DATA PREPARATION AND VISUALIZATION

Mathematical Economics Faculty

National Economics University
https://www.neu.edu.vn/

Data Transformation

Outline

- Why data transformation
- How to Scale Numerical Data
- How to Scale Data with Outliers
- How to Encode Categorical data

Basic Data Transformation

Transforming Data Using a Function or Mapping

- For many datasets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame
- Consider the following hypothetical data

food	ounces
bacon	4.0
Pulled pork	3.0
bacon	12.0
Pastrami	6.0
Corned beef	7.5
bacon	8.0
pastrami	3.0
Honey ham	5.0
Nova lox	6.0

- You wanted to add a column indicating the type of animal that each food came from
  - Write down a mapping of each distinct meat type to the kind of animal
  - Use the pd.Series.map() method to apply this mapping to the 'food' series

Basic Data Transformation

Discretization and Binning

Pd.cut()

Parameters	Descriptions
x	The input array to be binned. Must be 1-dimensional
bins	Int, sequence of scalars, or IntervalIndex • Defines the number of equal-width bins in the range of x. The range of x is extended by .1% on each side to include the minimum and maximum • Sequence of scalars: Defines the bin edges allowing for non-uniform width. No extension of the range of x is done • IntervalIndex: Defines the exact bins to be used. Note that IntervalIndex for bins must be non-overlapping
labels	Array or False, default None. Specifies the lables for the returned bins. Must be the same length as the resulting bins
precision	The precision at which to store and display the bins labels
duplicates	Default ‘raise’, ‘drop’. If bin edges are not unique, raise ValueError or drop non-uniques

Basic Data Transformation

Discretization and Binning

pd.qcut()

Parameters	Descriptions
x	The input array to be binned. Must be 1-dimensional
q	Int or list-like of float Number of quantiles. 10 for deciles, 4 for quartiles, etc. Alternately array of quantiles, e.g [0,.25,.5,.75,1.] for quartiles
labels	Array or False, default None. Specifies the lables for the returned bins. Must be the same length as the resulting bins
precision	The precision at which to store and display the bins labels
duplicates	Default 'raise', 'drop'. If bin edges are not unique, raise ValueError or drop non-uniques

Numerical Data Scaling Methods

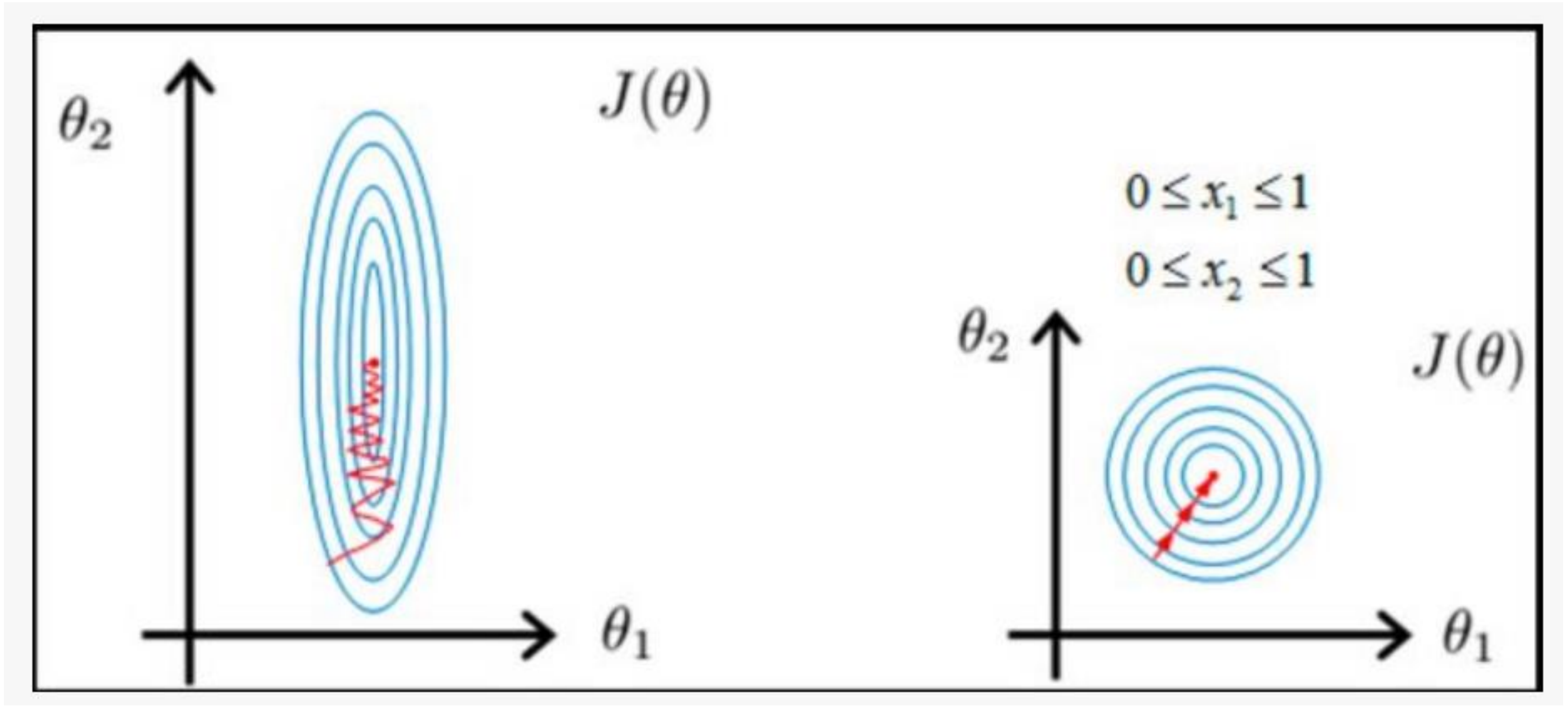
Why Scaling?

- Machine learning models learn a mapping from input variables to an output variable
- The scale and distribution of the data drawn from the domain may be different for each variable
- The difference in scale for input variables affects the algorithms that fit a model using a weighted sum of input
  - Ex: linear regression, logistic regression, artificial neural networks
- Algorithms that use distance measures between examples are also affected, such as k-nearest neighbors and support vector machines.
- There are also algorithms that are unaffected by the scale of numerical input variables, most notably decision trees and ensembles of trees like random forest

Numerical Data Scaling Methods

Why scaling?

Scaling data helps the gradient converge faster



Numerical Data Scaling Methods

Data Normalization

- Normalization is a rescaling of the data from the original range so that all values are within the new range of 0 and 1

A value is normalized as follows:

$$y = \frac{x - \min}{\max - \min}$$

- Example: For a dataset, the min and max observable values are 30 and -10. What is the normalize value of the value 18.8?

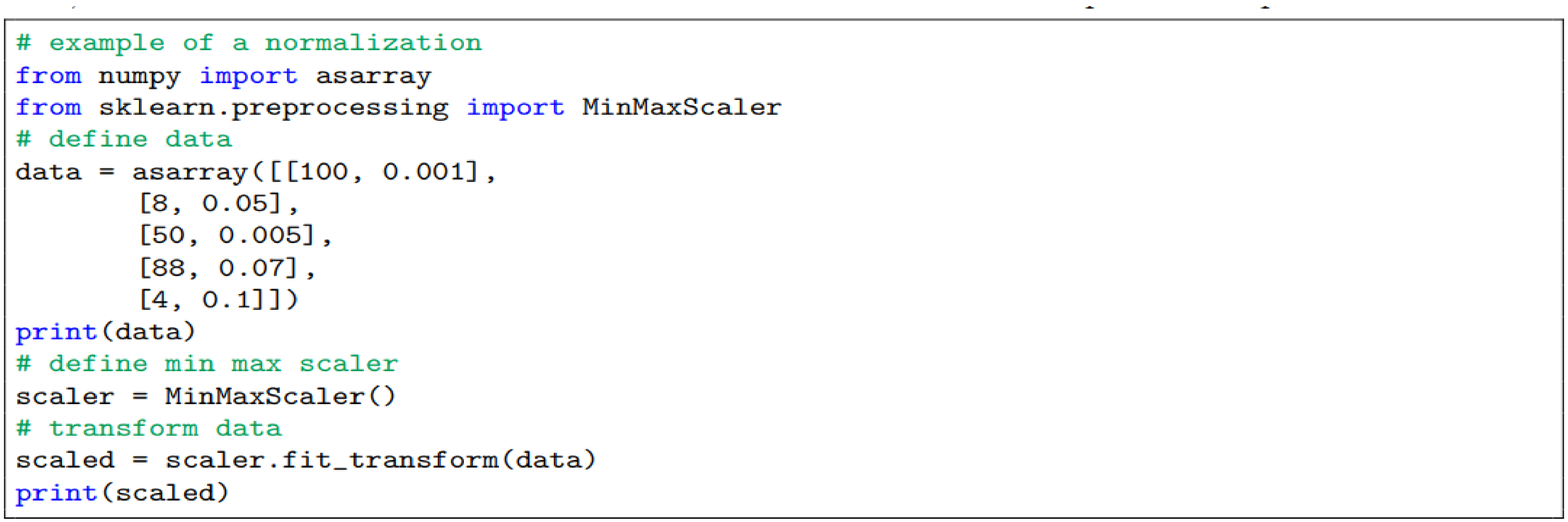
Numerical Data Scaling Methods

Data Normalization

- You can normalize your dataset using the scikit-learn object MinMaxScaler
  - **Fit the scaler using available training data**.
    - ✓ This is done by calling the fit() function.
    - ✓ This means that training data will be used to estimate the minimum and maximum observable values
  - **Apply the scale to training data**.
    - ✓ This is done by calling the transform() function
  - **Apply the scale to data going forward**.
    - ✓ This means you can prepare new data in the future on which you want to make predictions

Numerical Data Scaling Methods

Data Normalization



```python
# example of a normalization
from numpy import asarray
from sklearn.preprocessing import MinMaxScaler
# define data
data = asarray([[100, 0.001],
       [8, 0.05],
       [50, 0.005],
       [88, 0.07],
       [4, 0.1]])
print(data)
# define min max scaler
scaler = MinMaxScaler()
# transform data
scaled = scaler.fit_transform(data)
print(scaled)
```

Listing 17.2: Example of normalizing values in a dataset.

[sklearn.preprocessing.MinMaxScaler — scikit-learn 1.1.2 documentation]()

Numerical Data Scaling Methods

Data Standardization

- Standardizing a dataset involves rescaling the distribution of values so that the mean of observed values is 0 and the standard deviation is 1
- Standardization assumes that *your observations fit a Gaussian distribution* with a well-behaved mean and standard deviation.
- You can still standardize your data of this expectation is not met, but you may not get reliable results

Numerical Data Scaling Methods

Data Standardization

A value is standardized as follows:

$$y = \frac{x - \text{mean}}{\text{standard\_deviation}}$$

Where the mean is calculated as:

$$\text{mean} = \frac{1}{N} \times \sum_{i=1}^{N} x_i$$

And the standard_deviation is calculated as:

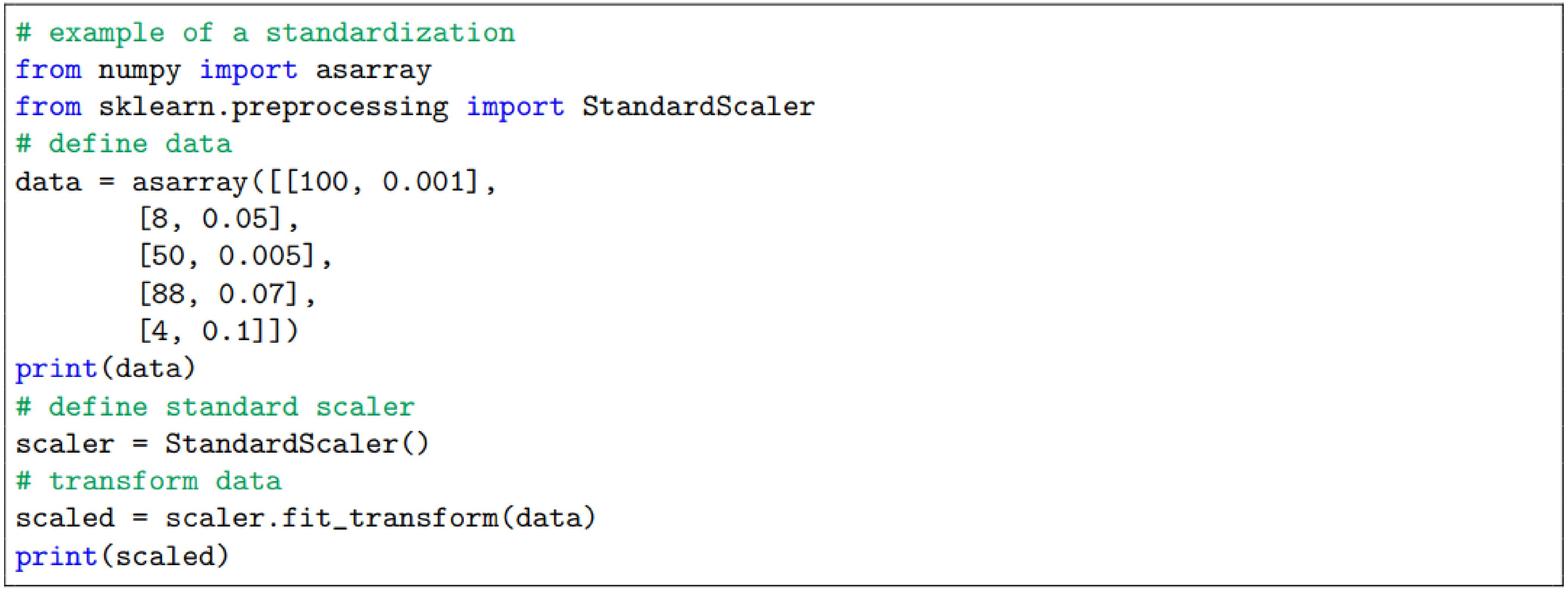
$$\text{standard\_deviation} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \text{mean})^2}{N-1}}$$

Ex: Standardize the value of 20.7 if we know the mean of data is 10 and the standard deviation of data is 5

Numerical Data Scaling Methods

Data Standardization

You can standardize your dataset using the scikit-learn object StandardScaler



```python
# example of a standardization
from numpy import asarray
from sklearn.preprocessing import StandardScaler
# define data
data = asarray([[100, 0.001],
       [8, 0.05],
       [50, 0.005],
       [88, 0.07],
       [4, 0.1]])
print(data)
# define standard scaler
scaler = StandardScaler()
# transform data
scaled = scaler.fit_transform(data)
print(scaled)
```

Common Questions

Q1. Should I Normalize or Standardize?

- ✓ A: Whether input variables require scaling depends on the specifics of your problem and of each variable. You may have a sequence of quantities as inputs, such as prices or temperatures. If the distribution of the quantity is normal, then it should be standardized, otherwise, the data should be normalized. This applies if the range of quantity values is large (10s, 100s, etc.) or
  small (0.01, 0.0001).
- ✓ If the quantity values are small (near 0-1) and the distribution is limited (e.g. standard deviation near 1), then perhaps you can get away with no scaling of the data

Common Questions

Q2: Should I Standardize then Normalize?

A: Standardization can give values that are both positive and negative centered around zero. It may be desirable to normalize data after it has been standardized. This might be a good idea of you have a mixture of standardized and normalized variables and wish all input variables to have the same minimum and maximum values as input for a given algorithm, such as an algorithm that calculates distance measures.

Common Questions

- **Q3: But Which is Best?**

  A: This is unknowable. Evaluate models on data prepared with each transform and use the transform or combination of transforms that result in the best performance for your data set on your model.

- **Q4: How Do I Handle Out-of-Bounds Values?**

  A**:** You may normalize your data by calculating the minimum and maximum on the training data. Later, you may have new data with values smaller or larger than the minimum or maximum respectively. One simple approach to handling this may be to check for such out-of-bound values and change their values to the known minimum or maximum prior to scaling. Alternately, you may want to estimate the minimum and maximum values used in the normalization manually based on domain knowledge.

How to Scale Data With Outliers

Data Standardization

- Standardization can become skewed or biased if the input variable contains outlier values
- To overcome this, the median and interquartile range can be used when standardizing numerical input variables, generally referred to as robust scaling
- One approach to standardizing input variables in the presence of outliers is to ignore the outliers from the calculation of the mean and standard deviation, then use the calculated values to scale the variable

$$\text{value} = \frac{\text{value} - \text{median}}{p_{75} - p_{25}}$$

How to Scale Data With Outliers

Robust Scaling Data

- Robust Scaling

$$\text{value} = \frac{\text{value} - \text{median}}{p_{75} - p_{25}}$$

- The robust scaler transform is available in the scikit-learn Python machine learning library via *the RobustScaler class*
  - The with_centering argument controls whether the value is centered to zero and defaults to True
  - The with_scaling argument control whether the value is scaled to the IQR, default True
  - The definition of the scaling range can be specified via the quantile_range argument. It takes a tuple of two integers between 0 and 100. ex: (25,75)

How to Scale Data With Outliers

Robust Scaler Transforms

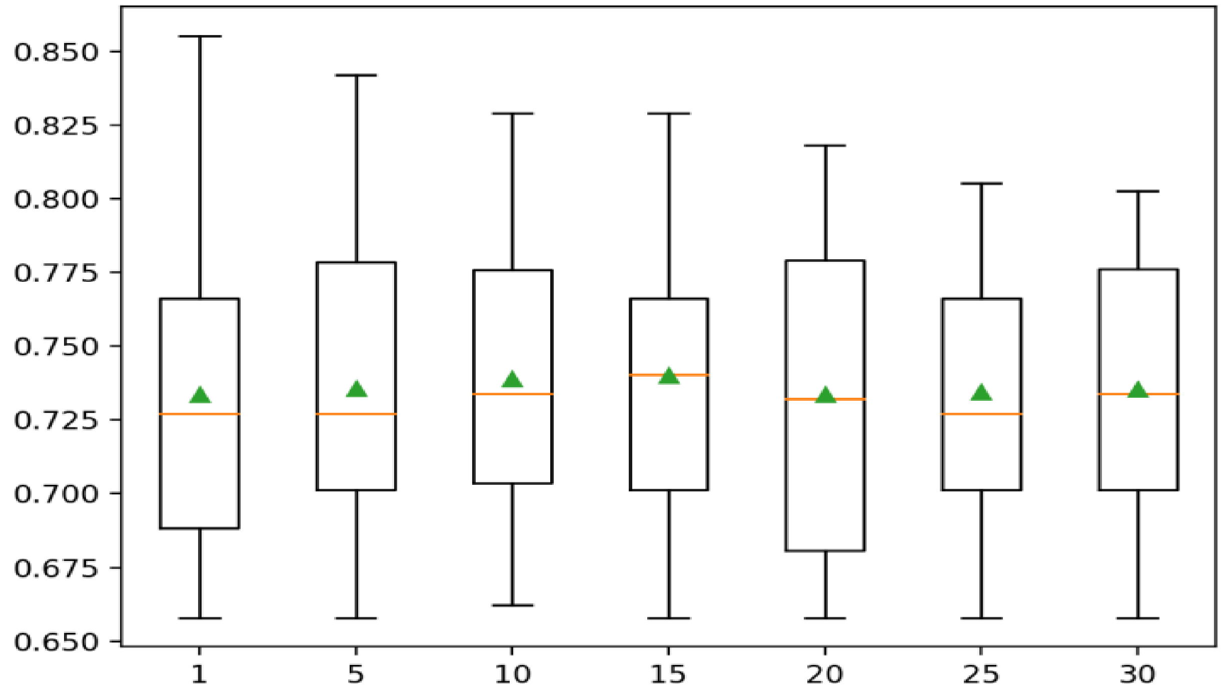


Figure 18.3: Box Plots of Robust Scaler IQR Range vs Classification Accuracy of KNN on the Diabetes Dataset.

How to Encode Categorical Data

Nominal and Ordinal Variables

- **Nominal Variable**: Variable comprises a finite set of discrete values with no rank-order relationship between values
- **Ordinal Variable**: Variable comprises a finite set of discrete values with a ranked ordering between values
- Three common approaches for converting ordinal and categorical variables to numerical values
  - Ordinal Encoding
  - One Hot Encoding
  - Dummy Variable Encoding

How to Encode Categorical Data

Ordinal Encoding

- In ordinal encoding, each unique category value is assigned an integer value.
  - Ex: red is 1, green is 2, blue is 3
- This ordinal encoding transform is available in the scikit-learn package via *OrdinalEncoder* class
- By default, it will assign integers to labels in the order that is observed in the data. If a specific order is desired, it can be specified via the *categories* argument as a list with the rank order of all expected labels

How to Encode Categorical Data

Ordinal Encoding

```python
# example of a ordinal encoding
from numpy import asarray
from sklearn.preprocessing import OrdinalEncoder
# define data
data = asarray([['red'], ['green'], ['blue']])
print(data)
# define ordinal encoding
encoder = OrdinalEncoder()
# transform data
result = encoder.fit_transform(data)
print(result)
```

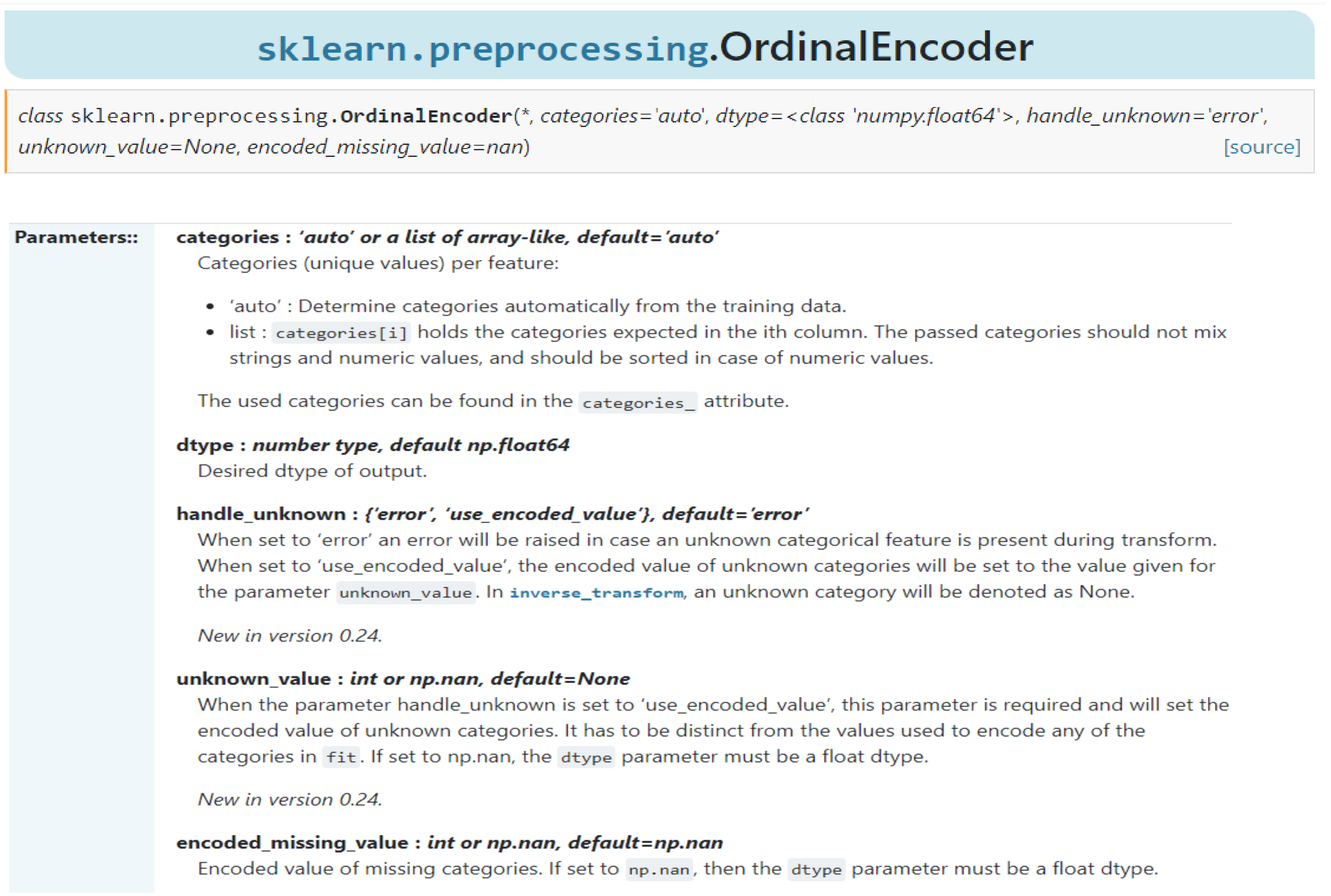
Listing 19.1: Example of demonstrating an ordinal encoding of color categories.

```
[['red']
 ['green']
 ['blue']]
[[2.]
 [1.]
 [0.]]
```

Listing 19.2: Example output from demonstrating an ordinal encoding of color categories.

How to Encode Categorical Data

Ordinal Encoding



sklearn.preprocessing.OrdinalEncoder

class `sklearn.preprocessing.OrdinalEncoder`(**, categories='auto', dtype=<class 'numpy.float64'>, handle_unknown='error', unknown_value=None, encoded_missing_value=nan*) [source]

Parameters::

categories : *'auto' or a list of array-like, default='auto'*

Categories (unique values) per feature:

- 'auto' : Determine categories automatically from the training data.
- list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values, and should be sorted in case of numeric values.

The used categories can be found in the `categories_` attribute.

dtype : *number type, default np.float64*

Desired dtype of output.

handle_unknown : *{'error', 'use_encoded_value'}, default='error'*

When set to 'error' an error will be raised in case an unknown categorical feature is present during transform. When set to 'use_encoded_value', the encoded value of unknown categories will be set to the value given for the parameter `unknown_value`. In `inverse_transform`, an unknown category will be denoted as None.

New in version 0.24.

unknown_value : *int or np.nan, default=None*

When the parameter handle_unknown is set to 'use_encoded_value', this parameter is required and will set the encoded value of unknown categories. It has to be distinct from the values used to encode any of the categories in `fit`. If set to np.nan, the `dtype` parameter must be a float dtype.

New in version 0.24.

encoded_missing_value : *int or np.nan, default=np.nan*

Encoded value of missing categories. If set to `np.nan`, then the `dtype` parameter must be a float dtype.

How to Encode Categorical Data

OneHotEncoding

- For categorical variables where no ordinal relationship exists, the integer encoding may not be enough or even misleading to the model
- Forcing an ordinal relationship via an ordinal encoding and allowing the model to assume a natural ordering between categories may result in poor performance or unexpected results
- In this case, a one hot encoding can be applied to the ordinal representation.
- This is where the integer encoded variable is removed and one new binary variable is added for each unique integer value in the variable

How to Encode Categorical Data

OneHotEncoding

COLOR
Red
Green
Blue

$\longrightarrow$

Red	Green	Blue
1	0	0
0	1	0
0	0	1

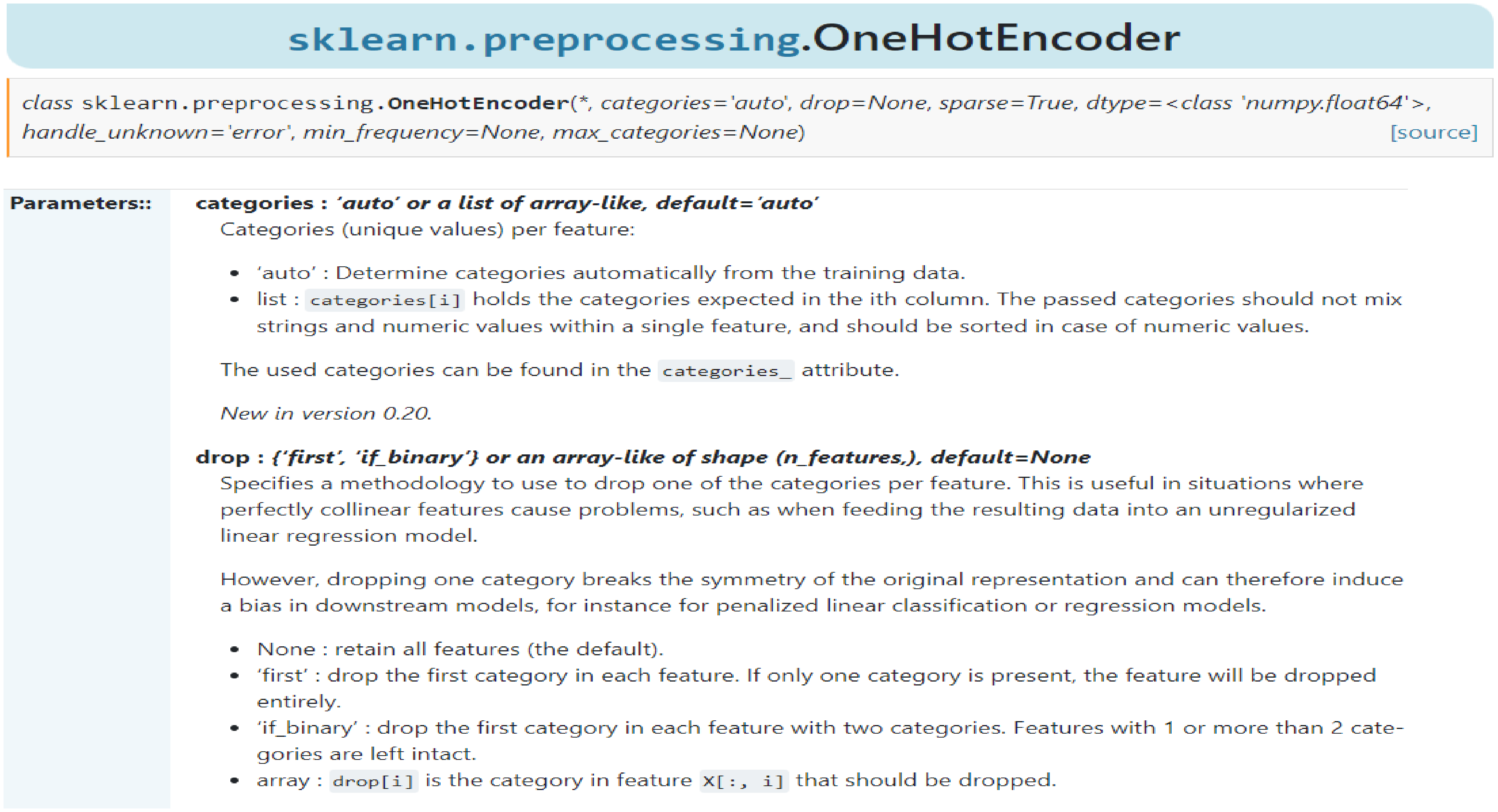
How to Encode Categorical Data

OneHotEncoding - Example

- The input to the *OneHotEncoder* transformer should be an array-like of integers or strings, denoting the value taken on by categorical (discrete) features.
- This creates a binary column for each category and returns a sparse matrix or dense array (depending on the *sparse* parameter)
- If you know all of the labels to be expected in the data, they can be specified via the *categories* argument as a list
- If new data contains categories not seen in the training dataset, the handle_unknown argument can be set to ‘ignore’ to not raise an error, which will result in a zero value for each label

How to Encode Categorical Data

OneHotEncoding - Example



sklearn.preprocessing.OneHotEncoder

class sklearn.preprocessing.**OneHotEncoder**(**, categories='auto', drop=None, sparse=True, dtype=<class 'numpy.float64'>, handle_unknown='error', min_frequency=None, max_categories=None*) [source]

Parameters::

categories : *'auto' or a list of array-like, default='auto'*

Categories (unique values) per feature:

- 'auto' : Determine categories automatically from the training data.
- list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values within a single feature, and should be sorted in case of numeric values.

The used categories can be found in the `categories_` attribute.

New in version 0.20.

drop : *{'first', 'if_binary'} or an array-like of shape (n_features,), default=None*

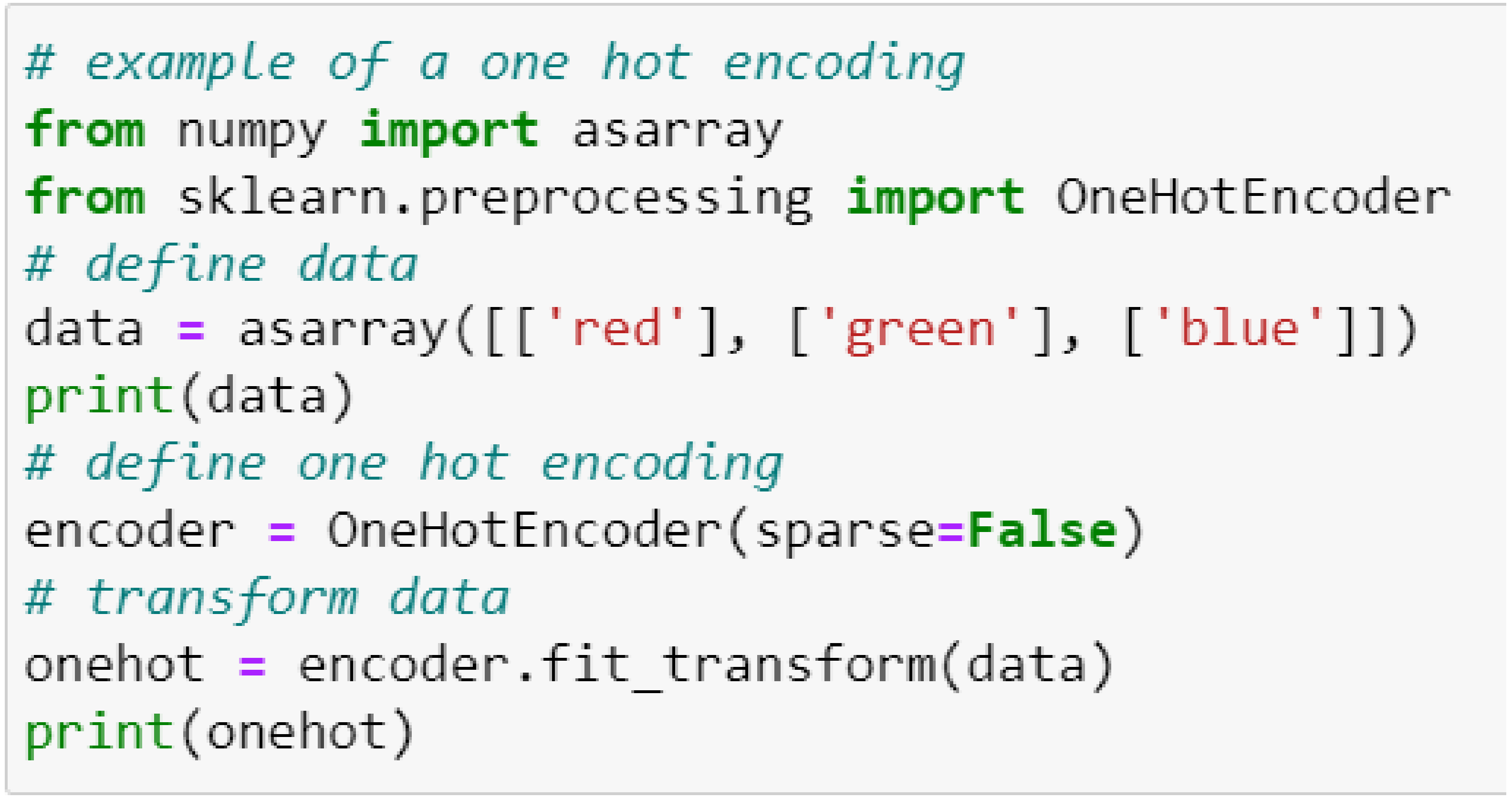
Specifies a methodology to use to drop one of the categories per feature. This is useful in situations where perfectly collinear features cause problems, such as when feeding the resulting data into an unregularized linear regression model.

However, dropping one category breaks the symmetry of the original representation and can therefore induce a bias in downstream models, for instance for penalized linear classification or regression models.

- None : retain all features (the default).
- 'first' : drop the first category in each feature. If only one category is present, the feature will be dropped entirely.
- 'if_binary' : drop the first category in each feature with two categories. Features with 1 or more than 2 categories are left intact.
- array : `drop[i]` is the category in feature `X[:, i]` that should be dropped.

How to Encode Categorical Data

OneHotEncoding - Example



```
# example of a one hot encoding
from numpy import asarray
from sklearn.preprocessing import OneHotEncoder
# define data
data = asarray([['red'], ['green'], ['blue']])
print(data)
# define one hot encoding
encoder = OneHotEncoder(sparse=False)
# transform data
onehot = encoder.fit_transform(data)
print(onehot)
```

How to Encode Categorical Data

Dummy Variable Encoding

- The one hot encoding creates one binary variable for each category. The problem is that this representation includes redundancy.
    - Ex: if we know that [1,0,0] represents blue and [0,1,0] represents green we don’t need another binary variable to represent red, instead we could use 0 values alone, e.g [0,0].
- This is called a dummy variable encoding, and always represents C categories with C-1 binary variables
- In addition to being slightly less redundant, a dummy variable representation is required for all regression models that have a bias term
    - For example, in the case of a linear regression model (and other regression models that have a bias term), a one hot encoding will cause the matrix of input data to become singular(its determinant is 0)

How to Encode Categorical Data

Common Questions

- Q1. What if I have a mixture of numeric and categorical data?
  - You can use the ColumnTransformer to conditionally apply different data transforms to different input variables
- Q2. What if I have hundreds of categories?
  - You can use a one hot encoding up to thousands and tens of thousands of categories. Also, having large vectors as input sounds intimidating, but the models can generally handle it
- Q3. What encoding technique is the best?
  - This is unknowable. Test each technique (and more) on your dataset with your chosen model and discover what works best for your case.

Make Distribution More Gaussian

Box-Cox Transformation

- Machine learning algorithms like Linear Regression and Gaussian Naive Bayes assume the numerical variables have a Gaussian probability distribution.
- Your data may not have a Gaussian distribution and instead may have a Gaussian-like distribution (e.g. nearly Gaussian but with outliers or a skew) or a totally different distribution (e.g. exponential).
- Power transforms(ex **Box-Cox transformation**) are a technique for transforming numerical input or output variables to have a Gaussian or more Gaussian-like probability distribution

Make Distribution More Gaussian

Box-Cox Transformation

- The original Box-Cox transformation is a one-dimensional transformation with one parameter λ

Let $y \in \mathbb{R}^n$ and $\lambda \in \mathbb{R}$

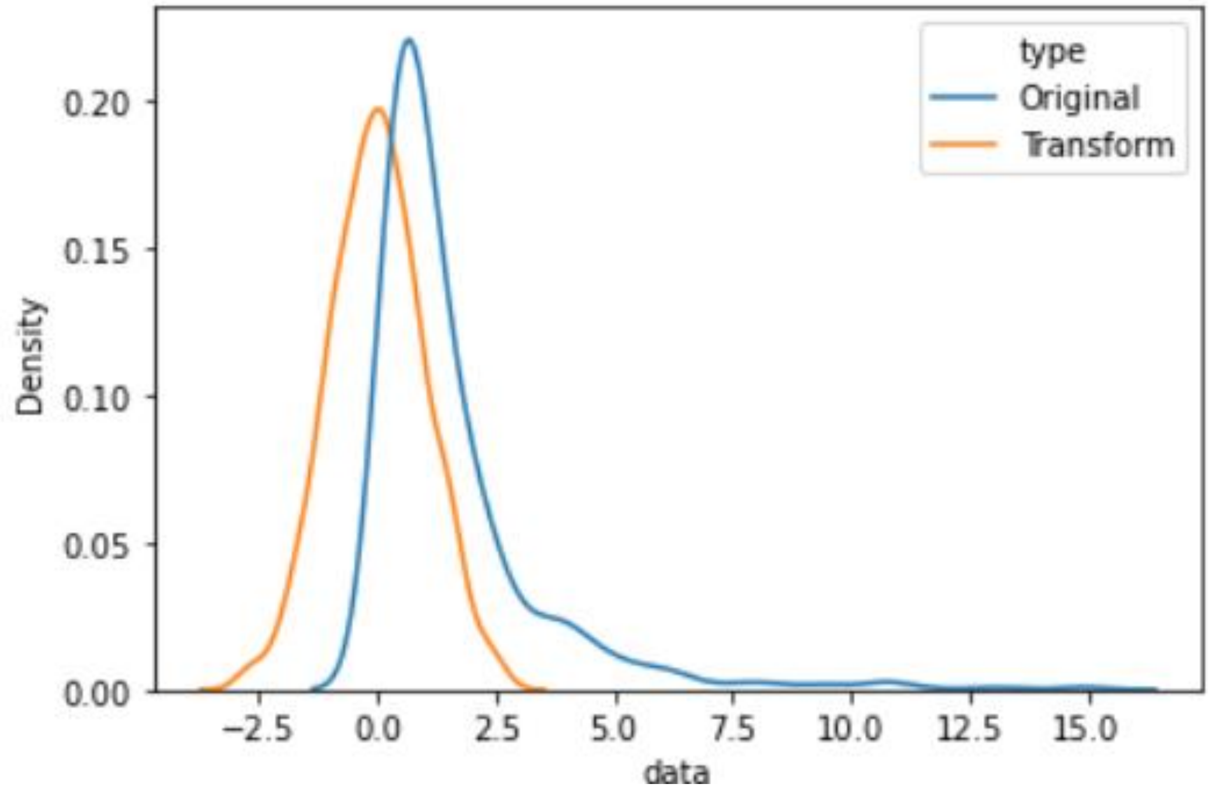
$$y_i^{(\lambda)} = \begin{cases} \frac{y_i^{(\lambda)} - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \ln(y_i) & \text{if } \lambda = 0 \end{cases}$$

- For λ >1 the transformation is convex
- For λ < 1 the transformation is concave

Make Distribution More Gaussian

Box-Cox Transformation

- The original Box-Cox transformation is a one-dimensional transformation with one parameter λ



Box-cox transformation with λ = 0.0097

Make Distribution More Gaussian

Box-Cox Transformation

- The optimization of the parameter λ can be done in several ways
  - Maximum Likelihood Estimator
  - Bayesian approach
- Some common values for lambda
  - λ = -1 is a reciprocal transform
  - λ = -0.5 is a reciprocal square root transform
  - λ = 0.0 is a log transform
  - λ = 0.5 is a square root transform
  - λ = 1.0 is no transform

Make Distribution More Gaussian

Yeo-Johnson Transformation

- Yeo and Johnson (2000) have proposed an new family of distributions that can be used without restrictions on that have many of the good properties of the Box-Cox power

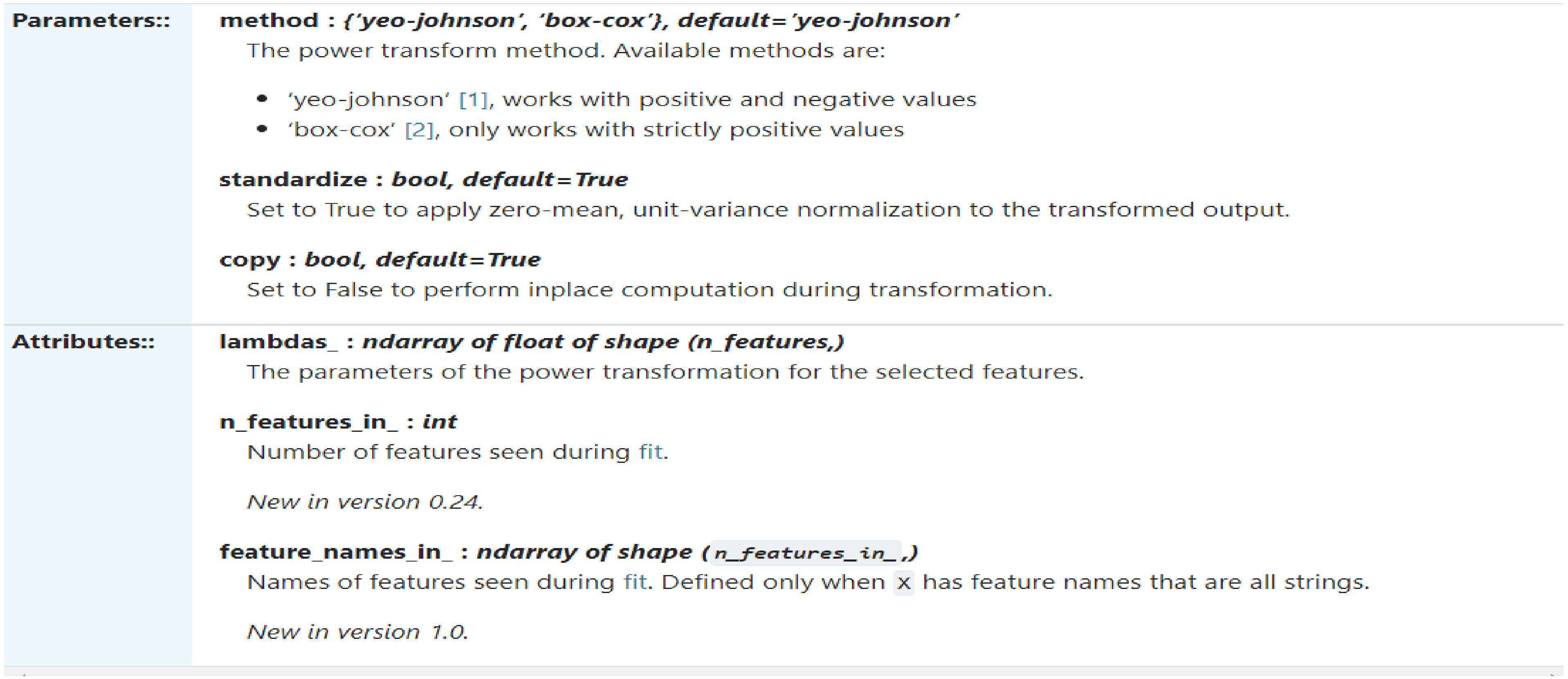
$$\psi(\lambda, y) = \begin{cases} ((y+1)^{\lambda} - 1)/\lambda & \text{if } \lambda \neq 0, y \geq 0 \\ \log(y+1) & \text{if } \lambda = 0, y \geq 0 \\ -[(-y+1)^{2-\lambda} - 1)]/(2-\lambda) & \text{if } \lambda \neq 2, y < 0 \\ -\log(-y+1) & \text{if } \lambda = 2, y < 0 \end{cases} \qquad (2)$$

- If y > 0, then the Yeo-Johnson transformation is the same as the Box-Cox power transformation of (y+1)
- If y<0. then the Yeo-Johnson transformation is the Box-Cox power transformation of (-y+1) but with power 2 - λ
- With both negative and positive values, the transformation is a mixture of these two

Make Distribution More Gaussian

PowerTransformation Class

- These power transforms are available in the scikit-learn Python machine learning library via the PowerTransformer class

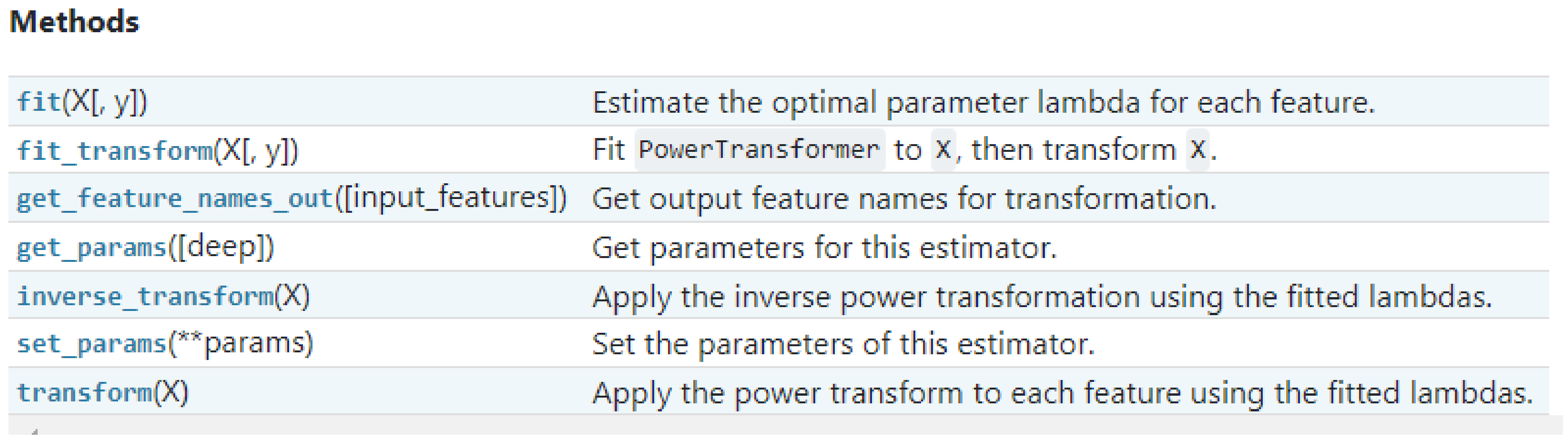


Parameters::	**method : *{'yeo-johnson', 'box-cox'}, default='yeo-johnson'*** The power transform method. Available methods are: • 'yeo-johnson' [1], works with positive and negative values • 'box-cox' [2], only works with strictly positive values **standardize : *bool, default=True*** Set to True to apply zero-mean, unit-variance normalization to the transformed output. **copy : *bool, default=True*** Set to False to perform inplace computation during transformation.
Attributes::	**lambdas_ : *ndarray of float of shape (n_features,)*** The parameters of the power transformation for the selected features. **n_features_in_ : *int*** Number of features seen during fit. *New in version 0.24.* **feature_names_in_ : *ndarray of shape (`n_features_in_`,)*** Names of features seen during fit. Defined only when `X` has feature names that are all strings. *New in version 1.0.*

Make Distribution More Gaussian

PowerTransformation Class

- These power transforms are available in the scikit-learn Python machine learning library via the PowerTransformer class



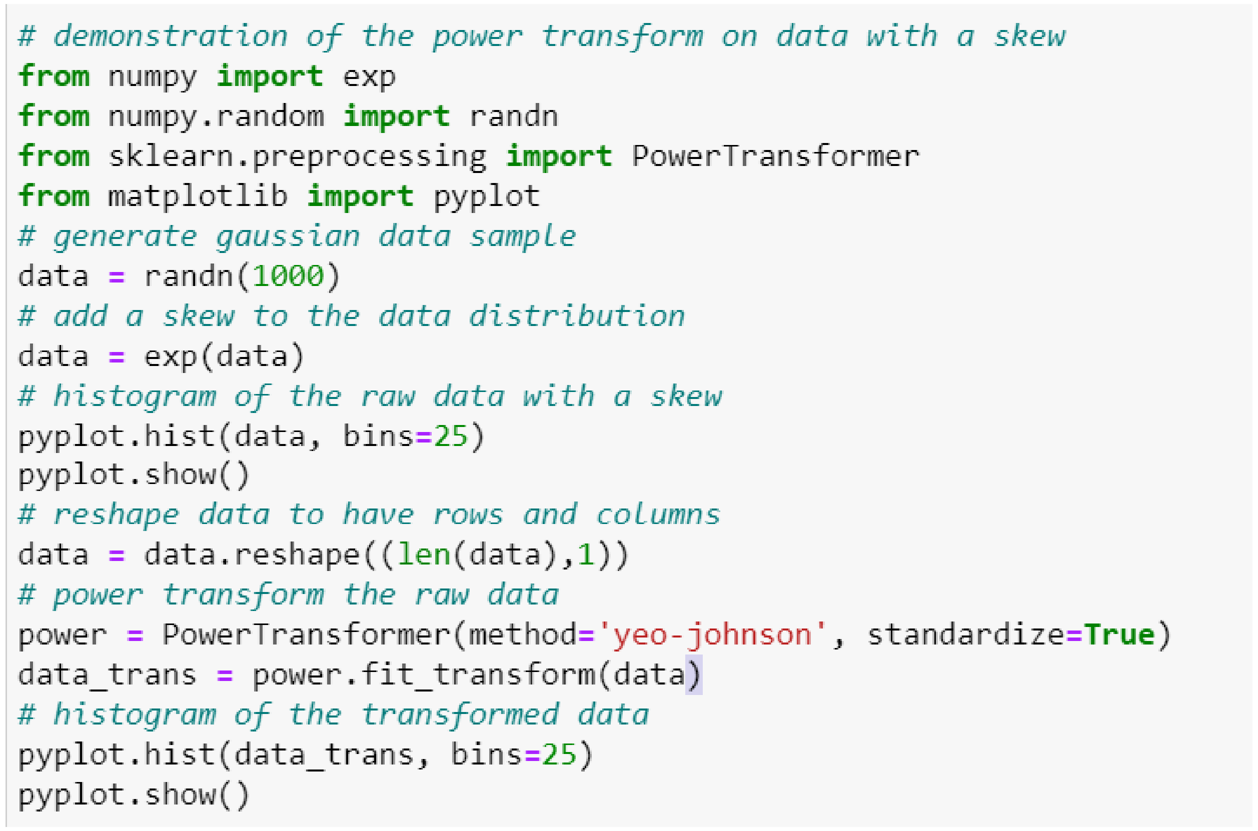
Methods

`fit`(X[, y])	Estimate the optimal parameter lambda for each feature.
`fit_transform`(X[, y])	Fit `PowerTransformer` to `X`, then transform `X`.
`get_feature_names_out`([input_features])	Get output feature names for transformation.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X)	Apply the inverse power transformation using the fitted lambdas.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Apply the power transform to each feature using the fitted lambdas.

Make Distribution More Gaussian

PowerTransformation Class Example

- These power transforms are available in the scikit-learn Python machine learning library via the PowerTransformer class



```python
# demonstration of the power transform on data with a skew
from numpy import exp
from numpy.random import randn
from sklearn.preprocessing import PowerTransformer
from matplotlib import pyplot
# generate gaussian data sample
data = randn(1000)
# add a skew to the data distribution
data = exp(data)
# histogram of the raw data with a skew
pyplot.hist(data, bins=25)
pyplot.show()
# reshape data to have rows and columns
data = data.reshape((len(data),1))
# power transform the raw data
power = PowerTransformer(method='yeo-johnson', standardize=True)
data_trans = power.fit_transform(data)
# histogram of the transformed data
pyplot.hist(data_trans, bins=25)
pyplot.show()
```

The ColumnTransformer

- The *ColumnTransformer* is a class in the scikit-learn Python machine learning that allows you to selectively apply data preparation transforms
- To use the *ColumnTransformer* you must specific a list of transformers.
- Each transformer is a three-element tuple that defines the name of the transformer, the transform to apply, and the column indices to apply it to.
  - For example, the ColumnTransformer below applies a OneHotEncoder to columns 0 and 1

    transformer = (‘cat’, OneHotEncoder(), [0,1])
- The example below applies a SimpleImputer with median imputing for numerical columns 0 and 1, and SimpleImputer with most frequent imputing to categorical columns 2 and 3.

  t = [('num', SimpleImputer(strategy='median'), [0, 1]),

  ('cat',SimpleImputer(strategy='most_frequent'), [2, 3])]

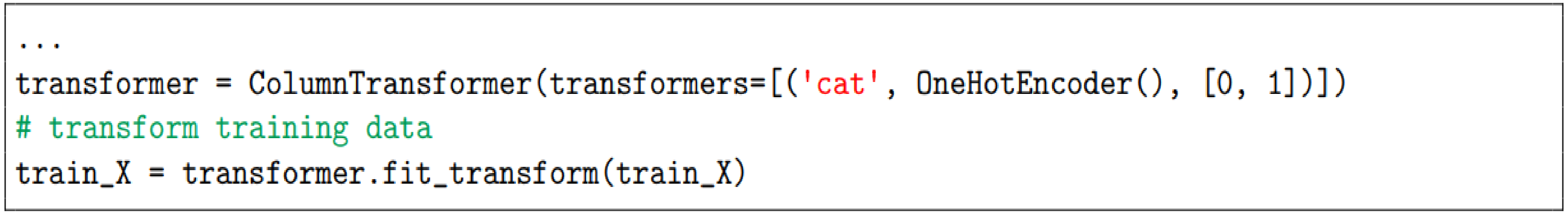
  transformer = ColumnTransformer(transformers=t)

The ColumnTransformer

- Any columns not specified in the list of transformers are dropped from the dataset by default; this can be changed by setting the remainder argument.
- Setting remainder=‘passthrough’ will mean that all columns not specified in the list of transformers will be passed through without transformation, instead of being dropped
- Ex

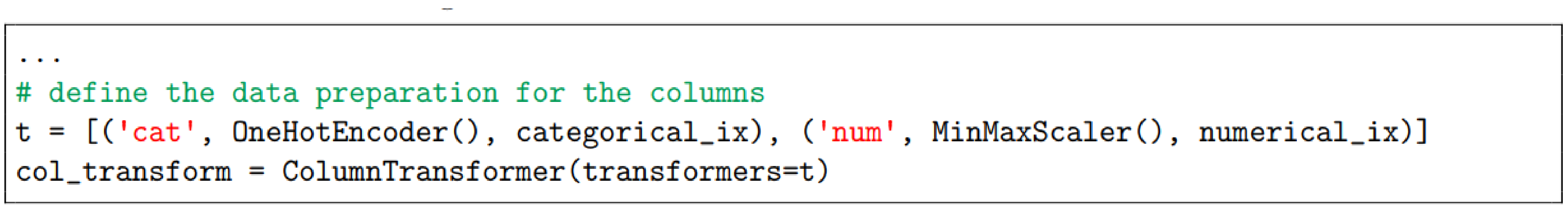
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [2, 3])], remainder='passthrough')

- Once the transformer is defined, it can be used to transform a dataset. For example:



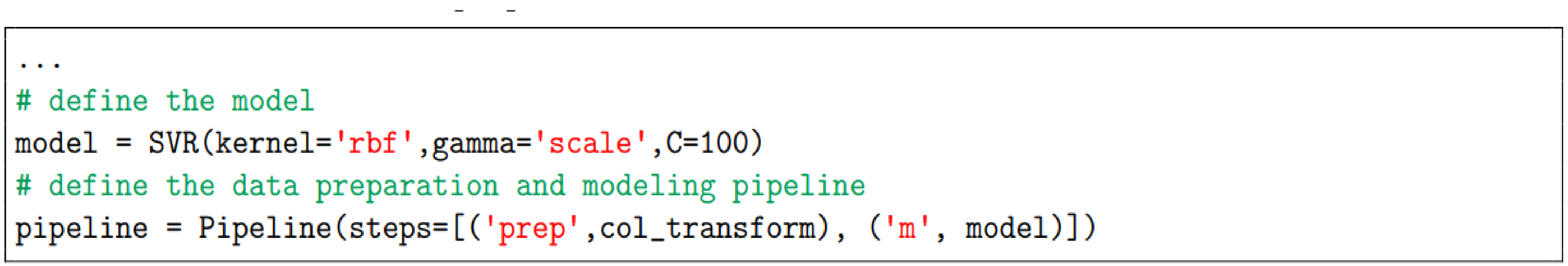
```
...
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [0, 1])])
# transform training data
train_X = transformer.fit_transform(train_X)
```

Data Preparation for the Abalone Regression Dataset



```
...
# define the data preparation for the columns
t = [('cat', OneHotEncoder(), categorical_ix), ('num', MinMaxScaler(), numerical_ix)]
col_transform = ColumnTransformer(transformers=t)
```

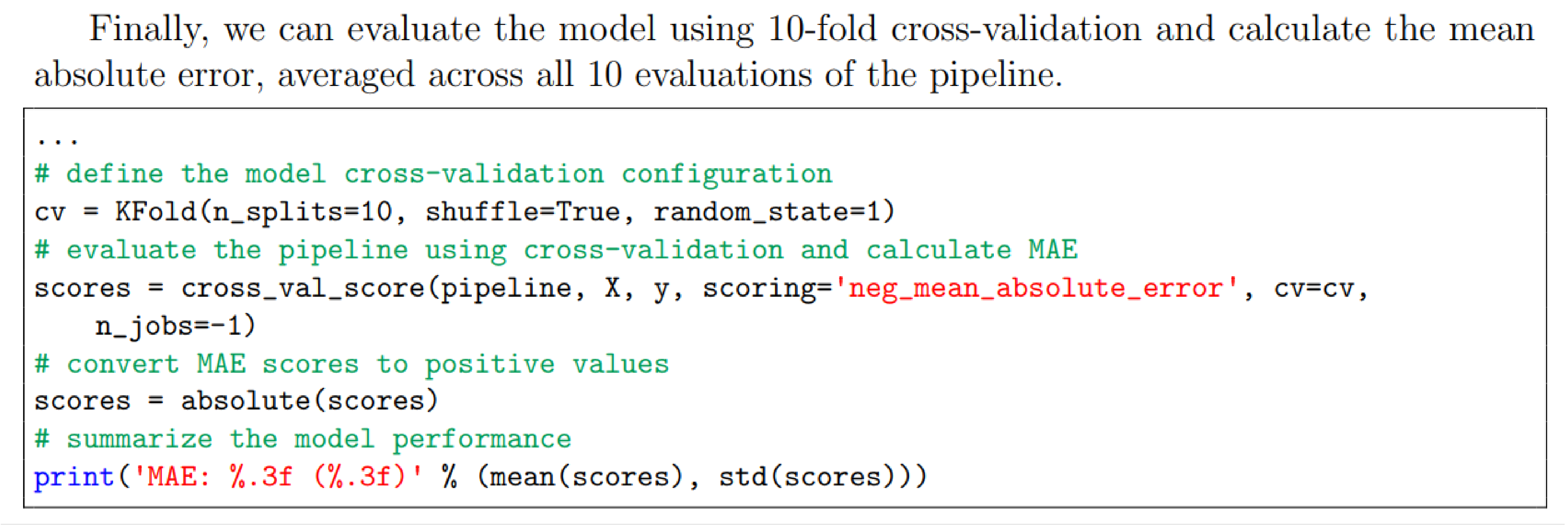
Listing 24.12: Example of using different data transforms for different data types.



```
...
# define the model
model = SVR(kernel='rbf',gamma='scale',C=100)
# define the data preparation and modeling pipeline
pipeline = Pipeline(steps=[('prep',col_transform), ('m', model)])
```

Listing 24.13: Example of defining a `Pipeline` with a SVR model.

Data Preparation for the Abalone Regression Dataset



Finally, we can evaluate the model using 10-fold cross-validation and calculate the mean absolute error, averaged across all 10 evaluations of the pipeline.

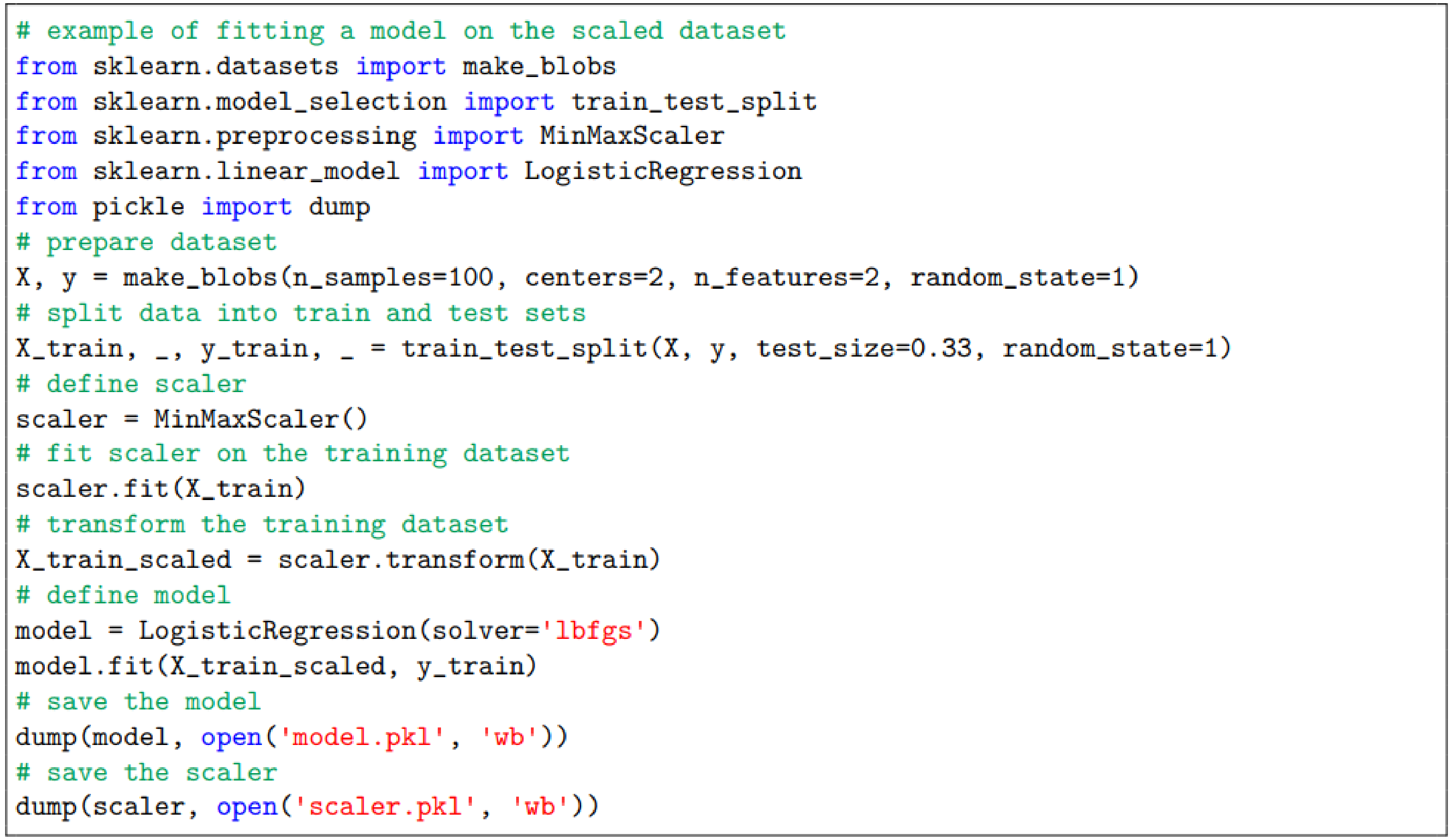
```
...
# define the model cross-validation configuration
cv = KFold(n_splits=10, shuffle=True, random_state=1)
# evaluate the pipeline using cross-validation and calculate MAE
scores = cross_val_score(pipeline, X, y, scoring='neg_mean_absolute_error', cv=cv,
    n_jobs=-1)
# convert MAE scores to positive values
scores = absolute(scores)
# summarize the model performance
print('MAE: %.3f (%.3f)' % (mean(scores), std(scores)))
```

How to Save and Load Data Transform

- It is critical that any data preparation performed on a training dataset is also performed on a new dataset in the future
- The correct solution to preparing new data for the model in the future is to also save any data preparation objects, like data scaling methods, to file along with the model

How to Save and Load Data Transform

Save Data Preparation Objects



```
# example of fitting a model on the scaled dataset
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from pickle import dump
# prepare dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# split data into train and test sets
X_train, _, y_train, _ = train_test_split(X, y, test_size=0.33, random_state=1)
# define scaler
scaler = MinMaxScaler()
# fit scaler on the training dataset
scaler.fit(X_train)
# transform the training dataset
X_train_scaled = scaler.transform(X_train)
# define model
model = LogisticRegression(solver='lbfgs')
model.fit(X_train_scaled, y_train)
# save the model
dump(model, open('model.pkl', 'wb'))
# save the scaler
dump(scaler, open('scaler.pkl', 'wb'))
```

Listing 26.5: Example of fitting a model on the scaled data and saving the model and scaler objects.

How to Save and Load Data Transform

Load Data Preparation Objects

```
# load model and scaler and make predictions on new data
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from pickle import load
# prepare dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# split data into train and test sets
_, X_test, _, y_test = train_test_split(X, y, test_size=0.33, random_state=1)
# load the model
model = load(open('model.pkl', 'rb'))
# load the scaler
scaler = load(open('scaler.pkl', 'rb'))
# check scale of the test set before scaling
print('Raw test set range')
for i in range(X_test.shape[1]):
  print('>%d, min=%.3f, max=%.3f' % (i, X_test[:, i].min(), X_test[:, i].max()))
# transform the test dataset
X_test_scaled = scaler.transform(X_test)
print('Scaled test set range')
for i in range(X_test_scaled.shape[1]):
  print('>%d, min=%.3f, max=%.3f' % (i, X_test_scaled[:, i].min(), X_test_scaled[:,
      i].max()))
# make predictions on the test set
yhat = model.predict(X_test_scaled)
# evaluate accuracy
acc = accuracy_score(y_test, yhat)
print('Test Accuracy:', acc)
```

Listing 26.6: Example of loading the saved model and scaler objects and use them on new data.