ML lifecycle I

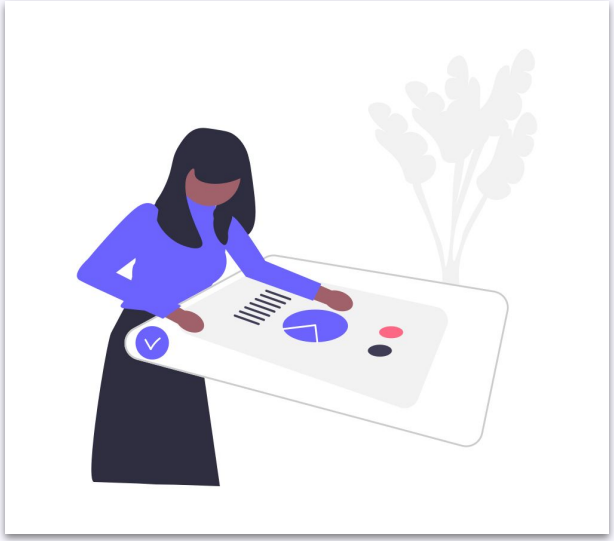
Big data management

Recap - logistics

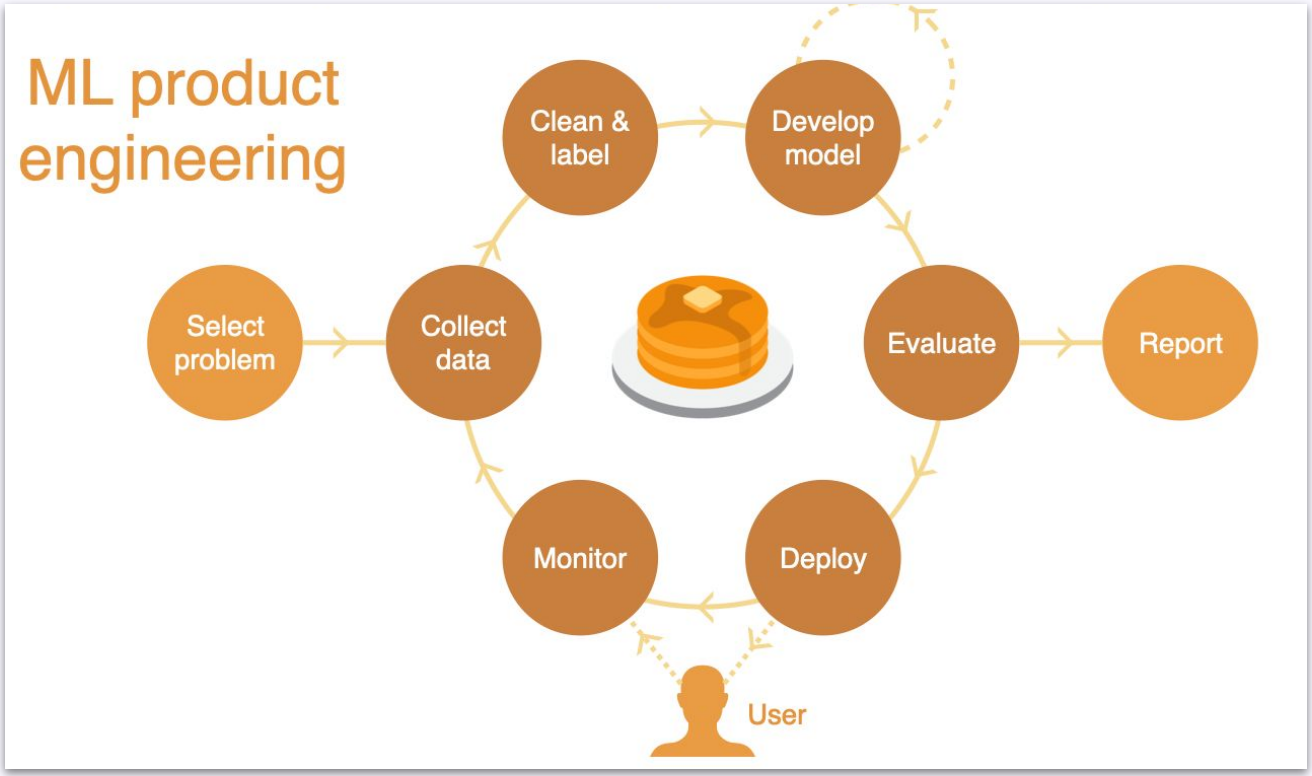
2 assignments not mandatory, BUT

exam based on assignments and lecture material

exercises: practice the tools introduced in the lectures



ML lifecycle



Today

Preprocessing and machine learning pipelines

Exercises: introducing the first assignment + sklearn pipelines

ML lifecycle

In this chapter, you will go through an example project end to end, pretending to be a recently hired data scientist in a real estate company.[1] Here are the main steps you will go through:

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Look at the big picture

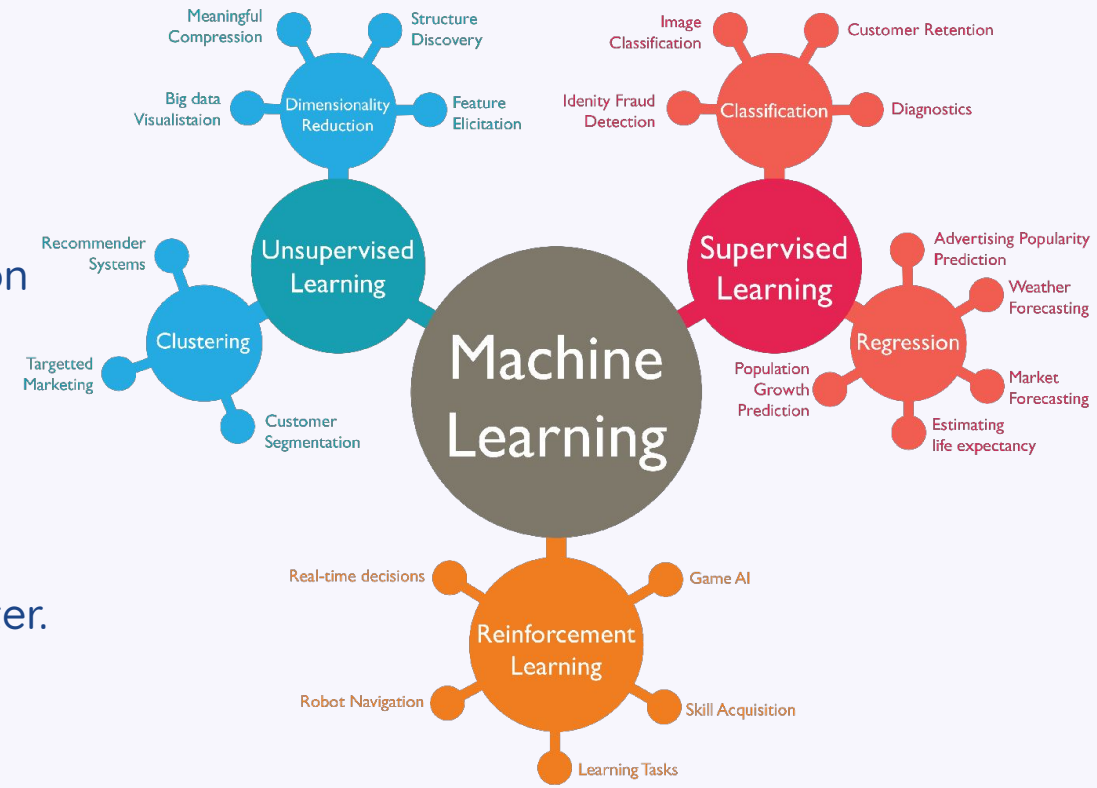
- Frame the problem

  Price prediction:

  - supervised learning
  - multiple, univariate regression
  - batch learning

- Find an evaluation criteria

  We will talk about this in detail later.

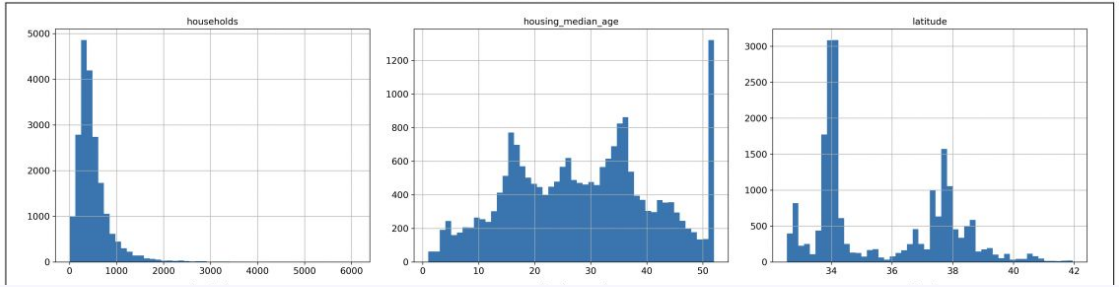


Get the data + initial inspection

- Create a working environment
- Get access to data
- Inspect (a subsample of) data
  - pandas
    - DataFrame.info()
    - DataFrame.describe()
    - DataFrame.value_counts()
  - matplotlib + pandas
    - DataFrame.hist()

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrc
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000(
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.00000



Splitting the test set

Good models can generalize on unseen data.

To understand how well the model generalizes, we split the data into:

- training data
  - for learning the model
- test data
  - generalization ability
  - the training data should have no statistical information from this test set

Splitting the test set

The test set needs to be separated **before** any preprocessing of the data that involves dataset statistics.

For example, if you scale the data before you split into train and test

the training data will contain information about the *test data*

=> data leakage

Also pay attention to:

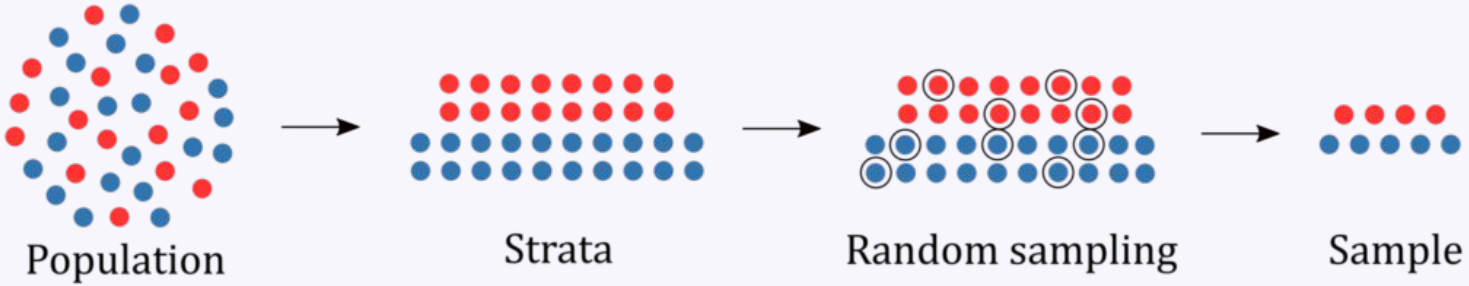
- randomness
  - set the random number generator's seed
- changing datasets

Great read about data processing: Common pitfalls and recommended practices

Splitting the test set

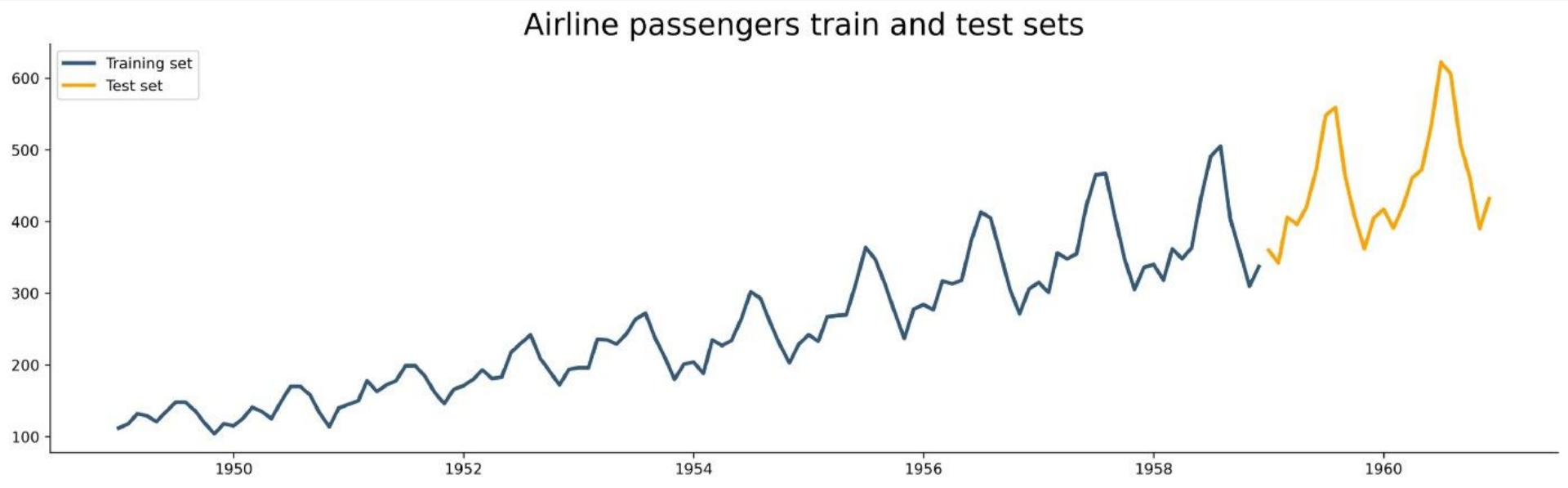
We sometimes need to ensure the test set is representative

=> stratified sampling



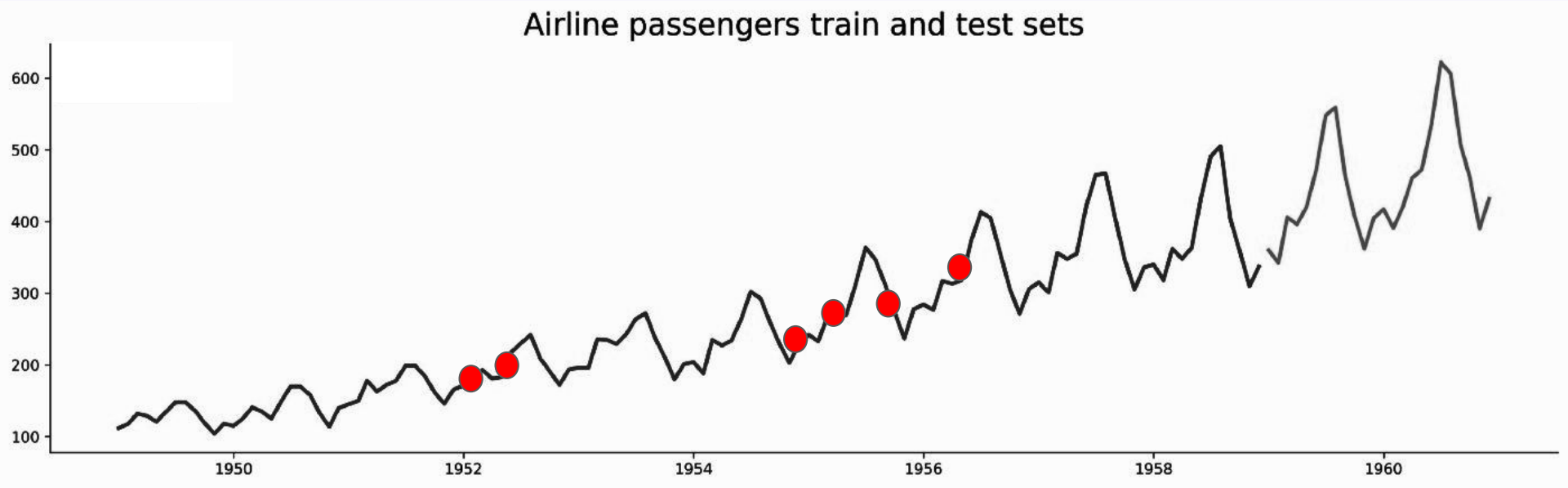
- from sklearn.model_selection import **train_test_split** - set stratify=column
- from sklearn.model_selection import **StratifiedShuffleSplit** (for cross-validation)

Timeseries train/test split



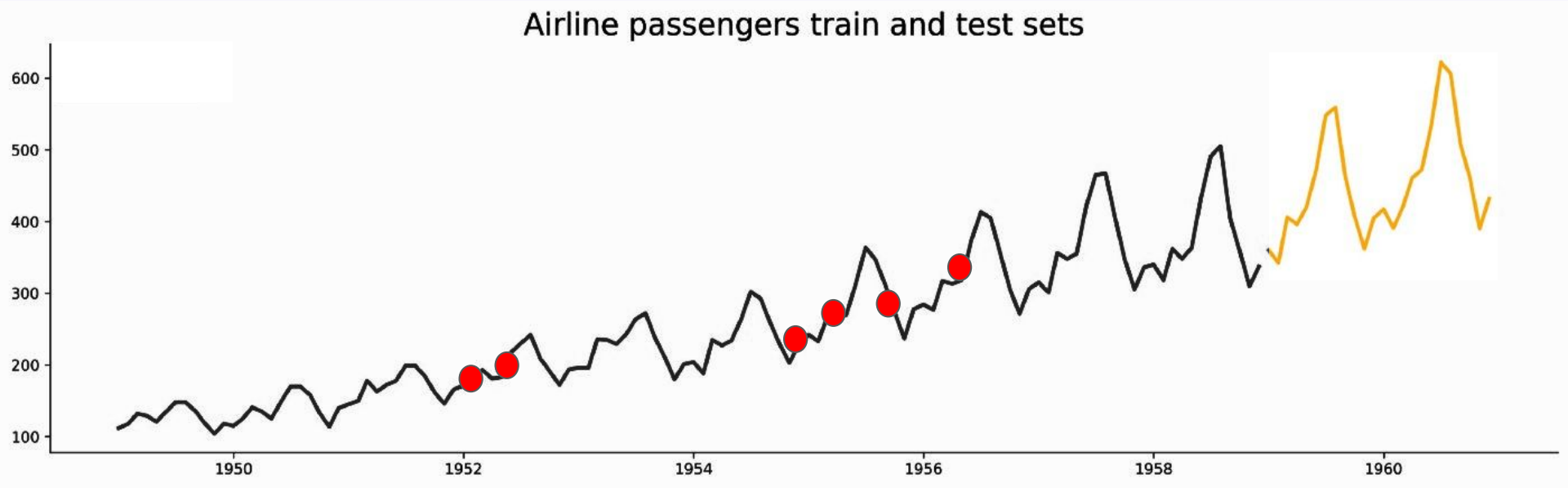
Data should **not** be shuffled before splitting into train and test if there is temporal dependence.

Timeseries train/test split



Data should **not** be shuffled before splitting into train and test if there is temporal dependence.

Timeseries train/test split



Data should **not** be shuffled before splitting into train and test if there is temporal dependence.

Training and testing

Good models can generalize on unseen data.

To understand how well the model generalizes, we split the data into:

- training data
  - for learning the model
- test data
  - generalization ability
  - the training data should have no statistical information from this test set

To avoid overfitting: the training dataset is split further

- validation set
  - a resampling procedure used to evaluate machine learning models

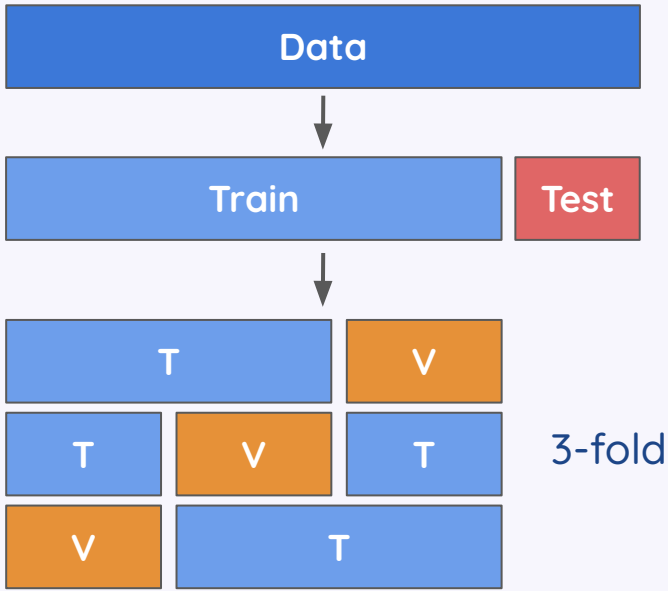
Timeseries train/test split

Data should **not** be shuffled before splitting into train and test if there is temporal dependence.

- from sklearn.model_selection import **train_test_split** - set shuffle=False
- from sklearn.model_selection import **TimeSeriesSplit** (for cross-validation)

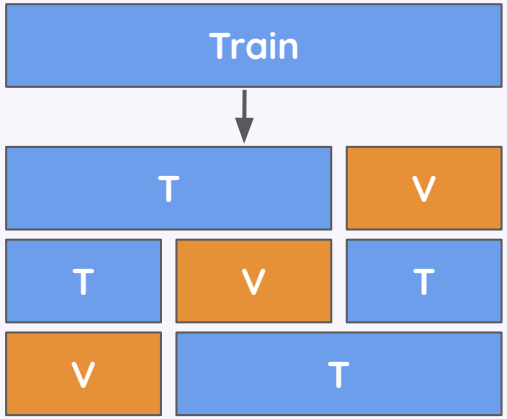
k-fold cross-validation

- The *training* dataset is partitioned into **k** equal sized sets
- k-1 sets are used for training and 1 subsample for validation
- The k results are averaged to produce a score



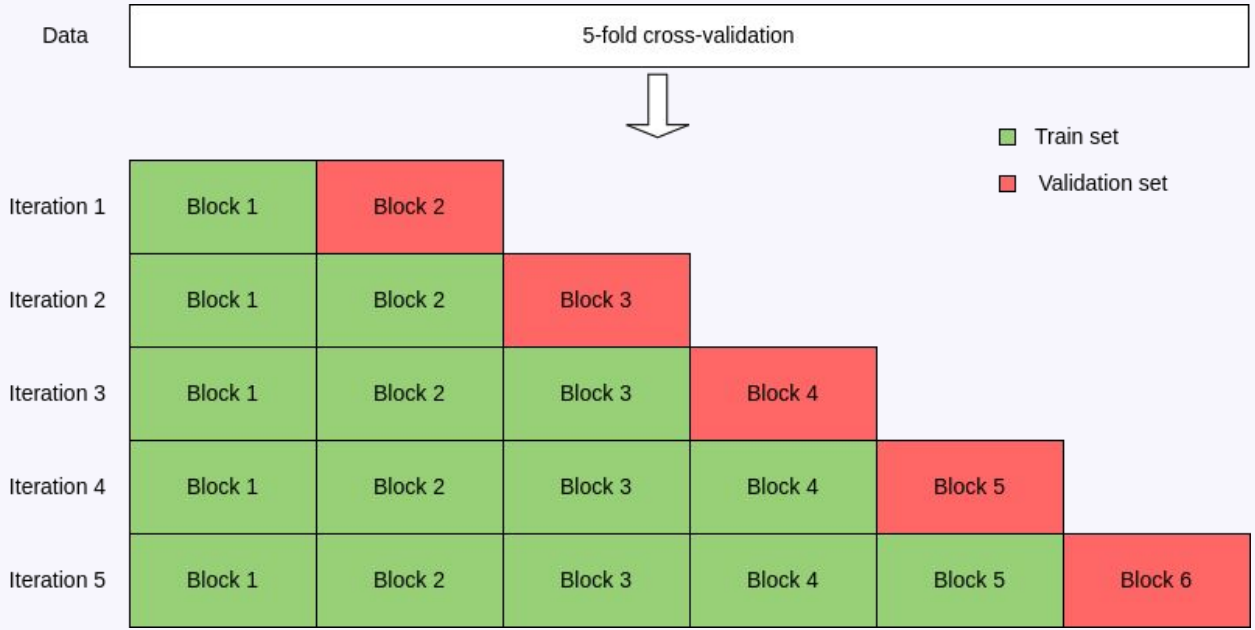
3-fold cross-validation

Cross-validation for time series data

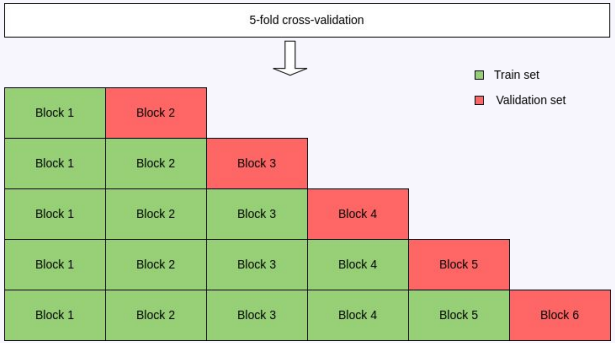


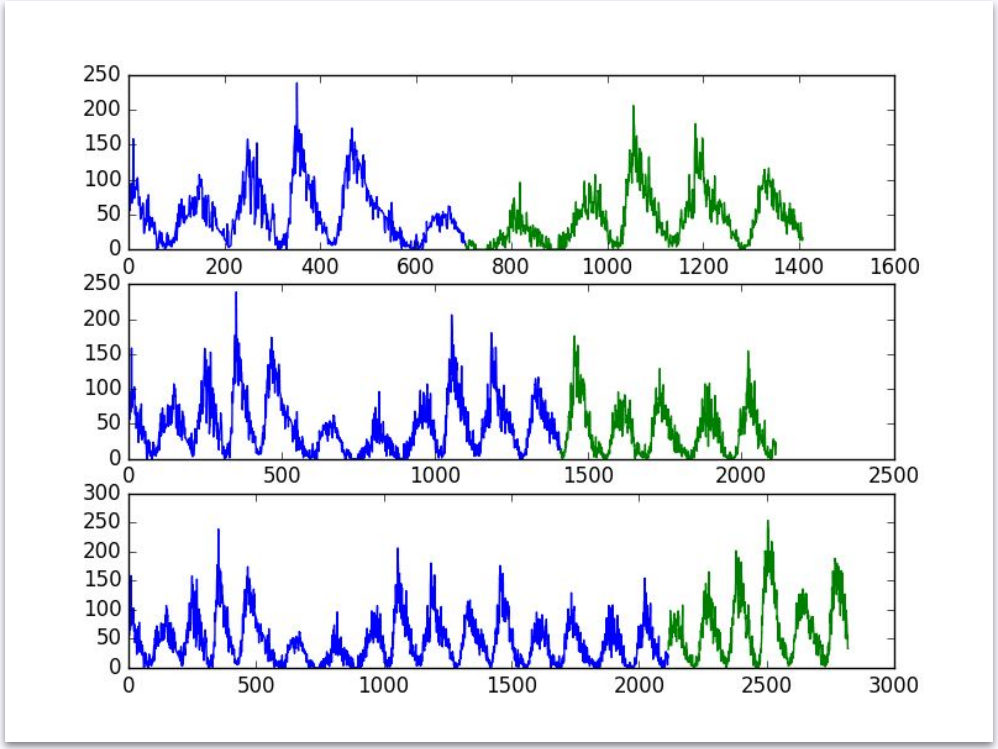
data that is not time dependent

timeseries data



Cross-validation for time series data

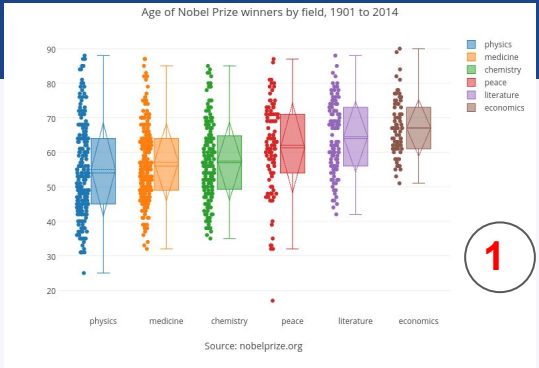


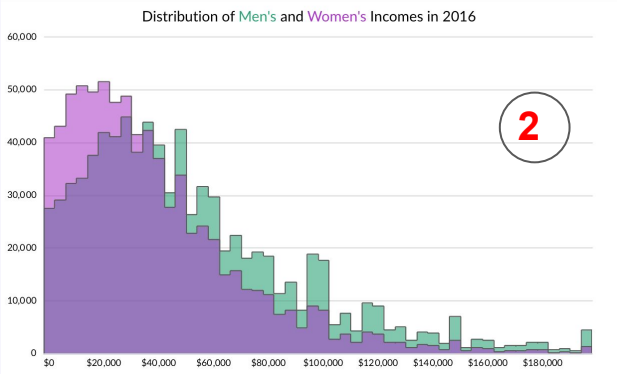


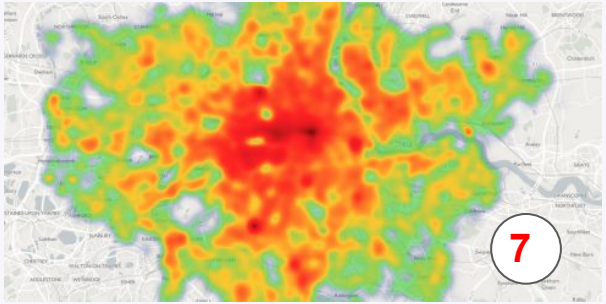
In this chapter, you will go through an example project end to end, pretending to be a recently hired data scientist in a real estate company.[1] Here are the main steps you will go through:

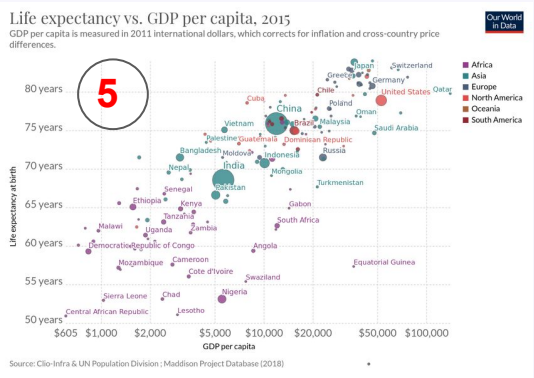
1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

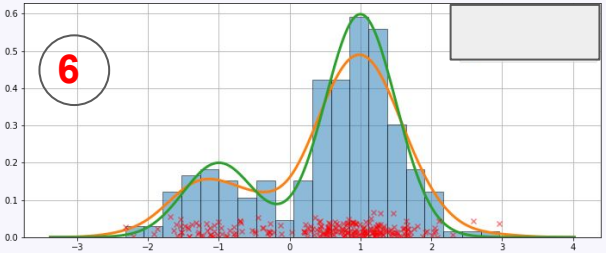
Data visualisation

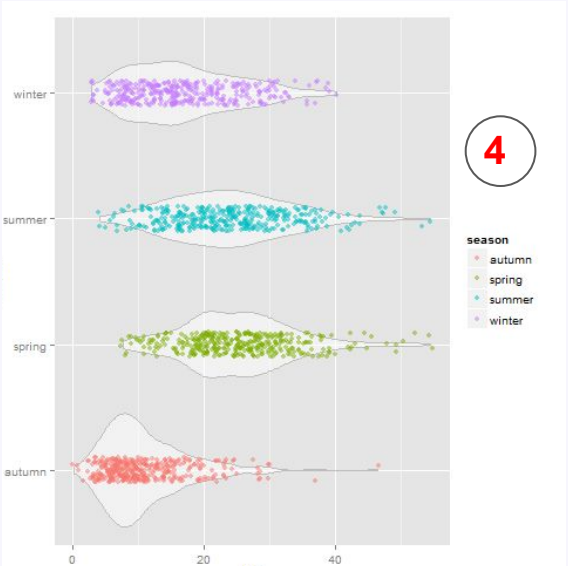






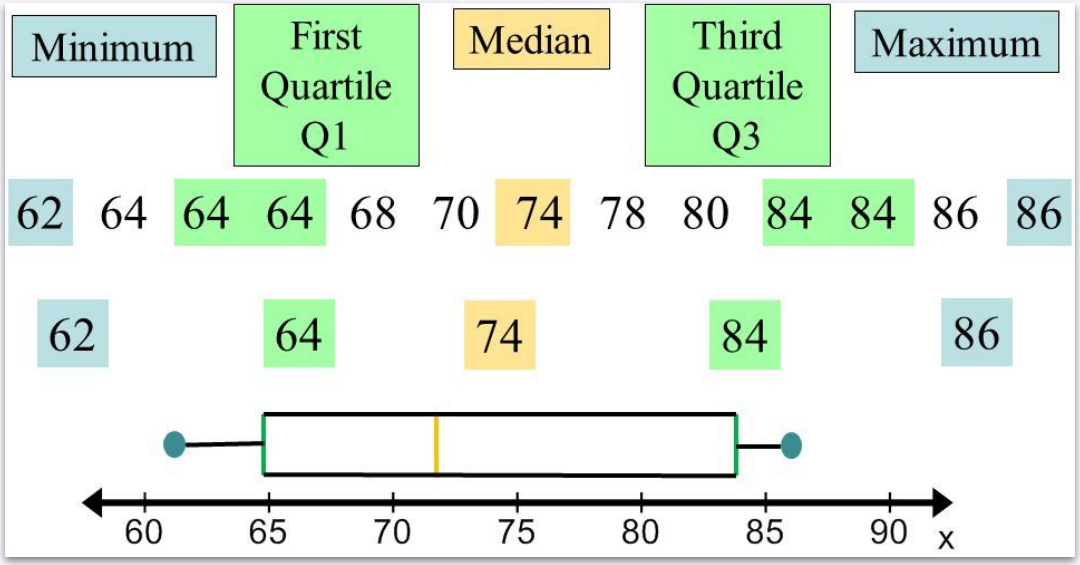






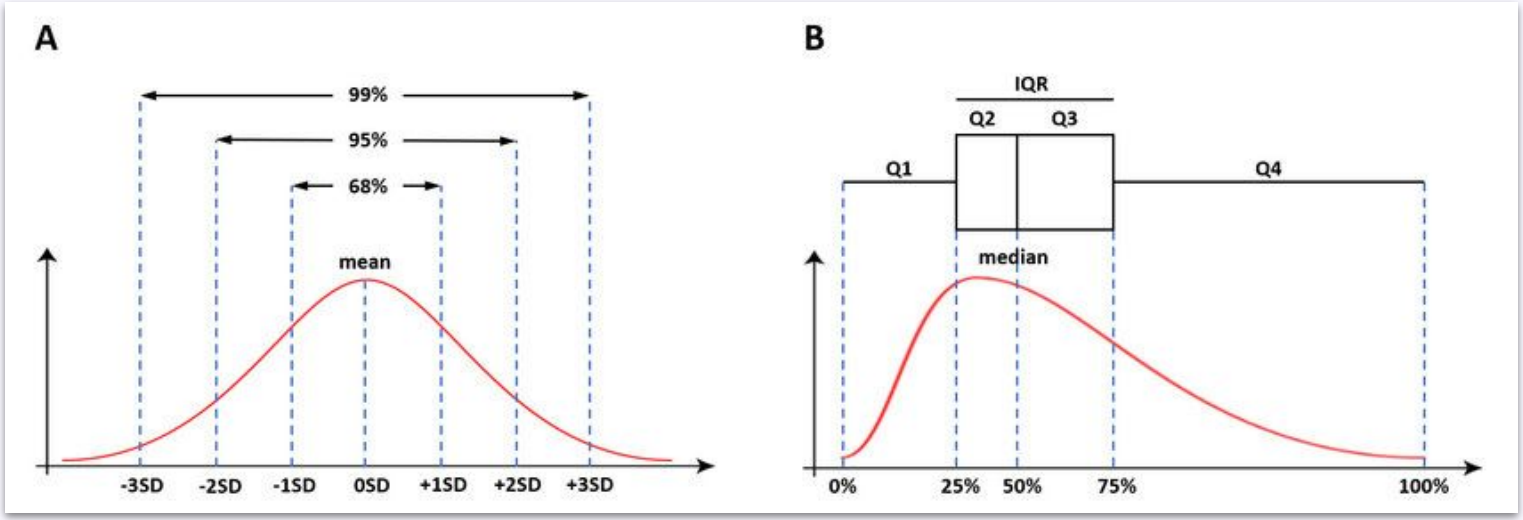
bar plot, box plot, scatter plot, histogram, kernel density plot, heatmap, violin plot

Location



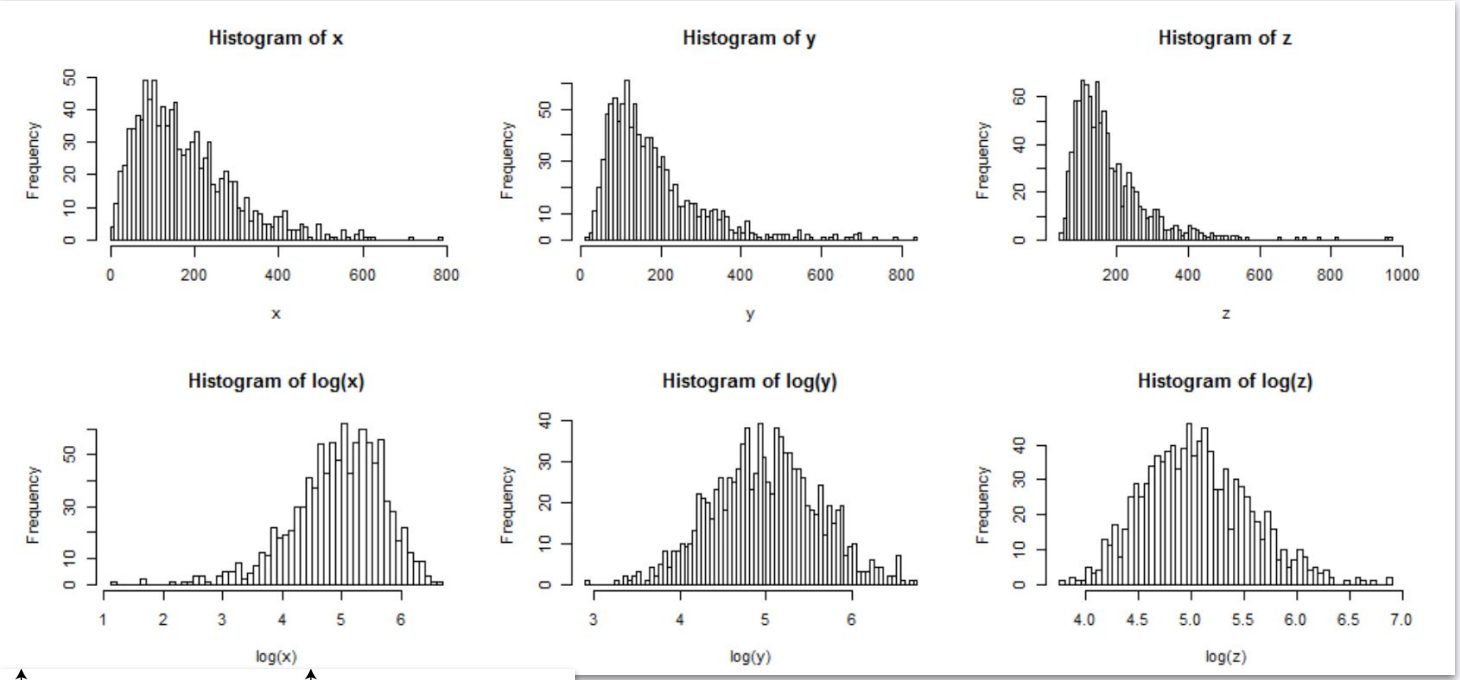
- Mean
- Median
- Mode
- Quartiles

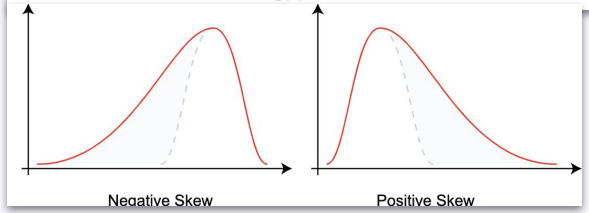
Variability/spread



- Variance
- Standard deviation
- Median absolute deviation
- Interquartile range

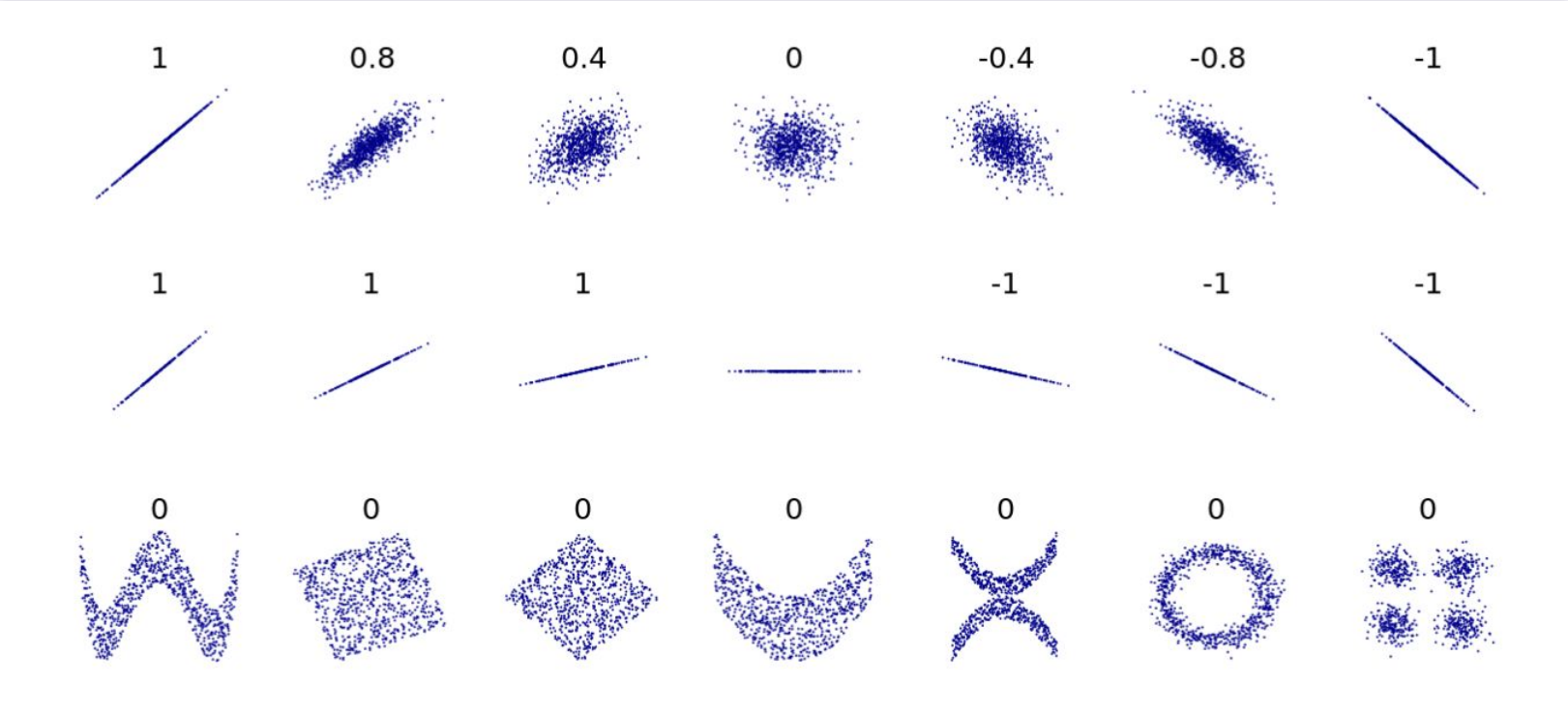
Shape





- Skewness
- Galton's measure of skewness

Multivariate analysis/dependence



Pearson’s correlation coefficient

Data exploration - summary statistics

	Importance
Location	data standardisation
Spread	data standardisation; outlier detection
Shape	identify imbalanced datasets; apply transformation
Dependence	strength of associations between observations

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Data cleaning

Based on your data exploration you should first:

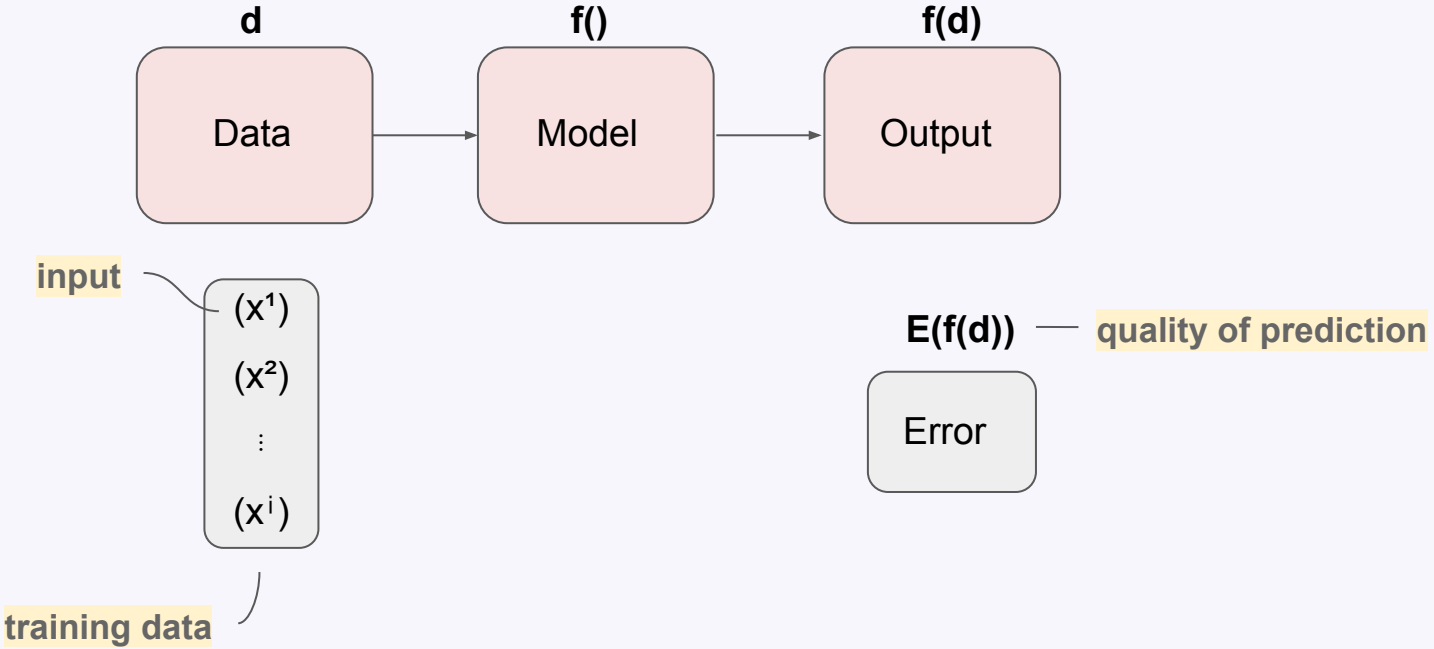
- drop any columns/features that are not relevant to the task
- combine datasets
  - pandas:
    - DataFrame.merge
    - DataFrame.resample
    - merge_asof
  - for timeseries make sure timestamps match:
    - downsample
    - upsample
- drop any columns/features that are highly correlated

Data preprocessing for machine learning

= steps taken to prepare data for analysis or a machine learning model

A machine learning model is simply a mathematical function.

Learning = finding this function that fits the input data

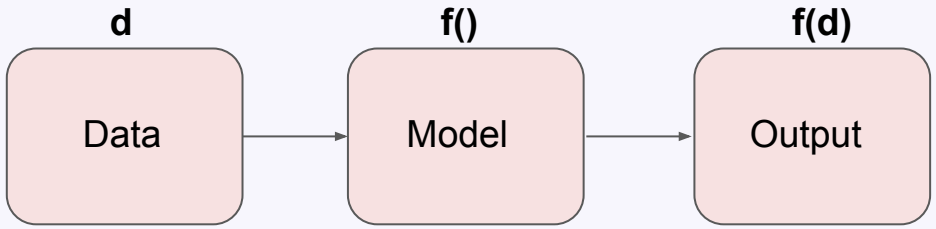


Data preprocessing for machine learning

= steps taken to prepare data for analysis or a machine learning model

A machine learning model is simply a mathematical function. Therefore:

- **missing data** needs to be handled
- data needs to be represented **numerically**
- data comes from different sources => different **scales** => rescaling
- some of the data is **noisy** and not useful for learning



Data preprocessing for machine learning

= steps taken to prepare data for analysis or a machine learning model

A machine learning model is simply a mathematical function. Therefore:

- **missing data** needs to be handled
- data needs to be represented **numerically**
- data comes from different sources => different **scales** => rescaling
- some of the data is **noisy** and not useful for learning

There are exceptions to these rules, mainly in the case of tree based models.

Data preprocessing

Missing values

Outlier detection

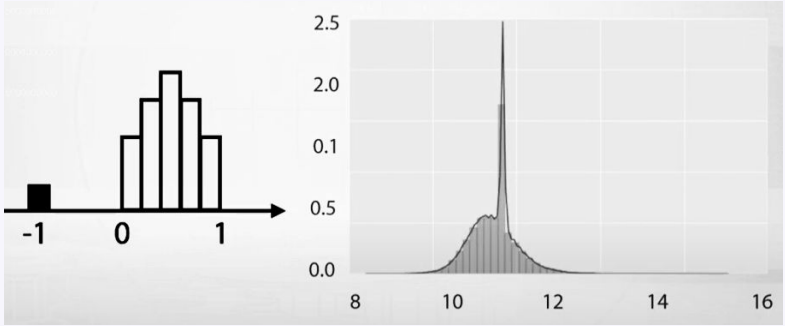
Feature scaling

Feature encoding

Missing values

Sources of missing values:

- Survey non response
- Sensor failure
- Changing data gathering procedure
- Database join



Types of missing values: NaN values, empty strings, outliers

First step: identify missing values

Plotting the data can help identify hidden missing values

Strategies for handling missing values

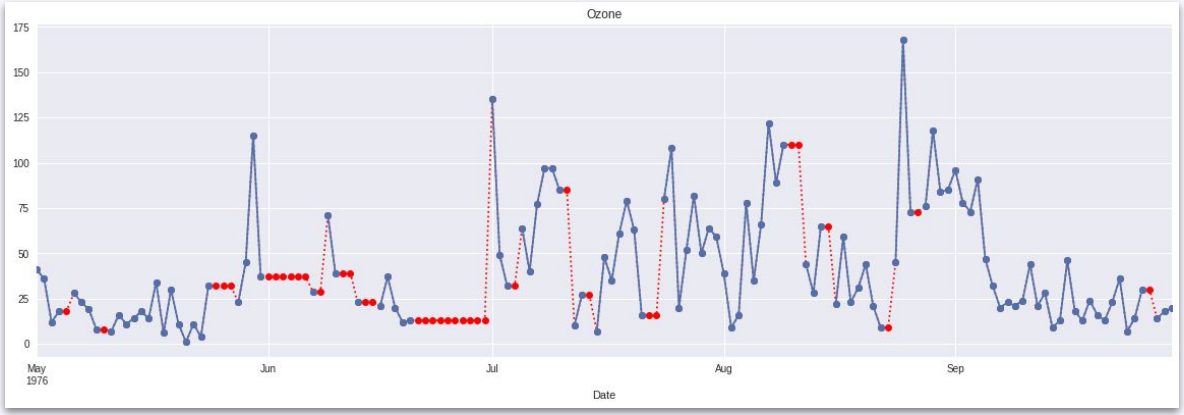
- Discard columns/rows with missing values
  - discard rows (samples) when only a few values are missing
  - discard a column when most of the values are missing for that feature

ID	C1	C2	C3	C3
A	92	4	2	2
B	9	13	NA	1
C	NA	NA	8	17
D	8	33	19	1
E	22	32	9	1

ID	C1	C2	C3	C3
A	92	4	2	2
D	8	33	19	1
E	22	32	9	1

Strategies for handling missing values

- Discard columns/rows with missing values
  - discard rows (samples) when only a few values are missing
  - discard a column when most of the values are missing for that feature
- Data imputation = fill in the missing values
  - based on the other values for a particular feature or multiple features



Data imputation

Fill in the missing value with:

- a constant value or a randomly selected value
- the most frequent category for categorical data
- the mean, median or mode of a feature for numerical data
- the result of a predictive model
  - K-nearest neighbours
  - linear or polynomial interpolation

`sklearn.impute`.SimpleImputer

`sklearn.impute`.KNNImputer

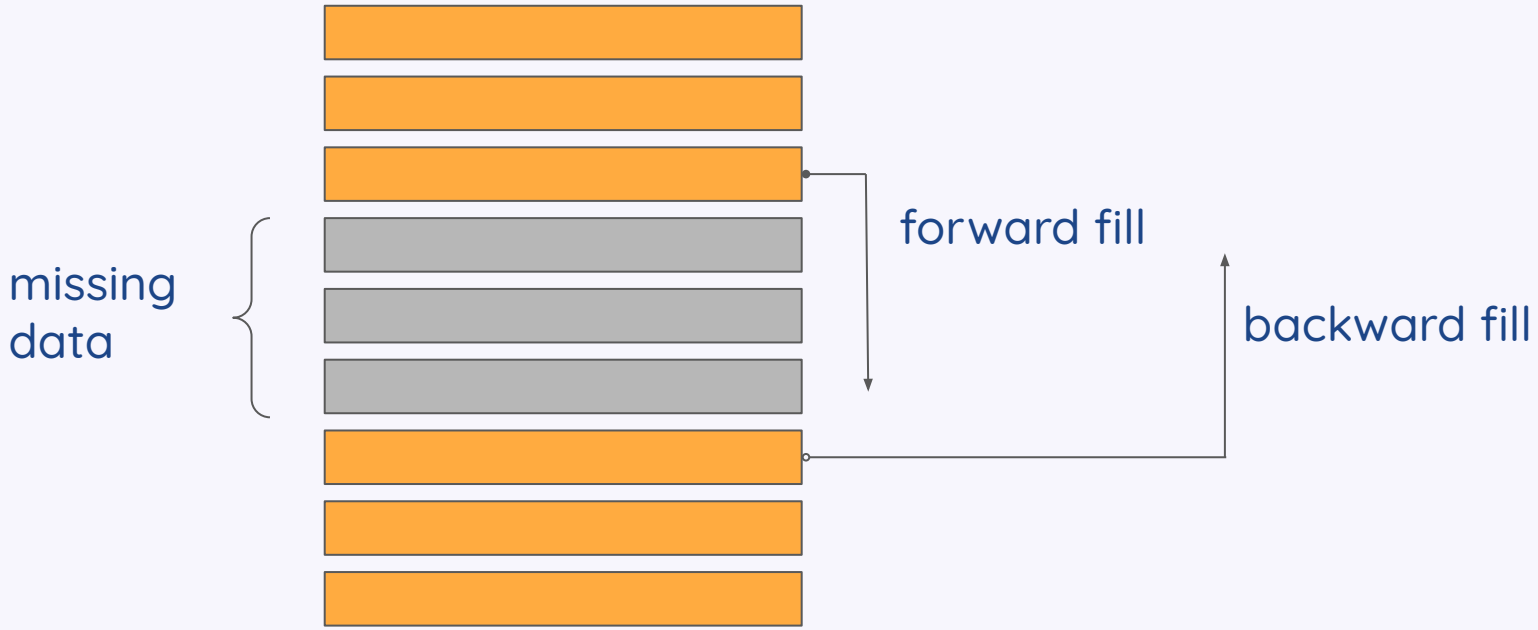
`sklearn.impute`.IterativeImputer

`pandas.DataFrame`.dropna

`pandas.DataFrame`.ffill

`pandas.DataFrame`.interpolate

Forward/backward fill



Forward/backward fill in Spark

Any challenges using forward or backward fill in **Spark**?

Forward/backward fill in Spark

Forward/backward fill propagates values **sequentially**

- each row needs to know what the last non-null value was before it

=> Order across the entire dataset

In Spark: data is split across multiple partitions on different machines.

Each machine processes its partition independently in parallel.

For very large datasets: partition data logically (for example by time).

Conditional imputation

Model one feature as a function of others

Possible implementation:

iteratively predict one feature as a function of others

`sklearn.impute`.IterativeImputer

Conditional imputation

Model one feature as a function of others

Possible implementation:

iteratively predict one feature as a function of others

`sklearn.impute`.IterativeImputer

bad computational scalability!

Imputation for prediction

Typical assumption in statistics - data is missing at random

=> missingness independent from unseen values

In applications - often not the case

=> informative missingness: add indicator

`sklearn.impute`.SimpleImputer(add_indicator=True)

Data preprocessing

Missing values

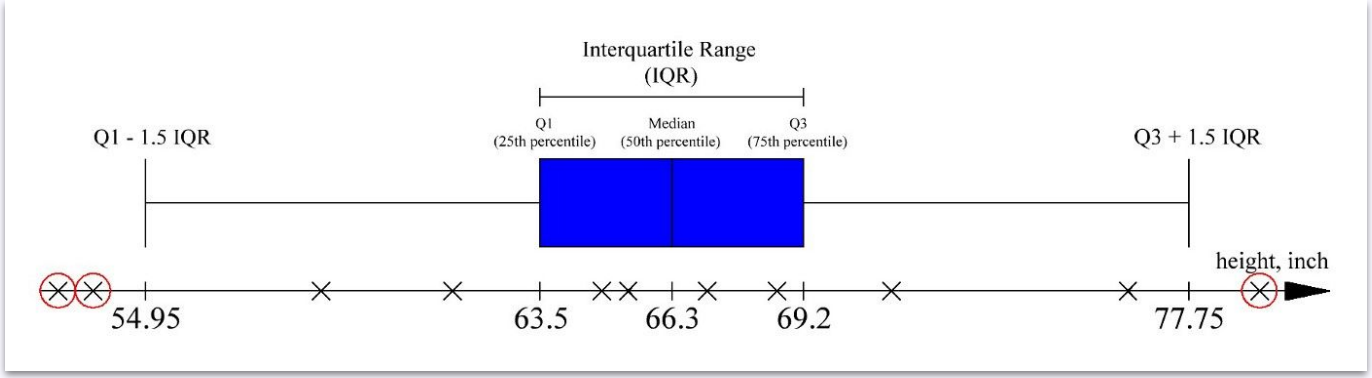
Outlier detection

Feature scaling

Feature encoding

Outlier detection (anomaly detection)

Detecting instances that deviate strongly from the norm
- for removal or for analysis.

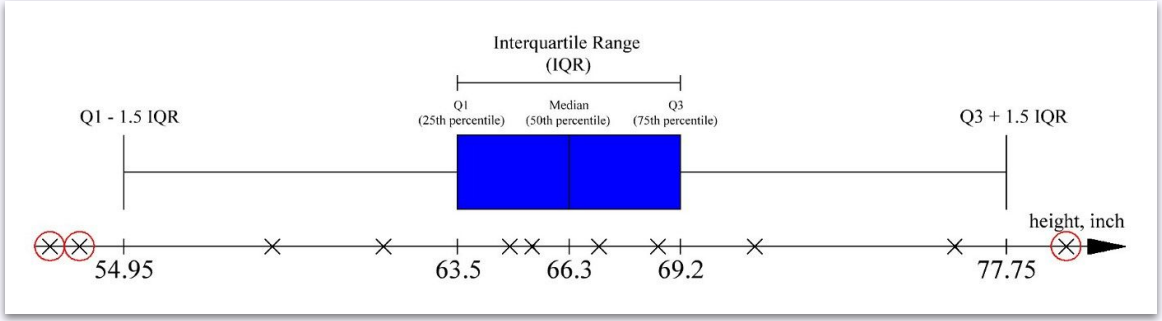


Tukey's method

$$[Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)]$$

most commonly k = 1.5

Outlier detection

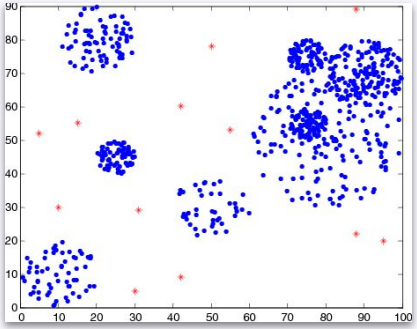


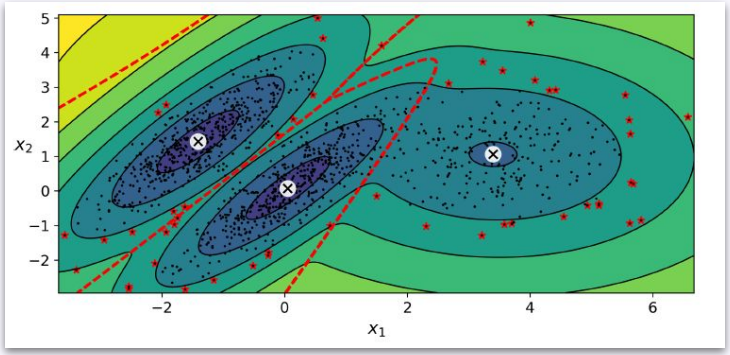
Tukey's method

$$[Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)]$$

most commonly k = 1.5

Gaussian mixture models





Data preprocessing

Missing values

Outlier detection

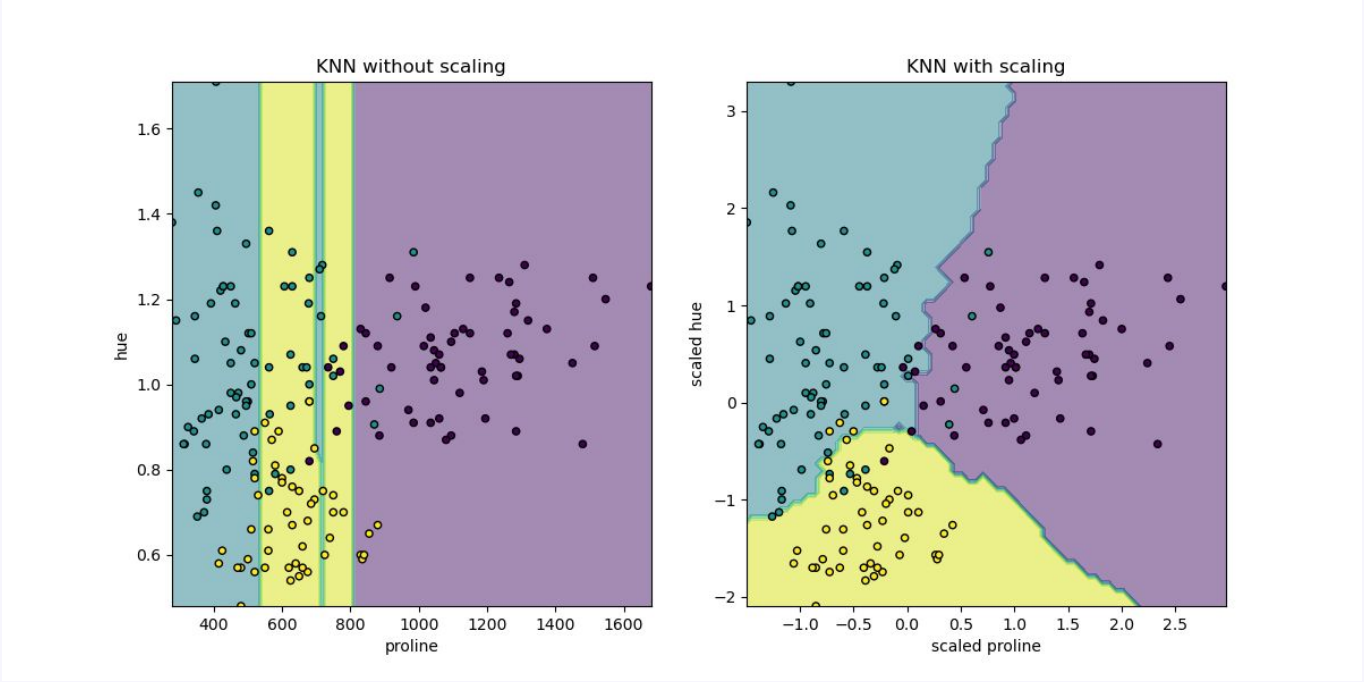
Feature scaling

Feature encoding

Feature scaling

Many ML models fail if the features have different scales.

(exception: some tree based models)



Feature scaling

Standardisation (Z-score Normalization)	Max-Min Normalization
$x_{\text{stand}} = \frac{x - \text{mean}(x)}{\text{standard deviation } (x)}$	$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$

- *Standardization* = subtracts the mean value and scales the data to unit variance
  - less (but still) affected by outliers
- *Normalization* = values are shifted and rescaled so that they end up ranging from 0 to 1

Naming: - standardization is sometimes called z-score normalization

- normalization is sometimes called min-max scaling.

Feature scaling - which to use?

- Tree-based models (Random Forest, XGBoost, Decision Trees): scaling often not necessary (typically scale-invariant)
- Linear models, SVM, k-NN: standardization preferred
- Neural networks: either, standardization often preferred; normalization for bounded outputs
- Outliers: standardization
- Need specific range: normalization

Data preprocessing

Missing values

Outlier detection

Feature scaling

Feature encoding

Feature encoding

Some models need data preprocessed to numerical values.

(Specifically non-tree based models, such as linear, svm, neural networks. For tree based models feature encodings may still be used to reduce memory requirements.)

In these cases all features need to be encoded numerically.

SAFETY-LEVEL (TEXT)	SAFETY-LEVEL (NUMERICAL)
None	0
Low	1
Medium	2
High	3
Very-High	4

Feature encoding

Feature encoding = transforming features to a numerical format

Label encoder

Frequency encoder

One hot encoder

Ordinal encoder

Reminder: here we are considering categorical data (ordinal or not). The number of categories (classes) is given by the data.

Label encoder

Alphabetical label encoder:

- encode with a value between 0 and n_categories-1

SAFETY-LEVEL (TEXT)	SAFETY-LEVEL (NUMERICAL)
None	0
Low	1
Medium	2
High	3
Very-High	4

Label encoder

Alphabetical label encoder:

- encode with a value between 0 and n_categories-1

Frequency label encoder:

- encode with a fraction = frequency of category
- `sklearn.preprocessing`.LabelEncoder
  - Use **only** for target variable (= the value you are trying to predict)

SAFETY-LEVEL (TEXT)	SAFETY-LEVEL (NUMERICAL)
None	0
Low	1
Medium	2
High	3
Very-High	4

Numerical value	Animal
1.5	cat
3.6	cat
42	dog
7.1	crocodile

Frequency encoding →

Numerical value	Animal_freq
1.5	0.5
3.6	0.5
42	0.25
7.1	0.25

Manual label encoder

You can manually encode categories, using for example a dictionary.

```
scale_mapper = {"Low":1, "Medium":2, "High":3}
df["Scale"] = df["Score"].replace(scale_mapper)
```

Use this carefully!

Label encodings do not make sense for every type of category and prediction model.

Example: transforming the categories “cat”, “mouse”, “dog” into 0, 1, 2 in a regression task would allow for operations such as 0.5*dog = mouse.

One hot encoder

id	color
1	red
2	blue
3	green
4	blue

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

- Encode features as a one-hot numeric array
- Creates a new binary column for each category
- The value for each column is either 0 or 1 => already scaled
- Not recommended for features with a large number of categories
  - as an alternative, can be applied together with dimensionality reduction

One hot encoder in *pandas* and *sklearn*

pandas.get_dummies

```
>>> s = pd.Series(list('abca'))
>>> pd.get_dummies(s)

   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```

sklearn.preprocessing.OneHotEncoder

```
>>> df = pd.DataFrame(data={'fruit': ['apple', 'apple', 'banana', 'orange', 'banana', 'apple'],
                            'size': ['large', 'medium', 'small','large', 'medium', 'small']})
>>> encoder = OneHotEncoder()
>>> one_hot_encoder = encoder.fit_transform(df[['fruit']])
>>> one_hot_encoder.get_feature_names())
['x0_apple', 'x0_banana', 'x0_orange']
```

	fruit	size
0	apple	large
1	apple	medium
2	banana	small
3	orange	large
4	banana	medium
5	apple	small

	x0_apple	x0_banana	x0_orange
0	1.0	0.0	0.0
1	1.0	0.0	0.0
2	0.0	1.0	0.0
3	0.0	0.0	1.0
4	0.0	1.0	0.0
5	1.0	0.0	0.0

Ordinal encoder

The features are converted to **ordinal integers.**

This results in a single column of integers

(0 to n_categories - 1) per feature.

Only use when order matters!

SAFETY-LEVEL (TEXT)	SAFETY-LEVEL (NUMERICAL)
None	0
Low	1
Medium	2
High	3
Very-High	4

Feature generation

- Datetime
  - periodicity - capture repetitive patterns in the data
  - *pandas* library has useful tools for handling datetime features

ID	Datetime	Count	year	month	day	dayofweek_num	dayofweek_name
0	2012-08-25 00:00:00	8	2012	8	25	5	Saturday
1	2012-08-25 01:00:00	2	2012	8	25	5	Saturday
2	2012-08-25 02:00:00	6	2012	8	25	5	Saturday
3	2012-08-25 03:00:00	2	2012	8	25	5	Saturday
4	2012-08-25 04:00:00	2	2012	8	25	5	Saturday

Datetime Properties

`Series.dt.date`
`Series.dt.time`
`Series.dt.year`
`Series.dt.month`
`Series.dt.day`
`Series.dt.hour`
`Series.dt.minute`
`Series.dt.second`
`Series.dt.microsecond`
`Series.dt.nanosecond`
`Series.dt.week`
`Series.dt.weekofyear`
`Series.dt.dayofweek`
`Series.dt.weekday`
`Series.dt.dayofyear`
`Series.dt.quarter`
`Series.dt.is_month_start`
`Series.dt.is_month_end`

Function transformers

You might need to implement a transformer from an arbitrary function

- for example if data is log-normally distributed - apply log transformation

```
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p, validate=True)
>>> X = np.array([[0, 1], [2, 3]])
>>> # Since FunctionTransformer is no-op during fit, we can call transform directly
>>> transformer.transform(X)
array([[0.        , 0.69314718],
       [1.09861229, 1.38629436]])
```

Sklearn terminology

- **Estimators** - any object that can estimate some parameters based on a dataset; example: nearest neighbours

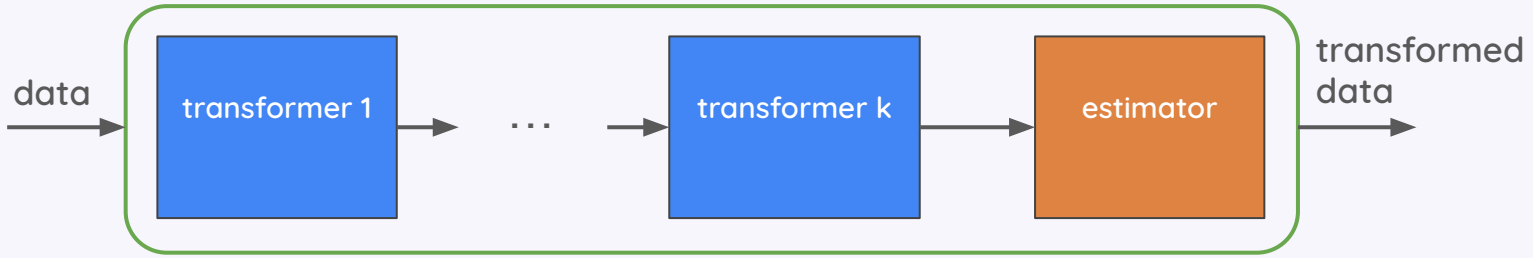
  `estimator = estimator.fit(data)`
- **Transformers** - some estimators can also transform a dataset; example: imputer; `fit()` to find the parameters, `transform()` to do the transformation; use `fit_transform()` for both
- **Predictors** - some estimators can also do predictions via `predict()`; use `score()` to evaluate predictions

Transformation pipelines

Transformation steps need to be executed in the right order.

Pipelines help with sequences of transformations.

sklearn pipelines: All but the last estimator must be transformers.



Transformation pipelines

Transformation steps need to be executed in the right order.

Pipelines help with sequences of transformations.

sklearn pipelines: All but the last estimator must be transformers.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Custom transformers

Some transformations should be done only on particular columns.

Use `ColumnTransformer()` mentioning:

`(name of transformation, transformation, column)`

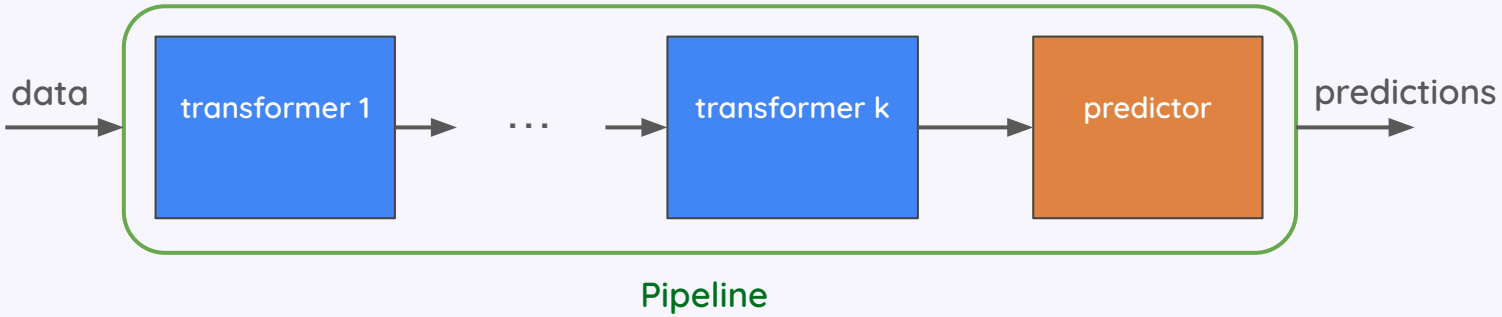
```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

Prediction pipelines




```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
...
...
pipeline = Pipeline(...)
pipeline.fit(X_train, y_train)
pipeline.predict(X_test)
```

The test set must use **identical** scaling to the training set. Pipelines do this automatically.

Purposes of using pipelines

Convenience and encapsulation

You only have to call fit and predict once on your data to fit a whole sequence of estimators.

Joint parameter selection

You can grid search over parameters of all estimators in the pipeline at once.

Safety

Pipelines help avoid leaking statistics from your test data into the trained model in cross-validation, by ensuring that the same samples are used to train the transformers and predictors.

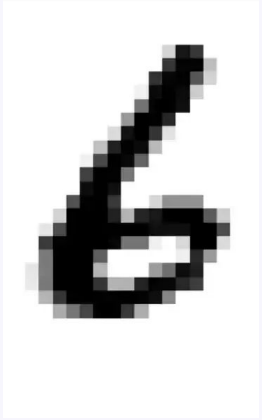
High-dimensional data

Many applications use data with hundreds or thousands of dimensions

=> many objects are very far away from each other

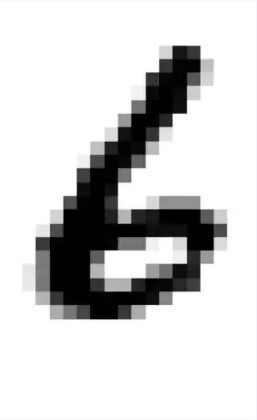
The curse of dimensionality

Often data has much lower intrinsic dimensionality than its representation.



Dimensionality reduction (DR)

The assumption is that data lies on or near a lower dimensional subspace than its representation.

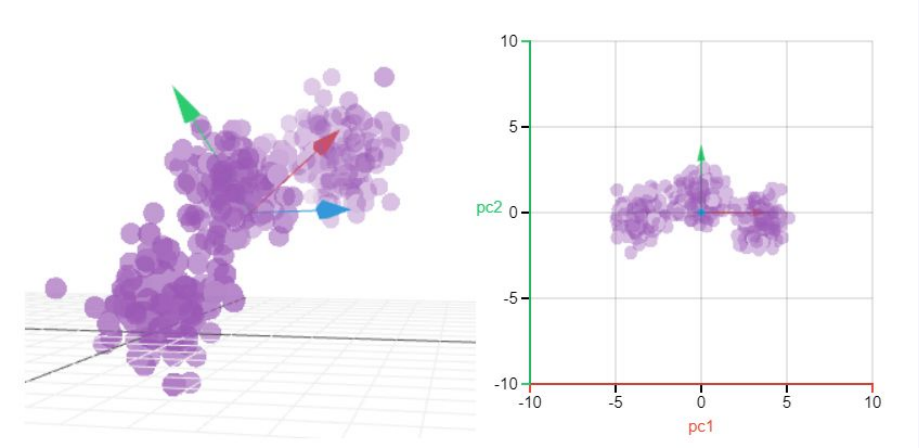


	Matrix	Alien	Star Wars	Casablanca	Titanic
Joe	1	1	1	0	0
Jim	3	3	3	0	0
John	4	4	4	0	0
Jack	5	5	5	0	0
Jill	0	0	0	4	4
Jenny	0	0	0	5	5
Jane	0	0	0	2	2

the data is actually 2 dimensional; rows can be reconstructed by scaling [1 1 1 0 0] and [0 0 0 1 1]

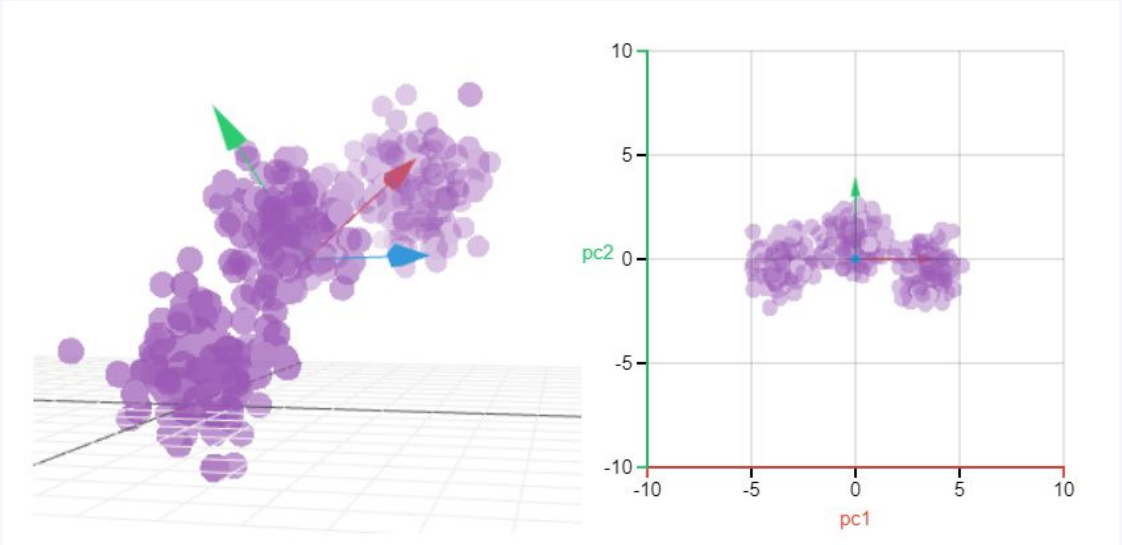
Why reduce dimensions?

- Remove redundant and noisy features
- Not all features are useful for a specific task
- Interpretation and visualization
- Easier storage and processing of the data



Dimensionality reduction (DR)

Goal: map a data vector from an original space to a new space with a lower dimension than the original, preserving the *structure* of the data



high-dimensional space ⟶ low-dimensional space

Preserving structure

Goal: map a data vector from an original space to a new space with a lower dimension than the original, preserving the *structure* of the data

How can we quantify how well structure is preserved?

Tension between preserving local and global structure

- local structure: the low-dimensional representation preserves each observation's neighborhood from the high-dimensional space
- global structure: the low-dimensional representation preserves the relative positions of neighborhoods from the high-dimensional space.

Dimensionality reduction (DR)

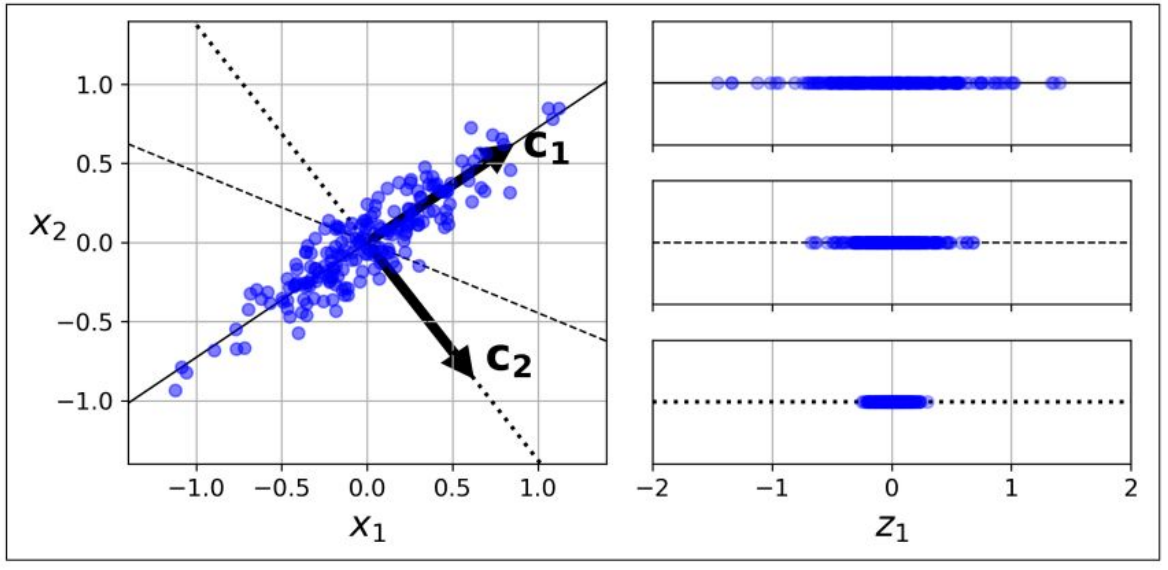
PCA: Principal component analysis

t-sne: t-Distributed Stochastic Neighbor Embedding

UMAP: Uniform Manifold Approximation and Projection

Principal components analysis (PCA)

Idea: find dimensions where data is most “spread out”



Dimensionality reduction with PCA

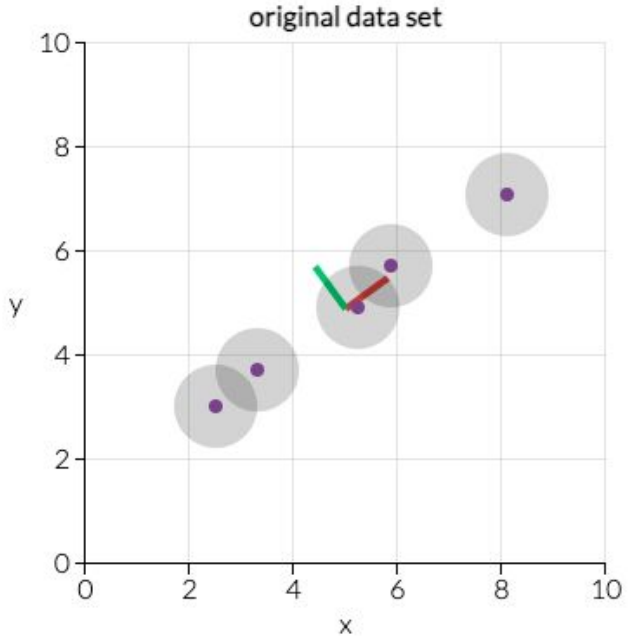
= Orthogonal linear transformation that transforms the data to a new coordinate system such that:

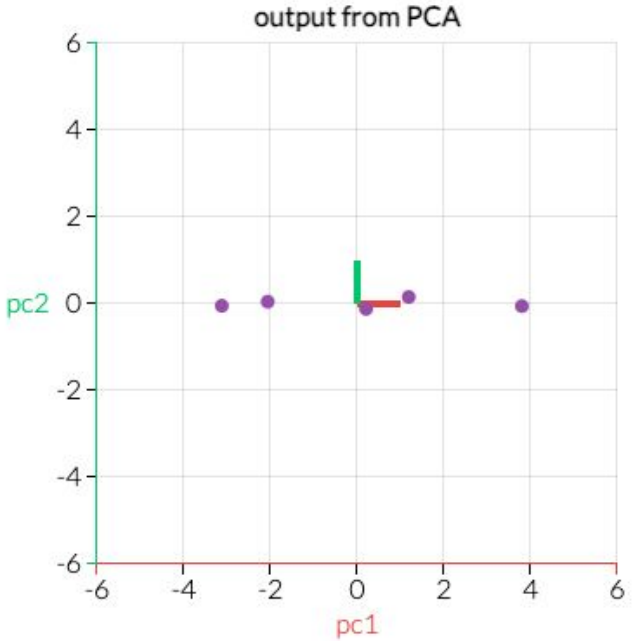
- the greatest variance by some scalar projection of the data comes to lie on the first coordinate (called the first principal component)
- the second greatest variance on the second coordinate
- ...

PCA has been shown to be effective in preserving the global structure of the data

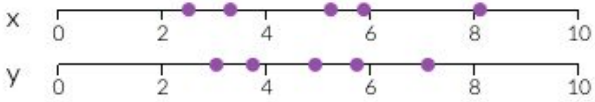
Principal components = sequence of unit vectors

The directions of the vectors are given by best-fitting lines

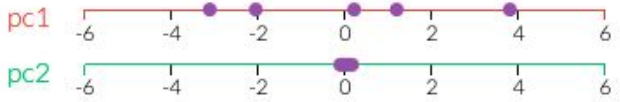




PCA is useful for eliminating dimensions. Below, we've plotted the data along a pair of lines: one composed of the x-values and another of the y-values.



If we're going to only see the data along one dimension, though, it might be better to make that dimension the principal component with most variation. We don't lose much by dropping PC2 since it contributes the least to the variation in the data set.



PCA steps

1. Compute covariance matrix of the features

2. Find the eigenvalues and eigenvectors

3. Project to new dimensions

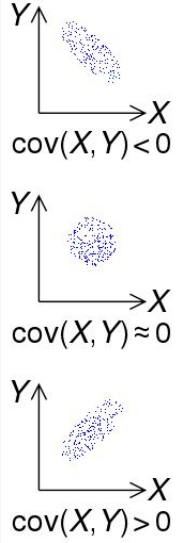
Covariance matrix

1. Compute covariance matrix of the features

- how spread out is the data in the given dimensions
- how one attribute changes based on another attribute

$$\text{cov}(x, y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

$$\begin{array}{c} \\ x \\ y \end{array}\begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{bmatrix} var(x) & cov(x,y) \\ cov(x,y) & var(y) \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ x \\ y \\ z \end{array}\begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} var(x) & cov(x,y) & cov(x,z) \\ cov(x,y) & var(y) & cov(y,z) \\ cov(x,z) & cov(y,z) & var(z) \end{bmatrix} \end{array}$$



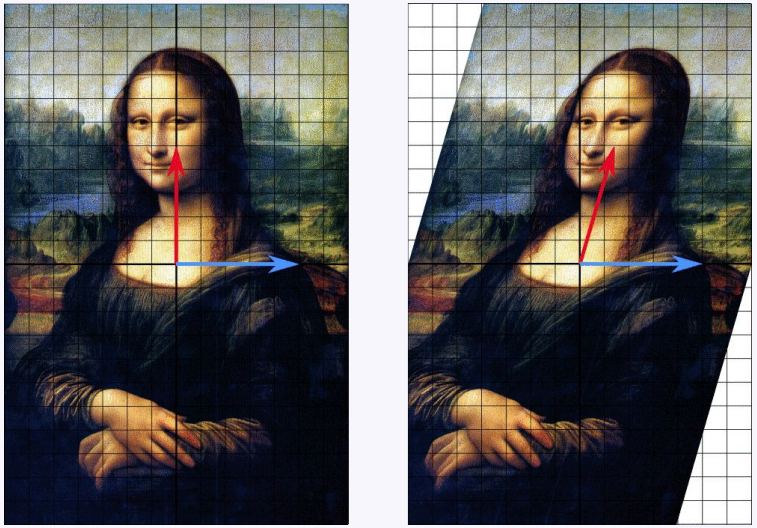
Eigenvalues and eigenvectors

2. Compute the eigenvectors of the covariance matrix

Eigenvector of a matrix = nonzero vector that changes at most by a constant factor when applied to that matrix

Eigenvalue = the constant factor above

$$A\mathbf{u} = \lambda\mathbf{u}$$



PCA steps

1. Compute covariance matrix of the features

$$\text{cov}(x,y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

$$\begin{array}{c} \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} var(x) & cov(x,y) & cov(x,z) \\ cov(x,y) & var(y) & cov(y,z) \\ cov(x,z) & cov(y,z) & var(z) \end{bmatrix} \end{array}$$

2. Find the eigenvalues and eigenvectors

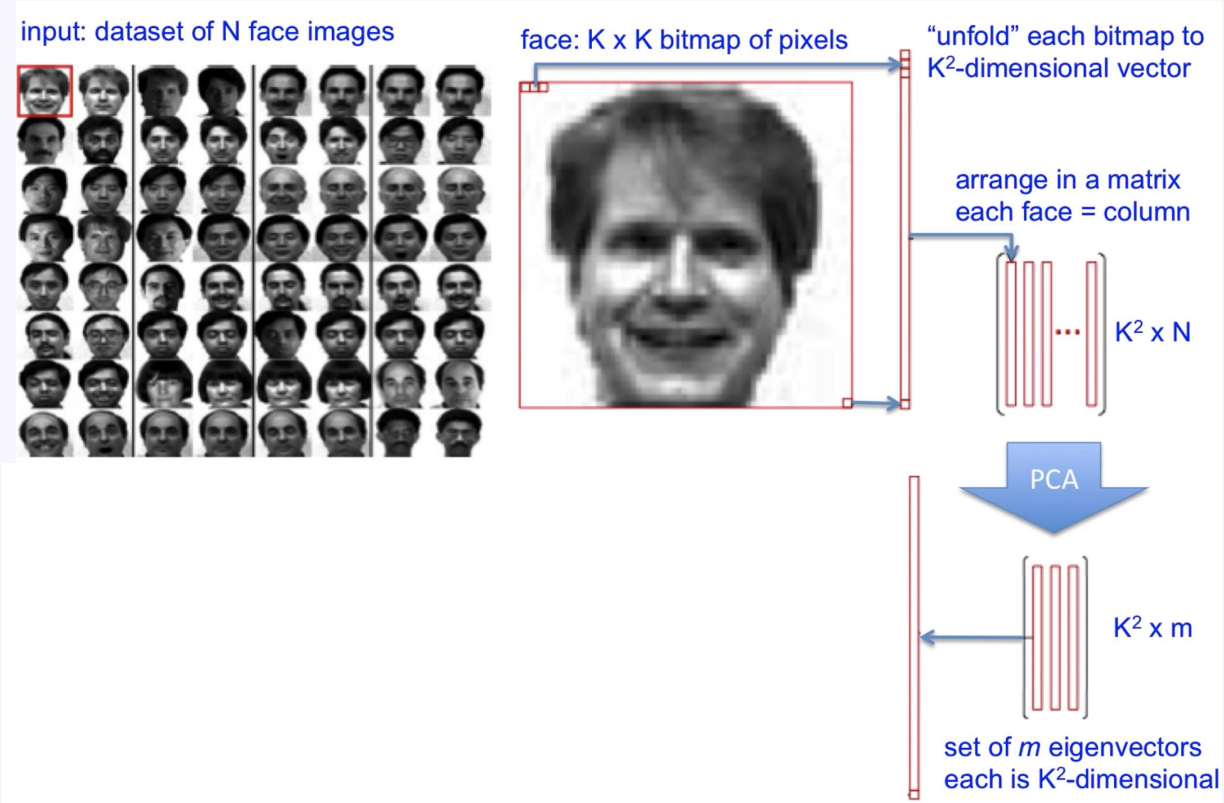
$$A\mathbf{u} = \lambda\mathbf{u}$$

3. Project to new dimensions

More details:

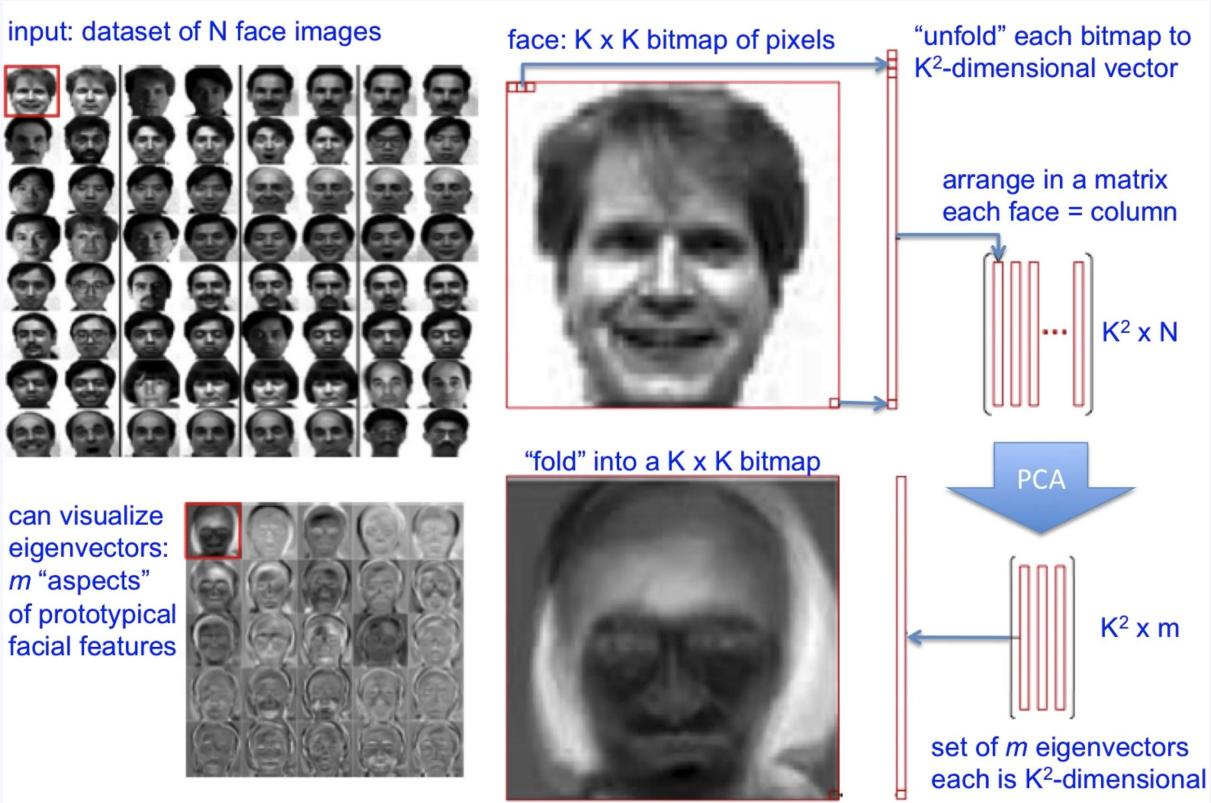
- statquest
- visualisations

PCA - Eigen faces



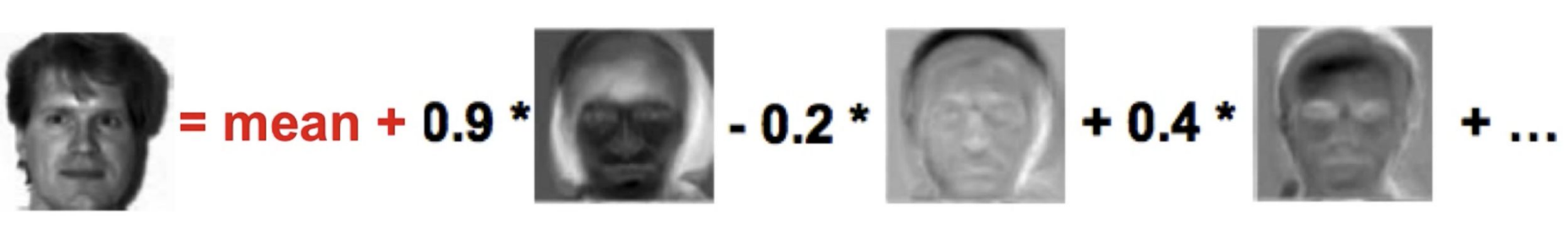
Source: Victor Lavrenko

PCA - Eigen faces



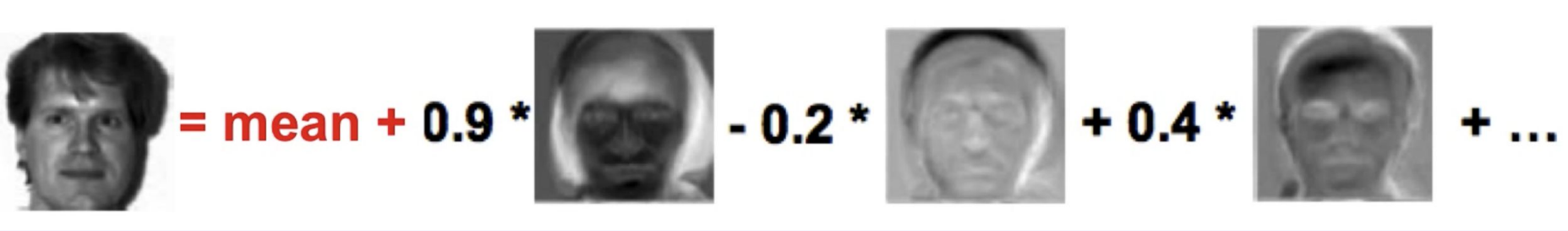
Source: Victor Lavrenko

Reconstructing the input



Source: Victor Lavrenko

Reconstructing the input



Source: Victor Lavrenko

PCA criticism

- Efficiency on large datasets
  - Incremental PCA
    - learning in a minibatch fashion
  - Randomized PCA (not discussed)
    - compute the first few principal components
- Interpretability
  - Sparse PCA
    - PCs are combinations of just a few original variables

Incremental PCA

PCA implementations generally support batch processing

=> the full dataset needs to fit into main memory

Incremental PCA gives an approximation of the PCA components using **partial computations**

Data is processed in a minibatch fashion

The explained variance ratio is updated with each batch

Interpretability

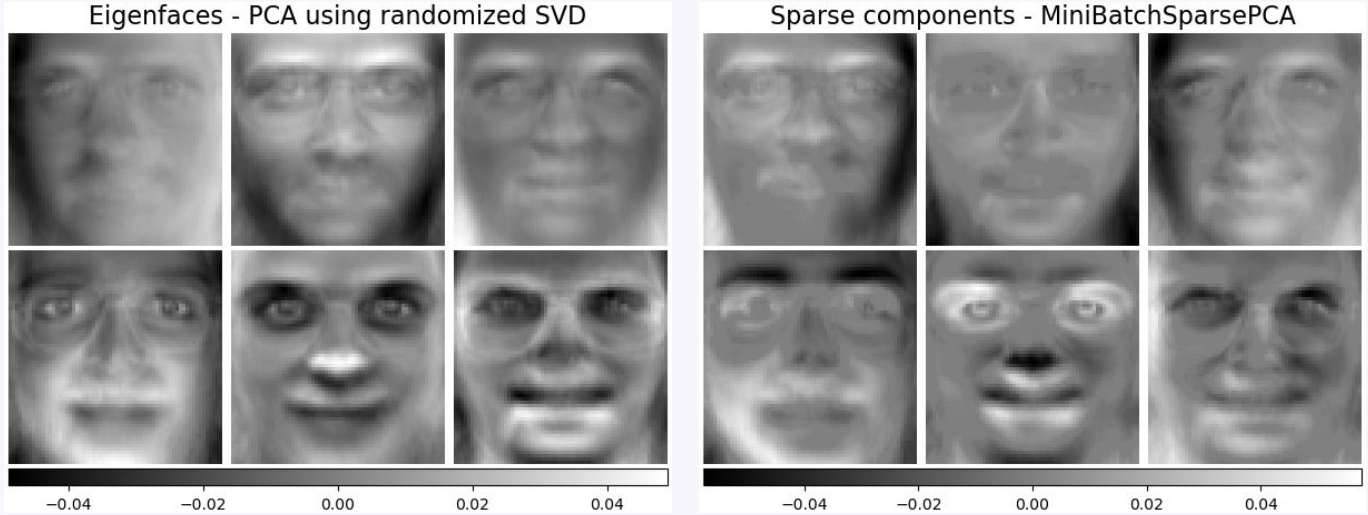
PCA: Low interpretability, since each eigenvector is a linear combination of all original features.

- In medical applications PCs generated during exploratory data analysis need to supply interpretable modes of variation
- In scientific applications such as protein folding, each original coordinate axis has a physical interpretation, and the reduced set of coordinate axes should too.

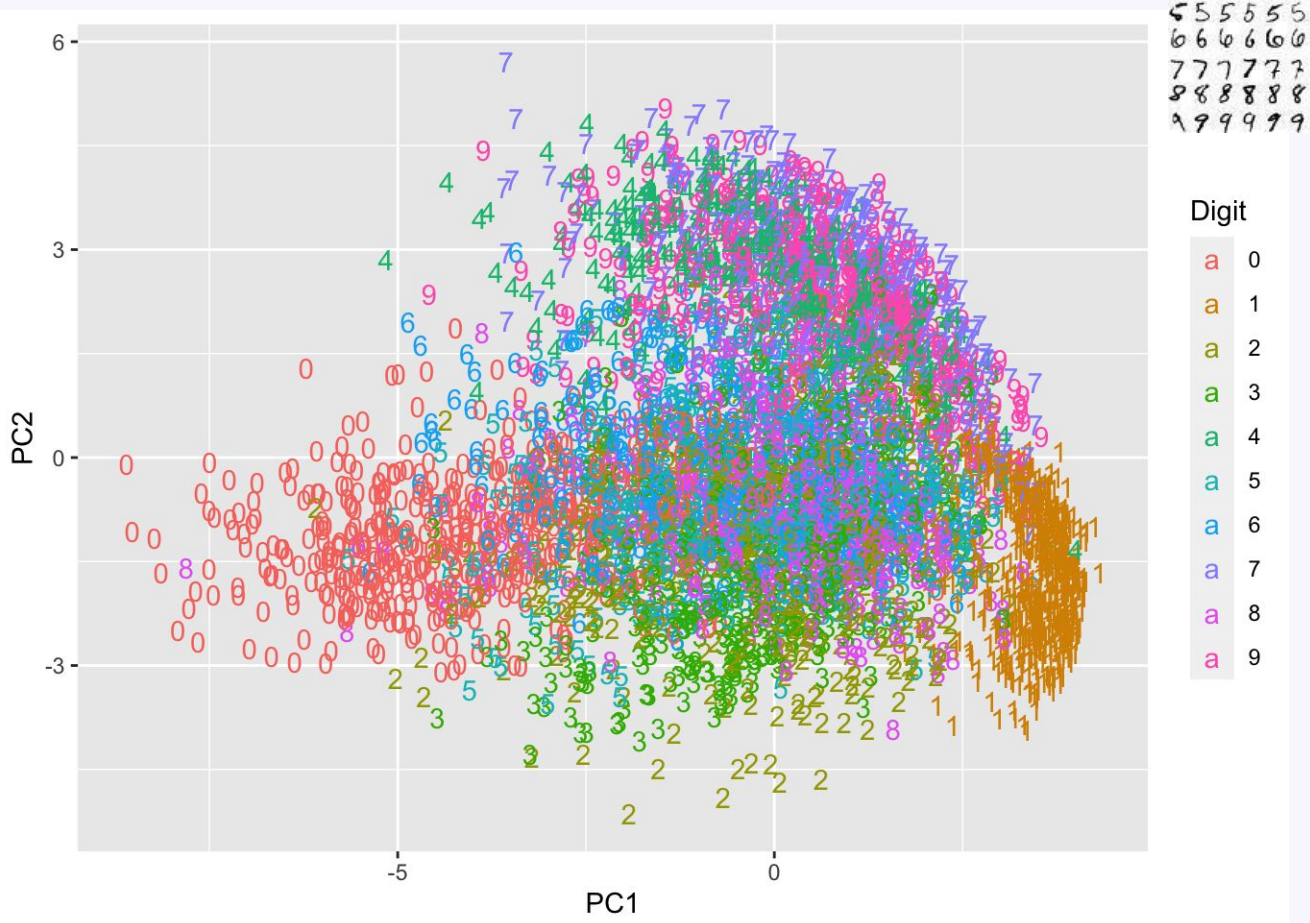
Sparse PCA

In PCA: the principal components are usually linear combinations of all input variables

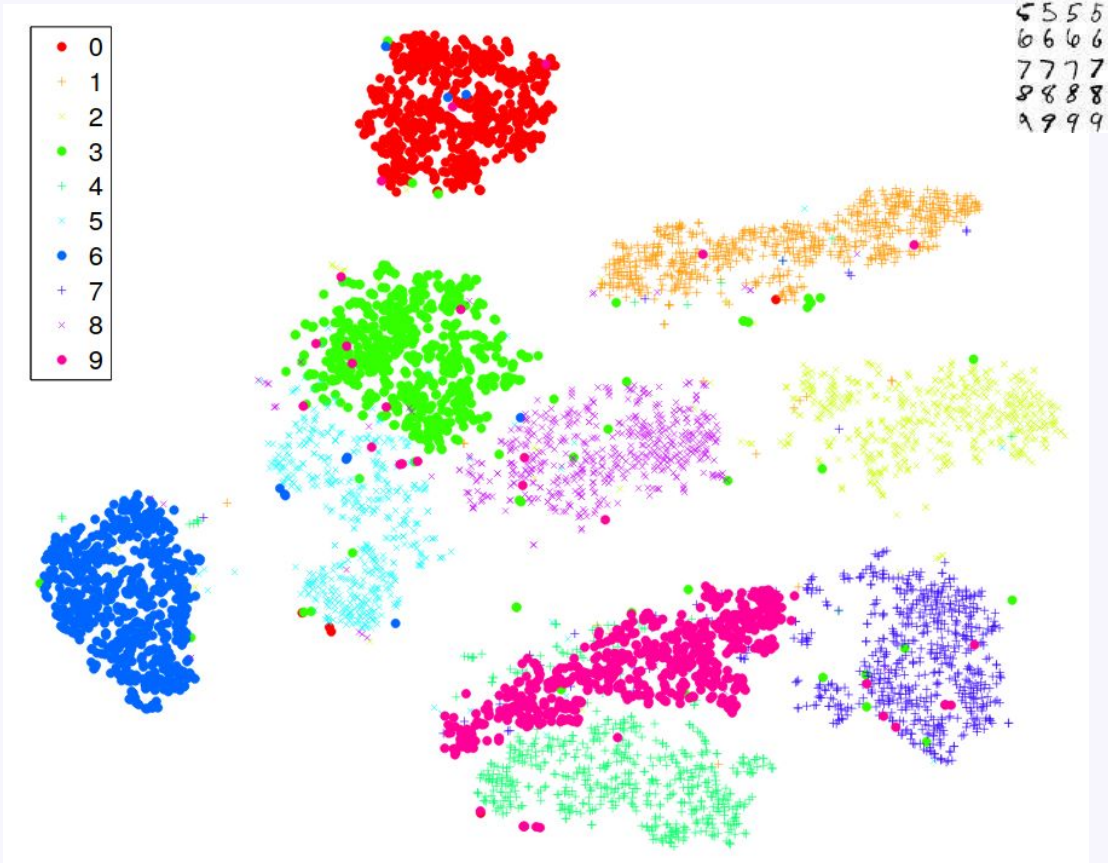
Sparse PCA: finds linear combinations of just a few input variables.



PCA on MNIST dataset



t-SNE on MNIST dataset



t-Distributed Stochastic Neighbor Embedding (t-SNE)

Paper

PCA - global structure preserving method

Cannot capture local non-linear structure

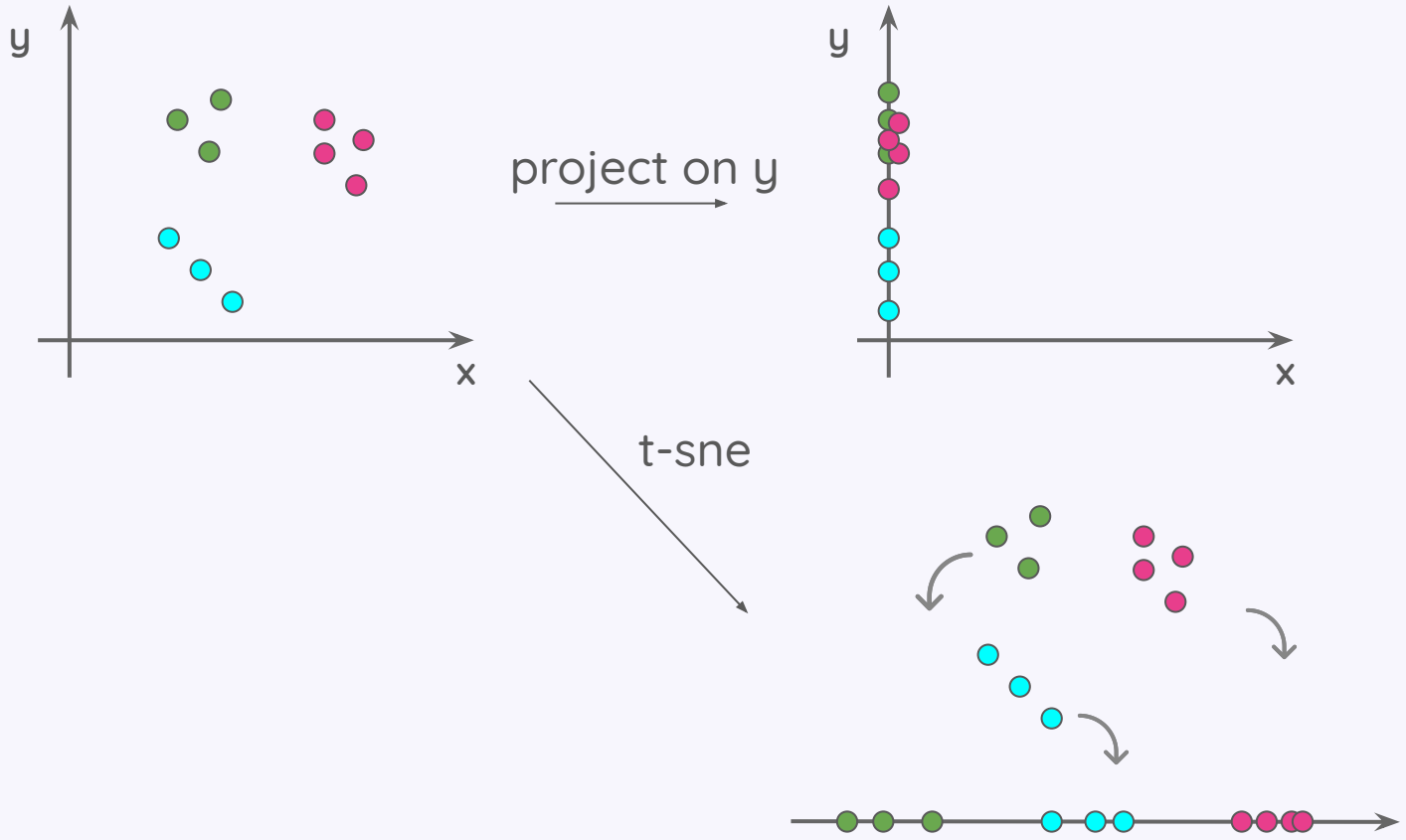
=> methods developed to preserve raw local distances between points

Difficult because in high dimensions many raw distances are similar

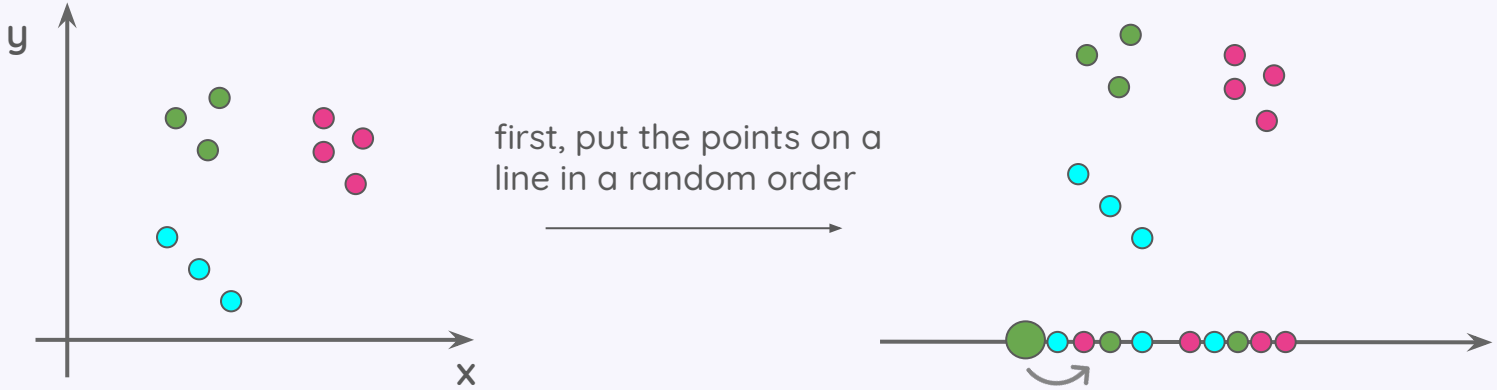
=> preserve neighbourhoods

t-SNE is a **non-linear technique** that reduces the data to 2 or 3 dimensions such that similar objects are modeled by nearby points

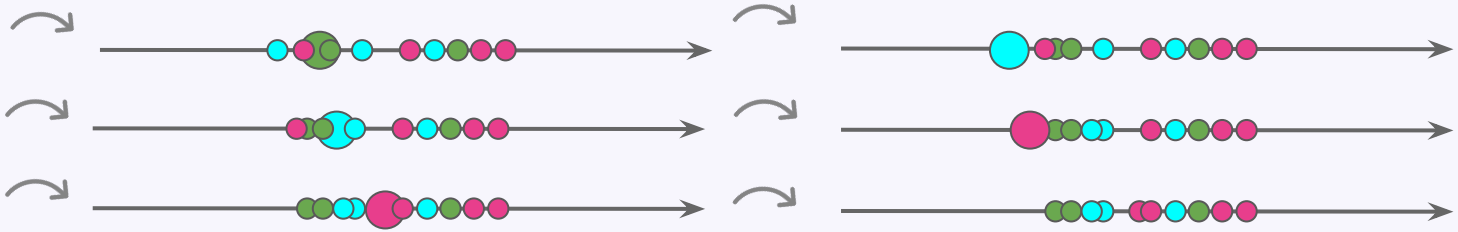
t-SNE



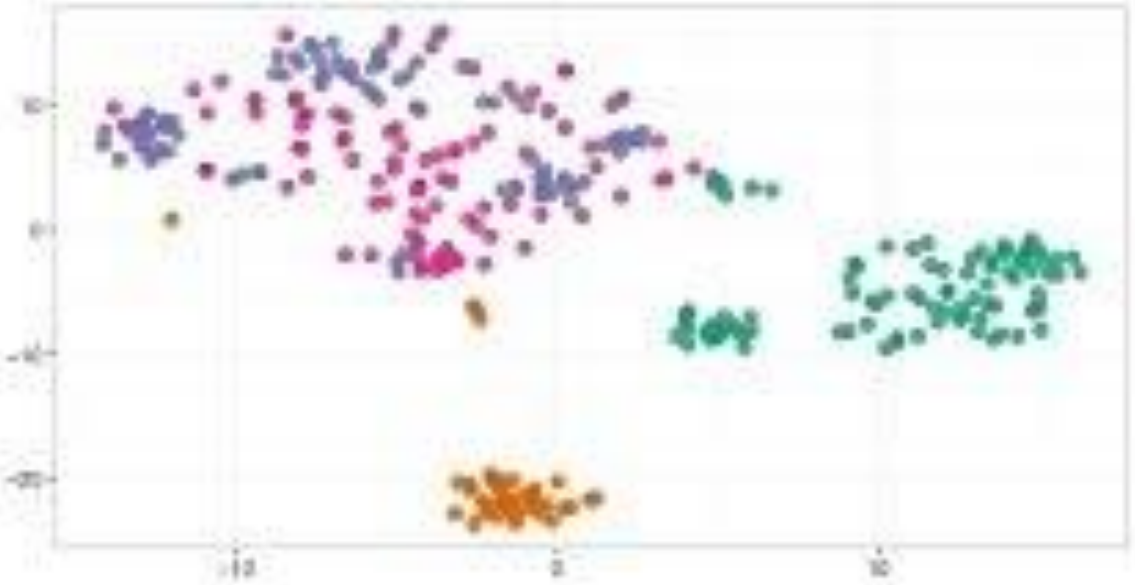
t-SNE



At each step, each point is moved on the line such that it is closer to its neighbours and far from dissimilar points.



t-SNE...



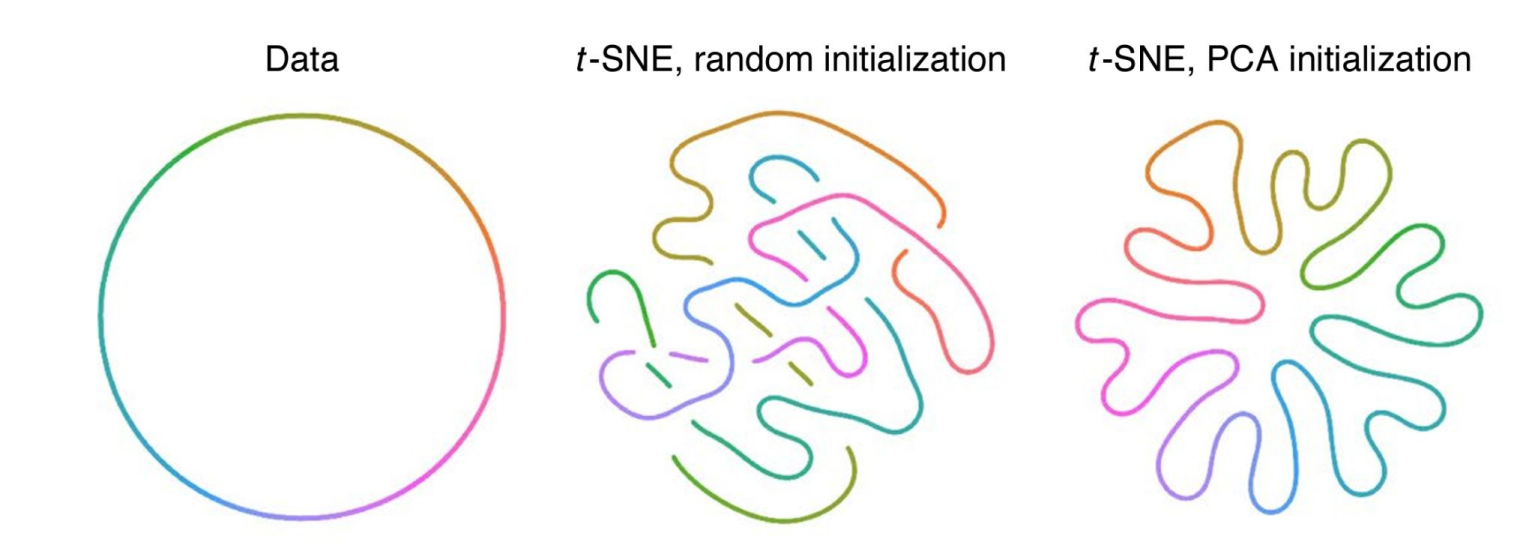
Clearly Explained!!!

Structure preservation

t-SNE preserved local structure (neighbours)

But struggles to preserve global structure ->

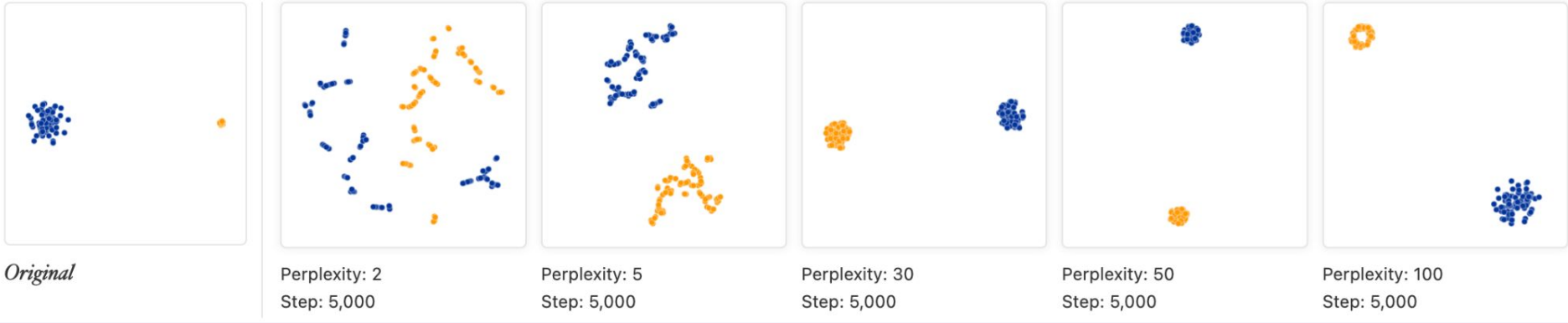
instead of random initialization, use PCA



How to Use t-SNE Effectively

Although extremely useful for visualizing high-dimensional data, t-SNE plots can sometimes be mysterious or misleading. By exploring how it behaves in simple cases, we can learn to use it more effectively.

Link to article



Original

Perplexity: 2
Step: 5,000

Perplexity: 5
Step: 5,000

Perplexity: 30
Step: 5,000

Perplexity: 50
Step: 5,000

Perplexity: 100
Step: 5,000

Uniform Manifold Approximation and Projection (UMAP)

Paper

t-SNE is known to sometimes produce spurious patterns

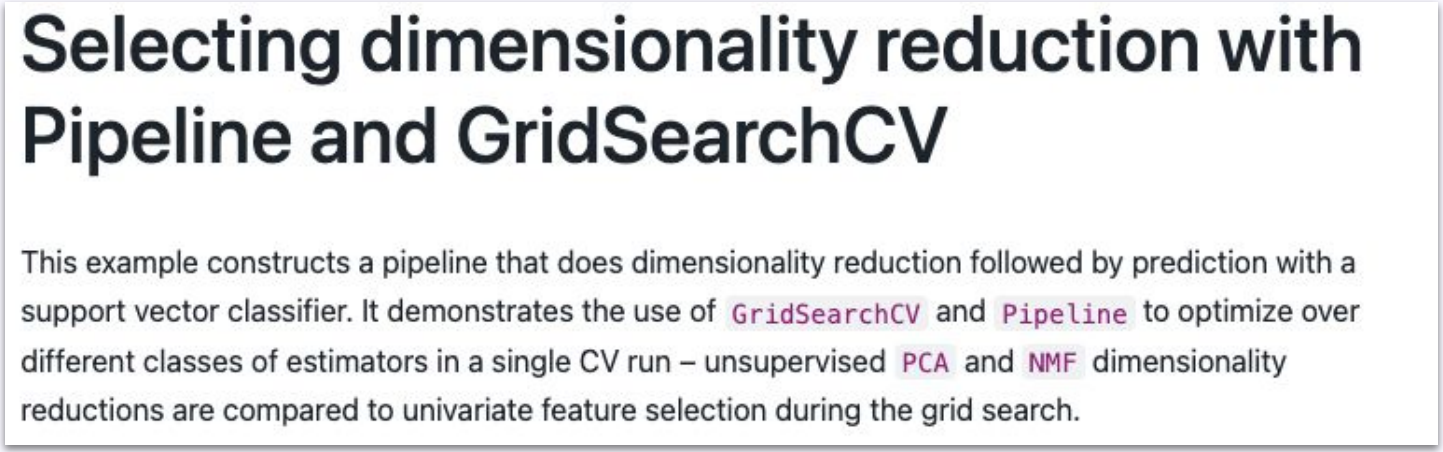
UMAP builds on t-SNE, improving:

- efficiency (running time)
- k-nearest neighbours accuracy
- preservation of global structure

Differences:

- UMAP uses *Spectral Embeddings* to initialize the low-dimensional values - deterministic
- t-SNE moves every point at each iteration; UMAP moves a small subset of points => scales better for large datasets

Example



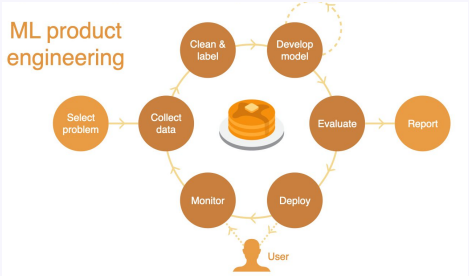
Selecting dimensionality reduction with Pipeline and GridSearchCV

This example constructs a pipeline that does dimensionality reduction followed by prediction with a support vector classifier. It demonstrates the use of `GridSearchCV` and `Pipeline` to optimize over different classes of estimators in a single CV run – unsupervised `PCA` and `NMF` dimensionality reductions are compared to univariate feature selection during the grid search.

An example of dimensionality reduction within a pipeline with sklearn

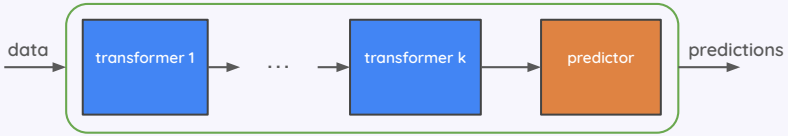
Summary

Building machine learning models = iterative process



Data exploration and cleaning

Pipelines - ensure transformations are done in order, consistently, without data leakage



Preprocessing of high-dimensional data