ETH Introduction to Machine Learning
Lecture Notes

Michael Aerni
aernim@student.ethz.ch

September 2019
(updated April 10, 2020)

Contents

Preface

This summary was created during the spring semester 2019 "Introduction to Machine Learning" lecture by Prof. Dr. Krause at ETH Zurich. It is based on the material taught during the lectures and tutorials.

The content should be mostly complete in terms of material. However, there are still quite a few open points, usually marked with a TODO in the text. While most TODOs are placeholders for derivations or examples, a few places were unclear and thus not very coherent.

In general, the goal of this summary was personal use. Therefore, there may be a few contentual and many typographical (and grammatical) errors. If you encounter substantial issues, feel free to write to me via aernim@student.ethz.ch. However, I take no responsibility for the content of this summary, decline any liability and do not claim the content to be my own. May this help you during your studies.

Chapter 1

General

1.1 Terminology

The *loss* $\ell(f(\boldsymbol{x}_i), \boldsymbol{y}_i)$ is defined on a single data point, the *cost* on the whole data set. The cost might also include penalty terms.

The *objective* refers to any function optimised during training.

Regression has the goal to predict real-valued labels/vectors, in general a mapping $f : \mathbb{R}^d \to \mathbb{R}$.

In *classification*, the target variable Y is discrete, i.e. categorical.

In *supervised learning*, data is available as input-output pairs.

A *decision rule/hypothesis* is a rule to assign a class to a sample.

The *hypothesis / hypothesis class* is the function / class of functions used to fit data.

A *surrogate loss* is used whenever we want to optimise a loss which is intractable. The surrogate loss approximates the desired loss during optimisation. The target loss is used during evaluation.

Parametric models have a constant number of parameters. Nonparametric models grow in complexity with the size of data. Nonparametric models are potentially much more expressive but also more computationally complex.

TODO: Difference between derivative and gradient, especially column/row vector.

1.2 Gradient Descent

Let $\boldsymbol{w}_0 \in \mathbb{R}^d$ be an arbitrary starting point, η_t the learning rate at time t and $\hat{R}(\boldsymbol{w})$ be a differentiable function which should be minimised. For $t = 1, 2, \ldots$ gradient descent calculates

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \nabla \hat{R}(\boldsymbol{w}_t)$$

Under mild assumptions, if the step size is sufficiently small, gradient descent converges to a stationary point. Thus, for convex objectives, it finds the optimal solution (if one exists).

If the step size is chosen too small, convergence may take very long. If the step size is too large, the objective might oscillate or even diverge. The step size can also be chosen adaptively.

The line search heuristic works as follows: Let $g_t = \nabla \hat{R}(\boldsymbol{w}_t)$. Pick $\eta_t \in \arg\min_{\eta \in (0,\infty)}(\hat{R}(\boldsymbol{w}_t - \eta \cdot g_t))$, i.e. the step size which minimises the objective w.r.t. the current gradient.

The bold driver heuristic works as follows:

- If the function decreases, increase the step size:

$$\hat{R}(\underbrace{\boldsymbol{w}_t - \eta_t g_t}_{\boldsymbol{w}_{t+1}}) < \hat{R}(\boldsymbol{w}_t) \Rightarrow \eta_{t+1} = \eta_t \cdot c_{acc}$$

- If the function increases, decrease the step size:

$$\hat{R}(\boldsymbol{w}_t - \eta_t g_t) > \hat{R}(\boldsymbol{w}_t) \Rightarrow \eta_{t+1} = \eta_t \cdot c_{dec}$$

Stochastic gradient descent uses only a single sample per iteration. The sample is picked uniform at random (with replacement) from the dataset, used to compute the gradient and to update the weights.

Stochastic gradient descent is guaranteed to converge under mild assumptions if $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$.

Using only a single sample at a time is computationally inefficient and might have large variance in the gradient estimate. It might thus lead to slow convergence.

Using mini-batch SGD reduces variance. It uses mini-batches and averages the gradients with respect to multiple randomly selected points.

1.3 Model Selection

Underfitting happens if the model is too simple, overfitting if the model is too complex.

As the model complexity increases, the training error usually decreases. However, the prediction error usually decreases up to a point, and then starts increasing again.

Risks

We usually assume that the data set is generated independently and identically distributed (i.i.d.) from some unknown distribution P, $(\boldsymbol{x}_i, y_i) \sim P(\boldsymbol{X}, Y)$. The i.i.d. assumption can be invalid, e.g. if

- Time series data
- Spatially correlated data
- Correlated noise

The goal is then to minimise the expected error (true risk) under P. The empirical risk is used to estimate the true risk.

In the following, $\ell(y, \hat{y})$ is a loss function and $\hat{y} = f(\boldsymbol{x}; \theta)$ is a prediction.

The population risk/true risk/expected error under P is

$$R(\boldsymbol{w}) = \int P(\boldsymbol{x}, y)\ell(y, \hat{y})d\boldsymbol{x}dy = \mathbb{E}_{\boldsymbol{x},y}[\ell(y, \hat{y})]$$

The true risk on a sample data set D is

$$\hat{R}_D(\boldsymbol{w}) = \frac{1}{|D|} \sum_{(\boldsymbol{x},y)\in D} \ell(y, \hat{y})$$

Under the i.i.d. assumption, according to the law of large numbers, $\hat{R}_D(\boldsymbol{w}) \to R(\boldsymbol{w})$ for any fixed $\boldsymbol{w}$ as $|D| \to \infty$.

Empirical risk minimisation on training data D is finding $\hat{\boldsymbol{w}}_D = \arg\min_{\boldsymbol{w}} \hat{R}_D(\boldsymbol{w})$. Ideally, we want to solve $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} R(\boldsymbol{w})$. Generalisation refers to a model which extracts all relevant information from the training data but also performs well on unseen data. TODO: What is the generalisation error? Is it $R(\hat{\boldsymbol{w}}_D)$?

For learning via empirical risk minimisation, uniform convergence is needed:

$$\sup_{\boldsymbol{w}} |R(\boldsymbol{w}) - \hat{R}(\boldsymbol{w})| \to 0 \text{ as } |D| \to \infty$$

This is not implied by the law of large numbers and depends on the model class. For example, if "bad" points converge slower than "good" points, the generalisation error might increase with increasing amounts of data.

In general, it holds that

$$\mathbb{E}_D[\hat{R}_D(\hat{\boldsymbol{w}}_D)] \leq \mathbb{E}_D[R(\hat{\boldsymbol{w}}_D)]$$

Thus, when evaluating performance on the training data, an overly optimistic estimate is achieved.

The general idea is thus to use two independent data sets $D_{train} \sim P$ and $D_{test} \sim P$. Optimisation happens on the training set, i.e.

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \hat{R}_{train}(\boldsymbol{w})$$

and evaluation on the test set

$$\hat{R}_{test}(\hat{\boldsymbol{w}}) = \frac{1}{|D_{test}|} \sum_{(\boldsymbol{x},y)\in D_{test}} (y - \hat{\boldsymbol{w}}^T\boldsymbol{x})^2$$

Then,

$$\mathbb{E}_{D_{train},D_{test}}[\hat{R}_{test}(\hat{\boldsymbol{w}}_{D_{train}})] = \mathbb{E}_{D_{train}}[R(\hat{\boldsymbol{w}}_{D_{train}})]$$

The test error itself is random. The variance increases for more complex models. Thus, using a single test set creates bias.

Cross-Validation

Using multiple test sets wastes data. Thus, for each candidate model m and $i = 1, \ldots, k$:

1. Split $D = D^{(i)}_{train} \uplus D^{(i)}_{val}$
2. Train model: $\hat{\boldsymbol{w}}_{i,m} = \arg\min_{\boldsymbol{w}} \hat{R}^{(i)}_{train}(\boldsymbol{w})$
3. Estimate error: $\hat{R}^{(i)}_{m} = \hat{R}^{(i)}_{val}(\hat{\boldsymbol{w}}_{i,m})$

The model can then be selected as

$$\hat{m} = \arg\min_{m} \frac{1}{k} \sum_{i=1}^{k} \hat{R}^{(i)}_{m}$$

Monte Carlo cross-validation picks a training set of given size uniformly at random and validates on the remaining points. The prediction error is then estimated over multiple random trials.

k-fold cross-validation partitions the training set into k folds. $k-1$ are used for training, 1 for validation. The prediction error is then estimated as the mean validation error while varying the validation folds. $k = 1$ is called leave-one-out cross-validation (LOOCV).

For large enough k, the cross-validation error estimate is very nearly unbiased.

If k is picked too small, there is little data for training and a risk to underfit the training set and overfit the test set. If k is picked too large, performance is usually better but the computational complexity increases. In practice, $k = 5$ or $k = 10$ often works well.

However, the error on the validation set will usually also be underestimated. If a model is selected, it is usually retrained with the full data set.

1.4 Standardisation

Standardisation ensures that each feature has zero mean and unit variance.

Let $\tilde{x}_{i,j}$ be the j-th feature of the i-th data point. Then

$$\tilde{x}_{i,j} = \frac{x_{i,j} - \hat{\mu}_j}{\hat{\sigma}_j}$$
$$\hat{\mu}_j = \frac{1}{n}\sum_{i=1}^{n} x_{i,j}$$
$$\hat{\sigma}_j^2 = \frac{1}{n}\sum_{i=1}^{n} (x_{i,j} - \hat{\mu}_j)^2$$

1.5 Feature Selection

There are various reasons to not use all available features, e.g. because of

- Interpretability
- Generalisation
- Storage / computation / cost

A naïve approach would be to try out all subsets of features. However, by using greedy feature selection, computational cost is massively reduced.

In the following, let $V = \{1, \dots, d\}$ be the set of all features and $\hat{L}(S)$ be the cross-validation error using features in $S \subseteq V$ only.

Greedy Forward Selection

Start with $S = \emptyset$ and $E_0 = \infty$.

Then, for $i = 1, \dots, d$, find the best element to add

$$s_i = \arg\min_{j \in V \setminus S} \hat{L}(S \cup \{j\})$$

and compute the new error

$$E_i = \hat{L}(S \cup \{s_i\})$$

If the new error increases, $E_i > E_{i-1}$, break, else update $S \leftarrow S \cup \{s_i\}$.

The problem with greedy forward selection is if there is some combination of features which improve the model together but do not improve it (or even make it worse) if used alone, i.e. dependent features.

Greedy Backward Selection

Start with $S = V$ and $E_{d+1} = \infty$.

Then, for $i = d, \dots, 1$, find the best element to remove

$$s_i = \arg\min_{j \in S} \hat{L}(S \setminus \{j\})$$

and compute the new error

$$E_i = \hat{L}(S \setminus \{s_i\})$$

If the new error increases, $E_i > E_{i-1}$, break, else update $S \leftarrow S \setminus \{s_i\}$.

Sparsity Trick in Linear Models

Explicitly selecting k features is equivalent to constraining $\boldsymbol{w}$ to have at most k non-zero entries in linear models, i.e. to be sparse.

The $\|\boldsymbol{w}\|_0$ is defined as the number of non-zero entries in $\boldsymbol{w}$. Then, we want to optimise the objective function so that $\|\boldsymbol{w}\|_0 \leq k$. Alternatively, a penalty term $\lambda\|\boldsymbol{w}\|_0$ can be added to the objective. However, this is a hard combinatorial optimisation problem.

The L_1 loss can act as a surrogate for the L_0 loss. This is called the sparsity trick.

Fisher consistency is defined as follows: Let $\psi : \mathcal{Y} \times \mathcal{S} \to \mathbb{R}$ be a surrogate loss and $\ell : \mathcal{Y} \times \mathcal{S} \to \mathbb{R}$ be a loss. ψ is consistent with respect to ℓ if every minimiser f of the surrogate risk $R_\psi(f)$ is also a minimiser of the risk function $R_\ell(f)$.

The L_1 norm is convex and thus, combined with convex losses, results in a convex optimisation problem.

While SGD can be used in principle, convergence is usually slow and coefficients are rarely exactly zero.

During cross-validation, a good approach is to start with a large λ (s.t. pretty much all weights are zero) and then gradually decrease it.

Comparison

For greedy methods, forward selection is usually faster if there are few relevant features, but it can miss dependencies. Backward selection can handle dependent features but may be very slow. Both methods are only heuristics and can result in non-optimal selections. Also, both methods can result in high computational cost.

L_1-regularisation is faster because training and feature selection happen jointly, but it only works for linear models.

1.6 Kernels

To find non-linear classification boundaries in linear classifiers, non-linear transformations of the feature vectors followed by linear classification can be used. In the case of, for example, polynomials however, $\mathcal{O}(d^k)$ dimensions are required to represent polynomials of degree k on d features. Kernels allow to implicitly use such a representation without the computational cost by implicitly calculating inner products in high-dimensional spaces.

The fundamental insight is that the optimal hyperplane lives in the span of the data, i.e.

$$\hat{\boldsymbol{w}} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$$

This follows from the Representer Theorem. Handwavily, starting gradient descent from $\mathbf{0}$ constructs and maintains such a representation.

The general Ansatz is to write

$$\boldsymbol{w} = \sum_{i=1}^{n} y_i \alpha_i \boldsymbol{x}_i$$

and recalculating the optimisation problems. From that follows that the objective only depends on inner products $\boldsymbol{x}_i^T \boldsymbol{x}_j$.

A kernel k is a function which computes

$$k(\boldsymbol{x}, \boldsymbol{x}') := \phi(\boldsymbol{x})^T \phi(\boldsymbol{x}')$$

Usually, k can be computed much more efficiently than $\phi(\boldsymbol{x})^T \phi(\boldsymbol{x}')$.

The *kernel trick* is expressing a problem such that it depends only on inner products and then replacing the inner products by kernels:

$$\boldsymbol{x}_i^T \boldsymbol{x}_j \Rightarrow k(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

Properties of Kernels

Let $\mathcal{X}$ be a data space.

A kernel is a function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ corresponding to some inner product $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle$ in some suitable space, satisfying

1. Symmetry: $\forall \boldsymbol{x}, \boldsymbol{x}' : k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}', \boldsymbol{x})$
2. Positive semi-definiteness: For any n and any set $S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subseteq \mathcal{X}$, the kernel (Gram) matrix $\boldsymbol{K}$ must be positive semi-definite.

Only positive semi-definiteness (instead of positive definiteness) is required due to the fact that the concatenation of functions $\langle \cdot, \cdot \rangle \circ \phi(\cdot)$ is calculated and ϕ could be non-injective, e.g. $\phi(\boldsymbol{x}) = 0$.

Alternatively, a symmetric function $k : \mathcal{X} \to \mathcal{X}$ is positive semi-definite if $\forall n > 0$, $\forall (a_1, \ldots, a_n) \in \mathbb{R}^n$, $\forall (x_1, \ldots, x_n) \in \mathcal{X}^n$,

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j k(x_i, x_j) \geq 0$$

A symmetric matrix $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ is positive semi-definite

$$\Leftrightarrow \forall \boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} \geq 0 \Leftrightarrow \text{All Eigenvalues of } \boldsymbol{M} \text{ are} \geq 0$$

Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel and $S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subseteq \mathcal{X}$ be a finite data subset. Then, the *kernel (Gram) matrix* is

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \ldots & k(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ \vdots & & \vdots \\ k(\boldsymbol{x}_n, \boldsymbol{x}_1) & \ldots & k(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{pmatrix} = \boldsymbol{\phi}^T \boldsymbol{\phi}$$

where

$$\boldsymbol{\phi} = \begin{pmatrix} \phi(\boldsymbol{x}_1) & \ldots & \phi(\boldsymbol{x}_n) \end{pmatrix}$$

The kernel (gram) matrix is positive semi-definite.

Conversely, from every symmetric positive semi-definite matrix $\boldsymbol{K} \in \mathbb{R}^{n \times n}$, we can construct a feature map $\phi : \mathcal{X} \to \mathbb{R}^n$ such that $\boldsymbol{K}_{i,j} = \phi(i)^T \phi(j)$:

$$\boldsymbol{K} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^T = \underbrace{\boldsymbol{U}\boldsymbol{D}^{\frac{1}{2}}}_{\boldsymbol{\phi}^T} \underbrace{\boldsymbol{D}^{\frac{1}{2}^T}\boldsymbol{U}^T}_{\boldsymbol{\phi}} = \begin{pmatrix} \phi(1) & \ldots & \phi(n) \end{pmatrix}^T \begin{pmatrix} \phi(1) & \ldots & \phi(n) \end{pmatrix}$$

Theorem (Mercer's Theorem) *Let $\mathcal{X}$ be a compact subset of $\mathbb{R}^d$ and $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ a kernel function.*

Then, k can be expanded in a uniformly convergent series of bounded functions ϕ_i so that

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{i=1}^{\infty} \lambda_i \phi_i(\boldsymbol{x}) \phi_i(\boldsymbol{x}')$$

Common Kernels

The Gaussian and Laplacian kernels compute an inner product in an infinite data space. In the following, $\mathcal{X}$ is the data space, $\mathcal{H}$ the inner product space (with $\langle \cdot, \cdot \rangle_{\mathcal{H}}$) and $\phi(\cdot)$ the feature map.

Linear Kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$$

$\mathcal{X} = \mathbb{R}^n$, $\mathcal{H} = \mathbb{R}^n$, $\phi(\boldsymbol{x}) = \boldsymbol{x}$.

Monomials of Degree m

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}')^m$$

Direct calculation is of order $\mathcal{O}(d^m)$ while the kernel is of order $\mathcal{O}(d)$.

Monomials of Degree up to m

$$k(\boldsymbol{x}, \boldsymbol{x}') = (1 + \boldsymbol{x}^T \boldsymbol{x}')^m$$

$\mathcal{X} = \mathbb{R}^n$, $\mathcal{H} = \mathbb{R}^{\binom{n+m}{m}}$.

The number of monomials up to degree m is $\mathcal{O}(\binom{d+m}{m})$.

Gaussian/RBF/Squared Exponential Kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{h^2}\right)$$

$\mathcal{X} = \mathbb{R}^n$, $\mathcal{H} = \mathbb{R}^\infty$.

h is the bandwitdh/lengthscale parameter. The smaller h, the more influence do samples have. Bigger h lead to smoother functions. If the bandwith is increased, it starts to behave like a linear kernel.

Laplacian Kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_1}{h}\right)$$

h is the bandwitdh/lengthscale parameter. The smaller h, the more influence do samples have. Bigger h lead to smoother functions. The Laplacian kernel decision boundary behaves sort of piecewise linear.

Matrix Kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x}$$

is a kernel if and only if $\boldsymbol{M}$ is symmetric, positive semi-definite.

ANOVA Kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{j=1}^{d} k_j(x_j, x_j')$$

where $x \in \mathbb{R}^d$, $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and $k_j : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ are valid kernels.

Semi-Parametric Regression Kernel

Often, parametric models are too rigid and non-parametric models fail to interpolate. Thus, an additive combination of linear and non-linear kernels can be used:

$$k(\boldsymbol{x}, \boldsymbol{x}') = c_1 \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{h^2}\right) + c_2 \boldsymbol{x}^T \boldsymbol{x}'$$

Kernel Engineering

Let $k_1 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ and $k_2 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be two kernels, $c > 0$ be a scalar, $f : \mathbb{R} \to \mathbb{R}$ be either a polynomial with positive coefficients or the exponential function and $\mathcal{V} : \mathcal{Z} \to \mathcal{X}$ be a mapping.

Then, the following are also valid kernels:

- $k(\boldsymbol{x}, \boldsymbol{x}') = k_1(\boldsymbol{x}, \boldsymbol{x}') + k_2(\boldsymbol{x}, \boldsymbol{x}')$
- $k(\boldsymbol{x}, \boldsymbol{x}') = k_1(\boldsymbol{x}, \boldsymbol{x}') \cdot k_2(\boldsymbol{x}, \boldsymbol{x}')$
- $k(\boldsymbol{x}, \boldsymbol{x}') = c \cdot k_1(\boldsymbol{x}, \boldsymbol{x}')$
- $k(\boldsymbol{x}, \boldsymbol{x}') = f(k_1(\boldsymbol{x}, \boldsymbol{x}'))$
- $k(\boldsymbol{z}, \boldsymbol{z}') = k_1(\mathcal{V}(\boldsymbol{z}), \mathcal{V}(\boldsymbol{z}'))$

Interpretation

Kernels can be interpreted as similarity functions. They compute a function around each data point. The resulting function is then a linear combination of those functions, scaled by the α_is.

1.7 Classification Metrics and Class Imbalance

Class Imbalance

Generally, the positive class is assumed to be rare. There are multiple ways class imbalance can be mitigated:

Subsampling is removing majority class samples from the data set (e.g. uniformly at random).

Upsampling is repeating data points from the minority class, possibly with small random perturbations.

Cost-sensitive classification methods change the loss function to consider the class distribution.

Downsampling results in a smaller data set, thus training becomes faster. However, data is wasted and information about the majority class may be lost. Upsampling makes use of all data but is slower and perturbations requires arbitrary choices. Cost-sensitive classification methods are a solution to those problems.

In cost-sensitive classification, the loss $\ell(\boldsymbol{w};\boldsymbol{x},y)$ is replaced as

$$\ell_{CS} = c_y \ell(\boldsymbol{w};\boldsymbol{x},y)$$

where c_+, c_- (in the binary case) control the tradeoff.

In the case of perceptron and SVM, the constants change the slope of the loss. They also scale the gradient.

In the case of binary classification, specifying c_+ and c_- is equivalent to specifying $\alpha = \frac{c_+}{c_-}$ and using $c'_+ = \alpha, c_- = 1$.

Instead of using a cost-sensitive classifier, one can also use a single classifier and vary the *classification threshold* τ ($y = sign(\boldsymbol{w}^T\boldsymbol{x} - \tau)$). This is similar to training multiple classifiers with different cost-sensitive parameters, but has lower computational cost. However, optimising the cost-sensitive loss should lead to better results.

Classification Metrics

		True label: Positive	True label: Negative	
Predicted label	Positive	TP	FP	$\Sigma = p_+$
	Negative	FN	TN	$\Sigma = p_-$
		$\Sigma = n_+$	$\Sigma = n_-$	

A confusion matrix is a generalisation of the table to c classes where the columns are the true class, the rows the predicted classes, and the fields contain the counts.

Let n be the total number of samples. $n = p_+ + p_- = n_+ + n_-$. Everything containing n refers to the true data, everything containing p to the predictions.

General metrics are

Accuracy :

$$\frac{TP + TN}{n}$$

Precision :

$$\frac{TP}{TP + FP} = \frac{TP}{p_+}$$

Recall / Sensitivity / TPR :

$$\frac{TP}{TP + FN} = \frac{TP}{n_+}$$

Specificity / TNR :

$$\frac{TN}{FP+TN} = \frac{TN}{n_-}$$

F1 Score :

$$\frac{2TP}{2TP+FP+FN} = \frac{2}{\frac{1}{prec}+\frac{1}{rec}} = 2 \cdot \frac{prec \cdot rec}{prec+rec}$$

F_β **Score** :

$$(1+\beta^2) \cdot \frac{prec \cdot rec}{\beta^2 prec + rec}$$

The F1 score is the harmonic mean between precision and recall. Precision and recall should not be averaged! The F_β score is a generalisation of the F1 score.

TODO: Micro and macro averaging of F-scores.

TODO: Cross-entropy loss.

In the *precision recall curve*, recall is used for the x-axis, precision for the y-axis. Being in the top-right corner is better.

More metrics are

True positive rate (TPR) :

$$\frac{TP}{TP+FN} = \frac{TP}{n_+} = recall$$

False positive rate (FPR) :

$$\frac{FP}{TN+FP} = \frac{FP}{n_-}$$

True negative rate (TNR) :

$$\frac{TN}{FP+TN} = \frac{TN}{n_-} = specificity$$

Suppose class + is predicted with probability p. Then, $\mathbb{E}[TPR] = \mathbb{E}[FPR] = p$. Thus, TPR and FPR establish a baseline: If we predict labels at random with probability p, the expected TPR and FPR are p.

The *Receiver Operator Characteristic (ROC) Curve* uses the FPR as the x-axis, TPR as the y-axis. Being on the line with slope 1 corresponds to random guessing. Being closer to the bottom right corner is worse than random guessing, being closer to the top left corner is better.

Theorem *Algorithm 1 dominates algorithm 2 in terms of the ROC Curve if and only if algorithm 1 dominates algorithm 2 in terms of the Precision Recall Curve.*

One algorithm dominates the other if its curve is strictly above the other curve.

The *Area under the Curve (AUC)* describes the area under the Precision Recall / ROC curve. Thus, for ROC, $AUC = \frac{1}{2}$ corresponds to random guessing, $AUC = 1$ is ideal.

1.8 Multi-Class Classification

In this section, let $y_i \in \mathcal{Y} = \{1, \ldots, c\}$ and D be the data set. In a multi-class classification scenario, we want a function $f : \mathcal{X} \to \mathcal{Y}$.

In one-vs-all (OVA) / one-vs-rest (OVR) classification, c binary classifiers $f^{(i)}$ are trained, one for each class. The positive examples are those from class c, negative examples those from all other classes.

The prediction is then

$$\hat{y} = \arg\max_i f^{(i)}(\boldsymbol{x})$$

i.e. take the classifier with the highest confidence.

The confidence of a classification is given as $|f^{(i)}(\boldsymbol{x})|$. For any $\alpha > 0$, multiplying it to the function output does not change the classification result, but the confidence. One solution for a consistent confidence measure would thus be to normalise the weights, i.e. $\boldsymbol{w} \leftarrow \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|_2}$. However, in practice, if regularisation is used, the magnitude $\|\boldsymbol{w}\|_2$ is generally under control.

If unit magnitude for all weights is assumed, we can consider the euclidian distance from a point to each decision boundary. Thus, in all overlapped areas, the decision boundaries are separated by equal angles.

One-vs-all classification only works if classifiers produce confidences of similar magnitude. Furthermore, individual classifiers almost always see a very imbalanced data set. Also, some class might not be linearly separable from the others, leading to failure.

In one-vs-one (OVO) classification, $\frac{c(c-1)}{2}$ classifiers are trained; one for each distinct pair of classes (i, j). The positive samples are those from class i, negative ones from class j.

The prediction is done using a voting scheme. The class with the highest number of positive predictions wins. However, ties may happen and need to be broken somehow.

The advantage of OVA is that it is faster as fewer classifiers need to be trained. However, it requires confidence and leads to class imbalance. OVO does not have confidence or imbalance issues, but is slower to train.

The alternatives are using other encodings or multi-class models.

1.9 Dimensionality Reduction

The basic challenge is: Given a data set $D = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \in \mathbb{R}^d$, find an *embedding* $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n \in \mathbb{R}^k$ with $k < d$.

The embedding may then be used for visualisation, regularisation (model selection), unsupervised feature discovery, and more.

Typically, a mapping $f : \mathbb{R}^d \to \mathbb{R}^k$ with $k \ll d$ is obtained.

In *linear dimensionality reduction*, the mapping is of the form $f(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x}, \boldsymbol{A} \in \mathbb{R}^{k \times d}$. In *nonlinear dimensionality reduction*, other mappings (parametric or non-parametric) are used.

Linear dimensionality reduction can be interpreted as a form of compression. It should be possible to accurately reconstruct the original data from the embedding.

1.10 Probabilistic Modelling

The goal of supervised learning is: Given training data $D = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$, we want to identify a hypothesis $h : \mathcal{X} \to \mathcal{Y}$ and minimise the prediction error (risk). The prediction error / true risk is defined according to some loss function ℓ and given as

$$R(h) = \int P(\boldsymbol{x}, y)\ell(y; h(\boldsymbol{x}))d\boldsymbol{x}dy = \mathbb{E}_{\boldsymbol{X},Y}[\ell(Y; h(\boldsymbol{X})]$$

The fundamental assumption is that the data set is generated i.i.d., i.e.

$$(\boldsymbol{x}_i, y_i) \overset{i.i.d.}{\sim} P(\boldsymbol{X}, Y)$$

In least squares regression, the risk is $R(h) = \mathbb{E}_{\boldsymbol{X},Y}[(Y - h(\boldsymbol{X}))^2]$. Assuming the true distribution $P(\boldsymbol{X}, Y)$ is known, the h minimising the risk is given as

$$\begin{aligned}
\min_h R(h) &= \min_h \mathbb{E}_{\boldsymbol{X},Y}[(Y - h(\boldsymbol{X}))^2] \\
&= \min_h \mathbb{E}_{\boldsymbol{X}}[\mathbb{E}_Y[(Y - h(\boldsymbol{x}))^2 \mid \boldsymbol{X} = \boldsymbol{x}]] \\
&= \mathbb{E}_{\boldsymbol{X}}[\min_{h(\boldsymbol{x})} \mathbb{E}_Y[(Y - h(\boldsymbol{x})^2 \mid \boldsymbol{X} = \boldsymbol{x}]]
\end{aligned}$$

since h can be considered independently for each $\boldsymbol{x}$.

Now, for a fixed $\boldsymbol{x}$, the optimal prediction is

$$y^*(\boldsymbol{x}) \in \arg\min_{\hat{y}} \underbrace{\mathbb{E}_Y[(\hat{y} - y)^2]}_{=\ell(\hat{y})}$$

This leads to

$$\begin{aligned}
\frac{d}{d\hat{y}}\ell(\hat{y}) &= \int 2(\hat{y} - y)p(y \mid \boldsymbol{x})dy \overset{!}{=} 0 \\
&\Rightarrow \underbrace{\int \hat{y} \cdot p(y \mid \boldsymbol{x})dy}_{=\hat{y}} = \underbrace{\int y \cdot p(y \mid \boldsymbol{x})dy}_{=\mathbb{E}[Y|\boldsymbol{X}=\boldsymbol{x}]} \\
&\Leftrightarrow \hat{y} = \mathbb{E}[Y|\boldsymbol{X} = \boldsymbol{x}]
\end{aligned}$$

Therefore, in least squares regression, the hypothesis minimising the true risk is given by the conditional mean

$$h^*(\boldsymbol{x}) = \mathbb{E}[Y \mid \boldsymbol{X} = \boldsymbol{x}]$$

This hypothesis is called the *Bayes' optimal predictor*.

In practice, with finite data, one strategy is to estimate the conditional distribution

$$\hat{P}(Y \mid \boldsymbol{X})$$

and for a test point $\boldsymbol{x}$ to predict the label

$$\hat{y} = \hat{\mathbb{E}}[Y \mid \boldsymbol{X} = \boldsymbol{x}] = \int y \cdot \hat{P}(y \mid \boldsymbol{X} = \boldsymbol{x}) dy$$

Conditional Maximum Likelihood Estimation

A common approach for conditional distribution estimation is to choose a particular *parametric form* $\hat{P}(Y \mid \boldsymbol{X}, \theta)$ and optimise the parameters using *maximum conditional likelihood estimation*:

$$\theta^* = \arg\max_{\theta} \hat{P}(y_1, \ldots, y_n \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \theta)$$

By factoring the density (as the data is i.i.d.), by applying the logarithm and switching the sign, this is equivalent to minimising the *negative log likelihood*

$$\theta^* = \arg\min_{\theta} -\sum_{i=1}^{n} \log \hat{P}(y_i \mid \boldsymbol{x}_i, \theta)$$

We denote the negative log likelihood by $L(\theta)$.

MLE has several nice statistical properties:

Consistency : Parameter estimate converges to true parameters in probability.

Asymptotic efficiency : Smallest variance among all well-behaved estimators for large n.

Asymptotic normality

Example: Regression with Gaussian Noise

Assume $Y = h(\boldsymbol{X}) + \varepsilon$, $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ (with known σ^2) and $h(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$. I.e., the model is linear and noise is Gaussian with known variance.

Then, $\hat{p}(y \mid \boldsymbol{X} = \boldsymbol{x}) = \mathcal{N}(y; \boldsymbol{w}^T \boldsymbol{x}, \sigma^2)$ and

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} L(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} -\sum_{i=1}^{n} \log \hat{p}(y_i \mid \boldsymbol{x}_i, \boldsymbol{w}, \sigma^2)$$

Furthermore,

$$\begin{aligned}
-\log \hat{p}(y_i \mid \boldsymbol{x}_i, \boldsymbol{w}, \sigma^2) &= -\log \mathcal{N}(y; \boldsymbol{w}^T \boldsymbol{x}, \sigma^2) \\
&= -\log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - \boldsymbol{w}^T \boldsymbol{x})^2}{2\sigma^2}\right) \\
&= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y - \boldsymbol{w}^T \boldsymbol{x})^2
\end{aligned}$$

and therefore

$$\begin{aligned}
\hat{\boldsymbol{w}} &= \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 \\
&= \arg\min_{\boldsymbol{w}} \underbrace{\frac{n}{2}\log(2\pi\sigma^2)}_{=\text{const w.r.t. } \boldsymbol{w}} + \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 \\
&= \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n}(y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2
\end{aligned}$$

Thus, under the conditional linear Gaussian assumption, maximising the likelihood is equivalent to least squares regression.

MLE for i.i.d. Gaussian noise

The above example holds more general.

Suppose $\mathcal{H} = \{h : \mathcal{X} \to \mathbb{R}\}$ is a class of functions. Further assume

$$P(Y = y \mid \boldsymbol{X} = \boldsymbol{x}) = \mathcal{N}(y \mid h^*(\boldsymbol{x}), \sigma^2)$$

for some $h^* \in \mathcal{H}$ and some $\sigma^2 > 0$.

The MLE for data $D = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ is given by

$$\hat{h} = \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n}(y_i - h(\boldsymbol{x}_i))^2$$

Thus, the MLE is given by the least squares solution, assuming noise is iid Gaussian with constant variance.

Bias-Variance Tradeoff

While MLEs have nice statistical properties, those only hold for $n \to \infty$. For finite n, overfitting must be avoided.

The bias variance tradeoff states for the sum of squared errors that

$$\text{Prediction error} = \text{Bias}^2 + \text{Variance} + \text{Noise}$$

where

Bias is the excess risk of the best *considered* model, compared to minimal achievable risk knowing $P(\boldsymbol{X}, Y)$ (i.e. given infinite data). Going away from the Bayes' optimal prediction, i.e. restricting function class, increases bias.

Variance is the risk incurred due to estimating using finite data

Noise is the risk incurred by optimal model (i.e. irreducible error)

Usually, bias and variance have to be traded via model selection and regularisation. High bias corresponds to underfitting while high variance corresponds to overfitting.

The MLE solution depends on the training data D, i.e. $\hat{h} = \hat{h}_D$.

We want to chose $\mathcal{H}$ to have small bias, i.e. have small squared error on average:

$$\mathbb{E}_{\boldsymbol{X}}[\underbrace{\mathbb{E}_D[\hat{h}_D(\boldsymbol{X})] - h^*(\boldsymbol{X})}_{\text{Bias}}]^2$$

The estimator itself is random and thus has some variance:

$$\mathbb{E}_{\boldsymbol{X}}[\textsc{Var}_D[\hat{h}_D(\boldsymbol{X})]^2] = \mathbb{E}_{\boldsymbol{X}}[\mathbb{E}_D[(\hat{h}_D(\boldsymbol{X}) - \mathbb{E}_{D'}[\hat{h}_{D'}(\boldsymbol{X})])^2]]$$

TODO: No idea if the variance formula is correct...

Even if the Bayes' optimal hypothesis h^* is known, we would still incur some error du to noise:

$$\mathbb{E}_{\boldsymbol{X},Y}[(Y - h^*(\boldsymbol{X}))^2]$$

Ultimately, for least squares regression it holds that

$$\underbrace{\mathbb{E}_D[\mathbb{E}_{\boldsymbol{X},Y}[(Y - \hat{h}_D(\boldsymbol{X}))^2]]}_{\text{Expected risk}}$$

TODO: Week 18, slide 20

The MLE for linear regression is unbiased if $h^* \in \mathcal{H}$. Furthermore, it is the minimum variance estimator among all unbiased estimators. However, it may still overfit. Thus, we can trade a small amount of bias for a potentially large reduction in variance by using regularisation.

Maximum a Posteriori Estimate

Bias can be introduced by expressing assumptions / a belief on the parameters. This is done using a *Bayesian prior*.

Assume the parameters θ are from a known *prior distribution* which is fixed before any data is observed. Particularly, θ and the $\boldsymbol{X}_i$ are independent.

Then, the *posterior distribution* of the parameter θ is given using Bayes' rule and $P(\theta \mid \boldsymbol{x}_{1:n}) = P(\theta)$ as

$$\underbrace{P(\theta \mid y_{1:n}, \boldsymbol{x}_{1:n})}_{\text{posterior}} = \frac{\overbrace{P(\theta)}^{\text{prior}} \overbrace{P(y_{1:n} \mid \theta, \boldsymbol{x}_{1:n})}^{\text{likelihood}}}{P(y_{1:n} \mid \boldsymbol{x}_{1:n})}$$

Finally, the *maximum a posteriori (MAP) estimate* is obtained by maximising $P(\theta \mid y_{1:n}, \boldsymbol{x}_{1:n})$ (or equivalently minimising the negative log term). Note that $P(y_{1:n} \mid \boldsymbol{x}_{1:n})$ is constant w.r.t. θ and can thus be ignored during optimisation.

MLE is a special case of MAP estimation. Choosing a uniform parameter prior during MAP estimation results in MLE.

Example: Ridge Regression

Assume $\theta = \boldsymbol{w} \sim \mathcal{N}(0, \beta^2 \boldsymbol{I})$, i.e. the $w_i \sim \mathcal{N}(0, \beta^2)$ independently.

Then,

$$\arg\max_{\boldsymbol{w}} P(\boldsymbol{w} \mid \boldsymbol{x}_{1:n}, y_{1:n}) = \arg\min_{\boldsymbol{w}} -\log P(\boldsymbol{w}) - \log P(y_{1:n} \mid \boldsymbol{x}_{1:n}, \boldsymbol{w}) + \underbrace{\log P(y_{1:n} \mid \boldsymbol{x}_{1:n})}_{\text{constant w.r.t. } \boldsymbol{w}}$$

For the negative prior log likelihood, we have

$$\begin{aligned}
-\log P(\boldsymbol{w}) &= -\log \prod_{i=1}^{d} P(w_i) \\
&= -\sum_{i=1}^{d} \log \mathcal{N}(w_i; 0, \beta^2) \\
&= -\sum_{i=1}^{d} \log \frac{1}{\sqrt{2\pi\beta^2}} \exp\left(-\frac{w_i^2}{2\beta^2}\right) \\
&= \frac{d}{2} \log 2\pi\beta^2 + \frac{1}{2\beta^2} \sum_{i=1}^{d} w_i^2 \\
&= \text{const} + \frac{1}{2\beta^2} \|\boldsymbol{w}\|_2^2
\end{aligned}$$

Combining the above form with the already known negative log likeli-

hood of a linear model with Gaussian noise, we get

$$\begin{aligned}
&\arg\min_{\boldsymbol{w}} -\log P(\boldsymbol{w}) - \log P(y_{1:n} \mid \boldsymbol{x}_{1:n}, \boldsymbol{w}) \\
&= \arg\min_{\boldsymbol{w}} \frac{1}{2\beta^2}\|\boldsymbol{w}\|_2^2 + \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2 \\
&= \arg\min_{\boldsymbol{w}} \underbrace{\frac{\sigma^2}{\beta^2}}_{\lambda}\|\boldsymbol{w}\|_2^2 + \sum_{i=1}^{n}(y_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2
\end{aligned}$$

Therefore, ridge regression is MAP estimation for a linear regression problem, assuming that the noise $P(y \mid \boldsymbol{x}, \boldsymbol{w})$ is i.i.d. Gaussian and the prior $P(\boldsymbol{w})$ is Gaussian.

λ is the ratio between the noise variance and the prior variance. Thus, a large λ means that we believe to have a lot of noise or the weights to be close to zero.

Example: Lasso Regression

Lasso regression / L_1-regularised linear regression corresponds to a linear regression problem with Gaussian noise and a Laplace parameter prior.

The density of a (univariate) Laplace distribution is given as

$$p(x; \mu, b) = \frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right)$$

The Laplace distribution has a spike around μ. With $\mu = 0$, this should bias the weights to be exactly zero. However, the Laplace prior might not be optimal for sparse solutions as we get non-zero weights with probability 1 (it is continuous).

TODO: Slide 30. The probability for a Gaussian distribution to have values far off from the mean is very low, exponentially low. The student-t distribution is heavier tailed, thus has only polynomial decay. This can help substantially for robustness towards outliers. Gaussian likelihood assumes that all points are very close to the prediction. This is generally not true, especially if we have large outliers. In those cases, we should use a different loss function, i.e. replacing Gaussian with a more heavier-tailed distribution.

TODO: student's t-distribution slides 19, p. 9

Regularisation via MAP inference

Regularised estimation can often be understood as MAP inference. The loss corresponds to the likelihood, the regulariser to the parameter prior. This

leads to the form

$$\arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \ell(\boldsymbol{w}^T \boldsymbol{x}_i; \boldsymbol{x}_i, y_i) + C(\boldsymbol{w}) = \arg\max_{\boldsymbol{w}} \prod_{i=1}^{n} P(y_i \mid \boldsymbol{x}_i, \boldsymbol{w}) P(\boldsymbol{w}) = \arg\max_{\boldsymbol{w}} P(\boldsymbol{w} \mid D)$$

where

$$C(\boldsymbol{w}) = -\log P(\boldsymbol{w})$$
$$\ell(\boldsymbol{w}^T \boldsymbol{x}_i; \boldsymbol{x}_i, y_i) = -\log P(y_i \mid \boldsymbol{x}_i, \boldsymbol{w})$$

This allows to exchange priors (regularisers) and likelihoods (loss functions).

Classification

In classification, the risk is

$$R(h) = \mathbb{E}_{\boldsymbol{X},Y}[\quad \underbrace{[Y \neq h(\boldsymbol{X})]}_{1 \text{ if } Y \neq H(\boldsymbol{X}),\ 0 \text{ otherwise}} \quad]$$

Assuming $P(\boldsymbol{X}, Y)$ is known, and $(\boldsymbol{x}_i, y_i) \sim P(\boldsymbol{X}, Y)$ iid, we get

$$\begin{aligned}
h^*(\boldsymbol{x}) &= \arg\min_{\hat{y}} \mathbb{E}_Y[[Y \neq \hat{y}] \mid \boldsymbol{X} = \boldsymbol{x}] \\
&= \arg\min_{\hat{y}} \sum_{y=1}^{c} P(Y = y \mid \boldsymbol{X} = \boldsymbol{x}) \cdot [y \neq \hat{y}] \\
&= \arg\min_{\hat{y}} \sum_{y:y\neq\hat{y}} P(Y = y \mid \boldsymbol{X} = \boldsymbol{x}) \\
&= \arg\min_{\hat{y}} 1 - P(Y = \hat{y} \mid \boldsymbol{X} = \boldsymbol{x}) \\
&= \arg\max_{\hat{y}} P(Y = \hat{y} \mid \boldsymbol{X} = \boldsymbol{x})
\end{aligned}$$

Thus, the *Bayes' optimal predictor* (classifier) is given by the most probable class:

$$h^*(\boldsymbol{x}) = \arg\max_{y} P(Y = y \mid \boldsymbol{X} = \boldsymbol{x})$$

A natural approach is therefore again to estimate $P(Y \mid \boldsymbol{X})$.

MAP Learning Summary

To summaries, learning through MAP inference works as follows:

1. Start with i.i.d. assumption on data points (can be relaxed).
2. Choose likelihood function (results in loss).

3. Choose prior distribution (results in regulariser).
4. Optimise for MAP parameters.
5. Choose hyperparameters via cross-validation.
6. Make predictions via Bayesian decision theory.

1.11 Bayesian Decision Theory

Let $P(y \mid \boldsymbol{x})$ be a conditional distribution over labels, $\mathcal{A}$ a *set of actions*, and $C : \mathcal{Y} \times \mathcal{A} \to \mathbb{R}$ be a *cost function*.

Bayesian decision theory (BDT) recommends to pick the action which minimises the *expected cost*:

$$a^* = \arg\min_{a \in \mathcal{A}} \mathbb{E}_y[C(y, a) \mid \boldsymbol{x}]$$

If the true distribution $P(y \mid \boldsymbol{x})$ was known, this decision implements the *Bayes optimal decision*. However, in practice, the distribution can only be estimated. If the action set is discrete, calculations are often simplified by calculating the expected cost for each action individually and then comparing them.

Example: Logistic Regression

In logistic regression, the estimated conditional distribution is

$$\hat{P}(y \mid \boldsymbol{x}) = Ber(y; \sigma(\hat{\boldsymbol{w}}^T \boldsymbol{x}))$$

the action set is $\mathcal{A} = \{+1, -1\}$ and the cost function $C(y, a) = [y \neq a]$.

Then, the action which minimises the expected cost is the most likely class:

$$\begin{aligned} a^* &= \arg\min_{a \in \mathcal{A}} \mathbb{E}_y[C(y, a) \mid \boldsymbol{x}] \\ &= \arg\max_y \hat{P}(y \mid \boldsymbol{x}) \\ &= sign(\boldsymbol{w}^T \boldsymbol{x}) \end{aligned}$$

Asymmetric Costs

Logistic regression is used again, however, with asymmetric cost. The estimated conditional distribution is

$$\hat{P}(y \mid \boldsymbol{x}) = Ber(y; \sigma(\hat{\boldsymbol{w}}^T \boldsymbol{x}))$$

the action set is $\mathcal{A} = \{+1, -1\}$ and the cost function is

$$C(y, a) = \begin{cases} c_{FP} & \text{if } y = -1 \text{ and } a = +1 \\ c_{FN} & \text{if } y = 1 \text{ and } a = -1 \\ 0 & \text{otherwise} \end{cases}$$

Let $p = P(Y = +1 \mid \boldsymbol{x})$ be the probability of a positive classification for the input $\boldsymbol{x}$. We can now defined the costs of positive predictions as

$$c_+ = \mathbb{E}_y[C(y, +1) \mid \boldsymbol{x}] = (1 - p) \cdot c_{FP} + p \cdot 0$$

and for negative predictions as

$$c_- = \mathbb{E}_y[C(y, -1) \mid \boldsymbol{x}] = (1 - p) \cdot 0 + p \cdot c_{FN}$$

The optimal prediction for the estimated distribution (using $p = P(y = +1 \mid \boldsymbol{x})$) is then given as

$$\begin{aligned} \text{predict} \hat{y} = +1 &\Leftrightarrow c_+ < c_- \\ &\Leftrightarrow (1 - p) \cdot c_{FP} < p \cdot c_{FN} \\ &\Leftrightarrow p > \frac{c_{FP}}{c_{FP} + c_{FN}} \end{aligned}$$

Uncertainty

Logistic regression is used again, however, with an additional doubt action. The estimated conditional distribution is

$$\hat{P}(y \mid \boldsymbol{x}) = Ber(y; \sigma(\hat{\boldsymbol{w}}^T \boldsymbol{x}))$$

The action set is now $\mathcal{A} = \{+1, -1, D\}$ where D stands for *doubt*. For some doubt penalty c, the new cost function is

$$C(y, a) = \begin{cases} [a \neq y] & \text{if } a \in \{+1, -1\} \\ c & \text{if } a = D \end{cases}$$

Then, the action minimising the expected cost is given by

$$a^* = \begin{cases} y & \text{if } \hat{P}(y \mid \boldsymbol{x}) \geq 1 - c \\ D & \text{otherwise} \end{cases}$$

Example: Least-Squares Regression

In least-squares regression, the estimated conditional distribution is

$$\hat{P}(y \mid \boldsymbol{x}) = \mathcal{N}(y; \hat{\boldsymbol{w}}^T \boldsymbol{x}, \sigma^2)$$

the action set is $\mathcal{A} = \mathbb{R}$ and the cost function $C(y, a) = (y - a)^2$.

Then, the action which minimises the expected cost is the conditional mean:

$$\begin{aligned} a^* &= \arg\min_{a \in \mathcal{A}} \mathbb{E}_y[C(y, a) \mid \boldsymbol{x}] \\ &= \mathbb{E}[y \mid \boldsymbol{x}] \\ &= \int y \cdot \hat{P}(y \mid \boldsymbol{x}) dy \\ &= \hat{\boldsymbol{w}}^T \boldsymbol{x} \end{aligned}$$

If we chose an asymmetric cost

$$C(y, a) = \underbrace{c_1 \max(y - a, 0)}_{\text{underestimation}} + \underbrace{c_2 \max(a - y, 0)}_{\text{overestimation}}$$

then the action which minimises the expected cost is

$$a^* = \hat{\boldsymbol{w}}^T \boldsymbol{x} + \sigma \cdot \Phi^{-1}\left(\frac{c_1}{c_1 + c_2}\right)$$

This shifts the prediction according to the costs.

Active Learning

The goal of active learning is to minimise the number of required labels. The general idea is to pick examples which the estimated conditional distribution is most uncertain about. This requires a uncertainty score u_j. A simple one is $u_j = -|\hat{P}(y_j = +1 \mid \boldsymbol{x}_j) - 0.5|$. The most uncertain example is then $j^* = \arg\max_j u_j$.

Given a pool of unlabelled examples $D_X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ and an initially empty set of labelled samples D. For $t = 1, 2, \ldots$, first estimate $\hat{P}(Y \mid \boldsymbol{x})$ given the current data set D. Then, pick the unlabelled example which the current estimate is most uncertain about:

$$i_t \in \arg\min_i |0.5 - \hat{P}(Y_i \mid \boldsymbol{x}_i)|$$

Finally, query the label y_{i_t} and update the data set $D \leftarrow D \cup \{(\boldsymbol{x}_{i_t}, y_{i_t})\}$.

Active learning violates the i.i.d. assumption! It can thus get stuck with bad models. There are also more advance selection criteria, e.g. querying the point which reduces uncertainty of other points as much as possible.

1.12 Generative Modelling

Discriminative models aim to estimate the conditional distribution

$$P(y \mid \boldsymbol{x})$$

Generative models aim to estimate the joint distribution

$$P(y, \boldsymbol{x})$$

Because discriminative models do not attempt to model $P(\boldsymbol{x})$, they are unable to detect outliers and can thus become overconfident.

A conditional distribution can always be derived from a joint one, but not vice versa:

$$P(y \mid \boldsymbol{x}) = \frac{P(\boldsymbol{x}, y)}{P(\boldsymbol{x})} = \frac{P(\boldsymbol{x}, y)}{\int P(\boldsymbol{x}, y) dy}$$

The typical approach for generative modelling is as follows:

1. Estimate prior on labels: $P(y)$
2. Estimate conditional distribution $P(\boldsymbol{x} \mid y)$ for each class y
3. Obtain posterior (predictive) distribution using Bayes' rule:

$$P(y \mid \boldsymbol{x}) = \frac{1}{Z} \underbrace{P(y)P(\boldsymbol{x} \mid y)}_{P(\boldsymbol{x}, y)}$$

where $Z = P(\boldsymbol{x})$ is a normalisation constant, i.e. $Z = \sum_y P(y)P(\boldsymbol{x} \mid y)$. If only $P(y \mid \boldsymbol{x})$ is desired, the normalisation constant can be omitted.

Thus, generative modelling attempts to infer the process according to which examples are generated.

Comparison

In generative modelling, we have a probabilistic model of each class. The decision boundary is where one model becomes more likely. It allows natural usage of unlabelled data. Discriminative modelling focuses on the decision boundary. It is more powerful (in terms of prediction) if a lot of examples are available. It also cannot use unlabelled data.

Generative models are always better if the model is well-specified. Otherwise, generative modelling is better in case of small amount of data, discriminative modelling in case of large amount of data.

Conjugate Distributions

A pair of prior distribution and likelihood function is called *conjugate* if the posterior distribution remains in the same family as the prior.

Conjugate priors thus allow for regularisation with almost no additional computational cost.

List of Conjugate Distributions

TODO: List of conjugate priors, slides 22 p. 36 and tutorial.

Dirichlet Prior, Multinomial Likelihood

The *Dirichlet* prior is conjugate with respect to the *Multinomial* likelihood.

Let $\boldsymbol{X} = (X_1, \ldots, X_n)$ and $X_i \sim Categorical(\boldsymbol{\theta})$ iid. $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$ is the vector of the probabilities for individual values. Furthermore, let N_j denote the the number of times j occurs in $\boldsymbol{X}$ and $\boldsymbol{N} = (N_1, \ldots, N_m)$.

Then, $\boldsymbol{N} \sim Multi(\boldsymbol{\theta}, n)$ is from a *multinomial* distribution.

The probability mass function is

$$P(\boldsymbol{N} \mid n, \boldsymbol{\theta}) = \underbrace{\frac{n!}{\prod_{j=1}^{m} N_j!}}_{\text{normalisation constant}} \prod_{j=1}^{m} \theta_j^{N_j}$$

For $Multi(\boldsymbol{\theta}, 1)$ is a categorical distribution, $Multi((\theta, 1-\theta), n)$ equivalent to a binomial distribution. The multinomial distribution is thus a generalisation of the binomial distribution.

The *Dirichlet distribution* has probability density function

$$P(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) = \underbrace{\frac{\Gamma(\sum_{j=1}^{m} \alpha_j)}{\prod_{j=1}^{m} \Gamma(\alpha_j)}}_{\text{normalisation constant}} \prod_{j=1}^{m} \theta_j^{\alpha_j - 1}$$

with $\alpha_j > 0$. We write $\boldsymbol{\theta} \sim Dir(\boldsymbol{\alpha})$.

If we have a Dirichlet prior $P(\boldsymbol{\theta})$ and a multinomial likelihood $P(\boldsymbol{X} \mid \boldsymbol{\theta})$, the resulting posterior $P(\boldsymbol{\theta} \mid \boldsymbol{X}) \propto P(\boldsymbol{X} \mid \boldsymbol{\theta}) P(\boldsymbol{\theta})$ is

$$P(\boldsymbol{\theta} \mid \boldsymbol{X}) = Dir(\boldsymbol{N} + \boldsymbol{\alpha})$$

Thus, $\boldsymbol{\alpha}$ act as pseudo-counts, similar to the beta prior and binomial likelihood.

The Dirichlet prior is also conjugate with respect to the Categorical likelihood.

The MLE of the multinomial likelihood is

$$\boldsymbol{\theta}_j^* = \frac{N_j}{n}$$

and the MAP estimate with a Dirichlet prior is

$$\boldsymbol{\theta}_j^* = \frac{N_j + \alpha_j - 1}{n + \sum_{j'=1}^{m} (\alpha_{j'} - 1)}$$

Chapter 2

Algorithms

2.1 Linear Regression

The model assumption is that $y \approx f(x)$ where $f(x)$ is linear (affine) in x, i.e. $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0$. We generally use a homogeneous representation:

$$\boldsymbol{w}^T\boldsymbol{x} + w_0 = \tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{x}}$$

where $\tilde{\boldsymbol{w}} = [w_1, \ldots, w_d, w_0]^T$ and $\tilde{\boldsymbol{x}} = [x_1, \ldots, x_d, 1]^T$. For equality, noise denoted by a random variable ε has to be accounted for, i.e. $y = \tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{x}} + \varepsilon$. For a complete dataset $\boldsymbol{X}$ with labels $\boldsymbol{y}$, this results in

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + \boldsymbol{\varepsilon}$$

The data set is $D = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ with $\boldsymbol{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$.
A residual is $r_i = y_i - \boldsymbol{w}^T\boldsymbol{x}_i$ and the vector of residuals is given as $\boldsymbol{r} = \boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}$.

Least-Squares Linear Regression

For least-squares linear regression, the empirical cost is

$$\hat{R}(\boldsymbol{w}) = \|\boldsymbol{r}\|_2^2 = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2$$

and we try to find $\boldsymbol{w}^*$ with

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2$$

This can be rewritten as

$$\hat{R}(\boldsymbol{w}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 = \boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}$$

The problem can be solved in closed form as

$$\boldsymbol{w}^* = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

where $\boldsymbol{X}$ is a matrix whose rows are the samples and $\boldsymbol{y}$ is a column vector containing the corresponding labels.

The objective function is convex, thus we can also use gradient descent to find the global optimum. With step size $\frac{1}{2}$, gradient descent converges linearly. For an ε, we can find x_t so that $f(x_t) - \min_x(f(x)) \leq \varepsilon$ in time $\mathcal{O}(\log \frac{1}{\varepsilon})$.

The gradient in the 1-dimensional case is given as

$$\nabla \hat{R}(w) = -2 \sum_{i=1}^{n} r_i \cdot x_i$$

and in the d-dimensional case as

$$\nabla \hat{R}(\boldsymbol{w}) = -2 \sum_{i=1}^{n} r_i \cdot \boldsymbol{x}_i^T = 2\boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{X} - 2\boldsymbol{y}^T\boldsymbol{X}$$

. Note that $\hat{R}(\boldsymbol{w}) \in \mathbb{R}^{1\times d}$.

The closed-form solution takes $\mathcal{O}(n^3)$ time. The gradient descent solution takes $\mathcal{O}(n \cdot d \log \frac{1}{\varepsilon})$ time.

Thus, gradient descent might converge faster. Furthermore, an optimal solution might not be required, and many problems do not admit a closed-form solution.

Instead of the squared error, other loss functions might be useful:

$$\hat{R}(\boldsymbol{w}) = \sum_{i=1}^{n} l_p(y_i - \boldsymbol{w}^T\boldsymbol{x}_i)$$

where $l_p = |r|^p$.

l_p loss is convex for $p \geq 1$. The l_1 loss puts less emphasis on large outliers. For a large p, the results are restricted to a region. For $p < 1$, the function becomes non-convex but is even more robust towards outliers.

The family of L_p norms is defined as

$$\|\boldsymbol{w}\|_p = \sqrt[p]{\sum_{i=1}^{d} |w_i|^p}$$

and their partial derivative

$$\frac{\partial \|\boldsymbol{w}\|_p^p}{\partial w_j} = p|w_j|^{p-1} \cdot sign(w_j)$$

Feature Transformations

We can fit non-linear functions via linear regression by using non-linear basis functions of the data:

$$f(\boldsymbol{x}) = \sum_{i=1}^{D} w_i \phi_i(\boldsymbol{x})$$

with $\phi : \mathbb{R}^d \to \mathbb{R}^D$. ϕ is called a feature map.

The model is still linear, but on the feature map, not on the original features.

To fit polynomial functions of d-dimensions, $\phi(\boldsymbol{x})$ is the vector of all monomials of degree up to k in variables $x_1, \ldots, x_d$. Monomials are polynomials with only one term and coefficient 1.

For a periodic function

$$y(t) = a_0 + \sum_{n=1}^{\infty} (a_n \cos nt + b_n \sin nt) + \varepsilon$$

we can use a feature map

$$\phi(t) = [1, \cos t, \sin t, \cos 2t, \sin 2t, \ldots]$$

Ridge Regression

Least squares regression is not strongly convex, prone to overfitting and may be ill-behaved if some features are correlated. The optimisation problem can be made strongly convex by adding a regularisation term $\lambda > 0$:

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 + \lambda \|\boldsymbol{w}\|_2^2$$

This results in the new empirical risk

$$\hat{R}_{ridge}(\boldsymbol{w}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2$$

The scale of $\boldsymbol{x}$ now matters! If the features are scaled, it changes the scale of λ. Thus, the data has to be standardised.

The closed-form solution is

$$\hat{\boldsymbol{w}} = (\boldsymbol{X}^T \boldsymbol{X} + n\lambda \boldsymbol{I})^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

The gradient is TODO: Is this really correct? (Derivative/gradient)

$$\nabla_w \hat{R}_{ridge}(\boldsymbol{w}) = \nabla_w \hat{R}(\boldsymbol{w}) + \lambda \nabla_w \|\boldsymbol{w}\|_2^2 = 2\boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{X} - 2\boldsymbol{y}^T \boldsymbol{X} + 2\boldsymbol{w}^T$$

resulting in an update step

$$\boldsymbol{w}_{t+1} = (1 - 2\lambda\eta_t)\boldsymbol{w}_t - \eta_t \nabla \hat{R}(\boldsymbol{w})$$

Ridge regression is also called L_2-regularisation or weight decay (as it decays weights to zero). If $\lambda \to \infty$ then $w \to 0$.

Lasso Regression

Lasso regression is L_1 regularised least squares linear regression.

The objective then becomes

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 + \lambda \|\boldsymbol{w}\|_1$$

The penalty, in theory, encourages coefficients to be exactly zero. This results in some form in automatic feature selection.

While in ridge regression, weights decay relatively smoothly, in lasso regression, some weights may heavily increase with increasing λ. This makes intuitive sense because one weight may suddenly "perform the work" that multiple weights did before.

Kernelised Linear Regression

In kernelised linear regression, regularisation is already built-in.

Let $\boldsymbol{K}$ be the Gram matrix and $\boldsymbol{k}_i$ the i-th column of the Gram matrix.

The kernelised objective for the squared loss is

$$\left(\sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_j, \boldsymbol{x}_i) - y_i \right)^2 = (\boldsymbol{\alpha}^T \boldsymbol{k}_i - y_i)^2$$

and for the L_2 regulariser

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha}$$

The kernelised optimisation objective is then

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \frac{1}{n} \underbrace{\sum_{i=1}^{n} (\boldsymbol{\alpha}^T \boldsymbol{k}_i - y_i)^2}_{\|\boldsymbol{\alpha}^T \boldsymbol{K} - \boldsymbol{y}\|_2^2} + \lambda \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha}$$

The closed-form solution (with regulariser) is given as

$$\hat{\boldsymbol{\alpha}} = (\boldsymbol{K} + \lambda \boldsymbol{I})^{-1} \boldsymbol{y}$$

For a data point $\boldsymbol{x}$, the prediction is

$$\hat{y} = \sum_{i=1}^{n} \hat{\alpha}_i k(\boldsymbol{x}_i, \boldsymbol{x})$$

Loss Functions

TODO: Functions from tutorial 3 TODO: Tutorial 3, 3.1

2.2 Perceptron

The general idea is to fit a model $h : \mathbb{R}^d \to \{+1, -1\}$ where $+1$ denotes the positive class, -1 the negative class: $h(\boldsymbol{x}) = sign(\boldsymbol{w}^T \boldsymbol{x})$.

In linear classification, the assumption is that the classification boundary is a hyperplane going through the origin.

The $\ell_{0/1}$ loss is defined as follows

$$\ell_{0/1}(\boldsymbol{w}; \boldsymbol{x}_i, y_i) = \begin{cases} 1 & \text{if } y_i \neq sign(\boldsymbol{w}^T \boldsymbol{x}_i) \\ 0 & \text{otherwise} \end{cases}$$

The goal is to minimise the $\ell_{0/1}$ loss. However, it is not convex and not differentiable. Thus, a surrogate loss is needed.

The perceptron loss is a convex surrogate for the $\ell_{0/1}$ loss, defined as

$$\ell_p(\boldsymbol{w}; \boldsymbol{x}, y) = \max(0, -y\boldsymbol{w}^T \boldsymbol{x})$$

and is a surrogate for the $\ell_{0/1}$ loss. It is 0 if the sign of the prediction and y match, and linear in the misclassification otherwise.

The gradient of the perceptron loss for a single sample is

$$\nabla_{\boldsymbol{w}} \ell_p(\boldsymbol{w}; \boldsymbol{x}_i, y_i) = \begin{cases} 0 & \text{if } \boldsymbol{w}^T \boldsymbol{x}_i y_i \geq 0 \\ -y_i \boldsymbol{x}_i & \text{if } \boldsymbol{w}^T \boldsymbol{x}_i y_i < 0 \end{cases}$$

Thus, the gradient of the empirical risk of the perceptron loss is

$$\nabla_{\boldsymbol{w}} \hat{R}(\boldsymbol{w}) = -\frac{1}{n} \sum_{i:(\boldsymbol{x}_i, y_i)\text{incorrect}} y_i \boldsymbol{x}_i$$

The perceptron algorithm performs stochastic gradient descent (1 sample at a time) on the perceptron loss function ℓ_p with learning rate 1. In practice, a different learning rate is used because the data is not usually linearly separable.

Theorem *If the data is linearly separable, the perceptron will obtain a linear separator.*

Kernelised Perceptron

The reformulation of the perceptron in terms of inner products is

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, -\sum_{j=1}^{n} \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j\}$$

and the kernelised objective

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, -\sum_{j=1}^{n} \alpha_j y_i y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)\}$$

Prediction for sample $\boldsymbol{x}$ is done as

$$\hat{y} = sign\left(\sum_{j=1}^{n} \alpha_j y_j k(\boldsymbol{x}_j, \boldsymbol{x})\right)$$

Training is very similar to the original formulation. If the wrong class for sample $\boldsymbol{x}_i$ is predicted, the update is

$$\alpha_{t+1,i} \leftarrow \alpha_{t,i} + \eta_t$$

2.3 SVM

Support vector machines perform maximum margin linear classification. Intuitively, this means to maximise the distance between the decision boundary and both classes. Formally, the margin is the euclidian distance between two hyperplanes parallel to the decision boundary.

The points which defined the distance from the decision boundary are called the support vectors.

General SVMs

SVM uses the Hinge loss which is defined as follows:

$$\ell_H(\boldsymbol{w}; \boldsymbol{x}, y) = \max(0, 1 - y\boldsymbol{w}^T\boldsymbol{x})$$

It only becomes zero if a sample is classified correctly with a certain "confidence". The Hinge loss is a convex upper-bound on the zero-one loss.

The constant 1 is used in the Hinge loss to ensure that weights do not explode. Any other constant can be compensated by simply scaling the weights accordingly.

The complete SVM objective is

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\} + \lambda \|\boldsymbol{w}\|_2^2$$

Both the Hinge loss and the regularisation term are required for the SVM to work. Furthermore, features have to be standardised. TODO: Do features really have to be standardised?

The gradient of the Hinge loss is

$$\nabla_{\boldsymbol{w}} \ell_H(\boldsymbol{w}; \boldsymbol{x}, y) = \begin{cases} 0 & \text{if } \boldsymbol{w}^T \boldsymbol{x}_i y_i \geq 1 \\ -y_i \boldsymbol{x}_i & \text{if } \boldsymbol{w}^T \boldsymbol{x}_i y_i < 1 \end{cases}$$

Thus, the gradient of the empirical risk for SVMs is

$$\nabla_{\boldsymbol{w}} \hat{R}(\boldsymbol{w}) = -\frac{1}{n} \sum_{i:\boldsymbol{w}^T \boldsymbol{x}_i y_i < 1} y_i \boldsymbol{x}_i + 2\boldsymbol{w}$$

If the data is linearly separable, SVM finds the maximum margin decision boundary.

A safe learning rate choice is $\eta_t = \frac{1}{\lambda t}$, i.e. use learning rate decay based on the regularisation parameter λ.

L_1-SVM

In L_1-SVM, the penalty term is replaced with a L_1 loss, thus the objective becomes

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\} + \lambda \|\boldsymbol{w}\|_1$$

The alternative penalty term encourages coefficients to be exactly zero and thus results in automatic feature selection.

Geometric Interpretation

TODO: Honestly, just read tutorial 4.

A *hyperplane* in $\mathbb{R}^d$ is the set of vectors fulfilling $\langle \boldsymbol{w}, \boldsymbol{x} \rangle = 0$. It is a $d-1$ dimensional subspace. For $b \in \mathbb{R}$, the set $\{\boldsymbol{x} \in \mathbb{R}^d \mid \langle \boldsymbol{w}, \boldsymbol{x} \rangle = b\}$ is a *translated/affine hyperplane*. Increasing b moves the hyperplane towards the direction of $\boldsymbol{w}$ and vice versa.

The *orthogonal projection* of $\boldsymbol{z}$ onto a hyperplane is given by $\hat{\boldsymbol{z}} = \boldsymbol{z} + t\boldsymbol{w}$ where $t = \frac{b - \langle \boldsymbol{w}, \boldsymbol{z} \rangle}{\|\boldsymbol{w}\|^2}$.

The *orthogonal distance* of $\boldsymbol{z}$ to a hyperplane $H = \{\boldsymbol{x} \in \mathbb{R}^d \mid \langle \boldsymbol{w}, \boldsymbol{x} \rangle = 0\}$ is given by

$$\frac{|\langle \boldsymbol{w}, \boldsymbol{z} \rangle|}{\|\boldsymbol{w}\|^2}$$

If $\boldsymbol{w}$ is a unit vector, the distance is directly given by $|\langle \boldsymbol{w}, \boldsymbol{z} \rangle|$.

From now on, $\boldsymbol{z}$ contains an additional entry 1 and $\boldsymbol{w}$ contains an additional entry $-b$. Therefore, all hyperplanes are through the origin.

Let D be a data set and H a hyperplane. The *margin* of H w.r.t. D is

$$\gamma_D(\boldsymbol{w}) := \min_{(\boldsymbol{x}, y) \in D} \frac{|\langle \boldsymbol{w}, \boldsymbol{x} \rangle|}{\|\boldsymbol{w}\|}$$

We now assume that

1. The data is linearly separable.
2. There exists a separating hyperplane with non-zero margin.

The *Hard-SVM* problem requires both assumptions. It is minimising

$$\frac{1}{2}\|\boldsymbol{w}\|^2$$

such that

$$y_i \cdot \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \geq 1 \text{ for all } i \in \{1, \ldots, n\}$$

The separating hyperplane assumption can be dropped by introducing *slack variables* $\xi_1, \ldots, \xi_n \geq 0$ which allow to violate some constraints. ξ_i denotes the amount the i-th constraint is violated.

This leads to the *Soft-SVM* problem. It is minimising

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{n} \xi_i$$

such that

$$\begin{aligned} y_i \cdot \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle &\geq 1 - \xi_i && \text{for all } i \in \{1, \ldots, n\} \\ \xi_i &\geq 0 && \text{for all } i \in \{1, \ldots, n\} \end{aligned}$$

where $C > 0$ denotes the softness of the problem.

If $C = 0$ then $\boldsymbol{w} = \boldsymbol{0}$ is a trivial solution; if $C \to \infty$ then the problem becomes similar to the Hard-SVM problem.

Fixing $\boldsymbol{w}$ and optimising ξ_i leads to the solution

$$\xi_i^*(\boldsymbol{w}) = \ell_H(y_i \cdot \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle)$$

and by plugging it in to the equivalent optimisation objective

$$\arg\min_{\boldsymbol{w}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{n} \ell_H(y_i \cdot \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle)$$

Kernelised SVM

The reformulisation of the SVM loss (without regulariser) with respect to inner products and kernels for sample i is

$$\max\{0, 1 - y_i \sum_{j=1}^{n} \alpha_j y_j k(\boldsymbol{x}_j, \boldsymbol{x}_i)\}$$

and for the regulariser

$$\lambda \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \lambda \boldsymbol{\alpha}^T \boldsymbol{D}_y \boldsymbol{K} \boldsymbol{D}_y \boldsymbol{\alpha}$$

where $\boldsymbol{K}$ is the Gram matrix and $\boldsymbol{D}_y$ contains the y_i on its diagonal.

This leads to the new objective

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y_i \boldsymbol{\alpha}^T \boldsymbol{k}_i\} + \lambda \boldsymbol{\alpha}^T \boldsymbol{D}_y \boldsymbol{K} \boldsymbol{D}_y \boldsymbol{\alpha}$$

where $\boldsymbol{k}_i = [y_1 k(\boldsymbol{x}_i, \boldsymbol{x}_1), \ldots, y_n k(\boldsymbol{x}_i, \boldsymbol{x}_n)]$.

Prediction for a data point $\boldsymbol{x}$ is done as

$$\hat{y} = sign\left(\sum_{i=1}^{n} \alpha_i y_i k(\boldsymbol{x}_i, \boldsymbol{x}) \right)$$

Multi-Class SVM

SVMs can be generalised to a multi-class scenario. The key idea is to maintain c weight vectors $\boldsymbol{w}^{(i)}$, one for each class. The prediction is then

$$\hat{y} = \arg\max_i \boldsymbol{w}^{(i)^T} \boldsymbol{x}$$

For each data point, it should hold that the prediction for the true class is separated by a margin from the class which has the second highest prediction, i.e.

$$\boldsymbol{w}^{(y)^T} \boldsymbol{x} \geq \max_{i \neq y} \boldsymbol{w}^{(i)^T} \boldsymbol{x} + 1$$

The multi-class Hinge loss is given as

$$\ell_{MC-H}(\boldsymbol{w}^{(1)}, \ldots, \boldsymbol{w}^{(c)}; \boldsymbol{x}, y) = \max\{0, 1 + \max_{j \neq y} \boldsymbol{w}^{(j)^T} \boldsymbol{x} - \boldsymbol{w}^{(y)^T} \boldsymbol{x}\}$$

The gradient is

$$\nabla_{\boldsymbol{w}^{(i)}} \ell_{MC-H}(\boldsymbol{w}^{(1:c)}; \boldsymbol{x}, y) = \begin{cases} 0 & \text{if } \boldsymbol{w}^{(y)^T} \boldsymbol{x} \geq \max_{i \neq y} \boldsymbol{w}^{(i)^T} \boldsymbol{x} + 1 \text{ or } i \neq y \text{ and } i \notin \arg\max_j \boldsymbol{w}^{(j)^T} \boldsymbol{x} \\ -x & \text{if } \boldsymbol{w}^{(y)^T} \boldsymbol{x} < \max_{i \neq y} \boldsymbol{w}^{(i)^T} \boldsymbol{x} + 1 \text{ and } i = y \\ +x & \text{otherwise} \end{cases}$$

i.e. zero if either the classification is correct or the weight does neither correspond to the class nor to the maximal prediction for the sample; minus x if the weight corresponds to the class and does not classify correctly; plus x if the weight does not correspond to the class but is the maximum prediction.

2.4 k-Nearest Neighbour

For a data point $\boldsymbol{x}$, predict the majority of labels of its k nearest neighbours. Nearness is measured according to some distance function.

The k-nearest neighbour classifier can be viewed as an alternative to the perceptron where the perceptron does not use a fixed k but chooses the neighbour's influence smoothly according to the kernel.

The KNN classifier can thus for example not capture global trends and depends on all data, not only wrongly classified samples.

Formally, the prediction is defined as

$$y = sign\left(\sum_{i=1}^{n} y_i \cdot [\boldsymbol{x}_i \text{ among } k \text{ nearest neighbours of } \boldsymbol{x}]\right)$$

2.5 Neural Networks

Using kernel methods has the advantage of resulting in rich feature maps and being able to fit any function with infinite data. However, choosing the right kernel is challenging and the computational complexity grows with the size of the data.

The key idea is to parameterise the feature maps. Thus, instead of finding

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \ell\left(y_i; \sum_{j=1}^{m} w_j \phi_j(\boldsymbol{x}_i)\right)$$

we also optimise over the parameters:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w},\boldsymbol{\theta}} \sum_{i=1}^{n} \ell\left(y_i; \sum_{j=1}^{m} w_j \phi(\boldsymbol{x}_i, \theta_j)\right)$$

The feature maps in neural networks are of the form

$$\phi(\boldsymbol{x}, \boldsymbol{\theta}) = \varphi(\underbrace{\boldsymbol{\theta}^T \boldsymbol{x}}_{z})$$

where $\boldsymbol{\theta} \in \mathbb{R}^d$ and $\varphi : \mathbb{R} \to \mathbb{R}$ is an activation function.

The term *artificial neural networks (ANN) / multi-layer Perceptrons* refers to nonlinear functions which are nested compositions of (variable) linear functions with (fixed) nonlinearities.

The *Universal Approximation Theorem* states that, TODO: UAT

Input layer refers to the values of $\boldsymbol{x}$. *Output layer* refers to the outputs of the network and is denoted by $\boldsymbol{f}$. *Hidden layers* are of the form

$$v_i^{(l)} = \varphi(\boldsymbol{w}_i^{(l)^T} \boldsymbol{v}^{l-1})$$

$v_i^{(l)}$ denotes the i-th entry of the l-th hidden layer ($\boldsymbol{v}^{(0)} = \boldsymbol{x}$). $w_{i,j}^{(l)}$ refers to the weight connecting unit i of layer l to unit j of the previous layer $l-1$.

The number of layers (excluding input) is written as L (there are $L-1$ hidden layers).

Units are indexed by indexing first within the units of a layer and then layer by layer.

Activation Functions

Logistic Sigmoid

The *(logistic) Sigmoid* activation function is

$$\varphi(z) = \frac{1}{1+\exp(-z)} = \frac{\exp(z)}{\exp(z)+1}$$

It has the range $\varphi : \mathbb{R} \to (0,1)$ and $\varphi(0) = 0.5$. It behaves almost linear between some threshold, almost constant outside of it.

A sigmoidal function is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point. Often, the logistic function (Sigmoid activation) is implied.

The derivative is

$$\varphi'(z) = \frac{\exp(-z)}{1+\exp(-z)} \cdot \frac{1}{1+\exp(-z)} = \varphi(z) \cdot (1-\varphi(z))$$

It is approximately zero unless z is approximately zero, thus often leading to vanishing gradients for deep models.

tanh

The *tanh* activation function is a version of Sigmoid, centred around zero, given as

$$\varphi(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

It has the range $\varphi : \mathbb{R} \to (-1,1)$ and $\varphi(0) = 0$.

It holds that $\varphi(z) = -\varphi(-z)$.

ReLU

The *ReLU* activation is given as

$$\varphi(z) = \max(z, 0)$$

i.e. it is zero for negative values, linear otherwise.

The derivative is

$$\varphi'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

It is not differentiable at zero.

Leaky ReLU

The *leaky ReLU* activation is given as

$$\varphi(z) = \max(z, \alpha z)$$

with $\alpha < 1$ being small, e.g. 10^{-3}. Its advantage is that the gradient does not vanish for $z < 0$.

The derivative, assuming $0 < \alpha < 1$, is

$$\varphi'(z) = \begin{cases} 1 & \text{if } z > 0 \\ \alpha & \text{otherwise} \end{cases}$$

It is not differentiable at zero.

Forward Propagation

Forward propagation performs the calculations layer by layer. It can be written in matrix notation by combining the $\boldsymbol{w}_i^{(l)}$ into a matrix

$$\boldsymbol{W}^{(l)} = \begin{pmatrix} \boldsymbol{w}_1^{(l)} \\ \vdots \\ \boldsymbol{w}_m^{(l)} \end{pmatrix}$$

such that $w_{i,j}^{(l)} = \boldsymbol{W}_{i,j}^{(l)}$.

Forward propagation in matrix notation starts with $\boldsymbol{v}^{(0)} = \boldsymbol{x}$.

Then, for each hidden layer $l = 1, \ldots, L-1$, calculate

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{v}^{(l-1)}$$
$$\boldsymbol{v}^{(l)} = \varphi(\boldsymbol{z}^{(l)})$$

where φ is applied element-wise.

Then, calculate the output layer as

$$\boldsymbol{f} = \boldsymbol{W}^{(L)} \boldsymbol{v}^{(L-1)}$$

For regression, $\hat{\boldsymbol{y}} = \boldsymbol{f}$ is picked. For binary classification, $\hat{y} = sign(f)$ is used, for multi-class $\hat{y} = \arg\max_i f_i$.

Backpropagation

In order to apply stochastic gradient descent, $\nabla_{\boldsymbol{W}} \ell(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x})$ or

$$\frac{\partial}{\partial w_{i,j}} \ell(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x})$$

respectively needs to be calculated.

Scalar Form

1. For each unit j on the output layer
   a) Compute the error signal
   $$\delta_j^{(L)} = \ell_j'(f_j)$$
   b) For each unit i on layer $L-1$, calculate the derivative
   $$\frac{\partial}{\partial w_{j,i}^{(L)}} = \delta_j^{(L)} v_i^{(L-1)}$$
2. For each hidden layer $l = L-1,\dots,1$, for each unit j on that layer
   a) Compute the error signal
   $$\delta_j^{(l)} = \varphi'(z_j^{(l)}) \cdot \sum_{i \in Layer_{l+1}} w_{i,j}^{(l)} \delta_i^{(l+1)}$$
   b) For each unit i on layer $l-1$, compute the derivative
   $$\frac{\partial}{\partial w_{j,i}^{(l)}} = \delta_j^{(l)} v_i^{(l-1)}$$

Matrix Form

1. For the output layer
   a) Compute the error
   $$\boldsymbol{\delta}^{(L)} = \boldsymbol{\ell}'(\boldsymbol{f}) = [\ell'(f_1),\dots,\ell'(f_p)]$$
   b) Calculate the gradient
   $$\nabla_{\boldsymbol{W}^{(L)}} \ell(\boldsymbol{W};\boldsymbol{y},\boldsymbol{x}) = \boldsymbol{\delta}^{(L)} \boldsymbol{v}^{(L-1)^T} \in \mathbb{R}^{p \times m}$$
2. For each hidden layer $l = L-1,\dots,1$
   a) Compute the error
   $$\boldsymbol{\delta}^{(l)} = \varphi'(\boldsymbol{z}^{(l)}) \underbrace{\odot}_{\text{pointwise multiplication}} (\boldsymbol{W}^{(l+1)^T} \boldsymbol{\delta}^{(l+1)})$$
   b) Compute the gradient
   $$\nabla_{\boldsymbol{W}^{(l)}} \ell(\boldsymbol{W};\boldsymbol{y},\boldsymbol{x}) = \boldsymbol{\delta}^{(l)} \boldsymbol{v}^{(l-1)^T} \in \mathbb{R}^{m' \times m''}$$

Training

Given a data set D we want to optimise the weights $\boldsymbol{W} = (\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L)})$.

For a given loss function $\ell(\boldsymbol{W}; \boldsymbol{x}, \boldsymbol{y})$, we want to do empirical risk minimisation, i.e. jointly optimise over all weights:

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} \underbrace{\sum_{i=1}^{n} \ell(\boldsymbol{W}; \boldsymbol{y}_i, \boldsymbol{x}_i)}_{\hat{R}(\boldsymbol{W})}$$

In a multi-output scenario, usually the sum of per-output losses is taken:

$$\ell(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x}) = \sum_{i=1}^{p} \ell_i(\boldsymbol{W}; y_i, \boldsymbol{x})$$

e.g. the sum of squared errors for regression.

Joint optimisation over all weights for all layers is generally a non-convex optimisation problem. However, a local optimum can still be found.

Weight Initialisation

The optimisation problem is non-convex, thus weight initialisation matters.

If all weights are initialised to 0, there will be no gradient. If the weights are initialised too small, the gradient can become arbitrarily small. Conversely, if the initial weights are too large, the gradients may explode.

In practice, random initialisation with carefully chosen randomness is done.

Theorem *Let X, Y be independent random variables with $\mathbb{E}[X] = 0$. Then*

$$Var[XY] = \begin{cases} Var[X] \cdot Var[Y] & \text{if } \mathbb{E}[Y] = 0 \\ Var[X] \cdot \mathbb{E}[Y^2] & \text{if } \mathbb{E}[Y] \neq 0 \end{cases}$$

If φ is the ReLU function and Y is symmetric, then

$$\mathbb{E}[\varphi(Y)^2] = \frac{1}{2} Var[Y]$$

TODO: Propagation of Variance, Slide 9 Week 13

The goal of weight initialisation is to keep the variance approximately constant to avoid vanishing and exploding gradients.

Glorot Initialiser (tanh)

Either use

$$w_{i,j} \sim \mathcal{N}\left(0, \frac{1}{n_{in}}\right)$$

or

$$w_{i,j} \sim \mathcal{N}\left(0, \frac{2}{n_{in} + n_{out}}\right)$$

He Initialiser (ReLU)

$$w_{i,j} \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right)$$

Learning Rates and Momentum

There are many different ways to choose a learning rate.

Learning rate decay starts with a small learning rate α and decreases it over time, e.g.

$$\eta_t = \min(\alpha_{min}, \alpha/t)$$

Learning rate schedules use a piecewise constant learning rate, i.e. decrease it after some number of epochs.

Momentum can help escaping local minima and can prevent oscillation. The key idea is to not only move in direction of the gradient, but also of the last weight update. For a momentum parameter $m < 1$ and a stored weight update $\boldsymbol{a}$, the new update is

$$\begin{aligned} \boldsymbol{a} &\leftarrow m \cdot \boldsymbol{a} + \eta_t \nabla_{\boldsymbol{W}} \ell(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x}) \\ \boldsymbol{W} &\leftarrow \boldsymbol{W} - \boldsymbol{a} \end{aligned}$$

Regularisation

Neural networks have a large number of parameters and are thus very prone to overfitting. The fact that (mini batch) SGD is used already provides some form of regularisation but there are further countermeasures.

Weight Penalties

A penalty term can be added to the loss to keep weights small:

$$\hat{\boldsymbol{W}} = \arg\min_{\boldsymbol{W}} \sum_{i=1}^{n} \ell(\boldsymbol{W}; \boldsymbol{y}_i, \boldsymbol{x}_i) + \lambda \|\boldsymbol{W}\|_2^F$$

Early Stopping

In general, training should not be ran until the weights converge.

One possibility is to monitor the prediction performance on a test set and stopping as soon as the validation error starts to increase.

Dropout

The key idea is to randomly ignore (drop out) hidden units during training. This can encourage sparse activations.

A parameter p specifies a probability with which a unit is kept. During each training iteration, each unit is set to 0 with probability $(1 - p)$. After training, the weights are downscaled to $p \cdot \boldsymbol{w}$ to ensure the expected value of the activations during training matches the expectation of activations during prediction.

Batch Normalisation

Training is usually stabilised by ensuring that training inputs ($\boldsymbol{x}$) are standardised. However, in deep learning, inputs are shifted and scaled through each layer. Batch normalisation normalises inputs again after each layer according to mini-batch statistics.

It reduces the internal covariate shift (not proven), enables larger learning rates and has some regularising effects.

Batch normalisation can be plugged into an existing model as

$$\varphi(\boldsymbol{W}\boldsymbol{x}) \longrightarrow \varphi(\boldsymbol{W}BN_{\gamma,\beta}(\boldsymbol{x}))$$

A batch normalisation layer $\boldsymbol{y}_i = BN_{\gamma,\beta}(\boldsymbol{x}_i)$ over a mini batch $\mathcal{B}$ with learned parameters γ, β works as follows:

1. $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}_i$
2. $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{x}_i - \mu_{\mathcal{B}})^2$
3. $\hat{\boldsymbol{x}}_i \leftarrow \frac{\boldsymbol{x}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$
4. $\boldsymbol{y}_i \leftarrow \gamma\hat{\boldsymbol{x}}_i + \beta$

Parameters γ and β allow to compensate for the scaling effect. They in principle allow to undo the normalisation. Those two parameters are also learned. They allow to explore the spectrum between normalised and unnormalised data.

Losses

For binary classification, we can use the *logistic loss* function to predict probabilities:

$$\ell_{\text{logistic}}(\boldsymbol{w}; \boldsymbol{x}, y) = \log\left(1 + \exp\left(-y\boldsymbol{w}^T\boldsymbol{x}\right)\right)$$

TODO: Binary cross-entropy loss.
TODO: Softmax
TODO: Cross-entropy loss.

Convolutional Neural Networks

Predictions should be unchanged under some data transformations, i.e. remain invariant. This could be achieved by data augmentation, specialised regularisation terms, invariance built into preprocessing, or by building it into the network structure. Convolutional neural networks are an example of the last one.

The general idea is that hidden layers share parameters, i.e. each hidden unit only depends on a close region of inputs around it, and weights are identical across all units on the layer. This reduces the number of parameters and encourages robustness against small translations.

The following is common terminology:

Filters/kernels are the values used for convolution and correspond to the weights.

Stride is the step size between two filter applications.

Padding is a border around the input image to control the output dimension.

The results for a single pixel is the inner product of the filter and the corresponding image patch, centred on that pixel. There are multiple filters per layer, applied in parallel and in full input depth (e.g. for images, filters are usually 3D).

Usually, zero-padding is used, i.e. padding the image with 0s.

If m different $f \times f$ filters are applied to an $n \times n$ image with padding p and stride s, the output dimensions are

$$\frac{n + 2p - f}{s} + 1$$

Pooling takes a region of a layer output and aggregates it to a single value, e.g. the maximum or average. This makes sense in many scenarios as it condenses global information and reduces the number of parameters.

(Early) CNN architectures generally follow the pattern: First, convolutions followed by pooling layers are computed. As the input reaches a certain size, the activations are flattened into a vector. Then, some fully connected layers are applied until the output is calculated.

Connection to Kernels

An ANN with a single hidden layer using the tanh activation learns functions of the form

$$f(\boldsymbol{x}) = \sum_{i=1} w_i \cdot tanh(\boldsymbol{\theta}_i^T \boldsymbol{x})$$

This is exactly the same type of functions learned by using the *tanh kernel*:

$$f(\boldsymbol{x}) = \sum_{i=1} \alpha_i \cdot tanh(\boldsymbol{x}_i^T \boldsymbol{x})$$

The advantage of ANNs is that they are flexible nonlinear models which may discover representations at multiple levels of abstractions. However, they suffer from noisy data and have many architectural challenges. Kernel methods are convex and robust against noise, but the model grows with data and does not allow for multiple layers.

2.6 k-means Clustering

Given a set of data points, they should be grouped into clusters of similar points. Points are typically represented either in high-dimensional Euclidean space or with distances specified by a metric/kernel.

k-means clustering is a model-based approach, i.e. it maintains cluster models and infers membership.

Each cluster is represented by a single *cluster centroid* $\boldsymbol{\mu}_i \in \mathbb{R}^d$. A point is assigned to its closest cluster centroid. This induces a Voronoi partition of the input space. Therefore, decision boundaries are linear and the problem can be interpreted as linear unsupervised classification.

k-means Problem

Points are represented in Euclidean space: $\boldsymbol{x}_i \in \mathbb{R}^d$. Clusters are represented by their centres: $\boldsymbol{\mu}_j \in \mathbb{R}^d$. Each point is assigned to its closest centre.

The goal is to pick centres to minimise the average squared distance. Thus, the empirical risk is given as

$$\hat{R}(\boldsymbol{\mu}) = \hat{R}(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k) = \sum_{i=1}^{n} \min_{j \in \{1,\ldots,k\}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|_2^2$$

and the resulting optimisation problem is

$$\hat{\boldsymbol{\mu}} = \arg\min_{\boldsymbol{\mu}} \hat{R}(\boldsymbol{\mu})$$

This is a non-convex objective. Furthermore, the problem is NP-hard, thus cannot be solved optimally in general.

k-means Algorithm / Lloyd's Heuristic

The k-means problem could be solved by SGD. However, there exists a heuristic for solving it without gradient-based methods. *Lloyd's heuristic* is an algorithm to solve the k-means problem approximately.

1. Initialise cluster centres

$$\boldsymbol{\mu}^{(0)} = [\boldsymbol{\mu}_1^{(0)}, \ldots, \boldsymbol{\mu}_k^{(0)}]$$

2. While not converged

   a) Assign each point $\boldsymbol{x}_i$ to closest centre:

$$z_i^{(t)} = \arg\min_{j \in \{1,\ldots,k\}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t-1)}\|_2^2$$

   b) Update centres as the mean of assigned data points:

$$\boldsymbol{\mu}_j^{(t)} = \frac{1}{n_j} \sum_{i:z_i^{(t)}=j} \boldsymbol{x}_i$$

   where $n_j = |\{i \mid z_i^{(t)} = j\}|$.

This algorithm is guaranteed to monotonically decrease the average squared distance in each iteration, i.e.

$$\hat{R}(\boldsymbol{\mu}^{(t)}) \leq \hat{R}(\boldsymbol{\mu}^{(t-1)})$$

It has runtime complexity of $\mathcal{O}(n \cdot k \cdot d)$ per iteration.

k-means++

k-means generally converges to a local optimum and its performance thus strongly depends on initialisation.

One way to solve this is to use *multiple random restarts* and taking the best solution. However, this prefers large clusters with many data points. Also, one could use a *farthest point heuristic*. However, that is prone to outliers.

k-means++ is an initialisation scheme which tries to combat both problems.

1. Start with random data point as centre:

$$i_1 \sim Uniform(\{1, \ldots, n\})$$
$$\boldsymbol{\mu}_1^{(0)} = x_{i_1}$$

2. For $j = 2 : k$

   a) Pick a data point as a centre randomly with probability proportional to the closest selected centre, i.e. pick i_j with probability

$$\frac{1}{z} \cdot \left(\min_{l \in \{1,\dots,j-1\}} \|\boldsymbol{x}_{i_j} - \boldsymbol{\mu}_l^{(0)}\|_2^2 \right)$$

   b) Set the data point as centre:

$$\boldsymbol{\mu}_j^{(0)} = \boldsymbol{x}_{i_j}$$

The expected cost is $\mathcal{O}(\log k)$ times that of the optimal solution, i.e.

$$\mathbb{E}[\hat{R}(\boldsymbol{\mu}^{(0)})] \leq \mathcal{O}(\log k) \cdot \min_{\boldsymbol{\mu}} \hat{R}(\boldsymbol{\mu})$$

Model Selection

Model selecting (i.e. determining k) is very difficult in general. The problem is that it is not possible to use a validation set to measure generalisation error because the cost on the validation set typically tends to decrease too with increasing k. If no *specific domain knowledge* is available, there some other approaches.

A *heuristic approach* is using an *elbow plot*. One can plot the cost as a function of k and then look for an "elbow", i.e. a point where increasing k only leads to diminishing returns in terms of cost. The elbow function is monotonically decreasing and convex.

A *regularisation approach* is obtained by adding a regularisation term to the cost function:

$$\boldsymbol{\mu}^* = \min_{k,\boldsymbol{\mu}} \hat{R}(\boldsymbol{\mu}) + \lambda k$$

This is equivalent to the elbow method for $\varepsilon = \lambda$, i.e. using a k so that the decrease in cost is less than λ.

Challenges

The general challenges with k-means are

- Generally converges to local optimum
- Number of required iterations can be exponential
- Cannot model clusters of arbitrary shape. On Euclidean space, clusters are assumed to be spheres.
- Determining k is difficult

The first two challenges are usually not a problem in practice. The third challenge can be fixed by using kernels. However, selecting k is a major unsolved problem to date.

2.7 PCA

Let $D = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}, \boldsymbol{x}_i \in \mathbb{R}^d$ be a data set with zero mean, i.e. $\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^n \boldsymbol{x}_i = \boldsymbol{0}$.

The *PCA problem* for $k < n$ is finding

$$(\boldsymbol{W}, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = \arg\min \sum_{i=1}^n \|\boldsymbol{W}\boldsymbol{z}_i - \boldsymbol{x}_i\|_2^2$$

where $\boldsymbol{W} \in \mathbb{R}^{d\times k}$ is orthogonal and $\boldsymbol{z}_i \in \mathbb{R}^k$.

Let Σ be the empirical covariance, given as

$$\Sigma = \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X} = \frac{1}{n}\sum_{i=1}^n \boldsymbol{x}_i\boldsymbol{x}_i^T$$

The solution to the PCA problem in closed-form is given by

$$\boldsymbol{W} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k)$$
$$\boldsymbol{z}_i = \boldsymbol{W}^T\boldsymbol{x}_i$$

where Σ is decomposed as

$$\Sigma = \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^T = \sum_{i=1}^d \lambda_i \boldsymbol{v}_i\boldsymbol{v}_i^T$$

and the λ_i are ordered in descending order, $\lambda_1 \geq \cdots \geq \lambda_d \geq 0$.

Then, the linear mapping $f(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x}$ projects $\boldsymbol{x} \in \mathbb{R}^d$ to a k-dimensional subspace $\mathbb{R}^k$. It minimises the reconstruction error with respect to the Euclidian norm.

Typically, if the data set is clustered, PCA preserves those clusters.

Derivation for $k = 1$

Assume $\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^n \boldsymbol{x}_i = \boldsymbol{0}$.

The goal is to represent data points $\boldsymbol{x}_i$ on a line $\boldsymbol{w} \in \mathbb{R}^d$ with coefficients z_i. Thus, we want $z_i\boldsymbol{w} \approx \boldsymbol{x}_i$.

To get a unique solution, we further require $\boldsymbol{w}$ to be normalised, i.e. $\|\boldsymbol{w}\|_2 = 1$. Then, we can jointly optimise

$$(\boldsymbol{w}^*, \boldsymbol{z}^*) = \arg\min_{\|\boldsymbol{w}\|_2=1, \boldsymbol{z}} \underbrace{\sum_{i=1}^n \|z_i\boldsymbol{w} - \boldsymbol{x}_i\|_2^2}_{\hat{R}(\boldsymbol{w},\boldsymbol{z})}$$

For a fixed $\boldsymbol{w}$, with $\|\boldsymbol{w}\|_2 = 1$, the optimal z_i is given as the orthogonal projection of $\boldsymbol{x}_i$ onto $\boldsymbol{w}$, i.e.

$$z_i^* = \boldsymbol{w}^T\boldsymbol{x}_i$$

This leads to the equivalent optimisation objective

$$\boldsymbol{w}^* = \arg\min_{\|\boldsymbol{w}\|_2=1} \underbrace{\sum_{i=1}^{n} \|\boldsymbol{w}\boldsymbol{w}^T\boldsymbol{x}_i - \boldsymbol{x}_i\|_2^2}_{\hat{R}(\boldsymbol{w})}$$

Now, the empirical risk can be reformulated:

$$\begin{aligned}
\hat{R}(\boldsymbol{w}) &= \sum_{i=1}^{n} (\boldsymbol{w}\boldsymbol{w}^T\boldsymbol{x}_i - \boldsymbol{x}_i)^T(\boldsymbol{w}\boldsymbol{w}^T\boldsymbol{x}_i - \boldsymbol{x}_i) \\
&= \sum_{i=1}^{n} \boldsymbol{x}_i^T\boldsymbol{w}\underbrace{\boldsymbol{w}^T\boldsymbol{w}}_{=1}\boldsymbol{w}^T\boldsymbol{x}_i - 2\boldsymbol{x}_i^T\boldsymbol{w}\boldsymbol{w}^T\boldsymbol{x}_i + \boldsymbol{x}_i^T\boldsymbol{x}_i \\
&= \sum_{i=1}^{n} \underbrace{(\boldsymbol{w}^T\boldsymbol{x}_i)^T(\boldsymbol{w}^T\boldsymbol{x}_i)}_{=(\boldsymbol{w}^T\boldsymbol{x}_i)^2} - 2\underbrace{(\boldsymbol{w}^T\boldsymbol{x}_i)^T(\boldsymbol{w}^T\boldsymbol{x}_i)}_{=(\boldsymbol{w}^T\boldsymbol{x}_i)^2} + \underbrace{\boldsymbol{x}_i^T\boldsymbol{x}_i}_{=\|\boldsymbol{x}_i\|_2^2} \\
&= \underbrace{\sum_{i=1}^{n} \|\boldsymbol{x}_i\|_2^2}_{=\text{ const w.r.t. } \boldsymbol{w}} - \sum_{i=1}^{n} (\boldsymbol{w}^T\boldsymbol{x}_i)^2
\end{aligned}$$

Thus, minimising the previous objective is equivalent to maximising the new objective

$$\boldsymbol{w}^* = \arg\max_{\|\boldsymbol{w}\|_2=1} \sum_{i=1}^{n} (\boldsymbol{w}^T\boldsymbol{x}_i)^2$$

By reformulating

$$\sum_{i=1}^{n} (\boldsymbol{w}^T\boldsymbol{x}_i)^2 = \sum_{i=1}^{n} \boldsymbol{w}^T\boldsymbol{x}_i\boldsymbol{x}_i^T\boldsymbol{w} = \boldsymbol{w}^T \underbrace{\sum_{i=1}^{n} (\boldsymbol{x}_i\boldsymbol{x}_i^T)}_{=n\Sigma} \boldsymbol{w}$$

this is again equivalent to the final objective

$$\boldsymbol{w}^* = \arg\max_{\|\boldsymbol{w}\|_2=1} \boldsymbol{w}^T\Sigma\boldsymbol{w}$$

where $\Sigma = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i\boldsymbol{x}_i^T$ is the empirical covariance assuming the data is centred.

We can take the eigendecomposition of Σ and write it as $\Sigma = \sum_{i=1}^{d} \lambda_i \boldsymbol{v}_i\boldsymbol{v}_i^T$ with $\lambda_1 \geq \cdots \geq \lambda_d \geq 0$. $\boldsymbol{w}$ can also be represented in the eigenbasis of Σ as $\boldsymbol{w} = \sum_{i=1}^{d} \alpha_i \boldsymbol{v}_i$. All $\alpha_i \geq 0$ and must sum to 1, are thus a probability distribution.

Now, the objective can be rewritten as

$$\begin{aligned}
\boldsymbol{w}^T \Sigma \boldsymbol{w} &= \left(\sum_{i=1}^{d} \alpha_i \boldsymbol{v}_i \right)^T \left(\sum_{j=1}^{d} \lambda_j \boldsymbol{v}_j \boldsymbol{v}_j^T \right) \left(\sum_{k=1}^{d} \alpha_k \boldsymbol{v}_k \right) \\
&= \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{k=1}^{d} \alpha_i \alpha_k \lambda_j \underbrace{(\boldsymbol{v}_i^T \boldsymbol{v}_j)}_{\delta_{i,j}} \underbrace{(\boldsymbol{v}_j^T \boldsymbol{v}_k)}_{\delta_{j,k}} \\
&= \sum_{i=1}^{d} \alpha_i^2 \lambda_i
\end{aligned}$$

From the constraint $\|\boldsymbol{w}\|_2 = 1$ we have $\sum_{i=1}^{d} \alpha_i^2 = 1$, thus the optimal solution is given by $\alpha_1 = 1$ and $\alpha_i = 0$ for all $i > 1$ because λ_1 is the maximum value.

Therefore, the maximum is given by the principal eigenvector of Σ. This generalises for $k > 1$ principal components.

Connection to SVD

Let $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ be the data matrix, centred. Furhtermore, let $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$ be the SVD of $\boldsymbol{X}$ where $\boldsymbol{U} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{V} \in \mathbb{R}^{d \times d}$ are orthogonal, and $\boldsymbol{S} \in \mathbb{R}^{n \times d}$ contains the singular values in decreasing order on its diagonal.

The top k principal components are then the first k columns of $\boldsymbol{V}$:

$$n\Sigma = \boldsymbol{X}^T \boldsymbol{X} = \boldsymbol{V}\boldsymbol{S}^T \underbrace{\boldsymbol{U}^T \boldsymbol{U}}_{=\boldsymbol{I}} \boldsymbol{S}\boldsymbol{V}^T = \boldsymbol{V} \underbrace{\boldsymbol{S}^T \boldsymbol{S}}_{=\boldsymbol{D}} \boldsymbol{V}^T$$

Model Selection

Cross-validation can be used to select the parameter k by evaluating the cost.

Another approach is to pick k so that most of the variance is explained. TODO: How?

Comparison to k-means

There exists a equivalent formulation for the k-means problem which is very similar to the PCA problem.

The PCA problem is

$$(\boldsymbol{W}, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = \arg\min \sum_{i=1}^{n} \|\boldsymbol{W}\boldsymbol{z}_i - \boldsymbol{x}_i\|_2^2$$

where $\boldsymbol{W} \in \mathbb{R}^{d \times k}$ is orthogonal and $\boldsymbol{z}_i \in \mathbb{R}^k$.

The k-means problem is

$$(\boldsymbol{W}, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = \arg\min \sum_{i=1}^{n} \|\boldsymbol{W}\boldsymbol{z}_i - \boldsymbol{x}_i\|_2^2$$

where $\boldsymbol{W} \in \mathbb{R}^{d \times k}$ is arbitrary and $\boldsymbol{z}_i \in E_k$ where E_k is the set of unit vectors in $\mathbb{R}^k$.

The discrete nature of the k-means problem makes it NP-hard. Conversely, PCA can be used to initialise k-means cluster centres.

The line between the centroids for $k = 2$ approximates the first principal component, the plane between the centroids for $k = 3$ the first two principal components, etc.

Kernelised PCA

For non-linear problems, PCA can be kernelised.

Let $k \geq 1$ be the target dimensionality and $\boldsymbol{K} \in \mathbb{R}^{n \times n}$ be the kernel/Gram matrix and its eigendecomposition $\boldsymbol{K} = \sum_{i=1}^{n} \lambda_i \boldsymbol{v}_i \boldsymbol{v}_i^T$ with $\lambda_1 \geq \cdots \geq \lambda_n \geq 0$.

The *kernel principal components* are given by $\boldsymbol{\alpha}^{(1)}, \ldots, \boldsymbol{\alpha}^{(k)} \in \mathbb{R}^n$ where

$$\boldsymbol{\alpha}^{(i)} = \frac{1}{\sqrt{\lambda_i}} \boldsymbol{v}_i$$

Given this, a new point $\boldsymbol{x}$ is projected as $\boldsymbol{z} \in \mathbb{R}^k$ with

$$z_i = \sum_{j=1}^{n} \alpha_j^{(i)} k(\boldsymbol{x}, \boldsymbol{x}_j)$$

The kernel has to be centred where $\boldsymbol{E} = \frac{1}{n}[1, \ldots, 1][1, \ldots, 1]^T$ as

$$\boldsymbol{K}' = \boldsymbol{K} - \boldsymbol{K}\boldsymbol{E} - \boldsymbol{E}\boldsymbol{K} + \boldsymbol{E}\boldsymbol{K}\boldsymbol{E}$$

Kernel-k-means refers to applying k-means onto kernel principal components.

Derivation for $k = 1$

For PCA, $\boldsymbol{w}^*$ is the leading eigenvector λ of $\boldsymbol{X}^T\boldsymbol{X}$. Thus,

$$\boldsymbol{X}^T \underbrace{\boldsymbol{X}\boldsymbol{w}}_{\boldsymbol{\beta}} = \lambda \boldsymbol{w} \Rightarrow \boldsymbol{w} = \frac{1}{\lambda} \boldsymbol{X}^T \boldsymbol{\beta} = \frac{1}{\lambda} \sum_{i=1}^{n} \beta_i \boldsymbol{x}_i$$

Therefore, with $\alpha_i = \frac{\beta_i}{\lambda}$, the general Ansatz holds:

$$\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i$$

Now, both the objective and constraint have to be expressed in terms of α and inner products.

Let $\boldsymbol{K}$ be the kernel/Gram matrix ($\boldsymbol{K}_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$) and $\boldsymbol{K}_i$ its i-th column.

For the objective, we have

$$\begin{aligned}
\boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} &= \sum_{i=1}^{n} (\boldsymbol{w}^T \boldsymbol{x}_i)^2 \\
&= \sum_{i=1}^{n} \left((\sum_{j=1}^{n} \alpha_j \boldsymbol{x}_j)^T \boldsymbol{x}_i \right)^2 \\
&= \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \alpha_j \boldsymbol{x}_j^T \boldsymbol{x}_i \right)^2 \\
&= \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_j, \boldsymbol{x}_i) \right)^2 \\
&= \sum_{i=1}^{n} \left(\boldsymbol{\alpha}^T \boldsymbol{K}_i \right)^2 \\
&= \boldsymbol{\alpha}^T \boldsymbol{K}^T \boldsymbol{K} \boldsymbol{\alpha}
\end{aligned}$$

For the constraints, we have

$$\begin{aligned}
\|\boldsymbol{w}\|_2 &= \boldsymbol{w}^T \boldsymbol{w} \\
&= \left(\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i \right)^T \left(\sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i \right) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j \boldsymbol{x}_i^T \boldsymbol{x}_j \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \\
&= \sum_{i=1}^{n} \alpha_i \boldsymbol{K}_i \boldsymbol{\alpha} \\
&= \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha} \\
&\stackrel{!}{=} 1
\end{aligned}$$

Thus, the goal is

$$\boldsymbol{\alpha}^* = \arg\max_{\boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha}} \boldsymbol{\alpha}^T \boldsymbol{K}^T \boldsymbol{K} \boldsymbol{\alpha}$$

The optimal solution is then obtained by

$$\boldsymbol{\alpha}^* = \frac{1}{\sqrt{\lambda_1}} \boldsymbol{v}_1$$

Scaling is required because

$$\begin{aligned}
\boldsymbol{\alpha}^{*T}\boldsymbol{K}\boldsymbol{\alpha}^{*} &= \frac{1}{\sqrt{\lambda_1}}\boldsymbol{v}_1^T\boldsymbol{K}\frac{1}{\sqrt{\lambda_1}}\boldsymbol{v}_1 \\
&= \frac{1}{\lambda_1}\boldsymbol{v}_1^T\boldsymbol{K}\boldsymbol{v}_1 \\
&= \frac{1}{\lambda_1}\boldsymbol{v}_1^T(\lambda_1\boldsymbol{v}_1) \\
&= \boldsymbol{v}_1^T\boldsymbol{v}_1 \\
&= 1
\end{aligned}$$

Maximum Variance Perspective

Instead of minimising the reconstruction error, PCA can also be interpreted as maximising the variance of the projected data.

The *projected variance* is

$$Q_R(\Sigma, \boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}\|\boldsymbol{w}^T\boldsymbol{x}_i - \boldsymbol{w}^T\bar{\boldsymbol{x}}\|_2^2 = \boldsymbol{w}^T\Sigma\boldsymbol{w}$$

where $\bar{\boldsymbol{x}} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}_i$ is the data mean and $\Sigma = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T$ the data covariance. If $\bar{\boldsymbol{x}} = \boldsymbol{0}$ then $\Sigma = \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X}$.

Now, if we want to maximise the projected variance,

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} \boldsymbol{w}^T\Sigma\boldsymbol{w} + \lambda(1 - \boldsymbol{w}^T\boldsymbol{w})$$

by setting the derivative to zero we get

$$\begin{aligned}
\frac{\partial}{\partial\boldsymbol{w}}\boldsymbol{w}^T\Sigma\boldsymbol{w} + \lambda(1 - \boldsymbol{w}^T\boldsymbol{w}) = 2\Sigma\boldsymbol{w} - 2\lambda\boldsymbol{w} = 0 \\
\Leftrightarrow \Sigma\boldsymbol{w} = \lambda\boldsymbol{w} \\
\Leftrightarrow \boldsymbol{w}^T\Sigma\boldsymbol{w} = \lambda
\end{aligned}$$

Thus, $\boldsymbol{w}$ is an eigenvector of Σ and $\boldsymbol{w}^T\Sigma\boldsymbol{w} = \lambda$, i.e. the variance is given by the eigenvalue of Σ.

The PCA solution selects the eigenvector of Σ with the largest eigenvalue λ. Thus, the PCA solution maximises the projected variance. The derivation in the k-dimensional case is equivalent, except that the trace of the arg max is taken.

It holds that variance plus error squared is constant. Thus, maximising the variance corresponds to minimising the error.

2.8 Autoencoders

The key idea is to learn the identity function $\boldsymbol{x} \approx f(\boldsymbol{x};\theta)$. We can also think of it as learning a compression and decompression function, both in one. $f(\boldsymbol{x};\theta) = f_2(f_1(\boldsymbol{x};\theta_1);\theta_2)$ where f_2 is the decoder, f_1 is the encoder. Both functions are learned jointly.

Neural network autoencoders are ANNs where there is one output unit for each of the d inputs and the number of hidden units k is typically smaller than the number of inputs.

An example objective function is the sum of squared errors:

$$\arg\min_{\boldsymbol{W}} \sum_{i=1}^{n} \|\boldsymbol{x}_i - f(\boldsymbol{x}_i;\boldsymbol{W})\|_2^2$$

The objective can be optimised using SGD. However, initialisation is important and challenging.

If all non-linearities of an autoencoder are the identity function, then fitting an autoencoder is equivalent to PCA.

2.9 Logistic Regression

The idea of logistic regression is to use a (generalised) linear model for the class probability. The hyperplane defines the decision boundary, all points on the positive side correspond to the positive class, on the other side to the negative class.

Therefore, $Y \in \{+1,-1\}$ and $P(Y = +1 \mid \boldsymbol{X} = \boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$ where σ is the logistic sigmoid function

$$\sigma(\boldsymbol{w}^T\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

This is called the *link function*.

TODO: Move this somewhere else? Maybe to prob. modelling? This relates to *generalised linear models*. A generalised linear model is a model with a linear decision boundary $\boldsymbol{w}^T\boldsymbol{x} + w_0$. A link function converts the mean of a distribution to a linear predictor, and the mean function converts the linear predictor into a mean. TODO: Isn't sigmoid actually the mean function, and logit the link function? TODO: Link function for Poisson.

From $P(Y = -1 \mid \boldsymbol{X} = \boldsymbol{x}) = 1 - P(Y = +1 \mid \boldsymbol{X} = \boldsymbol{x})$ we get

$$P(Y = y \mid \boldsymbol{X} = \boldsymbol{x}) = \sigma(y\boldsymbol{w}^T\boldsymbol{x}) = \frac{1}{1+\exp(-y\boldsymbol{w}^T\boldsymbol{x})}$$

The resulting model assumption is

$$P(y \mid \boldsymbol{x}, \boldsymbol{w}) = Ber(y;\sigma(\boldsymbol{w}^T\boldsymbol{x}))$$

i.e. linear with Bernoulli noise.

Maximum Likelihood Estimate

If $\boldsymbol{w}$ and the $\boldsymbol{x}_i$ are independent, it holds that

$$P(y_i \mid \boldsymbol{x}_i, \boldsymbol{w}) = \frac{P(y_i, \boldsymbol{x}_i \mid \boldsymbol{w})}{P(\boldsymbol{x}_i \mid \boldsymbol{w})} = \frac{P(y_i, \boldsymbol{x}_i \mid \boldsymbol{w})}{P(\boldsymbol{x}_i)} = \text{const} \cdot P(y_i, \boldsymbol{x}_i \mid \boldsymbol{w})$$

From that, and by assuming the samples are i.i.d., we can again minimise the conditional negative log likelihood

$$\arg\max_{\boldsymbol{w}} P(D \mid \boldsymbol{w}) = \arg\max_{\boldsymbol{w}} P(y_{1:n} \mid \boldsymbol{x}_{1:n}, \boldsymbol{w}) = \arg\min_{\boldsymbol{w}} -\sum_{i=1}^{n} \log P(y_i \mid \boldsymbol{x}_i, \boldsymbol{w})$$

The negative log likelihood of a single sample is

$$-\log P(y_i \mid \boldsymbol{x}_i, \boldsymbol{w}) = -\log \frac{1}{1 + \exp(-y\boldsymbol{w}^T\boldsymbol{x})} = \log(1 + \exp(-y\boldsymbol{w}^T\boldsymbol{x}))$$

This results in the *logistic loss* function:

$$\ell_{\text{logistic}}(\boldsymbol{w}; \boldsymbol{x}, y) = \log(1 + \exp(-y\boldsymbol{w}^T\boldsymbol{x}))$$

The empirical risk is thus given as

$$\hat{R}(\boldsymbol{w}) = \sum_{i=1}^{n} \log(1 + \exp(-y_i\boldsymbol{w}^T\boldsymbol{x}_i))$$

The logistic loss is convex and can therefore be optimised using SGD and similar techniques. It is for very large and very small z, it is approximately linear and corresponds to the perceptron loss. However, its derivative is never exactly zero, and correct classifications close to the decision boundary ($z \approx 0$) still exhibit a significant loss.

The gradient of the logistic loss is

$$\nabla_{\boldsymbol{w}} \ell_{\text{logistic}} = \underbrace{\frac{\exp(-y\boldsymbol{w}^T\boldsymbol{x})}{1 + \exp(-y\boldsymbol{w}^T\boldsymbol{x})}}_{1 - P(Y=y|\boldsymbol{x})} \cdot (-y\boldsymbol{x}) = \underbrace{\frac{1}{1 + \exp(y\boldsymbol{w}^T\boldsymbol{x})}}_{P(Y \neq y|\boldsymbol{x})} \cdot (-y\boldsymbol{x})$$

The SGD update step is thus

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta_t \cdot y \cdot \boldsymbol{x} \cdot P(Y = -y \mid \boldsymbol{x}, \boldsymbol{w})$$

Maximum a Posteriori Estimate / Regularisation

Similar to linear regression, we can perform MAP estimation / use a regulariser.

Using a Gaussian prior on the weights, corresponding to L_2 regularisation, results in the objective

$$\arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \log\left(1 + \exp\left(-y_i \boldsymbol{w}^T \boldsymbol{x}_i\right)\right) + \lambda \|\boldsymbol{w}\|_2^2$$

Using a Laplace prior on the weights, corresponding to L_1 regularisation, results in the objective

$$\arg\min_{\boldsymbol{w}} \sum_{i=1}^{n} \log\left(1 + \exp\left(-y_i \boldsymbol{w}^T \boldsymbol{x}_i\right)\right) + \lambda \|\boldsymbol{w}\|_1$$

The weight-update step for L_2 regularised logistic regression is

$$\boldsymbol{w} \leftarrow \boldsymbol{w}(1 - 2\lambda\eta_t) + \eta_t \cdot y \cdot \boldsymbol{x} \cdot \hat{P}(Y = -y \mid \boldsymbol{w}, \boldsymbol{x})$$

After finding the optimal $\hat{\boldsymbol{w}}$, prediction is done using the conditional distribution $P(y \mid \boldsymbol{x}, \hat{\boldsymbol{w}})$, which corresponds to predicting

$$\hat{y} = sign(\hat{\boldsymbol{w}}^T \boldsymbol{x})$$

Kernelised Logistic Regression

Logistic regression can also be kernelised.

Let $\boldsymbol{K}$ be the Gram matrix and $\boldsymbol{K}_i$ its i-th column.

The optimisation objective is

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \log\Big(1 + \exp\left(-y_i \boldsymbol{\alpha}^T \boldsymbol{K}_i\right)\Big) + \lambda \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha}$$

Classification is then done using the conditional distribution

$$\hat{P}(y \mid \boldsymbol{x}, \hat{\boldsymbol{\alpha}}) = \frac{1}{1 + \exp\left(-y \underbrace{\sum_{j=1}^{n} \hat{\alpha}_j k(\boldsymbol{x}_j, \boldsymbol{x})}_{=\boldsymbol{w}^T \boldsymbol{x}}\right)}$$

and we can again predict the more likely class as

$$\hat{y} = sign\left(\sum_{j=1}^{n} \hat{\alpha}_j k(\boldsymbol{x}_j, \boldsymbol{x})\right)$$

Multi-Class Logistic Regression

Logistic regression can be extended to the multi-class setting. The assumed distribution is then the categorical distribution instead of Bernoulli.

Let c be the number of classes. We maintain one weight vector $\boldsymbol{w}_i$ for each class $i \in \{1, \dots, c\}$. The respective class probability is modelled proportional to the total class probabilities. The function used is called the *Softmax* function and defined as

$$P(Y = i \mid \boldsymbol{x}, \boldsymbol{w}_1, \dots, \boldsymbol{w}_c) = \frac{\exp(\boldsymbol{w}_i^T \boldsymbol{x})}{\sum_{j=1}^{c} \exp(\boldsymbol{w}_j^T \boldsymbol{x})} = \hat{p}_i$$

with $\sum_{i=1}^{c} \hat{p}_i = 1$. This is a strict generalisation of a binary model.

The solution is not unique. If some scalar τ is added to each prediction it does not change:

$$\frac{\exp(\boldsymbol{w}_i^T \boldsymbol{x} + \tau)}{\sum_{j=1}^{c} \exp(\boldsymbol{w}_j^T \boldsymbol{x} + \tau)} = \frac{e^{\tau} \cdot \exp(\boldsymbol{w}_i^T \boldsymbol{x})}{e^{\tau} \cdot \sum_{j=1}^{c} \exp(\boldsymbol{w}_j^T \boldsymbol{x})} = \frac{\exp(\boldsymbol{w}_i^T \boldsymbol{x})}{\sum_{j=1}^{c} \exp(\boldsymbol{w}_j^T \boldsymbol{x})}$$

Uniqueness can be enforced by setting one weight vector to zero (e.g. $\boldsymbol{w}_c$) which corresponds to setting $\tau = -\boldsymbol{w}_c^T \boldsymbol{x}$ and $\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i - \boldsymbol{w}_c^T \boldsymbol{x}$.

This results in the cross-entropy loss

$$\ell_{CE}(y; \boldsymbol{x}, \boldsymbol{w}_1, \dots, \boldsymbol{w}_c) = -\log P(Y = y \mid \boldsymbol{x}, \boldsymbol{w}_1, \dots, \boldsymbol{w}_c)$$

TODO: Gradient of cross-entropy loss, Homework 6.

TODO: Softmax in ANN

TODO: Cross entropy in ANN

If we add a τ to each argument, the e^{τ} is going to cancel out, thus we keep the original distribution. W.l.o.g. we can pick $\tau = -w_c^T x$ which corresponds to setting one weight vector (e.g. the last one) to 0. TODO: Slide 29, multi-class logistic regression summary. Setting w_c to 0 forces uniqueness. This is called the cross-entropy loss.

Comparison to SVM

SVMs result in sparse solutions. This means that most of the α_i are zero. This does not hold for logistic regression as most of its α_i are non-zero.

SVMs also sometimes result in higher classification accuracy. However, logistic regression provides class probabilities which is difficult in SVMs.

2.10 Naive Bayes Model

Naive Bayes is a generative model.

The class labels $y \in \mathcal{Y} = \{1, \ldots, c\}$ are modelled as being generated from a categorical variable:

$$P(Y = y) = p_y, p_y \geq 0, \sum_{y=1}^{c} p_y = 1$$

The features are modelled conditionally independent given Y:

$$P(X_1, \ldots, X_d \mid Y) = \prod_{i=1}^{d} P(X_i \mid Y)$$

i.e. given a class label, each feature is generated independently of all other features.

This is a very strong modelling assumption which in practice often does not hold (but still works well).

The feature distributions $P(X_i \mid Y)$ still need to be defined. They lead to different Naive Bayes classifiers.

Decision Rules

Let $\hat{P}(y)$ and $\hat{P}(\boldsymbol{x} \mid y)$ be the estimated distributions.

To predict the label y for a new example $\boldsymbol{x}$, Bayes' theorem can be used

$$P(y \mid \boldsymbol{x}) = \frac{1}{Z} P(y) P(\boldsymbol{x} \mid y)$$

with

$$Z = \sum_{y=1}^{c} P(y) P(\boldsymbol{x} \mid y)$$

The label $\hat{y}$ can then be predicted using Bayesian decision theory:

$$\hat{y} = \arg\max_y \hat{P}(y \mid \boldsymbol{x}) = \arg\max_y \hat{P}(y) \prod_{i=1}^{d} \hat{P}(x_i \mid y)$$

Gaussian Naive Bayes Classifier

In a GNB, the features are modelled as *conditionally independent Gaussians*:

$$P(x_i \mid y) = \mathcal{N}(x_i \mid \mu_{y,i}, \sigma^2_{y,i})$$

Note that μ and σ^2 depend on the class and feature, i.e. there are $c \cdot d$ versions of each parameter.

MLE for Two Classes

Assume there are two classes, i.e. $\mathcal{Y} = \{+1, -1\}$. Let D be the data set.

We have $P(Y = +1) = p$ and $P(Y = -1) = (1 - p)$. The MLE for p is

$$\arg\max_p P(D \mid p) = \frac{n_+}{n_+ + n_-}$$

where n_+ is the number of positive samples, n_- the number of negative samples.

The conditional distribution for the data is $P(x_i \mid y) = \mathcal{N}(x_i; \mu_{y,i}, \sigma^2_{y,i})$. Then we have to do the usual MLE for Gaussian distributions. TODO: Result.

Summary

Let $D = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ be the data set with $y_i \in \{1, \ldots, c\}$.

The MLE for the class prior is

$$\hat{p}_y = \hat{P}(Y = y) = \frac{Count(Y = y)}{n}$$

The MLE for the feature distribution $\hat{P}(x_i \mid y) = \mathcal{N}(x_i; \hat{\mu}_{y,i}, \sigma^2_{y,i})$ is

$$\hat{\mu}_{y,i} = \frac{1}{Count(Y = y)} \sum_{j:y_j = y} x_{j,i}$$

$$\hat{\sigma}^2_{y,i} = \frac{1}{Count(Y = y)} \sum_{j:y_j = y} (x_{j,i} - \hat{\mu}_{y,i})^2$$

i.e. the empirical mean and variance for all examples with class y.

Prediction given $\boldsymbol{x}$ is

$$\hat{y} = \arg\max_y \hat{P}(y \mid \boldsymbol{x}) = \arg\max_y \hat{P}(y) \prod_{i=1}^{d} \hat{P}(x_i \mid y)$$

Binary Classification

In the binary case ($c = 2$), the above prediction is equivalent to

$$\hat{y} = sign\left(\log \overbrace{\frac{P(Y = +1 \mid \boldsymbol{x})}{P(Y = -1 \mid \boldsymbol{x})}}^{p} \right)$$

The *discriminant function* is defined as

$$f(\boldsymbol{x}) = \log \frac{P(Y = +1 \mid \boldsymbol{x})}{P(Y = -1 \mid \boldsymbol{x})}$$

Now two further assumptions are introduced. Assumption 1 is that $P(Y = +1) = P(Y = -1) = 0.5$, i.e. both classes have equal probability. Assumption 2 is that $\forall y \in \mathcal{Y} : \sigma_{y,i}^2 = \sigma_i^2$, i.e. the variance and the class are independent.

TODO: Derivation? Then,

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0$$

i.e. GNB produces a linear classifier with

$$w_0 = \log \frac{\hat{p}_+}{1 - \hat{p}_+} + \sum_{i=1}^{d} \frac{\hat{\mu}_{-,i}^2 - \hat{\mu}_{+,i}^2}{2\hat{\sigma}_i^2}$$
$$w_i = \frac{\hat{\mu}_{+,i} - \hat{\mu}_{-,i}}{\hat{\sigma}_i^2}$$

If the GNB and other assumptions hold (independent features, conditioned on class labels, coming from Gaussian, assumptions 1 and 2) then the resulting decision boundary is equivalent to that from logistic regression: Let $p(x) := P(Y = +1 \mid \boldsymbol{x})$.

$$\Rightarrow f(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x})}{1 - p(\boldsymbol{x})}$$
$$\Rightarrow \exp(f(\boldsymbol{x})) = \frac{p(\boldsymbol{x})}{1 - p(\boldsymbol{x})}$$
$$\Rightarrow p(\boldsymbol{x}) = \frac{\exp(f(\boldsymbol{x}))}{1 + \exp(f(\boldsymbol{x}))} = \frac{1}{1 + \exp(-f(\boldsymbol{x}))} = \sigma(f(\boldsymbol{x}))$$

Poisson Naive Bayes Classifier

Let $\boldsymbol{x}_i \in \mathbb{N}^d, y \in \{0, 1\}$ be the i.i.d. observations. The Poisson Naive Bayes Classifier assumes $Y \sim Bernoulli(p)$ and $P(\boldsymbol{x}_i \mid y_i) = \prod_{j=1}^{d} P(x_{i,j} \mid y_i)$ with $P(x_{i,j} \mid y_i) = Poiss(x_{i,j}; \lambda_{y_i,j})$.

Let $n_0 := |\{i : y_i = 0\}|$ and $n_1 := |\{i : y_i = 1\}|$.

The MLE for the labels is given as $p_0 = \frac{n_0}{n}$ and $p_1 = \frac{n_1}{n} = 1 - p_0$.

The MLE for the conditional distribution with $y \in \{0, 1\}$ is given as

$$\lambda_{y,j} = \frac{\sum_{i:y_i=y}^{n} x_{i,j}}{n_y}$$

i.e. the mean of all samples of class y.

It can be shown that the Poisson Naive Bayes Classifier produces a linear decision boundary.

Categorical Naive Bayes

Features may of course also take discrete values. Thus, it might not make sense to model them as Gaussians but e.g. as Bernoulli, Categorical or Multinomial.

In the categorical case, the class labels are still assumed to be generated from a categorical variable, i.e.

$$P(Y = y) = p_y$$

with $y \in \mathcal{Y} = \{1, \ldots, c\}$.

The features are modelled as (conditionally) independent categorical random variables:

$$P(X_i = c \mid Y = y) = \theta_{c|y}^{(i)}$$

where $\theta_{c|y}^{(i)}$ is the probability that the i-th feature takes value c for class y.

Estimation

The MLE for the class distribution, $\hat{P}(Y = y) = \hat{p}_y$, is again given as

$$\hat{p}_y = \frac{Count(Y = y)}{n}$$

The MLE for the distribution of feature i, $\hat{P}(X_i = c \mid Y = y) = \theta_{c|y}^{(i)}$, is given as

$$\theta_{c|y}^{(i)} = \frac{Count(X_i = c, Y = y)}{Count(Y = y)}$$

Prediction

Prediction for a new point $\boldsymbol{x}$ is done as usual as

$$\hat{y} = \arg\max_y \hat{P}(y \mid \boldsymbol{x}) = \arg\max_y \hat{P}(y) \prod_{i=1}^{d} \hat{P}(x_i \mid y)$$

Dropping the Naive Assumption

Removing the naive assumption for categorical features is much harder. In general, it requires exponentially many parameters (in d). We have to specify $P(X_1 = x_1, \ldots, X_d = x_d | Y = y)$ for each $x_1, \ldots, x_d$, exponentially many numbers. This is computationally infeasible and very easy to overfit.

However, there are models which replace the naive assumption with some weaker ones.

Mixing Distributions

Due to the naive assumption, each feature is modelled to be from a different conditional distribution.

Those distributions do not have to be of the same type, and even discrete and continuous distributions can be mixed. Fitting (MLE) and prediction stays the same.

Regularisation

MLE is prone to overfitting as it maximises the likelihood for exactly the given realisation. MLE has nice asymptotic properties but for finite data, it is prone to overfitting. There are two ways to combat overfitting in generative modelling.

One way is to restrict the model class, e.g. assumptions on the covariance structure. This results in fewer parameters.

Parameter Priors

In generative modelling, it is very natural to think about priors for regularisation. This poses a second form of regularisation.

Priors are always applied to the distribution of $P(Y)$. We want to put a prior distribution $P(\theta)$ and then compute the posterior distribution $P(\theta \mid y_1, \ldots, y_n)$.

TODO: Not 100 percent sure this makes sense.

Example: Beta Prior

Assume we have $c = 2$ and thus $P(Y = +1) = \theta$.

The *Beta-Distribution* is defined as

$$Beta(\theta; \alpha_+, \alpha_-) = \frac{1}{B(\alpha_+, \alpha_-)} \theta^{\alpha_+ - 1} (1 - \theta)^{\alpha_- - 1}$$

$B(\alpha_+, \alpha_-)$ is a normalisation constant to ensure the integral becomes 1. If α_+ is smaller than α_-, the distribution is more peaked and the peak is more to the right. If both are less than 1, the resulting distribution is bimodal. If α_+ and α_- go to infinity, the distribution is going to converge to a point mass on $\frac{\alpha_+}{\alpha_-}$.

The interesting property of the Beta-distribution is that its posterior is also a Beta-distribution, i.e. it is *conjugate* with the Binomial likelihood:

$$P(\theta \mid y_1, \ldots, y_n; \alpha_+, \alpha_-) = P(\theta \mid n_+, n_-, \alpha_+, \alpha_-) = Beta(\theta; \alpha_+ + n_+, \alpha_- + n_-)$$

The MAP estimate is then given as

$$\hat{\theta} = \arg\max_\theta P(\theta \mid y_1, \ldots, y_n; \alpha_+, \alpha_-) = \frac{\alpha_+ + n_+ - 1}{\alpha_+ + n_+ + \alpha_- + n_- - 2}$$

Thus the α_+, α_- act as pseudo-counts.

Issues

GNB models assume that features are generated independently given a class label. However, if there is (conditional) correlation between features, given class labels, then this assumption is violated!

For example, assume $P(Y = +1) = P(Y = -1) = 0.5$ and further features $x_2, \ldots, x_d = x_1$, i.e. are duplicates. A GNB classifier which only uses x_1 has

$$f_1(x) = \log \frac{P(Y = +1 \mid x_1 = x)}{P(Y = -1 \mid x_1 = x)}$$

where a classifier which uses all d features has

$$f_2(\boldsymbol{x}) = \log \frac{\prod_{i=1}^{d} P(X_i = x_i \mid Y = +1)}{\prod_{i=1}^{d} P(X_i = x_i \mid Y = -1)} = d \cdot f_1(x_1)$$

Thus the prediction gets blown-up, i.e. become overconfident (very close to 0 or 1), even though it should theoretically not change.

This is fine if only the most likely class matters, but bad if probabilities should be used for decision making. Logistic regression works better in that case because we do not look at $P(\boldsymbol{x})$ and thus do not have to explain correlation. Generally, logistic regression is more robust and better in terms of classification performance, but cannot provide information about $P(\boldsymbol{x})$.

A way to overcome this is to drop the naive assumption, i.e. use full Gaussian Bayes Classifiers.

2.11 Gaussian Bayes Classifiers

Gaussian Bayes Classifiers (GBC) are generalisations of GNB classifiers.

The class labels are modelled as generated from a categorical variable:

$$P(Y = y) = p_y$$

with $y \in \mathcal{Y} = \{1, \ldots, c\}$.

The features $\boldsymbol{x}$ are modelled as generated by multivariate Gaussians:

$$P(\boldsymbol{x} \mid y) = \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_y, \Sigma_y)$$

with $\boldsymbol{\mu}_y$ being the mean and Σ_y the covariance matrix.

Estimation

Let $D = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ be the data set.

The MLE for the class label distribution $\hat{P}(Y = y) = \hat{p}_y$ is given as

$$\hat{p}_y = \frac{Count(Y = y)}{n}$$

The MLE for the feature distribution $\hat{P}(\boldsymbol{x} \mid y) = \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_y, \Sigma_y)$ is given as

$$\hat{\boldsymbol{\mu}}_y = \frac{1}{Count(Y = y)} \sum_{i:y_i=y} \boldsymbol{x}_i$$

$$\hat{\Sigma}_y = \frac{1}{Count(Y = y)} \sum_{i:y_i=y} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_y)(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_y)^T$$

i.e. the empirical (class) mean and the empirical (class) covariance.

Note that now, for the (co)variance, d^2 instead of d parameters need to be estimated.

Discriminant Function

Given $P(Y = +1) = p$ and $P(\boldsymbol{x} \mid y) = \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_y, \Sigma_y)$.

We want

$$f(\boldsymbol{x}) = \frac{P(Y = +1 \mid \boldsymbol{x})}{P(Y = -1 \mid \boldsymbol{x})}$$

For GBC, this is given by

$$f(\boldsymbol{x}) = \log \frac{p}{1-p} + \frac{1}{2}\left[\log \frac{|\hat{\Sigma}_-|}{|\hat{\Sigma}_+|} + \left((\boldsymbol{x} - \hat{\boldsymbol{\mu}}_-)^T \hat{\Sigma}_-^{-1} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_-)\right) - \left((\boldsymbol{x} - \hat{\boldsymbol{\mu}}_+)^T \hat{\Sigma}_+^{-1} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_+)\right)\right]$$

with $|\cdot|$ denoting the (matrix) discriminant.

Variations

If Σ is a diagonal matrix, this corresponds to a *Gaussian Naive Bayes* classifier. The diagonal elements are the $\sigma_{y,i}^2$.

For $c = 2$, $P(Y = +1) = P(Y = -1) = 0.5$, Σ a diagonal matrix (naive), and equal covariances (not depending on class) $\hat{\Sigma}_- = \hat{\Sigma}_+ = \hat{\Sigma}$, this is of the same form as *Logistic Regression*. If the modelling assumptions are met, the predictions are the same.

For $c = 2$, $P(Y = +1) = P(Y = -1) = 0.5$ and equal covariances (not depending on class) $\hat{\Sigma}_- = \hat{\Sigma}_+ = \hat{\Sigma}$, this corresponds to *Fisher's Linear Discriminant Analysis*.

TODO: More here?

TODO: Big Picture, slides 22, p. 22

Comparison to GNB Classifiers

The conditional independence assumption of GNB models can lead to overconfidence where general GBC capture correlations among features and thus avoids overconfidence. However, GNB predictions can still be useful.

Also, GNB models are cheaper to fit: Fitting a naive model has parameters in $\mathcal{O}(c \cdot d)$ where a full GBC has parameters in $\mathcal{O}(c \cdot d^2)$.

2.12 Fisher's Linear Discriminant Analysis

Suppose we use a GBC. Further assume

- only two classes, $c = 2$.
- equal class probabilities, $p = P(Y = +1) = P(Y = -1) = 0.5$.
- equal class covariances, $\hat{\Sigma}_- = \hat{\Sigma}_+ = \hat{\Sigma}$.

Then, the discriminant function greatly simplifies to

$$f(\boldsymbol{x}) = \boldsymbol{x}^T \hat{\Sigma}^{-1}(\hat{\boldsymbol{\mu}}_+ - \hat{\boldsymbol{\mu}}_-) + \frac{1}{2}\left(\hat{\boldsymbol{\mu}}_-^T \hat{\Sigma}^{-1} \hat{\boldsymbol{\mu}}_- - \hat{\boldsymbol{\mu}}_+^T \hat{\Sigma}^{-1} \hat{\boldsymbol{\mu}}_+\right)$$

TODO: Derivation?

Under these assumptions, we then predict

$$\hat{y} = sign(f(\boldsymbol{x})) = sign(\boldsymbol{w}^T \boldsymbol{x} + w_0)$$

with

$$\begin{aligned} \boldsymbol{w} &= \hat{\Sigma}^{-1}(\hat{\boldsymbol{\mu}}_+ - \hat{\boldsymbol{\mu}}_-) \\ w_0 &= \frac{1}{2}\left(\hat{\boldsymbol{\mu}}_-^T \hat{\Sigma}^{-1} \hat{\boldsymbol{\mu}}_- - \hat{\boldsymbol{\mu}}_+^T \hat{\Sigma}^{-1} \hat{\boldsymbol{\mu}}_+\right) \end{aligned}$$

Thus the resulting classifier is linear.

This linear classifier is called *Fisher's Linear Discriminant Analysis* (LDA).

Comparison

Comparison to Logistic Regression

Fisher's LDA uses the discriminant function

$$f(\boldsymbol{x}) = \log \frac{P(Y = +1) \mid \boldsymbol{x})}{P(Y = -1 \mid \boldsymbol{x})}$$

which is derived $f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0$. It can be rearranged to

$$P(Y = +1 \mid \boldsymbol{x}) = \frac{1}{1 + \exp(-f(\boldsymbol{x})}$$

and by using the derived form

$$P(Y = +1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + w_0)$$

This is the same form as logistic regression, assuming homogeneous coordinates are used.

If the model assumptions are met, LDA will make the same prediction as logistic regression.

Fisher's LDA is a generative model, i.e. models $P(\boldsymbol{x}, y)$. Logistic regression is a discriminative model, i.e. models $P(y \mid \boldsymbol{x})$.

LDA can thus be used to detect outliers with $P(\boldsymbol{x}) < \tau$. It also assumes normality of $\boldsymbol{X}$, which is a very strong assumption. Logistic regression makes no assumptions on $\boldsymbol{X}$ but can also not be used to detect outliers. Generally, LDA is not very robust against violations of its assumptions where logistic regression, making less assumptions, is more robust.

Comparison to PCA

LDA can be viewed as a projection into a 1-dimensional subspace which maximises the ratio of between-class and within-class variance.

In contrast, PCA maximises the variance of the resulting 1-dimensional projection; it does not know about classes.

Quadratic Discriminant Analysis

There is no reason to assume that the variances are equal for different classes.

In the resulting, more general case, the discriminant is

$$f(\boldsymbol{x}) = \log \frac{p}{1-p} + \frac{1}{2}\left[\log \frac{|\hat{\Sigma}_-|}{|\hat{\Sigma}_+|} + \left((\boldsymbol{x} - \hat{\boldsymbol{\mu}}_-)^T \hat{\Sigma}_-^{-1} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_-)\right) - \left((\boldsymbol{x} - \hat{\boldsymbol{\mu}}_+)^T \hat{\Sigma}_+^{-1} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_+)\right)\right]$$

with $|\cdot|$ denoting the (matrix) discriminant. Note that this is precisely the discriminant of a GBC. It has a quadratic form.

The prediction is done as

$$\hat{y} = sign(f(\boldsymbol{x}))$$

2.13 Gaussian Mixture Models

Gaussian Mixture Models (GMMs) are generative models. Because generative modelling describes tries to describe all the data, it can naturally be used in the case of missing data. Unsupervised learning is an extreme case of missing data where all the labels are absent.

In GMMs, we assume the data was generated by a mixture of Gaussians. A *Gaussian mixture* is a convex combination of k Gaussian distributions:

$$P(\boldsymbol{x} \mid \theta) = P(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma, \boldsymbol{w}) = \sum_{i=1}^{k} w_i \cdot \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_i, \Sigma_i)$$

with $w_i \geq 0$ and $\sum_{i=1}^{k} w_i = 1$, i.e. defining a categorical distribution. The w_i are called the *mixture weights*.

Mixture models in general are convex combinations of k simple (base) distributions. Mixture models are more expressive than the base distributions as they allow for multimodal data representation. GMMs are thus mixture models where all base distributions are Gaussians.

The negative log likelihood, assuming the samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are i.i.d., is

$$(\boldsymbol{\mu}^*, \Sigma^*, \boldsymbol{w}^*) = \arg\min - \sum_{i=1}^{n} \log \sum_{j=1}^{k} w_j \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_j, \Sigma_j)$$

This is a non-convex objective. Because it contains the logarithm of a sum, no closed-form solution can be derived. Furthermore, SGD is challenging to apply because the covariance matrices must remain symmetric positive definite.

Let z be a cluster index and $\boldsymbol{x}$ a feature. The joint distribution $P(z, \boldsymbol{x}) = w_z \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_u, \Sigma_z)$ is identical to the generative model used by a GBC. However, in contrast to a GBC, the cluster index z is a latent variable, i.e. unobserved. Thus, fitting a GMM is equivalent to training a GBC without labels and a natural approach would be to try to infer the labels.

Hard-EM

Let $\theta^{(t)} = (\boldsymbol{w}^{(t)}, \boldsymbol{\mu}^{(t)}, \Sigma^{(t)})$ denote the parameters at step t.

First, (carefully) initialise the parameters $\theta^{(0)}$.

Then, for $t = 1, 2, \ldots$

E-step : Predict the most likely class for each data point.

$$z_i^{(t)} = \arg\max_z P(z \mid \boldsymbol{x}_i, \theta^{(t-1)}) = \arg\max_z \underbrace{P(z \mid \theta^{(t-1)})}_{w_z^{(t-1)}} \cdot \underbrace{P(\boldsymbol{x}_i \mid z, \theta^{(t-1)})}_{\mathcal{N}(\boldsymbol{x}_i; \boldsymbol{\mu}_i^{(t-1)}, \Sigma_i^{(t-1)})}$$

This gives the "completed" data set $D^{(t)}$.

M-step : Compute MLE for the GBC.

$$\theta^{(t)} = \arg\max_{\theta} P(D^{(t)} \mid \theta)$$

The intuitive interpretation is as follows: In the E-step, the (missing) class labels are inferred using the GBC of the previous round. This gives a completed data set for each time step. Then, in the M-step, MLE for a GBC is performed, using the previously inferred class labels.

Hard-EM is conceptually very similar to the k-means algorithm.

Issues

The Hard-EM algorithm has various issues. In particular, each point is forced to declare allegiance to some cluster, even though the model is uncertain. Intuitively, this tries to extract too much information from a single point.

A GMM with k mixture components can always represent mixtures with $< k$ components. By setting some mixture weights to 0, or putting multiple distributions directly on top of each other, all but one mixture components can be eliminated.

However, suppose a data set is generated by a single Gaussian distribution. If, for example, $k = 2$, the optimal solution for the Hard-EM algorithm are two mixtures which each contain roughly half of the points.

Thus, Hard-EM works poorly in practice if clusters are overlapping.

Soft-EM

To fix the issues of Hard-EM, we can allow a data point to be assigned to multiple clusters at the same time.

Suppose we have models $P(z \mid \theta)$ and $P(\boldsymbol{x} \mid \theta)$. Then, for each data point, the posterior distribution over cluster membership can be calculated, i.e. inferring distributions over the latent variables z.

$\gamma_j(\boldsymbol{x})$ is the *responsibility*, denoting the probability that $\boldsymbol{x}$ belongs to component j, i.e. $\gamma_j(\boldsymbol{x}) = P(Z = j \mid \boldsymbol{x}, \theta)$. This can be calculated as

$$\gamma_j(\boldsymbol{x}) = \frac{\overbrace{w_j}^{P(Z=j|\theta)} \overbrace{P(\boldsymbol{x} \mid \Sigma_j, \boldsymbol{\mu}_j)}^{P(\boldsymbol{x}|Z=j,\theta)}}{\underbrace{\sum_{l=1}^{k} w_l P(\boldsymbol{x} \mid \Sigma_l, \boldsymbol{\mu}_l)}_{P(x|\theta)}}$$

At the MLE $(\boldsymbol{\mu}^*, \Sigma^*, \boldsymbol{w}^*)$, it must hold that

$$\begin{aligned}
\boldsymbol{\mu}_j^* &= \frac{\sum_{i=1}^n \gamma_j(\boldsymbol{x}_i)\boldsymbol{x}_i}{\sum_{i=1}^n \gamma_j(\boldsymbol{x}_i)} \\
\Sigma_j^* &= \frac{\sum_{i=1}^n \gamma_j(\boldsymbol{x}_i)(\boldsymbol{x}_i - \boldsymbol{\mu}_j^*)(\boldsymbol{x}_i - \boldsymbol{\mu}_j^*)^T}{\sum_{i=1}^n \gamma_j(\boldsymbol{x}_i)} \\
w_j^* &= \frac{1}{n} \sum_{i=1}^n \gamma_j(\boldsymbol{x}_i)
\end{aligned}$$

All parameters are simply weighted averages, according to the responsibilities. The issue is that the equations are coupled, i.e. the γ_j depend on all the model parameters and vice versa.

The *Soft-EM* still allows to optimise them. For $t = 1, 2, \ldots$

E-step : Calculate cluster membership weights (aka responsibilities) for each point $\boldsymbol{x}_i$.

$$\gamma_j^{(t)}(\boldsymbol{x}_i) = \frac{w_j^{(t-1)} P(\boldsymbol{x}_i \mid \Sigma_j^{(t-1)}, \boldsymbol{\mu}_j^{(t-1)})}{\sum_{l=1}^{k} w_l^{(t-1)} P(\boldsymbol{x}_i \mid \Sigma_l^{(t-1)}, \boldsymbol{\mu}_l^{(t-1)})}$$

M-step : Fit clusters to weighted data points, i.e. calculate the closed form ML solution.

$$\boldsymbol{\mu}_j^{(t)} = \frac{\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)\boldsymbol{x}_i}{\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)}$$

$$\Sigma_j^{(t)} = \frac{\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)(\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t)})(\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t)})^T}{\sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)}$$

$$w_j^{(t)} = \frac{1}{n} \sum_{i=1}^{n} \gamma_j^{(t)}(\boldsymbol{x}_i)$$

The *E* stands for *Expected sufficient statistics*, the *M* stands for *Maximum likelihood solution*.

Hard- vs Soft-EM

Soft-EM generally results in higher likelihood values as it is able to deal with overlapping clusters. However, Hard-EM might be useful in certain scenarios.

The term "EM" alone typically refers to Soft-EM.

Constrained GMMs

There are special cases of GMMs by constraining the parameters:

Spherical assumes features are independent and have the same variance for each component.

$$\Sigma_j = \sigma_j^2 \cdot \boldsymbol{I}_d$$

Diagonal assumes features are independent, i.e. Gaussian Naive Bayes in an unsupervised scenario.

$$\Sigma_j = diag(\sigma_{j,1}^2, \ldots, \sigma_{j,d}^2)$$

Tied assumes all (or some) covariance matrices are equal.

$$\Sigma_1 = \cdots = \Sigma_d$$

Full imposes no restrictions.

This is done by tying together parameters in the MLE.

Each constraint reduces the number of parameters and thus acts as a form of regularisation.

Comparison to k-means

Assume uniform weights $w_1 = \cdots = w_k = \frac{1}{k}$ and identical, spherical covariances $\Sigma_{1:k} = \sigma^2 \cdot \boldsymbol{I}_d$.

Then, the Hard-EM algorithm is equivalent to the k-means algorithm. TODO: Derivation?

Conversely, the k-means algorithm can be understood as limiting case of Soft-EM for GMMs. The assumptions are the same as above, with additionally the variances tending to zero. If $\sigma \to 0$, it holds that

$$\gamma_j(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{\mu}_j \text{ is (unique) closest to } \boldsymbol{x} \\ 0 & \text{if not (and there are no ties)} \end{cases}$$

Initialisation

Fitting GMMs is a non-convex problem and harder than k-means, i.e. NP-hard. Thus, initialisation is important.

For initialisation, we typically do

Weights equally or uniformly.

Means randomly or via k-means++.

Variances spherical, e.g. according to empirical variance in the data.

Model Selection

For GMMs, the number of components k needs to be selected. Generally, the same techniques as for k-means can be used.

However, if the data actually stems from a mixture of Gaussians, cross-validation can be used to select k. We then aim to maximise the log-likelihood on the validation set. Even if the underlying process is not a mixture of Gaussians, cross-validation typically works fairly well and the validation set provides a signal.

Degeneracy

The Soft-EM optimises a log-likelihood, and thus might overfit.

Suppose the 1-dimensional case with a single data point x. Then, the negative log-likelihood is

$$-\log P(x \mid \mu, \sigma) = \frac{1}{2} \underbrace{\log(2\pi\sigma^2)}_{\to -\infty \text{ if } \sigma \to 0} + \frac{1}{2\sigma^2} \underbrace{(x-\mu)^2}_{0 \text{ if } \mu = x}$$

and the minimum is given by selecting $\mu = x$ and then letting $\sigma \to 0$. Then, the loss converges to $-\infty$.

Therefore, the optimal GMM chooses $k = n$ and puts one Gaussian on each data point with variance tending to 0. This is a case of overfitting and explains poor test set log-likelihood.

Degeneracy (i.e. variances tending towards 0) can be avoided by adding a small term ν^2 to the diagonal of the MLE:

$$\Sigma_j^{(t)} = \frac{\sum_{i=1}^n \gamma_j^{(t)}(\boldsymbol{x}_i)(\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t)})(\boldsymbol{x}_i - \boldsymbol{\mu}_j^{(t)})^T}{\sum_{i=1}^n \gamma_j^{(t)}(\boldsymbol{x}_i)} + \underbrace{\nu^2 \boldsymbol{I}}_{\text{additional term}}$$

From a Bayesian standpoint, this is equivalent to placing a (conjugate) Wishart prior on the covariance matrix, and computing the MAP instead of MLE.

ν can be chosen using cross-validation.

Use Cases

Mixture models are useful because they can encode assumptions about shape (e.g. fit ellipses instead of points), can be part of more complex models (e.g. classifiers), can output the likelihood of $P(\boldsymbol{x})$ and are naturally useful for semi-supervised learning.

Clustering

GMMs are naturally useful for clustering. Either one or multiple Gaussians can be used to model clusters.

If only one Gaussian is used per cluster, a normal GMM can be fitted. Points are then assigned such that $P(z \mid \boldsymbol{x}, \theta)$ is maximised.

However, assuming that each cluster has a Gaussian shape is a strong assumption. Thus, we can also model each class as a collection of clusters, i.e. each class is described as a mixture of Gaussians.

TODO: How exactly should this work?

Gaussian-Mixture Bayes Classifiers

GMMs can be used in GBCs to estimate $P(\boldsymbol{x} \mid y)$.

Thus, we first estimate the class prior $P(y)$ as usual. Then, for each class y, instead of estimating $P(\boldsymbol{x} \mid y)$ as a single Gaussian, it is estimated as a GMM:

$$P(\boldsymbol{x} \mid y) = \sum_{j=1}^{k_y} w_j^{(y)} \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_j^{(y)}, \boldsymbol{\Sigma}_j^{(y)})$$

Estimation is only done on the $\boldsymbol{x}_i$ with $y_i = y$.

Prediction is then done the same way as it is done with normal GBCs:

$$P(y \mid \boldsymbol{x}) = \frac{1}{Z} p(y) \sum_{j=1}^{k_y} w_j^{(y)} \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_j^{(y)}, \boldsymbol{\Sigma}_j^{(y)})$$

Density Estimation

GMMs can also be used for density estimation. $P(\boldsymbol{x})$ is modelled as a Gaussian mixture, $P(y \mid \boldsymbol{x})$ as some other model (e.g. logistic regression, ANN).

Then, the joint density is given as

$$P(\boldsymbol{x}, y) = P(\boldsymbol{x}) \cdot P(y \mid \boldsymbol{x})$$

This combines the robustness of discriminative models with the ability to detect outliers.

Anomaly Detection

Outlier detection is done by setting some *threshold* $\tau \in (0, 1)$. Then, $\boldsymbol{x}$ is an outlier if and only if $P(\boldsymbol{x}) < \tau$.

Setting the decision threshold if no examples of outliers are available is challenging. For example, a possible goal is to have a constant fraction of the probability mass be represented as anomaly, e.g. 1%. We can think of anomalies as a positive class. It is thus often called *one-class classification*.

If examples are available, cross-validation can be used to select τ. Varying τ trades false-positives against false-negatives. Thus, precision-recall or ROC curves can be used as an evaluation criterion. F1 score and related metrics can be used to select the number of clusters and other hyperparameters.

Semi-Supervised Learning

Semi-supervised learning can be interpreted as a missing-data problem where some but not all labels are missing. Semi-supervised learning is learning from a large amount of unlabelled data and a small amount of labelled data.

GMMs can easily be extended to semi-supervised learning.

For instances $\boldsymbol{x}_i$ with known label y_i, it must hold that $\gamma_j(\boldsymbol{x}_i) = [j = y_i]$. Thus, the EM algorithm can be modified. In the E-step, we differentiate between labelled and unlabelled samples. For unlabelled samples, we calculate the original responsibility $\gamma_j^{(t)}(\boldsymbol{x}_i) = P(Z = j \mid \boldsymbol{x}_i, \boldsymbol{\mu}^{(t-1)}, \Sigma^{(t-1)}, \boldsymbol{w}^{(t-1)})$. For labelled samples, we set $\gamma_j(\boldsymbol{x}_i) = [j = y_i]$. The M-step does not change.

Theory behind the EM Algorithm

Equivalent formulation

The *complete data log-likelihood* under θ is

$$\log P(\boldsymbol{x}_{1:n} \mid \theta)$$

or alternatively, including latent variables

$$\log P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta)$$

where z_i describes the responsibilities (as a random variable) of data point $\boldsymbol{x}_i$.

The *expected complete data log-likelihood* as a function of θ, written as $Q(\theta; \tilde{\theta})$, is

$$\begin{aligned} Q(\theta; \tilde{\theta}) &= \mathbb{E}_{z_{1:n}}[\log P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta) \mid \boldsymbol{x}_{1:n}, \tilde{\theta}] \\ &= \sum_{z_{1:n}} P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \tilde{\theta}) \log P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta) \end{aligned}$$

$Q(\theta; \tilde{\theta})$ is the EM objective.

We can show that the EM algorithm does the following: In the E-step, the expected data log-likelihood (as a function of θ) $Q(\theta; \theta^{(t-1)})$ is calculated. In the M-step, it is maximised:

$$\theta^{(t)} = \arg\max_{\theta} Q(\theta; \theta^{(t-1)})$$

$Q(\theta; \theta^{(t-1)})$ can be simplified:

$$\begin{aligned}
Q(\theta; \theta^{(t-1)}) &= \mathbb{E}_{z_{1:n}}\left[\log P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \\
&\overset{iid}{=} \mathbb{E}_{z_{1:n}}\left[\sum_{i=1}^{n} \log P(\boldsymbol{x}_i, z_i \mid \theta) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \\
&= \sum_{i=1}^{n} \mathbb{E}_{z_{1:n}}\left[\log P(\boldsymbol{x}_i, z_i \mid \theta) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \\
&= \sum_{i=1}^{n} \mathbb{E}_{z_i}\left[\log P(\boldsymbol{x}_i, z_i \mid \theta) \mid \boldsymbol{x}_i, \theta^{(t-1)}\right] \\
&= \sum_{i=1}^{n} \sum_{j=1}^{k} \underbrace{P(z_i = j \mid \boldsymbol{x}_i, \theta^{(t-1)})}_{\gamma_j(\boldsymbol{x}_i)} \cdot \log \underbrace{P(\boldsymbol{x}_i, z_i = j \mid \theta)}_{w_j \mathcal{N}(\boldsymbol{x}_i; \boldsymbol{\mu}_j, \Sigma_j)}
\end{aligned}$$

Thus, the EM objective function is equivalent to

$$Q(\theta; \theta^{(t-1)}) = \sum_{i=1}^{n} \sum_{z_i=1}^{k} \gamma_{z_i}(\boldsymbol{x}_i) \log P(\boldsymbol{x}_i, z_i \mid \theta)$$

Thus, the E-step is equivalent to computing the *expected sufficient statistics* $\gamma_z(\boldsymbol{x}_i) = P(z \mid \boldsymbol{x}_i, \theta^{(t-1)})$.

In the M-step, we compute

$$\theta^{(t)} = \arg\max_{\theta} Q(\theta; \theta^{(t-1)}) = \arg\max_{\theta} \sum_{i=1}^{n} \sum_{z_i=1}^{k} \gamma_{z_i}(\boldsymbol{x}_i) \log P(\boldsymbol{x}_i, z_i \mid \theta)$$

This is equivalent to training a GBC with weighted data. This has a closed-form solution (which is given in the EM algorithm pseudocode).

Convergence

Using the alternative formulation, we can prove that the EM algorithm monotonically increases the likelihood:

$$\log P(\boldsymbol{x}_{1:n} \mid \theta^{(t)}) \geq \log P(\boldsymbol{x}_{1:n} \mid \theta^{(t-1)})$$

First, we rewrite

$$P(\boldsymbol{x}_{1:n} \mid \theta) = P(\boldsymbol{x}_{1:n} \mid \theta) \cdot \frac{P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta)}{P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta)} = \frac{P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta)}{P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta)}$$

The logarithm and expectation are both monotonous functions and therefore do not change monotonicity. Thus, applying both functions to $P(\boldsymbol{x}_{1:n} \mid \theta)$

gives

$$
\begin{aligned}
&\mathbb{E}_{z_{1:n}}\left[\log P(\boldsymbol{x}_{1:n} \mid \theta) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \\
&= \mathbb{E}_{z_{1:n}}\left[\log \frac{P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta)}{P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta)} \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \\
&= \mathbb{E}_{z_{1:n}}\left[\log P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta) - \log P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \\
&= \underbrace{\mathbb{E}_{z_{1:n}}\left[\log P(\boldsymbol{x}_{1:n}, z_{1:n} \mid \theta) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right]}_{Q(\theta;\theta^{(t-1)})} - \underbrace{\mathbb{E}_{z_{1:n}}\left[\log P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right]}_{f(\theta)}
\end{aligned}
$$

Therefore, for monotonicity it suffices to show

$$
\begin{aligned}
Q(\theta^{(t)}; \theta^{(t-1)}) &\geq Q(\theta^{(t-1)}; \theta^{(t-1)}) \\
f(\theta^{(t)}) &\leq f(\theta^{(t-1)})
\end{aligned}
$$

The first inequality follows directly from the fact that in the M-step, we take $\theta^{(t)} = \arg\max_\theta Q(\theta; \theta^{(t-1)})$.

It remains to show that $f(\theta^{(t)}) \leq f(\theta^{(t-1)})$, i.e.

$$
\begin{aligned}
&\mathbb{E}_{z_{1:n}}\left[\log P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t)}) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \leq \mathbb{E}_{z_{1:n}}\left[\log P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \\
&\Leftrightarrow \mathbb{E}_{z_{1:n}}\left[\log P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}) - \log P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t)}) \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \geq 0 \\
&\Leftrightarrow \mathbb{E}_{z_{1:n}}\left[\log \frac{P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)})}{P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t)})} \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}\right] \geq 0 \\
&\Leftrightarrow \sum_{z_{1:n}} P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}) \log \frac{P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)})}{P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t)})} \geq 0 \\
&\Leftrightarrow KL(P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t-1)}) \| P(z_{1:n} \mid \boldsymbol{x}_{1:n}, \theta^{(t)})) \geq 0
\end{aligned}
$$

This holds because the KL-divergence is non-negative.

The *Kullback-Leibler divergence* $KL(p\|q)$ for two distributions p, q is defined as

$$
KL(p\|q) = \mathbb{E}_{\boldsymbol{x}\sim p}\left[\log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}\right] = \begin{cases} \sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} & \text{discrete} \\ \int p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} d\boldsymbol{x} & \text{continuous} \end{cases}
$$

The KL-divergence is non-negative and in general not symmetric.

Hard-EM Convergence

TODO: Hard-EM convergence

EM Algorithm more generally

The EM algorithm is much more widely applicable. It can be used whenever the E and M steps are tractable, i.e. it is possible to *compute* and *maximise* the complete data likelihood.

The EM algorithm can even be used in the case of missing features, not only missing labels.

Bernoulli Mixture Models

TODO: Bernoulli mixture models, tutorial 14

Useful Concepts

Entropy

For a discrete probability distribution p, the *entropy* is defined as

$$\mathcal{H}(p) = \sum_x -p(x) \log p(x)$$

The more "spread out" a distribution is, the higher its entropy, and the more concentrated the lower its entropy. A uniform distribution has the highest entropy.

Jensen's Inequality

If f is a convex function, *Jensen's inequality* is

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(x)]$$

If X is constant, this is equal. If f is a concave function, the inequality is reversed.

KL Divergence

The *KL-divergence* measures difference between two distributions. Let p and q be discrete probability distributions. Then, the KL-divergence is defined as

$$KL(p\|q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

The KL-divergence is only defined if for each x $q(x) = 0 \Rightarrow p(x) = 0$, and in that case is zero at that point.

It is also generally not symmetric and becomes zero if $p = q$.

It can be shown that the KL-divergence is always positive using Jensen's inequality:

$$KL(p\|q) = -\sum_x p(x) \log \frac{p(x)}{q(x)} \geq -\log \sum_x p(x) \frac{p(x)}{q(x)} = 0$$

2.14 Generative Adversarial Networks

The previously considered generative models were very simple and fail to capture high-dimensional, complex data such as audio or images. Thus, the goal is to use neural networks for generative modelling.

Given a sample $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, *implicit generative models* are models of the form

$$\boldsymbol{X} = G(\boldsymbol{Z}; \boldsymbol{w})$$

where $\boldsymbol{Z}$ is a simple distribution (e.g. low-dimensional Gaussian), G is a flexible non-linear mapping (e.g. neural network) and $\boldsymbol{X}$ is the data-generating process.

The key challenge with implicit generative models is to compute the data likelihood, which is generally not possible. If we start with a simple distribution and know $P(z)$, what is $P(\boldsymbol{x})$? This not only depends on the prior but also on the parameters. The distribution is very complicated, intractable in general, and therefore the maximum likelihood cannot just be calculated as usual. Thus, surrogate objective functions are required for training. This leads to *Variational Autoencoders* (VAEs) and *Generative Adversarial Networks* (GANs).

Formulation

The key idea is to optimise the parameters such that the samples from the model are hard to distinguish from the real data samples. Distinguishing is a classification problem, thus a discriminator network is trained to do so. This is called *discriminative learning*.

Now, two neural networks are trained simultaneously. The *generator* network $G : \mathbb{R}^m \to \mathbb{R}^d$ tries to generate realistic examples; the *discriminator* network $D : \mathbb{R}^d \to [0, 1]$ tries to distinguish real and generated samples.

The discriminator wants

$$D(\boldsymbol{x}) = \begin{cases} \approx 1 & \text{if } x \text{ is real} \\ \approx 0 & \text{if } x \text{ is generated} \end{cases}$$

The generator wants

$$D(G(\boldsymbol{z})) \approx 1 \text{ for samples } \boldsymbol{z}$$

This results in the following *minimax* game:

$$\min_G \max_D \mathbb{E}_{\boldsymbol{x} \sim \text{Data}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim \boldsymbol{Z}}[\log(1 - D(G(\boldsymbol{z})))]$$

Here, the logarithm was used, but there are other monotonic transformations used.

Training

The original objective optimises over G and D directly. However, we want to fix the function class (i.e. neural network architecture) and optimise over weights:

$$\min_{\boldsymbol{w}_G} \max_{\boldsymbol{w}_D} \underbrace{\mathbb{E}_{\boldsymbol{x}\sim\text{Data}}[\log D(\boldsymbol{x};\boldsymbol{w}_D)] + \mathbb{E}_{\boldsymbol{z}\sim\boldsymbol{Z}}[\log(1 - D(G(\boldsymbol{z};\boldsymbol{w}_G);\boldsymbol{w}_D)]}_{M(\boldsymbol{w}_G,\boldsymbol{w}_D)}$$

From the conflicting objective, optimising

$$\min_{\boldsymbol{w}_G} \max_{\boldsymbol{w}_D} M(\boldsymbol{w}_G, \boldsymbol{w}_D)$$

requires finding a *saddle point* rather than a minimum.

If G and D have enough capacity, the data-generating distribution is actually a saddle point of $\min_{\boldsymbol{w}_G} \max_{\boldsymbol{w}_D} M(\boldsymbol{w}_G, \boldsymbol{w}_D)$. This is because $\max_{\boldsymbol{w}_G} M(\boldsymbol{w}_G, \boldsymbol{w}_D)$ is up to constants the Jensen-Shannon divergence $JS(P_{\text{Data}} \| P_G)$.

The *Jensen-Shannon divergence* between two distributions p, q is defined as

$$JS(p\|q) = \frac{1}{2} KL(p\|\frac{p+q}{2}) + \frac{1}{2} KL(q\|\frac{p+q}{2})$$

and is sort-of a symmetric version of the KL-divergence. It holds (under some restrictions) that $JS(p\|q) = 0 \Leftrightarrow p = q$.

Commonly, (mini-batch) SGD is applied to the generator and discriminator simultaneously:

$$\boldsymbol{w}_G^{(t+1)} = \boldsymbol{w}_G^{(t)} - \eta_t \nabla_{\boldsymbol{w}_G} M(\boldsymbol{w}_G, \boldsymbol{w}_D^{(t)})$$
$$\boldsymbol{w}_D^{(t+1)} = \boldsymbol{w}_D^{(t)} + \eta_t \nabla_{\boldsymbol{w}_D} M(\boldsymbol{w}_G^{(t)}, \boldsymbol{w}_D)$$

where the different signs indicate the difference of minimising/maximising. There are also other variants where the networks are updated sequentially.

Challenges

GANs are notoriously hard to train, and many tricks/heuristics exist to mitigate some problems.

Oscillations / Divergence

Because two conflicting objectives are optimised, it might happen that the weights oscillate around a local minimum or even diverge.

Mode Collapse

It might happen that the generator models certain modes of the data well but completely ignores others.

This can happen if the discriminator cannot detect that only samples from a certain mode are generated. It might also happen that the well-modelled modes switch as soon as the discriminator has adjusted.

Data Memorisation

The GAN objective is similar to minimising a JS-divergence. In practice, if the JS divergence of the sample data would be minimised, the optimal generator would just uniformly sample from the data set.

Thus, the model is prone to memorising the data set instead of actually learning the distribution. By going away from the idealised setting, i.e. making restricting the discriminator capacity, we get better samples.

Evaluation

Evaluating GANs is currently an unresolved issue. Because the likelihood is intractable, it cannot be calculated on a holdout set.

There exist various heuristics, but no domain-independent solutions.