# DSC510: Introduction to Data Science and Analytics
# Lab 4: Data Preparation

University of Cyprus
Department of
Computer Science

Pavlos Antoniou

Office: B109, FST01

# Prepare data for machine learning

- Data preparation is the process of making raw data ready for analysis and predictive modeling. It involves three main steps:
  - **1. Gather & Combine**
    - Collect data from databases, files (.csv, .json), APIs, social media, or statistical services.
    - Combine datasets (e.g., add features from multiple sources) to enrich the information for modeling.
  - **2. 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).
  - **3. 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
  - When NOT to delete rows:
    - Missing values may carry important information
      - Example: missing ages could correspond to older or 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 column by column
  - Categorical data (e.g., device type, country): add a new category *"Unknown"*
  - Numerical data (e.g., age):
    - Use mean/median on all values of the column
    - or aggregation (e.g. mean) on similar category values
  - Time-series data: use interpolation (see sampling slides)
  - Predictive imputation: build a model to predict missing values
- Correcting Erroneous Values
  - Detect outliers/invalid values with statistical analysis or visualization (e.g., box plots)
  - Examples:
    - Non-numeric value in a numeric column
    - Age < 0 or > 100
  - Strategy: delete or recode → then treat as missing values

# Cleaning data

- Handling Outliers (values that differ significantly from the rest)
  - Can distort models (e.g., regression, k-NN – especially distance-based models)
  - 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 |

# Missing values manipulation

- Evaluate the number of missing values per column:

```
In [ ]: # isnull() converts missing values to True, False otherwise; sum()
        works on Boolean values, treating True=1 and False=0, counts 1s
        missing_values_count = nfl_data.isnull().sum()

        # look at the # of missing points in the first ten columns
        missing_values_count[0:10]
```

```
Out[ ]: Date              0
        GameID            0
        Drive             0
        qtr               0
        down          54218
        time            188
        TimeUnder         0
        TimeSecs        188
        PlayTimeDiff    374
        SideofField     450
        dtype: int64
```

- 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, you might consider dropping it from the analysis, as it could contribute little to the model's accuracy

# Drop columns with missing values

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

```
Out[ ]:          Date      GameID Drive  ... ExPoint_Prob TwoPoint_Prob Season
        0  2009-09-10  2009091000     1  ...             0             0   2009
        1  2009-09-10  2009091000     1  ...             0             0   2009
        2  2009-09-10  2009091000     1  ...             0             0   2009
        3  2009-09-10  2009091000     1  ...             0             0   2009
        4  2009-09-10  2009091000     1  ...             0             0   2009

        [5 rows x 41 columns]
```

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

```
Out[ ]: Columns in original dataset: 102
        Columns with na's dropped: 41
```

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

```
nfl_data = nfl_data.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
        nfl_data = nfl_data.fillna(0)
        # replace all NA's with 0 for a specific column
        nfl_data['yacWPA'] = nfl_data['yacWPA'].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
        nfl_data = nfl_data.bfill(axis=0).fillna(0)
```

# 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 [ ]: # Using Sklearn's simple imputer
from sklearn.impute import SimpleImputer
import numpy as np
my_imputer = SimpleImputer(missing_values=np.NaN, strategy='mean')
nfl_data[['yacWPA']] = my_imputer.fit_transform(nfl_data[['yacWPA']])
```

DataFrame `nfl_data[['yacWPA']]` accepted.
Series `nfl_data['yacWPA']` **not** accepted

  - SimpleImputer takes two arguments such as 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 method is invoked on the instance of SimpleImputer to impute the missing values

# Fill in missing values with imputation

- Strategy = mean

| | Date | GameID | ... | yacWPA | Season |
|---|---|---|---|---|---|
| 0 | 2009-09-10 | 2009091000 | ... | **NaN** | 2009 |
| 1 | 2009-09-10 | 2009091000 | ... | 0.0368989644153847 6 | 2009 |
| 2 | 2009-09-10 | 2009091000 | ... | **NaN** | 2009 |
| 3 | 2009-09-10 | 2009091000 | ... | -0.156238531986491 3 | 2009 |
| 4 | 2009-09-10 | 2009091000 | ... | **NaN** | 2009 |

Before imputation

`nfl_data.head()`

| | Date | GameID | ... | yacWPA | Season |
|---|---|---|---|---|---|
| 0 | 2009-09-10 | 2009091000 | ... | **-0.010492** | 2009 |
| 1 | 2009-09-10 | 2009091000 | ... | 0.0368989644153847 6 | 2009 |
| 2 | 2009-09-10 | 2009091000 | ... | **-0.010492** | 2009 |
| 3 | 2009-09-10 | 2009091000 | ... | -0.156238531986491 3 | 2009 |
| 4 | 2009-09-10 | 2009091000 | ... | **-0.010492** | 2009 |

After imputation

# 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

- Similar to label encoding, but respects category order
- Used for **ordinal categorical features** (with natural, ordered values) and when **retaining the order is important**
- Encoding reflects the sequence

| | Degree |
|---|---|
| 0 | High school |
| 1 | Masters |
| 2 | Diploma |
| 3 | Bachelors |
| 4 | Bachelors |
| 5 | Masters |
| 6 | Phd |
| 7 | High school |
| 8 | High school |

```
Natural order:
'High school':1
'Diploma':2
'Bachelors':3
'Masters':4
'Phd':5
```

# Label & Ordinal Encoding: `OrdinalEncoder`

```
import category_encoders as ce
import pandas as pd
df=pd.DataFrame(
{'Degree':['High school', 'Masters', 'Diploma',
'Bachelors', 'Bachelors', 'Masters', 'Phd', 'High
school', 'High school']})

#Original data
df
```

| | Degree |
|---|---|
| 0 | High school |
| 1 | Masters |
| 2 | Diploma |
| 3 | Bachelors |
| 4 | Bachelors |
| 5 | Masters |
| 6 | Phd |
| 7 | High school |
| 8 | High school |

```
# create object of Ordinal encoding
ordinal_encoder = ce.OrdinalEncoder(
mapping=[{'col':'Degree','mapping':{'None':0,'High
school':1,'Diploma':2,'Bachelors':3,'Masters':4,
'Phd':5}}])

#fit and transform data
df['Ordinal'] =
ordinal_encoder.fit_transform(df['Degree'])
df
```

| | Degree | Ordinal |
|---|---|---|
| 0 | High school | 1 |
| 1 | Masters | 4 |
| 2 | Diploma | 2 |
| 3 | Bachelors | 3 |
| 4 | Bachelors | 3 |
| 5 | Masters | 4 |
| 6 | Phd | 5 |
| 7 | High school | 1 |
| 8 | High school | 1 |

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

# 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 data
df_onehot = onehot_encoder.fit_transform(df)
df_onehot
```

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

Dummy variables

| | Degree | Ordinal |
|---|---|---|
| 0 | High school | 1 |
| 1 | Masters | 4 |
| 2 | Diploma | 2 |
| 3 | Bachelors | 3 |
| 4 | Bachelors | 3 |
| 5 | Masters | 4 |
| 6 | Phd | 5 |
| 7 | High school | 1 |
| 8 | High school | 1 |

| | Degree_High school | Degree_Masters | Degree_Diploma | Degree_Bachelors | Degree_Phd | Ordinal |
|---|---|---|---|---|---|---|
| 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1 |
| 1 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 4 |
| 2 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 2 |
| 3 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3 |
| 4 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3 |
| 5 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 4 |
| 6 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 5 |
| 7 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1 |
| 8 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1 |

# Drawbacks of One-Hot Encoding

- Creates **one new column per category** → 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.).
- Step 2: Apply cyclical encoding → map each value onto the unit circle using sine and cosine: `x = sin(2π * value / max_value), y = cos(2π * value / 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
    - $x_1 = size\ (0 - 2000m^2)$
    - $x_2 = number\ of\ bedrooms\ (1 - 5)$
    - $Euclidean\ distance(X_1, X_2) = \sqrt{(523 - 127)^2 + (4 - 2)^2}$
    - Distance is governed by features having boarder range of values

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

Feature x1 with high magnitudes weights a lot more (dominates) in the distance calculations than feature x2 with lower magnitudes

  - When distance is used by algorithms **make sure features are on a similar scale**
  - Target variable is not necessary to be scaled

# 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]

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

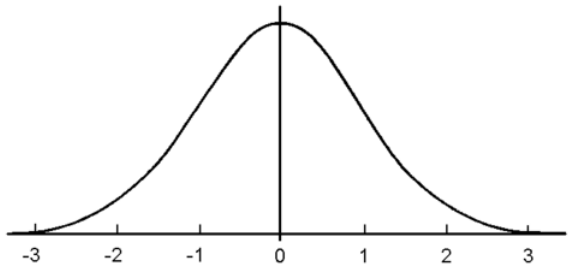
  - Scikit-learn module: MinMaxScaler or MaxAbsScaler
    - MinMaxScaler: Transforms features by scaling each feature to a given range ($x_{min}$ → $x_{max}$).

```
from sklearn.preprocessing import MinMaxScaler
df = pd.DataFrame({'A': [4, 3, 7], 'B': [13, 2, 8] })
# create the scaler object
scaler = MinMaxScaler(feature_range=(0, 1))
# train the scaler (find min and max)
scaler.fit(df)
# scale the dataset (apply the transformation)
minMaxRescaledX = scaler.transform(df)
print(minMaxRescaledX)
```

```
minMaxRescaledX =
scaler.fit_transform(df)
```

http://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing

# Data Transformation: Scaling




- Feature standardization
  - Rescales **each feature** individually to make values have zero mean ($\mu = 0$) and unit variance ($\sigma^2 = 1$)

$$x = \begin{bmatrix} 4 & 13 \\ 3 & 2 \\ 7 & 8 \end{bmatrix}, x_{i,j,std} = \frac{x_{i,j} - mean(x_j)}{\sigma} \Rightarrow x_{std} = \begin{bmatrix} -0.39 & 1.86 \\ -0.98 & -1.226 \\ 1.37 & 0.07 \end{bmatrix}$$

  - 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
df = pd.DataFrame({'A': [4, 3, 7], 'B': [13, 2, 8] })
scaler = StandardScaler()
# train the standardizer (find mean, std) and standardize the dataset
standardRescaledX = scaler.fit_transform(df)
print(standardRescaledX)
```

# 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 25$^{th}$ and 75$^{th}$ 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
df = pd.DataFrame({'A': [4, 3, 7], 'B': [13, 2, 8] })
rscaler = RobustScaler().fit(df)
# train the standardizer (find median, quantiles) and standardize the dataset
robustRescaledX = rscaler.fit_transform(df)
print(robustRescaledX)
```

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 (e.g., education levels 1-3, 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?

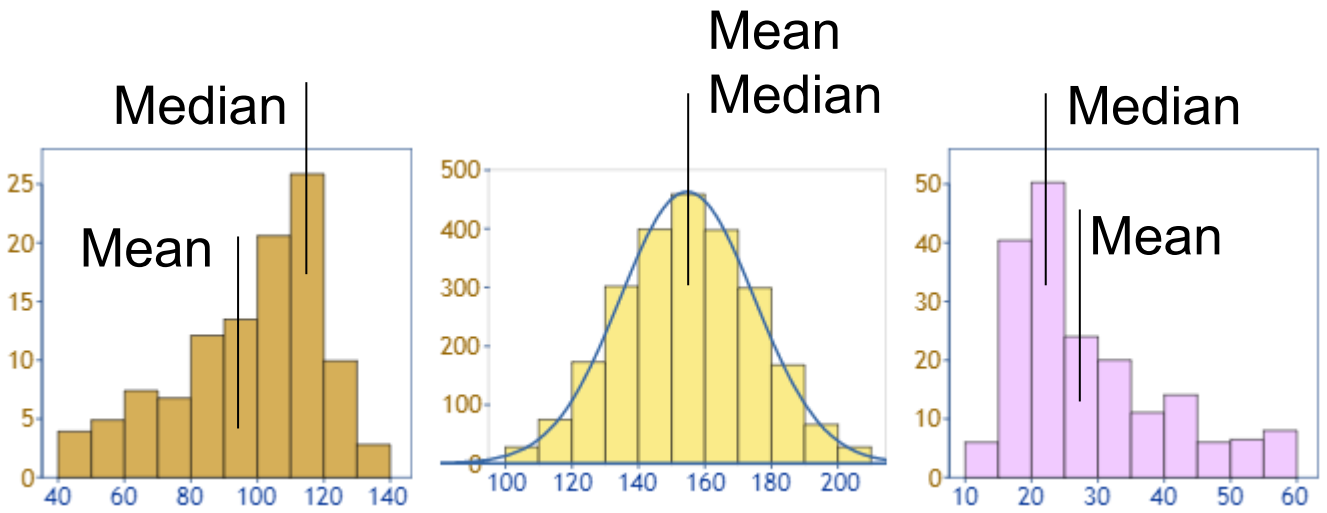
- Scaling features being one-hot-encoded **is not recommended**
- Binary values don’t require scaling
  - One-hot encoding produces binary (0/1) columns. These already express presence (1) or absence (0) clearly. Scaling adds no meaningful information.
- Scaling distorts categorical meaning
  - Scaling 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: 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



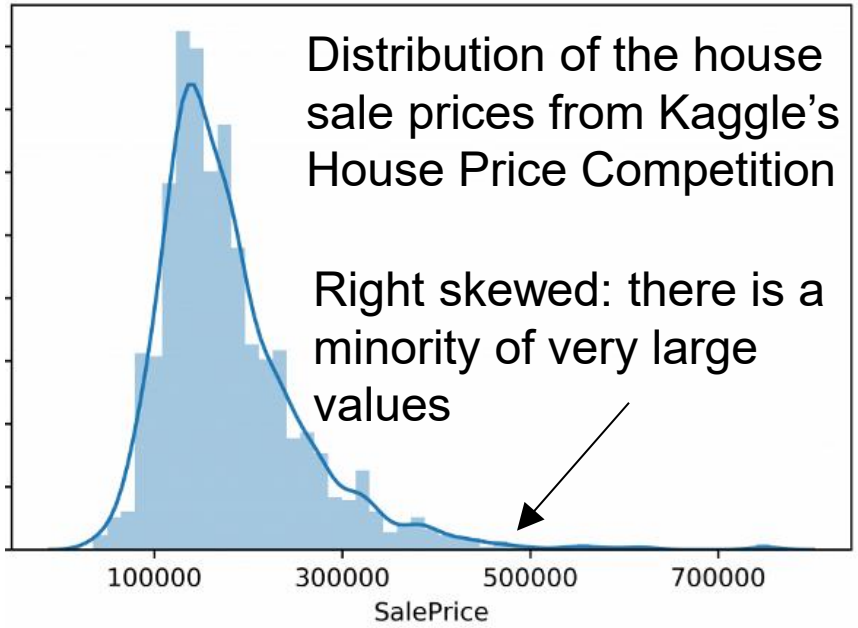

**Left** / **negative** skew: Long tail is on the left / negative side of the peak

**No** skew (symmetrical distribution)

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

# Effects of skewed data

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




  - A regression model for predicting house sale prices or using house sale price as input feature (using the above dataset) will be trained on a much larger number of moderately priced houses and will be less likely to successfully predict the price for the most expensive houses

# 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

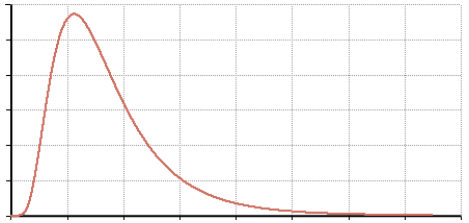
- Square Root (SQRT) transformation

```
import numpy as np
np.sqrt(df.column)
```

- Log(arithmic) transformation

```
import numpy as np
np.log(df.column)
```

- work well on right skewed distributions



- applicable on features with strictly positive values (sqrt and log cannot be applied on negative values)

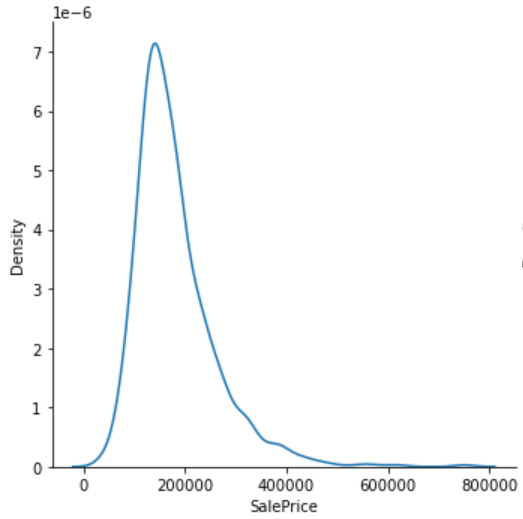
- Boxcox & Yeo-Johnson transformations
  - Box-Cox can handle both right and left skewed distributions but can only be applied to values that are strictly positive
  - Yeo-Johnson can also handle both right and left skewed distributions and can be applied to both positive and negative values

```
from scipy.stats import boxcox
df['bc_col'] = boxcox(df['col'])
```

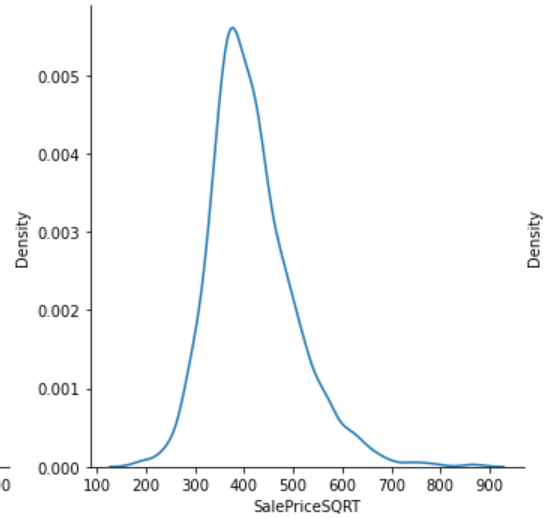
```
from scipy.stats import yeojohnson
df['yj_col'] = yeojohnson (df['col'])
```

# Unskewing transformations: Examples

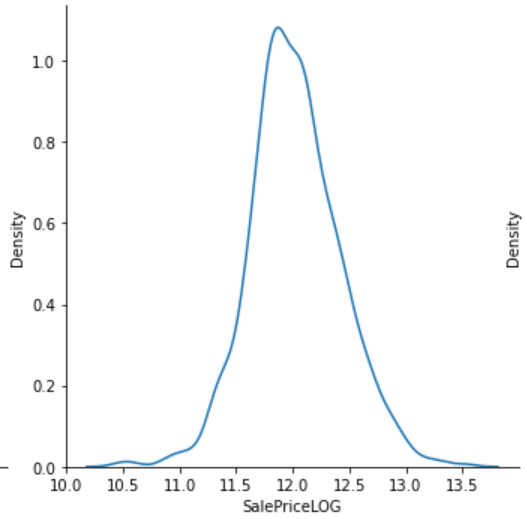
- Pandas .skew() method can be used to measure skewness of data



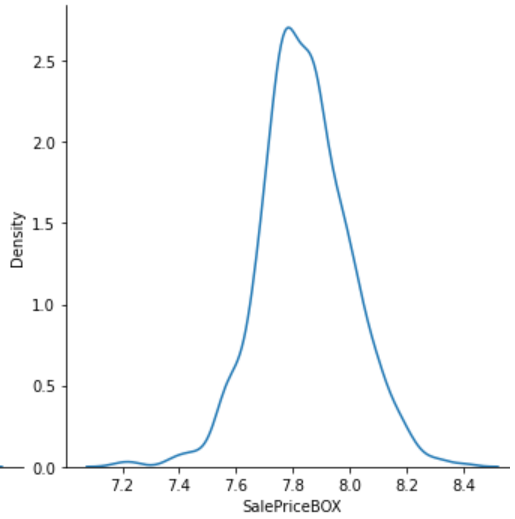

Original SalePrice column
Skewness: 1.8828757



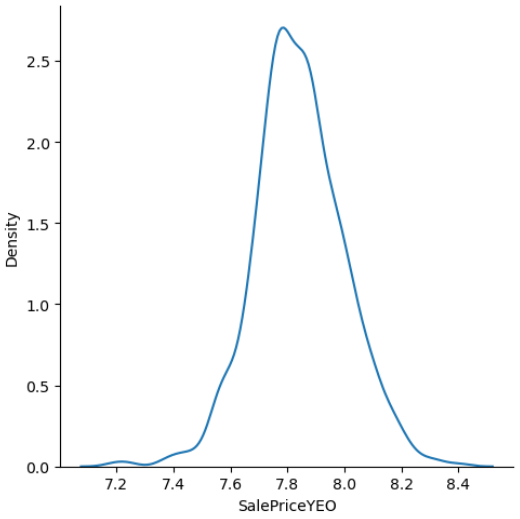

SQRT transformation
Skewness: 0.9431527




LOG transformation
Skewness: 0.1213351




BoxCox transformation
Skewness: -0.0086530



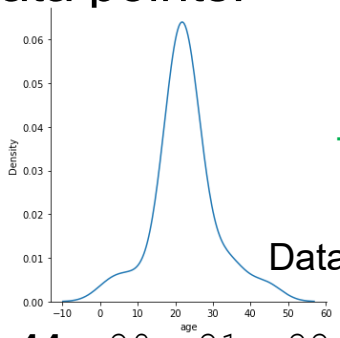

YeoJohnson transform
Skewness: -0.0086536

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

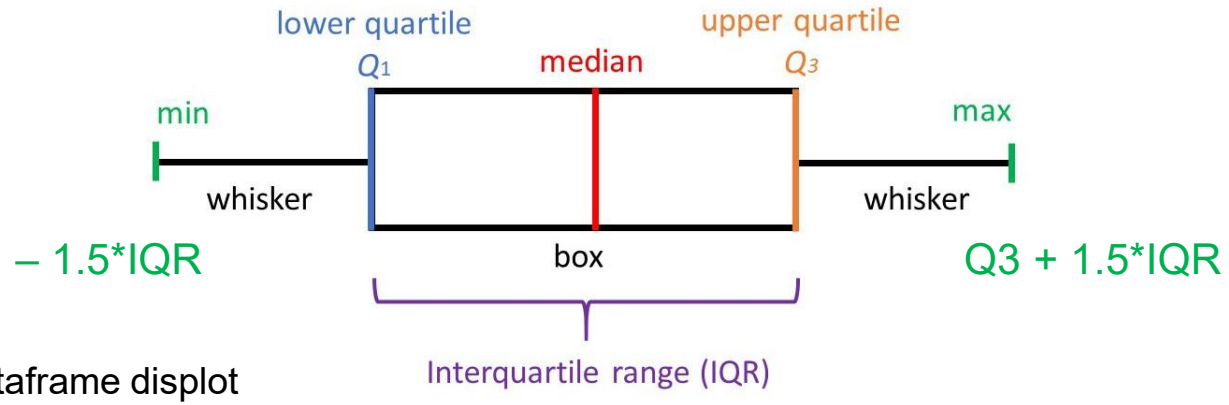
# APPENDIX

# Removing outliers

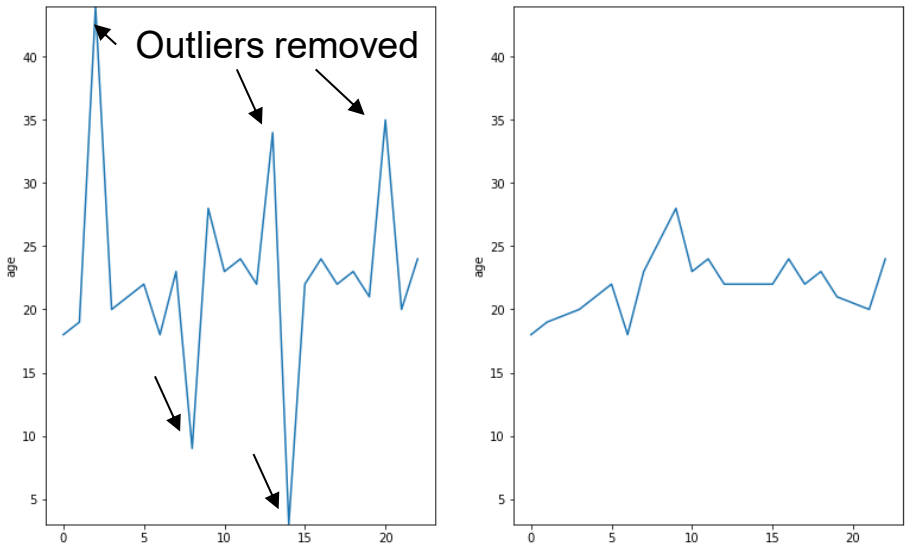
- Methods for removing outliers on each feature independently:
  - Interquartile Range (IQR) method
    - Outliers are considered data points:
      - below Q1 – 1.5*IQR
      - above Q3 + 1.5*IQR




Dataframe displot




```
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()])
```



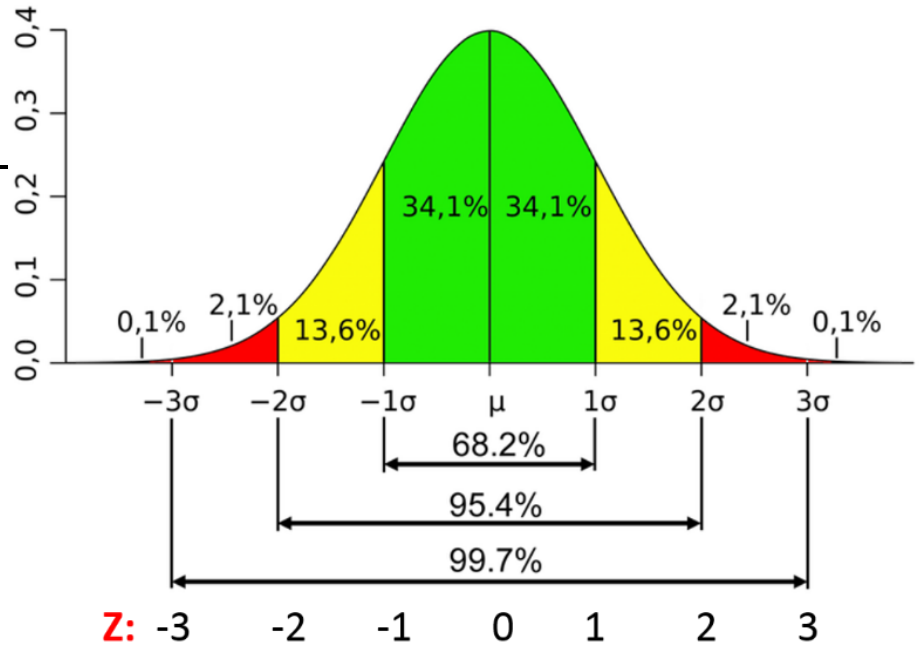

Initial dataframe



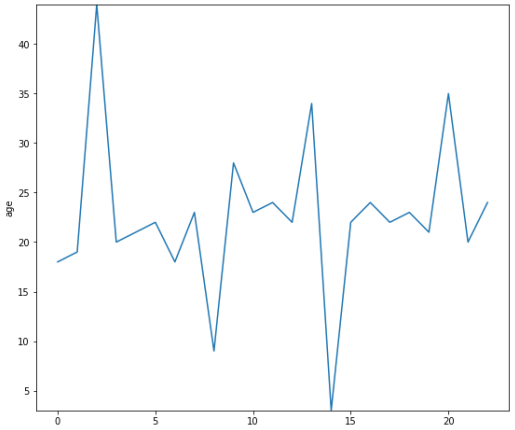

After outlier removal

# Removing outliers

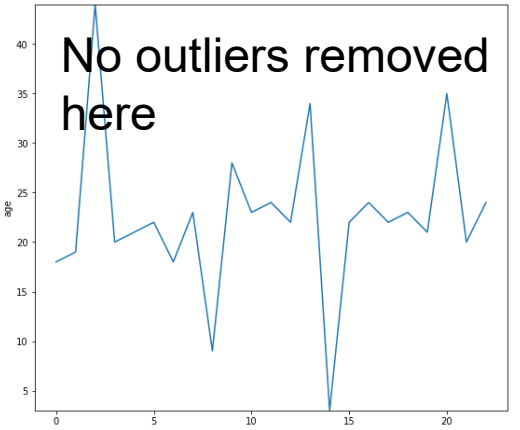
- Mean (μ) and Standard Deviation (σ) method
  - **For features that follow the normal distribution**
  - Outliers are considered data points:
    - below μ – 3*σ
    - above μ + 3*σ




```
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()])
```




Initial dataframe




After outlier removal

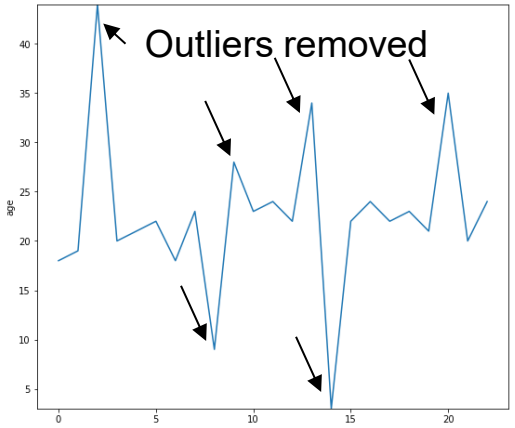
# Removing outliers

- 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*mad
    - above median + 3*mad

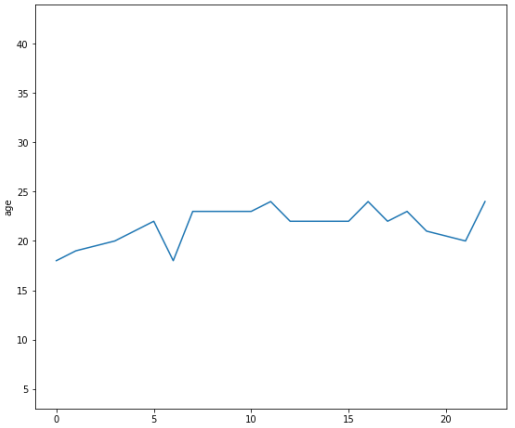
$$MAD = Median(|X_i - median|)$$

Step 1: Find the median.
Step 2: Subtract the median from each x-value using the formula $|x_i - median|$.
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

# 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