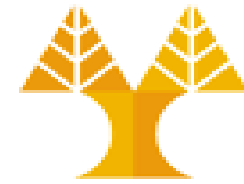


# EPL448: Data Mining on the Web – Lab 6



University of Cyprus  
Department of  
Computer Science

Παύλος Αντωνίου

Γραφείο: B109, ΘΕΕ01

# Prepare data for machine learning



- Data preparation is the process of gathering, combining, structuring and organizing data so it can be used in Exploratory Data Analysis (statistical analysis and visualization) and Predictive Modelling
  - **Gather/combine data:** Finding the right data. This can come from databases, files (.csv, .json, .txt, .xml) or Rest APIs (web-based apps).
  - **Preprocess data:** Organize your selected data by formatting, **cleaning**, **encoding** and **(re-)sampling**.
    - Cleaning data is the fixing/deleting/filling in missing data
    - Encoding is the conversion of labeled/categorical data into numerical data
    - Resampling is about changing the frequency of observations
  - **Transform data:** Transform preprocessed data by engineering features using **scaling**, **unskewing**, **feature selection and feature extraction (next lab)** for achieving better performance of predictive modelling methods
    - Scaling is the process of rescaling or standardizing or normalizing features
    - Unskewing is making features' distribution symmetrical

# Cleaning data

---



- **Fixing** up formats
    - Often when data is coming from various international sources may (a) involve mixed formats, and/or (b) not follow the expected numeric syntax
      - decimal separator is dot (need to be replaced if comma)
      - there is no thousands separator (need to be removed if any)
      - monetary symbols before or after numbers (need to be removed if any)
    - Data may not follow the default (expected by Python plots or functions) formats
      - e.g. dates as integers 20090609231247 instead of the expected format 2009-06-09 23:12:47 (ISO 8601 format) need to be transformed
-

# Cleaning data

---

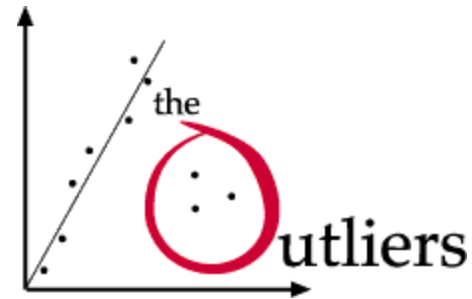


- **Deleting** missing values
    - may delete rows if the number of these rows is relatively small compared to the dataset
      - e.g. do not account for more than 10% of all lines
    - may delete rows if rows to be deleted are not important
      - e.g. do not contain info about a specific category in dataset
        - missing ages in some rows may correspond to older and more privacy or conscious users and are important in the decision-making process, so cannot be removed
        - This is not an easy task, especially if we are not familiar with the dataset
    - action can be performed with Pandas `dropna()`, see next slides
-

# Cleaning data



- **Filling** in missing values
  - For categorical data (e.g. device type, countries) makes sense to create a new category 'unknown'
  - For numerical values (e.g. age) makes sense to use:
    - Statistical aggregations such as mean or median of either (a) all rows of the column or (b) take into account rows belonging to the same category of that of the missing value
    - Interpolation: applied on time-series data, see slides about sampling
  - Build a predictors to predict a missing value based on columns that do have data
- **Correcting** erroneous values
  - For some columns, some values can be identified as obviously incorrect
    - E.g. find a number in a gender column
    - an age column with values below 0 or well over 100 (**outliers**)



# Cleaning data

---



- **Standardizing** categories
    - When data collected directly from users, especially from text fields → spelling mistakes, language differences → a given answer may be provided in multiple ways
      - E.g. country: USA, United States, U.S
      - E.g. dates: 1982-10-01, 1/10/1982
    - Goal: standardize values to ensure that there is only one version of each value
-

# Missing values manipulation



- Missing values are marked as NaN

```
In [ ]: # Read a dataset with missing values (download zipped dataset from here)
nfl_data = pd.read_csv('NFL Play by Play 2009-2016 (v3).csv', dtype='unicode')
nfl_data.head()
```

```
Out[ ]:   Date          GameID  ...          yacWPA  Season
0  2009-09-10  2009091000  ...          NaN    2009
1  2009-09-10  2009091000  ...  0.03689896441538476  2009
2  2009-09-10  2009091000  ...          NaN    2009
3  2009-09-10  2009091000  ... -0.1562385319864913  2009
4  2009-09-10  2009091000  ...          NaN    2009
```

```
In [ ]: nfl_data.isnull().head()
```

```
Out[ ]:   Date  GameID  Drive  qtr  down  ...  Win_Prob  WPA  airWPA  yacWPA  Season
0  False  False  False  False  True  ...  False  False  True  True  False
1  False  False  False  False  False  ...  False  False  False  False  False
2  False  False  False  False  False  ...  False  False  True  True  False
3  False  False  False  False  False  ...  False  False  False  False  False
4  False  False  False  False  False  ...  False  False  True  True  False
```

# Missing values manipulation



- There is a number of methods to deal with missing values in the data frame:

<b>df.method()</b>	<b>description</b>
<b>dropna()</b>	<b>Drop observations (rows) with at least one missing value</b>
dropna(how='all')	Drop observations (rows) where all cells is NA
<b>dropna(axis=1)</b>	<b>Drop the columns where at least one value is missing</b>
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
<b>fillna(0)</b>	<b>Replace missing values with a specified value</b>
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

IMPORTANT NOTE: All methods return a **new** DataFrame object unless the `inplace` parameter is set to `True`



# Missing values manipulation



```
In [ ]: # get the number of missing data points per column; sum True values
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
```

# Drop rows with missing values



```
In [ ]: # remove all the rows that contain a missing value
nfl_data.dropna()
```

```
Out[ ]:   Date  GameID  Drive  qtr  down  time  TimeUnder  TimeSecs  PlayTimeDiff  SideofField  ...  yac|
-----
0 rows × 102 columns
```

It looks like that's removed all our data! This is because every row in our dataset had at least one missing value.

# 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 also define in which columns to look for missing values.

```
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. the nfl_data dataframe is modified in place
nfl_data.fillna(0, inplace=True)
# replace all NA's with 0 for a specific column
nfl_data['yacWPA'].fillna(0, inplace=True)
```

- 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 to them

```
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.fillna(method = 'bfill', axis=0, inplace=True).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']])
```

- 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  ...  0.03689896441538476  2009
2  2009-09-10  2009091000  ...      NaN    2009
3  2009-09-10  2009091000  ... -0.1562385319864913  2009
4  2009-09-10  2009091000  ...      NaN    2009
```



```
      Date      GameID  ...      yacWPA  Season
0  2009-09-10  2009091000  ... -0.010492    2009
1  2009-09-10  2009091000  ...  0.03689896441538476  2009
2  2009-09-10  2009091000  ... -0.010492    2009
3  2009-09-10  2009091000  ... -0.1562385319864913  2009
4  2009-09-10  2009091000  ... -0.010492    2009
```

Before imputation

`nfl_data.head()`

After imputation

# Removing outliers

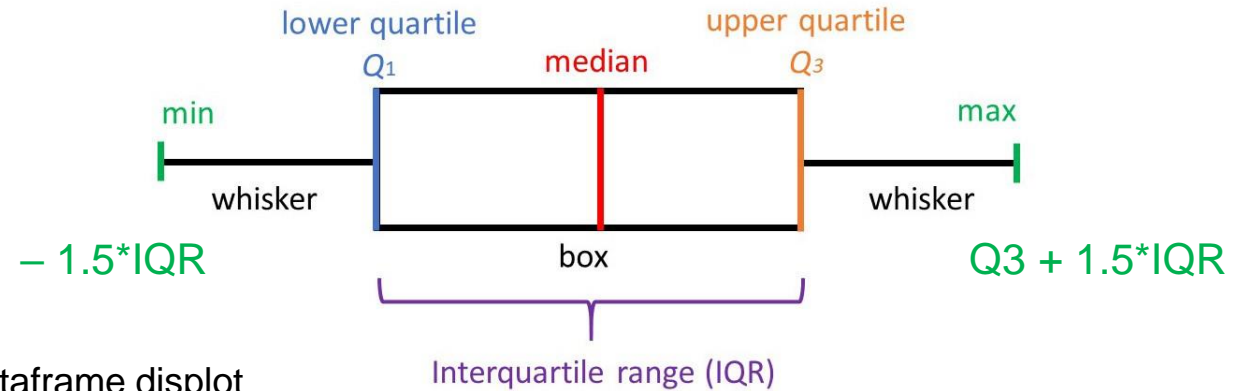
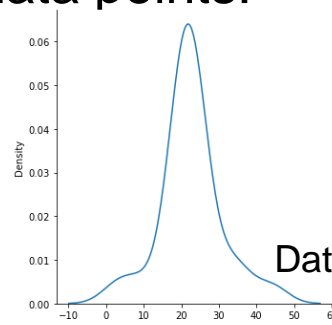


- Methods for removing outliers **on each feature independently**:

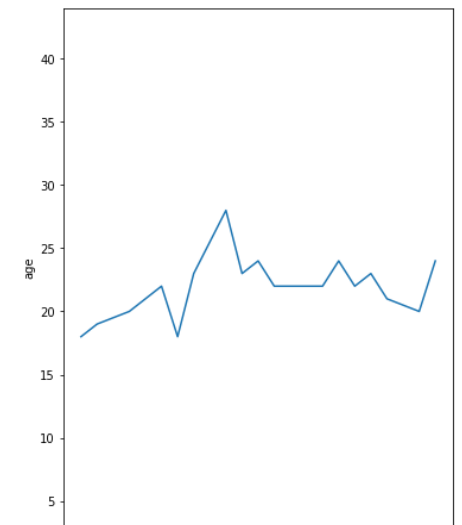
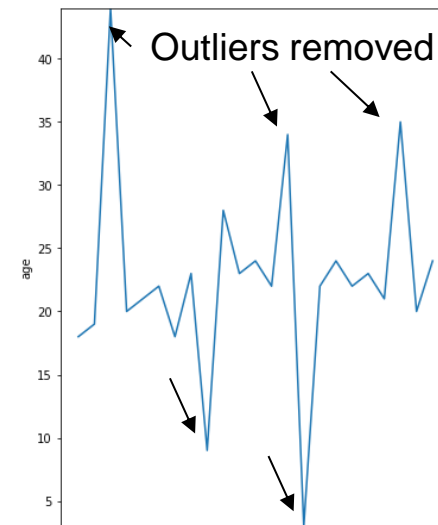
- Interquartile Range (IQR) method

- Outliers are considered data points:

- below  $Q1 - 1.5 \cdot IQR$
- above  $Q3 + 1.5 \cdot IQR$

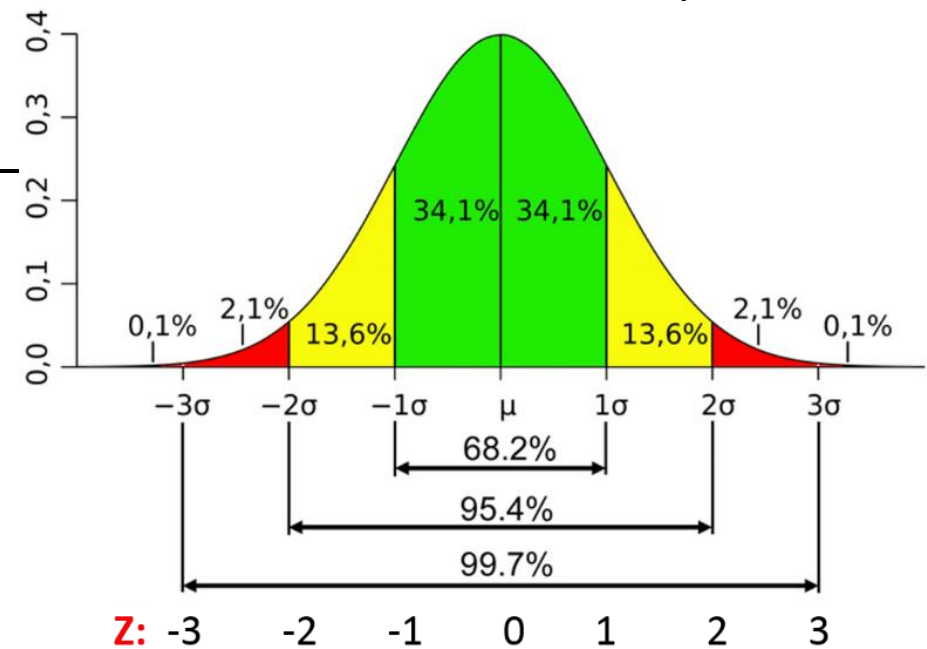


```
import seaborn as sns
import matplotlib.pyplot as plt
df2 = pd.DataFrame({'age': [18, 19, 44, 20, 21, 22, 18, 23, 9, 28, 23, 24, 22, 34, 3, 22, 24, 22, 23, 21, 35, 20, 24]})
plt.subplot(1,2,1)
sns.lineplot(data=df2, y=df2['age'], x=df2.index)
plt.ylim([df2['age'].min(), df2['age'].max()])
Q1 = df2['age'].quantile(0.25)
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()])
```

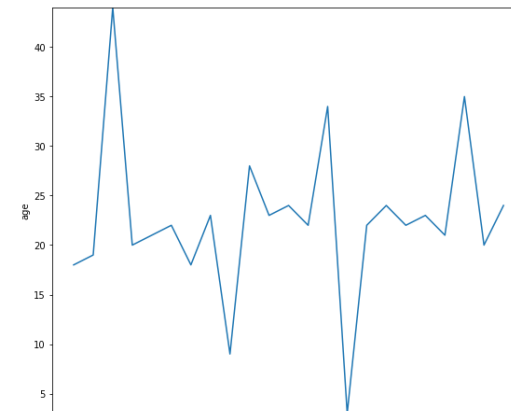


# Removing outliers

- Mean ( $\mu$ ) and Standard Deviation ( $\sigma$ ) 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()])
```



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

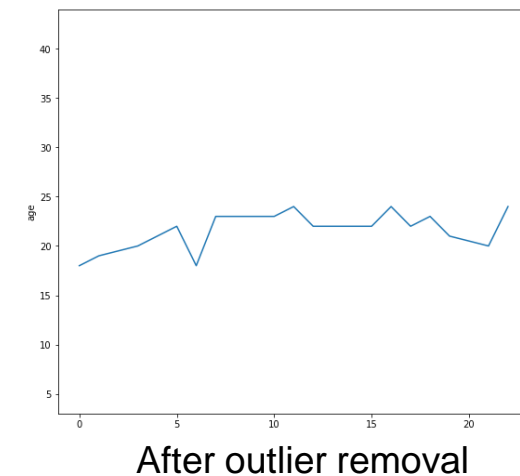
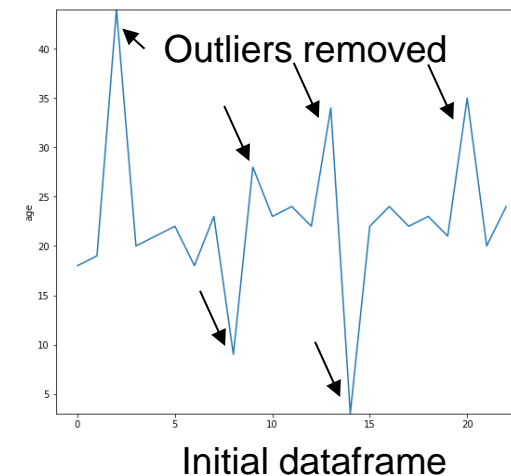
$$MAD = \text{Median}(|X_i - \text{median}|)$$

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()])
```



# 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

# Encoding categorical data

---



- Machine learning models require all input and output variables to be numerical
- Categorical data must be encoded to numbers
- Popular techniques:
  - Label Encoding or Ordinal Encoding
  - One-Hot Encoding
  - 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](http://contrib.scikit-learn.org/category_encoders))

# Label or Ordinal Encoding



- We use this categorical data encoding technique **when the categorical feature is ordinal** (with natural, ordered values)
- In this case, **retaining the order is important.**
- Encoding should reflect the sequence
- Therefore, each label (or category value) is converted into an integer value

	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 or Ordinal Encoding



```
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(return_df=True,
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: labels with high ordinal values possess higher “weight” and may be considered of higher importance

Note: If no mapping is given, order is automatically chosen by the encoder

# One-Hot Encoding

---



- We use this categorical data encoding technique when the features are nominal (do not have any order)
  - For each label of a categorical feature, we create a new feature (column)
  - Each label is mapped with a binary feature containing either 0 or 1
    - 0 represents the absence, and 1 represents the presence of that category value
  - These newly created binary features are known as Dummy variables
  - The number of dummy variables depends on the labels (categories) present in the categorical variable
-

# One-Hot Encoding



```
# Create object for One-hot encoding
onehot_encoder=ce.OneHotEncoder(cols=['Degree'],handle_unknown='return_nan', return_df=True,
use_cat_names=True)
#fit and transform data
df_onehot = onehot_encoder.fit_transform(df)
df_onehot
```

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

# Dummy Variable Encoding

---



- The one-hot encoding creates one binary feature for each category
    - ➔ this representation includes redundancy
      - For example, if we have a feature namely “color” with three labels (blue, green, red), in one-hot encoding we know that  $[1, 0, 0]$  represents “blue” and  $[0, 1, 0]$  represents “green” but it is redundant to use the (third) binary variable to represent “red” as  $[0, 0, 1]$  => instead we can represent 3 labels with two variables:  $[1, 0]$  for “blue”,  $[0, 1]$  for “green” and  $[0, 0]$  for “red”
        - We can eliminate one binary feature
  - This is called a dummy variable encoding, and always represents N labels (categories) with N-1 binary variables.
-



# Dummy Variable Encoding



```
df_dummy = pd.get_dummies(df, drop_first=True)  
df_dummy
```

Dummy variables

	Degree	Ordinal		Ordinal	Degree_Diploma	Degree_High school	Degree_Masters	Degree_PhD
0	High school	1		0	1	0	0	0
1	Masters	4		1	0	0	1	0
2	Diploma	2		2	1	0	0	0
3	Bachelors	3	➔	3	0	0	0	0
4	Bachelors	3		4	0	0	0	0
5	Masters	4		5	0	0	1	0
6	Phd	5		6	0	0	0	1
7	High school	1		7	0	1	0	0
8	High school	1		8	0	1	0	0

# Drawbacks of One-Hot & Dummy Encoding

---



- If there are multiple labels (categories) in a feature → we need a similar number of dummy variables to encode the data
    - For example, a feature with 30 different values (labels) will require 29-30 new dummy variables for coding
  - If there are multiple categorical features in the dataset we will end with a high number of binary features
  - Due to the massive increase in the dataset, coding slows down the learning of the model along with deteriorating the overall performance that ultimately makes the model computationally expensive.
-

# Cyclical feature encoding



- When dealing with time-dependent data (e.g. months, days, hours) it's important to encode the properties of time properly
  - We can decompose datetime string to a set of new features: month, day of the month (1, 2, .. 31), hour, minute, day (Sun, Mon, ... Sat)
    - The numerical values of each column distort the notion of proximity, i.e. in the hour feature, midnight is represented by 0 and eleven (PM) in the evening is represented by 23 => large difference in weights
  - Cyclical encoding: a better way is to represent time of day as a point on the unit circle, using sine and cosine transformation

	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

- Majority of clustering / classification algorithms use the notion of distance (e.g. Euclidean) between 2 points

- Example

- Classify houses with 2 features

- $x_1 = \text{size (0 – 2000m}^2\text{)}$

- $x_2 = \text{number of bedrooms (1 – 5)}$

- $\text{Euclidean distance}(X_1, X_2) = \sqrt{(523 - 127)^2 + (4 - 2)^2}$

- Distance is governed by features having boarder range of values

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

Feature x1 with high magnitudes weights a lot more in the distance calculations than feature x2 with low magnitudes

- When distance is used by algorithms **make sure features are on a similar scale**

- Target value (to be predicted) is not necessary to be scaled

# Data Transformation: Scaling data

---



- Feature rescaling
    - Some examples of algorithms where feature scaling matters are:
      - **k-nearest neighbors** using Euclidean distance – classification algorithm
      - **k-means** using Euclidean distance – clustering algorithm
      - **logistic regression**, **SVMs**, perceptrons, neural networks etc.
        - if you are using gradient descent/ascent-based optimization, otherwise some weights 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 (bagging like **RandomForest**, boosting) are unaffected by the scale of feature variables
-

# Data Transformation: Scaling data



- Feature rescaling

- Rescale **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 ( $x_{\min} \rightarrow x_{\max}$ ).

```
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)
```

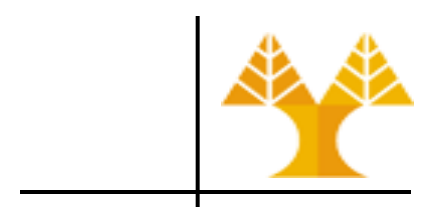
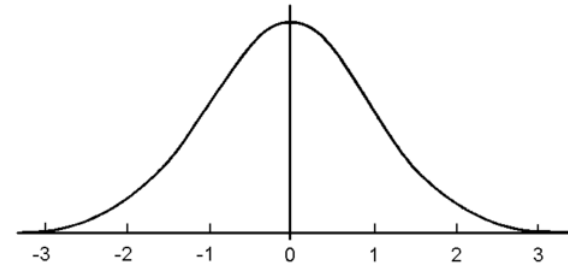
```
minMaxRescaledX = scaler.transform(df)
```

```
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 ( $\sigma^2 = 1$ )

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

- Centers data around zero

- Not recommended for sparse data: destroys sparseness

- Useful for algorithms such as the SVM (RBF kernel)

- 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)
```

Assumes that the feature distribution fits the **Gaussian distribution** (bell curve) with a well-behaved mean and standard deviation. You can still standardize your data if this expectation is not met, but you may not get reliable results.

# Data Transformation: Scaling data



- Feature standardization
  - When data contains outliers, Standard Scaler can be often misled since mean value and variance can be influenced by outliers
    - MinMaxScaler is sensitive to the presence of outliers as well
  - Rescales **each feature individually** to make values have zero median (median=0) and unit interquartile range (IQR=1)
  - **Centers data around zero**
  - 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?

---



- At the end of the day, the choice of using normalization (MinMax scaler), or standardization (Standard scaler / Robust scaler) will depend on your problem and the machine learning algorithm you are using
  - There is no hard and fast rule to tell you when to scale your features
  - You can always start by fitting (training) your machine learning model to (a) raw, (b) normalized and (c) standardized data and comparing the performance of all models to see which technique achieves the best results in terms of prediction performance
  - Scaling of target variable is generally not required
-

# Data Transformation: Scaling data



- Normalize observations
  - Normalize **each observation** (row) independently of other rows so that its norm (l1 or **l2**) equals 1

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

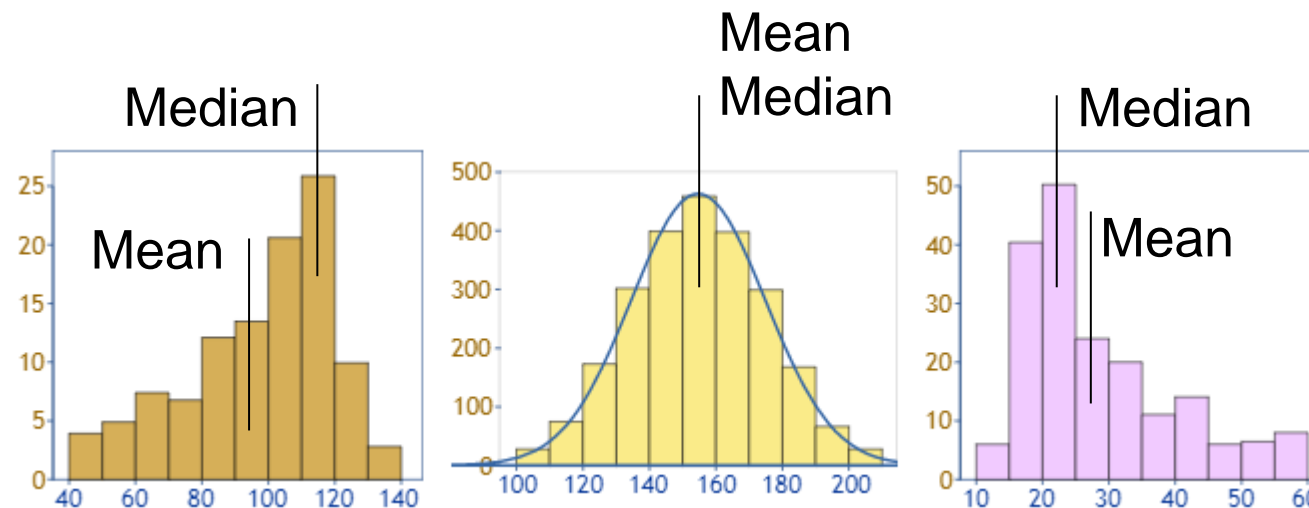
$\sqrt{0.29^2 + 0.96^2} = 1$

- Useful for sparse datasets (lots of zeros)
- Common operation for text classification or clustering
  - dot product of two l2-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



- Data 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

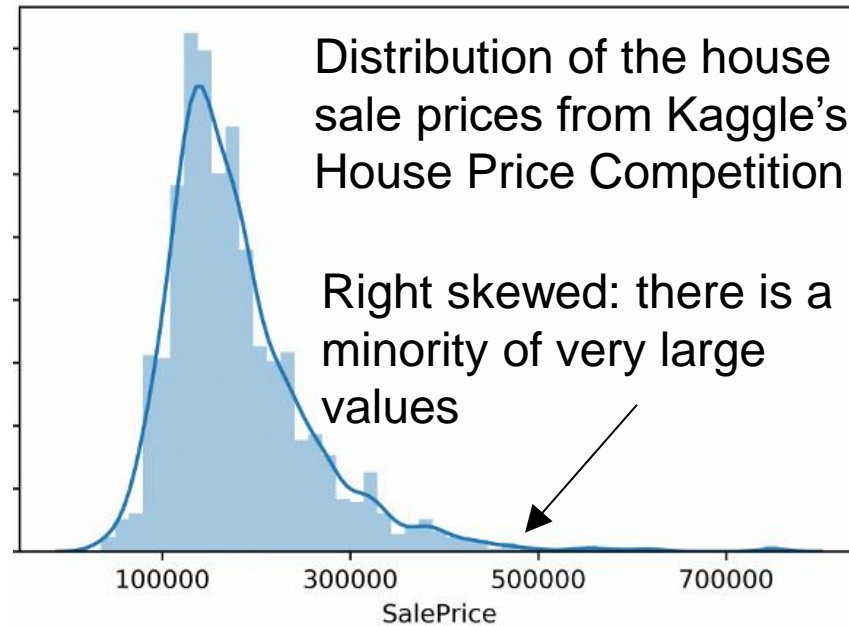
**No skew**  
(symmetrical  
distribution)

**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



- When removing skewness, transformations are attempting to change the shape of the distribution; to make it symmetric (Gaussian)
  - The reason is simply that if the dataset can be transformed to be statistically close enough to a Gaussian dataset, some machine learning algorithms such as Linear Regression, Gaussian Naïve Bayes are able to achieve better predictive performance (see Lab 8 for more info)
  - However, other machine learning models e.g. decision trees and ensembles of trees (bagging, boosting) are not affected by skewness (see Lab 8 for more info)
- Unskewing transformations are recommended to be applied on highly-skewed variables (input features and target variable)
- **Min-max scaler, standard scaler and robust scaler do not change the skew (shape) of the distribution; other techniques are needed**

# Unskewing transformations



- **Square Root** (SQRT) transformation

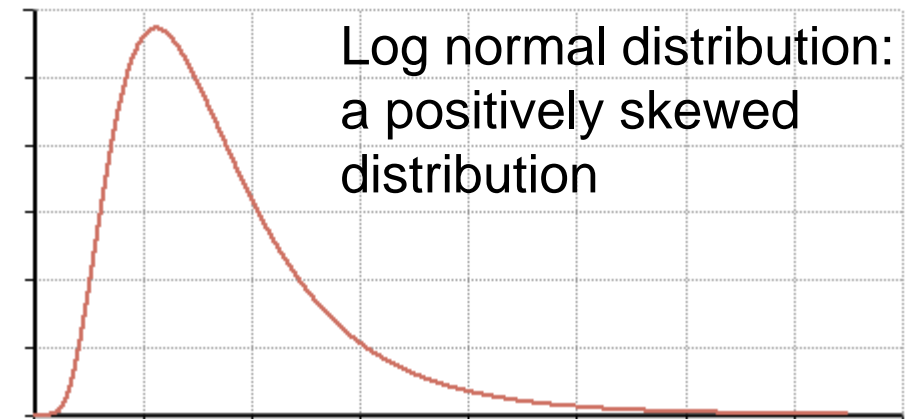
- can work well on positively skewed continuous data: `np.sqrt(df.column)`

```
import numpy as np
```

- **Log**(arithmic) transformation

- one of the most popular transformations to deal with skewed data: `np.log(df.column)`

- **Hint:** if the original data does not follow or approximate log-normal distribution, then log transformation does remove or reduce skewness



- **Boxcox** & Yeo-Johnson transformations

- **Box-Cox** procedure can only be applied to values that is **strictly positive**
  - **Yeo-Johnson** can be applied to **both positive and negative** values

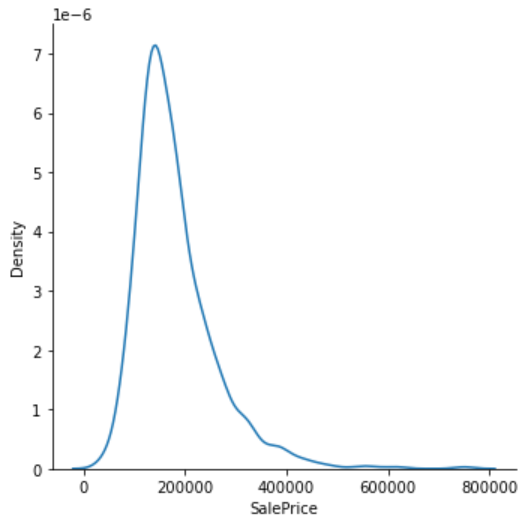
```
from scipy.stats import boxcox  
df['bc_col'] = boxcox(df['col'])
```

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

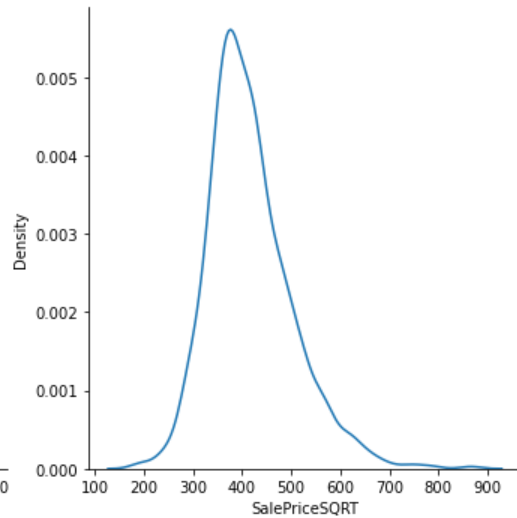
# Unskewing transformations: Examples



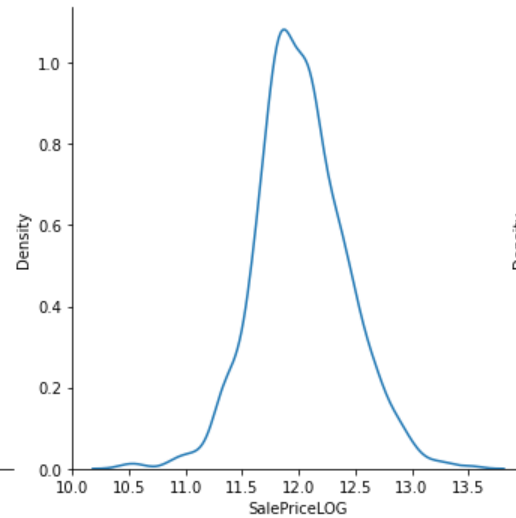
- Pandas `.skew()` method can be used to measure skewness of data



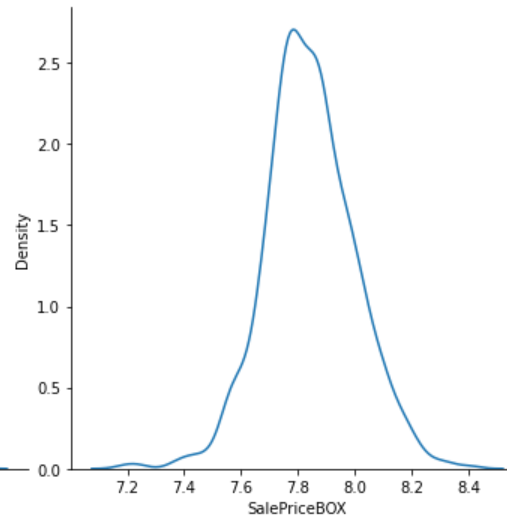
Original SalePrice column  
Skewness: 1.8828757



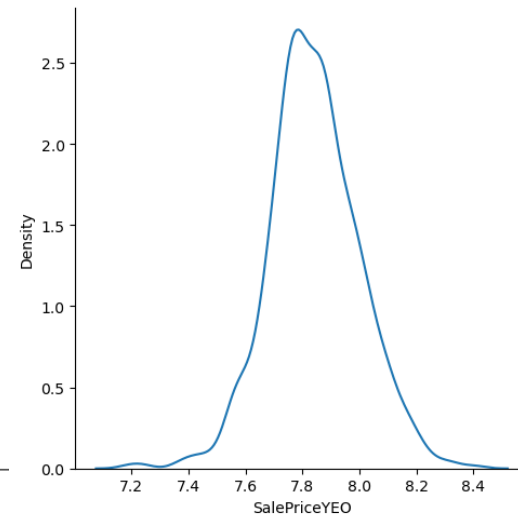
SQRT transformation  
Skewness: 0.9431527



LOG transformation  
Skewness: 0.1213351



BoxCox transformation  
Skewness: -0.0086529



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](#)

# Resampling data

---



- Resampling involves changing the frequency of time series observations
- Two types of resampling are:
  - Upsampling: Where you increase the frequency of the samples, such as from minutes to seconds
  - Downsampling: Where you decrease the frequency of the samples, 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 may have daily data and want to predict a monthly problem. So you need to downsample it to monthly data prior developing your model
  - there is an extremely high number of observations that needs to be diminished so as 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)
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
```

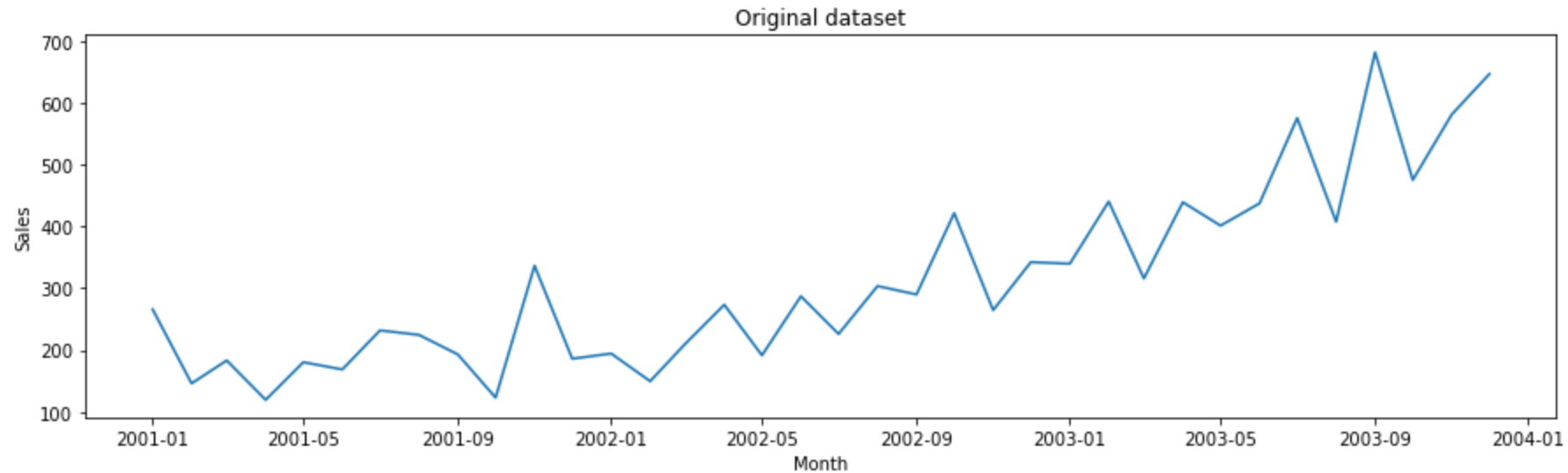
```
shampoo_df = shampoo_df.set_index('Month')
print(shampoo_df.head())
```

	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

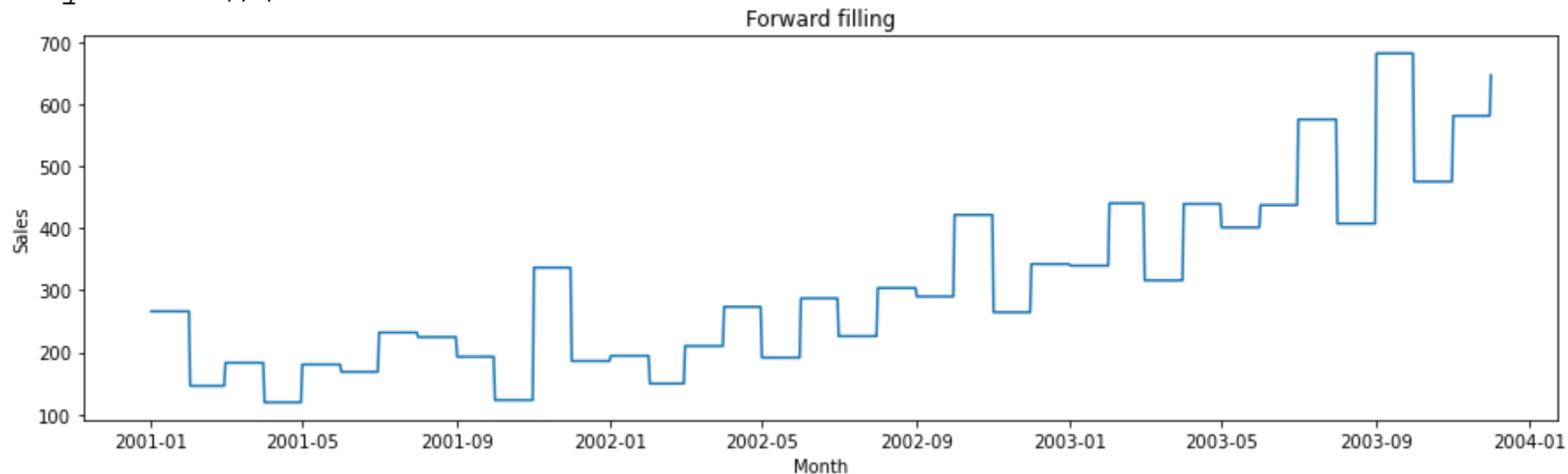
```
# forward fill
```

```
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 ('T'), hour ('H'),  
day ('D'), week ('W'), month ('M'),  
quarter ('Q'), year ('Y')

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

Month	Sales
2001-01-01	266.0
2001-01-02	266.0
2001-01-03	266.0
2001-01-04	266.0
2001-01-05	266.0



# Resampling data – Upsampling filling strategies

---



<code><a href="#">.ffill</a>([limit])</code>	Forward fill the values.
<code><a href="#">.backfill</a>([limit])</code>	Backward fill the new missing values in the resampled data.
<code><a href="#">.bfill</a>([limit])</code>	Backward fill the new missing values in the resampled data.
<code><a href="#">.pad</a>([limit])</code>	Forward fill the values.
<code><a href="#">.nearest</a>([limit])</code>	Resample by using the nearest value.
<code><a href="#">.fillna</a>(method[, limit])</code>	Fill missing values introduced by upsampling.
<code><a href="#">.asfreq</a>([fill_value])</code>	Return the values at the new freq, essentially a reindex.
<code><a href="#">.interpolate</a>([method, axis, limit, ...])</code>	Interpolate values according to different methods.

---

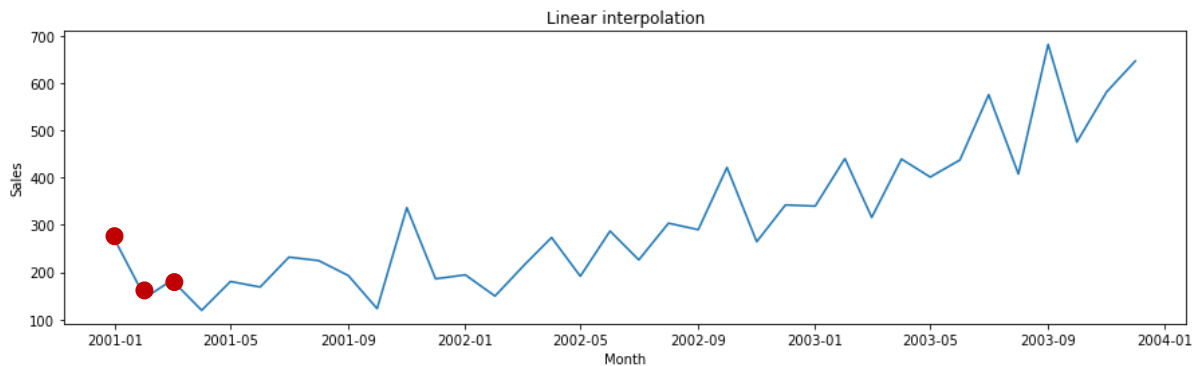
# Resampling data – Upsampling



- Resample by day, filling by interpolation

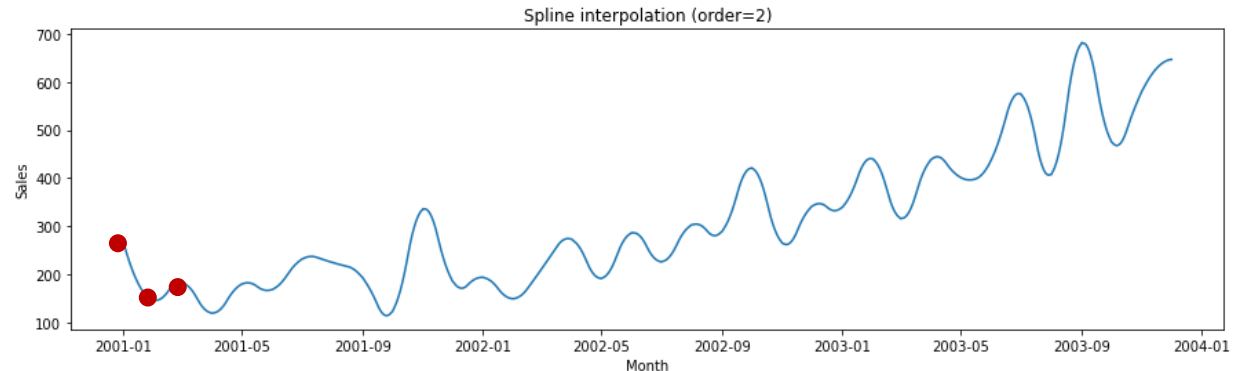
## # linear interpolation

```
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

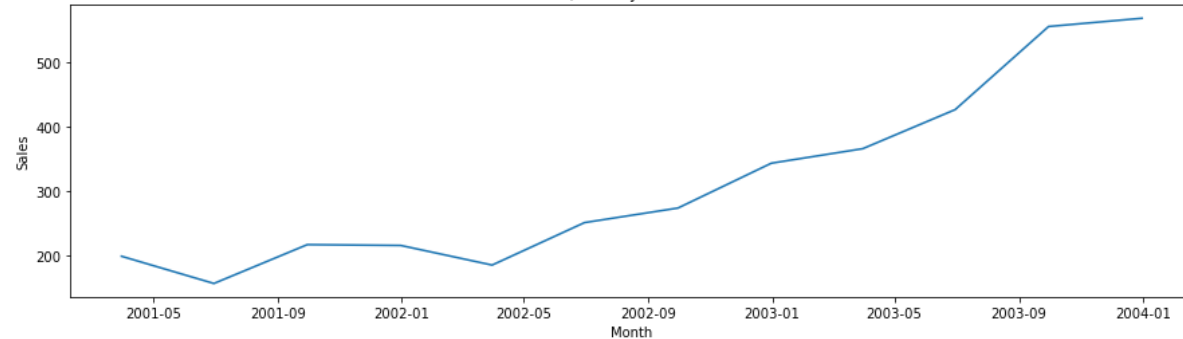
## # sum aggregation

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

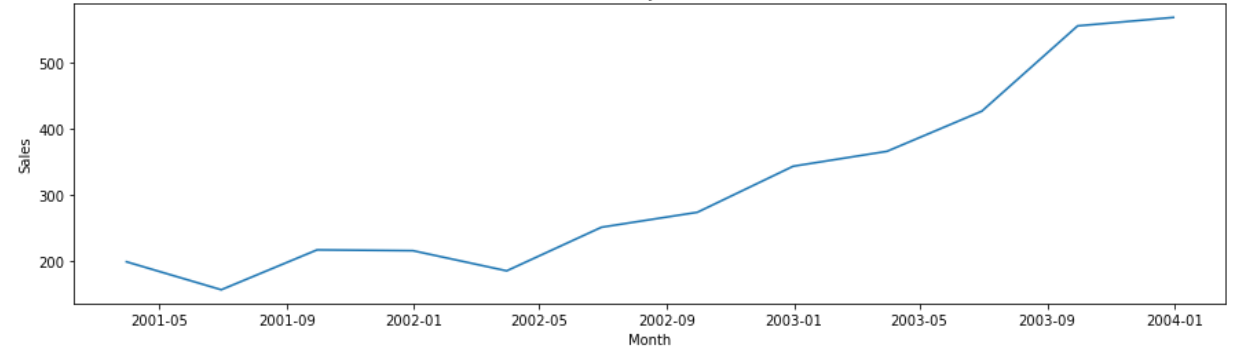
## # mean aggregation

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

Quarterly (mean)



Quarterly (mean)



# 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.