

DSC510: Introduction to Data Science and Analytics

Lab 4: Data Preparation

Pavlos Antoniou
Office: B109, FST01



University of Cyprus
Department of
Computer Science



Data science project workflow

1. **Problem definition** – understand the business/research question.
2. **Data collection** – gather data from files, databases, APIs
3. **Exploratory Data Analysis (EDA)** – explore structure, summarize with statistics, visualize patterns, detect anomalies. – LAB2, LAB3
 - EDA is iterative: usually do a first pass on raw data (before data preparation) to understand issues, then clean, then explore again more reliably
4. **Data preparation** – fix/clean missing values, inconsistencies, errors, perform basic transformations. – LAB4, LAB5
5. **Modeling** – build and evaluate machine learning models. – LAB 6, LAB7, LAB8
6. **Deployment & monitoring/maintenance** – put the model into production, track performance and provide maintenance.

Step Before Data Preparation: Data Splitting



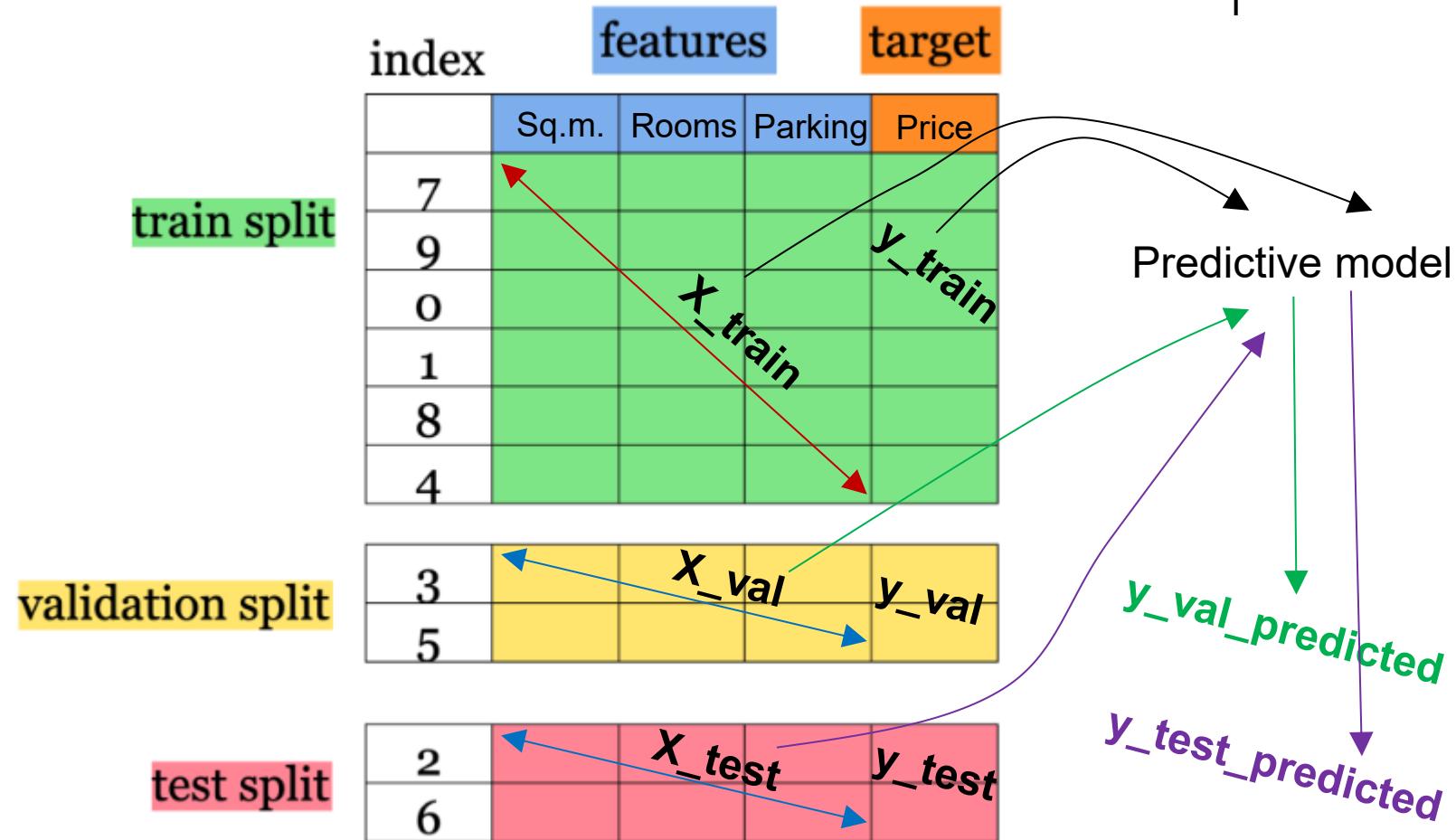
- A predictive model is an algorithm or mathematical framework that uses historical data to forecast future outcomes, events, or behaviors.
- 3 phases to prepare a predictive model: Training - Validation - Testing
 - Each phase serves a **different purpose**, and using the same data for all would lead to **data leakage and overfitting**
- **Split original dataset** into 3 smaller datasets
 - **Training dataset**: Dataset used to train the model and **evaluate parameters**. The model **sees** and **learns** from this data | 70-80%
 - If this is the only data the model ever sees, it may memorize it rather than generalize.
 - **Validation dataset**: Used to provide unbiased **evaluation of model**, **fine-tune** the model **hyperparameters*** and **compare different models**. The model occasionally sees this data, but **never “learns”** from this. | 10-15%
 - **Test dataset**: Used only at the end to get unbiased **evaluation of the final model** on unseen data. | 10-15%



Training / validation / test datasets

	X			y
index	features		target	
	Sq.m.	Rooms	Parking	Price
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				

Initial (original) dataset



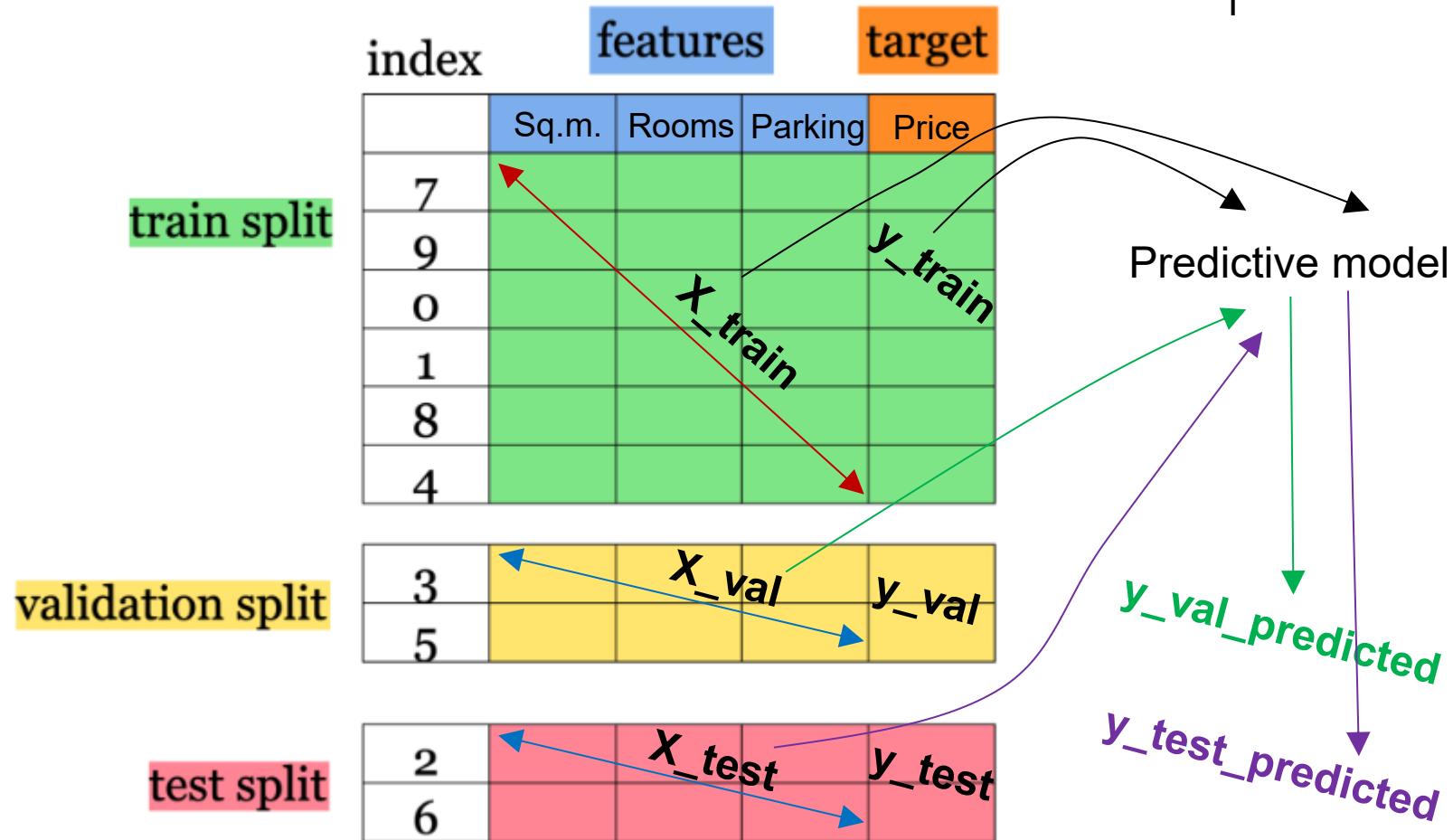
- During testing phase, predictive modelling technique sees both features (X_{train}) and the target (y_{train}) values
- During validation and testing phases, only features (X_{val} , X_{test} respectively) are given as input to predictive technique so as to predict the target values. Predictive model is evaluated on its effectiveness to correctly predict the target values by comparing the predicted with the original target values.



Cross-validation (CV)

	X			y
index	features			target
	Sq.m.	Rooms	Parking	Price
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				

Initial (original) dataset



- Shuffling & splitting is not performed just once, but multiple times → Cross validation (CV) process
- Cross-validation is a model evaluation technique where the dataset is shuffled and split multiple times into training and validation subsets. The model is trained and tested repeatedly on these different splits to ensure that model performance is consistent and not dependent on a particular data division.



Cross-validation (CV)

- If cross-validation (CV) is used for model selection and hyperparameter tuning (as is often the case), a separate validation set *is not needed*
 - In this case, the original dataset is typically split into train and test sets only (e.g. 80% / 20%).
 - During cross-validation (e.g., K-fold CV), the *training set is internally partitioned* into multiple training and validation folds to evaluate and tune models.
 - The **test set** remains completely unseen until the final evaluation of the selected model.
-

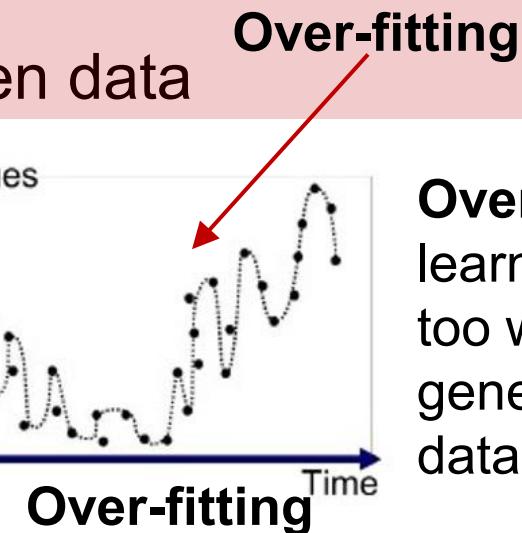
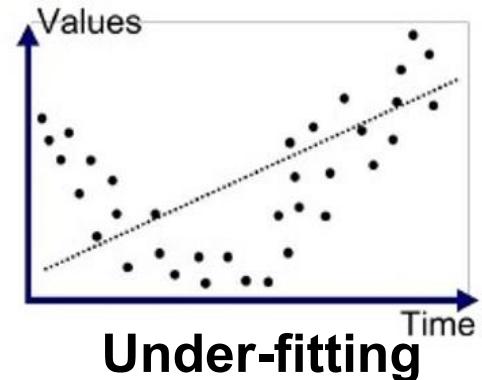


Splitting datasets against overfitting

- **Training** a predictive modelling technique and **evaluating** its performance on the **same data** a methodological **mistake** because may lead to:

- High accuracy on seen data
- Low accuracy (fail to predict / classify) on unseen data

Under-fitting: model fails to capture underlying pattern in training data



Over-fitting: model learns training data too well but fails to generalize to new data

- Splitting dataset can prevent overfitting and ensures that model generalizes well on unseen data



When to split a dataset? Why?

- Splitting the dataset should typically be **one of the first steps** in a data science project, before performing any data preprocessing and transformations
- Why split the dataset early?
 - Avoid **data leakage**: avoid passing information from test (unseen) set to training set
 - Example: Filling missing values using information from the entire dataset (e.g., the column mean) can leak information from the test set into the training set, potentially biasing the model.
 - Realistic evaluation: **test dataset should simulate new, truly unseen data** (as in real life scenarios)

Best practices in a data science project



- Initial data exploration – Lab2, Lab3
 - Perform exploratory data analysis (EDA) to understand the structure of the dataset
 - This includes understanding data types, basic statistics (e.g. mean, std, median), feature/target distributions, checking for missing values and data inconsistencies, getting an initial sense of the data
- Split the dataset
 - Split the data into training and test sets
- Data preprocessing – Lab4
 - After splitting, perform all preprocessing steps (such as missing values imputation, scaling, encoding) separately on each dataset
 - **Fit (train) the preprocessing tools** (like imputers, scalers and encoders) **on the training set** and then **apply these fitted tools to the test set**.

Best practices in a data science project



- Feature Engineering (Selection, Extraction) – Lab5
 - Conduct feature engineering (feature selection / extraction) based solely on the training data
 - Apply the same feature transformations to the test set
- Model Training and Tuning – Lab6, Lab7, Lab8
 - Train your models using the training set using cross-validation
 - Use the validation set to tune hyperparameters and select the best model
 - Finally, evaluate the final best model on the test set to get an unbiased estimate of its performance



Prepare data for machine learning

- Data preparation is the process of making raw data ready for analysis and predictive modeling. It involves two main steps:
 - **1. Preprocess**
 - **Clean:** fix, delete, or fill missing values.
 - **Encode:** convert categorical/text data into numerical form.
 - **Resample:** adjust the frequency of observations (important for time-series).
 - **2. Transform**
 - **Scale:** normalize or standardize features.
 - **Unskew:** adjust distributions to be more symmetrical.
 - **Feature Engineering:** select or extract features to improve model performance (covered in the next lab).



Cleaning data

- Fixing Data Formats
 - Mixed or inconsistent formats
 - Data from international sources may have varying formats that don't match expected numeric or date conventions.
 - Numeric issues
 - Decimal separator may be a comma → convert to dot (.)
 - Thousands separators may exist → remove them
 - Monetary symbols before or after numbers → strip them
 - Date/time issues
 - Dates may appear as integers (e.g., 20090609231247) instead of ISO 8601 (2009-06-09 23:12:47)
 - Transformation is needed to match Python's expected formats for plotting and analysis



Cleaning data

- **Deleting Missing Values**
 - When to delete rows:
 - If the number of missing-value rows is small relative to the dataset
 - Example: less than 10% of all rows
 - If the rows do not contain important information
 - Example: missing a non-critical category
 - If the target variable is missing
 - You cannot train a supervised ML model without a target label
 - When NOT to delete rows:
 - Missing values may carry important information
 - Example: missing ages may correspond to privacy-conscious users, relevant for decision-making
 - Key point:
 - Deleting missing values is not trivial, especially for unfamiliar datasets
 - Implementation:
 - Can be done in Python using `pandas.dropna()` (see next slides)



Cleaning data

- **Filling Missing Values**
 - Usually done on each column separately
 - Categorical data (e.g., country): use “Unknown”, or most frequent
 - Numerical data (e.g., age):
 - Use aggregation function (e.g. mean, median) on all non-missing values of the column
 - or aggregation function on similar category values (based on the target variable)
 - Time-series data: use interpolation (see resampling in Lab11)
 - Predictive imputation: build a model to predict missing values
- **Correcting Erroneous Values**
 - Detect invalid values with statistical analysis or visualization (e.g., box plots)
 - Examples:
 - Non-numeric value in a numeric column
 - Age < 0 or > 110
 - Strategy: recode or delete → then treat as missing values



Cleaning data

- **Handling Outliers** (values that significantly deviate from the majority)
 - Can distort models (e.g., regression, k-NN – especially distance-based models – will be discussed in another lab)
 - Removing them should be done carefully → sometimes they carry meaningful information
 - See [Appendix](#) for outlier removal techniques
- **Standardizing Categories**
 - User-entered data often contains inconsistencies:
 - Spelling mistakes, language differences, multiple formats
 - Examples:
 - Country → USA, United States, U.S.
 - Dates → 1982-10-01, 1/10/1982
 - Goal: ensure one consistent version per value



Missing values manipulation

- Methods to deal with missing values in the data frame:

df.method()	description
dropna()	Drops observations (rows) with at least one missing value
dropna(axis=1)	Drops the columns where at least one value is missing
dropna(thresh = 5)	Drops rows that contain less than 5 non-missing values
fillna(0)	Replaces missing values with a specified value
ffill()	Replace missing values by propagating the last valid observation to next valid
bfill()	Replace missing values by using the next valid observation to fill the gap
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values



Load dataset

- Synthetic dataset with different data types (numerical, categorical, dates) and missing values.

```
In [ ]: df = pd.read_csv('synthetic dataset.csv', parse_dates=['Join_Date'])
df.head()
```

Out []:

	ID	Name	Age	Degree	Score	Salary	Hours	Error_Rate	Satisfaction	Join_Date	Sales
0	1	Aaron	24.0	Master	79.0	93297.0	51.7	NaN	0.64	2023-07-22 08:43:44	3385.0
1	2	Caroline	41.0	Bachelor	55.0	26012.0	47.1	NaN	0.61	2023-11-10 22:30:09	1982.0
2	3	Briana	30.0	Bachelor	50.0	56059.0	NaN	NaN	0.61	2023-06-14 13:30:15	1014.0
3	4	Michael	49.0	Master	51.0	88910.0	46.2	NaN	0.58	2022-12-18 15:31:34	2696.0
4	5	Andrew	58.0	PhD	73.0	72290.0	42.3	NaN	0.64	2024-04-05 03:26:19	667.0

- `read_csv()` does not recognize dates by default; they are loaded as strings → `parse_dates=['Join_Date']` tells pandas “When reading this column, try to convert it into `datetime64`”



Missing values manipulation

- Evaluate the percentage of missing values per column:

```
In [ ]: # isnull() converts missing values to True, False otherwise; mean()
works on Boolean values, treating True=1 and False=0, then multiplying
with 100 we can find the percentage of 1s
missing_values_count = df.isnull().mean()*100
# look at the # of missing points in the first ten columns
missing_values_count
```

Out[]:

ID	0.0
Name	0.0
Age	7.4
Degree	0.0
Score	5.0
Salary	4.2
Hours	3.0
Error_Rate	79.2
Satisfaction	4.4
Join_Date	5.2
Sales	0.0

- The more missing values a feature (column) has, the less reliable the data in that column might be → feature is less important
- Knowing how many missing values exist helps assess whether to keep, impute, or drop those columns
 - If a column or row has too many missing values (e.g. Error_Rate), you might consider dropping it from the analysis, as it could contribute little to the model's accuracy

```
df = df.drop(columns=['ID', 'Error_Rate'])
```



Useless column



Drop columns with missing values

```
In [ ]: # remove all columns with at least one missing value  
columns_cleaned = df.dropna(axis=1)  
columns_cleaned.head()
```

```
Out[ ]:
```

	Name	Degree	Sales
0	Aaron	Master	3385.0
1	Caroline	Bachelor	1982.0
2	Briana	Bachelor	1014.0
3	Michael	Master	2696.0
4	Andrew	PhD	667.0

```
In [ ]: # just how much data did we lose?  
print("Columns in original dataset:", df.shape[1])  
print("Columns with na's dropped:", columns_cleaned.shape[1])
```

```
Out[ ]: Columns in original dataset: 9  
Columns with na's dropped: 3
```

We can specify which columns to check for missing values and drop only the rows where those columns contain NaN

```
df = df.dropna(subset=['down', 'Sideoffield'])
```



Fill in missing values

- One option we have is to specify what we want the NaN values to be replaced with

```
In [ ]: # replace all NA's with 0 (save in a new dataframe)
df_zero = df.fillna(0)
# replace all NA's with 0 for a specific column (replace column)
df['Age'] = df['Age'].fillna(0)
```

- Another option is to replace missing values with the first valid value comes after it in the same column
 - This makes a lot of sense for datasets where the observations have some sort of logical order

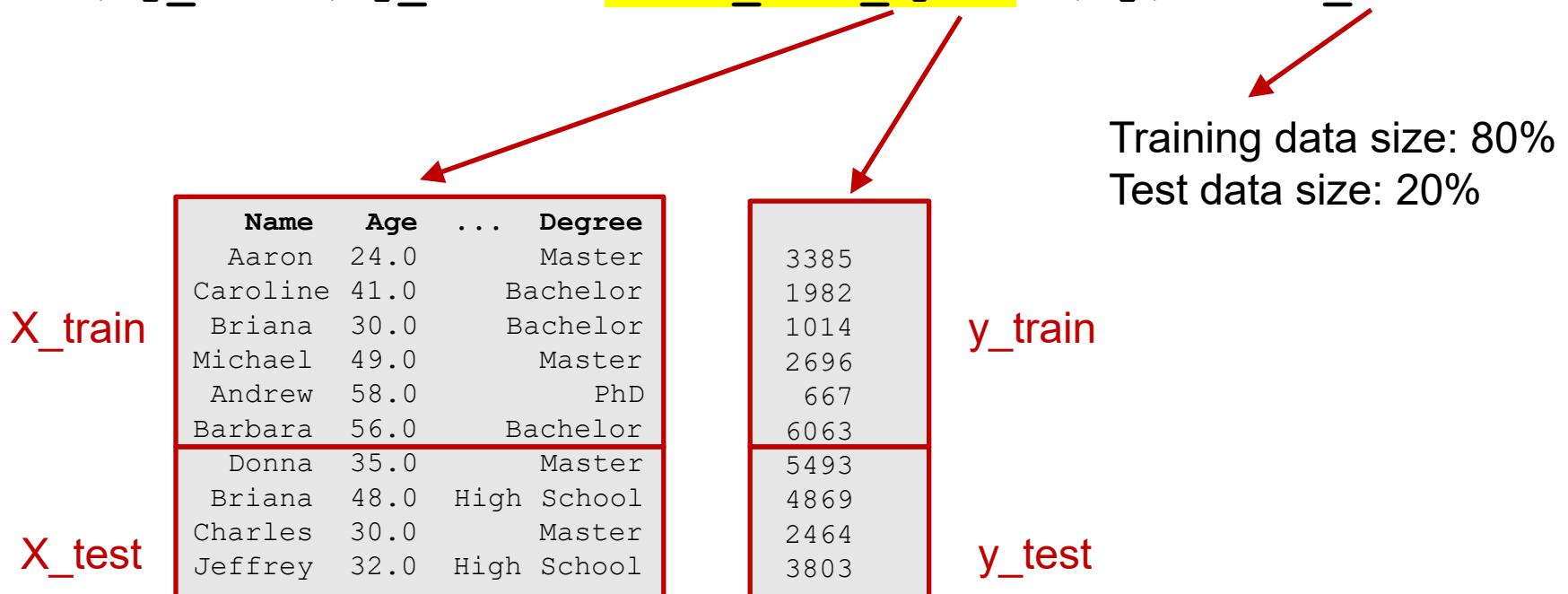
```
In [ ]: # replace all NA's the first valid value that comes after it in the
        # same column, then replace all the remaining na's (if any) with 0
df = df.bfill(axis=0).fillna(0)
```

- filling all missing values with a constant (like zero) does *not* cause data leakage – constant value carries no information from unseen data

Data splitting



```
# features  
X = df.drop(columns=[ 'Sales' ] )  
# target  
y = df['Sales']  
  
from sklearn.model_selection import train_test_split  
x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=0.80)
```





Fill in missing values with imputation

- Imputation fills in the missing value with some number
 - Imputed value won't be exactly right in most cases, but it usually gives more accurate models than dropping the column entirely

```
In [ ]: from sklearn.impute import SimpleImputer # Using Sklearn's simple imputer
import numpy as np
my_imputer = SimpleImputer(missing_values=np.Nan, strategy='mean')
X_train[['Age']] = my_imputer.fit_transform(X_train[['Age']])
X_test[['Age']] = my_imputer.transform(X_test[['Age']])
```

- SimpleImputer's basic two arguments: missing_values and strategy
 - Strategy can be set to mean, median, most_frequent, constant (with fill_value argument)
 - Numerical missing values: mean, median, most frequent, constant
 - Categorical missing values: most frequent, constant
 - fit_transform is applied only the training dataset to both calculate mean (fit) only from training data and then fill (transform) missing values
 - transform is applied on test dataset to fill (transform) missing values with the mean calculated on the training dataset



Encoding categorical data

- Machine learning models require all features to be numerical
- Categorical text-based data (e.g. a column with “male”, “female” values) must be encoded to numbers
- Popular techniques:
 - Label Encoding
 - Ordinal Encoding
 - One-Hot Encoding (or Dummy Variable Encoding)
 - Effect Encoding
 - Bin counting
 - Feature Hashing
- Scikit-learn lib involves a few encoders but **category_encoders** lib has more with useful properties
`conda install -c conda-forge category_encoders`
(website: http://contrib.scikit-learn.org/category_encoders)



Label Encoding

- Assigns a unique integer value to each category of a categorical feature
 - e.g. "Red" → 0, "Green" → 1, "Blue" → 2
- Used for **nominal** (unordered) **categorical features**
 - numbers assigned are arbitrary - don't represent ranking or size
- Potential issue: implies ordinal relationships between categories
 - e.g. Red (0) seems to be closer to Green (1) than to Blue (2)
 - high ordinal values possess higher “weight” and may be considered of higher importance especially in distance-based ML techniques



Ordinal Encoding

- Used for **ordinal categorical features** (with natural, ordered values) and when **retaining the order is important**
- Similar to label encoding, but numbers assigned reflect the sequence (based on the category order)

```
x_train[ ['Degree' ] ].value_counts()
```

```
Degree
Master      108
Bachelor    102
High School 99
PhD         91
Name: count, dtype: int64
```

Natural order:
'High School':1
'Bachelor':2
'Master':3
'PhD':4

Label & Ordinal Encoding: OrdinalEncoder



```
import category_encoders as ce
import pandas as pd

# create object of Ordinal encoding
ordinal_encoder = ce.OrdinalEncoder( mapping=[{'col':'Degree', 'mapping':{'None':0,'High School':1,'Bachelor':2,'Master':3,'PhD':4}}] )
```

```
#fit and transform training data and then transform test
X_train['Degree_Ordinal'] = ordinal_encoder.fit_transform(X_train['Degree'])
X_train[['Degree', 'Degree_Ordinal']].head()
```

	Degree	Degree_Ordinal
249	High School	1
433	PhD	4
19	Master	3
322	Bachelor	2
332	High School	1

Note: If no mapping is given, order is automatically chosen by the encoder → **Label encoding**

```
X_test['Degree_Ordinal'] = ordinal_encoder.transform(X_test['Degree'])
```



One-Hot Encoding

- Used for **nominal** (unordered) **categorical features**.
 - Prevents imposing false orderings (as happens with label encoding).
 - Creates a new binary feature (column) for each category value.
 - 1 = category present
 - 0 = category absent
 - These binary features are called dummy variables.
 - Number of dummy variables = number of categories in the original feature.
 - Preferred over label encoding for distance-based models.
-

One-Hot Encoding: OneHotEncoder



```
# Create object for One-hot encoding
onehot_encoder=ce.OneHotEncoder(cols=['Degree'], use_cat_names=True)
#fit and transform training data
X_train_onehot = onehot_encoder.fit_transform(X_train)
X_train_onehot.head()
```

a list of columns to encode, if None, all string columns will be encoded

Dummy variables

	Name	Age	Degree_High School	Degree_PhD	Degree_Master	Degree_Bachelor	Score	Salary	Hours	Satisfaction	Join_Date	Degree_Ordinal
249	Wanda	52.0		1	0	0	0	96.0	80576.0	47.1	0.89	2025-05-14 23:58:54
433	James	25.0		0	1	0	0	60.0	34716.0	32.4	0.75	2024-02-02 21:10:37
19	Christopher	25.0		0	0	1	0	88.0	76532.0	47.3	0.83	2025-05-07 23:32:30
322	Robert	39.0		0	0	0	1	74.0	72290.0	51.7	0.98	2023-03-12 10:19:35
332	Edward	43.0		1	0	0	0	51.0	37347.0	55.7	0.57	2022-11-17 17:43:21



Drawbacks of One-Hot Encoding

- Creates **one new column per category value** → many columns for high-cardinality features (with high number of unique values)
 - Results in **high-dimensional datasets** (with very high number of features) when multiple categorical features are present
 - Can **slow model training and hurt performance** due to increased computational cost
-



Cyclical feature encoding

- For time-dependent features (e.g., month, day, hour), standard numeric encoding can be misleading.
 - Example: in the hour feature, 0 (midnight) and 23 (11 PM) are numerically far apart, but in reality they are adjacent in time.
- Step 1: Extract time components from datetime (month, day, hour, minute, weekday, etc.) – see [here](#)
- Step 2: Apply cyclical encoding → map each value onto the unit circle using sine and cosine: $x = \sin(2\pi * \text{value} / \text{max_value})$, $y = \cos(2\pi * \text{value} / \text{max_value})$
- This encoding preserves the cyclical nature of time (e.g., hours, days, months).

	datetime	temperature	hour
9	2012-10-01 21:00:00	12.776627	21
10	2012-10-01 22:00:00	12.789767	22
11	2012-10-01 23:00:00	12.802906	23
12	2012-10-02 00:00:00	12.816046	0
13	2012-10-02 01:00:00	12.829185	1

```
data['hour_sin'] = np.sin(2 * np.pi * data['hour']/23.0)  
data['hour_cos'] = np.cos(2 * np.pi * data['hour']/23.0)
```

	datetime	temperature	hour	hour_sin	hour_cos
10	2012-10-01 22:00:00	12.789767	22	-2.697968e-01	0.962917
11	2012-10-01 23:00:00	12.802906	23	-2.449294e-16	1.000000
12	2012-10-02 00:00:00	12.816046	0	0.000000e+00	1.000000
13	2012-10-02 01:00:00	12.829185	1	2.697968e-01	0.962917

11 PM is close to 12 midnight in terms of sin and cos



Data Transformation: Scaling data

- Feature rescaling
 - Some classification/regression/clustering techniques (see next slide) use the notion of distance (e.g. Euclidean) to measure similarity between 2 observations
 - Example
 - Classify houses with 2 features
 - $x_1 = \text{size} (0 - 2000m^2)$
 - $x_2 = \text{number of bedrooms} (1 - 5)$
 - $\text{Euclidean distance}(X_1, X_2) = \sqrt{(523 - 127)^2 + (4 - 2)^2}$
 - Distance is governed by features having boarder range of values
 - When distance is used by algorithms **make sure features are on a similar scale**
 - Target variable is not necessary to be scaled

$$X = \begin{bmatrix} 523 & 4 \\ 127 & 2 \\ 25 & 1 \end{bmatrix}$$

Feature x_1 with high magnitudes weights a lot more (dominates) in the distance calculations than feature x_2 with lower magnitudes



Data Transformation: Scaling data

- Some examples of algorithms where feature scaling matters are:
 - k-nearest neighbors (**kNN**) for classification uses Euclidean distance
 - k-means for clustering uses Euclidean distance
 - gradient descent/ascent-based optimization used in **logistic regression**, **Support Vector Machines (SVMs)**, neural networks etc.
 - Weights for features with higher magnitudes will update much faster than others
 - linear discriminant analysis (**LDA**), principal component analysis (**PCA**)
 - you want to find directions of maximizing the variance (under the constraints that those directions/eigenvectors/principal components are orthogonal)
- Decision trees and ensembles of trees are unaffected by the scale of feature variables. Examples:
 - bagging like RandomForest
 - boosting like AdaBoost, Gradient Boosting, XGBoost, LightGBM, CatBoost



Data Transformation: Scaling data

- Feature normalization
 - Rescales **each feature** individually into a given range, e.g. [0, 1]

$$x = \begin{bmatrix} 4 & 13 \\ 3 & 2 \\ 7 & 8 \end{bmatrix}, x_{i,j,desc} = \frac{x_{i,j} - \min(x_j)}{\max(x_j) - \min(x_j)} \Rightarrow x_{desc} = \begin{bmatrix} 0.25 & 1 \\ 0 & 0 \\ 1 & 0.55 \end{bmatrix}$$

- Scikit-learn module: [MinMaxScaler](#) or [MaxAbsScaler](#)
 - MinMaxScaler: Transforms features by scaling each feature to a given range ($x_{\min} \rightarrow x_{\max}$).

```
from sklearn.preprocessing import MinMaxScaler
# create the scaler object
scaler = MinMaxScaler(feature_range=(0, 1))
X_train_scaled = X_train.copy() # copy of the original training dataset
columns_to_scale = ['Age', 'Score', 'Hours']
# train the scaler (find min and max) on the training dataset
scaler.fit(X_train[columns_to_scale])
# scale the dataset (apply the transformation)
X_train_scaled[columns_to_scale] = scaler.transform(X_train[columns_to_scale])
```



Data Transformation: Scaling data

- Feature normalization
 - Rescales **each feature** individually into a given range, e.g. [0, 1]

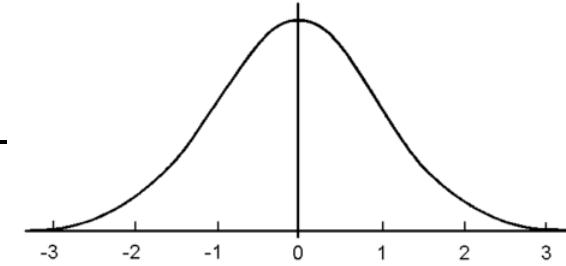
$$x = \begin{bmatrix} 4 & 13 \\ 3 & 2 \\ 7 & 8 \end{bmatrix}, x_{i,j,desc} = \frac{x_{i,j} - \min(x_j)}{\max(x_j) - \min(x_j)} \Rightarrow x_{desc} = \begin{bmatrix} 0.25 & 1 \\ 0 & 0 \\ 1 & 0.55 \end{bmatrix}$$

- Scikit-learn module: [MinMaxScaler](#) or [MaxAbsScaler](#)
 - MinMaxScaler: Transforms features by scaling each feature to a given range ($x_{\min} \rightarrow x_{\max}$).

```
from sklearn.preprocessing import MinMaxScaler
# create the scaler object
scaler = MinMaxScaler(feature_range=(0, 1))
X_train_scaled = X_train.copy() # copy of the original training dataset
columns_to_scale = ['Age', 'Score', 'Hours']
# train the scaler (find min and max) and scale the training dataset
X_train_scaled[columns_to_scale] = scaler.fit_transform(X_train[columns_to_scale])
```

On `X_train` we can apply `.fit_transform()` instead of `.fit()` and `.transform()` separately.
But on `X_test`, only `.transform()` must be applied.

Data Transformation: Scaling



- Feature standardization
 - Rescales **each feature individually** to make values have zero mean ($\mu = 0$) and unit variance ($\sigma^2 = 1$)
$$x = \begin{bmatrix} 4 & 13 \\ 3 & 2 \\ 7 & 8 \end{bmatrix}, x_{i,j, std} = \frac{x_{i,j} - \text{mean}(x_j)}{\sigma} \Rightarrow x_{std} = \begin{bmatrix} -0.39 & 1.86 \\ -0.98 & -1.226 \\ 1.37 & 0.07 \end{bmatrix}$$
 - Centers values around zero and adjusts their spread so that variance is 1
 - Benefits features that are approximately normally distributed (gaussian)
 - Useful for distance-based algorithms such as the SVM (RBF kernel) and when using gradient-based optimization methods
 - Scikit-learn module: [StandardScaler](#)

```
from sklearn.preprocessing import StandardScaler
sscaler = StandardScaler()
X_train_standard_scaled = X_train.copy()
# train the standardizer (find mean, std) and standardize the dataset
X_train_standard_scaled[columns_to_scale] = sscale.fit_transform(X_train[columns_to_scale])
```



Data Transformation: Scaling data

- Feature robust standardization
 - When data contains outliers, mean value and variance used by the Standard Scaler can distort the rescaled values
 - MinMaxScaler is also sensitive to the presence of outliers as well
 - Robust standardization is to rescale **each feature individually** to make values have zero median (median=0) and unit interquartile range (IQR=1)
 - Centers values around 25th and 75th percentiles (within the IQR)
 - Benefits features with non-gaussian distributions, particularly those with outliers or skewed (long-tailed) distributions
 - Scikit-learn module: [RobustScaler](#)

```
from sklearn.preprocessing import RobustScaler
rscaler = RobustScaler()
X_train_robust_scaled = X_train.copy()
# train the robust scaler (find median, quantiles) and robust scale the dataset
X_train_robust_scaled[columns_to_scale] = rscaler.fit_transform(X_train[columns_to_scale])
```

[Compare the effect of different scalers on data with outliers](#)

When to normalize or standardize features?



- When features have different scales (e.g., age in years vs. income in thousands) & distance-based algorithms (e.g., k-NN, SVM) or gradient-based algorithms (e.g., logistic/linear regression) will be used
 - Normalization (MinMaxScaler):
 - Works best for features with different ranges and few or no significant outliers
 - Standardization (StandardScaler):
 - Works well for features with approximately normal distributions and few/no influential outliers
 - Robust Scaling (RobustScaler):
 - Best for features with many outliers or skewed (long-tailed) distributions
-

Scale or normalize label / ordinal encoded data when using distance-based algorithms?



- Depends on the nature of your data
- When to scale:
 - Encoded feature has many levels → prevents it from dominating
 - Ordinal encoding with meaningful, evenly spaced values (e.g., 1-5 survey ratings) → can use scaling that preserves relationships (e.g. MinMax scaling)
- When not to scale:
 - Purely categorical / arbitrary values without natural order (e.g., colors 0-2)
 - Scaling such values (getting them closer or further) can imply a relationship that doesn't exist

Scale or normalize one-hot encoded data when using distance-based algorithms?



- Scaling features being one-hot-encoded **is not recommended**
 - Binary values don't require scaling
 - One-hot encoding produces binary (0/1) columns. These already express presence (1) or absence (0) clearly. Scaling adds no meaningful information.
 - Scaling distorts categorical meaning
 - Scaling would turn 0/1 values into continuous values (e.g., 0.5) that have no valid categorical interpretation.
-



When to normalize or standardize target?

- Usually **not required**, but can help in regression tasks
- When to scale:
 - Target has a wide value range (e.g., income in hundreds to millions) → prevents large values from dominating
 - Using gradient descent-based models (e.g., linear regression) → speeds up convergence



Data Transformation: Scaling data

- Normalize observations (rows)
 - Normalize **each observation** (row) independently of other rows so that its norm ($\|1\|$ or $\|2\|$) equals 1

$$x = \begin{bmatrix} 4 & 13 \\ 3 & 2 \\ 7 & 8 \end{bmatrix}, x_{i,j,norm} = \frac{x_{i,j}}{\sqrt{\sum_{k=0}^n x_{i,k}^2}} \Rightarrow x_{norm} = \begin{bmatrix} 0.29 & 0.96 \\ 0.83 & 0.55 \\ 0.66 & 0.75 \end{bmatrix}$$

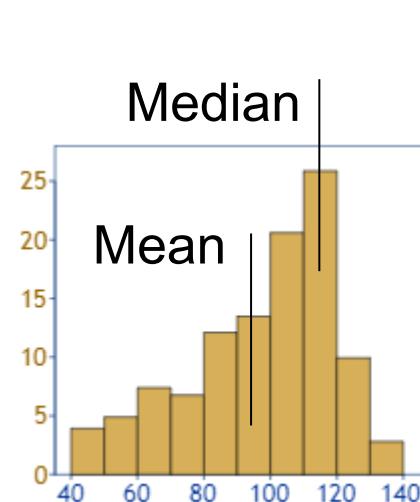
$\sqrt{0.29^2 + 0.96^2} = 1$

- Normalizing to unit norm helps ensure that each row contributes equally to the distance metrics used in algorithms, preventing any single row from disproportionately affecting the results.
 - Useful for sparse datasets (lots of zeros) to prevent zeros from skewing data
- Commonly used in text classification or clustering
 - dot product of two $\|2\|$ -normalized TF-IDF vectors is the cosine similarity of the vectors and is the base similarity metric for the Vector Space Model
- Scikit-learn module: [Normalizer](#)

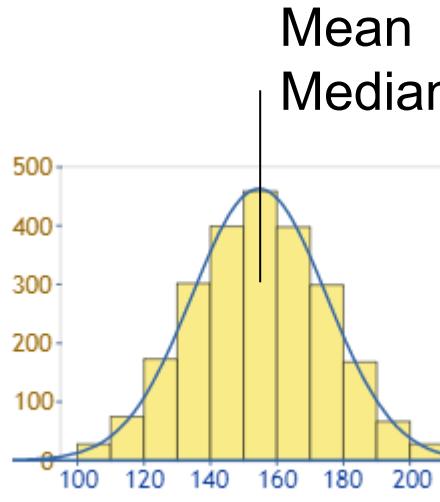


Data Transformation: Unskewing data

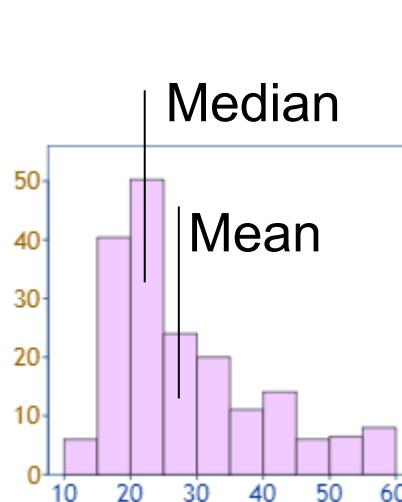
- A variable is skewed when its distribution curve is asymmetrical as compared to a normal distribution curve that is perfectly symmetrical
- *Skewness* is the measure of the asymmetry
 - The skewness for a gaussian or normal distribution is 0



Left / negative skew:
Long tail is on the left / negative side of the peak



No skew (symmetrical distribution)

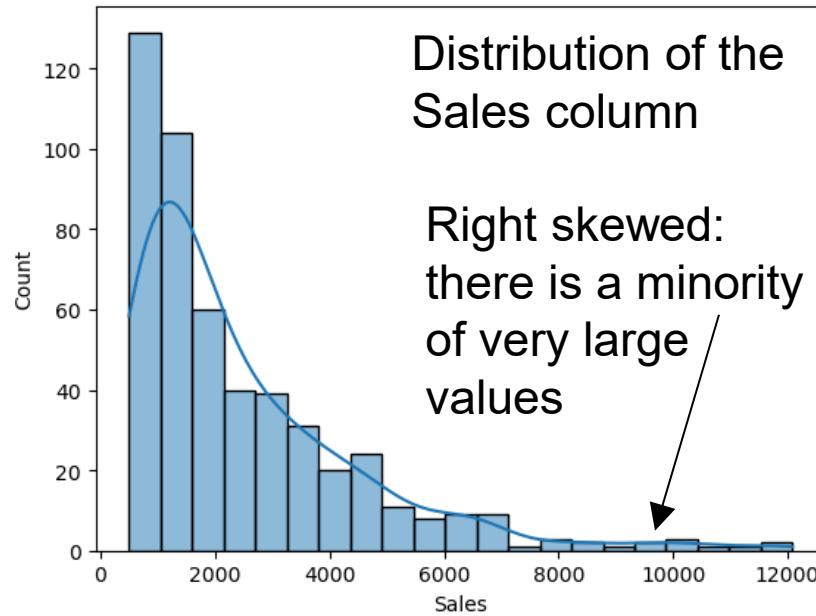


Right / positive skew:
Long tail is on the right / positive side of the peak



Effects of skewed data

- Skewness of (feature or target) variable may degrade the predictive model's ability to predict values towards the long tail side



- A regression model for predicting Sales or using Sales as input feature will be **trained on a much larger number of low and moderate Sales values** and will be **less likely** to successfully **predict the Sales values of high-performing individuals**



Unskewing transformations

- **Goal:** Transform skewed data to be more symmetric (Gaussian)
- **Why:** Some models (Linear/Logistic Regression, SVM, Gaussian NB) work better with near-Gaussian data
- **Not needed for:** Tree-based models (Decision Trees, Random Forests, Boosting) → not affected by skewness
- **Where:** Apply on highly skewed features or target
- Scalers (MinMax, Standard, Robust) do not change the skew (shape) of the distribution – use other transformations (next slide)



Unskewing transformations

- **Square Root (SQRT) transformation**

```
import numpy as np  
y_train_sqrt = np.sqrt(y_train)  
y_test_sqrt = np.sqrt(y_test)
```

- **Log(arithmic) transformation**

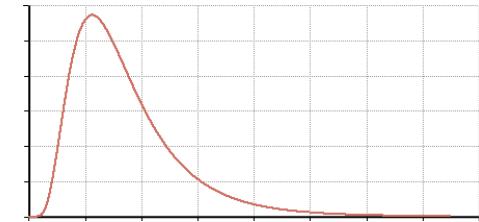
```
import numpy as np  
y_train_log = np.log(y_train)  
y_test_log = np.log(y_test)
```

- **Boxcox & Yeo-Johnson transformations**

- **Box-Cox** can handle both right and left skewed distributions but can only be applied to values that are **strictly positive**

- **Yeo-Johnson** can also handle both right and left skewed distributions and can be applied to **both positive and negative** values

- work well on right skewed distributions
- applicable on features with strictly positive values (sqrt and log cannot be applied on negative values)



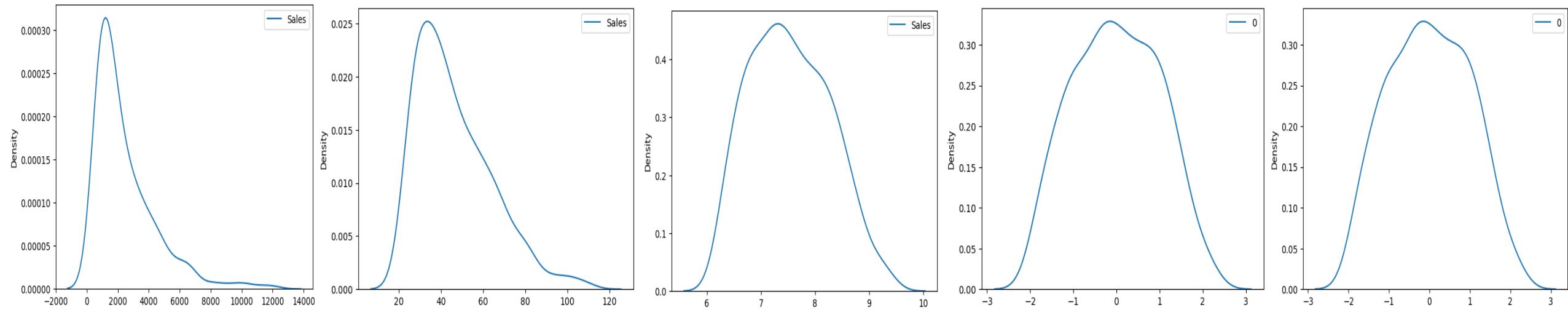
```
from sklearn.preprocessing import PowerTransformer  
pt_bc = PowerTransformer(method='box-cox')  
y_train_bc = pt_bc.fit_transform(y_train)  
y_test_bc = pt_bc.transform(y_test)
```

```
pt_yj = PowerTransformer(method='yeo-johnson')
```



Unskewing transformations: Examples

- Pandas .skew() method can be used to measure skewness of data



Original Sales column
Skewness: 1.815971

SQRT transformation
Skewness: 0.9551

LOG transformation
Skewness: 0.24818

BoxCox transformation
Skewness: 0.036628

YeoJohnson transform
Skewness: 0.036744

- Source code and results are available in .ipynb file in course website
- A quite descriptive document on skewness can be found [here](#)



APPENDIX



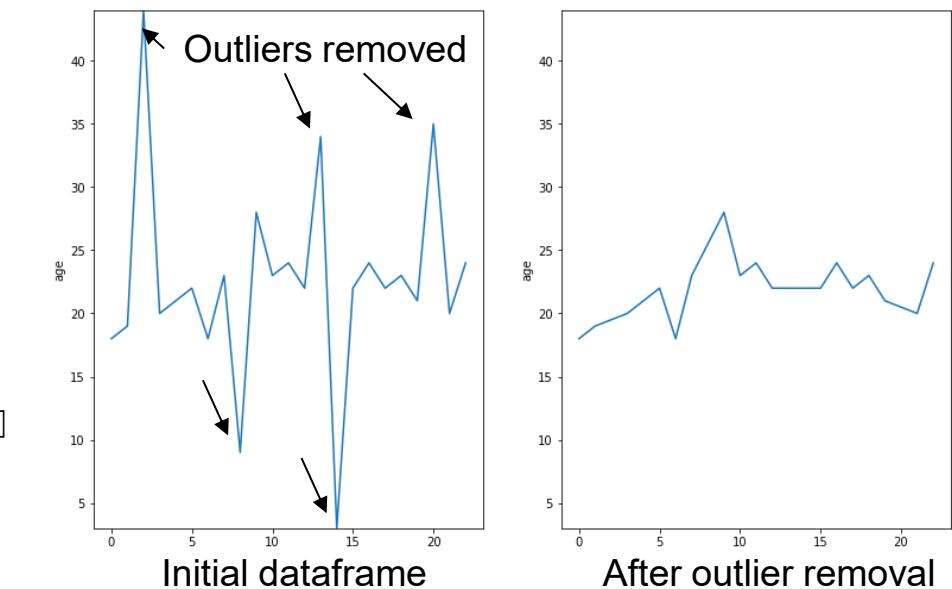
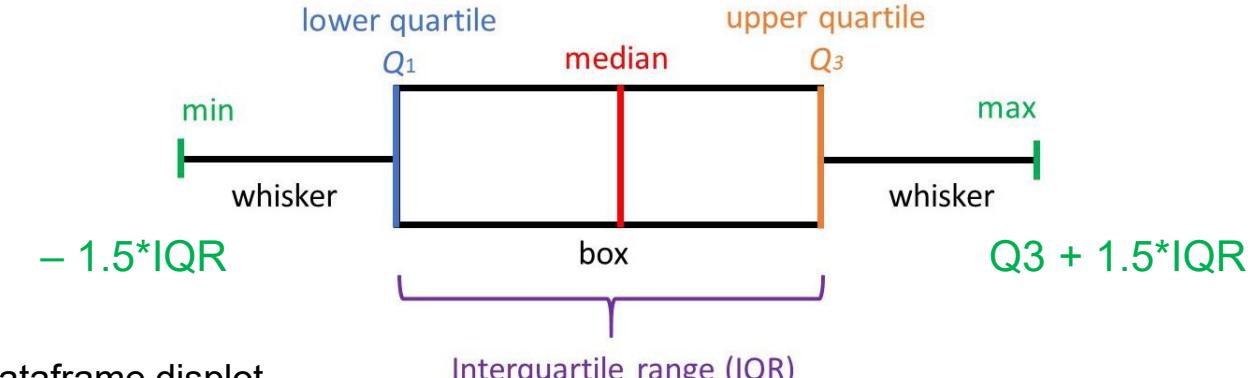
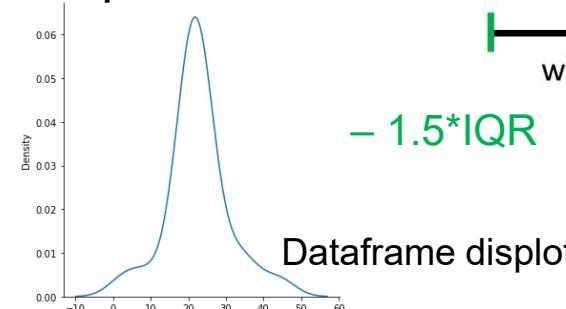
Removing outliers

- Methods for removing outliers **on each feature independently:**

- Interquartile Range (IQR) method

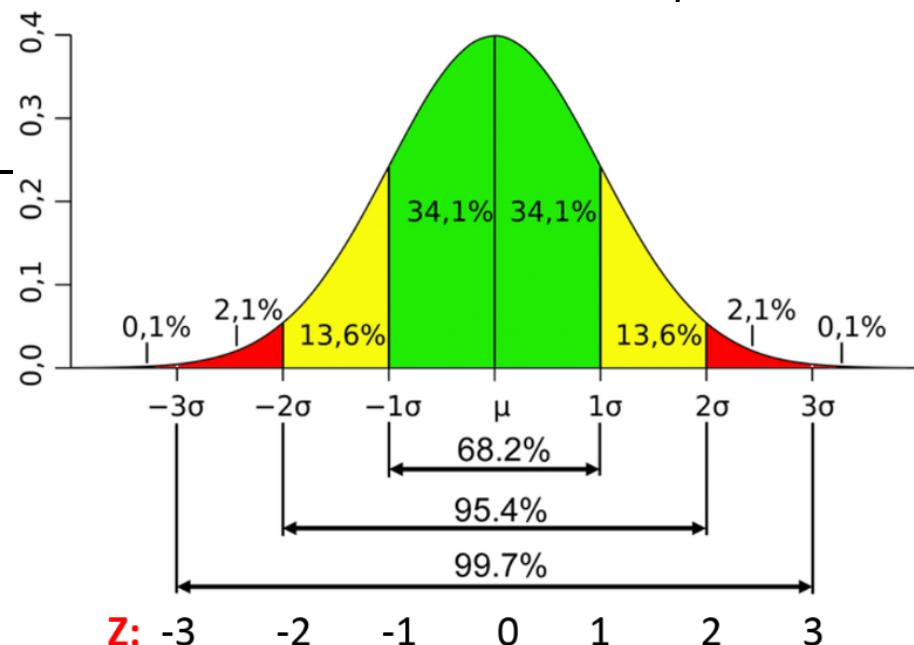
- Outliers are considered data points:
 - below $Q_1 - 1.5 \cdot IQR$
 - above $Q_3 + 1.5 \cdot IQR$

```
import seaborn as sns
import matplotlib.pyplot as plt
df2 = pd.DataFrame({'age': [18, 19, 44, 20, 21, 22, 18, 23, 9, 28, 23, 24, 22, 34, 3, 22, 24, 22, 23, 21, 35, 20, 24]})
plt.subplot(1,2,1)
sns.lineplot(data=df2, y=df2['age'], x=df2.index)
plt.ylim([df2['age'].min(), df2['age'].max()])
Q1 = df2['age'].quantile(0.25)
Q3 = df2['age'].quantile(0.75)
IQR = Q3-Q1
maximum = Q3 + 1.5*IQR
minimum = Q1 - 1.5*IQR
df3 = df2[ (df2['age'] > minimum) & (df2['age'] < maximum) ]
plt.subplot(1,2,2)
sns.lineplot(data=df3, y=df3['age'], x=df3.index)
plt.ylim([df2['age'].min(), df2['age'].max()])
```

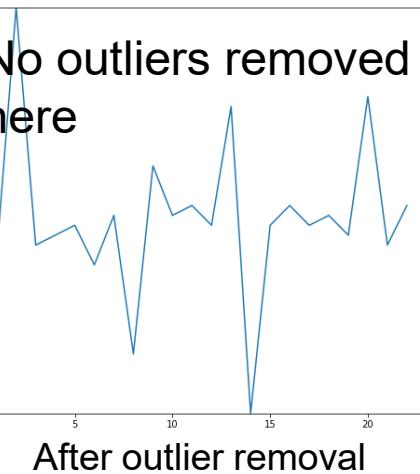
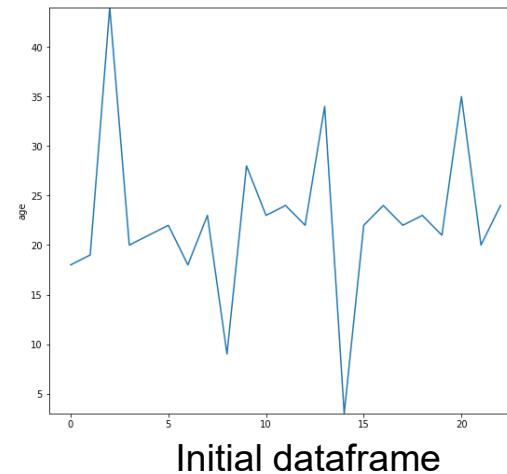


Removing outliers

- Mean (μ) and Standard Deviation (σ) method
 - For features that follow the normal distribution
 - Outliers are considered data points:
 - below $\mu - 3\sigma$
 - above $\mu + 3\sigma$



```
mean = df2['age'].mean()  
std = df2['age'].std()  
maximum = mean + 3*std  
minimum = mean - 3*std  
df4 = df2[ (df2['age'] > minimum) & (df2['age'] < maximum) ]  
plt.subplot(1,2,2)  
sns.lineplot(data=df4, y=df4['age'], x=df4.index)  
plt.ylim([df2['age'].min(), df2['age'].max()])
```





Removing outliers

– Median and Median Absolute Deviation (mad) method

- Replaces the mean and standard deviation with more robust statistics such as the median and median absolute deviation
- Outliers are considered data points:
 - below median – $3 \cdot \text{mad}$
 - above median + $3 \cdot \text{mad}$

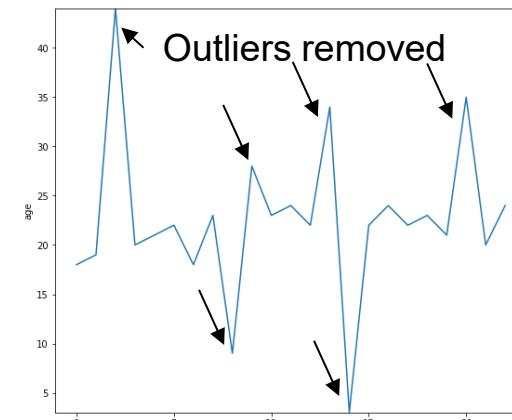
$$MAD = \text{Median}(|X_i - \text{median}|)$$

Step 1: Find the median.

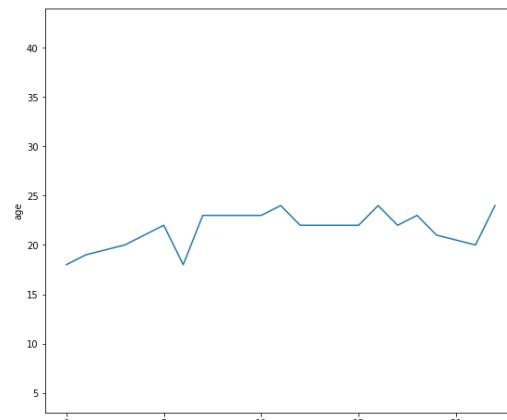
Step 2: Subtract the median from each x-value using the formula $|x_i - \text{median}|$.

Step 3: find the median of the absolute differences.

```
import scipy as sp
median = df2['age'].median()
mad = sp.stats.median_abs_deviation(df2['age'])
maximum = median + 3*mad
minimum = median - 3*mad
df4 = df2[ (df2['age'] > minimum) & (df2['age'] < maximum) ]
plt.subplot(1,2,2)
sns.lineplot(data=df4, y=df4['age'], x=df4.index)
plt.ylim([df2['age'].min(), df2['age'].max()])
```



Initial dataframe



After outlier removal

Extract time components from datetime feature



- # Extract time components
 - df['year'] = df['datetime'].dt.year
 - df['month'] = df['datetime'].dt.month
 - df['day'] = df['datetime'].dt.day
 - df['hour'] = df['datetime'].dt.hour
 - df['minute'] = df['datetime'].dt.minute
 - df['second'] = df['datetime'].dt.second
 - df['weekday'] = df['datetime'].dt.day_name() # or .dt.weekday (0=Mon)
 - df['week'] = df['datetime'].dt.isocalendar().week # ISO week number
-



Mean/std vs Median/mad

- Mean and std are highly affected by outliers
 - All values (including outliers) are used to calculate the mean and std
- Median and MAD are not highly affected by outliers
 - Outlier changes only center value(s) which are used to calculate the median
- Example:
 - dataset: {2,3,5,6,9}
 - mean = 5, std = 2.738, median = 5, mad = 2
 - Add outlier value 1000 to dataset
 - dataset: {2, 3, 5, 6, 9, 1000}
 - mean = 170.83, std = 406.21, median = $(5+6)/2 = 5.5$, mad = 3
 - The outlier
 - increases mean by 165.83 and std by 403.472
 - increases median by 0.5 and mad by 1



APPENDIX

Resampling will be further discussed in lab about Timeseries data



Resampling data

- Resampling involves changing the frequency of time-dependent features (called timeseries)
- Two types of resampling are:
 - Upsampling: When you increase the frequency of the samples (higher granularity), such as from minutes to seconds
 - Downsampling: When you decrease the frequency of the samples (lower granularity), such as from minutes to hours
- Resampling may be required if:
 - **data is not available at the same frequency that you want to make predictions**
 - For example, you have a feature measured on a daily basis and you want to make predictions on a monthly basis => you need to downsample it to a monthly level
 - there is an extremely high number of observations that needs to be diminished to speedup both EDA and ML algorithms execution time
 - Need for downsampling



Resampling data – Example

- Shampoo dataset: describes the **monthly** number of sales of shampoo over a 3-year period (2001 to 2003) – 36 observations
- Load dataset

```
from pandas import read_csv  
from datetime import datetime
```

```
shampoo_df = read_csv('shampoo.csv')  
print(shampoo_df.head())
```

	Month	Sales
0	1-01	266.0
1	1-02	145.9
2	1-03	183.1
3	1-04	119.3
4	1-05	180.3

```
# convert Month feature (e.g. from 1-01 to 2001-01-01) to follow the ISO 8601 format  
shampoo_df['Month'] = shampoo_df['Month'].map(lambda m: datetime.strptime('200'+m, '%Y-%m'))
```

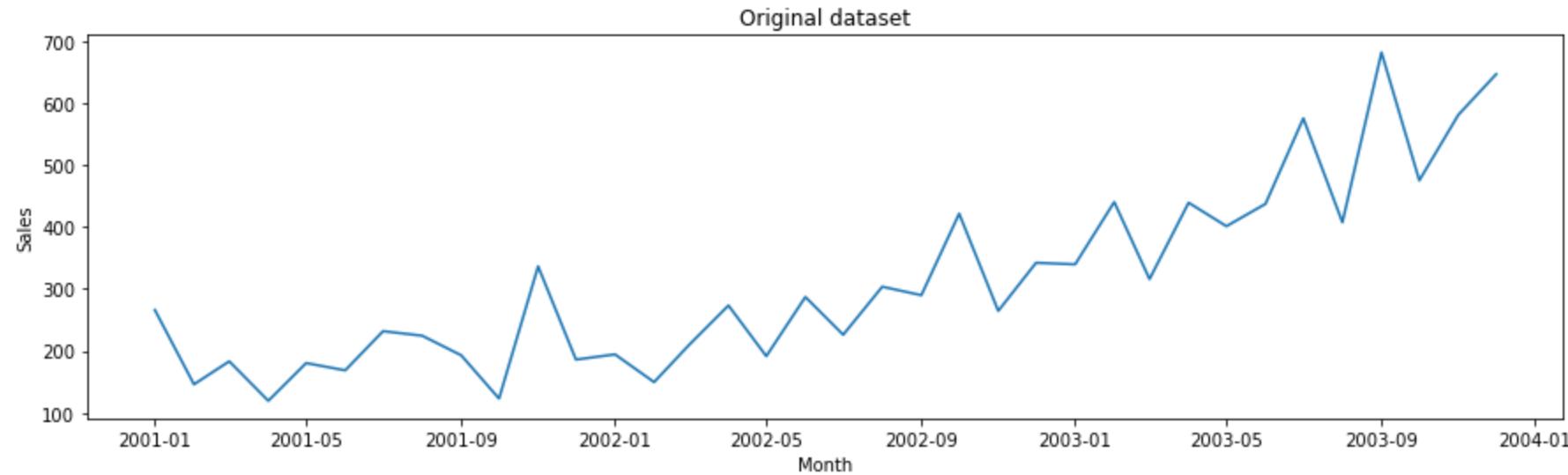
```
# dataframe must have a datetime-like index in order to use resample function  
# set Month feature as index  
shampoo_df = shampoo_df.set_index('Month')  
print(shampoo_df.head())
```

Month	Sales
2001-01-01	266.0
2001-02-01	145.9
2001-03-01	183.1
2001-04-01	119.3
2001-05-01	180.3



Resampling data – Example

```
plt.figure(1, figsize=(15, 4))
sns.lineplot(data=shampoo_df, x=shampoo_df.index, y=shampoo_df.Sales)
plt.title('Original dataset')
plt.show()
```





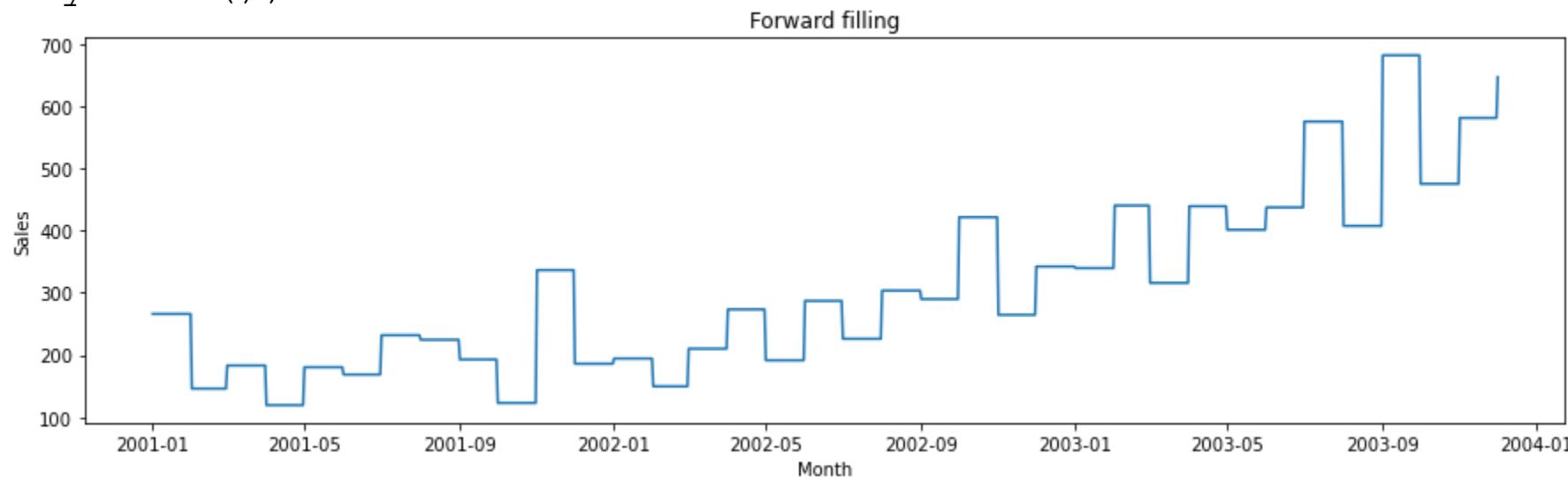
Resampling data – Upsampling

- Resample by day

```
# forward fill  
daily=shampoo_df.resample('D').ffill()  
plt.figure(1,figsize=(15,4))  
sns.lineplot(data=daily, x=daily.index, y=daily.Sales)  
plt.title('Forward filling')  
plt.show()  
print(daily.head())
```

Resampling can be performed by:
second ('s'), minute ('min'), hour ('h'),
day ('D'), week ('W'), month end ('ME'),
quarter end ('QE'), year end ('YE')

Forward-filling imputed missing values
using the last observed value.



Resampling data – Upsampling filling strategies



`.ffill([limit])`

Forward fill the values.

`.backfill([limit])`

Backward fill the new missing values in the resampled data.

`.bfill([limit])`

Backward fill the new missing values in the resampled data.

`.pad([limit])`

Forward fill the values.

`.nearest([limit])`

Resample by using the nearest value.

`.fillna(method[, limit])`

Fill missing values introduced by upsampling.

`.asfreq([fill_value])`

Return the values at the new freq, essentially a reindex.

`.interpolate([method, axis, limit, ...])`

Interpolate values according to different (linear or non-linear) methods.

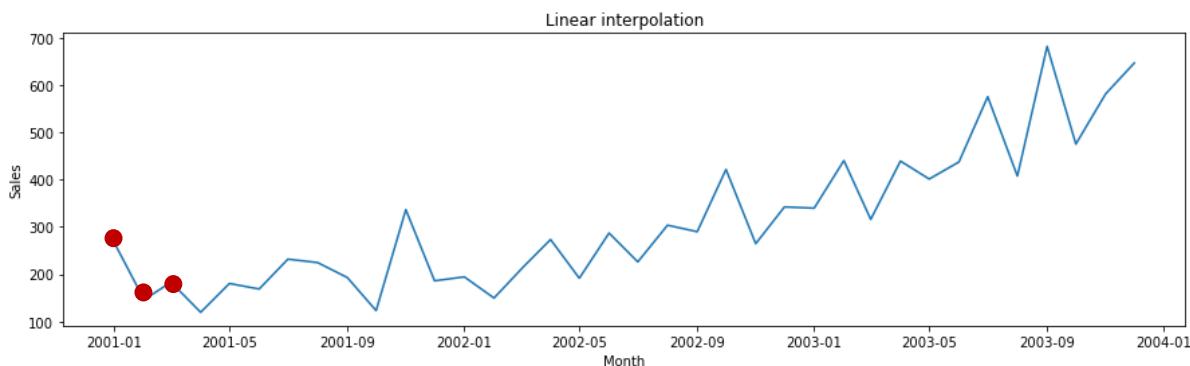


Resampling data – Upsampling

- Resample by day, filling by interpolation

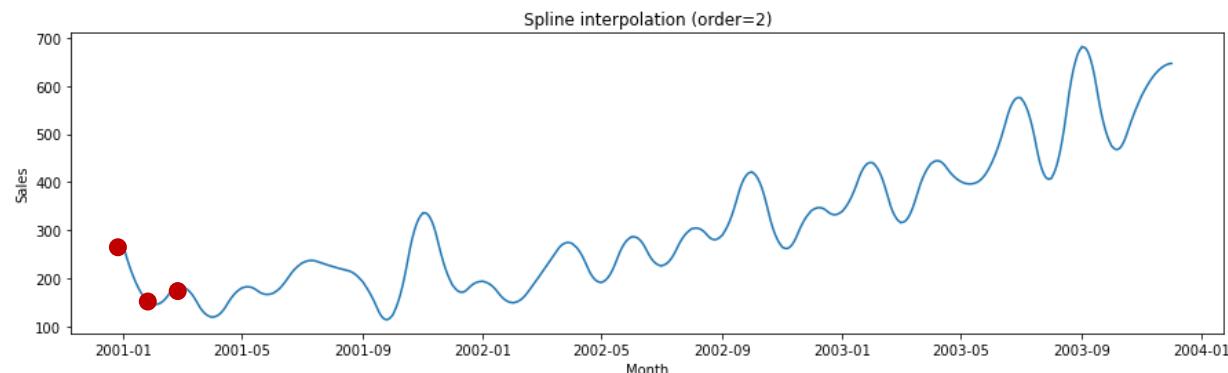
```
# linear interpolation
```

```
daily=shampoo_df.resample('D').interpolate(method='linear')
plt.figure(1,figsize=(15,4))
sns.lineplot(data=daily, x=daily.index, y=daily.Sales)
plt.title('Linear interpolation')
plt.show()
```



```
# spline interpolation
```

```
daily=shampoo_df.resample('D').interpolate(method='spline', order=2)
plt.figure(1,figsize=(15,4))
sns.lineplot(data=daily, x=daily.index, y=daily.Sales)
plt.title('Spline interpolation (order=2)')
plt.show()
```





Resampling data – Downsampling

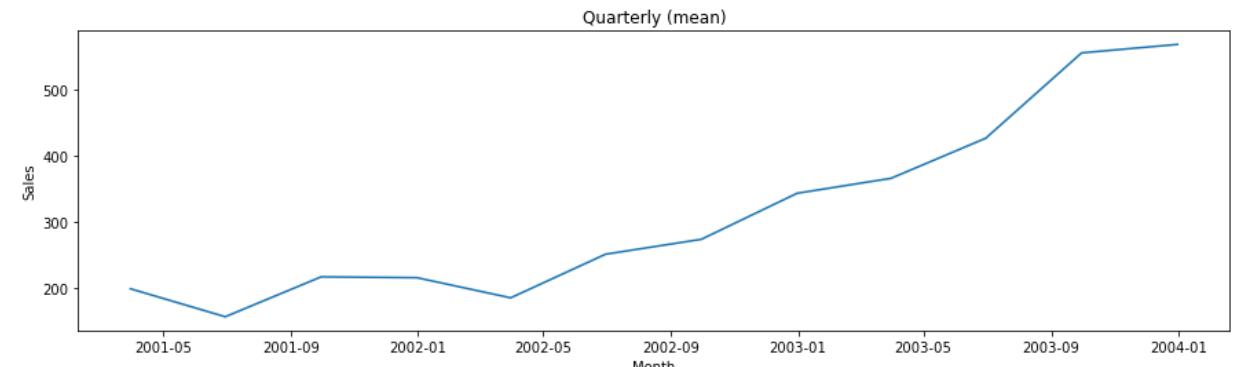
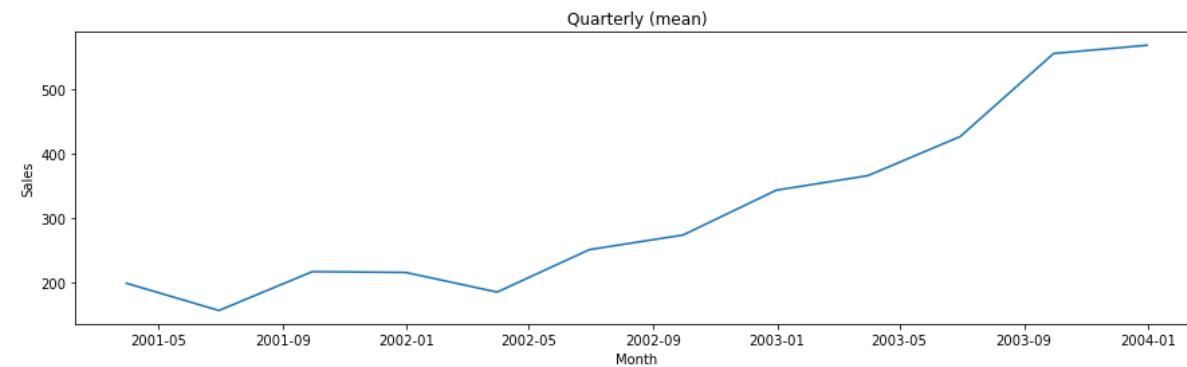
- Resample by quarter, aggregate by sum and mean

```
# sum aggregation
```

```
quarterly=shampoo_df.resample('QE').sum()
plt.figure(1,figsize=(15,4))
sns.lineplot(data=quarterly, x=quarterly.index,
y=quarterly.Sales)
plt.title('Quarterly (sum)')
plt.show()
```

```
# mean aggregation
```

```
quarterly=shampoo_df.resample('QE').mean()
plt.figure(1,figsize=(15,4))
sns.lineplot(data=quarterly, x=quarterly.index, y=quarterly.Sales)
plt.title('Quarterly (mean)')
plt.show()
```



Resampling data – Downsampling aggregation strategies



<code>.first([_method, min_count])</code>	Compute first of group values.
<code>.last([_method, min_count])</code>	Compute last of group values.
<code>.max([_method, min_count])</code>	Compute max of group values.
<code>.mean([_method])</code>	Compute mean of groups, excluding missing values.
<code>.median([_method])</code>	Compute median of groups, excluding missing values.
<code>.min([_method, min_count])</code>	Compute min of group values.
<code>.prod([_method, min_count])</code>	Compute prod of group values.
<code>.std([ddof])</code>	Compute standard deviation of groups, excluding missing values.
<code>.sum([_method, min_count])</code>	Compute sum of group values.
<code>.var([ddof])</code>	Compute variance of groups, excluding missing values.