Machine Learning (SS2022)	TU Wien
Summary	
29.06.2022	Summary ML

From: Thinklex

To: The masses

1 Preliminaries

This summary is for the Machine Learning course of the Vienna University of Technology and was initially created for the Summer Term 2022 in a process of studying for the exam. Most of the basic subjects are covered in depth in this summary, but some of the more advanced topics (like SVM or Reinforcement Learning) are not covered in depth.
This summary is available on GitHub, if you find typos, errors, want to add something (content, links, figures, ...) or just want to contribute you can do this here: `https://github.com/alexl4123/ml-summary`.
If the URL doesn't work, you can find it in GitHub if you search for:

- User: alexl4123
- Repo-Name: ml-summary

2 Introduction, Data stuff and Experiments

Machine Learning (ML) is a subfield of a field which is generally known as "Artificial Intelligence", whereas Machine Learning is the subdiscipline which *studies algorithms that can learn from data and make predictions on data*. Machine Learning can further be subdivided into three fields:

- *Unsupervised-ML*: Generally wants to find structure in unlabeled data, has many words which are sometimes used as synonyms, like *Data Mining*. Also Multivariate-Statistics has a many unsupervised techniques, like Principal-Component-Analysis (PCA), which can be used for dimensionality reduction.
- *Supervised-ML*: **Is the main topic of this class**, although some other methods of ML are also taught. This subdiscipline tries to predict *unknown data*, by observing *known* data. Sometimes *offline-learning* is used as a synonym for supervised learning, which underscores the fact, that for a supervised method the data must be known in advance to learn from it. Supervised methods are generally subdivided into *classification*, which tries to predict classes, and *regression*, which tries to predict numeric values.
- *Reinforcement-Learning (RL)*: This process of learning is inspired by nature, i.e. how most animals learn, e.g. one does not touch a hot plate twice (except one is a scientist...). Other names for RL are *online-learning*, which highlights the point, that learning is done repeatedly, and one more name is *trial-and-error-learning*, which tries to point out, that one needs to try out certain actions, which may fail, to learn something. The *exploitation-exploration* dilema is the most present one in this are of learning.

As some names are sometimes used in multiple occasions, the following list provides some terminology which is required for the lecture:

- *Feature*: A column of the dataset table.
- *Observation*: A row of the dataset table.
- *Model*: The model is the algorithm, which is used for the machine-learning process.
- *Model-Fitting*: A model is fitted (aka. trained) with some data, to make sense of the data.
- *Prediction*: Given a fitted model, one can use unlabeled data to predict the labels.
- *Hyper-Parameters*: Hyper-Parameters are the parameters which are used by the model. E.g. for a Neural-Network this is the amount of layers and neurons per layer.
- *Performance of Model*: Can either be the *effectiveness*, i.e. the value of the performance measure, or the *efficiency*, i.e. the computational cost. In the lecture and in this summary performance is mostly used as a synonym for effectiveness.

The general process for analyzing data and making sense of it can be viewed in the *Data Science Process*, which basically has five steps:

1. **Ask** a question: *What is the scientific goal?*, *What do you want to predict or estimate?*,...
2. **Get** the data: *How were the data sampled?*, *Which data are relevant?*,...
3. **Explore** the data: *Plot the data*, *Are there anomalies?*,...
4. **Model** the data: *Build the model*, *fit the model*, *evaluate the model*,...

5. *Communicate and visualize the results.*

2.1 Data

This lecture considers four types of data:

- *Nominal*: Values are distinct symbols, like "green", "blue", etc. Is regarded as a *qualitative* data. Relation or arithmetic operations do not make sense on this data type.
- *Ordinal*: Values have a rank among them, like "large" > "medium" > "tiny". Therefore order computations make sense, but distance measure or most of the other arithmetic operators do not. Is regarded as a *qualitative* data type.
- *Interval*: Are ordered and measured in fixed and equal units, therefore distance measurement makes sense, but most arithmetic operations do not (like multiplication, etc.). Is regarded as a *quantitative* data type. An example is for instance temperature measured in degree celcius (e.g. 20 degrees are not twice as warm as 10 degrees).
- *Ratio*: Are ordered, distance measurements make sense and most arithmetic operations make sense. An example is e.g. temperature measured in kelvin, because e.g. 273.15 kelvin are twice as "warm" as 136.575 kelvin. Is regarded as a *quantitative* data type.

2.1.1 Scaling/Normalization/Standardisation

If multiple quantitative features are present in a dataset, then they may exhibit vastly different value ranges. This might be a problem, if a model is based on the difference between values of different features. Therefore in these problems, one must scale the features, which can be done in various different ways. Some methods which can be used include:

- Log-Transforming each feature, i.e.: $z_i = log(x_i)$
- Min-Max-Scaling: $z_i = \frac{x_i - min(x)}{max(x) - min(x)}$. Transforms value range to $[0, 1]$.
- Z-score standardisation: $z_i = \frac{x_i - \overline{x}}{s}$, where $\overline{x}$ denotes the empirical expected value and s denotes the empirical standard deviation (square root of empirical variance). Transforms value range to $(-\infty, +\infty)$ with mean 0.

2.1.2 Qualitative Data encondings

If one has qualitative data, like the nominal feature eye-color, with values "black", "pink" and "white", then one has several options to proceed:

- *One-hot-encoding* (1-to-N Coding): For this the original feature vector "eye-color" is removed from the dataset and in our case three new vectors are introduced: "black", "pink" and "white". Each new vector is now a boolean vector, which denotes that a certain observation has or does not have the specific feature.
- *Label-encoding*: Another possibility is to stick with the current feature "eye-color", but change the content to a numeric representation, e.g. associate "black" with 0, "pink" with 1 and "white" with 2, then change the nominal representation to the numeric one.

2.1.3 Missing Values

For some datasets not all values are known, e.g. a certain observation only has 9 of the 10 features. Standard procedures exists for this:

- Deletion of feature (column), seems mostly to be a bad practice, as the feature might contain very important information for other observations (especially costly, if there are not many features).
- Deletion of observation (row), seems a better option than deleting the feature entirely, although sometimes also costly (if there are not many datapoints).
- Data imputation, seems to be a good idea, but value then only holds true to a certain extend. Popular measures for imputation are *mean/median* value, *random selection*, *regression*, *clustering* or *nearest-neighbor* (so actually some super-/unsupervised methods).

2.2 Experiments

Generally from the experiment one should get a good estimation how well the model works. Therefore the main task of the experiment is to evaluate the model performance in terms of the performance measure, for which several different possibilities exist (see Classification and Regression).
The general structure of the experiment phase mostly consists of a *training* and a *test* phase. This is e.g. achieved by splitting the original dataset into two datasets, one for training and one for testing. This is done to get a good estimate how well the model works in real conditions with unseen data.
One method which tries to even get better estimates on the true model performance is *Cross-Validation (CV)*. For this one repeatedly performs train-tests-splits of the data and averages the results. E.g. for a five-fold-CV one splits the data into five equal parts (each representing 20% of the data) and then performs five experiments. The first one uses the first four parts as the training data and the fifth part for testing. The second one uses the first three parts and the fifth part for training and the fourth for testing... this continues as long as a part has not been used for testing (i.e. 5 times). Then one averages the results to get the CV-score.

Another CV approach is *leave-p-out CV*, which uses p observations in the test set. This method is mostly computational infeasable.

2.2.1 Data Leakage

In the experiment design it is important to mitigate the threat of data-leakage, which leaks information from the training set into the test set. This is bad, as so one cannot get a true estimate how well the model performs, therefore this has to be avoided.
Important is here, to perform methods like scaling not on the whole dataset, but on the test/train set individually.

2.2.2 Experiment design for Hyper-Parameter-Tuning

For this one should perform another split, e.g. 60% training set for HPP, 20% test set for HPP, 20% test set for sanity check.
So the HPP-tuning should be done on the first 60/20 split, after they HPPs have been selected then another check on the other 20% is performed.

2.2.3 Stratification

In the normal case, the train/test-split is performed completely random, which can lead to a situation where one class is not present in the train- or test-set. To mitigate this one can also use stratification. This method assures, that the classes are approximately equally represented in the test and in the train set.
The problem with this method is, that this might not reflect a real world scenario.

2.2.4 Bootstrapping

A bootstrap is a random sample of a dataset, which was drawn by sampling with replacement, i.e. an observation might occur multiple times (or not at all) in a bootstrap-sample.

2.3 Overfitting/Underfitting

The overfitting/underfitting problem is a general supervised-ML problem. Overfitting means, that the model is trained so well on the training set, that it cannot generalize to unseen data. Underfitting in contrast means, that the algorithm didn't really learned anything at all and therefore needs to consider more datapoints.
If one has a good performance for the training set, but a bad performance for the test set, this accounts to overfitting. In contrast to that, underfitting usually shows bad performance in the training set as well as in the test set.

For this context also the *bias* and *variance* problemi/tradeoff arises. High *bias* can be seen as something similar to underfitting, whereas high *variance* is similar to overfitting. This is as, bias is an error fundamental to the assumptions of the data, whereas high variance is caused by small fluctuations in the data, which leads to big differences in the output.
The important thing here is, that bias and variance can simultaneously be high and minimizing both of them simultaneously is hard.

3 Classification

A classification problem arises, when one wants to classify unseen data, e.g. the classification of a car maker according to visual features of the car.

3.1 Performance Measures

For classification problems, there are multiple performance measures. For a binary classification task one may choose between *Accuracy*, *Precision* and *Recall*.

		Actual value (groundtruth)		
		true	false	
Prediction / Test outcome	true	True Positive (TP)	False Positive (FP, Type I Error)	Precision $\frac{TP}{TP+FP}$
	false	False Negative (FN, Type II Error)	True Negative (TN)	
		Recall/Sensitivity $\frac{TP}{TP+FN}$	Specificity (True negative rate)	Accuracy $\frac{TP+TN}{TP+TN+FP+FN}$

Additionally the *F1-Score* exists (seems not to be used in the lecture), which can be computed as:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

There are also various other performance measures, which one can look up in the internet.

3.1.1 Performance Measures for Multiclass Classification

Again several options possible, the lecture presents *micro-average*, *macro-average* and *costs of misclassification*. To understand those three performance measurements, have a look at the confusion table below (note that the ground truth is "vertical", which is consistent with the above notion, but in the lecture they switched it):

		Groundtruth			
		1	2	3	Sum
Predictions	1	20	10	0	30
	2	0	0	5	5
	3	5	10	95	110
Sum		25	20	100	145
Accuracy		0.8	0	0.95	

Micro-Average: Can now be calculated, by summing up all true classifications of all cells and then divide by the total amount of observations. In the following the micro average for our example is calculated, where n resembles the amount of classes:

$$acc_{micro} = \frac{\sum_{i=1}^{n} TP_i}{\#samples} = \frac{20 + 0 + 95}{145} = 0.79$$

Macro-Average: Can be calculated, by averaging the accuracies of each individual class. Again the formula for our case is shown:

$$acc_{macro} = \frac{\sum_{i=1}^{n} acc_i}{n} = \frac{0.8 + 0 + 0.95}{3} = 0.58$$

Costs of missclassification: For this one must define another matrix, the *cost matrix*, which defines how much a missclassification costs. In our case we assume different costs for different missclassifications:

Cost Matrix		Groundtruth		
		1	2	3
Predictions	1	0	20	1
	2	2	-1	2
	3	1	40	0

In our case this results in costs of:

$$costs = \sum_{i \in \{1...n\}, j \in \{1...n\}} confusionMatrix_{ij} \cdot costMatrix_{ij}$$

$$costs = 20 \cdot 0 + 0 \cdot 2 + 5 \cdot 1 + 10 \cdot 20 + 0 \cdot -1 + 10 \cdot 40 + 0 \cdot 1 + 5 \cdot 2 + 95 \cdot 0 = 615$$

3.2 Models

In the lecture different classifiers were presented, they are summarized in the following.

3.2.1 K-Nearest-Neighbor (KNN)

The KNN classifier predicts the class of an observation i by computing a distance measure to other observations and selecting in a second step the k-nearest points. Then a majority vote is done among the k-nearest points and the class most often contained among the k-points is selected for the observation i. If a stalemate between two classes exists (i.e. in the environment are as many neighbors for class m as for n), then this may be resolved with a random choice, or by other means.

Regarding the distance measure, several ways for computing the distance do exist. The most popular are the $euclidean-$ and the $cityblock - distance$ measures.

The maximum value for k is the amount of observations in the training dataset, if this value is selected one obtains a classifier which always predicts the majority class.
Computation wise some performance improvements compared to a naive implementation exists, one such improvement is the *K-d-Tree*, which recursively splits space along points. Also other calculation methods than a simple majority vote on the k-nearest-neighbors exist.

3.2.2 Perceptron

A perceptron wants to model a single brain cell (i.e. neuron) by a very abstract view on the matter. The figure below depicts the main components of a perceptron, the *inputs* (x_i), the *weights* (w_i), the

accumulation (Σ) and the *activation-function* (f).
In principle the accumulation function sums up the following term:

$$\Sigma = \sum_{i=1}^{n} w_i \cdot x_i$$

.
This is then passed into the activation function, i.e. $y = f(\Sigma)$, which is then the output.

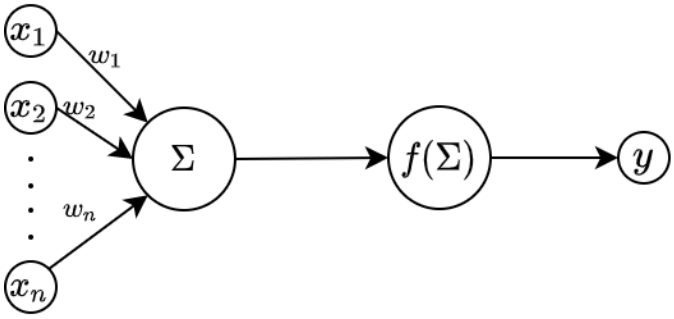

Figure 1: Principle of a Perceptron

For the activation function several different options are possible, among them are the Threshold/Heaviside-, linear-, sigmoid-, tanh- and RELU-functions. The pictures below show the five functions:

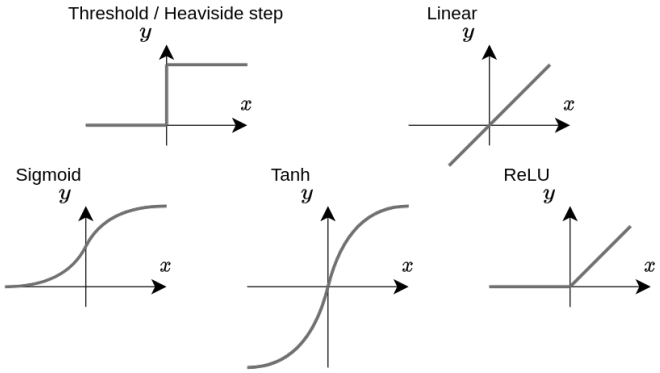

Figure 2: Activation function 0

Learning is achieved by repeatedly taking observations (from the training set) into account, then compute the output for each observation and check if it matches the correct output. I.e. one checks $y = y'$, where y is the prediction and y' the true value.
Then one can update the weights of the perceptron according to the learning function. One can derive this learning function by computing the gradient. For this we must first define the loss, which is $L = y' - y$. Then we take the squared loss and build the derivative to each w_i:

$$\frac{\partial L^2}{\partial w_i} = 2 \cdot L \cdot f'(\sum_{j=1}^{n} w_j \cdot x_j) \cdot x_i$$

This is now used to update the weight-values, for this one changes the 2 to the *learning-rate-parameter* α:

$$w_i = w_i + \alpha \cdot L \cdot f'(\sum_{j=1}^{n} w_j \cdot x_j) \cdot x_i$$

A perceptron can only linearly seperate data, therefore it can learn an *and*- or *or*-Gate, but not a XOR-Gate. Further if the data is messy, such that it is not linearly seperable it can also not learn it, but will *oscilate* - therefore another stopping criteria is needed.

3.2.3 Decision Trees

Decision Trees (DTs) are a rather old model, which can be depicted as a tree, where the inner nodes are decision nodes and the leaf nodes the classes.
They can either be constructed by experts or learned from data. The two easiest models of a Decision Tree are the *1R (One Rule; Decision Stump)* tree and the *ZeroR (Zero Rule)* trees. The first has just one rule, whereas the second one always returns the majority class. This is useful for a baseline classifier, i.e. DummyClassifier.
The **basic algorithm for the decision tree learning** goes repeatedly through all remaining features and selects the one where splitting seems to be the best. After the split this is done until all observations are predicted correctly, or if this is not possible one must result to other means. These other means consist normally of applying the majority function, but also a random draw is possible, etc...; To compute the "best" splitting measures 4 measures are presented in the lecture:

- Error Rate: Goal is to minimize error rate. I.e. we look at all possible splits and take the one which is best, then repeat. One can use as the error rate for instance $1 - acc_{micro}$, or $1 - acc_{macro}$.
- Information Gain: For this a few more terms have to be introduced:
  - l represents the amount of different classes (i.e. for eye-colors "white", "pink" and "black" $l = 3$).
  - Entropy: $H(X) = -\sum_{i=1}^{l} p(x_i) \cdot log_2\, p(x_i)$
  - Entropy measures the impurity of a feature and is 1 for a completely impure feature (e.g. 50% class 1 and 50% class 2) and 0 for a feature, where all values are the same.
  - The goal is now to reduce entropy, which can be done by Information Gain (IG). It is calculated between the difference of the current entropy value and the sum of the relative entropies of the new splits (See next point).
  - $IG(X_1, ..., X_j, ...X_k) = H(X) - \sum_{i=1}^{k} p(X_i) \cdot H(X_i)$
- Gini Impurity (Gini index)
  - l represents the amount of different classes (i.e. for eye-colors "white", "pink" and "black" $l = 3$).
  - Measures inequality between values of a distribution.

  - "How often a randomly chosen element from the set would be incorrectly labeled, if it was randomly labeled according to the distribution of labels in the subset"
  - $IG(p) = \sum_{i=1}^{l} p_i(1 - p_i)$
- Relative Information Gain
  - For this measure it is assumed, that one has already computed the Information-Gain IG.
  - The problem with the information gain is, that if one has a feature with very many different attributes like ID, then this feature would result in a perfect information gain and will always be selected. But this leads to a massive overfit, therefore the *Relative Information Gain (IGR)* is introduced. For this the *Split-Information-Gain (V)* is calculated.
  - l represents the amount of different classes (i.e. for eye-colors "white", "pink" and "black" $l = 3$).
  - $V(X_1, ..., X_j, ...X_k) = -\sum_{i=1}^{k} \frac{|X_i|}{|X|} \cdot log_2 \frac{|X_i|}{|X|}$
  - $IGR(X_1, ..., X_j, ...X_k) = \frac{IG(X_1,...,X_j,...X_k)}{V(X_1,...,X_j,...X_k)}$

A problem for the above sketched learning algorithm is, that it could massively overfit the data. A method to combat this is *(post-)pruning*, which removes parts of the tree by replacing the decision node with a majority vote. Also *pre-pruning* is possible, which is done before/during the construction of the tree.
A further problem with decision trees is, that there stability is questionable at best, i.e. small changes in the training data may lead to significantly different datasets.

3.2.4 Random Forests

Is a combination of *Boostrapping* (see 2.2.4) and *Decision Trees* (see 3.2.3).
The basic idea is, that

1. one first creates multiple datasets by making bootstrap samples
2. then constructs for each such sample a decision tree
3. and finally combines the decision trees in a certain way (e.g. Majority-Voting).

This works, as the decision trees are highly unstable and therefore small deviations in the training set, may lead to significantly different decision trees.
For evaluation one can use the default methods, but one additional is proposed:

Out-of-bag error (OOB): As we use bootstrapping, it is likely that for a single bootstrap-sample not all observations from the original dataset are present. Therefore one can add the missing observations into a *bag*. Then one evaluates the entries of the bag.
This process is now applied to each bootstrap sample, then one computes the average error of the observation and finally aggregates over all bootstrap samples.

The exact calculation should be the following, where N is the set of observations, B the set of bootstrap samples, function $in(i, j)$ returns true iff observation i is contained in bag j, $value(i, j)$ is either 0 (if i is wrongly predicted) or 1 (if i is correctly predicted) and $decisionFunction(oob_i)$ can be implemented in various ways, basically it needs to aggregate all predictions of the observation,

which can for instance be done with a majority vote function or a mean calculation.
Note that in the lecture the Out-of-bar error is defined twice, although the defnitions are similar they differ in the details - the other one calculates the average out of bag error per bag, and then averages again (second algorithm).

```
oob... List of observations

for i in N:
    oob_i ← []
    for j in B:
        if in(i,j):
            oob_i.append(value(i, j))

    oob_i ← decisionFunction(oob_i)

oob_value ← (Σ_{i∈N} oob_i) / |N|

return oob_value
```

```
oob... List of observations

for i in B:
    oob_i ← []
    for j in i: (for each element in the bag)
        oob_i.append(value(j, i))

    oob_i ← (Σ_{j∈i} oob_i[j]) / |i|

oob_value ← (Σ_{i∈B} oob_i) / |N|

return oob_value
```

3.2.5 Naive Bayes

Is a form of simple statistical modelling, which assumes:

- All attributes are equally important
- All attributes are statistically independent (this actually doesn't hold most of the time)

We can now state the question, what is the probability of the class given an instance, where evidence E are the observations non-class attribute values and event H is the **class** value.
Bayes theorem tells us, that the *posteriori* probability $P(H|E)$, given the *a priori* probability $P(H)$, the *likelihood* $P(E|H)$ and the *marginal* probability $P(E)$ is:

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

As we assume that the attributes (We have n, the set of all attributes is denoted as N), are independent, we can take use of the product rule:

$$P(H|E) = \frac{P(E_1|H) \cdot P(E_2|H) \cdot ... \cdot P(E_n|H) \cdot P(H)}{P(E)} = \frac{P(H) \cdot \Pi_{i \in N} P(E_i|H)}{P(E)}$$

For this we can state, that we actually do not need the *marginal* probability ($P(E)$), as this value is constant for the given E, which normalizes the result to obtain a probability, but a likelihood is also fine for our purposes. This leaves us with ($\propto$ means proportional):

$$P(H|E) \propto P(H) \cdot \Pi_{i \in N} P(E_i|H)$$

The goal of Naive Bayes can now be reduced to finding class c which maximizes the above equation, so:

$$p = \underset{c \in H}{\operatorname{argmax}} P(H = c) \Pi_{i \in N} P(E_i|H = c)$$

If the probability of a single $P(E_i|H)$ is zero, than the posteriori will also be zero, which can be seen as a problem. Therefore the *laplace estimator/correction/smoothing* exists, which is for a given probability calculation, where $|N|$ is the total amount of samples, n_i the amount for i, α is the laplace correction value and d is the amount of different $i's$ (i.e. for an attribute with 3 different classes it is $d = 3$):

$$\theta = \frac{n_i + \frac{\alpha}{d}}{|N|}$$

The next question arises, when one wants to take numeric attributes into account: How to do this? To fix this problem, normally a *Gaussian* probability distribution of the values are assumed:

$$\overline{x} = \frac{1}{|N|} \sum_{i \in N} x_i$$

$$s = \sqrt{\frac{1}{|N| - 1} \sum_{i \in N} (x_i - \overline{x})^2}$$

$$f(x) = \frac{1}{\sqrt{2\pi} \cdot s} e^{-\frac{1}{2} \cdot \left(\frac{x - \overline{x}}{s}\right)^2}$$

As probability densities are not probabilies, one must integrate them to gain them. But in principle after the transformation it works similar to the qualitative data approach we had above.

3.2.6 Covering Algorithms

At each step of the algorithm a rule is identified that "covers" some of the instances. Is somehow similar to a Decision Tree in the sense that it is basically a set of rules. The difference is most obvious in the multiclass case, where the covering algorithm focuses on a single class (i.e. generates all rules for the one class), where the decision tree takes all classes at the same time into account.
Covering algorithms work, that they add tests to the rule that is currently constructed. In contrast

to this, decision trees work by adding tests to the tree that is under construction.
The goal is now to select a test, which maximizes accuracy. For this t denotes the number of instances covered by the rule, p the number of positive instances covered by the rule, $t-p$ are the errors made by the rule, therefore one strives to maximize $\frac{p}{t}$. We are finished when $\frac{p}{t}=1$.
In the lecture the pseudocode for *PRISM* is provided, which basically works according to the above pricinple, by starting at one class and do the rule generation according to the $\frac{p}{t}$ maximization mechanism. Then it adds such rules, until all instances with the class are identified correctly. Then it moves on to the next class, etc., etc.

3.2.7 Bayesian Networks for Classification

Although in the lecture Bayesian Networks are studied generally and therefore not with a focus on classification, in the following it is still regarded as to be contained in the classifier section. Before reading this section, be sure to read Section 3.2.5.
A Bayesian Network consists of a set of variables (nodes), a set of directed arcs between the variables, together the nodes and arcs form a acyclic graph and the probabilities for every node given its parents is known.

As in the Naive Bayes, given evidence E one can select the class with the highest probability given E, i.e. $\text{argmax}_{c\in H}\, P(H=c|E)$. In the Naive Bayes case we applied the product rule and discarded the marginal probability to obtain an easier to compute solution.
In the Bayesian Network case one can use the structure of the network to ease computation, i.e. that the probabilities for every node given its parents is known, therefore we only need to consider the part of the evidence E which is part of the parents of node H.

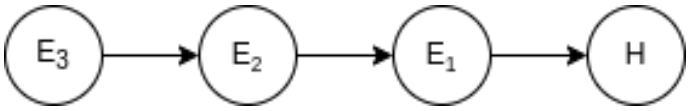

Figure 3: A Bayesian Network

Generally the computation of a joint probability, e.g. $P(H=a, E_1=t, E_2=f)$, can be computed by the joint probability distribution. For example, consider the Bayesian Network depicted in figure 3. To calculate the Joint Probability Distribution given our example, we get:

$$P(H=a, E_1=t, E_2=t) = \sum_{e_3\in E_3} P(H=a|E_1=t)\cdot P(E_1=t|E_2=t)\cdot P(E_2=t|E_3=e_3)\cdot P(E_3=e_3)$$

Note that $P(H=a|E_1=t)$ and $P(E_1=t|E_2=t)$ are constant.
An algorithm for efficiently calculating the joint probability distribution is the *Variable Eliminiation Algorithm*. Given a complete joint probability calculation with many sums, it takes out constant paths, that some parts don't need to be calculated again. Finding the best order is NP-hard, therfore usually heuristics are used for this process.

D-Seperation: To check whether two variables are independent given some evidence (I think this is useful to speed up computation of the joint probabilities), one can use D-Seperation. In general in D-Seperation one must distinguish between three types of Node-Structures (See Figure 4 and the three points below):

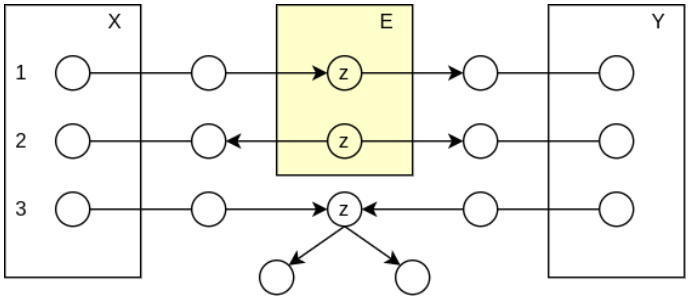

Figure 4: D-Seperation

1. Is called a seriel connection. X and Y are seperated given that z is in the evidence E.
2. Is called diverging connection. X and Y are seperated given that z is in the evidence E.
3. Is called converging connection. X and Y are seperated, if z is NOT in the evidence E and further no descendent of z is in E.

The next question to answer regarding Bayesian Networks is how to construct these things?
In general two options are possible: Experts construct it, or it is learned from the data. Further it can be stated that human experts are better in finding the structure of the network, whereas machines are better for calculating the probabilities.
Given a network one can calculate the probabilites by counting, e.g. for calculating the probability $P(H = t|E_1 = t)$ one counts $\#(H = t \wedge E_1 = t)$ and divides this by $\#(E_1 = t)$. The other probabilities can be calculated accordingly.
Learning the structure is harder. As a measurement for how good the model fits the data, the *goodness of fit* measure is introduced, where D denotes the data set and M denotes the model, j denotes an individual observation and i denotes the attributes/nodes:

$$P(D|M) = \Pi_j P(s^j|M) = \Pi_j \Pi_i P(N_i = v_i^j|Parents(N_i), M)$$

Further the log likelihood is introduced:

$$logP(D|M) = \sum_j \sum_i P(N_i = v_i^j|Parents(N_i), M)$$

A good model should have a good fit of the data and further should have a low complexity (not many parameters), so the objective should be to maximize (where α is a parameter that controls how important it is to reduce the complexity of the network and $\#M$ denotes the amount of parameters):

$$logP(D|M) - \alpha \#M$$

With this performance measure in mind, one can construct search algorithms for finding the best structure. For instance one can use meta heurstics, like hill climbing, evolutionary algorithms, tabu search, simulated annealing, etc., etc..
For the hill climbing/local search method, one can use this five step process:

1. Construct an initial network

2. Calculate the score of the current BN network
3. Generate the neighborhood by modifying the current network
4. Select one of the networks in the neighborhood as a new current network for the next iteration
5. Go to step 3 if termination criteria is not fulfilled

3.3 Support Vector Machines (SVMs)

SVMs are a type of classifier, which try to seperate linear seperable data. They are intrinsically binary classifiers, i.e. can only classify between two classes. To extend them to multiclassification one can use for instance 1 vs *all* (Classify one class against all other classes, create a model for each class) classification.
SVMs are a method which try to seperate data in a "best possible way". In a basic mode they only seperate linear data. When one wants to perform non-linear classification, one can use the *kernel trick*, to map the data into high dimensional feature space which is again linearly seperable.
The name of SVMs come from "Support Vectors", which are vectors in the data-space which seperate the data linearly (seperating line/margin line).

The lecture just gives a rough (but very mathematical) overview, therefore in the following a few key terms are listed and explaiend:

- Hard-Margin: If the data is exactly linearly seperable, one can exactly classify the data
- Soft-Margin: If the data is not exactly linearly seperable, one can use a soft-margin to allow some observations to be misclassified
- Kernel: Is a mathematical function, which transform the data.
- Non-linear kernels: If the data is not linearly seperable, then it might be in a higher-dimension.

In general SVMs seem to be good for rather small datasets and SVMs are not always better. Further it is noted, that kernels can also be used with other algorithms and not only SVMs.
As far as the literature shows, an SVM always finds the *optimal hyperplane* by e.g. Lagrange Optimization (`https://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf`).

4 Regression

Regression is all about predicting numeric values, e.g. house prices.

4.1 Performance Measures

Compared to Classification other performance measures must be used, in the following a few of them are presented:

- **Mean Squared Error (MSE)**:
  - When we have n samples, the MSE is calculated by taking the mean of the squared differences between the predicted $\hat{(Y_i)}$ and the observed (aka. true) Y_i values.
  - $MSE = \frac{1}{n} \cdot \sum_{i=1}^{n} (Y_i - \hat{Y_i})^2$
- **Root Mean Squared Error (RMSE)**:
  - $RMSE = \sqrt{MSE}$
- **Mean Absolute Error (MAE)**:
  - Given n samples, the MAE is calculated by averaging the absolute errors.
  - $MAE = \frac{1}{n} \cdot \sum_{i=1}^{n} |Y_i - \hat{Y_i}|$
- **Median Absolute Error (MAD)**:
  - Given n samples, the MAD is calculated by taking the median of the absolute errors.
  - $MAD = median(|Y_1 - \hat{Y_1}|, ..., |Y_n - \hat{Y_n}|)$
- R^2**-Score**:
  - Is a measure for the goodness of fit.
  - Has a range of theoretically $(-\infty, 1]$, where 1 represents the best value.
  - In SciKit-Learn it is defined as (note, $\hat{Y_i}$ denotes the prediction and $\overline{Y_i}$ denotes the mean of the observed/true values):
  - $R^2 = 1 - \frac{u}{v} = 1 - \frac{\sum_{i=1}^{n} (Y_i - \hat{Y_i})^2}{\sum_{i=1}^{n} (Y_i - \overline{Y_i})^2}$

4.2 Models

Many of the models described in section 3.2 can be transformed into a regression task, further a few new ones are introduced. All of the following are discussed in the lecture:

4.2.1 Linear Regression

Is the task of fitting a line $y = w_0 + \sum_{j=1}^{l} w_j \cdot x_j$, where l denotes the dimensions to predict. The task is then to calculate the best values for w_j, for which one must at first define a cost metric which should be minimized. This cost metric is the *Residual Sum of Squared (RSS)*:

$$RSS(w) = \sum_{i=1}^{n}(y_i - (w_0 + \sum_{j=1}^{l} w_j \cdot x_{ij}))^2$$

The solution to this minimization problem can either be found in an analytical way or by an iterative algorithm, the gradient descent algorith, which basically works by calculating the gradient and moving along the gradient to a better solution (basically the same for the Perceptron in 3.2.2, here again α denotes the learning rate):

$$w_j = w_j - \alpha \frac{\partial RSS(w)}{\partial w_j}$$

This update is performed until convergence, where all w_j are updated simultaneously. Further an analytical approach exists, which is shown below how one may derive it. For this one takes the RSS, then does a bit of Matrix magic (just transformations) and then takes the derivative. After taking the derivative the resulting part is set to zero (strongest gradient) and one has the analytical approach:

$$\begin{aligned} RSS(w) &= \sum_{i=1}^{n}(y_i - (w_0 + \sum_{j=1}^{l} w_j \cdot x_{ij}))^2 \\ &= (\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w}) \\ &= \mathbf{y}^T\mathbf{y} - 2w^T\mathbf{X}^T\mathbf{y} + w^T\mathbf{X}^T\mathbf{X}w \\ \frac{\partial RSS(w)}{\partial w} &= 0 - 2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}w \\ w &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \end{aligned}$$

4.2.2 Other regression techniques

Other regression techniques exist, like polynomial-, ridge- and lasso-regression.
Polynomial regression strives to fit data, which is better predicted by a non linear approach.

Ridge and **Lasso** regression both want the same things: Minimize the amount of needed variables which should therefore prevent overfitting. They do this by adding an additional term to the cost function (where $||w||_2^2$ is normally the euclidean norm (in the lecture the square root is not taken) and the $||w||_1$ is normally the manhatten distance):

$$\begin{aligned} Cost_{Ridge} &= RSS(w) + \lambda \cdot ||w||_2^2 \\ Cost_{Lasso} &= RSS(w) + \lambda \cdot ||w||_1 \end{aligned}$$

4.2.3 KNN-Regression

Similar to normal KNN, so first calculating nearest neighbors, but then instead of taking the majority vote for the class, one can calculate the mean among the nearest points - done. E.g. for a $k = 5$ KNN:

$$y_i = \frac{\sum_{j \in NearestFive(x_i)} \hat{y}_j}{5}$$

4.2.4 Regression- and Model-Trees

Similar to decision trees, but the splitting criterion is different (minimizes intra-subset variation), the termination criterion is different (standard deviation becomes small enough), the pruning criterion is different (based on numeric error measure) and the prediction is different (predicts average value of instances). To predict a certain value, one traverses through the tree, then goes to the leaf node and looks up the value there.
More sophisticated method is the Model-Tree, which has for each leaf node a linear-regression function.

5 Reinforcement Learning (RL)

Is a type of learning that is inspired from nature (trial and error learning), e.g. one does not touch a hot place twice (except one is a scientist). It is different from supervised learning, as in RL must make sense of the world from it's own experience, where an agent quickly comes into the exploration/exploitation dilema, which means that an agent wants to take the best possible action, but an agent can only take this action when it knows that this action is the best and for this exploration is necessary.

5.1 Components of a RL system

For RL to work out several things are necessary:

- Policy/Action - Selects an action which an agent should take, e.g. Selecting a slot machine in a casino
- Reward Signal - Immediate reward from the environment, e.g. Reward from the slot machine.
- Value Function - What the agent thinks, a policy is worth, e.g. The agent thinks, that slot machine B is best.
- Model/State (Optional) - Model of the environment

The lecture presents a few tabular solution methods, where *tabular* just means, that state and action spaces are small and that they can be saved in a table.

5.2 Action selection methods

Different possible, in the lecture:

- Greedy action selection: Always take the best, i.e. $A_t = \text{argmax}_a\, Q_t(a)$
- ϵ-Greedy action selection: With probability $1 - \epsilon$: Make greedy selection. With probability ϵ make a random draw.
- Roulette wheel selection: Create a probability function where better actions have a higher probability of selection and make a random draw from this probability function.
- And they also provide the *Upper-Confidence-Bound Action Selection* mechanism, which works like: $A_t = \text{argmax}_a \left[Q_t(a) + c\sqrt{\frac{lnt}{N_t(a)}} \right]$

5.3 Multi Armed Bandit (MAB)

Several possible options for actions possible, task is to select the best one. We do not have a state. And the goal can be defined mathematically as ($q_*(a)$ is the true probability/value of action a, E is the expectation, R_t is the reward, $A_t = a$ is the selected action a):

$$q_*(a) = E[R_t | A_t = a]$$

So the goal of the MAB is to model $Q(a)$ as close as possible to $q_*(a)$.
The basic update of the values can be defined as ($1_{A_i=a}$ is 1 if a was selected at time t, otherwise 0):

$$Q_t(a) = \frac{sum\ of\ rewards\ when\ a\ taken\ prior\ to\ t}{number\ of\ times\ a\ taken\ prior\ to\ t} = \frac{\sum_{i=1}^{t-1} R_i \cdot 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}}$$

The above formula has one significant downside: It cannot be computed *on-the-fly/online*, i.e. it must be computed with all previous values after each update. Therefore the lecture also provides an incremental version of the above value estimation.
Further it provides an update for non stationary problems, i.e. where $q_*(a)$ changes over time. They fix this by using the incremental version but give more recent values a higher priority.
Another evaluation method is, that at the beginning one does not initialize the $Q(a)$ with 0, but takes an optimistic approach.

5.4 Markov Decision Process (MDP)

As in MABs we have feedback, but now also states/a model of the environment. The basic difference for action selection and value updates is, that one now selects the best action per time-state tuple and also updates the value per time-state tuple, not just per time.

In this context also the *Bellman Equation* is mentioned, which expresses a relationship between the value of a state and the values of its successor states. The bellman equation can be computed by dynamic programming.

Further *Monte Carlo Methods* exist, which sample sequences and then learn from the sequences.

The last topic in the lecture is *Temporal-Difference learning*, which is a central idea of reinforcement learning. They can learn directly from experience without a model. They operate as Monte Carlo Methods on episodes, but learn after each time step. *Q-Learning* is a very popular Temporal-Difference algorithm.

6 Feature-Selection, Auto-ML, Hyper-Parameter-Tuning, Ensemble Learning and similar things

6.1 Feature Selection

One should select features that minimize redundancy and maximize the relevance to the target. This means, that for say 10 attributes in a given dataset, one wants to select those, which seem to be the most informative.
A naive approach is to tryout all subsets of the 10 features, which results in 2^{10} possibilities - which is infeasible, for a large amount of attributes, therefore the task of features selection is NP-hard.

One can use search to find a local optimum of features to use, by e.g. using a genetic algorithm. Other strategies are for instance using a greedy approach, which comes in two flavors: *Forward Selection* starts with an empty set of features, then features are added iteratively by comparing how good they improve the performance measure, until the desired number of features is selected.
Backward elimination starts with all features selected and eliminates all those features iteratively, which seem to not contribute to the current performance of the model, until we arrive at the desired number of features.

Other possibilities for feature selection include dimensionality reductions, such as the Principal Component Analysis (PCA), Analysis of Correlation or Factor Analysis.
Again other (supervised) methods exist, like entropy/information gain, mutual information, maximum likelihood estimation, odds ration,...

Two further methods are presented, a wrapper (using the algorithm as a performance measure, by using search strategies) and a filter (using another proxy measure, a lot faster...) strategy.

6.2 Meta Learning

Can be described with *Learning about learning* and deals with many machine learning techniques, like model selection, method combination or Hyper-Parameter-Tuning.
To begin, one must know the *No Fre Lunch (NFL)* theorem, which states that "any two algorithms are equivalent when their performance is averaged across all possible problems" ([1]) .

6.2.1 Algorithm/Model Selection

The first problem discussed is the one of *Algorithm Selection*, where it is hard to evaluate the performance of an algorithm, because taking the NFL theorem into account it is hard where should it be evaluated: On a single problem?, or on a certain set of problems? or on something entirely different?
The lecture provides two possibilities for possible performance measurements:

- Closed Classification World Assumption (CCWA)
  - Assume that the problems the algorithm works on, are a certain subset of real life problems.

  - Therefore one can show, that the algorithm works better on this subset.
  - Is favored, as therefore one can easily compare algorithms, but in practice it is sometimes hard to characterize real-life classification tasks.
- Open Classification World Assumption (OCWA)
  - Assume that there is no such structure.
  - Class of tasks, in which algorithm performs well is characterized.

An example for a detailed formulation of a CCWA is the Rice Framework (see Figure 5), which is formulated like *For a given problem instance $x \in P$, with features $f(x) \in F$, find the selection mapping $S(f(x))$ into algorithm space A, such that the selected algorithm $\alpha \in A$ maximizes the performance mapping $y(\alpha(x)) \in Y$.*
This basically means, that we want to learn S (the selection mapping), which is just a fancy term for associating a problem with an algorithm (finding the best algorithm α for problem x).

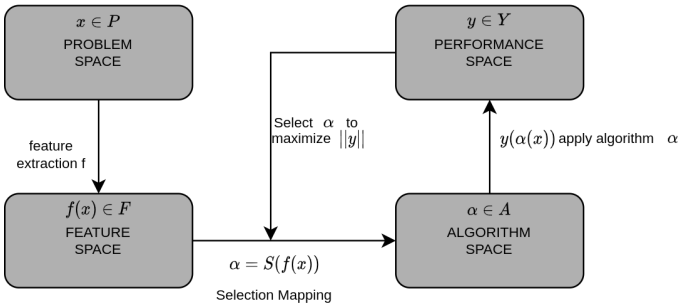

Figure 5: Rice-Framework

In Meta Learning it is very important to select the right features for a given problem, for instance one can use Statistical-/Information-theoretic-(like *Number of Attributes*, *Number of classes*, *Some ratios*, *Averages*), Model-based- (like for a decision tree the properties of it (maximum tree depth,...)), Landmarking- (see below) or Computational-cost-features.
Landmarking is a process which takes algorithms and checks on which problems the algorithms work well (these algorithms are called *Landmarker*s). With the performances of these Landmarkers on certain problems, one can state something about the nature of these problems.
Further and important for algorithm selection, these landmarkers should be highly efficient in terms of performance, so they should be fast. Also one should be able to associate an algorithm with landmarkers, e.g. for algorithm a1, the landmarkers i1 and i2 correspond roughly to the problems, on which it works well.

6.2.2 Hyper-Parameter Optimization (HPO)

HPO is an important topic for tweeking machine learning models. It can be described as a search problem, therefore normal metaheuristics, etc. can be used for it.
The easiest techniques to use are Grid- and Random-search. Grid search is an exhaustive search, which just searches all different possibilities in a well defined multi-dimensional-grid.
Randomized search selects a few random hyper-parameters and checks their performance.

A very promising technique is *Sequential Model-based Bayesian Optimization*. It is a general framework for minimizing blackbox functions *f*. As the name suggests it uses bayes as the baseline for it's optimization, so remember:

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

As we are again not interested in the specific probability we can drop $P(E)$ as in Naive-Bayes ($\propto$ means proportional):

$$P(H|E) \propto P(E|H) \cdot P(H)$$

The bayesian optimization process now assumes a probability distribution $P(H)$, which model "how well certain Hyper-Parameters work", i.e. the higher the better. At the beginning of the Bayesian-Optimization process this must not be the "true" value, therefore it iteratively it's prediction by computing the posterior $P(H|E)$, by taking into account new evidence E which was sampled from the true function.
The function that approximates the true $P(H)$ is called **Surrogate Function** and should be efficient. The function which selects the best algorithm selection from the surrogate function is called **Acquisition Function**. A pretty good explanation can be found in: `https://machinelearningmastery.com/what-is-bayesian-optimization/`

6.2.3 AutoML

Combines several things like *Feature Selection*, *Feature Preprocessing*, *Feature Construction*, *Model Selection* and *Parameter Optimization*.

6.3 Ensemble Learning

Ensemble Learning can be seen as a sort of learning, which combines several classifiers to improve performance of the overall model. An example for this is the Random Forest classifier, which combines serveral Decision Trees.
The terminology for ensemble learning is quite clearly defined: There exist Homogenous (i.e. all classifiers are of the same type) and Heterogeneous (i.e. classifiers may differ) ensemble learning techniques. Random Forests are homogeneous, as they only use Decision Trees.
One can combine the classifiers by majority voting, i.e. selecting the class which most classifiers predict. For this it is important to note, that a classifier may have different types of output:

1. Type 1 - Abstract Level: Classifier produces only the class/label prediction
2. Type 2 - Rank Level: Classifier produces a list of probable class labels
3. Type 3 - Measurement Level: Classifier procudes probability distribution for each labels.

Depending what the output of the classifiers are, the ensemble decision maker may make decisions not only on majority voting, but e.g. maximizing the probabilites, or averaging them.
When using majority voting, one can estimate the accuracy of the ensemlbe when one assumes independence between the classifiers and all classifiers have equal accuracy:

$$P_{maj} = \sum_{m=|L/2|+1}^{L} \binom{L}{m} p^m (1-p)^{L-m}$$

This would in theory lead to the conclusion, that one may just add an infinite amount of classifiers and gain 100% accuracy, but this does not work in real life... I think the reason for this is, that the classifiers may fail on the same observations, as the observations are inherently hard to predict.

6.3.1 Bagging

Tries to achieve independent classifiers by varying the training set, as it is done with Random Forests. I.e. one takes bootstrap samples from the data set.
For this to work out, the classifiers should be unstable and class decision vote is taken by plurality vote.

6.3.2 Boosting

Boosting is a process, where a classifier depends on a prdecessor. In the process the errors from the predecessor is analysed and decided which part of the data to focus on and weights are set for "hard" parts.
Initially the weights are **equal** amont all training samples, e.g. $\frac{1}{n}$ for each observation.

The iterative process works as in the following: A sample is taken, and classifiers are trained. Then the weights are adjusted to give more importance to misclassified samples. Subsequent classifiers should thus focus on the hard samples by either directly utilizing weights in the learning algorithm or by sampling techniques which sample the hard parts multiple times.

AdaBoost is an iterative ensemble learning algorithm, which has a pool of classifiers h and a number of iterations T. The final classifier is a linear combination of the individual classifiers, which are comprised of the classifiers h and a coefficient α:

$$H(x) = sign(f(x)) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$$

When learning AdaBoost, it takes care of the values which were misclassified previously. The classifier coefficient is calculated by how good the classifier works on the current weighted data set.

Gradient Boosting is another boosting technique which combines gradient descent with boosting. Is again an iterative process, where in each stage a new weak learner is introduced to compensate the shortcomings of the previous weak learners. In Gradient Boosting the shortcomings are identified by gradients.

7 Multi-Layer Perceptrons and Deep-Learning

7.1 Multi-Layer Perceptrons (MLP)

Combines several perceptrons into layers, where each layer is composed of perceptrons. MLPs are *feed-forward* Neural Networks, which means, that they do not have cycles.
This means, that e.g. we have 5 neurons in the first layer, 3 in the second and 4 in the third. Generally the first layer is described as the **input** layer, the last layer as the **output** layer and all layers in between are called **hidden** layers.
Each neuron has the same structure as a perceptron, with an activation function, which were already discussed in the perceptron part (See Section 3.2.2). For MLPs it mostly holds, that all neurons have the same activation function. Further it is noted that for some tasks, like multiclass-classification, it is necessary to give the output of the last layer as a probability distribution. This can be achieved by the **Softmax** layer/function. It scales all outputs to a probability distribution, i.e. all outputs some up to 1 and each single one is in the range $[0, 1]$.

Learning is in principle for each neuron similar to perceptron learning, the difference is that now on cannot simply train all neurons at once. For this one starts at the output layer, trains each neuron, then moves one layer to the front. For this layer one must now first calculate the relative errors, which they receive from the layer further at the output. After this calculation one can now train them and then move one layer back. One can iteratively continue this process, until one reaches the input layer. This process is called **Backpropagation**.
A pretty good pseudocode algorithm for Backpropagation can be found in ([2]).

Further the lecture distinguishes between three different types of learning, for this one must know the concept of epochs. An epoch is one forward and one backpropagation pass over all of the provided training samples (`https://www.kaggle.com/code/residentmario/full-batch-mini-batch-and-online-notebook`).

- Full-Batch: Computes the true gradient of the epoch, by computing the gradient of each training case independently, then summing the resultant vectors together.
- Mini-Batch: One splits a single epoch into multiple batches and updates the weight values for each such batch.
- Stochastic-Gradient-Descent: After each single data instance, the weights are updated (is termed as Stochastic as this is not the true gradient of all the data in the epoch).

One further advanced topic is MLP-regularisation, which tries to prevent overfitting of models. Can be implemented similar to rdige/lasso regression (just adding additional cost).

7.2 Deep-Learning

So what is Deep Learning? There are several definitions, in general it is something that is associated with neural networks, more precisely Multi-Layer perceptrons (see previous Section), which is composed of several smaller "learnable units/functions". Therfore the learnable function (the Neural-Network as a whole) is a stack of many simpler functions, that often have the same form.

Exact definitions for Deep-Learning are hard to give, one such definition simply says, that there must be multiple (≥ 2) hidden-layers, which would also include very shallow networks.
According to the lecture such networks are not seen as "deep", they do not provide an exact definition, but just state that the network must have "very many layers".

The main reason for "goind deep" is, that they perform well on certain practical tasks, such as Image Recognition. In theory "going deep" is not necessary, as a neural network with just one hidden layer (and arbtitrarily many nodes) can learn any function, but it is much harder for such a shallow structure to find a suitable network. Finding a deep one is up to now a more pracitcal approach.
Still such network are inherently hard to train, therefore one tries to reuse certain aspects of other deep-networks, such as the **architecture** or even the **parameters** via **Transfer Learning**.

In contrast to classical-machine-learning, Deep-Learning features also learning of the relevant features, how to process them and generally feature extraction (See section 8.1). A very often used class of networks for such a task is the class of *Convolutional Networks*. They typically consist of three types of layers:

- Convolutional Layer
  - Basically applies filters to an image.
  - Such filters are used for edge detection, sharpening, etc.
  - As the convolutional layer is learned, the filters are also learned.
  - In the lecture slided they provide a few typical filters.
  - Further this layer handles things like rotation of images, sizes, etc.
- Sub-Sampling Layer / Pooling step / Downsampling
  - Reduces the size of feature maps (so condences the information)
- Fully connected MLP
  - Is a fully connected MLP
  - Typically at the end of the Deep-Network, to provide classification.

In the lecture they quickly go over the following CNN architectures: *LeNet*, *AlexNet*, *ZFNet*, *VGG Net*, *GoogLeNet*, *ResNet*, *Densenet* and *SqueezeNet*.
To think of a new functioning network is very time and resource intensive, therefore reusing other architectures is highly recommended. One such way to do it is called Transfer Learning, which takes part of a pre-trained network for a different domain, for a different source task and adapts the rest of it for your task at hand. One can do this by cutting off the top layer(s) of a network and replace these with a supervised objective for a certain target domain.
The bottom n layers can either be frozen (i.e. not updated at all, recommended if one has a scarce database) or fine-tuned (i.e. updated, recommended if one has many labels). This combination is often better in practice than specialized classifiers.
One can also try a similar approach on unlabeled data, where the task is to train a model on just unlabeled data. Here one can take a pretrained network, cut off the upper part and do unsupervised learning on top of it (like clustering).

Further it is noted that one can visualize how individual layers work. This is in the context of interpretable AI, so one can see what is going on in the layesr. Another topic is data augmentation, which is important in the training process, as it strives to prevent the network from learning irrelevant facts. Lastly the lecture recommends to use the *RELU* activation function, as for some other functions the problem of a *weak gradient* can appear, e.g. take the Sigmoid function, if you look at it from a distance it looks like a heaviside function...
Another tip is to use dropout while learning, which is a technique that randomly removes a part of the neurons. And finally they discuss various alternative methods for the standard gradient-descent algorithm, with a focus of how to select the learning rate α. They present the *Stochastic Gradient Descent with Momentum* algorithm, which "builds up momentum" if the gradients keep staying in the same direction. Analogy is a ball which rolls down a hill.

7.3 Recurrent Neural Networks (RNN)

Have cycles in their neural network structure, which means, that they can operate over sequences of data, like sound. Often they have a sort of state (memory) and further the input and output size is variable.
Can architecturally be seen as a combination of recurrent layers, where each recurrent layer processes an input together with their output. Learning is basically done via backpropagation.

Long Short-Term Memory (LSTM) can store longer term memory, where at each step the old value can be kept or updated to a new value.
Gated Recurrent Units (GRUs) have similar performance but fewer parameters than LSTM. Other architectures are for instance auto-encoders, etc. etc.

8 Advanced Topics in ML

8.1 Feature Extraction

The following techniques are for "manual-feature-extraction" and can be sometimes ommitted for Deep-Learning.

8.1.1 Text

For text one can extract features by analysing the data. The lecture presents the Bag-oF-Words technique, which is done in the following steps:

1. Tokenization - Cut character sequence into word tokens (e.g. single word like "car" ord phrases like "a state-of-the-art solution")
2. Normalization - Map text and query term to same form (e.g. U.S.A to USA)
3. Stemming - Words with the same stem should sometimes be regarded as the same, e.g. "authorize", "authorization"
4. Stop words - Removal of "unnecessary" words like "to", "a",...
5. Then one can create a bar-og-words, i.e. removing the original text column from the data and adding as many columns as there are token vectors.

8.1.2 Image

Different techniques exist, like Edge detection, Corner detection, Scale-invariant feature transformation, filters Bag-of-visual-words.
For some of those things one can use a filter. A filter is represented as a filter-kernel, which is basically a matrix, which is then applied to the image via the mathematical process of folding.

8.1.3 Audio

Often a series of steps, which analyzes the data according to some transformations and spectrum analyzers.

8.2 Robustness/Adversarial ML

One of the current biggest problems for ML is it's robustness and the social/real-life implications of this. Take for example the Google-Photos bug, where Google-Photos classified a person of color as a Gorilla, which lead to an outcry.
Other attacks comprise of attacks on Image-Classification, such as when one fools a Image Classifier to predict a Coala as a Banana. Such attack are mostly *Evasion Attacks*, which try to fool the prediction steps by providing minimal perturbations of the input to achieve other outputs. There are several methods, that try to combat this/exploit this. Sometimes it is only necessary to change a single pixel to obtain completely ridiculous classifications, like classifying a horse as a frog. But this can also be dangerous, e.g. when a self driving car classifies a stop shield as a 90 km/h shield...

Possible attacks for a ML model include:

- Evasion attacks - Fool prediction steps, by minimal perturbations of the input
- Poisoning (Backdoor) attacks - Attack during the learning of the model, goal is to embed a specific pattern that can trigger the malicious behavior. Possible defense: Pruning the network, which might reduce accuracy but generalization may impŕove.
- Model inference/inversion - Determine if a person participated in a medical DB
- Model Extraction/Stealing - Attacker tries to learn model approximation in as few queries as possible
- Confidentiality attacks - Exposure of sensitive data
- Availability attacks - Disruption of critical service
- Integrity attacks - Unauthorized modification of data

8.3 Explainability of AI

Is currently a major research topic in AI, which tries to find out how to inform humans, how an AI arrived at it's decision. This is done via explanations, where it is inherently harder for some machine learning models than for others to explain something.
E.g. it is pretty straight forward for a decision tree to explain a certain decision, whereas for a Deep-Neural-Network it may not be the case.
The researchers in this field distinguish between three types of explainability:

- *Intrinsic*: Model-inherent (i.e. linear model or decision-tree)
- *Post-Hoc*: Extracting information from the model
- *Ex-ante*: Data statistics, bias in data, definition of task

And further different types of explanations are discussed:

- Feature statistics/visualizations
- Model internals (e.g. weights)
- Examples and counter-examples
- Proxy models: simpler, easier to understand surrogate model

Further it is pointed out, that the quality of explanations is also important, e.g. things like generalizations, truthfulness and contrastiveness (not just "why x?" but also "why x not y?") are important.
In the end they compare the explainability of different models to each other (Decision Trees best, Deep learning worst).

8.4 Privacy and AI

As for AI large amounts of data are needed and this data may contain sensitive information, privacy and privacy protection is a big concern. A few solutions exist (but further research is needed) in the following a few examples are provided:

- Data Sanitation/Sanitisation

  - Pseudonymisation - removal of directly identifiying information (like Name, Social-Security Number, etc.)
  - Anonymisation - Removal of Quasi Identifiers (QI) - QIs are defined as a set of data, that when combined can identify a single individual, therefore e.g. Birthdate, ZIP Code, gender and occupation can be seen as QIs, when combined together.
    * k-Anonymity: Generalisation of values

- Differential Privacy
  - Concept, that the risk to my privacy should not substantially increase as a result of participating in a statistical database.
  - For achieving differential privacy it is usually necessary to modify the dataset a little, which means adding some randomness or noise in some places.
  - Where to add the noise? Input- (before running the alg.), internal- (randomize internals of alg.), or Output-pertubation (after alg. has run)

- Federated Machine Learning
  - Data is shared among multiple parties, but all want to learn a common model
  - The data shall not be centrally aggregated
  - Different levels of federation:
    * Distributed learning - Data initially centralised, but computation distributed for efficiency
    * Federated learning - Data is distributed from the beginning but common model is learned (through privacy preserving mechanisms)
    * Decentralised learing - Data is distributed from the beginning and not shared. Common model without a central aggregator.

- Secure Computation
  - Trusted party assumption: Single point of failure
  - Data stays with the data owner - therefore data is not aggregated, retrieves only final models.
  - For secure computation: Question is, can we compute a function f in a secure way, while not trusting anyone? - Secure Multiparty Computation (SMPC), Homomorphic Encryption (Allows computation on encrypted data)

8.5 Statistical Significance Testing

When is a classifier "better" than another? The idea is to create a statistical significance test for this.
The H_0 is that the results of both classifiers are drawn from the same distribution. If the H_0 can be rejected, then one can state, that they are not from the same.
A simple test for this is the *McNemar's* test, which is based on a χ^2 test. The test statistic can

be computed according to the following formula (note that N_{01} denotes, the samples that the first classifier got wrong, but the second one correct and N_{10} the contrary):

$$\chi^2 = \frac{(N_{01} - N_{10})^2}{N_{01} + N_{10}}$$

After the computation one may have a look on the χ^2 table, or compute it with ones favorite statistics-program, and then reject/or not-reject it (**Before** one must set a rejection threshold, which is typically $\alpha = 0.05$, but other ones can also be set).
Larger datasets are preferable, as one can easier spot a difference between two models.

Another test one can use is the *paired t-Test*, which is a method used to check if the mean difference between pairs of measurements is zero or not.

References

[1] W.G. Macready D.H. Wolpert. "Coevolutionary free lunches". In: *IEEE Transac. on Evolutionary Computation* (2005), pp. 721–735.

[2] Peter Norvig Stuart J. Russel. *Artificial Intelligence: A mordern Approach*. eng. Third Edition. Pearson PLC, London: Pearson, 2010.