**List Comprehension**

new\_list = [expression for item in iterable if condition]

for i, j in d.items():

print(i, j)

**Output:**

A [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

B [1, 2, 3, 4, 5]

C [1, 2, 3]

In Pandas, you can handle missing data **(NaN values)** using various methods. Here are the methods provided by Pandas to deal with NaN values:

isna() / isnull(): Returns a boolean mask indicating missing values (True for NaN values, False for non-NaN values).

notna() / notnull(): Returns a boolean mask indicating non-missing values (False for NaN values, True for non-NaN values).

dropna(): Removes rows containing NaN values.

fillna(): Fills NaN values with specified values.

replace(): Replaces specified values (e.g., NaN) with other values.

interpolate(): Performs linear interpolation to fill NaN values with estimated values based on adjacent data points.

ffill() / pad(): Forward fills NaN values with the previous valid value along the specified axis.

bfill() / backfill(): Backward fills NaN values with the next valid value along the specified axis.

fillna() with method='ffill' or method='bfill': Forward or backward fill NaN values.

fillna() with method='nearest': Fills NaN values with the nearest valid value along the specified axis.

error=actual−predicted

from sklearn.metrics import mean\_absolute\_error

from sklearn.model\_selection import train\_test\_split

The purpose of using the **"random\_state"** parameter is to ensure that the results of a random operation are **consistent across different runs of the program**. Random operations will produce the same results every time they are executed with the same "random\_state" value. In other words, setting a fixed "random\_state" ensures that the random component behaves predictably and reproducibly, leading to consistent outcomes across different runs of the program.

iloc is used to pass indices and used with small brackets

loc is used to pass actual index names and column names

groupby Is same as SQL goup by

sort\_values sort the df by column name

rename (rename columns by passing a dictionary)

Three Approaches for missing values:

1. A Simple Option: Drop Columns with Missing Values

The simplest option is to drop columns with missing values.

1. A Better Option: Imputation

Imputation fills in the missing values with some number. For instance, we can fill in the mean value along each column.

1. An Extension To Imputation

Imputation is the standard approach, and it usually works well. However, imputed values may be systematically above or below their actual values (which weren't collected in the dataset). Or rows with missing values may be unique in some other way. In that case, your model would make better predictions by considering which values were originally missing.

from sklearn.impute import SimpleImputer

# Imputation

my\_imputer = SimpleImputer()

imputed\_X\_train = pd.DataFrame(my\_imputer.fit\_transform(X\_train))

imputed\_X\_valid = pd.DataFrame(my\_imputer.transform(X\_valid))

# Imputation removed column names; put them back

imputed\_X\_train.columns = X\_train.columns

imputed\_X\_valid.columns = X\_valid.columns

Three Approaches for categorical values

1. Drop Categorical Variables

The easiest approach to dealing with categorical variables is to simply remove them from the dataset. This approach will only work well if the columns did not contain useful information.

1. Ordinal Encoding

Ordinal encoding assigns each unique value to a different integer.

This approach assumes an ordering of the categories: "Never" (0) < "Rarely" (1) < "Most days" (2) < "Every day" (3).

This assumption makes sense in this example, because there is an indisputable ranking to the categories. Not all categorical variables have a clear ordering in the values, but we refer to those that do as ordinal variables. For tree-based models (like decision trees and random forests), you can expect ordinal encoding to work well with ordinal variables.

from sklearn.preprocessing import OrdinalEncoder

# Make copy to avoid changing original data

label\_X\_train = X\_train.copy()

label\_X\_valid = X\_valid.copy()

# Apply ordinal encoder to each column with categorical data

ordinal\_encoder = OrdinalEncoder()

label\_X\_train[object\_cols] = ordinal\_encoder.fit\_transform(X\_train[object\_cols])

label\_X\_valid[object\_cols] = ordinal\_encoder.transform(X\_valid[object\_cols])

1. One-Hot Encoding

One-hot encoding creates new columns indicating the presence (or absence) of each possible value in the original data. To understand this, we'll work through an example.

from sklearn.preprocessing import OneHotEncoder

# Apply one-hot encoder to each column with categorical data

OH\_encoder = OneHotEncoder(handle\_unknown='ignore', sparse=False)

OH\_cols\_train = pd.DataFrame(OH\_encoder.fit\_transform(X\_train[object\_cols]))

OH\_cols\_valid = pd.DataFrame(OH\_encoder.transform(X\_valid[object\_cols]))

# One-hot encoding removed index; put it back

OH\_cols\_train.index = X\_train.index

OH\_cols\_valid.index = X\_valid.index

# Remove categorical columns (will replace with one-hot encoding)

num\_X\_train = X\_train.drop(object\_cols, axis=1)

num\_X\_valid = X\_valid.drop(object\_cols, axis=1)

# Add one-hot encoded columns to numerical features

OH\_X\_train = pd.concat([num\_X\_train, OH\_cols\_train], axis=1)

OH\_X\_valid = pd.concat([num\_X\_valid, OH\_cols\_valid], axis=1)

# Ensure all columns have string type

OH\_X\_train.columns = OH\_X\_train.columns.astype(str)

OH\_X\_valid.columns = OH\_X\_valid.columns.astype(str)

**Pipe Line**

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import OneHotEncoder

*# Preprocessing for numerical data*

numerical\_transformer = SimpleImputer(strategy='constant')

*# Preprocessing for categorical data*

categorical\_transformer = Pipeline(steps=[

('imputer', SimpleImputer(strategy='most\_frequent')),

('onehot', OneHotEncoder(handle\_unknown='ignore'))

])

*# Bundle preprocessing for numerical and categorical data*

preprocessor = ColumnTransformer(

transformers=[

('num', numerical\_transformer, numerical\_cols),

('cat', categorical\_transformer, categorical\_cols)

])

from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n\_estimators=100, random\_state=0)

from sklearn.metrics import mean\_absolute\_error

*# Bundle preprocessing and modeling code in a pipeline*

my\_pipeline = Pipeline(steps=[('preprocessor', preprocessor),

('model', model)

])

*# Preprocessing of training data, fit model*

my\_pipeline.fit(X\_train, y\_train)

*# Preprocessing of validation data, get predictions*

preds = my\_pipeline.predict(X\_valid)

*# Evaluate the model*

score = mean\_absolute\_error(y\_valid, preds)

print('MAE:', score)

Cross Validation

from sklearn.model\_selection import cross\_val\_score

*# Multiply by -1 since sklearn calculates \*negative\* MAE*

scores = -1 \* cross\_val\_score(my\_pipeline, X, y,

cv=5,

scoring='neg\_mean\_absolute\_error')

**XGBoost**

my\_model = XGBRegressor(n\_estimators=1000, learning\_rate=0.05, n\_jobs=4)

my\_model.fit(X\_train, y\_train,

early\_stopping\_rounds=5,

eval\_set=[(X\_valid, y\_valid)],

verbose=False)