DAT200

Sammendrag.

Chapter 1

Supervised learning

□ Labeled data, direct feedbak, predict outcome/future.
Learn a model from labeled training data. Train on data where the desired output is already known.

■ Classification

Discrete labels, such as "yes"/"no".
The goal is to predict the categorical label ("yes"/"no") of new instances based on past observations.

■ Regression

The output is a continuous value, such as a percentage.

Reinforcement learning

□ Decision process, reward system, learn series of actions.
The goal is to develop a system that improves its performance based on interactions with the environment, through trial and error.

Unsupervised learning

□ No labels, no feedback, find hidden structure in data.
Explores the structure of the data to extract meaningful information, without knowing the outcome variable or reward function. **Clustering** is a common technique that creates subgroups of the data closely related. **Dimensionality reduction** can improve the computational performance, by reducing the data complexity while keeping the most relevant information.

Preprocessing

Changing the data to improve the model. Usually changing the shape of the data.
□ **Scaling** the data, by transforming the features within the range [0, 1] or a standard normal distribution with $mean = 0$ and $standard\ deviation = 1$, is also required by some algorithms in order to work.
□ Dimensionality reduction in order to remove highly correlated features and compressing the data into lower dimension, leads to faster learning and may lead to improved predictive performance.

Chapter 2

Artificial neuron

A simple binary classification (predicting 1 or -1) uses a decision function $\phi(z)$ that takes a linear combination of input values and corresponding weights to predict the output. $z = w_1x_1 + \cdots + w_mx_m$ and compares it to a threshold θ in the function

$$\phi(z) = \begin{cases} 1 & z \geq \theta \\ -1 & otherwise. \end{cases}$$

To simplify this, we introduce an extra weight w_0, called the **bias unit**, by bringing the threshold θ to the other side, such that $z = w_0x_1 + w_1x_1 + \cdots + w_mx_m = \mathbf{w}^T\mathbf{x}$ where $x_0 = 1$, which gives

$$\phi(z) = \begin{cases} 1 & z \geq 0 \\ -1 & otherwise. \end{cases}$$

Perceptron

1. Initialize the weights (close to zero randum numbers)
2. For each training sample, $\mathbf{x}^{(i)}$: (a) Compute the output value $\hat{y}$. (b) Update the weights.

The weights are updated by Δw_j, where the $\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$. η is the **learning rate**, typically between 0 and 1, and decides how big the weight updates should be.

Cost function

The cost function, J, in the case of Adaline to learn the weights is the sum of squared errors (SSE) between the calculated outcome and the true class label:

$$J(\mathbf{w}) = \frac{1}{2}\sum_i [y^{(i)} - \phi(z^{(i)})]^2$$

This function is convex, and we can therefore optimize by calculating the **gradient descent** $\nabla J(\mathbf{w})$, such that $\Delta\mathbf{w} = -\eta\nabla J(\mathbf{w})$. Therefore $\Delta w_j = \eta\sum_i [y^{(i)} - \phi(z^{(i)})]x_j^{(i)}$

Chapter 3

Logistic regression

Despite its name, logistic regression is a model for **binary classification**, not regression. Logistic regression can also be generalized to multiclass settings, known as multinomial logistic- or softmax regression. Or via One-versus-Rest technique.

In logreg the odds, $\frac{p}{(1-p)}$, where p is the probability for the wanted outcome, is used. The function $logit(p) = log\frac{p}{(1-p)}$ is defined, and further, the logistic **sigmoid function** $\phi(z) = \frac{1}{1+e^{-z}}$. With $z = \mathbf{w}^T\mathbf{x}$. The sigmoid function ($\phi(z)$) is between 0 and 1, and we can introduce the prediction

$$\hat{y} = \begin{cases} 1 & z \geq 0 \\ 0 & otherwise. \end{cases}$$

The corresponding cost function for logreg therefore becomes

$$J(\mathbf{w}) = -\sum_i y^{(i)}log[\phi(z^{(i)})] + (1 - y^{(i)})log[1 - \phi(z^{(i)})]$$

$y = 1$: the cost is high for small $\phi(z)$. $y = 0$: the cost is high for large $\phi(z)$.
□ This means that wrong predictions will be penalized greater.

■ Overfitting

□ When a model performs well on the training data but does not generalize well to unseen data (test data) which is the same as saying the model has high variance.

The reason for overfitting can be that the data has too many parameters, leading to a complex model that tries to accomodate all the features.

■ Underfitting

High bias, or underfitting, is the cause of a too simple model that is unable to capture the pattern in the training data well which in turn leads to low performance on unseen data.

Regularization

□ Handels collinearity, filters out noise and prevents overfitting.
One way to find a good bias-variance tradeoff (minimum over-/underfitting) is to tune the complexity of the model via regularization.
Penalizes extreme weights by introducing an additional bias.

■ L2

L2-regularization introduces a bias

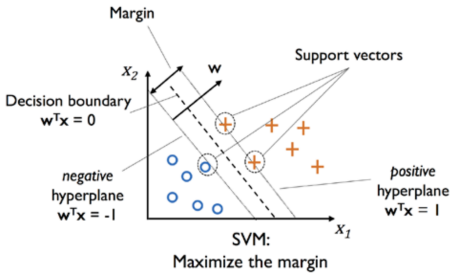
$$\frac{\lambda}{2}||\mathbf{w}||^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

where λ is the **regularization parameter** (which is the inverse of C).
The cost function for L2-regularization therefore becomes

$$J(\mathbf{w}) = -\sum_{i=1}^{n}\left[-y^{(i)}log[\phi(z^{(i)})] - (1-y^{(i)})log[1-\phi(z^{(i)})]\right] + \frac{\lambda}{2}||\mathbf{w}||^2$$

SVM (support vector machine)

With perceptron we minimized misclassification errors. With SVMs our optimization objective is to maximize the margin. The margin being the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called **support vectors**.



The reason for trying to maximize the margin is to get the smallest possible generalization error. The algorithm say that all negative-class examples should fall one one side of the negative hyperplane, and all the positive-class examples should fall behind the positive hyperplane: $y^{(i)}(w_0 + \mathbf{w}^T\mathbf{x}^{(i)}) \geq 1$ for all i. Which is the same as minimizing $\frac{1}{2}||\mathbf{w}||^2$.

■ Slack variable

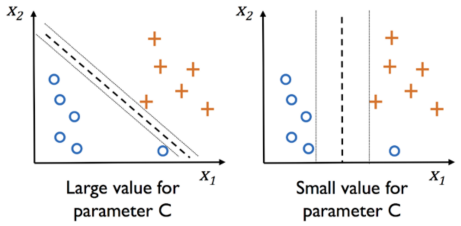
When dealing with data that isnt linearly separable, the slack variable ξ is introduced.

$$w_0 + \mathbf{w}^T\mathbf{x}^{(i)} \geq 1 - \xi^{(i)} if y^{(i)} = 1$$
$$w_0 + \mathbf{w}^T\mathbf{x}^{(i)} \leq -1 + \xi^{(i)} if y^{(i)} = -1$$

and the new bias is

$$\frac{1}{2}||\mathbf{w}||^2 + C\left(\sum_i \xi^{(i)}\right)$$

We can therefore control the penalty for misclassification through C.



■ **Kernel**

In order to solve nonlinear problems we can **kernelize** SVMs.
☐ Project data onto a higher-dimensional space via a mapping function ψ where the data is linearly separable. Two-dimensional data can be mapped into three dimensional space, where the classes are separable.
With high-dimensional data this is computationally expensive, so here we can use the **kernel trick**. This is done by defining a kernel function

$$\kappa(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T \phi(x^{(j)})$$

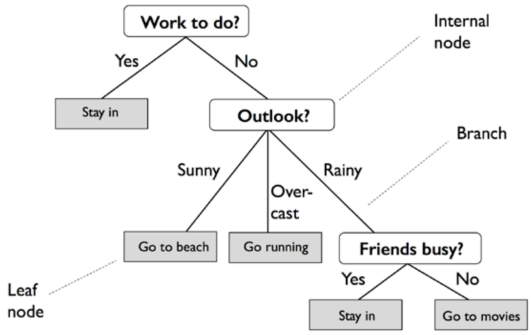
The most common kernel is the **RBF (radial basis function)** also called the Gaussian kernel:

$$\kappa(x^{(i)}, x^{(j)}) = exp\left[-\frac{||x^{(i)} - x^{(j)}||^2}{2\sigma^2}\right] = exp\left[-\gamma||x^{(i)} - x^{(j)}||^2\right]$$

where γ is a hyperparameter. With a high value leading to overfitting.

Decision tree

☐ Interprets the data, by breaking down our data by asking it a series of questions.



The algorithm starts at the tree root and splits the data on the feature that results in the largets **information gain** (IG). The splitting procedure can be repeated at each child node until the leaves are pure. Meaning that the training examples at each node each belong to the same class. In practise this can lead to a deep tree, resulting in overfitting, and we therefore set a limit for the max depth (we **prune** the tree).
In order to split the nodes at the most informative features, we define an IG-function which we want to maximize.

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Where f is the feature to perform the split; D_p and D_j are the dataset of the parent and jth child node; I is the **impurity** measure; N_p is the total number of training examples at the parent node; and N_j is the number of examples in the jth child node.

There are different impurity functions: **Gini** (I_G), **entropy** (I_H) and the classification error (I_E).

$$\textbf{Gini}: \ I_G(t) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

$$\textbf{Entropy}: \ I_H(t) = -\sum_{i=1}^{c} p(i|t) log_2 p(i|t)$$

$$\textbf{Error}: \ I_E(t) = 1 - max\{p(i|t)\}$$

Where $p(i|t)$ is the proportion of the examples that belong to class i for a particular node t.
The entropy is therefore 0 if all examples at a node belong to the same class, and maximal if we have a uniform class distribution. The Gini impurity is maximal if the classes are perfectly mixed.
In a binary class setting, the entropy is 0 if $p(i = 1|t) = 1$ or $p(i = 0|t) = 0$. If the classes are distributed uniformly with $p(i = 1|t) = 0.5$ and $p(i = 0|t) = 0.5$ the entropy is 1. The Gini impurity is maximal if the classes are perfectly mixed, for example in a binary class setting ($c = 2$), $I_G(t) = 0.5$.

Random forest

□ Creates k decision trees (sample size n, with d random features (without replacement), splitting until (for instance) IG is maximised) and classifies by a **majority vote** of the k trees in the "forest".

KNN (K-nearest neighbors)

Checks what class the k nearest neighbors are and classifies by a majority vote.
The distance is calculated with

$$d(x^{(i)}, x^{(j)}) = \sqrt[p]{\sum_k |x_k^{(i)} x_k^{(j)}|^p}$$

Where it is the Euclidean distance if $p = 2$ or the Manhattan distance if $p = 1$.

Chapter 4

Missing values

Either drop rows or columns, or impute missing values (where **mean imputation** is the most common).

■ Transformer

A transformer is an estimator that has the methods fit and $transform$.
The fit method is used to learn the parameters from the training data, and the $transform$ method uses those parameters to transform the data.

■ Estimator

An estimator has a *predict* method, but can also have a $transform$ method.

Categorical data

■ Ordinal

Categorical values that can be sorted or ordered (XL > L > M).

■ Nominal

Categorical values that cannot be sorted or ordered (green, blue, red).

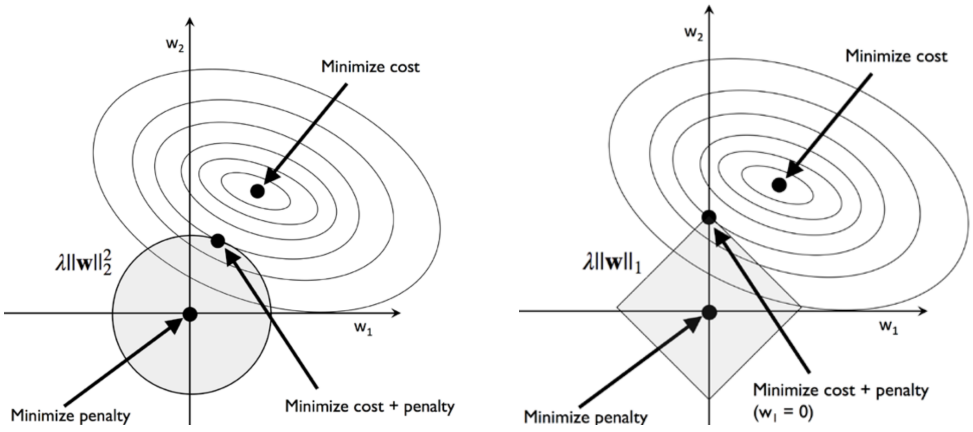
Feature importance

■ Regularization

L2- and L1 regularization is used as penalties against model complexity. Regularization penalizes large individual weights.

$$\textbf{L2}: \ ||\mathbf{w}||_2^2 = \sum_{j=1}^{m} w_j^2$$

$$\textbf{L1}: \ ||\mathbf{w}||_1 = \sum_{j=1}^{m} |w_j|$$



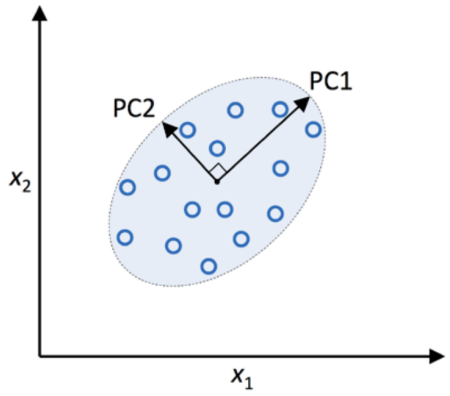
(a) L2 (b) L1

We see that for L1, the optimal (minimum) cost + penalty renders w_1 useless ($= 0$). Since the contours of L1-regularization is sharp, it is likely that the optimum is located on the axes, which encourages **sparsity**.

Chapter 5

PCA (principal component analysis)

☐ Unsupervised data compression.
Maximises variance, and reduces dimensionality to n orthogonal principal components.



Before PCA:

1. Standardize the d-dimensional dataset.
2. Construct the covariance matrix.
3. Decompose the covariance matrix to its eigenfunctions.
4. Select k largest eigenfunctions by eigenvalue.
5. Construct a projection matrix $\mathbf{W}$ from the k eigenfunctions.
6. Transform the d-dimensional dataset $\mathbf{X}$, using $\mathbf{W}$, to obtain the k-dimensional feature subspace.

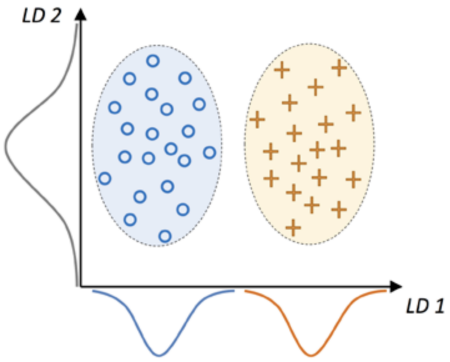
   $\mathbf{XW} = \mathbf{X'}$

The covariance between to features x_j, x_k are:

$$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^{n} \left[\left(x_j^{(i)} - \mu_j \right) \left(x_k^{(i)} - \mu_k \right) \right]$$

LDA (linear discriminant analysis)

☐ Supervised dimensionality reduction for maximizing class separability.
Similar to PCA, but instead of finding orthogonal principal components, LDA finds the feature subspace that optimizes class separability.



The number of linear discriminants is at most $c-1$ (c is the amount of classes).
Before LDA:

1. Standardize the d-dimensional dataset.
2. Compute the d-dimensional mean vector for each class.
3. Construct the between-class scatter matrix $\mathbf{S}_B$, and the within-class scatter matrix $\mathbf{S}_W$.
4. Compute eigenfunctions of the matrix $\mathbf{S}_W^{-1}\ \mathbf{S}_B$
5. Select k largest eigenfunctions by eigenvalue.
6. Construct a $d \times k$-dimensional transformation matrix $\mathbf{W}$, where the eigenvectors are the columns of the matrix.

   $\mathbf{XW} = \mathbf{X'}$

The number of linear discriminants is at most $c-1$ (c is the amount of classes).

■ **Within-class scatter matrix, $\mathbf{S}_W$**

$$\mathbf{S}_W = \sum_{i=1}^{c} \mathbf{S}_i = \sum_{i=1}^{c} \left(\sum_{x \in D_i} (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T \right)$$

Where $\mathbf{S}_i$ is the scatter matrices of each class i.

■ **Covariance matrix**

A normalized version of the scatter matrix.

$$\Sigma_i = \frac{1}{n_i} \mathbf{S}_i = \frac{1}{n_i} \sum_{x \in D_i} (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T$$

■ **Between-class scatter matrix, $\mathbf{S}_B$**

$$\mathbf{S}_B = \sum_{i=1}^{c} n_i (\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$$

KPCA (kernel principal component analysis)

□ Nonlinear dimensionality reduction.
Kernel function ϕ: $\mathbb{R}^d \to \mathbb{R}^k$ where $(k \gg d)$, we use the kernel trick so that we do not have to calculate the eigenvectors explicitly $\kappa(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T \phi(x^{(j)})$.

1. We compute the kernel (similarity) matrix $\mathbf{K}$:

$$\mathbf{K} = \begin{bmatrix} \kappa(x^{(1)}, x^{(1)}) & \dots & \kappa(x^{(1)}, x^{(n)}) \\ \vdots & \ddots & \vdots \\ \kappa(x^{(n)}, x^{(1)}) & \dots & \kappa(x^{(n)}, x^{(n)}) \end{bmatrix}$$

2. We center the kernel matrix:

   $\mathbf{K'} = \mathbf{K} - \mathbf{1}_n\mathbf{K} - \mathbf{K}\mathbf{1}_n + \mathbf{1}_n\mathbf{K}\mathbf{1}_n$

   Where $\mathbf{1}_n$ is an $n \times n$-dimensional matrix where all values are equal to $\frac{1}{n}$.

3. We collect the k eigenvectors of the centered kernel matrix based on their corresponding eigenvalues.

Chapter 6

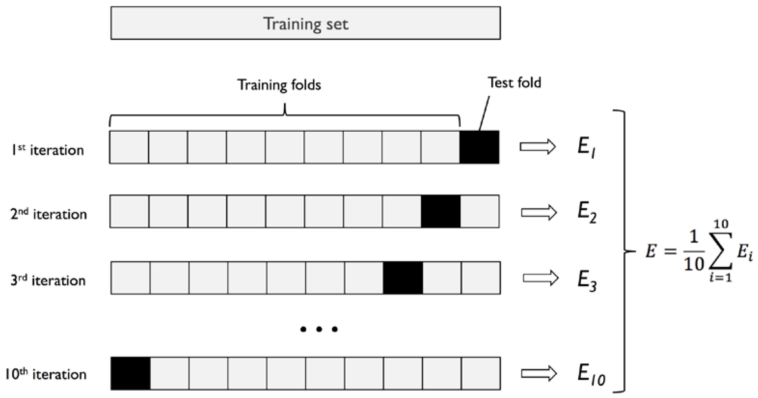
A "Pipeline" combines preprocessing, feature selection and predicting into one function.

K-fold cross-validation

The training data is split into k folds without replacement. $k-1$ are used for training, and 1 is used for performance evaluation. This is repeated k times, and the average performance is calculated.
For small training sets k should be large. For instance "LLOCV" (leave-one-out-cross-validation), where $k = n$, so that only one training example is used for testting each iteration.

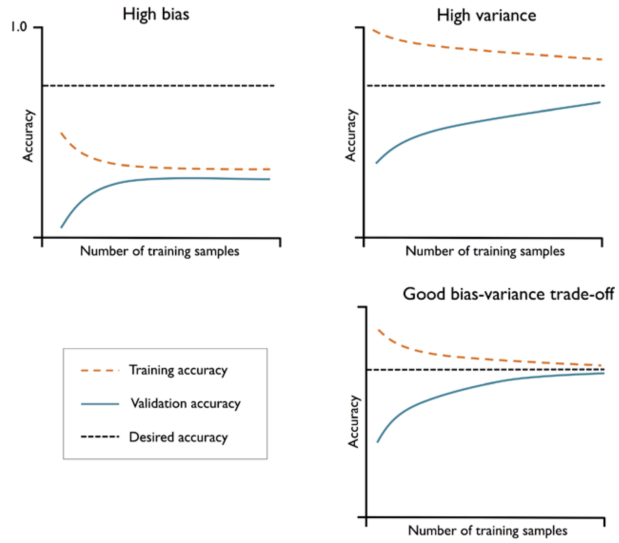
■ **Stratified K-fold**

The class label proportions are preserved in each fold to ensure that each fold is representative of the training dataset.



Learning curves

High bias refers to underfitting. To fix this, one could increase the number of parameters of the model. High variance indicates a large gap petween training- and cross-validation accuracy. To fix this, one could reduce the complexity of the model or increase the regularization parameter.



Validation curves

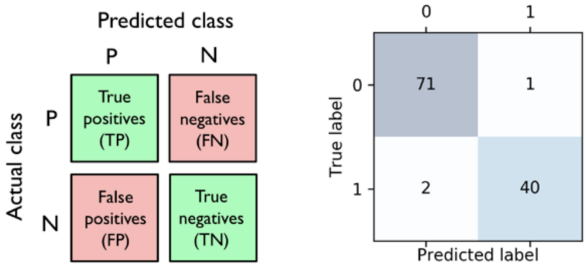
Adresses under- and overfitting. Instead of plotting the training and test accuracies as a function of the sample size, we vary the hyperparameters and plot the accuracies as a function of them.

Grid search

A grid search goes through the wanted hyperparameter combinations, and obtain the optimal combination.

Confusion matrix

☐ A square matrix that reports the counts of **true positive**, **true negative**, **false positive** and **false negative**.
True: correctly classified of the predicted. False: wrongly classified of the predicted.



(a) (b)

In the example (b), 71 of the examples belonging to class 0 (TN) and 40 of the examples belonging to class 1 (TP) is correctly classified. The model incorrectly misclassified two examples from class 1 as class 0 (FN) and one example from class 0 as class 1 (FP).

$$Error = \frac{FP + FN}{FP + FN + TP + TN} \quad \text{and} \quad Accuracy = 1 - Error$$

$$\textbf{False positive rate} : FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$\textbf{True positive rate} : \; TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

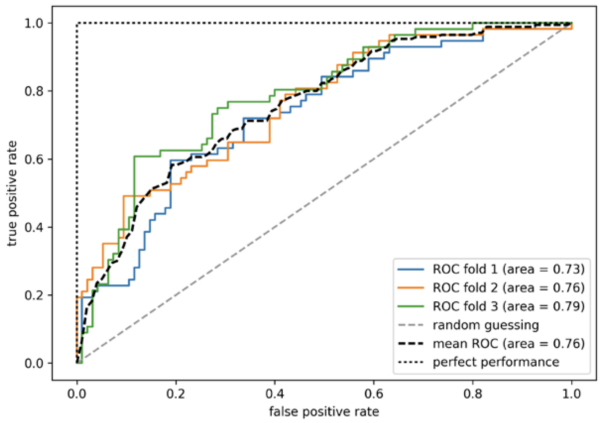
$$\textbf{Precision} : PRE = \frac{TP}{TP + FP}$$

$$\textbf{Recall} : REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

$$F1 = 2\frac{PRE \; \times REC}{PRE + REC}$$

ROC (receiver operating characteristic)

Tool to select models for classification based on their performance with respect to the FPR and TPR. The diagonal of a ROC graph can be interpreted as *random guessing*, and models that fall below the diagonal are considered as worse than random guessing.
A perfect model would fall into the top-left corner of the graph with a TPR of 1 and an FPR of 0.



The ROC AUC (area under curve) is used to characterize the performance of a model. The greater the area under the curve the better the model.

■ **For multiclass classification**

One-vsersus-all
□ **micro-average** is calculated from the individual TPs, TNs, FPs, and FNs of the system. For a k-class system, the precision score is:

$$PRE_{micro} = \frac{TP_1 + \ldots + TP_k}{(TP_1 + \ldots + TP_k) + (FP_1 + \ldots + FP_k)}$$

Each **instance** is equally important.
□ **macro-average** is the average scores of the different systems. The precision is therefore:

$$PRE_{macro} = \frac{PRE_1 + \ldots + PRE_k}{k}$$

Each **class** is equally important.

■ **Class imbalance**

If a training-dataset is 90% of one class and 10% of the other, one could use this, and get a 90% accuracy, without the model learning anything about the data.
A workaround for this can be done by implementing one or more of these:

a. Penalizing wrong predictions in the minority class greatly. (In scikit-learn: class_weight='balanced')

b. Upsample the minority class and downsample the majority class.

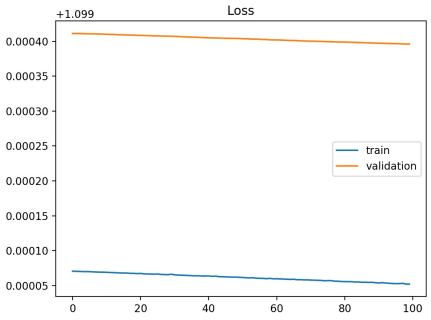
   Draw random samples from the minority class with replacement the same number of times as majority instances, and add them (duplicate) so that there is an equal amount of minority and majority class instances.

c. Downsample majority class.

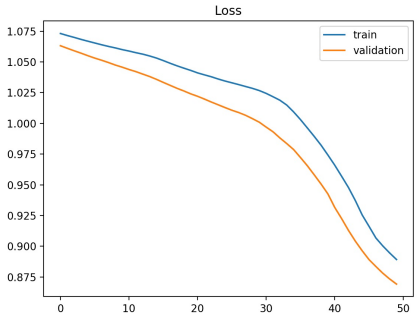
   Opposite of (b.)

Learning curve

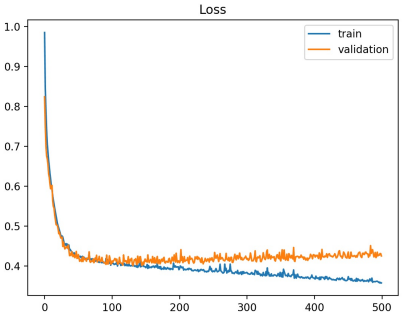
Line plot of learning (y-axis) over experience (x-axis). It is common to use a score that is minimizing, such as loss or error, where smaller numbers indicate more learning, where "0" is equal to no mistakes (perfect model).
□ **Training curve**: Learning curve from the training dataset which gives an idea of how well the model is learning.
□ **Validation curve**: Learning curve from a validation dataset which gives an idea of how well the model is generalizing.



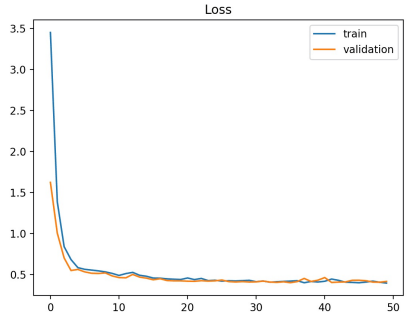
(a) Underfit



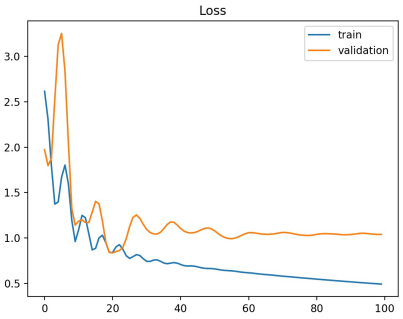
(b) Underfit



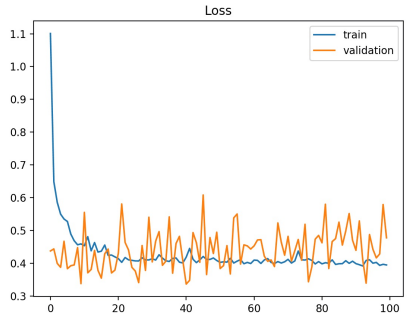
(c) Overfit



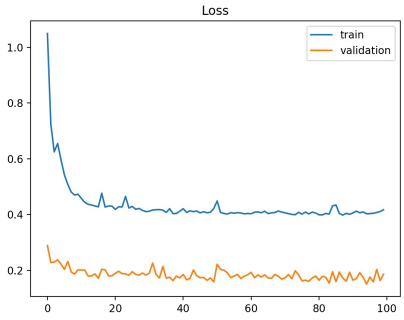
(d) Good fit



(e) Unrepr. training set



(f) Unrepr. val. set



(g) Val. easier than train

■ **Underfit**

An underfit learning curve has either a big gap between train- and validation curves (a), or a rapid decrease towards the end (which signals that there is more to be gained from further training) (b).

■ **Overfit**

Does not generalize well. An overfit learning curve is characterized by either the training loss continues to decrase with expreience (c), or if the validation curve decreases to a point and begins to increase again.

■ **Good fit**

The training- and validation curves decreases to a point of stability with a minimal gap between (d).
Continued training of a good fit will likely lead to an overfit.

■ **Special cases:**

Training set too small and/or unrepresentative (e).
Validation set too small and/or unrepresentative (f).
Validation set easier to predict than training set (g).

Chapter 7

Ensemble learning combines different weak classifiers into a robust model. Random Forest is a type of enseble learning, where descision tree is the "weak classifier".
There are different methods for combining the weak classifiers. A majority vote is common, or a weighted sum. Where prior knowledge about the different weak classifiers is used in order to provide the weight of each prediction (some classifiers are more trustworthy than others). Example: rather trust one doctor with 40 years of experience, than five high-scoolers.

Bagging

Technique that is closely related to majority vote. Instead of training the classifiers on the whole train set, we select a subset (with replacement) for each classifier to be fitted on. The predictions are then combined using for instance majority voting.

AdaBoost (adaptive boosting)

In boosting, the ensemble usually consist of very simple base classifiers (weak learners), which is quite close to random guessing. Boosting is used to focus on training examples that are hard to classify.

1. Draw a random subset of training examples d_1, without replacement from the training dataset D, to train a weak learner C_1.
2. Draw a second training subset d_2, without replacement from D and add 50 % of the examples that were previously misclassified to train a weak learner C_2.
3. Find the training examples d_3, which C_1 and C_2 disagree upon, and use these to train a third weak learner C_3.
4. Combine C_1, C_2 and C_3 via majority voting.

The algorithm looks as follows:

1. Set $\mathbf{w}$ to uniform weights (with $\sum_i w_i = 1$).
2. for j in m boosting rounds, do:

a. Train a weak learner: C_j = train(**X,y,w**).
b. Predict class labels: $\hat{\mathbf{y}}$ = predict(C_j,**X**).
c. Compute weighted error rate: $\epsilon = \mathbf{w} \cdot (\hat{\mathbf{y}} \neq \mathbf{y})$.
d. Compute coefficient: $\alpha_j = 0.5 \log \frac{1-\epsilon}{\epsilon}$
e. Update weights: $\mathbf{w} = \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$.
f. Normalize weights to sum to 1: $\mathbf{w} = \mathbf{w} / \sum_i w_i$.

3. Compute final prediction: $\hat{\mathbf{y}} = (\sum_{j=1}^{m} [\alpha \times \text{ predict}(C_j, \mathbf{X})] > 0)$.

Chapter 10

Regression analysis predicts continuous targets. Aims to minimize SSE (Sum of Squared Error).

RANSAC (RANdom SAmple Consensus)

Robust with outliers.

1. Select (a random number of) inliners and fit the model. (Points the regression line should go through)
2. Test all other data points against the fitted model (errors), and add those points within a user-given tolerance to the inliners.
3. Refit using all inliners.
4. Esitmate the error of the fitted model.
5. Terminate the algorithm if the performance meets a threshold, or if a fixed number of iterations are reached. If not, repeat all steps.

■ R2 (coefficient of determination)

The MSE (Mean Squared Error) is given by

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2$$

and

$$R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{MSE}{Var(y)}$$

which describes how good the model fits the data. $R^2 = 1$ is the best possible value, which says that the MSE is 0, and the model is therefore perfect.

■ Ridge regression

L2-penalized model, where the penealty is:

$$\lambda ||\mathbf{w}||_2^2 = \lambda \sum_{j=1}^{m} w_j^2$$

■ LASSO regression

L1-penalized model, where the penealty is:

$$\lambda ||\mathbf{w}||_1 = \lambda \sum_{j=1}^{m} |w_j|$$

■ **ElasticNet regression**

Combination of L2 and L1 penalty:

$$\lambda_1||\mathbf{w}||_2^2 + \lambda_2||\mathbf{w}||_1 = \lambda_1 \sum_{j=1}^{m} w_j^2 + \lambda_2 \sum_{j=1}^{m} |w_j|$$

Chapter 11

Unsupervised learning (clustering)

K-means

Creates K clusters.

1. Randomly pick k centeroids as initial cluster centers.
2. Assign each point to the nearest centeroid.
3. Move the centeroid to the center of the points assigned to it.
4. Repeat steps 2. and 3. until convergence or max-iterations is reached.

K-means++

Places the K centeroids far away from each other. Yields more consistent results.

Soft/fuzzy clustering

Assigns probabilities to points, based on their position with respect to the centeroids. A point close to a centeroid is most likely that cluster, while an outlying point has some uncertainty as to which cluster it belongs to.

Elbow method

A method to find the optimal number of clusters.
Uses within-cluster SSE, to estimate the optimal number of clusters. From the plot (# clusters x SSE) we choose the # of clusters where the SSE "breaks"/converges.

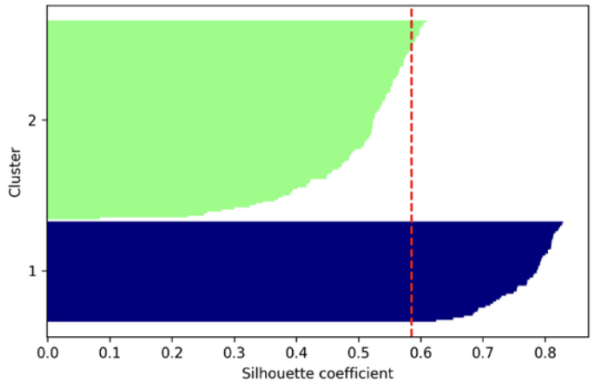
Silhouette plots

To calculate the silhouette coefficient:

1. Calculate the cluster cohesion, $a^{(i)}$, as the average distance between an example, $x^{(i)}$, and all other points in the same cluster.
2. Calculate the cluster separation, $b^{(i)}$, from the next closest cluster as the average distance between the example, $x^{(i)}$, and all examples in the nearest cluster.
3. Calculate the silhouette, $s^{(i)}$, as the difference between cluster cohesions and separation divided by the grater of the two:

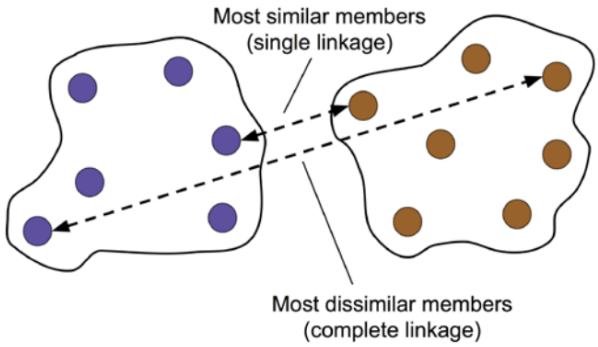
$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{max\{b^{(i)}, a^{(i)}\}}$$

The coefficients is in the range -1 to 1, and is 0 if the cluster separation and cohesion are equal ($b^{(i)} = a^{(i)}$).
The coefficient is ideally 1.
If the silhouettes are similar to each other, the clustering is good. If it however looks like this, it is bad clustering:



Hierarcical tree

Grouping clusters in bottom-up fashion. The most common algorithms are single linkage or complete linkage:



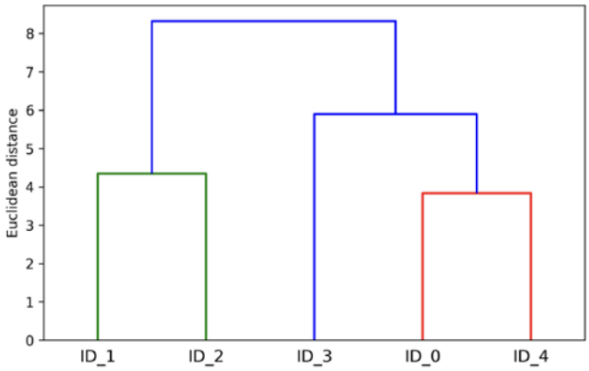
The two closest clusters are merged, until a single cluster is left.
Incorrect code:

```
row_clusters = linkage(row_dist, method="complete", metric="euclidean")
```

Correct code:

```
row_clusters = linkage(df.values, method="complete", metric="euclidean")
```

```
row_clusters = linkage(pdist(df, metric="euclidean"), method="complete")
```



■ **DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**

"Infects" points within a given radius ϵ. Points with $MinPts$ or more neighbours are **core points**, and points with fewer neighbours are **border points**. Points which are neither, are **noise points**.