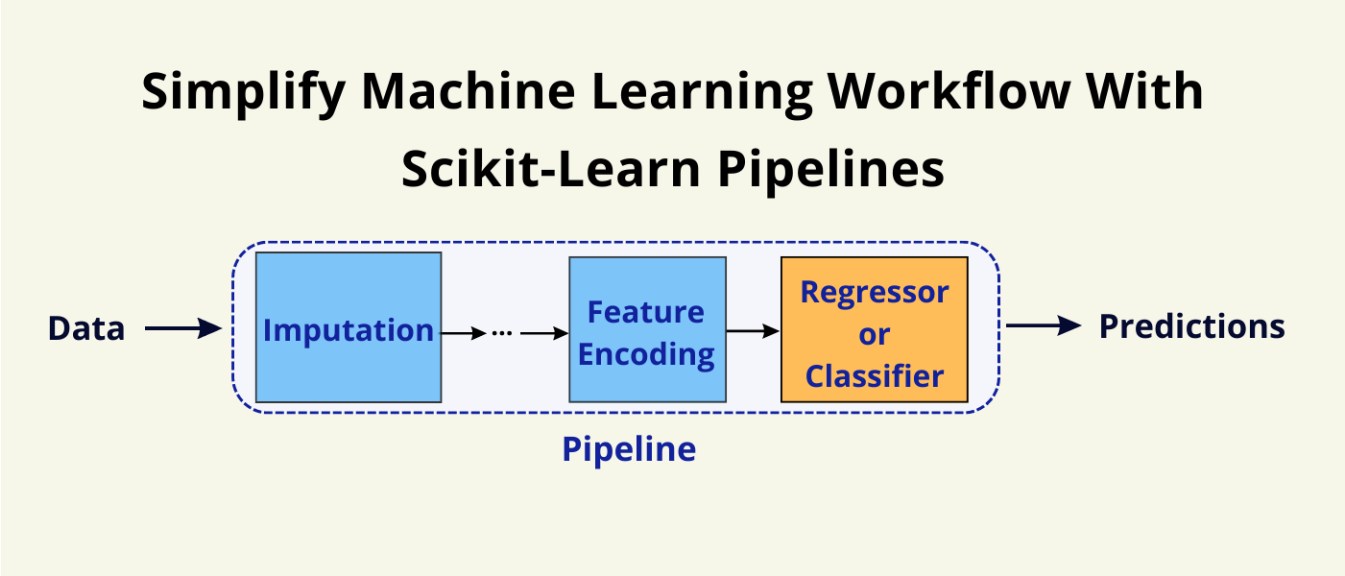
Column Transformer

Column Transformer

ML Pipeline

ML Pipeline

What is Pipeline?



How to use a pipeline?

```
### Simple example of a pipeline
from sklearn.pipeline import Pipeline

pipe = Pipeline(
    steps=[
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        ("regressor", KNeighborsRegressor()),
    ]
)
```

- Syntax: pass in a list of steps.
- The last step should be a model/classifier/regressor.
- All the earlier steps should be transformers.

Alternative syntax: make_pipeline

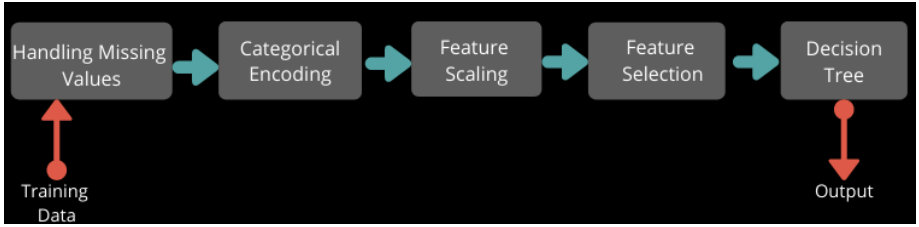
```
from sklearn.pipeline import make_pipeline

pipe = make_pipeline(
    SimpleImputer(strategy="median"), StandardScaler(), KNeighborsRegressor()
)
```

- Shorthand for Pipeline constructor
- Does not permit naming steps
- The names of steps are set to lowercase of their types automatically; e.g., StandardScaler() would be named as standardscaler

A transformer and pipeline example

```
#1st Imputation Transformer
trf1 = ColumnTransformer([
        ('impute_age',SimpleImputer(),[2]),
        ('impute_embarked',SimpleImputer(strategy='most_frequent'),[6])
    ],remainder='passthrough')

#2nd One Hot Encoding
trf2 = ColumnTransformer([
        ('ohe_sex_embarked', OneHotEncoder(sparse=False, handle_unknown='ignore'),[1,6])
    ], remainder='passthrough')

#3rd Scaling
trf3 = ColumnTransformer([
    ('scale', MinMaxScaler(), slice(0,10))
])

#4th Feature selection
trf4 = SelectKBest(score_func=chi2,k=8)

#5th Model
trf5 = DecisionTreeClassifier()
```

```
pipe = Pipeline([
    ('trf1', trf1),
    ('trf2', trf2),
    ('trf3', trf3),
    ('trf4', trf4),
    ('trf5', trf5)
])
```



ML Pipeline

ML Pipeline