Applied Data Analysis

01 - Handling data

Data analysis: **inspecting, cleaning, transforming and modeling data** to discover useful information. Start with **raw data** coming from **data sources**, **model it for the usecase** and store it in **data warehouses**, after that standardize and use your data (**data wrangling**)

Scientific method 2.0: **data-driven science**, asking a question, getting data coherent to the question, exploring it to **get some results and form an hypothesis** (differs from the classical scientific method because it's formed during the process)

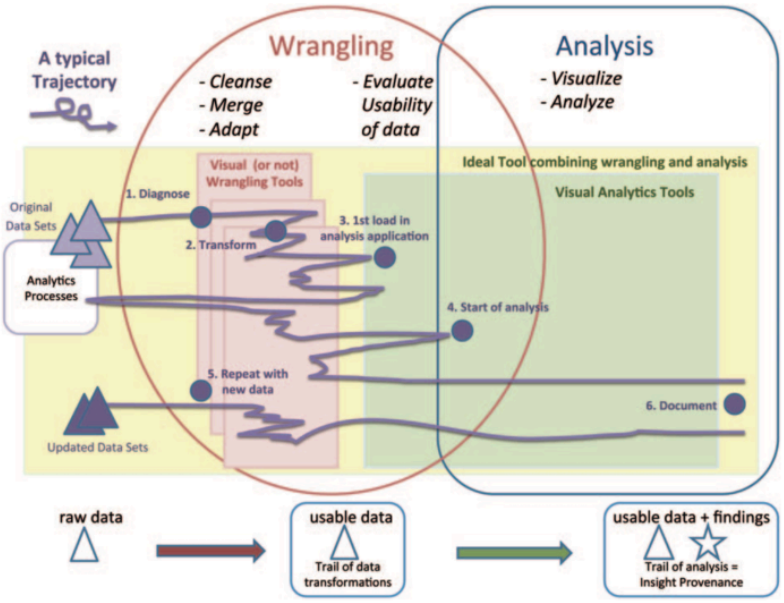
Data model: specifies how you think about the world, different types:

- **Flat model**: one type of entity, everything has the same attributes (**text format** like CSV, logs). To use it you either parse the text (can be expensive) or you store it in **binary format** instead ("as is"), quicker and lighter (especially for **big data**)
- **Relational model**: many types of entities, connected by **relationships**, **SQL** for **core data manipulations** (**declarative** (you describe what you want not how to do it (≠ OOP languages)). **Pandas** with **DataFrames** (equivalent to SQL table), fast, lightweight, SQL-like functions, easy to plot data but tables must fit into memory (slow/impossible for big data)
- **Document model**: **hierarchy** of entities, **JSON/XML**, the document structure is a **tree** so it's processed via tree traversal (BFS/DFS) or jq
- **Network model**: complex **network of entities**

HTML Parsing: tricky as there is **semantically irrelevant information** in between the structured information we want. To not overload websites and be efficient most websites offer an **API to get the content** (most common framework is **REST** (request an URL, get a text response)). Alternatively **download web crawling datasets** (1B+ already crawled and parsed web pages)

Wikidata: from **Wikipedia**, easy way to **get a lot of data through API or database dumps**

Data wrangling: **extract and standardize raw data**, combine multiple data sources, clean **data anomalies**. Problems usually are missing, incorrect or misrepresented data, 75% of problems require human intervention. Combine **automation with visualizations** to help in cleaning and finding problems (50%-80% of the process), having access to **docs about the dataset** and **nicely parseable data formats** also helps. **Over-sanitizing data** comes with **tradeoffs**



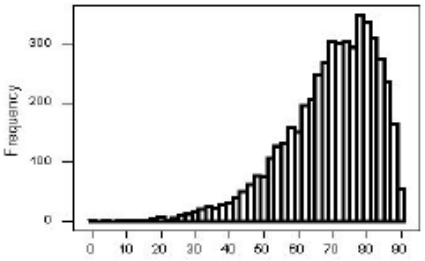
02 - Visualizing data

Uses for data visualization:

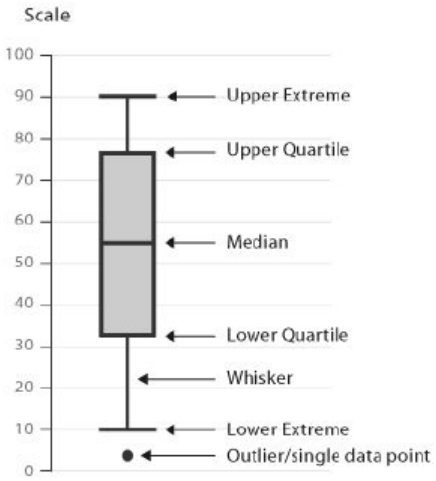
- **Analysis**, support reasoning about information (e.g. finding relationships, influences and structure)
- **Communication**, inform and persuade others
- **Decision making**, easier evaluation of potential courses of action

*Data visualization can also be **interactive**, especially for delivering the results. For data exploration **static visualization** is still used*

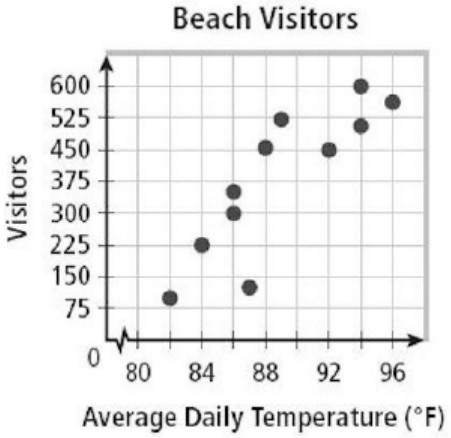
Histograms: **single variable**, easy to recognize **skewed distributions**. **2+ distinct peaks** often **suggest 2+ distinct populations** of samples, explore using colors



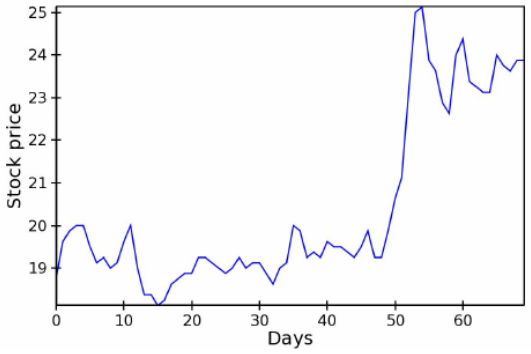
Box plots: **single variable**, good for **comparing different groups**, **evaluate distributions** and **outliers**



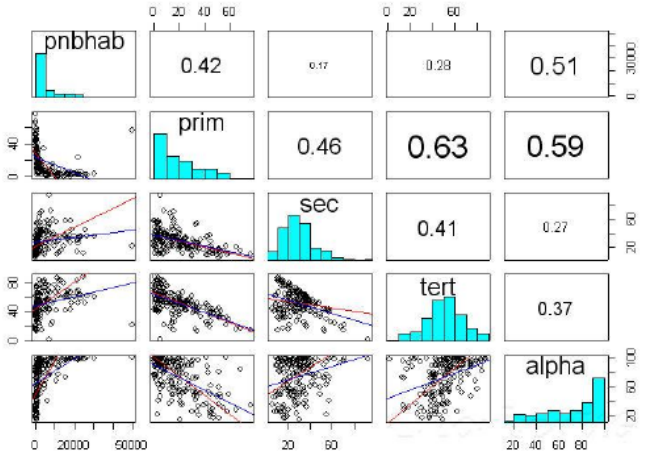
Scatter plots: **two variables**, expose **relationships between two variables**



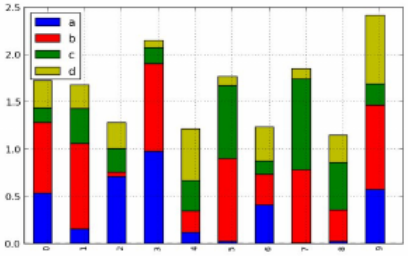
Line plots: **two variables**, used **if relationship is functional** (e.g. after binning and aggregating)



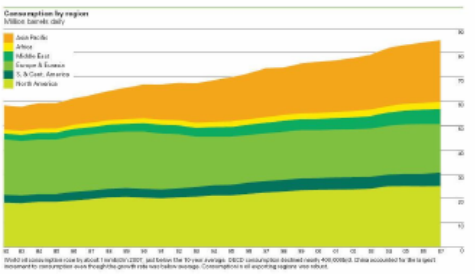
> 2 variables: **scatter plot matrix, stacked plots, parallel-coordinates, radar charts** (using colors)

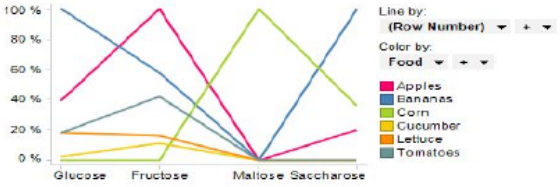


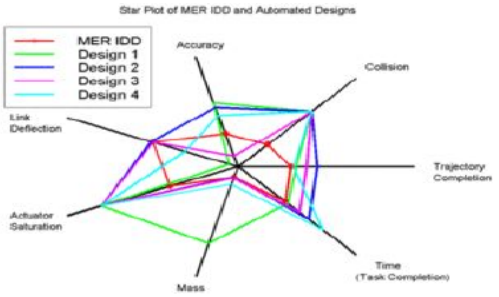
Stack variable and color variables categorical, height variable continuous:



Color variable categorical, stack and height variables continuous:



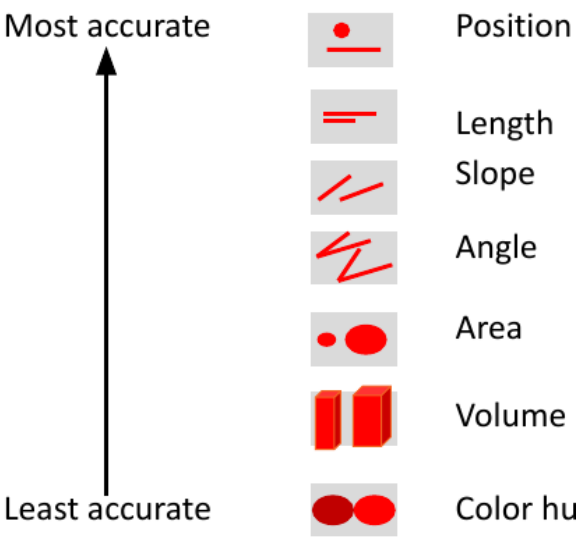




Dimensionality reduction: allow **visualization of high-dimensional continuous data** using **PCA principal components** (**dimensions** of the dataset with the **highest variation** and orthogonal)

Weird data can be just weird**, when you encounter it first **assume it's a bug** and try to fix it, if not possible then you made an **interesting discovery

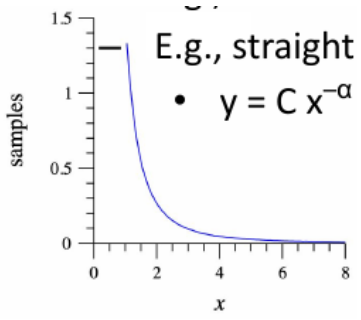
Just noticeable difference (JND): $\frac{\Delta I}{I} = k$, I intensity, ΔI increase from I so that you notice a difference, k constant. Most continuous variations in stimuli are perceived in discrete steps. It's important to **use visual contrast but don't to abuse it**, pick only certain aspects

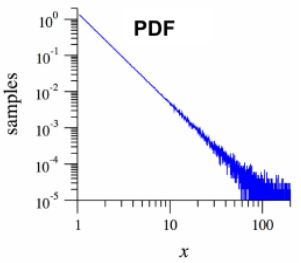


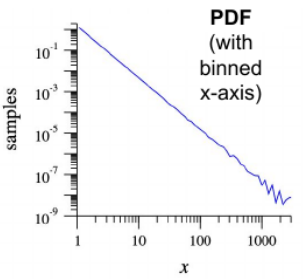
Logarithmic axis: useful for **heavy-tailed distributions**

Probability Density Function (PDF): function that describes **how probability is distributed across all possible continuous values of a variable**, the probability of a certain specific number is zero (as there are infinite numbers in a continuous interval), so you compute probabilities of the variable being in a range: $P(a \leq X \leq b) = \int_a^b f(x)dx$ where $f(x)$ is the PDF

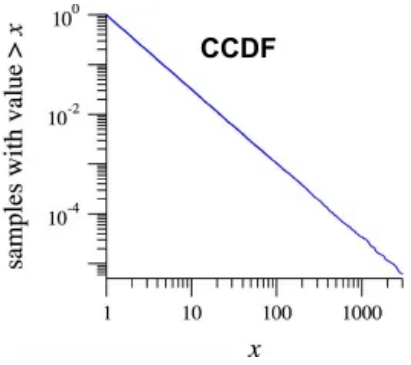
Power laws: $p(x) = Cx^{-\alpha}$, **decreasing**, **very large values are rare, but not very rare** (e.g. body size vs city size), some tricks are needed (e.g. for $\alpha <= 3$, var $\to \infty$ and for $\alpha <= 2$, also mean $\to \infty$, using **log on both axes makes the plot a line**)







Cumulative Density Function (CDF): PDF with no lower bound $\to P(X \leq a) = \int_{-\infty}^{a} f(x)dx$, **complementary** variant (**CCDF**) is $P(X > a) = 1 - P(X \leq a)$, **monotonically decreasing**, the **CCDF of a power law distribution is also a power law** with $C' = \frac{C}{\alpha - 1}$ and $\alpha' = \alpha - 1$. The **CCDF** can be easier displayed with the y-axis ticks like $(n \dots i)/n$



Interactive plots libraries: **D3.js** (**most used**, low-level), **Vega** (built on top of D3.js, uses JSON), Vincent (Python-to-Vega translator) and **Bokeh** (**big data** visualization for Python and Scala)

03 - Describing Data

Micro-average: **average of all elements**

Macro-average: **average of the averages of groups**

Robust statistic: **not sensitive to extreme values** (e.g. **median and quartiles**, NOT min, max, mean and std), necessary for **heavy-tailed distributions** where it's all about extreme values

Generalized means: **transform** data into a **difference space** (via function f), **take mean there**, then **transform back** into the original space (via f^{-1}) (e.g. **root mean square**, **geometric** (log and exp) and **harmonic** (1/x))

Distributions: before applying a model the **distribution of the data should be determined** (i.e. how it was generated) using **statistical tests** (e.g. goodness-of-fit, kolmogorov-smirnoff, normality), some important ones are:

- **Normal (Gaussian)**: continuous, **models many natural phenomena**
- **Poisson**: **counts of events happening** at a constant rate **in a fixed interval** (e.g., number of arrivals at a service desk per hour)
- **Exponential**: continuous, models **waiting time between independent Poisson events** (e.g., time between website visits)
- **Binomial / Multinomial**: discrete, number of **successes in n independent trials with fixed probability** *p*, multinomial if more than two outcomes

- **Power-law / Zipf / Pareto / Yule**: **heavy-tailed distributions**, common in **natural and social systems** (e.g., city sizes)

*Finite samples introduce **uncertainty** that **should always be reported** with error bars*

P-value: **probability of observing data as extreme** (or more) **as the data you already observed** assuming a (null) hypothesis is true (e.g. 16 heads out of 20 coin flips, what is the probability of having this outcome? if it's too low you might think the coin is not fair), tells you **how surprising the data you observed is** assuming the null hypothesis is true

Hypothesis testing: **quantify uncertainty** by making an hypothesis and then **computing p-values on observed data** assuming the hypothesis is true (the **higher it is the more certain we can be**). Used to **gain (weak and indirect) support for a hypothesis H_A by ruling out a null hypothesis H_0**, for example have H_0 = fair coin, H_A = unfair coin, S = test statistic = 50 - # heads after 100 coin flips = how much is deviating from the expected 50 heads and **p-value** =$P(S \geq s|H_0)$ = **probability of having s heads more than expected** after 100 flips assuming the coin is fair. The **null hypothesis H_0 can be rejected if the p-value is below a significance level α** which control the **false-rejection rate** (5/1/0.5/0.1%)

Bayes factors: **alternative to p-value**, $\frac{P(Data|H_0)}{P(Data|H_A)}$

Choosing a test statistic: depends on the **question to answer**, **data type**, **sample size** and if the samples being compared are in the **same population or not**

Confidence intervals: used to **quantify uncertainty**, calculate **intervals of a parameter of interest** μ (e.g. mean of a feature) **so that $\gamma\%$ of times the study is repeated, the parameter of interest is in the interval**, for the values in the confidence intervals the null hypothesis cannot be rejected if $\alpha \geq 1 - \gamma$. Intervals are computed with **parametric** (assume **test statistic follows a known distribution**) or **non-parametric methods** (bootstrap sampling of empirical data). Shown with **error bars**, min value of a confidence interval is at index $n * (0.5 - (\gamma/2))$, max is $n * (0.5 + (\gamma/2))$

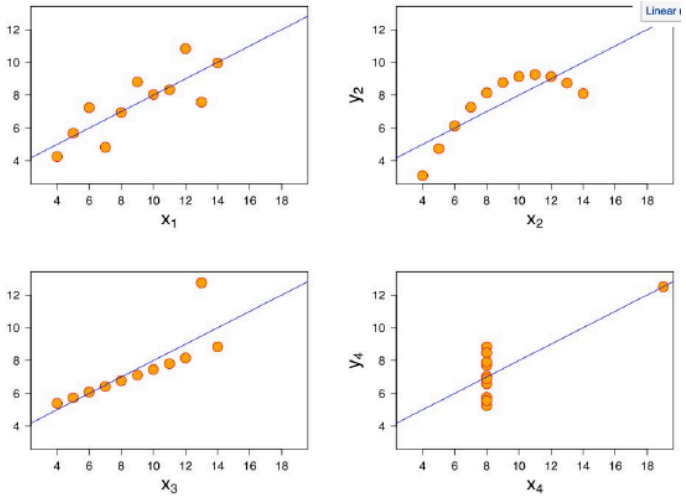
Empirical data: experimental data (obtained through set up experiments) + observational data (obtained through polls on past experience, no set up)

Bootstrap sampling: **random sampling with replacement** used when the sample pool is limited, works because normally when creating/obtaining new samples you get duplicate samples

*If you do **many experiments** you are **bound to find something**, the **significance level should be adjusted down** for that (**Sidak Correction**: $1 - (1 - \alpha)^{\frac{1}{k}}$ instead of α where $(1 - \alpha)^k$ is the probability of null hypothesis being respected in k experiments, **Bonferroni Correction**: $\frac{\alpha}{k}$)*

Correlation coefficient: **Pearson's** for amount of **linear depedence**, **Spearman's for rank correlation**

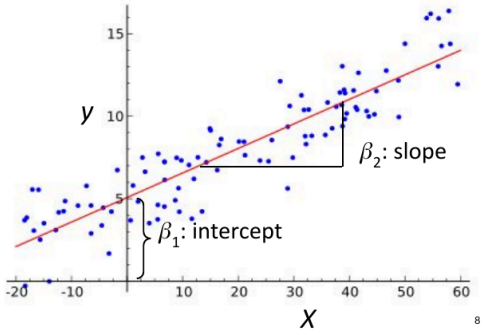
Anscombe's quartet: **same mean and variance**, **looking at data graphically is important**, **statistics is not enough**



Simpson's paradox: a **trend that appears** separately **in groups of data disappears when the groups are combined**, **looking at data in separated groups is helpful!**

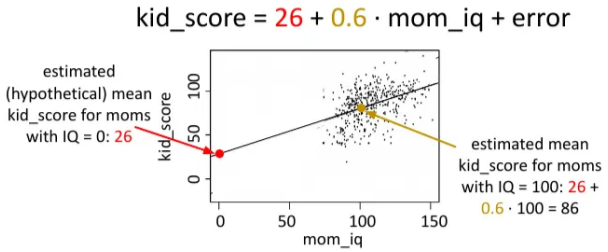
04 - Regression analysis for disentangling data

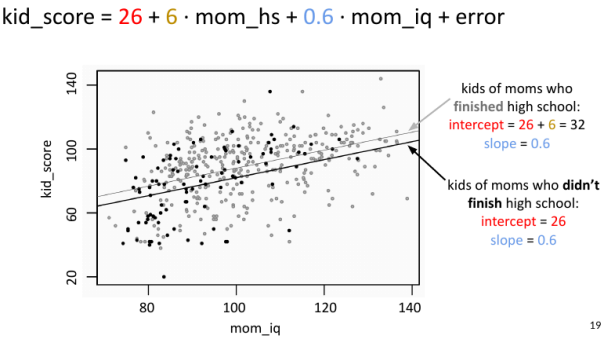
Linear regression: given n **data points** (X_i, y_i) where X_i is a k dimensional vector of predictors/features of the ith data point and y_i is its scalar outcome. The goal is to **find the optimal vector** β **of coefficients for approximating** y_i **as a linear function of** X_i: $y_i = X_i\beta + \epsilon_i$ where ϵ_i is the error term that should be as small as possible (generally $X_{i1} = 1$, intercept of line). **Predictors are combined linearly** (i.e. sum) **but can be any function of the raw input** (e.g. **logs, polynomials, interactions**). **Model is valid if it includes all relevant predictors**/features, **generalizes to new unknown cases** and predicts accurately **the phenomenon of interest**. There should be **no interaction between data points**, and ideally **normality/gaussianity of errors** with equal variance



Least squares: optimality criterion, since we want ϵ_i as small as possible we could also say that we want the **sum of squared errors to be as small as possible** (sign is irrelevant) and find $\hat{\beta}$ that minimizes it $\sum_{i=1}^{n}(y_i - X_i\hat{\beta})^2$

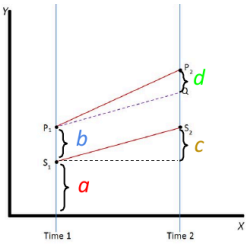
Linear regression for causal modeling and descriptive data analysis: comparing average outcomes across subgroups of data. When using Least Squares on $y_i = \beta_1 + \beta_2 x_i + \epsilon_i$, β_1 is the estimated mean outcome for data points with $x_i = 0$, β_2 is the difference in estimated mean outcome between data points whose x_i differ by 1





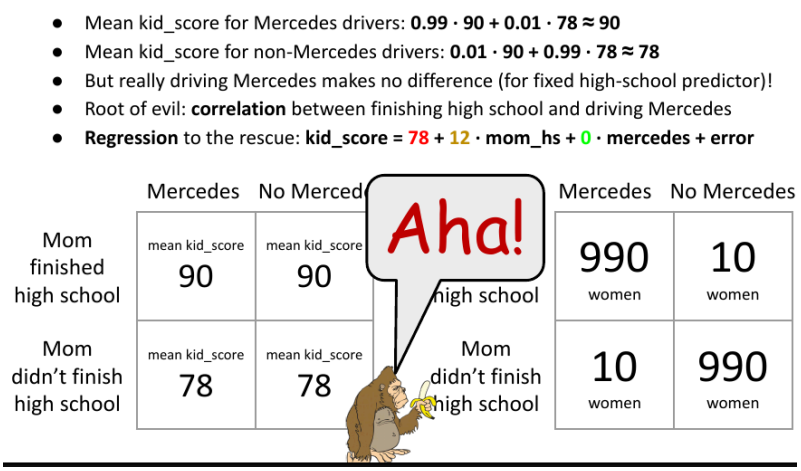
Also nice for **investigating quantification of uncertainty** (**significance**). Example: group P receives a treatment at time 2, groups S doesn't, P and S do not start out the same at time 1

Elegant linear model with binary predictors:
$y_{it} = a + b \cdot \text{treated}_i + c \cdot \text{time2}_t + d \cdot (\text{treated}_i \cdot \text{time2}_t) + \text{error}_i$
d = treatment effect
All of this with one single regression!
You get quantification of uncertainty (significance) for free!



Predictors/features interaction: when they are **multiplied with one another in the regression** function

Computing means themselves won't tell us the full story, *training a linear regression model allows us to understand how each feature influences the outcomes*



Statistical software when fitting linear regression gives a lot more data, like the ***p-value of estimating such an extreme*** β ***coefficient assuming that the true coefficient*** *expressing the real world data distribution* ***was zero*** *(****null hypothesis****)*

Residual: $r_i = y_i - X_i\hat{\beta}$, **estimation error on data point** i, assuming you used least squares on linear regression, the **mean of residuals is 0**, the **variance of residuals is the average squared distance of the predicted outcome from the real** one (**unexplained variance**)

R^2 **score**: **coefficient of determination**, $1 - \hat{\sigma}^2/s_y^2$, where $\hat{\sigma}^2$ is the **unexplained variance** and s_y^2 is the **variance of outcomes** y

Applying linear transformations to predictors: does **not change predicted outcomes or model fit** (R^2). **Subtracting the mean of a predictor for all data points** is an example, the **intercept** (β_1) now is the **estimated mean outcome when each predictor has mean value** (if there are interactions it's the same but each other predictor has to have its mean value). After that you could **divide a predictor by its standard deviation for all data points** (you get **z-scores**), **all predictors have same scale** (how many standard deviations a predictors is distant from its mean), great for **miscellaneous units in predictors**

Logarithmic outcomes: when your **outcomes** y **follow a heavy-tailed distribution** and are **positive**, **apply log to them**. **Turns an additive model into a multiplicative** one. Increasing b_1 by 1 multiples Y by b_1

$$\log y_i = b_0 + b_1 X_{i1} + b_2 X_{i2} + \cdots + \epsilon_i$$

sides yields

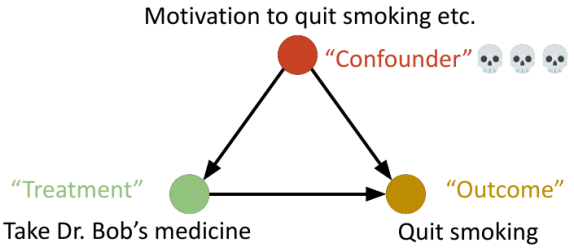
$$\begin{aligned} y_i &= e^{b_0 + b_1 X_{i1} + b_2 X_{i2} + \cdots + \epsilon_i} \\ &= B_0 \cdot B_1^{X_{i1}} \cdot B_2^{X_{i2}} \cdots E_i \end{aligned}$$

Logistic regression: perfect for **binary outcomes**, **equivalent to a neural network without hidden units and using cross-entropy loss and the sigmoid as activation function**

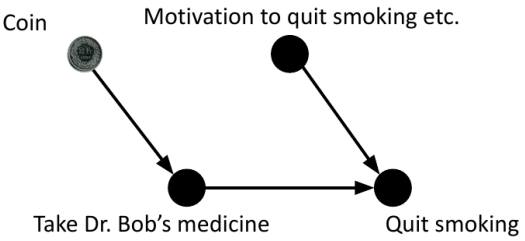
Poisson regression: perfect for **non-negative integer outcomes** (e.g. counts)

05 - Causal analysis of observational data

*When using **observational data** (data that was obtained through observing the world without intervention) to draw insights you might **look over confounder variables***



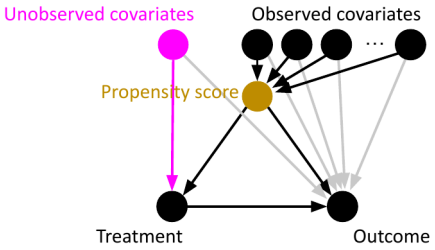
Randomized controlled experiments: **holy grail**, two experimental conditions/treatments (**medicine vs placebo**), participants are **assigned treatments randomly** with equal probabilities, **actual treatment and control/placebo groups are indistinguishable** (reduces influence of confounder variables). **Often expensive or impossible to carry out**



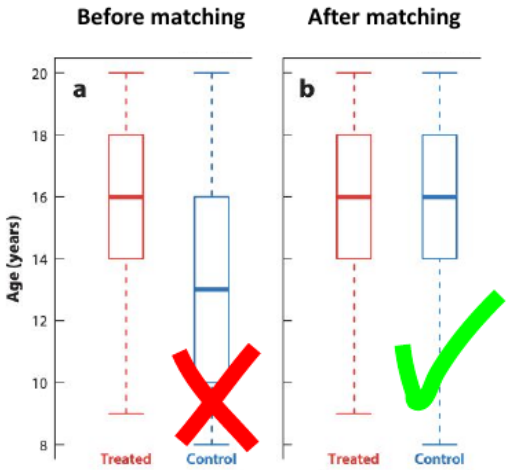
Natural experiments: **nature assigns treatments** to individuals, **close to the best**

Observational studies: **no intervention**, just **observation of reality**, very **feasible** but **drawing valid conclusions is difficult as subjects self-select to be treated**

Matching: allows **controlling for many potential confounders in observational studies**, **pair up 2 identical people** (1 treated and 1 placebo/control) and **compare outcomes**. The issues are **understanding if two people are identical** (**unobserved covariates can make two people with identical observed covariates be very different**, **unobserved covariates are ignored in the naive model**) and the fact that you **cannot match on all covariates if they are many** (**combinatorial explosion**). It makes sense to do it only if the **distribution of the covariates is not the same for the control and treated groups**



Propensity score: **compress observed covariates into the probability to receive a treatment**, P(subject is treated | observed covariates), this also means that P(observed covariates | treated = 1, propensity score = p) = P(observed covariates | treated = 0, propensity score = p), achieves **BALANCE** (**equal distribution of observed covariates for the same propensity score**). **Solves issues with matching** as you just need to **pair up people with the same propensity score** (**naive model because we only use observed covariates**)



Matching algorithms: **match subjects into pairs** (1 treated and 1 control) **with approximately same propensity scores**. Compute a **bipartite graph** where **each subject is connected to all other subjects** and the **edge weights are absolute or squared difference of propensity scores**, find **minimum matching via Hungarian algorithm**

Sensitivity analysis model: quantify **how much the naive model may be wrong in assuming that observed covariates determine similarity** in people, Γ defines a **max percentage of probability that identical-looking individuals may differ** in the naive model. So **after coming up with causal conclusions** you could say that they are **valid unless two identical-looking people have**, as opposed to our assumptions, **hugely different odds of receiving treatment** (i.e. huge Γ). And then reject that hypothesis as **common sense suggests that is not the case**

Odds: probability of something happening / probability of something not happening

Rosenbaum sensitivity analysis formula: uses **bounded odds ratio (OR)**: $\frac{1}{\Gamma} \leq \frac{\pi_k/(1-\pi_k)}{\pi_l/(1-\pi_l)} = \frac{P(k\ treated \mid k\ or\ l\ treated)}{P(l\ treated \mid k\ or\ l\ treated)} \leq \Gamma,\ \ \Gamma \geq 1$ whenever $x_k = x_l$ (same covariates), π_i = true probability for i to receive treatment, for $\Gamma = 1$ naive model is true, $\Gamma = 2$ people with same observed covariates could have different odds of receiving treatment up to a factor of 2, $\Gamma = \infty$ random (tautology)

Example: under naive model there is a very small p-value for the null hypothesis that smoking does not increase lung cancer risk, tobacco lobby says there are unobserved confounders that increase both the probability to enjoy smoking and the probability of lung cancer. Under sensitivity analysis model, increasing Γ increases the p-value for the null hypothesis, to reach a p-value of $0.05 \rightarrow \Gamma = 6$ which means that there exists an unobserved confounder that makes it 6 times more likely for one of two apparently similar individuals to get lung cancer

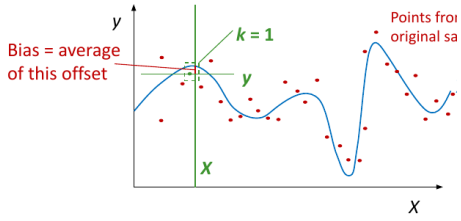
*Proving your science is about **building the causal study the right way** (**mechanical** part, creating pairs of people with similar observed covariates and propensity score) and **convincing everyone that your findings are not caused by unobserved covariates** (**scienfitic** fun part)*

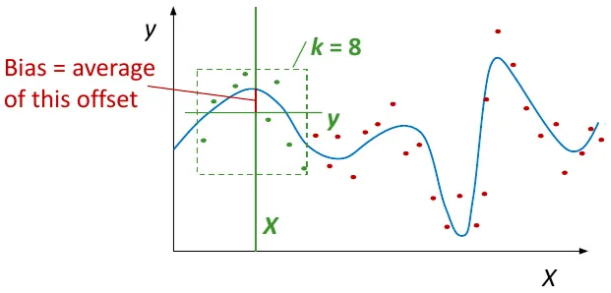
06 - Learning from data: Supervised learning

Supervised Learning: given **input/output pairs (X, y) that are related via a function** $y = f(x)$ that we would like to learn, **classification if y is discrete**, **regression if y is continuous**

Bias/Variance tradeoff: given a random training sample D, compute model f_D (**statistical estimate of the true function** $f(x)$), the error is $\mathbb{E}[(f_D(X) - y)^2]$ (expectation over D) that can be decomposed in $Error^2 = Bias^2 + Variance = \mathbb{E}[f_D(X) - y]^2 + \mathbb{E}[(f_D(X) - \mathbb{E}[f_D(X)])^2]$ (variance is included in the error because it means that for **a tiny change in the dataset a completely different model would be trained** (overfitting + increased error)). **Complex models** (**w/ many parameters and features**) have **low bias and high variance**, **Simple models** are **the opposite**

k nearest neighbors (kNN): find k **closest matches** in a labeled dataset given an input, return the **most frequent** (classification) **or average** (regression) **label among the k neighbors**, the **data is the model** since **no training is needed** (just **querying**), **simple** (**minimal configuration**, requires k**, similarity metric and weighting of neighbors' labels**) and **improves with more data**. **Smaller** k makes the **model** and its decision boundary **more complex thus low bias and high variance**





kNN distance measures: opposite of similarities

Euclidean Distance: Simplest, fast to compute

$$d(x, y) = \|x - y\|$$

Cosine Distance: Good for documents, images, etc.

$$d(x, y) = 1 - \frac{x \cdot y}{\|x\|\|y\|}$$

Jaccard Distance: For set data:

$$d(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|}$$

Hamming Distance: For string data:

$$d(x, y) = \sum_{i=1}^{n} (x_i \neq y_i)$$

Manhattan Distance: Coordinate-wise distance

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

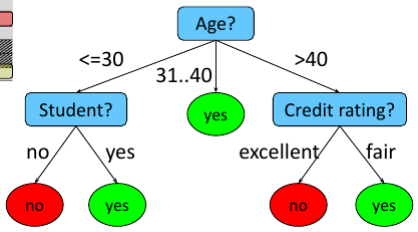
Edit Distance: for strings, especially genetic data.

Leave-one-out (LOO) Cross-Validation: **split data** into training and test subsets, **for each point** in the training set **predict using kNN with all other points in the training set and measure the LOO error rate** (classification) or **squared error** (regression). Repeat for **different values of** k and **pick the one with lowest LOO error**, finally **evaluate performance on test subset**

Curse of dimensionality: data in high dimensions is much sparser, and this **could make the distance between points be very high** (**so high that kNN would not work**), normally it does not happen because **samples tend to live in dense clusters**

Decision tree: **classification algorithm**, **nodes** are **tests on a single attribute**, **branches** are **attribute values of the parent node**, **leaves represent a class**. **Find a decision tree that maximizes classification accuracy** (**NP-hard**). Heuristically solved with **greedy top-down tree construction + pruning**, **each node partitions based on the most discriminative attribute** at that point in the decision tree (so that the **resulting branches are as homogeneous as possible**), **discriminative** power based on **information gain** (**ID3, C4.5**) or **Gini impurity** (**CART**), **stop partitioning** (adding nodes) if **all samples in a branch have the same class or if there are no attributes left**, **assign classes based on majority** voting. **Tends to overfit** and is **sensitive to small perturbation in the data**, it's **non-incremental** so if new data comes out if needs to be retrained. **Deep tree ⇒ high variance, low bias**

	income	student	credit_rating	buys_computer
<=30	high	no	fair	no
<=30	high	no	excellent	no
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
>40	medium	no	excellent	no



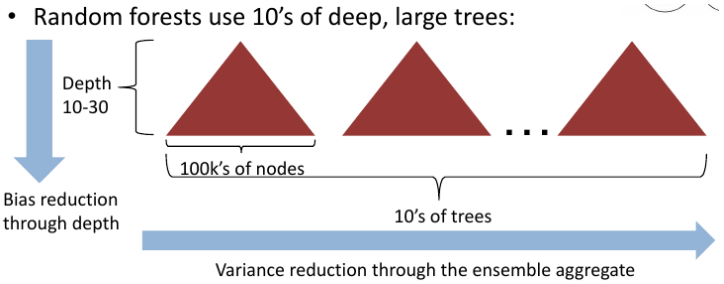
Entropy in Attribute Selection: given a set of samples S in a branch of the tree that has P positive and N negative samples, **entropy** is $H(P,N) = -\frac{P}{P+N}log_2\frac{P}{P+N} - \frac{N}{P+N}log_2\frac{N}{P+N}$

Information gain: given attribute A that partitions S in $S_1, \dots, S_v$, **entropy of the attribute** is $H(A) = \sum_{i=1}^{v}\frac{P_i+N_i}{P+N}H(P_i,N_i)$, the **information gain** is $Gain(A) = H(P,N) - H(A)$

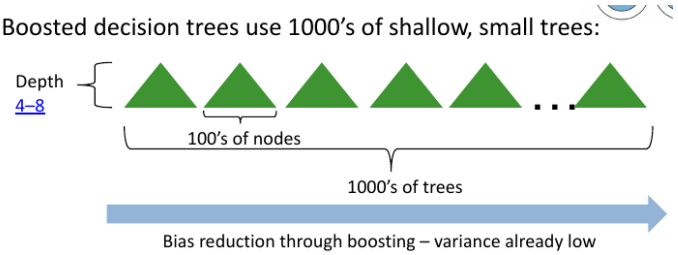
Pruning: tree construction phase **does not filter out noise and leads to overfitting**, solutions are **stopping partitioning when the number of data points in a leaf would be below a threshold** or after building the tree **replace nodes with leaves labeled with majority class if the classification accuracy on the validation set does not get worse**

Ensemble methods: collection of **simple or weak learners made into a single, better learner**. Can be **bagging** (**train learners in parallel on different samples of data**, then combine by voting (discrete y) or averaging (continuous y)), **boosting** (**train learner iteratively after filtering/weighting samples based on output of previous** learners) or **stacking** (**combine outputs from various models using a second-stage learner** like linear regression)

Random forests: **bagging ensemble method**, draw K **bootstrap samples**, **grow each decision tree** by doing the following **for each node**: **select a random set of** m (e.g. $\sqrt{p}$) **out of** p **features** and **split on the most discriminative feature**. **Aggregate the prediction of the trees** at testing time **to assign the final class**, we want **learners to be different from one another**. **Easy to implement and parallelize**, very popular but it **needs many passes** over the data



Boosted decision trees: **each tree is trained to predict** ("correct") **residual errors of previous trees**, thus **reducing bias. Final prediction is the sum of predictions made by the trees**



Logistic Regression: for **binary classification linear regression is not good** (output can be way below 0 or above 1) so we want to use a **linear function to get probabilities as** y **since they are in** $[0,1]$, we do this by **applying the log-odds to** $y \Rightarrow log(\frac{y}{1-y})$ and doing linear regression on that so that the output of the model is $[-\infty, +\infty]$. To **convert outcomes of the log-odds linear regression model back to probabilities we use the sigmoid**. Find the **best model using maximum likelihood or negative log likelihood** (MSE wouldn't work since it assumes data is generated from a Normal distribution), it's like using the **Cross-Entropy loss and assuming** y **is generated from a Bernoulli distribution**

Performance Criteria:

- **Predictive performance**: **accuracy, AUC/ROC, precision, recall and F1-score**
- **Speed and Scalability**: **time for training and inference of the model**, **in memory vs on disk processing** and communication cost

- **Robustness**: **handling noise, outliers and missing values**
- **Interpretability**: **understanding the model and its decisions** (black vs white box)
- **Compactness of the model**: **mobile and embedded devices**

07 - Applied machine learning

Classification pipeline: **Data collection**, **Model selection** and **Model assessment**

Data collection: definition of the **features that describe a data item and the class label** (requires **domain knowledge**)

Features: **continuous**, **ordinal** (e.g. “small”, “medium”, “large”) or **categorical** (e.g. country, gender)

Feature engineering: generating **new features using simple stats**, it’s a **form of art** so you should **look at how people did it**

Before 2012 (ImageNet CNN paper) ***features were manually cleverly designed*** *and were the juicy part of Machine Learning, after that with* ***Deep Learning features and model are learned*** *together*

Data labeling: often needed as **not already available**, used to be done by undergrads or expensive domain experts, now it’s **done through crowdsourcing with majority vote** (scammers are caught by putting easy questions for which if you put the wrong answer you don’t get paid)

Discretization/Binning: **some classifiers or feature selection methods** require discrete features, they also **allow linear classifiers to learn non-linear decision boundaries** (transforms curves into independent steps linearly seperable). **Unsupervised** (**equal width** (divide range into **bins of same range width**, bad for skewed data), **equal frequency** (divide range into **bins of same sample size**) or **clustering**) or **Supervised** (start with fine-grained discretization, test the hypothesis that **membership in two adjacent intervals of a feature is independent of the class** (through **chi-squared test**), if they are merge them)

Feature Selection: **removing irrelevant features** to have **more efficient training**, **less overfitting** and **more interpretability**. The possible combinations of retained features are 2^N. **Collectively relevant features may look individually irrelevant**!

Offline Feature Selection: **features are ranked according to their individual predictive power**, **independent of the classifier** (both a pro and a con), wrongly **assumes features are independent**. For **continuous features** the **Pearson’s correlation coefficient** is used (captures linear dependence). For **categorical features Mutual information** (goes beyond linear dependence) or **Chi-squared method** (checks the p-value of the independence of a specific feature with the class)

Online Feature Selection: **greedily add** (for **Forward** type) **or remove** (for **Backward** type) **features and evaluate** on validation dataset, **stop when performance does not improve**, **interacts positively with the classifier** + no feature-independence assumption but **computationally intensive**

Feature Normalization: some **classifiers can’t handle features with very different scales**, the **ones with bigger scales end up dominating** the others

Logarithmic scaling: $x'_i = log(x_i)$, considers **orders of magnitude instead of actual value**, good for **heavy-tailed features**

Min-max scaling: $x'_i = (x_i - m_i)/(M_i - m_i)$ where $x'_i \in [0, 1]$, M_i and m_i are the **max and min values** of x_i, **not good if data has outliers**

Standardization: $x'_i = (x_i - \mu_i)/\sigma_i$ where μ_i and σ_i are the **mean value and standard deviation** of x_i, new feature has mean 0 and standard deviation 1. **Assumes the data is generated by a Gaussian** distribution, **not meaningful for heavy-tailed data** (log-scaling can be applied before)

Model selection: picking the **model**, its **hyperparameters** and the **loss function**. 3-way split the data to **train the model** (**training set**), **find its best hyperparameters** (**validation set**) and **evaluate its performance** (**testing set**)

Cross-validation: used when you don't have enough data to do a 3-way split, **split data into K folds**, **train on $K-1$ folds** and **test on the remaining fold**, **repeat K times** so that you try all combinations. **Execute many times with different hyperparameters** and pick the **model with the best average performance**

Model assessment: **estimating the performance of a model** ideally under real-word conditions or at least with **held-out test data**

Confusion matrix: usual for **binary classification**, based on **true positives, true negatives, false positives and false negatives**

		Class	
		Pos	Neg
Classified	Pos	TP	FP
	Neg	FN	TN

Accuracy: $A = \frac{TP+TN}{N}$, appropriate when **classes are not skewed** and **all errors have same importance**

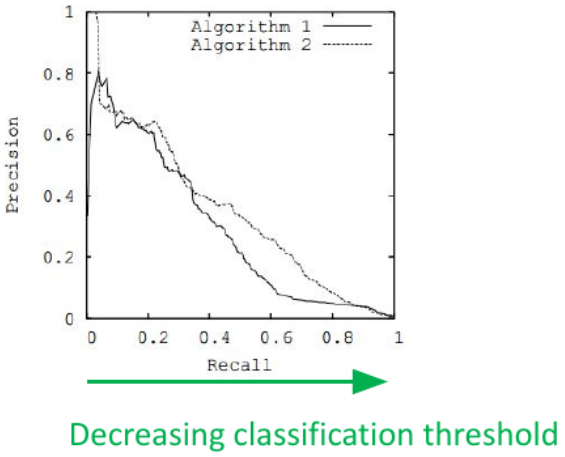
Precision: $P = \frac{TP}{TP+FP}$, what fraction of positive predictions are actually positive? **true positives rate**

Recall: $R = \frac{TP}{TP+FN}$, what **fraction of actually positive examples did I recognize**?

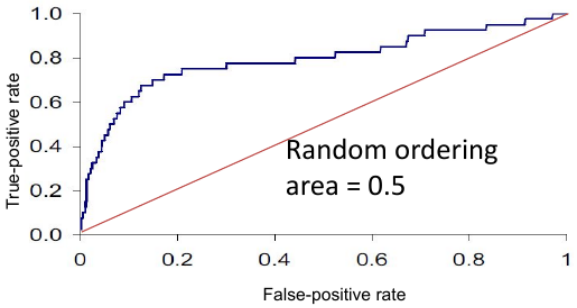
Specificity: $S = \frac{TN}{TN+FN}$, what fraction of negative predictions are actually negative? **true negative rate**

F1-score: $F1 = 2\frac{P \cdot R}{P+R}$ **harmonic mean of precision and recall** (can also be differently weighted), **single metric** to compare classifiers

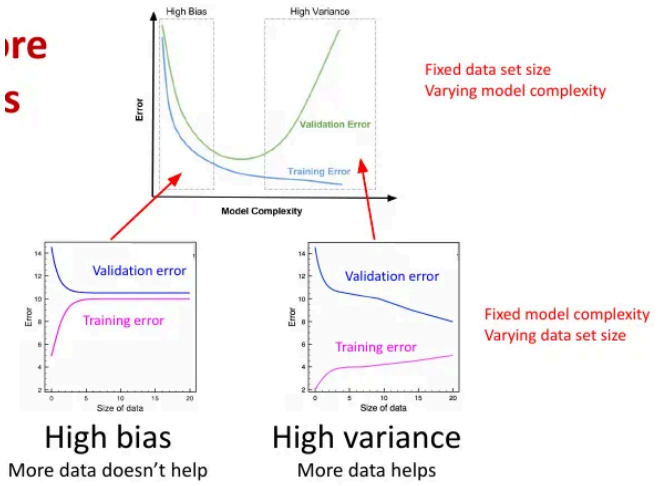
Precision/Recall curve: **every point of the graph is a different classification threshold** at which we compute precision and recall. As you increase the threshold recall gets higher because you classify positively more and more samples, precision goes down because most of those predictions are wrong (false positives)



ROC curve: Receiver-Operating Characteristic, **ROC AUC is the area under the ROC curve**, should be **above 0.5** (random classifier). **Recall** on the **y-axis** and **1 - Specificity** on the **x-axis**, **varying thresholds**



Estimating Bias and Variance using the validation error, also detects overfitting:



08 - Unsupervised Learning

Unsupervised Learning: given **only samples** X of the data, we **compute a function** $f \ \ s.t.\ y = f(x)$ **is a simpler representation**, called **clustering if** y **is discrete** or **dimensionality reduction if** y **is continuous**

Clustering: given a **set of points and a notion of distance** between points, **group the points into some clusters** such that **their members are close** (i.e. similar) to each other and **members of different clusters are far apart** from each other. **Similarity is defined via euclidean**, **cosine distance for vectors** and **jaccard index for sets**. Can have **quantitative** (**scalable**, and can handle **high dimensionality**) or **qualitative properties** (**types of features and shapes** they can handle), **robust to noise** and **outliers** (points **far from all other points**), and include **user constraints**, have **improved user interpretability and usability**. Used for **data exploration** (especially in high-dimensions), **partitioning data** (for fine-grained analysis), **marketing** (building personas), **data labeling** (when having **few labels**, you can assign labels to clusters), **feature discretization** (discretize a continuous feature by assigning it a category value based on its cluster made on that feature, **alternative to bins**) and **data compression** (**clusters represent data points that can be then discarded** instead of meaning)

Cluster bias: **wrongly assuming that the underlying distribution is discrete and therefore wrongly performing clustering** (to produce discrete labels/number of clusters), but in reality it's continuous and a continuous model do better

Hard clustering: items assigned to a **unique cluster**

Soft clustering: cluster membership is a **probability distribution over all clusters**

Clustering is hard: with large amounts of data or features (curse of a dimensionality)

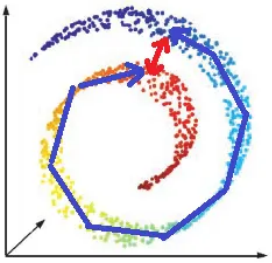
Clustering applications: **movies in netflix**, have a dimension representing movies seen by a particular customers, **cluster movies based on which customers have seen them**, find movies of similar genre (recommendation). **Finding topics in docs**, have a dimension representing the presence of a particular word, **cluster documents based on the words they contain**, find docs of similar topics

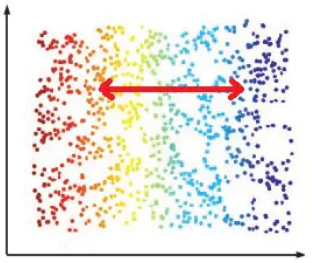
Hierarchical clustering: clusters form a **tree-shaped hierarchy** (so **clusters contain clusters**). **Divisive** (top-down), **start with one cluster and split it**. **Agglomerative** (bottom-up), **each point is a cluster** and you **combine the two "nearest" clusters** in one. Normally $O(N^3)$ (for each point compute the distance between all points), with priority queue $O(N^2 log N)$

Agglomerative Hierarchical clustering in $\mathbb{R}^n$: a **cluster** with many points it's **represented by its centroid** (**average of points in it**) and the **nearness of clusters** is represented by the **distance between their centroids**

Agglomerative Hierarchical clustering not in $\mathbb{R}^n$: a **cluster** with many points it's **represented by its clustroid** (**data point closest to all other points**, measured as average distance from all points, sum of squared distances or max single distance) and the **nearness of clusters is represented by the intercluster distance** (minimum of the single distances between any two points or their average, or distance between clustroids) or the **cohesion of their union** (diameter of merged cluster (maximum distance between points in the merged cluster) or average distance of points in the merged cluster)

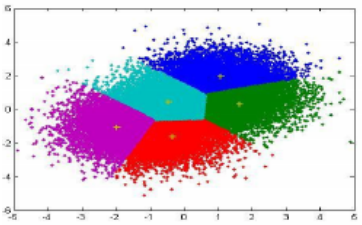
Flat clustering: **no inter-cluster structure** (clusters do not contain clusters, **unlike hierarchical clustering**), create a set of **initial clusters**, **assign the nearest cluster to some points**, **recompute clusters**, repeat





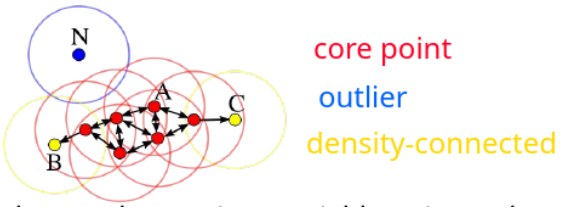
K-Means clustering: popular **flat** clustering, NP-hard, **assigns each point** to one of **k clusters** so that the **total distance of points to their centroids is minimized**. Solved by a simple **greedy algorithm** (**solutions may be suboptimal**) that for **each point it assigns the cluster with the closest centroid and then recomputes the centroid**. Iterate for a **fixed number of iterations or until there are little to no changes in cluster tightness** (sum of distances of points from the centroid). **Requires some initial clusters**, a notion of a **mean and distance**, **specifying how many clusters k** and **doesn't handle noisy data and outliers well**. **Produces ONLY convex spherical shapes** and works well for **data compression**

K-Means++: constructs **random good spaced initial clusters**, choose as **center of the first cluster a random data point**, **for all other data points** compute the **distance to the closest already picked cluster center**, make a **new cluster with as center a data point randomly chosen with probability proportional to the squared distance** calculated previously. **Helps finding the global minimum**



Choosing k for K-Means: **for each possible k**, run **k-means** and compute the **average S silhoutte width for all data points** ($s(i) = \frac{b(i)-a(i)}{max\{a(i),b(i)\}}$, $a(i)$ average distance to points in own cluster, $b(i)$ average distance to points in closest other cluster). Pick k for which S is highest, you are **maximizing distance to the closest cluster and minimizing intra-cluster distance**

DBSCAN: **density-based spatial clustering of applications with noise**, **follows the shape of dense neighborhoods of points** (even if not spherical). **Core points have at least $minPts$ reachable neighbours in a sphere of diameter ϵ** around them. **Border points** are within the ϵ radius of a Core point (**directly density-reachable**). Point q **is density-reachable from** p if there is a **series of points** $p \to p_1 \to \cdots \to p_n \to q$ such that $p_i \ \forall i$ is a Core point. All **points non density-reachable from any other points are outliers/noise**. Points p, q are **density-connected** if there is an **ancestor point** o such that p and q are density-reachable from o. A **cluster is a set of points that are mutually density-connected**, so **if a point is density-reachable from a cluster point then it's in it**. Using appropriate indexing structures overall algorithm takes $O(nlogn)$, neighbor search becomes harder in higher dimensions (**curse of dimensionality**). It creates **neighborhoods of points with high density and leaves out outliers** (less sensitive), $minPts$ regulates how many points have to be in a dense region, ϵ how big is the region around a single point when considering density



09 - Handling text data

*Modern data is mostly **unstructured text coming from the internet**, social medias and news*

Document retrieval: given a **document collection and a query rank all docs in the collection by similarity to the query**, web **search engines**. Straightforward approach is **neighbour search**, the **hard part is determining the distance function and scaling it** (typically **cosine distance**). **Filter documents by presence of query terms** (with efficient full-text indexing) to reduce number of pages to rank

Document classification: given a **document and a set of classes** (e.g. topics), **decide to which of the classes the document belongs**, **supervised learning with a large collection of documents**, each is represented as a **feature vector**. **TF-IDF matrix** can be used as **feature matrix** but having **more words** (**features**) **than documents lead to overfitting** (**high model feature capacity**), solved by **using more data**, **decreasing model capacity** (feature selection, **regularization**, dimensionality, reduction) or **ensemble methods** (randomforest)

Sentiment analysis: given a **document calculate a sentiment score that captures how positive/negative it is** (e.g. infer what people think of something), can be **treated like classification** (same approach as document classification) or **regression** (also pretty much the same)

Topic detection: given an **unlabeled document collection, determine a set of prevalent topics in the docs and for each document to which topics it belongs** (e.g. trending topics on X), solved with **hierarchical or point-assignment clustering** (e.g. K-Means, DBSCAN) **or matrix factorization**, also represents documents as **feature vectors**

Feature vectors: nearly **all ML methods use them**, for text we **convert arbitrarily long strings to fixed-length vectors** (traditionally **bag of words**, now **text embeddings mappings**)

Bag of words: compute a **vector with all the words appearing in your collection of documents**, **convert each document to a vector** in which **at every index it's stored how many times the word associated with that index appears in the document**, **very sparse and high dimensional**

Zipf's law: **power law**, the **probability of observing a word scales inversely with its frequency rank**, $p(w_i) \propto 1/i$ where w_i is the i-th most frequent word

Bag of words matrix: each **row is the vector representing a document through bag of words**, very sparse so sparse matrix format is used. **Normalization and weighting of rows and/or columns** increase performance

Character encoding: **improves bag of words**, **mapping chars to bytes** (ASCII, unicode), each **file is written with an encoding** that should also be used for reading, use **UTF-8/16**

Language identification: **improves bag of words**, **libraries look for common letter trigrams** (e.g. "eau", "ghi")

Tokenization: **improves bag of words**, **maps strings into sequence of tokens**, can be done manually but it's better to use **libraries**, **varies depending on the language** (easy in english, not so easy in german)

Stopword removal: **improves bag of words**, very frequent short words that carry little information for most tasks and dillute it. **Stopword lists depend on the task**, a **good heuristic is removing words appearing in at least p% of all documents**

Word normalization: **improves bag of words**, **casefolding** (making everything **lower-case**, makes sense to do it **only if you have a few documents and want to reduce sparsity**), **stemming** (**map different forms**

of the same word to a normalized form, **discards information** so typically **not done anymore**, **especially for english** as it's not morphologically rich (e.g. german)) and **lemmatization** (**stemming on steroids as it also considers surrounding words** to better map them to the normalized form, not done as it **requires complex mappings**)

n-grams: **multiple words together can have better and more precise meaning than a single out of context word** (e.g. operating system), if you **compute all possible n-grams** (brute-force) then you have **combinatorial explosion**. **Feature selection is smarter**, for **bigrams** (n=2) **keep them if mutual information between the two tokens is large**. For n>2 retain n-grams matching wikipedia anchor texts, satisfying frequent sequence mining or passing compressive feature learning

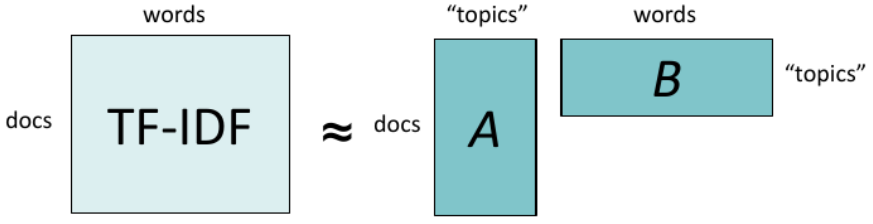
Inverse document frequency (IDF): **not all words are equally informative** (like in **stopword removal**), the **most informative ones are often the ones appearing more rarely**. $docfreq(w) = \#$ documents that contain word w, $N = \#$ documents, $idf(w) = -log(docfreq(w)/N) = log(N) - log(docfreq(w))$, how much **discriminating power the event "randomly drawing a document that contains w" holds**

Term frequency (TF): what bag of words captures

TF-IDF matrix: entry in **row** d (document) **and column** w (word) **has value** $tf(d, w) * idf(w)$, same as scaling columns w of bag of words by constant $idf(w)$ (**column normalization**, also standardization or min-max scale could help). **Vectors lengths are greater for longer docs**, this **throws off ML algorithms** because long documents have vectors far away from the ones of short documents (magnitude) so the **dot product with a long document will be big in most cases** (making it seem the documents are similar, and also producing **errors of high magnitude**) so we do **L2** (makes vector distance from the origin 1) or **L1** (all rows sum to 1, **like a probability distribution**) **normalization**. **Two docs are similar if they have cosine similar rows** in the TF-IDF matrix, **two words are similar if they have cosine similar columns** instead

Topic Detection with TF-IDF: **cluster rows of the TF-IDF matrix** (**K-means**, each row is a data point), **manually inspect clusters and label them** with descriptive names. **Clustering can be difficult** because dimensionality is large (way more words than docs), **curse of dimensionality and many outliers**

Latent Semantic Analysis for Topic Detection with Matrix Factorization: solves the problem before, **assume that docs and words have representation in the latent "topic space"**, TF-IDF modeled as dot product of doc's vectors and word's vectors in the topic space. **Topic space dimensionality is way smaller** compared to the one of the word space. Topics are interpretable in doc space (A's cols) and word space (B's cols), because **topic representation is a vector** (not a probability distribution over topics). It's an **optimization problem**, find A, B **such that** AB **is as close as possible to the TF-IDF matrix** (T), minimize $\sum_{d=1}^{N}\sum_{w=1}^{M}(T_{dw} - A_dB_w)^2$ where A_d is the d-th row of A and B_w is the w-th column of B, called **Latent Semantic Analysis (LSA)**. Efficient way to accomplish it **using singular-value decomposition** (**SVD**) where $T = USV^T$ (columns of U and rows of V are unit vectors ($||v|| = \sqrt{\sum_i v_i^2}$) and orthogonal to each other (dot product 0), S is diagonal (with decreasing values) and **captures "importance" of each topic** (amount of variation in corpus w.r.t. topic)), if you want k topics keep only the first k columns of U and V, and the first k rows and columns of S (so $A = U', B = S'V'^T$). You can now **cluster rows of A instead of TF-IDF which has dimensionality same as topic space** (way lower than word space)



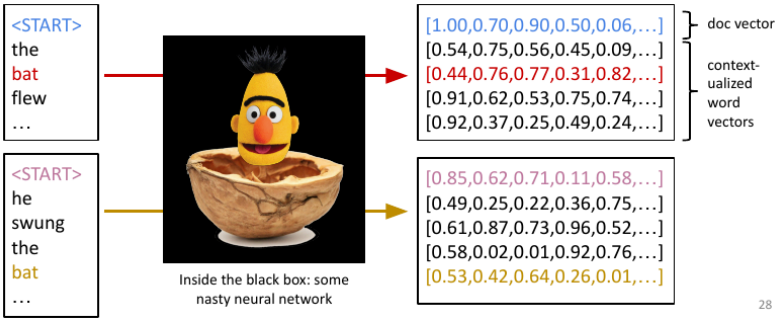
Advantages of dense vectors with LSA: we can **tell that different words with the same meaning are semantically equal** (since **columns have semantic meaning**, not just 1 if word is present in a doc or 0 like in **sparse TF-IDF rows**). A **disadvantage** of both **LSA and TF-IDF** is that **we lose info about word proximity and syntax**, instead of full docs **consider windows of** L (local contexts) **consecutive words to left and right of the target word** (so **each row is a context**, not a doc). **word2vec** does this and also, instead of TF-IDF, it **assigns to each context point mutual information** (PMI) $M[c,w] = PMI(c,w) = log(\frac{P(c,w)}{P(c)P(w)})$ (how much more likely are c and w to occur together if they were independent)

Regularization: **penalizes very large absolute values in weights**, for example say in the training data we see a certain feature being always linked to a certain label, normally we would set incredibly high weights on that feature because it reduces the training error but if in unseen test data there are some samples not following that rule we would get them wrong (we are **overfitting on the training data**)

Latent Dirichlet Allocation (LDA): **probabilistic topic modeling**, treats a **document as a bag of words** (order doesn't matter) and a **topic as a probability distribution over words** (topic "sports" has high probabilities for "ball"), each **document has a (latent) distribution over topics** (e.g. 60% politics, 40% comedy). **Assumes the document was generated starting from a probability distribution** d **over topics** ("generative story") and **contains** n **words**, each **word was picked by sampling a topic** t **from** d and then a **word from the sampled topic distribution** t. **Unsupervised**, topics come out on their own but you define how many, **topics are found so that their distributions maximize the likelihood of the observed document**

Representing texts: word vectors represent well words, to **represent sentences we could sum/average word vectors** (also what bag of words are with one-hot encoded vectors of words), more recently **learning algorithms for longer texts** (sent2vec, **CNNs**, recurrent neural networks, **transformer**-based models like **BERT**)

Contextualized word vectors: needed because **classic word vectors cannot understand the meaning of a word depending on its context** (e.g. bat like the animal or like baseball), **BERT does it**



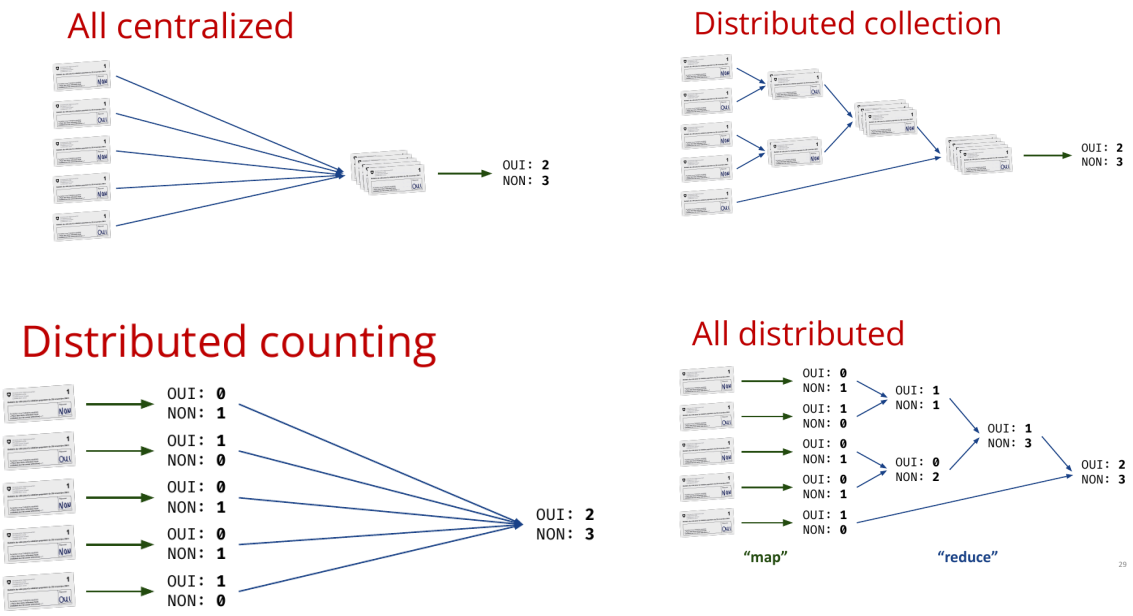
NLP pipeline: tokenization, sentence splitting, part-of-speech (POS) tagging, named-entity recognition (NER), coreference resolution, parsing (shallow, constituency or dependency). Implemented in **libraries** through **sequential models that have a fixed order of steps** (not always optimal) in which **early errors propagate downstream**, **current research** is trying to **learn all tasks jointly**

10 - Scaling to massive data

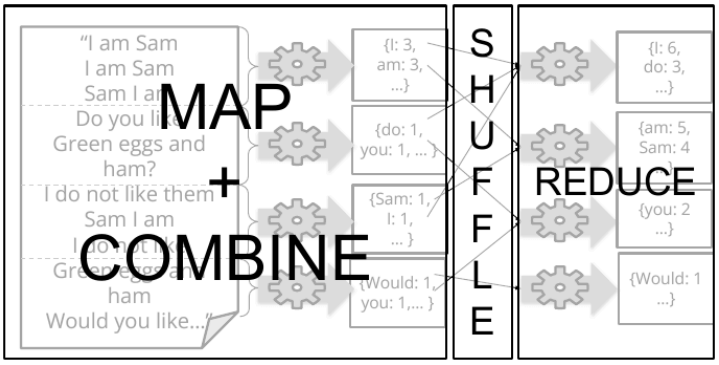
Data is growing faster than computation speed**, the **data sources are becoming enormous** and they often **can't be loaded into memory** as a whole. Slow read speeds → must distribute, stalling CPU speeds → must parallelize, RAM bottlenecks → must stream. **Data science lives in the cloud

Cluster computing with cheap consumer hardware: many **low-end servers combined** (easy to add capacity, **cheaper per CPU and disk**) connected in a **consumer network**, **increased complexity in software** (fault tolerance, virtualization) and **unreliable + uneven performance + latency**. **Splitting work across machines** and dealing with failures **is hard**

Distributed aggregation: example with ballots



MapReduce/Hadoop: normally you would **count occurrencies of words in a sentence using an hashtable** (dict), if the **document is very big you could split it** into portions and **map it in parallel**, then **combine the end results** by adding them (**divide and conquer**). **MapReduce** was developed to **have everything needed for this task in cluster computing**, you just need to implement **Mapper and Reducer classes**. If you have failures just launch another task



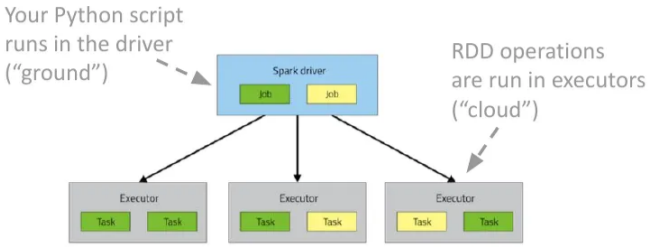
Spark: **high-level functional programming API** implemented in **Scala**, additional APIs in Python, Java and R, many abstractions (Core: RRDs, Derived: dataframes, SQL, ML toolkit)

```
users = (sc.textFile('users.tsv') # user <TAB> age
           .map(lambda line: line.split('\t'))
           .filter(lambda user_age: 18 <= int(user_age[1]) <= 25))

pages = (sc.textFile('pageviews/*.tsv')  # user <TAB> url
           .map(lambda line: line.split('\t')))

counts = (users.join(pages)
               .map(lambda user_age_url: (user_age_url[1][1], 1))
               .reduceByKey(lambda x, y: x + y)
               .takeOrdered(5))
```

Spark RDDs: **resilient distributed dataset**, to the dev it's just a dataset, under the hood the **dataset lives in the cloud split over several machines and replicated**, can be **processed in parallel** and can be transformed in a single real dataset (if small), typically read from a **distributed file system** (**HDFS**, can also be written to it)



RDD actions: outputs a value returned to the driver (**from cloud to ground**), Examples are: **collect (return all dataset elements), count (dataset length), reduce(func) (computes a value), take(n) (like head), saveAsTextFile**

RDD transformations: outputs another RDD (remains in the cloud), **lazy executed** (not executed until it's necessary) **triggered by actions** (if you never look at the data, there's no point in manipulating it, can be **optimized based on usage**). Examples are: **map, filter, flatMap**, **sample(withReplacement, fraction, seed), union(otherDataset), intersect(otherDataset), distinct**, **groupByKey, reduceByKey, sortByKey, join(otherDataset) (like SQL)**

Variables broadcasting: `my_set = set(...); rdd2 = rdd1.filter(lambda x: x in my_set)` bad idea because `my_set` will be shipped with every task per data partition even if they are on the same executor, instead do `my_set = sc.broadcast(set(...)); rdd2 = rdd1.filter(lambda x: x in my_set.value)` this way is **copied once to each executor and persisted across tasks**. **Broadcasted values are read-only**

RDD Accumulators: used for **counting rows while performing transformations**, write only for executors, read only for the driver. It would not work otherwise because **modifications to local/global variables are not propagated to the driver**

Side effects via accumulators!

```
counter = sc.accumulator(0)
def f(x): counter.add(1); return x*2
rdd2 = rdd1.map(f)
```

RDD persistance:

```
rdd2 = rdd1.map(f1)
list1 = rdd2.filter(f2).collect()
list2 = rdd2.filter(f3).collect()
```

} `rdd1.map(f1)` transformation is executed twice

```
rdd2 = rdd1.map(f1)
rdd2.persist()
list1 = rdd2.filter(f2).collect()
list2 = rdd2.filter(f3).collect()
```

} Result of `rdd1.map(f1)` transformation is cached and reused (can choose between memory and disk for caching)

Spark DataFrames: **bridging the gap between distributed computing and Pandas Dataframes** that are higher level (they are **just a table with rows and columns**)

```python
users = spark.read.csv("users.csv", header=True)
views = spark.read.csv("pageviews/*.csv", header=True)
counts = (
    users[users.age.between(18, 25)]
      .drop("age")
      .join(views, on="name")
      .groupby("url").count()
      .sort_values("name", ascending=True)
      .head(5)
).toPandas()
```

79

```python
counts = spark.sql("""
  SELECT url, COUNT(*)
  FROM users
  JOIN views ON users.name == views.name
  WHERE 18 <= age AND age <= 25
  GROUP BY url
""")

counts.toPandas()
```

11 - Networks

Network: a **real-world system of dependent variables** (e.g. society, the web)

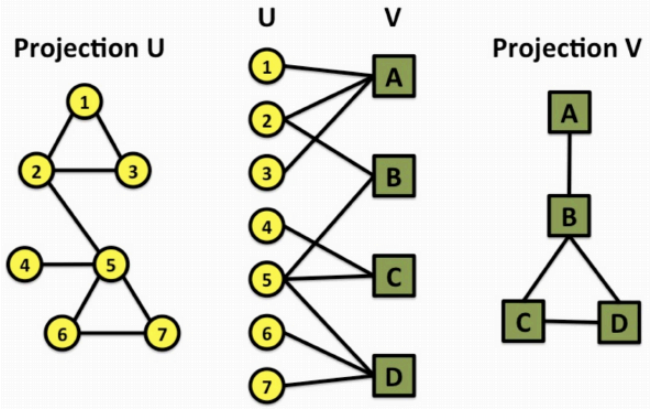
Graph: **mathematical abstraction for describing networks**, you **can make a graph out of everything** (but it **may not correspond to a meaningful network**). Entities are represented with nodes/vertices V, relationships/interactions are represented with edges/links E, the graph is $G = (V, E)$. Links can be **directed** (symmetrical, e.g. friendships on Facebook) or **undirected** (arcs, e.g. following on Instagram)

Node degree: number k_i of nodes adjacent to node i, for directed graphs you have both **in-degree** k_i^{in} and **out-degree** k_i^{out}

Average node degree: $\bar{k} =< k >= \frac{1}{N}\sum_{i=1}^{N} k_i = \frac{2E}{N}$

Weighted graphs: **each link has a weight associated** to it (e.g. roads)

Bipartite graphs: nodes can be divided into two disjoint sets U and V such that every link connects a node in U to one in V (e.g. users and the movies they rated). **Folded/projections networks** only look **how nodes in U are connected with one another through the nodes in V**, and viceversa



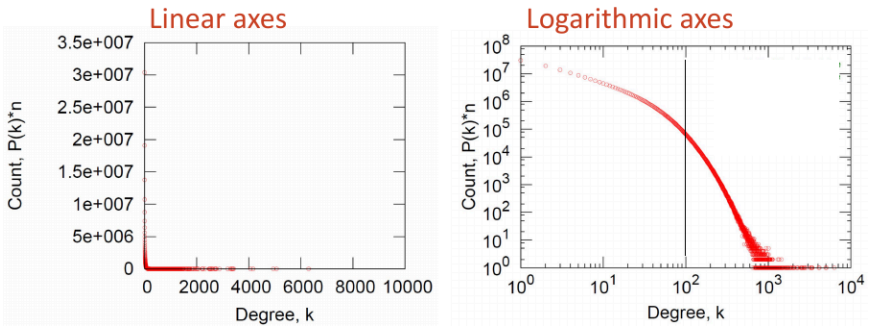
Representing graphs:

- **Adjacency matrix**, $A_{ij} = 1$ if there is a link from node i to node j, otherwise 0. **Usually sparse**
- **Edge list**, represent graph as a **list of edges**
- **Adjacency list**, **for each node have a list of adjacent edges**, **easier** to work with **if network is large or sparse** and we need to **retrieve all neighbors** of a node

Erdős-Rényi (ER): simplest and **most popular model for generating random graphs**, edges exist with probability p independently from one another, average clustering coefficient of a node is linearly proportional to p

Real-world networks tend to share some general properties:

- **sparsity**
- **degree distribution**, the probability that a randomly chosen node has degree k ($N_k = \#$nodes with degree $k \rightarrow P(k) = \frac{N_k}{N}$) **is often skewed in the real-world**



- **triadic closure**, "a friend of my friend is my friend", measured by clustering coefficient C_i of node i, $C_i = (e_i = \#$edges between the neighbors of $i)/(\#$potential edges between the neighbors of $i) = \frac{e_i}{k_i(k_i-1)/2}$, in real-world networks C_i is normally very large
- **community structure**, triadic closure makes **real networks cluster into locally dense communities with strong ties inside**, **communities are connected via weak ties that fill "structural holes"**. Communities are often overlapping
- **average shortest-path length**, how many steps long is the shortest path between any two nodes on average is usually not so long in real-world networks
- **navigability**, there exists short paths in the real word that **connect any two nodes** (e.g. wikispeedia) **partly due to hubs** (skewed degree distribution) that make it also **easier to find short paths**
- **homophily/heterophily**, **similar nodes are prone to be closely connected** with one another, the opposite for dissimilar nodes is also true

Node centrality: used to measure **node importance**, map to each node i a scalar value $C(i)$ capturing its importance in the overall network

Degree centrality: simplest, **the more neighbors you have the more central you are**, $C(i) = k_i$ (**easily rigged** (e.g. with scam accounts))

Closeness centrality: **the closer you are to all nodes** (**easier to reach**) **the more central you are**, $Fairness(x) =$ total distance to x from other nodes, $C(x) = 1/Fairness(x)$. **Defined only for connected graphs**

Harmonic centrality: variant of closeness centrality where $C(x) =$ total reciprocal distance of x to other nodes, **defined even for disconnected graphs**

Betweenness centrality: **the more shortest paths go through you the more important you are** (source and destination nodes do not count). $C(i) =$ average fraction of all shortest paths in the network that pass through node i. **Expensive to compute** because it requires to know all shortest paths

Katz centrality: **the more paths lead to you the more central you are**, **robust generalization of degree centrality that counts also neighbors at all distances**, $C(i) = \sum_{k=1}^{\infty} \sum_{j=1}^{N} \alpha^k (A^k)_{ji}$ with $(A^k)_{ij}$ the k-th power of the number of length-k paths connecting nodes i and j

PageRank centrality: **recursive** definition, $C(i) = x_i$ is **high if** x_i **receives inlinks from many other central nodes**, $x_i = \sum_j a_{ji} \frac{x_j}{L(j)}$ and $L(j) = \sum_i a_{ji}$ total outgoing link from node j. Matrix notation: $x = Mx$ where M is computed from the adjacency matrix A by dividing each column by $L(j)$ and x is

eigenvector of M with eigenvalue 1. x_i is a fraction of time a random walker will have spent in node i after a very long ($\rightarrow \infty$) random walk (**steady state of the Markov chain induced by the network**)