Kontextsensitive Systeme

SoSe 2020

1 Introduction

- **Defining Context:**
  - Parts of a discourse that sourround a word and throw light on its meaning
  - Synonym for: Sourrounding, Situation, meta-information
  - **Aspects**: Geographical, Physical, Organisational, Social, Emotional, User, Time, etc.
  - Computer Science:
    - Computer Linguistic: surrounding information of text
    - System Architecture: virtual environment to suspend running program
    - Lexical context which determines name resolution
- **Pervasive Computing Systems:**
  - Convenient access through new class of appliances to relevant information
  - with ability to take action on it **when and where you need**
- **Why context?**
  - Human communication:
    - situational info is implicit, to increase **conversational bandwith**
  - **Goal**: Make Interaction with computer systems more efficient
    - limiting factor of IT: often not processor/memory but limits of human attention
    - Mobile Computing: situation changes frequently and user is preoccupied with other tasks
    - **Context-awareness is enabling technology** → overcome bottleneck
- **Different Aspects of context**
  - Three important aspects: Schilit et. al
    - Where are you? Who are you with? What resources are nearby?
  - Four Dimensions of context: Gross and Specht
    - Location, Identity (preferences, knowledge), Time (working hours), Environment or Activity
- **Context classification:**
  - Active vs. Passive: set of environmental states and settings that
    - active context: determines applications behaviour (critical)
    - passive context: in which an application event occurs and is interesting to user (relevant)
  - Primary vs. Secondary: information that characterizes situation of entity
    - primary context: location - where, identity- who, time - when and activity - what
    - secondary context: anything that can be found on basis of primary context
  - Context-awareness of systems:
    - if it uses context to provide relevant information / services to user
    - where relevancy depends on users task

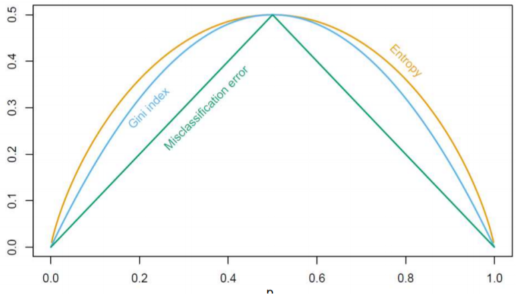
- **Classes of context-sensitive applications:**
  1. Proximate selection & contextual information:
     - **Trigger** = manual / explicit & **Output** = Information access
     - **Examples**:
       * google now → weather in the city (filtered and selected information)
       * show nearest available printers based on location
       * parking assistance that is triggered from driving backwards
  2. Contextual commands:
     - **Trigger** = manual / explicit & **Output** = Command execution
     - **Examples**:
       * Depending on the situation → emergency button call family / ambulance
       * Use nearest printer for printing after command is executed
       * multi purpose buttons that work differently in every context
  3. Automatic contextual reconfiguration
     - **Trigger** = automatic / implicit & **Output** = Information access
     - **Examples**:
       * change Notification from sound to vibration depending on situation
       * use idle computers found nearby for storage rather than remote or local disks
       * start parking assistance when driving forward very slowly
  4. Context-triggered actions
     - **Trigger** = automatic / implicit & **Output** = Command Execution
     - **Example:**
       * automatic closing / opening of windows depending on light conditions
       * Contextual reminders like if at workplace show certain messages
- **Motivation: Context through Sensors:**
  - Human Context Recognition:
    * Aristoteles: sight, hearing, touch balance, joint motion and acceleration
    * Additional: smell, taste, pain and thermoception
  - Sensors:
    * Enable interaction by providing context
    * Number of smartphone sensors is increasing:
      · Light, Proximity, Acceleration, Touch, Rotation, Temperature, Compass, etc.
      · E.g. tilt → Orientation and Light-Sensor → Brightness
- **Classical Sensor Systems: Measurement Process**
  - Classic measurement:
    * **Idea**: precisely control measurement conditions
    * Direct measurement, controlled placement and controlled environment
    * **But**: only appropriate if precision is more important than cost
  - DIN 1319:
    * Define measurement task & unit as well as all conditions exactly
    * Select and calibrate measurement device and specify the process
    * Take the measurements and calculate noise influence and systematic error
    * Calculate overall result and quantitative error margin

- **Sensors and Context:**
  - **Context** c transforms **signal** s to s^* at **state** x: $c(x, s) = s^*$
  - Context Recognition: learn $c^{-1}(s, s^*) = \hat{x}$
    - **But:** typically x is nominal, c is no function and c^{-1} does not exist
- **Complexity of Recognizing Contexts:**
  - Complex context can only be sensed indirectly: affect environment and captured through sensors
  - **Problem**: Sensors capture only partial effects and are noisy
  - <u>Representable Context:</u>
    - Information that can be encoded and is separable from activity
    - Delineable: define contexts relevant for application in advance
    - Stable: determination of relevance of contexts on an activity can be made once
  - **Dey**: Context built-in at design time (context representable and processable)
  - **Dourish**: technology becomes meaningful as individuals engage with it (context is emergent)
- **Detection Chain**
  1. Physical Phenomenon: visible phenomenon in real world
  2. Detection: Data comes from a sensor
  3. Normalization: properly scale data
  4. Feature Extraction & Reduction: compression, filtering and preparing of sensor data
  5. Classification: interprets features and assigns class for them
  6. Further abstraction: high level information and reasoning
  7. Activity: action depending on output
- **Data Processing Chain:**
  - <u>Problem</u>:
    - Input: analog, continuous sensor signal
    - Output: symbolic, discrete context class
  - <u>Sampling sensors yields discrete measurements</u>
    - High volume, depending on sampling rate
  - <u>Theory: sensor data as direct input for classifier</u>
    - But: greatly increases classifier complexity
    - Solution extract information of importance
- **Resampling and windowing data**
  1. <u>Analog Sensor Signal:</u>
     - Transform from time domain and specify period of time to be classified
     - **Problem**: Window lengths greatly affects classification results
  2. <u>Sensor Sampling:</u>
     - Nyquist/Shannon not directly applicable: understanding theoretical possible
     - Reconstruction not necessary and features do not need to be perfect
  3. <u>Sample Windows:</u>
     - Long: Smearing of context / latency (better: rolling window)
     - Short: Information does not characterize context and noisy results

2 Classification and Error

- **Step 1: Sample Windows**
  - Specify period of time to be classified: segment data into windows of that time length
  - **Problem**: Window length greatly affects classification results
  - Practice: empirical evaluation for each application
- **Step 2: Feature Extraction**
  - Goal: extract important information for context differentiation (classification) → reduce volume
  - Importance of information is depends on context and sensors
  - Features can be either in time or frequency domain (Fourier Transformation)
- **Step 3: Classification**
  - Features append to feature vector → training with additional labels
  - Classification: uses feature vector to predict next occurrence
- **K-Nearest Neighbour: most simple classifier**
  - **Approach**: store all training vectors and labels
    * For each classification vector: assign label based on "closest" vectors
    * $d(p,q) = d(q,p) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$
  - Metrics for distance calculation
    * Manhattan, Hamming, Mahalanobis, Euclidean
  - **Good**: simple to implement, and search can be efficient through trees as storage
  - **Bad**: low recognition rates, susceptible to normalization, lazy, memory consumption
  - Big Data:
    * Distances can be a problem, but if saving a lot of data is not an issue it can be quite good
- **Naive Bayes: Modelling by Probability**
  - **Approach**: learn how data is distributed probabilistically and use it to infer context
  - **Bayes Theorem:** $posterior = \frac{prior * liklihood}{evidence}$
  - For Classification:
    * Formula: $p(C|F_1,..,F_n) = \frac{p(C)p(F_1,...,F_n|C)}{p(F_1,..,F_n)}$
    * $\hat{c} = argmax_{k \in 1,..,K} p(C_k) \prod_{i=1}^{n} p(F_i|C_k)$
  - **Nomogram**: visualizing Naive Bayes - add logarithmic values
  - **Good**: good results and low computational complexity
  - **But**: assumption of uncorrelated data and oversimplified distributions
- **Decision Tree:**
  - **Approach:** model data as tree such that nodes are decisions and leaves are labels
    * Minimize statistical measures like Entropy or GINI and choose Split-Attribute with best value
    * Repeat until datapoints are correctly classified
  - Formulas:
    * $Entropy(S) = -p_1 log_2(p1) - p2 * log_2(p_2)$ level of uncertainty
    * $Gain(S,A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_V|}{|S|} Entropy(S_V)$

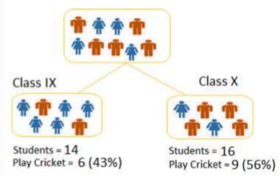
- **Node impurity metrics for Decision Trees**



- **Example for calculations in Decision Trees:**

Gini

Split on Class



Class IX = $(0.43)^2 + (0.57)^2 = 0.51$
Class X = $(0.56)^2 + (0.44)^2 = 0.51$
Weighted Gini for Split Class =
$(14/30)*0.51 + (16/30)*0.51 = \mathbf{0.51}$

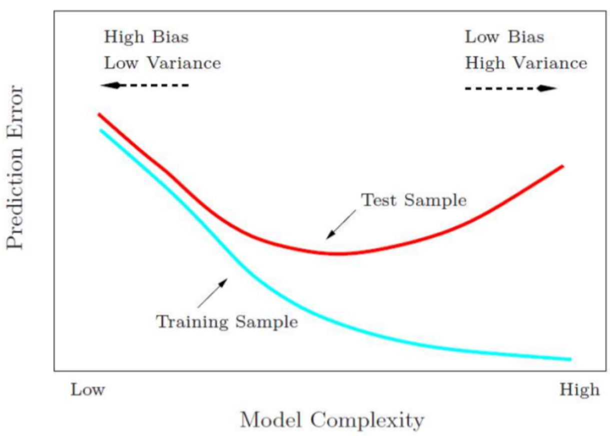
Entropy

		Play Golf		
		Yes	No	
Outlook	Sunny	3	2	5
	Overcast	4	0	4
	Rainy	2	3	5
				14

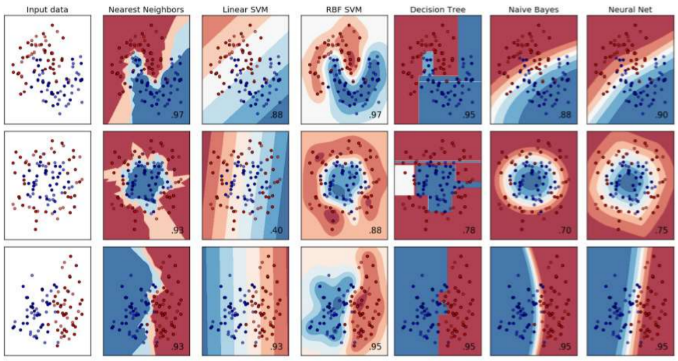
E(PlayGolf, Outlook) = **P**(Sunny)***E**(3,2) + **P**(Overcast)***E**(4,0) + **P**(Rainy)***E**(2,3)
= (5/14)*0.971 + (4/14)*0.0 + (5/14)*0.971
= 0.693

G(PlayGolf, Outlook) = **E**(PlayGolf) – **E**(PlayGolf, Outlook)
= 0.940 – 0.693 = 0.247

- **Extensions:**
  - Dealing with numeric attributes:
    - discretise using a comparison: $<, >, etc.$
    - Computational expensive but: evaluate all possibilities and choose split with max Gain
  - Further optimizations:
    - Random uniform subsampling and dynamic programming for reuse of results
  - Reduced error pruning:
    - Replace leave nodes with most popular class
    - Leave if classification is not affected
  - Cost complexity pruning:
    - Remove subtree $prunt(T, t)$ that minimizes: $\frac{err(prune(T,t),S)-err(T,S)}{|leaves(T)|-|leaves(prune(T,t))|}$
- **Bias-Variance-Tradeoff: Model Complexity**



- **Support Vector Machines:**
  - **Approach**: create hyperplanes which separate data with highest margin (convex constraint opt.)
  - Kernel trick: Increase dimensionality with transform $\rightarrow$ linear classification in non-linear space
  - **Good**: High accuracy and accuracy is not affected by high dimensions
  - **Bad**: prone to overfitting, choice of kernel and parameters critical
- **Artificial Neural Networks**
  - **Approach**: generate network of neurons (weights and thresholds) which minimizes a cost function
    1. Network with input for each feature and output for each class
    2. Network for each class with inputs and binary output for class
  - **Neuron:** aggregation of weighted inputs with non-linear activation function
  - **Training:** Backpropagation $\delta w_i = -\eta \frac{\partial E(w_i)}{w_i}$
  - **Bad**: only broad heuristics for choosing the number of neurons, learnable through backpropagation
- **Classifier Overview:**



- **Dimensionality Reduction:**
  - Autoencoder
    * **Idea:** Input and Output-Layer have same size, hidden Layer has fewer neurons
    * Lower dimensional representation through minimizing error between input and output
  - Principal Component Analysis
    * Highly covariant / correlated features deliver redundant information
    * **Approach:**
      · Analyze covariance (correlation) matrix
      · Transform data into principle components (find eigenvectors and eigenvalues)
      · Ignore components with low weightings
    * **Good**: removes covariance from features and reduces features and computation
    * **Bad**: does not indicate value of a feature, all features must be calculated for transformation
- **How to find the right features:**
  - Related work / empiric results / heuristics
  - Brute force: try them all out and every combination
  - Optimized: Feature selection algorithms
- **Machine Learning:**
  - Unsupervised: PCA, Clustering, Autoencoder $\rightarrow$ possible for feature learning
  - Supervised: mapping inputs to outputs (classification / regression)
  - Semi-Supervised: not every datapoint is labeled

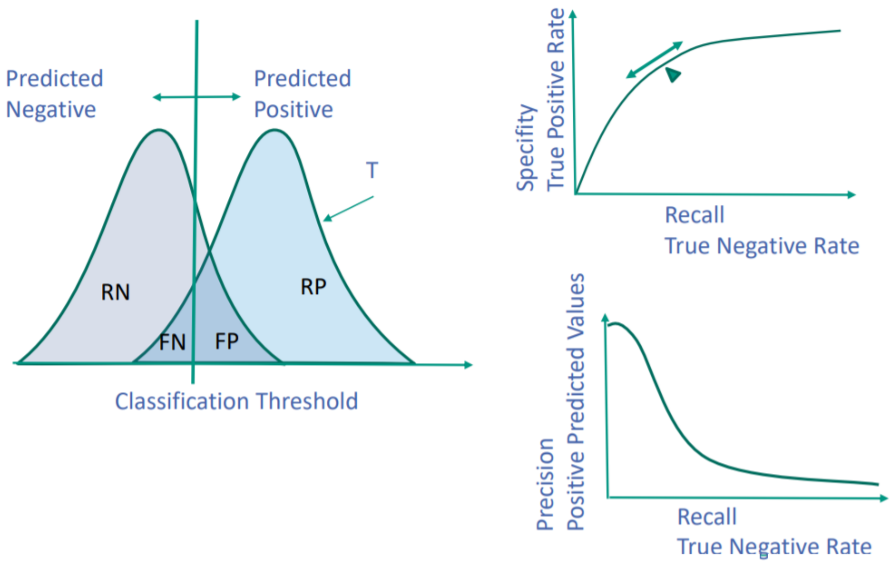
3 Classification Error

- **Evaluating Classifiers: Training and Testing**
  - **Goal**: Predict realistic error of classifier
  - **But**: labels need to be known (supervised learning)
  - Error Rate:
    * Positives: instance is predicted correctly
    * Negatives: instance is predicted incorrectly
    * Accuracy: proportion of correct classifications over whole set for testing
- **Resubstitution Error Rate: Performance Test on Train-Data**
  - Indicates only if code works and shows if function can be approximated
    * does not estimate effect of unseen data
    * not a good indicator for performance on future data
  - Shows effect of model complexity but **error rate is not always 0%**
    * Problems with data quality, statistical assumptions of the algorithm
    * Features do not result in a clear discriminate classification
  - **Solution**: Train-Test-Split
    1. Number of correct classifications: train error rate → lower is better
    2. Predictive Accuracy Evaluation: test error rate → lower is better
- **Training and Test-Set**
  - **Idea:** independent samples that played no part in formation of testing rules
  - **Assumption**: both samples are representative for underlying problem
  - Sets may differ in context parameters: e.g. data from different countries
  - **Operation Stages:**
    1. build basic structure
    2. optimize parameter settings, use (N:N) re-substitution but without test-data
  - **Typically**: Train, Validation and Test data
- **Train-Test: Size comparison:**
  - **Training**: larger set → better classifier
    * **But**: error rat of Resubstitution (N;N) **only** tells if algorithm is applicable
  - **Test**: larger set → more accurate error estimate
  - **Dilemma**: how to choose the sizes if data is limited
    * Holdout-method: one third for testing and rest for training
    * Can be applied in repeated fashion and also include a validation set
    * Once competed, all data can be used to build the final classifier

- **Confusion Matrix:**

Prediction / Ground Truth	Prediction Positive	Prediction Negative	F_1-Score 2 * (PPV*R) / (PPV+R)
Condition Positive (P)	True Positive (TP)	False Negative (FN)	Recall (R) TP / P
Condition Negative (N)	False Positive (FP)	True Negative (TN)	Specifity TN / N
Prevalence P / (P + N)	Precision (PPV) TP / (TP + FP)	NPV TN / (TN + FN)	**Accuracy (TP+TN)/ (PC+NC)**

- **Receiver Operating Curve**



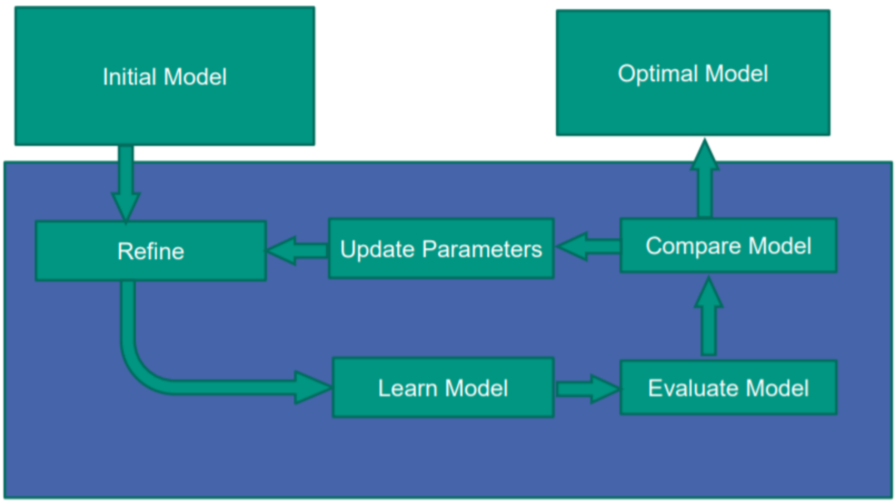
- **Singular metrics: better**
  - F-Score: harmonic mean between precision and recall
  - G-Score: geometric mean of precision and recall
  - Cohens kappa: observed accuracy vs. expected accuracy
- **Repeated Holdout:**
  - More reliable by repeating the process with sub-samples
    1. Each iteration selects proportion for random training, rest for testing
    2. Different error rates are averaged to get the overall error
  - **Problem**: still no optimum, since the test sets overlap
  - **Solution:** Cross-Validation (best 10-Fold CV)
    1. Split data into x subsets of equal size
    2. Use each subset in turn for testing and rest for training
    3. Error Rates are averaged to get the overall error

- **Leave-One-out CV (N-1;1)**
  - **Idea:**
    - Divide data into set of m subsamples of equal size
    - Use one sample for testing and train on remaining (m-1)-samples
    - This means the number of folds is equal to number of training instances
  - **Computation**: build n classifiers for n instances
  - **Error rate** = successful predictions / n
  - **Good:** best use of data, no random subsampling,
  - **Bad:** no stratification, expensive, assumption of statistical independence
  - **Worst-Case-Example:** 2 classes with uniform distribution, prediction = Mode, Accuracy = 0
- **Dealing with statistical independence: e.g. time series**
  - **Solution**: leave complete subject ot of validation → measure inter-subject performance
  - <u>Often better represents real use case:</u>
    - Learn classifier data from many users and apply for unknown user
- **Bootstrap**: probability 0.632
  - **Idea**: better estimate for accuracy in small samples
    - Sample of N with N-times resampling as training-set
    - Rest forms test set with probability (1-1/n) = 0.368
  - Correction of test error: $err = 0.632 * err_{Test} + 0.368 err_{Train}$
  - **Problem**: independence of both sets can be hard
- **Classifier Selection: No Free Lunch**
  - ML is based on search and optimization of models and model parameters
  - <u>No Free Lunch Theorem:</u>
    - for certain types of mathematical problems the cost of finding a solution
    - averaged over all problems is the same for any solution method
    - → no single best algorithm for all data
  - But **which** learning **scheme performs better**? → 10-Fold CV for comparison
  - **Model selection criterion**: MDL / Ocams-Razor → smallest theory that describes all facts
    - Space required to describe theory + space required for mistakes
- **Improvements on CV:**
  - Sample multiple times and check if mean accuracy for scheme "A" is better than "B"
  - **In Practice:** limited data and limited estimates for computing the mean
    - paired- t-test: samples are paired and same CV is applied twice
    - one-tailed binomial test: model accuracy better than no information
    - Mcnemar-Test: check if model is biased, distribution same as in data
- **Classification with costs:**
  - **Idea**: only predict high-cost class when very confident about prediction
    - Expected cost: dot product of vector of class probabilities and appropriate cost column
    - Choose class that minimizes expected cost
  - Cost-sensitive learning: resampling of instances / weighting of instances accordingly
  - ML-Model may take costs into account like naive bayes

- **Ensemble Learning: Combining multiple models**
  - **Idea**: build different simple experts and let the decide together
  - **Good**: often improves predictive performance
  - **Bad**: usually produces output that is hard to analyze (but there are solutions to some extent)
- **Bagging:**
  - Combine predictions by voting/averaging: reduces variance and usually more classifiers are better
  - Idealized version
    1. Build classifier for n splitted training sets
    2. Combine classifier predictions
  - Learning scheme is unstable → small changes in data can have big impact
    - Randomization: use random parameters or initial weights to build different classifiers
  - **Using cost**: use confidence as weighting
  - **Forest vs. SVM**: same solutions can be obtained
- **Boosting:**
  - Models are influenced by performance of previously built ones (also uses voting)
  - Encourages new model to become expert for instances previously misclassified
  - **Intuition:** combine weak learners to strong one
  - **Forest vs. Adaboost:** similar but booster would perfectly separate if possible
- **Stacking: Stacked generalization**
  - Combine predictions of base learners in meta learner (no voting)
  - Usually different schemes are combined as input for meta learner
    - CV-like: cant use predictions on training data to generate data for level 1 model
  - Hard to analyze but can be used as initialization for automatic model selection

4 Optimizing Classification

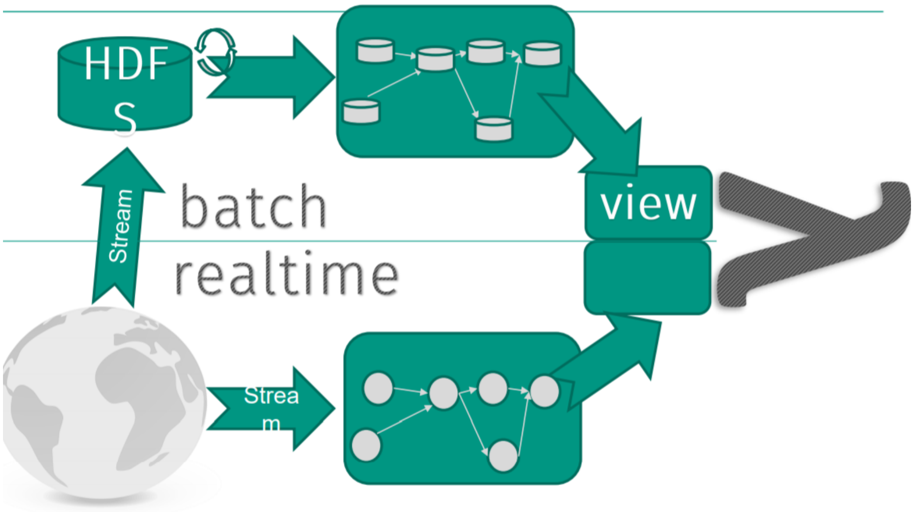
- **Feature Selection:** Find best classifier by turning of features
  - Metaheuristics:
    - Optimization-like techniques for feature subsets (sequentially turning on/off)
  - Filters:
    - **Pipeline**: all features → subset → model → performance
    - Rank features using mutual information, correlation, similarity, etc.
    - **Tradeoff** between: relevancy and redundancy: E.g. Minimum-Redundancy-Maximum-Relevance
    - Select variables regardless of the model
    - E.g. eliminate correlated features or least significant principle components
  - Wrappers:
    - **Pipeline**: all features → subset ⇄ model → performance
    - Evaluate subset of features using a learning a
  - Embedded Methods:
    - **Pipeline:** all features → subset → model + performance
    - Coupled with ML: e.g. recursive feature elimination of SVM (remove low weights)
    - **Other methods:** L1 regularization (LASSO) or decision tree pruning
    - Computational between Wrappers and Filters
- **General Requirements for Optimization Methods:**
  - Requirements: search method / blackbox but no gradient usage
  - Should include: flexible search space definition and scalable search (parallelize)
  - Maybe also: constraints
  - Blackbox-Optimization:

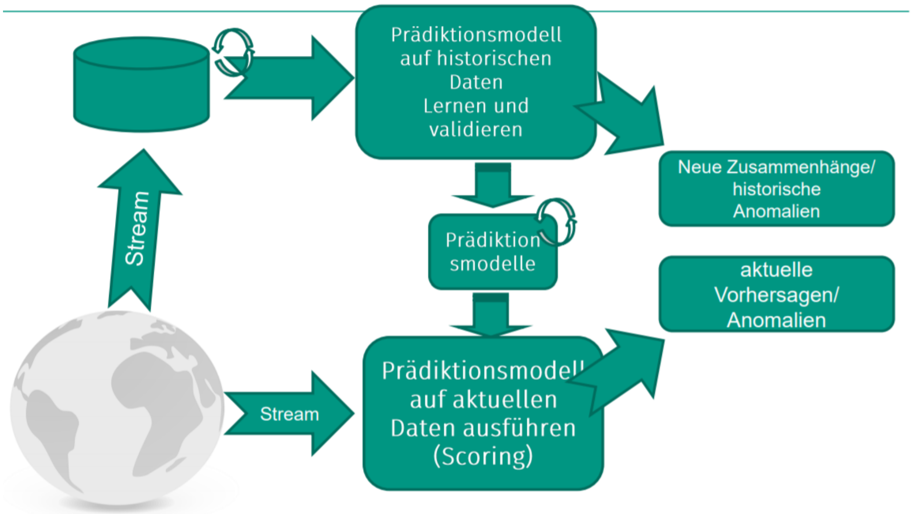


- **Bayesian Optimisation**
  - **Idea**: use when function evaluation is expensive and gradient is not available
    * Surrogate Model: approximate the Process
    * Acquisition Function: is maximized balancing of exploitation (obj.) and exploration (surrogate)
  - Choice of Surrogate Model:
    * Many dimensions: Random Forest
    * Categorical / hierarchical parameters: Gaussian process model with specialized kernel
    * Otherwise: Gaussian process model with standard kernel
  - Choice of Acquisition Function:
    * Portfolio viable: use a portfolio of acquisition functions
    * Absolute Min/Max known: MPI
    * Otherwise: GP-LCB, EI, etc.
  - **Good**: fully leveraged by choice of surrogate, data efficiency
  - **Bad**: inherently sequential, output vairabil
- **Evolutionary Algorithms:**
  - Initial Population: create initial population of random individuals
  - Iteratively:
    1. Evaluation: compute objective values of solution candidates
    2. Fitness Assignment: use objective values to determine fitness values (ACC, R^2, etc.)
    3. Selection: Select fittest individuals for reproduction (question about diversity)
    4. Reproduction: create new individuals form mating pool by crossover and mutation
  - **TPOT-System:** Input Data + Cleaning but rest is done via evolutionary algorithms
  - Neuro Evolution:
    * Evolving instead of standard ML-Pipelines → built upon homogenious structures (Neurons)
    * Good crossover-strategies exist but search space size increases gradually
    * Small and thus complex models are preferred
    * **Good**: prior knowledge initial population, parallelisation, output is variable
    * **Bad**: data efficiency is bad due to much randomness
- **Reinforcement-Learning:**
  - Agent tries to maximize his reward by taking actions which influence enviornment
  - Usually modelled using MDPS:
    1. Controller samples architecture A with probability p
    2. Train child network with architecture A to get accuracy R
    3. Compute gradient of p and scale it by R to update controller
  - **Good**: parallel, data efficient but converges slow, output is variable (e.g. sequence models)
  - **Bad**: prior knowledge cannot be used
- **Ensemble Learning & AutoML**
  - Classifiers can not only be used as surrogate function but also to find good initializations
  - **Initializing**: find best model
    * Build architecture similar to stacking
    * Classifier that predicts how a classifier will perform and select good ones
    * Maybe compare datasets: distribution, etc. and use it to predict accuracy of a classifier
    * Use bag of best classifiers and construct ensemble → AutoML (Meta-Learning)

5 Big Data: Batch Processing

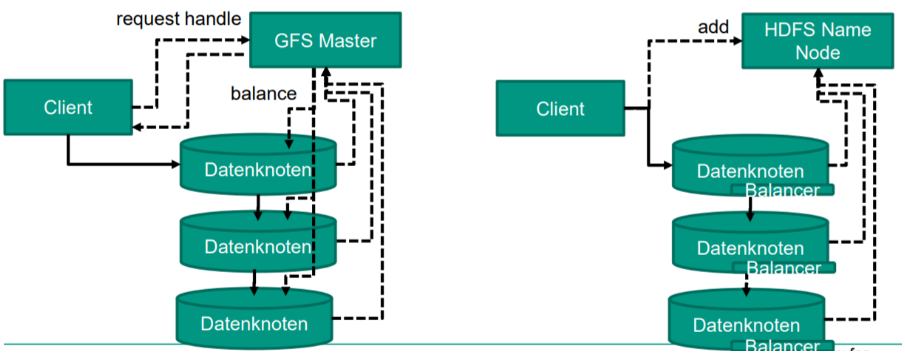
- **Lambda-Architectur**
  - **Idea**: ML-Task cannot be started everytime from scratch → precalculation
    * Query is split into realtime and batch tasks (e.g. CSS = offline learning + Realtime Classification)
    * Batch-Layer: Hadoop or Spark (Answer bas)
    * Realtime-Processing: Storm or Spark





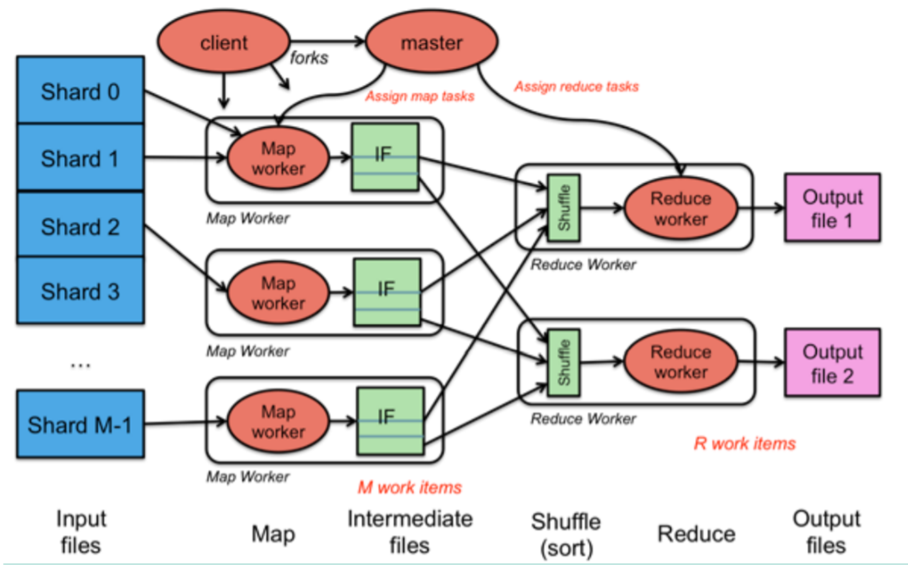
- **Realtime data processing: requirement for businesses**
  - Context-Sensitive Application: involves influence on decisions and latency / efficiency is important
  - **Business Requirement:** reduce processing time and design strategy = zero-latency enterprise
  - E.g. Context-aware online advertising, real-time search, high frequency trading, social networks
  - <u>Stream of events that flow into system at given time</u>
    * Complex Event Systems: typical "if than rules" and now use classifier every input
    * E.g. Event-Pattern detection, event filtering, event aggregation and transformation
    * Scoring: Prediction Model Markupt Language (PMML), trained model is used in realtime system

- **PMML Support:**
  - SQL-Server: Decision Trees / Clustering Models (Distribution-based)
  - Apache Spark: Clustering, K-Means, Regression Models (Linear, Ridge, Binary, Logistic), SVM
  - PMML package: in R with lots of models and support for transformation
  - **PMML Alternatives:** serialize as native structure (pickle RDS) and use same framework (sklearn)
  - PFA (Portable Format for Analytics): successor to PMML
- **Hadoop: New way of batch processing**
  - Software platform that lets one write and process vast amounts of data:
    * MapReduce: offline computing engine
    * HDFS: Hadoop distributed file system
    * HBase: online data acess
  - Why is it usefull:
    * **Scalable**: reliable store and process petabytes
    * **Economical**: distributes data and processing across clusters
    * **Efficient**: distributing data and parallel processing on nodes where data is located
    * **Reliable**: automatically maintains multiple copies and redeploys data if failure
  - Assumptions: written with large clusters in mind
    * Batch-Processing: emphasis on throughput opposed to low latency
    * Large datasets: gigabyte to terabytes in HDFS
    * Model: write-once-read-many access
    * Moving: computation cheaper than moving data
- **Hadoop: What does it do?**
  - Implements Googles MapReduce using HDFS which divides applications into many small blocks
  - HDFS creates multiple replicas of data blocks for reliability
  - MapReduce processes data where it is located (target Cluster with order of 10.000 nodes)



- **HDFS: Hadoop Distributed File System**
  - File system designed to run on commodity hardware → similarities with existing systems
  - **However**: differences to other distributed file systems are significant
    * Highly fault-tolerant and designed to be deployed on low-cost hardware
    * Provides high throughput access to application data and suitable for applications with large data
    * relaxes POSIX requirements to enable streaming access to file system data
    * Part of Apache Hadoop Core project

- **MapReduce Paradigm:**
  - Intended as internal search/indexing application but now extensively used by more organizations
  - Functional style programming with natural parallel capabilities across large clusters
  - Map and Reduce are **user** written functions
  - Underlying system takes care of:
    1. Partitioning input data and scheduling execution across several machines
    2. Handling machine failures and managing inter-machine communication (success key)
  - Map-Phase: Datasets assigned to task tracker
    * Run time partitions input and provides it to different Map-Instances
    * Map $(key, value) \rightarrow (key^*, value^*)$
    * Data functional operation will be performed emitting mapped key and value pairs (e.g. processing)
  - Reduce phase:
    * Run time collects $(key^*, value^*)$ pairs and distributes them to several Reduce function
    * Each Reduce function gets pairs with the same key^* and produces single file output
    * Master node collects answers to all subproblems and combines them in some way
    * Forms answer to original problem (e.g. data collection and digesting, merging)
- **5 common steps of parallel computing:**
  1. Prepare Map() input: row-wise and emit key value pairs per row $\rightarrow$ Map input: $list(k1, v1)$
  2. Run user-provided Map() code: $\rightarrow$ Map output: $list(k2, v2)$
  3. Shuffle Map output to Reduce processors, also group similar keys and input to same reducer
  4. Run user-provided Reduce() code: custom reducer designed by dev and emit key and value
     - Reduce input: $(k2, list(v2))$
     - Reduce output: $(k3, v3)$
  5. Produce final output: master node collects all reducer output and combines them in one file

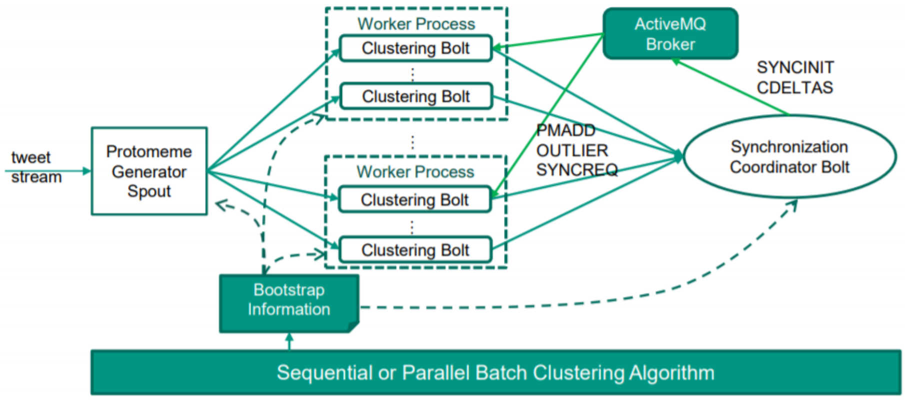


- **R-tools: using Hadoop**
  - **RHadoop**: Datanalytics with Hadoop via R functions
  - **rhdfs**: R package providing Hadoop HDFS access to R, managing distributed files
  - **rmr**: MapReduce interface for R

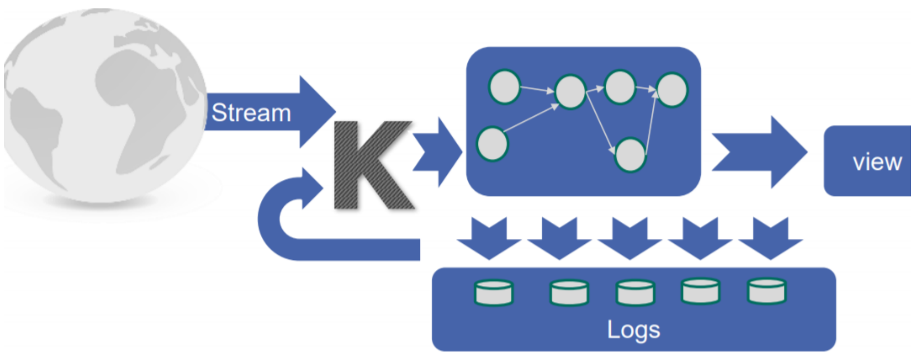
- **Challenges in Classification:**
  - Models work on single in memory model: parallelization issue but only in learning phase
  - High dimensional data: hadoop typically works by sorting in one dimension → transformation needed
  - Hadoop build for batch processing:
    * Realtime answer in classification phase can be a problem (e.g. waiting for one node)
    * Mostly used for learning and score Models on different platform
    * **Alternative**: streaming tricks → reduce chunk size / use custom scheduler
- **Comparison of different MapReduce Systems:**
  - Classical hadoop: approach similar to read and write csv
    * Data is sorted and location of data is known
    * Good for big amounts of data but today in-memory is more efficient
  - Spark: better utilizing in-memory
  - Flink: allows for iterations until some state is reached
    * Supports different workloads: stream, graph, iterative
- **KNN with Map-Reduce:**
  - For amount of data approaching infinity → error rate < 2* Bayes error rate
  - Naive version: computation intensive for large sets → exact solution through linear scan or tree structure
  - Scalable NN-Algorithm: e.g. through dimension reduction PCA
  - <u>Locality Sensitive Hashing:</u>
    * Hash-Function to map Datapoints that are near to each other
    * Works in principle as dimension reduction
    * Does not need to work perfect → for big k not all points need to be considered
    * Typically Tradeoff between: accuracy and speed
- **Boosting with Map-Reduce:**
  - Map-task: run boosting on its data and return sorted list of weak classifiers
  - Reduce-Task: generate merged classifier and find its weight
  - Key: use features as key and compute them on one node → good for sparse data
  - XGBoost: Supported in many frameworks and almost linear in number of processors (trees)

6 Realtime & Apps

- **Microbatching vs. Streaming:**
  - Every stream can be represented as microbatch → small chunks of data
  - **But**: keep data in-memory and avoid saving and reading from drive
  - Why Real Time Stream Processing:
    * Streams on Batch-System → often plagued by latencies
    * Processing system must keep up with event rate → load shedding
    * MapReduce / Hadoop store and process at scale but not for realtime systems → no hack viable
      · Latency can be reduced to get similar effects
      · But set of requirements differs fundamentally from batch processing
- **Storm Concepts:**
  - **Streams**: unbounded sequence of tuples
  - **Spout**: Source of Stream, e.g. read from Twitter streaming API
    * Tuple Tree → Spout tuple not fully processed until all tuples in tree completed
    * If not completed within specified timeout → tuple is replayed
    * Reliability API → Guaranteeing Message Processing
  - **Bolts**: Processes input streams and produces new streams, e.g. Functions, Filters, Aggregations
  - **Topologies**: Network of spouts and bolts



- **Kappa-Architectures**
  - Compared to Lambda-Architecture only streams and Log Files → no separation
  - **Idea:** buffered data-streams from log files for analysis through Kafka



- **Why frameworks:**
  - Multiple use cases: processing, computation, etc.
  - Data types, size. velocity: scalability
  - Mission critical data: fault-tolerance
  - Time series / pattern analysis: reliability
- **Context adaption examples:**
  - **Peer service:** take advantage of room projector for presentation
  - **Location semantics:** disable phone ringtone in quiet places
  - **Internals state:** decrease playback quality when battery power is low
  - **User task:** show parking spots / gas stations when driving
  - **Environmental conditions:** detailed indications when visibility is low
- **Problem**: Mindset Mismatch
  - Software systems today are produced according to a manufacturing model
  - **Mindset**: finished product is constructed and shipped and should act like any other machine
    → reliable but oblivious to surrounding
  - Paradigmatic Shortcomings: traditional if / else
    * **Software rigidness**: variability points are hard-coded in architecture, difficult adding new ones
    * **Lack of modularity**: tight coupling of business and infrastructural code → difficult maintenance
    * Mindset Mismatch: tools oblivious to context in which application runs → not adaptable software
    * Can grow quite complex and same goes for software engineering pattern like strategies
- **Context-aware programming languages:**
  - **Hypothesis**: lack of adaptability due to unavailability of context-aware programming languages
  - **Solutions? AOP, FOP or COP** e.g. AOP with different aspects embedded in code
  - Context Oriented Programming (COP):
    * Have different Layers and selectively turn them on or off → Behavioural variation
    * Instead of If-Else Structure: behaviour depends simply on context
    * Design Concepts:
      · **Context group:** collection of environmental situations sharing same characteristics
      · **Context**: represents single environmental situation (connected / not-connected)
  - Other approaches:
    * Event Condition Action: trigger event if condition is matched and activate action (big and complex)
    * Rapid-Prototyping: works with lots of visual tools and similar to Layering-Approach
    * Logic Programming: Precondition (temp.) → Postcondition(modify temp) ∧ HTTP
- **Variability in Live Contexts: Outlook**
  - Search for system that is: broad usable, adaptable and extensible, modular
  - Up till now: no groundbreaking tools to get context-awareness available in systems

7 Multi-Sensor Activity Context Detection for Wearable Computing

- **Paper overview:**
  - Wearable computing applications **central part** of user context is **human activity**
  - It should be **automatically acquired** through **sensors** to avoid annoyance
  - Activity is measured when it occurs through **sensors all over the body**
  - This Paper deals with the these sensors → placement & extraction of data
- **System Architecture:**
  - Acquisition system is modular designed and provides communication to host system
  - Acceleration sensors are mounted on small boards which are wired → wireless connects PC
  - Recognition Algorithm: Bayes Classification
    * Naive Bayes classifier: $p(a|x) = \frac{p(a)}{p(x)} \prod_{i=1}^{n} p(x_i|a)$
    * Features: running mean and variance computed over window of 50 sampels
- **Experimental Setup:**
  - Goal: recognize everyday postures and activities: sitting, standing, walking, writing, hand shaking
  - Sensor Placement and Number: all major joints (12 sensors)
- **Results and Discussion**
  - Results get better the more sensors are used → activities with more sensors are better detected
  - Physical activity is central for context-aware and user-centred applications
  - Platform in this paper demonstrates context extraction using acceleration sensors
  - Still work to do: sensors for more complex activities and inference with only acceleration sensors

8 Reducing the Dimensionality of Data with Neural Networks

- **Paper overview:**
  - High-dimensional data can be converted to low-dimensional by MLP
  - Idea: small central layer to reconstruct high-dimensional input vectors
  - Fine-Tuning through gradient-descent and starting with good initial weights
  - Can work much better than PCA for reduction of data
- **Dimensionality Reduction:**
  - Facilitates classification, visualization and communication of high-dimensional data
  - Most common used is PCA → find directions with greatest variance
  - **Here**: Non-linear generalization of PCA with adaptive multilayer encoder Network
  - Algorithm:
    * Training a multi-layer auto-encoder can be problematic → Pre-Training to get good weights
    * Restricted-Boltzmann machine can be used for training → finding good configurations
    * After pretraining multiple layers of feature detectors, the model is "unfolded"