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), APIs, social media, statistical services
    - We can combine data (e.g. use features from multiple sources) to provide more information to the predictive modelling technique, helping it make better predictions
  - **Preprocess data**: Organize your selected data by formatting, cleaning, encoding and resampling.
    - Cleaning data is the fixing/deleting/filling in missing data
    - Encoding is the conversion of labeled/categorical text-based data into numerical data
    - Resampling is about changing the frequency of observations (in time-depended data)
  - **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 usually on a column-by-column basis
  - 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) the rest values of the column or (b) take into account values 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
- Correcting erroneous values
  - In a dataset, some values can be identified (using statistical analysis or visually e.g. see distribution or box plots) as obviously incorrect
    - E.g. find a number in a gender column, an age column with values below 0 or well over 100
  - Can be deleted and then treated as missing values

Cleaning data

- Handling outliers (=values that differ significantly from other observations)
  - Important step in data preprocessing, as outliers can distort your model's performance (e.g. in regression and distance-based algorithms like k-NN)
  - Decision to remove outliers should be made carefully, as sometimes outliers can represent important variability in the data
  - Outlier removal techniques can be found in Appendix
- 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  ...   0.0368989644153847 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
```

```
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

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

df.method()	description
dropna()	**Drop observations (rows) with at least one missing value**
dropna(axis=1)	**Drop the columns where at least one value is missing**
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	**Replace 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

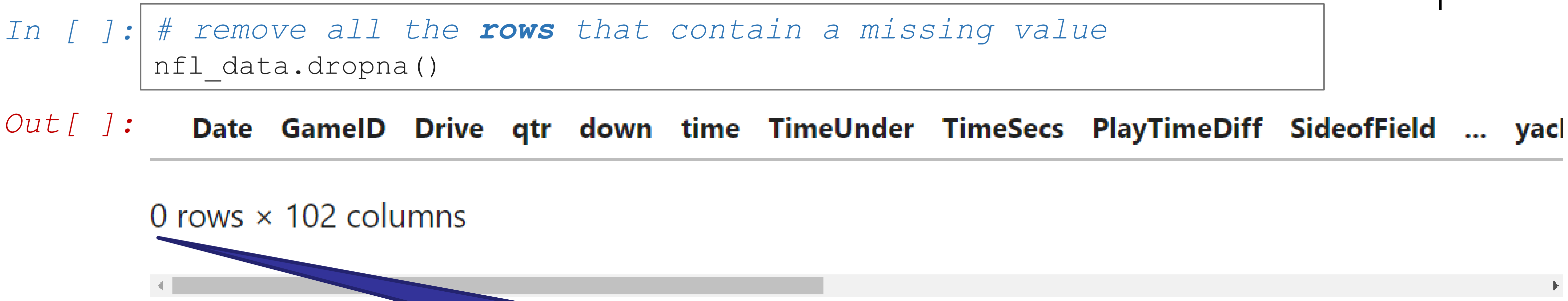
- Evaluate the number of missing values per column:

```
In [ ]: # 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
```

- 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 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  ...  yacl

        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(axis=1, 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  ...   0.0368989641538476    2009
2  2009-09-10  2009091000  ...                  NaN    2009
3  2009-09-10  2009091000  ...  -0.1562385319864913    2009
4  2009-09-10  2009091000  ...                  NaN    2009
```

Before imputation

`nfl_data.head()`

After imputation

```
         Date      GameID  ...               yacWPA  Season
0  2009-09-10  2009091000  ...            -0.010492    2009
1  2009-09-10  2009091000  ...   0.0368989641538476    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
```

Encoding categorical data

- Machine learning models require all input variables (features) to be numerical
- Categorical text-based data 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
```

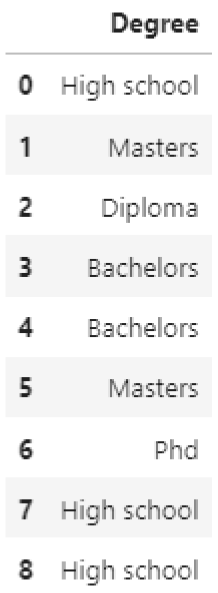
(website: http://contrib.scikit-learn.org/category_encoders)

Label Encoding

- Assigns a unique integer value to each category in a categorical feature
  - For example, if a feature has three categories: "Red", "Green", "Blue", Label Encoding could assign them numerical values like:
    - "Red" → 0, "Green" → 1, "Blue" → 2
- We use label encoding **when the categorical feature is nominal** (without inherent order)
- The numbers assigned are arbitrary and don’t carry any meaning in terms of 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

- Ordinal encoding is essentially label encoding, where each category is a assigned a unique value. However, ordinal encoding takes into account the order of the categories
- We use ordinal encoding **when the categorical feature is ordinal** (with natural, ordered values) and **retaining the order is important**
- Encoding should reflect 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**

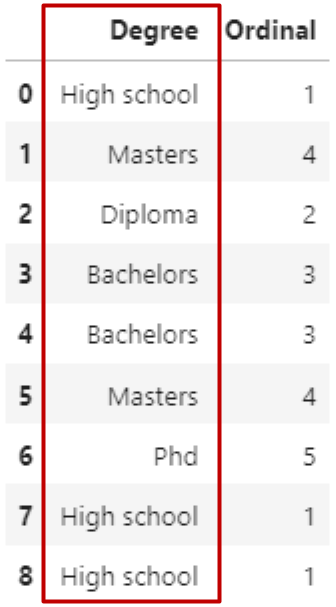
One-Hot Encoding

- We use this categorical data encoding technique when the features are nominal (do not have any order) and we want to avoid imposing ordinal relationships between categories (as in label encoding)
- For each label (value) 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 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 can be used over label encoding on nominal data when distance-based machine learning techniques will be used**

One-Hot Encoding: `OneHotEncoder`

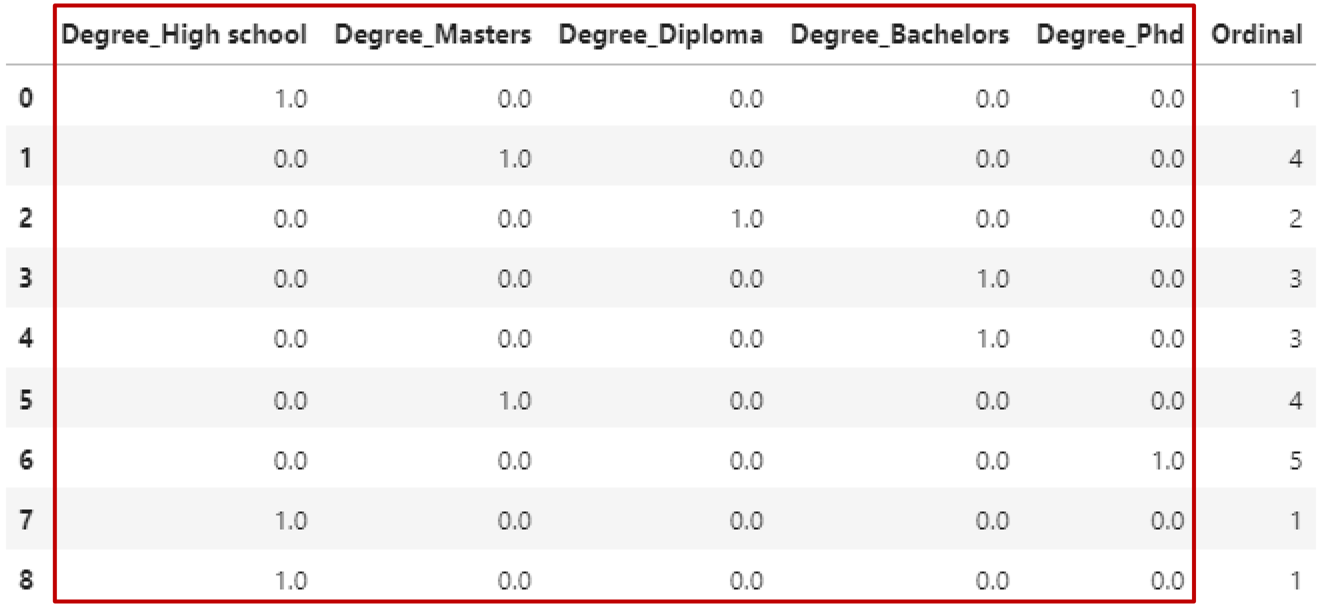
```
# Create object for One-hot encoding
onehot_encoder=ce.OneHotEncoder(cols=['Degree'], use_cat_names=True)
#fit and transform data
df_onehot = onehot_encoder.fit_transform(df)
df_onehot
```

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



	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

Dummy variables



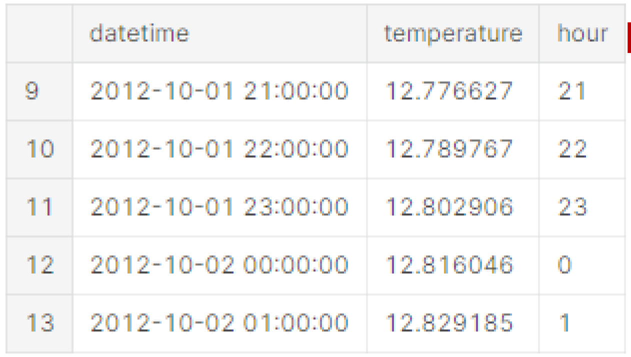
	Degree_High school	Degree_Masters	Degree_Diploma	Degree_Bachelors	Degree_Phd	Ordinal
0	1.0	0.0	0.0	0.0	0.0	1
1	0.0	1.0	0.0	0.0	0.0	4
2	0.0	0.0	1.0	0.0	0.0	2
3	0.0	0.0	0.0	1.0	0.0	3
4	0.0	0.0	0.0	1.0	0.0	3
5	0.0	1.0	0.0	0.0	0.0	4
6	0.0	0.0	0.0	0.0	1.0	5
7	1.0	0.0	0.0	0.0	0.0	1
8	1.0	0.0	0.0	0.0	0.0	1

Drawbacks of One-Hot

- 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 will require 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.

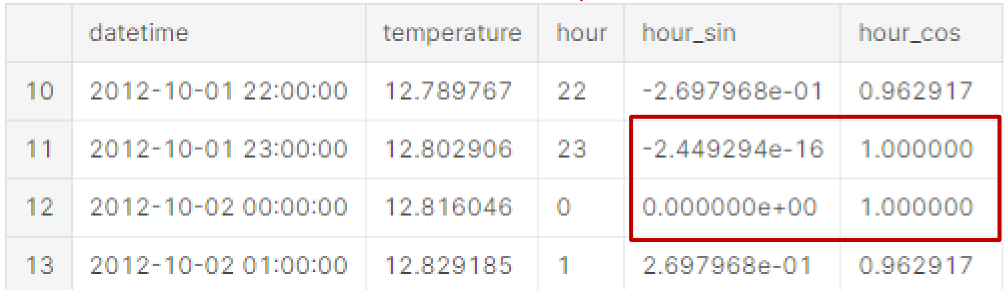
Cyclical feature encoding

- When dealing with time-dependent data (e.g. months, days, hours) it’s important to encode the properties of time properly
  – Decompose datetime string to a set of new features: month (1-12), day of the month (1, 2, .. 31), hour (0-23), minute (0-59), day (Sun→1, Mon→2, .. Sat→7)
    - 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
  - 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 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

- 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 rescaling
  - Rescale **each feature** individually into a given range, e.g. [0, 1]

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

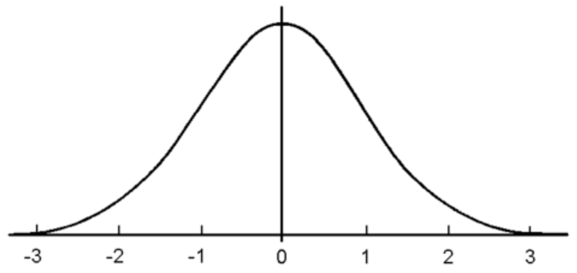
  - Scikit-learn module: MinMaxScaler or MaxAbsScaler
    - MinMaxScaler: Transforms features by scaling each feature to a given range ($x_{min} \rightarrow x_{max}$).

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

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

  - 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 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
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?

- ANS: When features have different scales + distance-based or gradient-based predictive techniques are to be used
- The choice of using normalization (MinMax scaler), standardization (Standard scaler) or robust standardization (Robust scaler) depends on the nature (distribution) of data (features)
- Normalization is applied on features having different ranges without outliers or with outliers which don’t significantly affect the mean if you want to keep the values within a specific range
- Standardization is applied on features are that approximately normally distributed and either have no outliers or have outliers but they don’t significantly affect the mean
- Robust standardization is applied on features with significant outliers or skewed (long-tailed) distributions

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

- Depends on the nature of your data
- When to scale:
  - If an encoded feature has many levels, scaling might help normalize its influence compared to other features
  - If using ordinal encoding and values represent a meaningful quantitative scale:
    - the ordinal values are evenly spaced (e.g., survey responses like “Strongly Disagree” to “Strongly Agree” encoded as 1 to 5), min-max scaling will preserve the underlying quantitative relationships (other scaler may not be appropriate → distort relationships)
- When not to scale:
  - If the values are purely categorical with arbitrary intervals
    - In ordinal encoding for example, education levels (e.g., "High School" → 1, "Bachelor's" → 2, "Master's" → 3) don’t have meaningful, fixed numerical intervals
    - In label encoding which assigns arbitrary integer values to categories (e.g., "Red" = 0, "Green" = 1, "Blue" = 2), these integers don't reflect a true order or distance

  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 creates binary (0 or 1) columns. Scaling these values won’t add useful information, as they already clearly indicate the presence (1) or absence (0) of a category.
  - Categorical represenation is distorted
    - Scaling one-hot encoded features would produce non-binary values that don’t make intuitive sense in the context of categorical representation. For instance, a scaled value of 0.5 would have no clear interpretation between 0 and 1.
  - Distance-based algorithms
    - Distance-based algorithms (like KNN) can handle binary features effectively without scaling by using a custom distance metric

When to normalize or standardize target?

- Normalization and standardization of target variable is generally not required but can be beneficial especially in regression tasks, under specific circumstances:
  - **Wide Range of Values**: If the target variable has a wide range of values (e.g., income ranging from hundreds to millions), models like **linear regression** might give undue weight to large values. Thus, normalizing or standardize the target variable can help the model avoid being biased toward large numbers
  - For algorithms like **gradient descent** (used in linear regression, neural networks), scaling the target can be essential because the magnitude of the target values can affect the convergence speed

Data Transformation: Scaling data

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

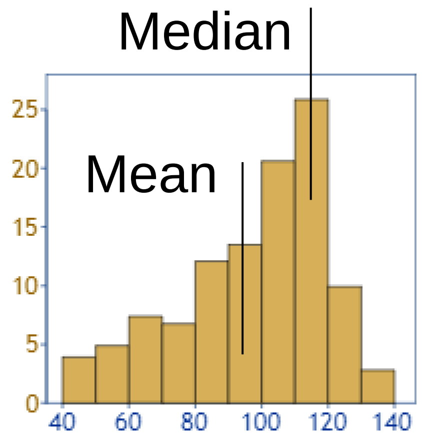
$$\sqrt{0.29^2 + 0.96^2} = 1$$

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

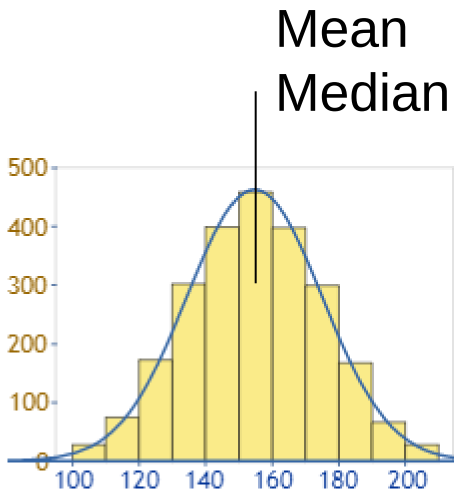
  - Normalizing to unit norm helps ensure that each row contributes equally to the distance metrics used in algorithms, preventing any single row from disproportionately affecting the results.
    - Useful for sparse datasets (lots of zeros) to prevent zeros from skewing data
  - Commonly used in text classification or clustering
    - dot product of two 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

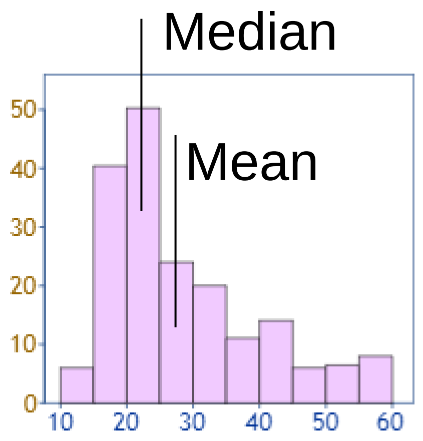
- Data is skewed when its distribution curve is asymmetrical as compared to a normal distribution curve that is perfectly symmetrical
- *Skewness* is the measure of the asymmetry
  - The skewness for a gaussian or normal distribution is 0



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



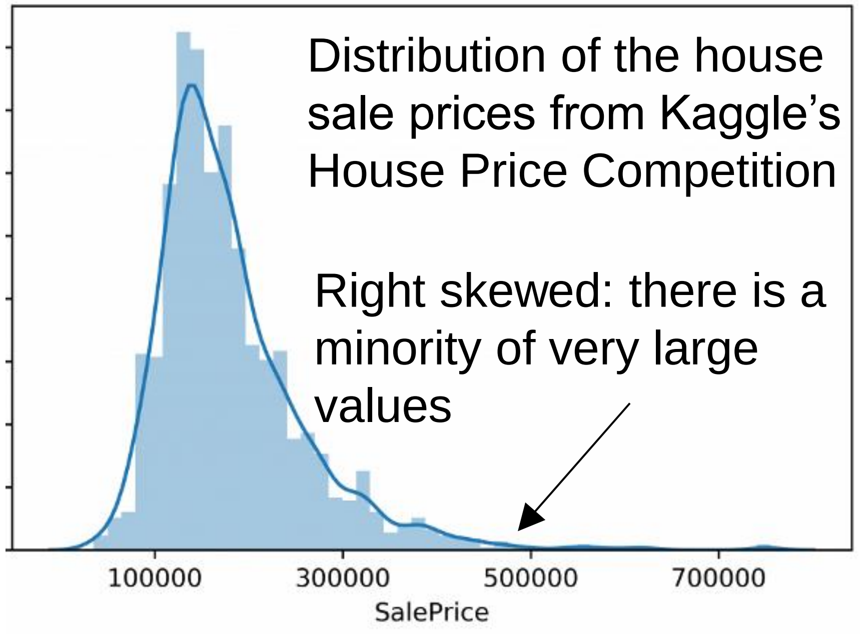
No skew (symmetrical distribution)



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)
  - If a dataset can be transformed to be statistically close enough to a Gaussian dataset, some machine learning algorithms such as Linear Regression, Logistic Regression, SVM (with RBF kernel), 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

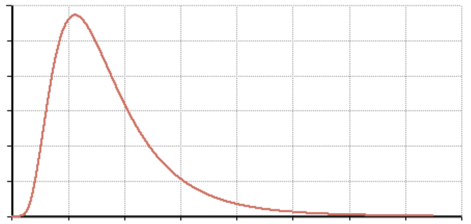
- Square Root (SQRT) transformation

```
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



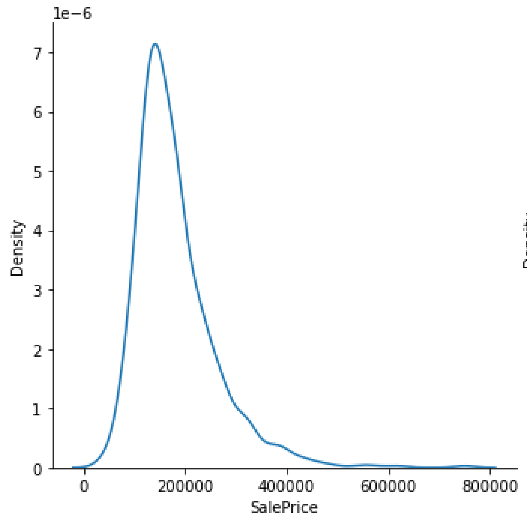
- applicable on features with strictly positive values (sqrt and log cannot be applied on negative values)

- 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

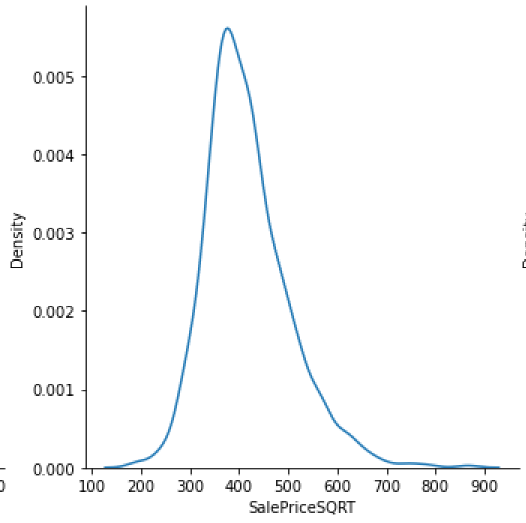
```
from scipy.stats import boxcox
df['bc_col'] = boxcox(df['col'])

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

Unskewing transformations: Examples

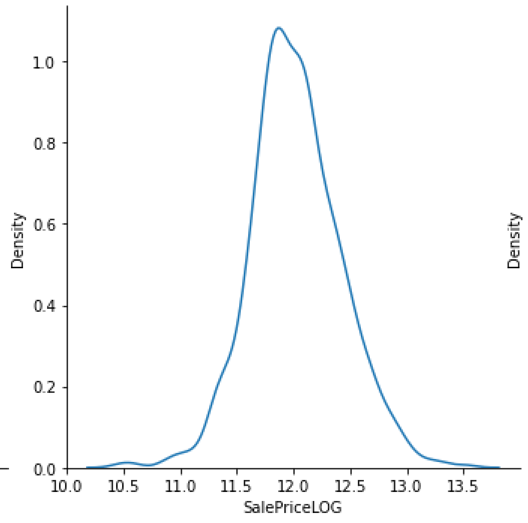
- Pandas .skew() method can be used to measure skewness of data



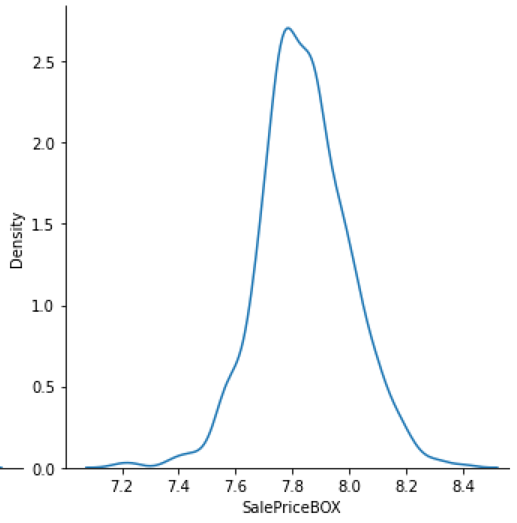
Original SalePrice column
Skewness: 1.8828757



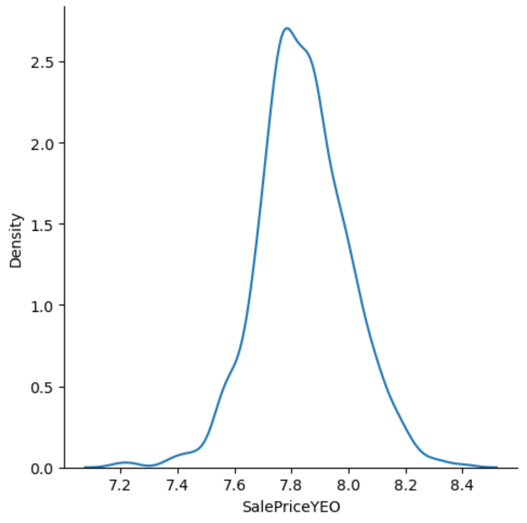
SQRT transformation
Skewness: 0.9431527



LOG transformation
Skewness: 0.1213351



BoxCox transformation
Skewness: -0.0086529



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

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

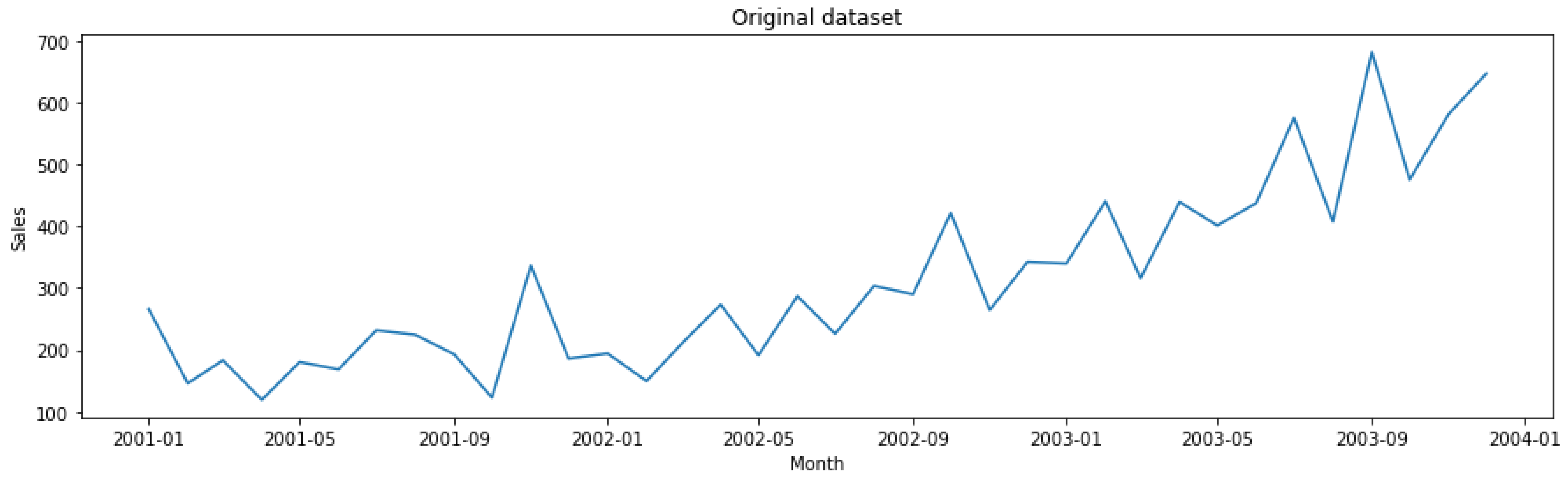
```
from pandas import read_csv
from datetime import datetime

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

# convert Month feature (e.g. from 1-01 20 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
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
```

```
            Sales
Month
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

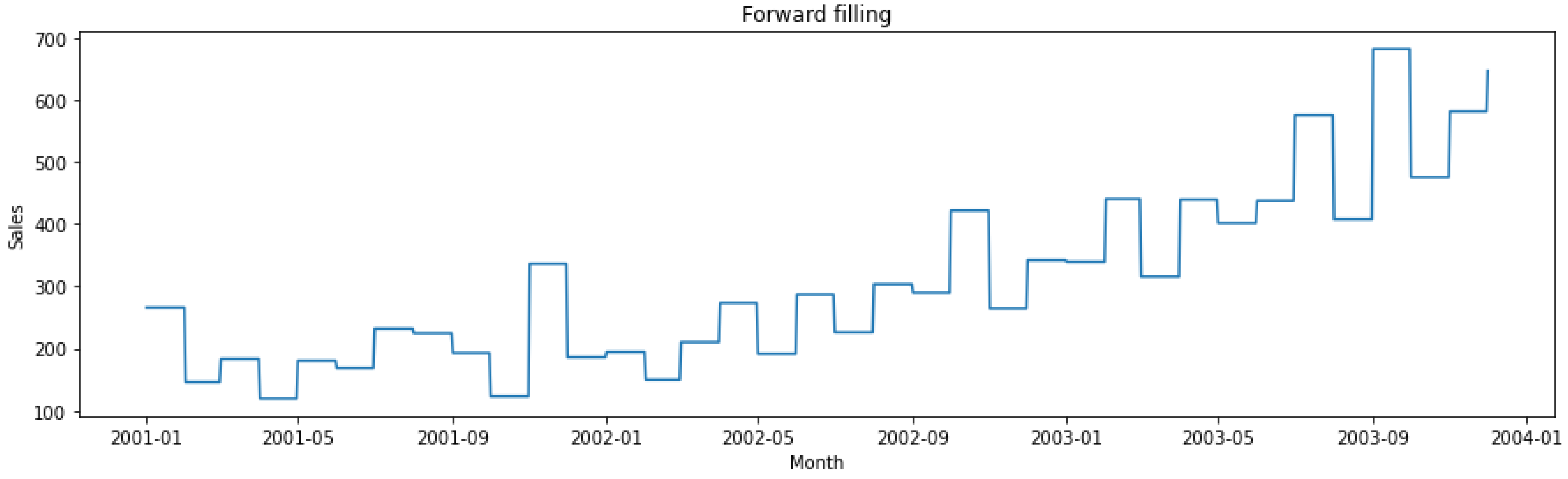
- Resample by day

```
# 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.

```
            Sales
Month
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

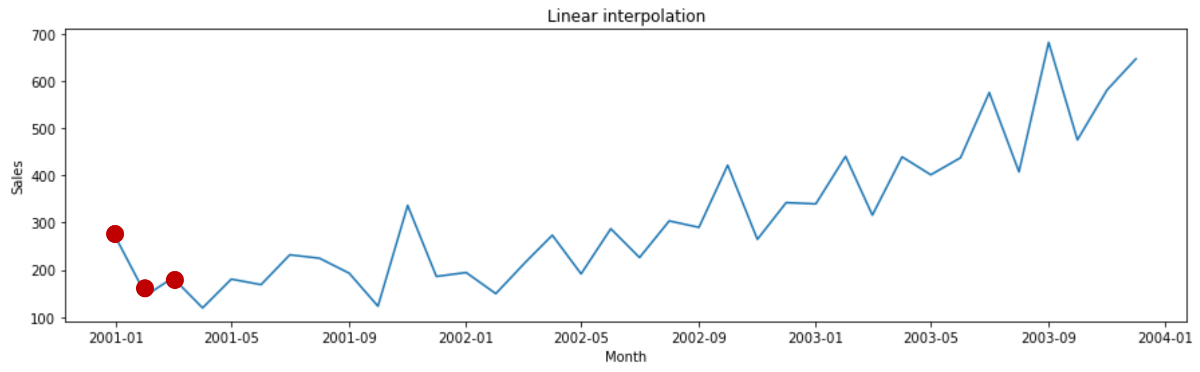
`.ffill([limit])`	Forward fill the values.
`.backfill([limit])`	Backward fill the new missing values in the resampled data.
`.bfill([limit])`	Backward fill the new missing values in the resampled data.
`.pad([limit])`	Forward fill the values.
`.nearest([limit])`	Resample by using the nearest value.
`.fillna(method[, 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 methods.

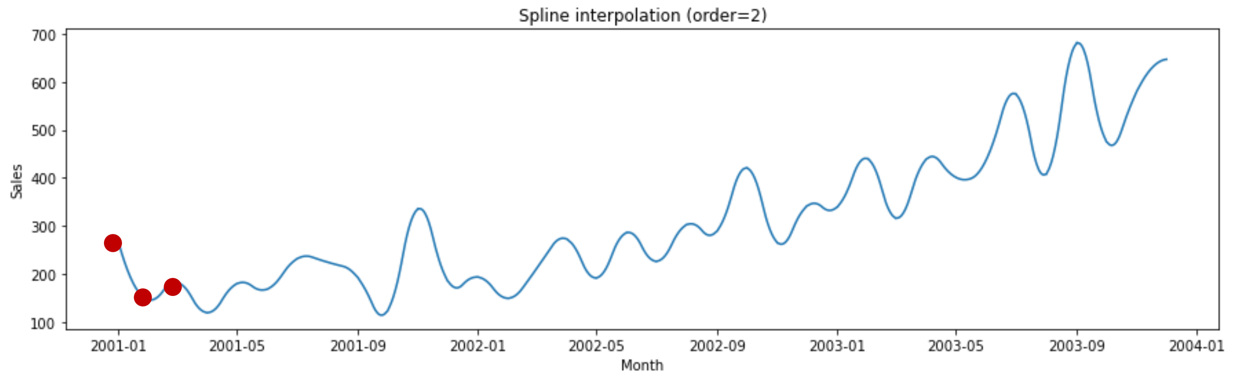
Resampling data – Upsampling

- 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()
```



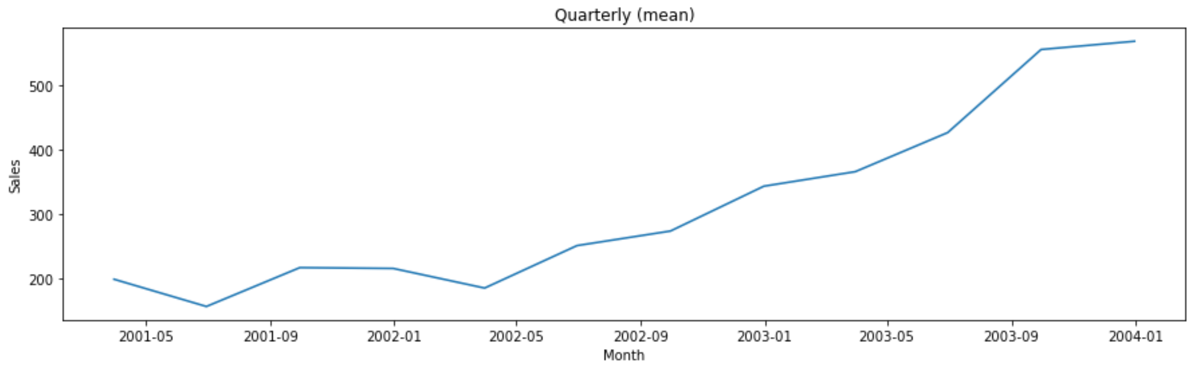


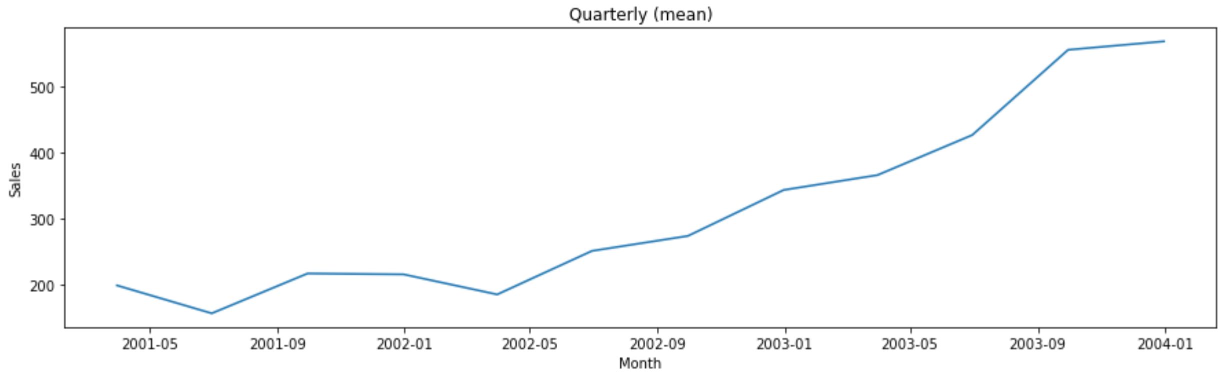
Resampling data – Downsampling

- 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()
```





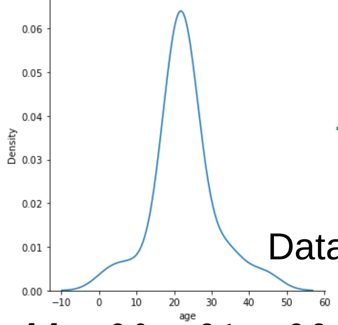
Resampling data – Downsampling aggregation strategies

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

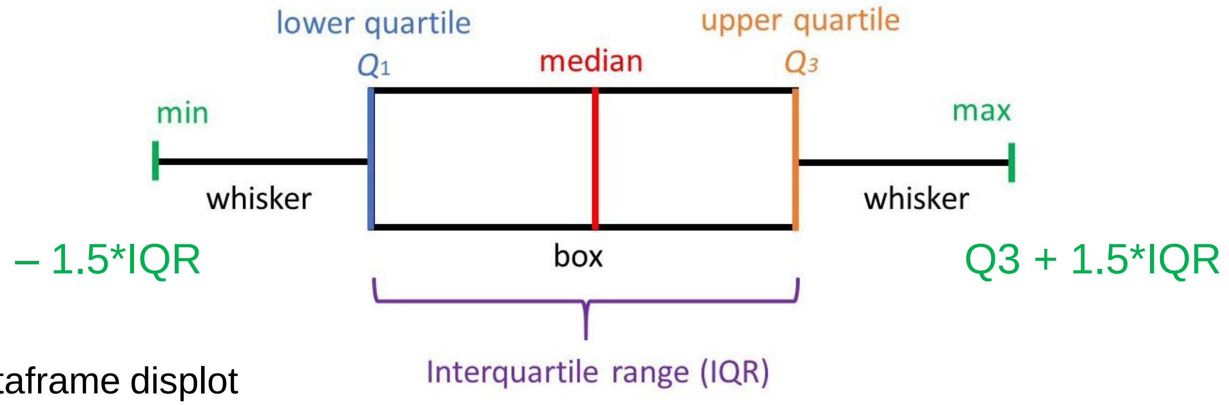
APPENDIX

Removing outliers

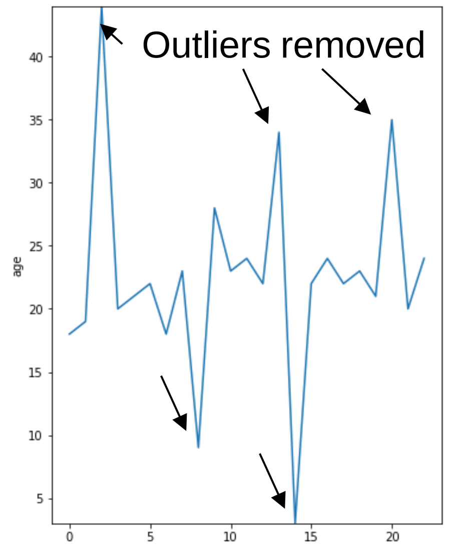
- Methods for removing outliers on each feature independently:
  – Interquartile Range (IQR) method
    - Outliers are considered data points:
      – below Q1 – 1.5*IQR
      – above Q3 + 1.5*IQR



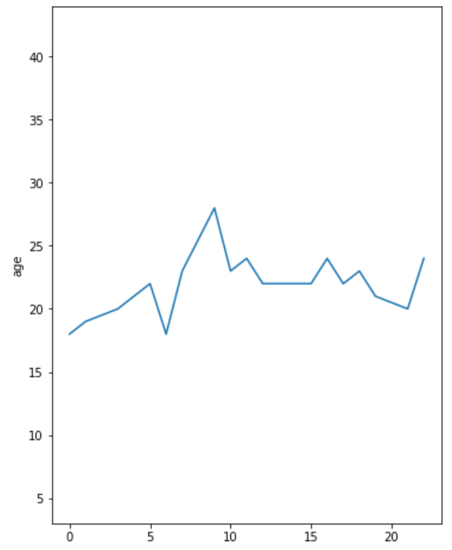
Dataframe displot



```
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()])
```



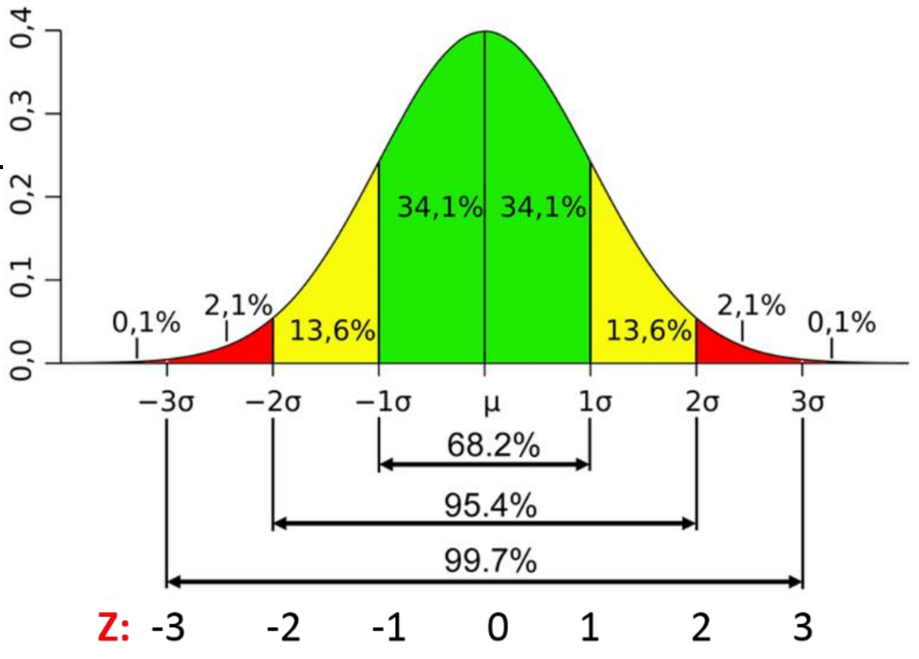
Initial dataframe



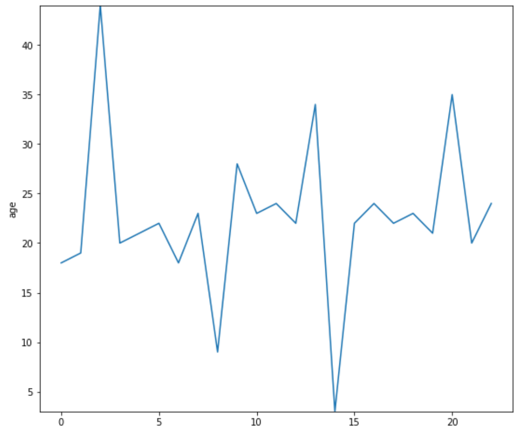
After outlier removal

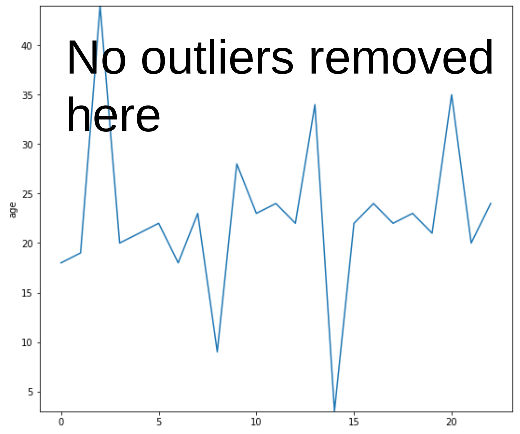
Removing outliers

- 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

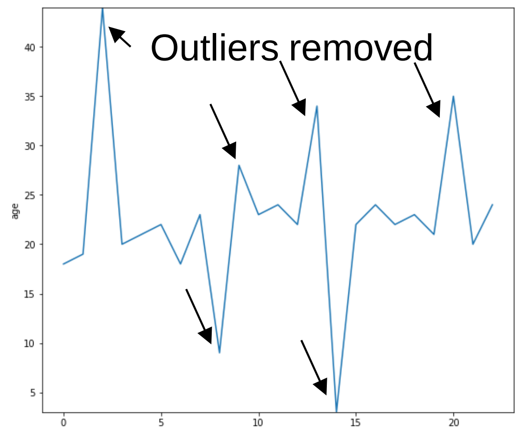
– Median and Median Absolute Deviation (mad) method

- 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

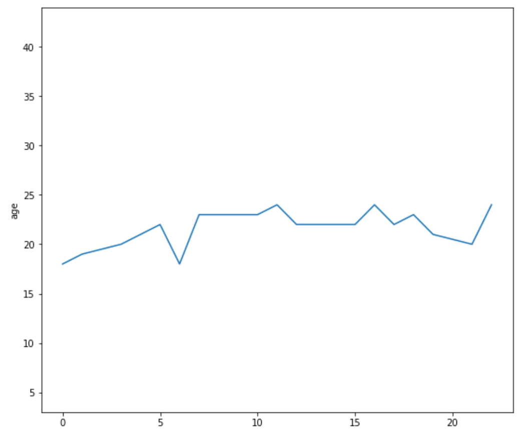
$$MAD = Median(|X_i - median|)$$

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