Summary Machine Learning for the Quantified Self

Phillip Lippe

June 26, 2019

Contents

1 Introduction

1.1 Definitions

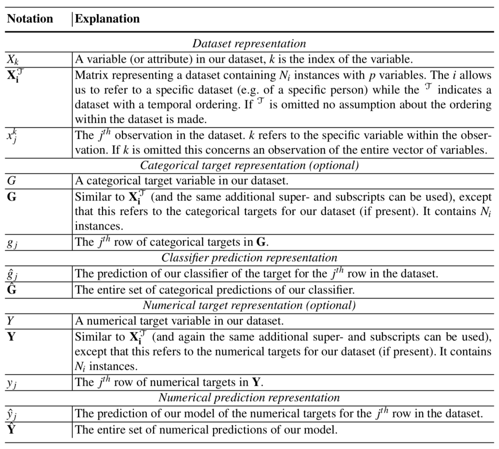
- The quantified self can be defined as:

  > *The quantified self is any individual engaged in the self-tracking of any kind of biological, physical, behavioral, or environmental information.*
  > *The self-tracking is driven by a certain goal of the individual with a desire to act upon the collected information.*

- **Augemberg** (2012): there are various types of measurements, including:
  - Physical activities (movement by accelerometer, steps, etc.)
  - Diet (calories consumed, fat, protein, etc.)
  - Psychological states (mood, emotions, depression, etc.)
  - Mental and cognitive traits (IQ, reaction, memory, etc.)
  - Environmental (location, weather, etc.)
  - Situational (time of the day, context, etc.)
  - Social variables (influence, role in a group, etc.)
- **Choe** (2014): distinguish quantified selves into three categories based on their goal.
  - Improved health (cure or manage a condition, execute a treatment plan, achieve a goal)
  - Improve other aspects of life (maximize work performance, be mindful)
  - Find new life experiences (have fun, learn new things)
- **Gimpel** (2013): identified five (non-exclusive) factors for quantified self motivation
  - Self-healing (to become healthier)
  - Self-discipline (to experience rewarding aspects of it)
  - Self-design (to control and optimize "yourself", as e.g. on sport)
  - Self-association (to be associated with the movement of QS)
  - Self-entertainment (to experience entertainment value)
- Machine Learning (automatically identifying patterns from data) is slightly different in the setting of Quantified Self because
  - we have to deal with sensory noise
  - there might be missing measurements
  - we have temporal data (feature engineering) with irregular time points
  - there can be an interaction with a user (advice for training/mood improvements, etc.), but we cannot try out every possibility
  - Learn across multiple datasets/users
- *Comment: Most of the above definitions need to be memorized for the exam*

1.2 Basic Terminology and Notation

- Measurement = one value for an attribute recorded at a specific time point
- Time series = series of measurements in temporal order
- Further notation:
  - For matrix $\boldsymbol{X}$, the columns are the different measurements like accelerometer, and rows are the time points (if dataset is temporally ordered, otherwise random list)



Notation	**Explanation**
	Dataset representation
X_k	A variable (or attribute) in our dataset, k is the index of the variable.
$\mathbf{X_i^{\mathcal{T}}}$	Matrix representing a dataset containing N_i instances with p variables. The i allows us to refer to a specific dataset (e.g. of a specific person) while the $^{\mathcal{T}}$ indicates a dataset with a temporal ordering. If $^{\mathcal{T}}$ is omitted no assumption about the ordering within the dataset is made.
x_j^k	The j^{th} observation in the dataset. k refers to the specific variable within the observation. If k is omitted this concerns an observation of the entire vector of variables.
	Categorical target representation (optional)
G	A categorical target variable in our dataset.
$\mathbf{G}$	Similar to $\mathbf{X_i^{\mathcal{T}}}$ (and the same additional super- and subscripts can be used), except that this refers to the categorical targets for our dataset (if present). It contains N_i instances.
g_j	The j^{th} row of categorical targets in $\mathbf{G}$.
	Classifier prediction representation
$\hat{g}_j$	The prediction of our classifier of the target for the j^{th} row in the dataset.
$\hat{\mathbf{G}}$	The entire set of categorical predictions of our classifier.
	Numerical target representation (optional)
Y	A numerical target variable in our dataset.
$\mathbf{Y}$	Similar to $\mathbf{X_i^{\mathcal{T}}}$ (and again the same additional super- and subscripts can be used), except that this refers to the numerical targets for our dataset (if present). It contains N_i instances.
y_j	The j^{th} row of numerical targets in $\mathbf{Y}$.
	Numerical prediction representation
$\hat{y}_j$	The prediction of our model of the numerical targets for the j^{th} row in the dataset.
$\hat{\mathbf{Y}}$	The entire set of numerical predictions of our model.

Figure 1: Overview of notation used in this course.

1.3 Basic overview of Sensory Data

- Different sensors available on mobile devices, such as:
  - *Accelerometer*: Measures the changes in forces upon the phone in the x-y-z plane
  - *Gyroscope*: Orientation of the phone compared to the earth's surface
  - *Magnetometer*: Measures x-y-z orientation compared to the earth's magnetic field
- Transforming raw data of time series require selecting a step size Δt, and combine sensory data over this interval. See Section 3 for techniques
- A large Δt gives (maybe too) coarse-grained data, but we have in the end a smaller dataset and lower standard deviation. The opposite is gained by a smaller Δt (fine-grained data, but large dataset and high stddev)

2 Handling Sensory Noise

2.1 Outlier Detection

- "*An outlier is an observation point that is distant from other observations*"
- Outliers can be caused by measurement errors, or variability of the data (e.g. very high heart rate due to pushing someone's limits)
- Outliers can be detected by either *domain knowledge* (known in what range to expect value, e.g. heart rate should not be over 220), or without by filtering noise. We distinguish between two ways for doing so:
  - *Distribution-based*: assume a certain distribution of the data, and remove all points with a likelihood lower than a certain threshold
  - *Distance-based*: focus on the distance between data points, and mark those as outliers which are far apart
- After detecting the outliers, we can replace them with *unknown* values/value missing tag.

2.1.1 Distribution-based outlier detection

- **Chauvenet's criterion**: assume a normal distribution for a single attribute
  - We can fit a normal distribution by calculating the mean and stddev of the data
  - For each point, calculate the probability $P(X \leq x_i^j) = \int_{-\infty}^{x_i^j} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(u-\mu)^2}{2\sigma^2}} \partial u$ (instance i from jth attribute)

  - A point is an outlier if:

$$P(X \leq x_i^j) < \frac{1}{c \cdot N} \quad \text{or} \quad \left(1 - P(X \leq x_i^j)\right) < \frac{1}{c \cdot N}$$

    Thus, the probability of a point being an outlier decreases with the number of observations for this attribute (as the likelihood increases to observe rare values)
  - Mostly $c = 2$ is chosen.

- **Mixture models**: assume the data to be described by K normal distributions $p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \sigma_k)$
  - Use EM algorithm to optimize maximum-likelihood of the data
  - A point is considered as an outlier if it has a lower probability than a certain threshold
  - Both threshold and number of distributions K is a hyperparameter to optimize

2.1.2 Distance-based outlier detection

- Outlier detection based on a distance metric $d(x_i^j, x_k^j)$ (as e.g. Euclidean distance) which can also be across multiple attributes
- **Simple distance-based approach**:
  - Call two points close if they are within a distance d_{min} (hyperparameter)
  - A point x_i^j is considered as an outlier if:

$$\frac{\sum_{n=1}^{N} \mathbb{1}\left(d(x_i^j, x_n^j) > d_{\text{min}}\right)}{N} > f_{\text{min}}$$

    Hence, we look if the number of points within the range of d_{min} are at least $1 - f_{\text{min}}$.
  - Example values of hyperparameters: $d_{\text{min}} = 0.1, f_{\text{min}} = 0.99$
  - Not working if we have multi-modal distribution (does not take local densities into account)
- **Local outlier factor**: use local densities to determine outliers.
  - Define k_{dist} for point x_i as the maximum distance in set of its k closest neighbors.
  - The reachability distance of x_i **from** x is defined as:

$$k_{\text{reach\_dist}}(x_i, x) = \max\left(k_{\text{dist}}(x), d(x, x_i)\right)$$

    Note that this distance is *not* symmetric as it uses the kth nearest neighbors of a point.
  - The local reachability distance of a point is defined by:

$$k_{\text{lrd}}(x_i) = \frac{|k_{\text{distnh}}(x_i)|}{\sum\limits_{x \in k_{\text{distnh}}(x_i)} k_{\text{reach\_dist}}(x_i, x)}$$

    Hence, it is high if a point is very close to others.
  - Outlier if neighbor points have much higher local reachability points than actual point:

$$k_{\text{lof}}(x_i) = \frac{\sum\limits_{x \in k_{\text{distnh}}(x_i)} k_{\text{lrd}}(x)}{|k_{\text{distnh}}(x_i)| \cdot k_{\text{lrd}}(x_i)}$$

2.2 Missing value imputation

- Due to outliers or measuring errors, we might have missing values in our dataset
- We can use simple methods like replace it by the mean or median of the other observed data, or also use more advanced methods that take the values of the other attributes at this observation into account, or a local time window.
  - Example for the latter: **interpolation** $x_i^j = x_{i-k}^j + k \cdot \frac{x_{i+l}^j - x_{i-k}^j}{l+k}$

2.2.1 Kalman Filter

- Combine outlier detection and imputation into a single model
- Therefore, we keep a latent state s_t, for which x_t are the observations in this states (Kalman filter relates x_t and s_t)
- The next value of a state is defined as:

$$s_t = F_t s_{t-1} + B_t u_t + w_t$$

  where u_t is a control input state (as e.g. sending a message), w_t is white noise, and F_t and B_t are learned matrices
- The measurements associated with s_t can be predicted by:

$$x_t = H_t s_t + v_t$$

  where v_t is again white noise.
- We can predict the next state (without noise) by $\hat{s}_{t|t-1} = F_t \hat{s}_{t-1|t-1} + B_t u_t$
- The error at time t compared to the observations x_t is then $e_t = x_t - H^T \hat{s}_{t|t-1}$
  - If we observe (after training/modeling) a high error, we can assume a value to be an outlier, and replace it with the prediction of the Kalman Filter.
- Given this error, we can update our prediction accordingly: $\hat{s}_{t|t} = \hat{s}_{t|t-1} + K_t e_t$ where K_t takes the expected prediction error into account (based on the white noise w_t and v_t)

2.3 Transforming the Data

- Transform data to extract most useful data, and get rid of remaining noise
- Different approaches can be used
- **Lowpass filter**: filter out high-frequent noise
  - We assume that our signal has a certain periodicity, but we are only interested in certain parts of the frequency band (noise is mostly very high-frequent)
  - The low-pass filter can remove those by weighting each periodicity by its frequency:

$$|G(f)|^2 = \frac{1}{1 + (f/f_c)^{2n}}$$

    with $|G(f)|$ as the magnitude, f_c is the cutoff frequency (magnitude halved), and n the order of the filter
- **Principal Component Analysis**: find components that explain most of the variance in the data
  - Select number of components based on the explained variance. Other, low-variance components are removed to reduce noise
  - Problem: we loose insight in the data because the components are not easily interpretable anymore

3 Feature Engineering

- Create useful features from temporal data

3.1 Time Domain

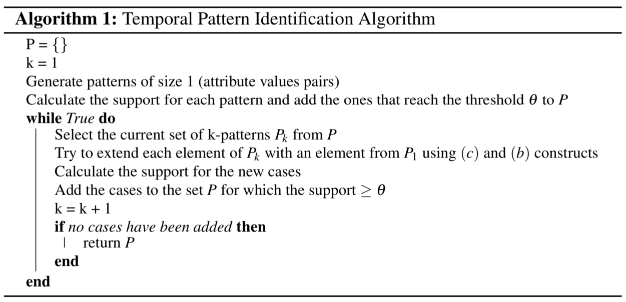
- Summarize values of a certain attributes in a window size of λ steps before. If we would take the time steps t and $t-1$ into account, our value for λ is 1
- Note that we cannot compute any values for the time steps $t = 1, .., \lambda$.

3.1.1 Numerical

- Aggregate values by mean, min, max, stddev, etc. (including current time step)
- We could also use coefficient between first and last value interpolation (gradient of attribute)

3.1.2 Categorical

- Generate patterns of occurrences of categorical values. We distinguish between successive `(b)` and co-occurring `(c)` actions/classes. Example patterns:
  - Activity level = high `(c)` Activity = running
  - Activity = running `(b)` Activity = running
- If we have a window size of λ, we just see if there is any time step within $t-\lambda, ..., t-1, t$ where the activities are co-occuring `(c)`, or there is one activity happening (arbitrary number of time steps) before another `(b)`.
- We can find important patterns by determining the support of such. This is important as the number of patterns exponentially increases with the number of categories/attributes, and only frequent patterns are interesting.
- The support of a pattern is defined as the proportion of the processed time steps at which this pattern would occur.
- The algorithm for finding such patterns starts with single attribute patterns, and extends only those which have sufficient support. In the end, we add all patterns (including the single-attribute) with enough support.



```
Algorithm 1: Temporal Pattern Identification Algorithm
P = {}
k = 1
Generate patterns of size 1 (attribute values pairs)
Calculate the support for each pattern and add the ones that reach the threshold $\theta$ to $P$
while True do
    Select the current set of k-patterns $P_k$ from $P$
    Try to extend each element of $P_k$ with an element from $P_1$ using $(c)$ and $(b)$ constructs
    Calculate the support for the new cases
    Add the cases to the set $P$ for which the support $\geq \theta$
    k = k + 1
    if no cases have been added then
        return $P$
    end
end
```

Figure 2: Pattern identification algorithm

3.1.3 Mixed data

- We can also combine numerical and categorical attributes for features
- Hereby, we create categories from numerical data by looking at them **qualitatively** (greetings from QR). For example, we can define ranges like low, medium and high, or look at the trend/gradients as *increasing* or *decreasing*
- Afterwards, we can apply the categorical approach on those

3.2 Frequency Domain

- Apply Fourier transformation on data within a window of λ (plus the current time point t) to extract periodicity of the data
- Assume a base frequency of $f_0 = \frac{2\pi}{\lambda+1}$ (or $f_0 = \frac{N_{\text{sec}}}{\lambda+1}$ in seconds) which is the lowest frequency with a complete sinusoid in it.
- We look at all the frequencies $\{0 \cdot f_0, 1 \cdot f_0, ..., \lambda \cdot f_0\}$ and determine the corresponding amplitudes
- Our features can be:
  - Frequency with highest amplitude

  - Frequency-weighted signal average $\frac{\sum_{k=0}^{\lambda} a_{t-\lambda}^t(k) \cdot f(k)}{\sum_{k=0}^{\lambda} a_{t-\lambda}^t(k)}$
  - *Power Spectrum Entropy*: Amount of information in the signal

$$x\_pse = -\sum_{k=0}^{\lambda} p_{t-\lambda}^t(k) \ln p_{t-\lambda}^t(k), \quad \text{with } \; p_{t-\lambda}^t(k) = \frac{|a_{t-\lambda}^t(k)|^2}{\sum_{i=0}^{\lambda} |a_{t-\lambda}^t(i)|^2}$$

3.3 Unstructured data

- How to handle non-temporal/unstructured data like text, audio, images, etc.
- Here we focus on text. The standard pipeline contains for steps:
  - *Tokenization*: split sentence into smallest parts
  - *Lower case*: put all words to lower case to have no difference in such
  - *Stemming*: reduce words to their stem to remove all small variations (like tense, etc.)
  - *Stop word removal*: remove known, uninformative stop words

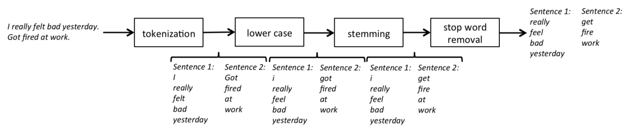

I really felt bad yesterday.
Got fired at work.
tokenization
lower case
stemming
stop word removal
Sentence 1: really feel bad yesterday
Sentence 2: get fire work
Sentence 1: I really felt bad yesterday
Sentence 2: Got fired at work
Sentence 1: i really feel bad yesterday
Sentence 2: got fired at work
Sentence 1: i really feel bad yesterday
Sentence 2: get fire at work

Figure 3: Pipeline of text processing

- Three approaches in general
  - **Bag of words**: count occurrences of n-gram within text. These counts are the features for the text
  - **TF-IDF**: BoW does not take "uniqueness" of word into account. Thus, TF-IDF takes occurrences of a word in a text and in the whole corpus into account
  - **Topic modeling**: assume that any text is a combination of k topics. Perform LDA to get these topics, and topic distribution of text is its features.

4 Clustering

- Using the features engineered before to cluster instances
- Two different learning setups
  - *Per instance*: cluster data points/instances of quantified selfs. If multiple quantified selfs are available, we concatenate the datasets.
  - *Per person*: cluster different quantified selfs. Here, we consider all recorded instances of a person as a single data point, and we compare datasets/persons and cluster them.

4.1 Distance Metrics

- We need different distance metrics per scenario
- We have to distinguish between feature-level and dataset-level

4.1.1 Feature-level distance metrics

- For *numerical* features, we can use the minkowski distance $\left(\sum_k \left|x_i^k - x_j^k\right|^q\right)^{1/q}$ which subsumes the Euclidean and Manhatten ($q = 1$). However, we need to consider the scaling of the features (assumed to be equal)
- For *categorical* features, we can use the Gower's similarity
  - For binary attributes (called *dichotomous*) $s(x_i^k, x_j^k) = 1$ if x_i^k and x_j^k are present (i.e. are 1), else 0
  - For categorical, we have $s(x_i^k, x_j^k) = \mathbb{1}(x_i^k = x_j^k)$
  - For numerical values in a range R, the Gower's similarity is $s(x_i^k, x_j^k) = 1 - \frac{|x_i^k - x_j^k|}{R}$
  - Similarity over multiple attributes is the mean of them
  - Note that this is a similarity and not a distance (correlated by Similarity $\sim$ 1/Distance)

4.1.2 Dataset-level distance metrics

- We have to distinguish between datasets with and without temporal ordering
- **Non-temporal personal level distance metrics**: three different approaches possible
  1. Summarize values per attribute over the entire dataset into a single number, as e.g. take the mean, min, max, stddev, etc. On these, we can use the same distance metrics as before
  2. Estimate parameters of a distribution that describes the dataset, such as a normal distribution with $\mathcal{N}(\mu, \sigma^2)$. On the parameters μ, σ^2 we can apply the same distance metrics as before
  3. Compare the distributions of values for an attribute with a statistical test, such as the Kolmogorov Smirnov test. The distance metric would be $1 - p$ where p is the p-value returned by the test.
- **Temporal personal level distance metrics**: again, three different approaches
  1. *Feature-based*: extract features from the two time series, such as those from Section 3 (time and frequency domain).
  2. *Model-based*: we try to fit a model on the two time series, and use those parameters to compare them. For example, we could use dynamical systems or similar
  3. *Raw-data based* uses a distance per point.
     - For example, it can assume a equal number of points in both datasets, and just takes e.g. the Euclidean Distance per time point
     - Alternatively, we can also take a possible lag into account (shifted dataset). Then we compute the cross correlation coefficient $ccc(\tau, x^l_{qs_i}, x^l_{qs_j}) = \sum_{k=-\infty}^{\infty} x^l_{k,qs_i} \cdot x^l_{k+\tau,qs_j}$.
     - Optimize τ by $\arg\min_\tau \sum_{k=1}^{p} \frac{1}{ccc\left(\tau, x^l_{qs_i}, x^l_{qs_j}\right)}$. Note that we have a single τ for all attributes
     - **Dynamic Time Warping**: make best pairs of instances in the sequence to find minimum distance. Allows different frequencies of activities
       * Two conditions for pairing:
         Monoticity condition: time order has to be preserved. We cannot go "back" in time
         Boundary condition: the first and last point must be aligned of the two time series
       * Algorithm similar to finding shortest path in a graph.

```
Algorithm 3: Dynamic Time Warping
dtw_distance($\mathbf{X}^{\mathcal{T}}_{\mathbf{qs_i}}, \mathbf{X}^{\mathcal{T}}_{\mathbf{qs_j}}$) :
for $k \in 1, \dots, N_{qs_i}$ do
    cheapest_path($k, 0$) $= \infty$
end
for $k \in 1, \dots, N_{qs_j}$ do
    cheapest_path($0, k$) $= \infty$
end
cheapest_path(0,0) = 0
for $k = 1, \dots, N_{qs_i}$ do
    for $l = 1, dots, N_{qs_j}$ do
        $d =$ distance($x_{k,qs_i}, x_{l,qs_j}$)
        cheapest_path $= d + min$({cheapest_path($k-1, l$), cheapest_path($k, l-$
        $1$), cheapest_path($k-1, l-1$)})
    end
end
return cheapest_path($N_{qs_i}$, $N_{qs_i}$)
```

Figure 4: Algorithm of dynamic time warping

       * The drawbacks of this methods are that it is computational expensive ($\mathcal{O}(N \cdot M)$), and that the distance metric might not always be the best fit (i.e. should it be allowed to align the first point of sequence 1 with the last point of sequence 2?)
       * *Comment: For this algorithm, it helps more to practice it several times instead of writing it down in all details.*

4.2 Clustering approaches

- Overview of different clustering approaches
- **K-means**: define k cluster means. A point is assigned to the cluster to which mean it is the closest. Means are updated by the assigned points.

  - Using Silhoutte score to select best k/determine whether a clustering is good:

$$\text{silhoutte} = \frac{\sum_{i=1}^{N} \frac{b(x_i)-a(x_i)}{\max(a(x_i),b(x_i))}}{N} \quad \text{where } a(x_i) = \frac{\sum_{\forall x_j \in C_l} d(x_i,x_j)}{|C_l|} \quad (x_i \in C_l)$$
$$\text{and} \quad b(x_i) = \min_{C_m \neq C_l} \frac{\sum_{\forall x_j \in C_m} d(x_i,x_j)}{|C_m|}$$

    In text, the silhoutte score compares the average distance of a point with others within a cluster ($a(x_i)$) with the distance of points with the next closest cluster ($b(x_i)$). The larger the score, the better (between$[-1,1]$)

- **K-medoids**: very similar to k-means, but we use actual points as cluster centers instead of artificial ones
  - Choose new cluster means as the point with the minimum distance to all other points in the cluster
  - More suitable if certain points in search space might not make sense
  - For example, k-medoids is known to work better for person-level clustering

4.2.1 Hierarchical clustering

- Perform clustering in an iterative approach
- **Divisive clustering**: start with one cluster with all points in it, and in each step, perform one split
  - Define the dissimilarity of a point to all other points in a cluster as the average distance

$$\text{dissimilarity}(x_i, C) = \frac{\sum_{x_j \neq x_i \in C} \text{distance}(x_i, x_j)}{|C|}$$

  - When creating a new cluster C', we add the most dissimilar points (in order of dissimilarity) until a point is more dissimilar to the points in C' than points in C.
  - If we have multiple clusters, we choose the cluster to split which has the greatest distance between any points in the cluster
- **Agglomerative clustering**: start with all points in separate clusters, and merge them step by step. Merge decision can be based on different criteria (equal to distance metric between clusters):
  - *Single linkage*: merge the two clusters with the minimum distance between any two points

$$d_{SL}(C_k, C_l) = \min_{x_i \in C_k, x_j \in C_l} \text{distance}(x_i, x_j)$$

  - *Complete linkage*: merge the two clusters where the maximum distance between any two points is minimal

$$d_{SL}(C_k, C_l) = \max_{x_i \in C_k, x_j \in C_l} \text{distance}(x_i, x_j)$$

  - *Group average*: merge the two clusters with the average distance between all points is minimal

$$d_{SL}(C_k, C_l) = \frac{\sum\limits_{x_i \in C_k, x_j \in C_l} \text{distance}(x_i, x_j)}{|C_k| \cdot |C_l|}$$

  - *Ward's criterion*: merge the two clusters where the increase of standard deviation by the combined cluster is minimal

$$d_{SL}(C_k, C_l) = \sigma^2_{C_k \cup C_l} - \left(\sigma^2_{C_l} + \sigma^2_{C_k}\right)$$

4.2.2 Subspace clustering

- The problem of standard clustering algorithms for a huge number of features is that the distance between two points get uninformative (small distance in all features compared to big difference in only one feature). Hence, the clusters get less meaningful as well
- Better approach is therefore to look at subspaces in the feature space.
- Pseudo algorithm:

1. For all features, define intervals of the feature space. This leads to units $u = u_1, ..., u_p$ where $u_i(l)$ is the lower-bound for attribute i in this unit, and $u_h(l)$ the upper bound respectively. Note that units do not require to cover all features, but can look at subspaces only (same to setting lower bound to $-\infty$ and upper to ∞).
2. Determine selectivity of a unit as proportion of points in them. We call a unit "dense" if it contains more points/higher proportion than a certain threshold (hyperparameter).
3. Units are connected to a cluster if they
   - share a common face which is defined as having the lower bound equals to an upper bound of another unit (or other way round), and the same upper and lower bound for all other attributes.
   - or when they share a unit to which they both have a common face.

- In the end, we strive to find dense units using a combination of attributes to form clusters
- To reduce number of attributes, we can start to make units with a single attribute, and add more attributes iteratively based on some *fancy* algorithm. After having found the units, we can create the clusters

5 Supervised Learning

- The perspective on supervised learning in this course is summarized in Figure 5

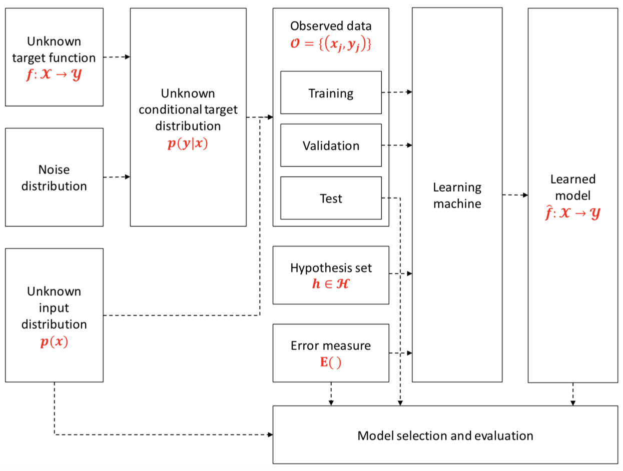

Unknown target function $f: \mathcal{X} \to \mathcal{Y}$
Noise distribution
Unknown input distribution $p(x)$
Unknown conditional target distribution $p(y|x)$
Observed data $\mathcal{O} = \{(x_i, y_i)\}$
Training
Validation
Test
Hypothesis set $h \in \mathcal{H}$
Error measure E( )
Learning machine
Learned model $\hat{f}: \mathcal{X} \to \mathcal{Y}$
Model selection and evaluation

Figure 5: Overview of supervised learning framework

- Discussion of error measuring
  - *Risk* $E(h, f)$ describes the distance between our hypothesis h and the target function f
  - *Loss* is the point-wise risk $e(h(x), f(x))$
  - Given the evidence $p(x)$, we can determine the risk by $E(h, f) = \int e(h(x), f(x))p(x)dx$
  - However, this integral can (usually) not be computed, and only approximated by Monte-Carlo integration.
  - For definitions of e, we can use metrics like F1 or accuracy (classification), or MSE etc. (regression)
  - The in-sample error is the average loss over all training points $E_{in}(h) = \frac{1}{N}\sum_{(x,y)\in\mathcal{O}_{\text{train}}} e(y, h(x))$
  - The out-of-sample error accordingly for points not in the training set:
    $E_{out}(h) = \int_{\mathcal{X}\setminus\mathcal{O}_{\text{train}}} e(h(x), f(x))p(x)dx$
- We select the model with the lowest in-sample error, but need to be careful with overfitting

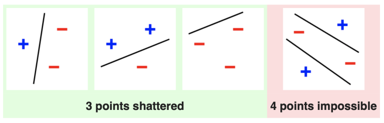
5.1 PAC Learnability and VC dimensionality

- "Probably approximately correct learning"
- A hypothesis set is PAC learnable when it can be shown that given any value of δ, ϵ there is an N (number of samples) where with probability $1 - \delta$ the difference between the in-sample and out-of-sample error is less than ϵ.

- *Probably*: $1 - \delta$, *Approximate correct*: $|E_{out}(\hat{f}) - E_{in}(\hat{f})| < \epsilon$
- For a finite set of M hypotheses, we determine it by:

$$E_{out}(\hat{f}) \leq E_{in}(\hat{f}) + \sqrt{\frac{1}{2N} \log \frac{2M}{\delta}}$$

  Hence, every finite set of hypotheses is PAC learnable, and we can calculate the expected error given number of samples N, hypothesis set size M, and probability δ
- For infinite set of hypotheses, we can look at VC dimensionality
  - We say that a set of input vectors X is shattered by a hypothesis set $\mathcal{H}$ if it can represent all possible labeling
  - The VC dimension of $\mathcal{H}$ is an X with the highest cardinality D. Note that not all possible point sets of cardinality D must be shattered by $\mathcal{H}$. It is sufficient if it is true for at least one.
  - Example for a perceptron:



    The VC dimension of a perceptron in 2 dimensions is 3, because there exists no set of 4 points that can be shattered (i.e. for which we can learn any labeling)
  - If the hypothesis set can represent any labeling for an arbitrary large set X, it has a VC dimension of ∞
  - Important finding: all hypothesis set with a finite VC-dimension are PAC learnable
  - We can
- Some implications from this study
  - Given a few training samples, it is easy to get a low in-sample error. But with increasing number of samples, the out-of-sample error decreases
  - In addition, for a fixed N, we can study the influence of more complex hypotheses and find the best compromise
  -

6 Predictive Modeling without Notion of Time

- Any predictor that does not explicitly take time into account (except the temporal features from time/frequency domain etc.)
- Different learning setups possible
  - *Individual*: train and test on a single user
  - *Population - unknown user*: train on a set of users, test on a different set of users
  - *Population - unseen data*: train on a set of users, test on the same users but different data

6.1 Preventing overfitting

- One big issue in the context of QS is that algorithms can easily overfit. This can be due to the big amount of features, the noise contained in them, and the usually small datasets we have
- **Feature selection**
  - To prevent the models to overfit on not useful features, we can reduce the number of features to the essential ones
  - *Forward selection*: start with empty set, and iteratively add most predictive feature. At every iteration, we need to run a model on the previously added features plus any of the other features left. Stop when accuracy does not improve significantly anymore
  - *Backward selection*: start with all features, and iteratively remove the least predictive feature. Similar to forward selection, just doing the whole algorithm reversed.
- **Regularization**: add a term to the error function to punish more for more complex models. Examples include L1/L2 regularization for NN, points per leaf for decision trees, etc.

7 Predictive Modeling with Notion of Time

7.1 Time Series

- Understanding the periodicity and trends in data in the time domain. Can be used for forecasting or control (e.g. how can we influence the trend)
- Is build up by three components:
  - *Seasonality/Periodicity*: any periodic/repeating pattern over any frequency (e.g. seconds, hours or days)
  - *Trend*: how the mean evolves over time
  - *Irregular variations*: noise, everything left after we remove periodicity and trend
- **Stationarity**: assumption/requirement for many algorithms applied on time series
  - In general, stationarity means that the statistical properties of a process generating a time series do not change over time.
  - We call a time series stationary if trends and periodicity are removed (mean is constant), and the variance of the remaining irregular variations is constant over time
  - Additionally, the lagged auto correlation should be constant over time/lags λ and close to 0:

$$r_\lambda = \frac{\sum_{t=1}^{N-\lambda}(x_t - \bar{x})(x_{t+\lambda} - \bar{x})}{\sum_{t=1}^{N}(x_t - \bar{x})^2}$$

    It can provides clues of underlying pattern in the data, which should not be the case for stationary ones.
  - If a time series is not stationary, we can mostly transform it to one by removing trend, periodicity, and try to stabilize the variance

7.1.1 Filtering and Smoothing

- To determine the trend of a signal, one of the simplest approaches is filtering and/or smoothing
- Simplest filtering is for a window size of $\pm q$ (creates a new, filtered time series):

$$z_t = \sum_{r=-q}^{q} a_r x_{t+r}$$

- **Differencing**
  - For removing a trend, the most effective technique is differencing (or gradient filter): $z_t = x_t - x_{t-1} = \nabla x_t$. As we expect the trend to be low-frequent, the gradient is therefore small.
  - We can also apply this operator multiple times, leading to a d-th order differencing ($\nabla^d x_t$). For $d = 2$, we get $z_t = \nabla^2 x_t = x_t - 2x_{t-1} + x_{t-2}$. A d-th order differencing can remove trends than can be approximated by a polynomial of order d or lower.
  - Drawback of differencing is that the variance of the remaining time series increases. Can be improved by using a better approximation of trend than x_{t-1}, as for example a exponential filtered signal $e_t = \sum_{r=-q}^{0} \frac{\alpha(1-\alpha)^{|r|}}{2-\alpha} x_{t+r}$ (with e.g. $\alpha = 0.05$), and use that for the differencing: $z_t = x_t - e_t$
  - Still, we have to be careful as we might remove low-frequent periodicity as well ($\partial \sin(0.1 \cdot x)/\partial x = 0.1 \cdot \cos(0.1 \cdot x) \Rightarrow$ dampen signal by factor of 10).
  - If we would want to remove periodicity, we could apply the differencing operator not on two adjacent points, but two points that are moved by 1 period as the difference between those is expected to be zero (note that we need to know/estimate the frequency for that)

7.1.2 ARIMA

- "Autoregressive Integrated Moving Average Model"
- We try to estimate a model that describes the empirical data well and can be used to forecast/predict new values
- For this, we learn/determine a mapping of time point t to probability distribution P_t
- In ARIMA, we assume P_t to be modeled by a combination of a autoregressive process (AR), and a moving average (MA) over white noise W_t:

  $$P_t = \underbrace{\phi_1 P_{t-1} + ...\phi_p P_{t-p} + W_t}_{\text{Autoregressive process}} + \underbrace{\theta_1 W_{t-1} + ...\theta_q W_{t-q}}_{\text{Moving Average}}$$

  Note that an AR can be expressed by an infinite MA, and the other way round. But to reduce the number of parameters, both concepts are used here.
- To remove the drifts of mean (trend), we apply differencing of order d on P_t ($V_t = \nabla^d P_t$).
- Optimization of parameters
  - For p we can look at the autocorrelation between x_t and x_{t-p} to find patterns
  - For other parameters, gridsearch with objective function as the fit to the data we have
- Note that ARIMA does not take seasonality/periodicity into account. Hence, we either have to add it externally, remove it beforehand or model it as well (ARIMAX)

7.2 Recurrent Neural Networks

- Unfolding for gradient calculation
- **Echo State Networks**: "cheap" RNNs without backprop through time
  - Three weight matrices:
    - $\boldsymbol{W}^{\textbf{in}}$: are the weights from the input layer to the memory (or here also called *reservoir*).
    - $\boldsymbol{W}$: are the weights over time steps (or internally in the reservoir).
    - $\boldsymbol{W}^{\textbf{out}}$: specify the weights from the reservoir to the output
  - $\boldsymbol{W}^{\textbf{in}}$ and $\boldsymbol{W}$ are randomly initialized and **fixed** during training ,while $\boldsymbol{W}^{\textbf{out}}$ is learned (either by SGD or pseudo inverse)
  - Initialization of $\boldsymbol{W}$ need to satisfy the *Echo State property* which state that the effect of a previous state should gradually decrease over time (prevent exploding values)
  - We can ensure this by randomly initializing a matrix, dividing by its spectral radius and (optionally) scale it down even further.
  - There are different initialization heuristics to optimize this process, but all underly the *No Free Lunch* theorem (optimizing for one use case will make it worse for others)

7.3 Dynamical Systems

- Build domain knowledge-based models that cover temporal relationships between attributes by the meas of differential equations. Furthermore, they assume a numerical state. Very simple model for velocity:

  $$y_{vel}(t) = y_{vel}(t-1) + \gamma_1 \cdot y_{acc}(t)$$
- Models still contain parameters (as γ_1 above) that can/need to be tuned
- **Parameter optimization**: three main approaches
  - *Simulated Annealing*: similar to an EA with a population size of 1.
    - We have a single solution which we randomly initialize first. At each iteration, we take a random step, and compare the difference in score.
    - If the new point is better than the old one, we replace it. Otherwise, we replace it with a probability based on the distance between the scores, and the number of steps we have already taken.

    * Note that the probability decreases with number of steps. Thus, we switch from exploration in the first iterations to exploitation in the last.
  - *Genetic Algorithms*:
    * We represent parameter values as a bit string (arbitrary number of bits per parameter, all together concatenated into one genotype), and initialize a couple of them in our population
    * At each iteration, we choose a set of parents from our population (based on fitness value), and perform crossover as well as mutation on the children
    * Perform survivor selection, or just completely replace the old generation by the new one
  - *NSGA-II*: multi-criteria optimization GA
    * "Non-Dominated Sorting Genetic Algorithm"
    * Used when multiple targets need to be optimized, and there is no fixed tradeoff between both
    * Therefore, we find Pareto fronts in our population (individuals that are not Pareto dominated by any other individual in our population)
    * We create several Pareto fronts by iteratively creating one for our population, and then remove all individuals on it from the population, and start again
    * Interested in a wide spread of individuals/coverage of Pareto front $\Rightarrow$ weight individuals on the Pareto front by the distance to other points (points on the border set to infinity because they are the best for a certain objective). We use this weight for survivor selection where we iteratively add individuals until we have enough for a new population. Note that we of course prioritize the individuals that are on a earlier Pareto front

8 Reinforcement Learning

- RL for ML4QS to learn from interactions with user and influencing him
- General overview of how to integrate RL in ML4QS is shown in Figure 6

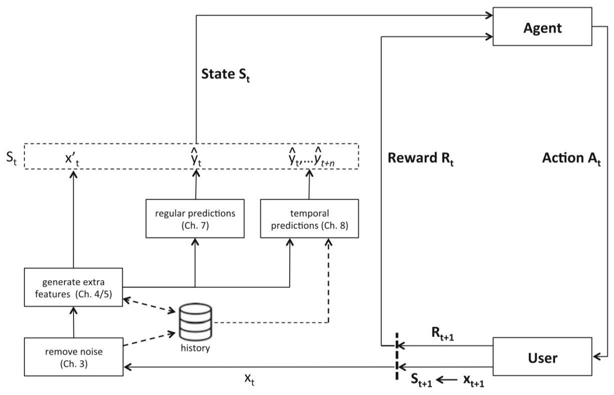

Agent
State S_t
S_t
x'_t
$\hat{y}_t$
$\hat{y}_t, \ldots \hat{y}_{t+n}$
Reward R_t
Action A_t
regular predictions (Ch. 7)
temporal predictions (Ch. 8)
generate extra features (Ch. 4/5)
history
remove noise (Ch. 3)
R_{t+1}
User
x_t
$S_{t+1} \leftarrow x_{t+1}$

Figure 6: Reinforcement Learning in the loop for ML4QS

- **Markov Property**: $\mathbb{P}\{R_{t+1} = r, S_{t+1} = s | S_0, A_0, R_0, ..., S_t, A_t, R_t\} = \mathbb{P}\{R_{t+1} = r, S_{t+1} = s | S_t, A_t\}$
  The conditional probabilities of future state and rewards solely depend on the last state S_t, A_t.
- For every problem that satisfies this property, we can easily create a Markov Decision Process with a finite set of states as in Figure 7
- **SARSA**:
  - On-policy optimization, update Q-values by:
  $$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$
  - Popular policies are ϵ-greedy or softmax over q-values for different actions
- **Q-Learning**:
  - Off-policy optimization, update by taking the maximum Q-value over next state:
  $$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{A'(S_{t+1})} Q(S_{t+1}, A') - Q(S_t, A_t))$$

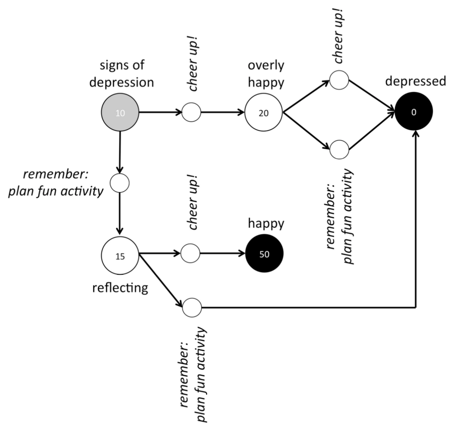


Figure 7: Markov Decision Process for simple example

- **Eligibility traces**: update frequently seen states in a single run more
  - If we have seen a state and action combination more frequently in our history, then we want to increase the weight of the update because it is more eligible (i.e. more responsible for the outcome). We can determine the eligibility by:

$$Z_t(s,a) = \begin{cases} \gamma\lambda Z_{t-1} + 1 & \text{if } s = S_t \wedge a = A_t \\ \gamma\lambda Z_{t-1} & \text{otherwise} \end{cases}$$

  - In our learning algorithms, we can incorporate this by increasing the weight of the update, as e.g. in Q-Learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{A'(S_{t+1})} Q(S_{t+1}, A') - Q(S_t, A_t)) \cdot Z_t(s,a)$$

- Usually, the Q-values are stored in a table. If the number of states and actions are very large, this is not feasible. Alternative is to learn a function/model, that takes as input the state and action, and predicts the Q-value.
- For continuous state spaces, we can discretize it by e.g. the **U-tree** algorithm
  - Start with a single unit/leaf/discrete state where all continuous states are mapped to
  - Collect data by trial and error for a while and estimate the Q-values
  - On the collected data for each leaf, we test whether we can find splits for any attribute X_i with a significant difference in Q-values
  - Choose X_i and its split with the lowest p-value, and create new leafs. Continue until maximum number of leafs is reached