Machine Learning for Complex Networks

Prof. Dr. Ingo Scholtes

Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany

ingo.scholtes@uni-wuerzburg.de

Lecture 09
Supervised Graph Learning and Neural Networks

June 26, 2024

Notes:

- **Lecture L09:** Supervised Graph Learning and Neural Networks 26.06.2024
- **Educational objective:** We introduce logistic regression and apply it to supervised learning in graphs. We introduce feed-forward neural networks and show how we can use backpropagation and stochastic gradient descent to train them for supervised graph learning tasks.
  - Supervised Learning and Logistic Regression
  - Node Classification and Supervised Link Prediction
  - Perceptron and Feed-Forward Neural Networks
  - Backpropagation & Stochastic Gradient Descent (SGD)

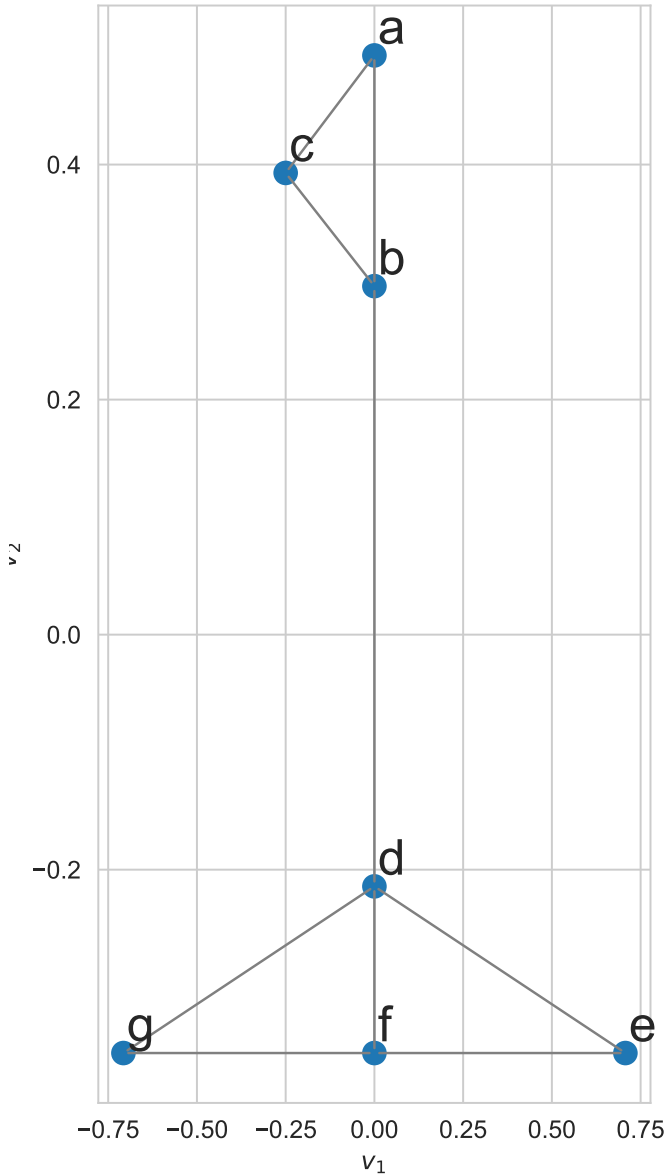
- **Exercise sheet 08:** Neural Networks and Supervised Link Prediction 03.07.2024

Motivation

- so far, we focused on **unsupervised learning tasks**
- we introduced matrix factorization techniques to obtain **Euclidean representations of networks**
- we can use those representations to apply **supervised learning techniques**

classification problem

- for features $\mathbb{F}$ and discrete classes $\mathbb{C}$ learn **classifier** $C : \mathbb{F} \rightarrow \mathbb{C}$
- **applications in graph learning**
  1. node/link classification
  2. supervised link prediction
  3. graph classification



representation of nodes in **two-dimensional feature space**

Notes:

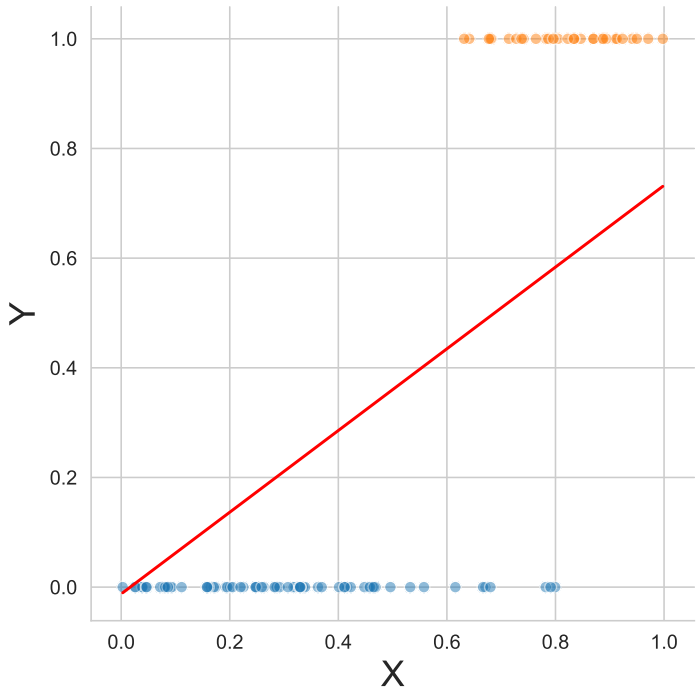
- In all lectures so far, we have focused on **unsupervised** graph learning tasks such as community detection or (unsupervised) similarity-based link prediction. In the last lecture, we further introduced graph representation learning based on matrix factorization, which we can view as a dimensionality reduction for graph data.
- The mapping of nodes to vectors in a Euclidean space is the foundation to apply **supervised learning in graphs**. In this lecture, we will reconsider two graph learning tasks that we have already considered and cast them as a **supervised classification problem**. We specifically consider **node classification**, which we can view as a supervised variant of community detection, i.e. we consider data where we have ground truth data on group labels that we can use to train a classifier. Moreover, we will introduce supervised link prediction, which can be cast as binary "node pair" classification, i.e. we use node pairs in a training network to train a supervised classifier.
- We will introduce three different supervised learning techniques that can be used to address this. We start with logistic regression, a statistical classification technique for which we will benefit from our discussion of statistical inference and likelihood maximization from previous weeks. We use this to illustrate gradient-based optimization and generalize it to the perceptron classification algorithm, which is the fundamental building block of (deep) neural networks. We finally introduce feed-forward neural networks, which enables us to address node classification and supervised link prediction with non-linear patterns.

Classification by regression?

- can we classify data with **linear model**

$$Y = \beta_0 + \beta_1 X$$

where dependent variable $Y \in \mathbb{R}$ encodes k classes $\mathbb{C}$?



problem 1

1. $\hat{Y} = \beta_0 + \beta X \in \mathbb{R}$
2. need to **map prediction to discrete classes** $\mathbb{C}$, e.g. choose class closest to prediction

problem 2

1. we impose total order on $\mathbb{C}$
2. $\{A, B, C\} \to \{0, 1, 2\}$ or $\{A, B, C\} \to \{0, 2, 1\}$?
3. for $k > 2$ **order** affects classification

Notes:

- We start with an introduction of **classification by logistic regression**. Consider a linear model for a dependent variable Y and an independent variable X, i.e. we model a linear relationship $Y = \beta_0 + \beta_1 X$ where β_0 is the intercept (or "bias") β_1 is the slope of the regression line.
- Why can we not simply address classification by a linear regression where the dichotomous dependent variable is encoded by numerical values zero and one? While this seems simple, it introduces two problems:
- First, the fitted model assumes values in $\mathbb{R}$ and predictions are thus not limited to categorical values. We must thus map the predicted values $\hat{y}$ to the possible classes of our classification problem. In principle, we could solve this problem, e.g., by assigning the class whose value is closest to the predicted value.
- The second problem is more severe and it is due to the fact that classification is a fundamentally different problem than regression. In classification we do not assume that classes are ordered, i.e. we consider $\mathbb{C}$ to be an unordered set. However, any mapping of the elements of an unordered set to $\mathbb{R}$ necessarily imposes a total order on the elements and this order actually affects the coefficients of our fitted model. If we only have $k = 2$ classes (i.e. we have a binary classification problem with dichotomous classes), changing the ordering of the classes only changes the sign of the slope. We can account for this by reverting our decision rule accordingly. However, if we have more than two classes, the ordering will change the slope of the fitted regression model in a way that we cannot account for by reverting our decision rule. For this reason, we cannot apply linear regression to classification if $k > 2$.

Classification by regression?

- consider **statistical classification**, i.e. learn $P(C = c|X)$ that object with feature X belongs to class c
- for dichotomous $\mathbb{C} = \{0, 1\}$ consider

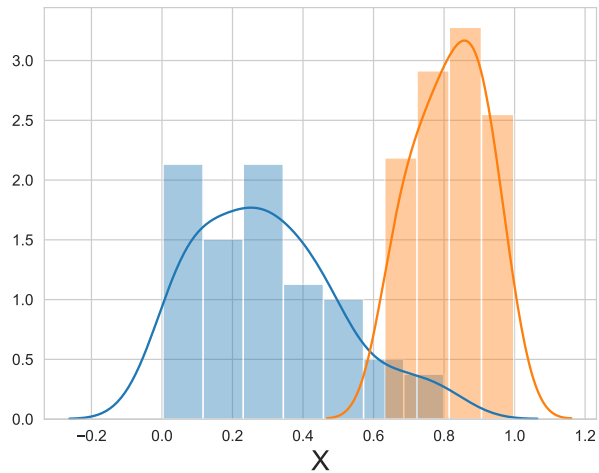
$$P(Y = 1|X) = \beta_0 + \beta_1 X$$

i.e. **linear model for probability of class** 1

- we need transformation function σ such that

$$P(Y = 1|X) = \sigma(\beta_0 + \beta_1 X) \in [0, 1]$$

- binary statistical classification $\rightarrow$ **regression with non-linear transformation**



distribution of feature X within two classes $\mathbb{C} = \{0, 1\}$

Notes:

- However, we can use a trick to nevertheless apply linear regression to **binary classification** (where the ordering in which encode the two classes does not matter). Rather than using the independent variable(s) to model the class, we can instead model the probability of a class, i.e. we replace the dependent variable by a probability.
- We now have a real-valued dependent variable, but we now face another problem: A linear regression model assumes values in $\mathbb{R}$ while we are interested in a model that is limited to values in $[0, 1]$. To address this, we consider a linear model with an additional (necessarily non-linear) mapping function σ, that maps any value in $\mathbb{R}$ to a value in $[0, 1]$ that we can interpret as probability.

Logit and logistic function

- for $p \in (0, 1)$ consider **logarithm of odds**

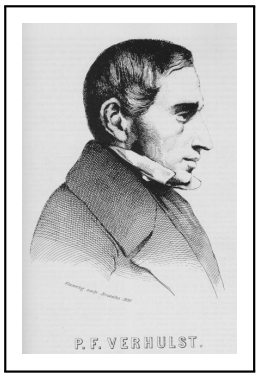
$$\text{logit}(p) := \log_b \frac{p}{1-p}$$

where b is a logarithm base

- **logit function** maps probabilities in $(0, 1)$ to $\mathbb{R}$, where base b determines **units**
- for $b = e$ inverse is called **logistic function**

$$\sigma(x) := \text{logit}^{-1}(x) = \frac{1}{1 + e^{-x}}$$

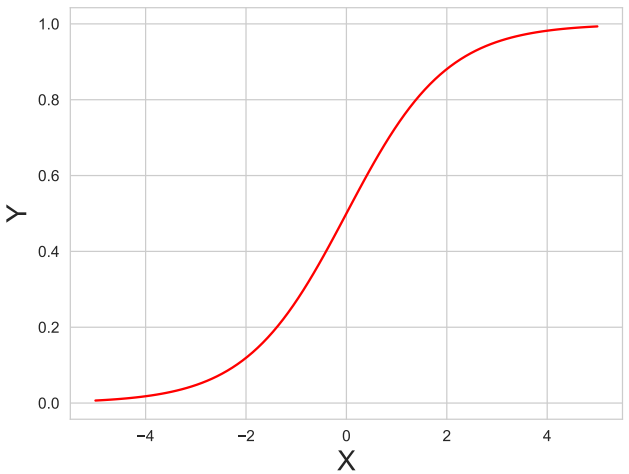
- s-shaped (sigmoid) function that maps values in $\mathbb{R}$ to $(0, 1)$
- special case of **softmax function** $\sigma : \mathbb{R}^k \to (0, 1)^k$ for $k = 1$



Pierre François Verhulst

1804 - 1849

image credit: Wikipedia, CC-BY-SA



plot of $\sigma(x) := \text{logit}^{-1}$ in $[-5, 5]$

Notes:

- How do we choose the mapping function? We first consider the so-called **logit** function, which – for any probability $p \in [0, 1]$ – is defined as the logarithm of odds, i.e. the logarithm of p divided by the probability $1 - p$. This odds ratio intuitively captures how much more likely it is that an event happens, vs. the event not happening. You may have heard about odds in the context of gambling, where it is used to determine the payout of a bet. The odds of an event that happens with probability $\frac{1}{2}$ is 1:1, the odds of an event with probability $p = \frac{1}{3}$ is 1:2 and so on. Since for any $p > 0$ the odds ratio assumes values in $(0, \infty)$, the log-odds assumes values in $(-\infty, \infty)$.
- Since the logit-function maps values from $(0, 1)$ to $\mathbb{R}$, the inverse of this function, the so-called logit-function maps values from $\mathbb{R}$ to $(0, 1)$. It defines an s-shaped sigmoid function, as shown in the plot above.
- As we shall see later, the logistic function plays an important role in the context of neural networks, where it is often used as an activation function that determines the output of a single neuron based on the linear combination of inputs. The logistic function, which maps inputs $\mathbb{R}$ to $(0, 1)$, is a special case of the multi-variate **softmax function**, which maps inputs from $\mathbb{R}^k$ to $(0, 1)^k$.

Logistic regression

- consider linear model for $P(Y=1|X)$ with **logistic transformation**

$$P(Y=1|X) = \sigma(\beta_0 + \beta_1 X) = \frac{1}{1+e^{-(\beta_0+\beta_1 X)}}$$

- we can rewrite this to

$$\frac{P(Y=1|X)}{1-P(Y=1|X)} = e^{\beta_0+\beta_1 X}$$

and thus

$$\log \frac{P(Y=1|X)}{P(Y=0|X)} = \beta_0 + \beta_1 X$$

- **logistic regression model** = linear model for logarithm of odds

Jane Worcester
? - 1989

image credit: www.hsph.harvard.edu

Notes:

- We can now **use the logistic function σ as non-linear transformation in our linear model of the class probability**, i.e. our learning equation is now defined as shown above. We can rewrite this to an equation, where the left side becomes the logit function and on the right side we simply have our linear model. In other words, we obtain a regression model for the logarithm of class odds. You now see that changing the labeling of the two classes on the left changes the sign of the logarithm and thus corresponds to a change of the signs of the model parameters.
- Since the logarithm is a monotonic transformation, we have that for $\beta_1 > 0$ an increase in X increases the odds of class Y, just like the in the standard linear model.
- We call this approach **logistic regression**, and it is hard to underestimate its practical relevance. Compared to other classification techniques, it is extremely simple, makes few assumptions about the data, has relatively small potential for overfitting and is not particularly sensitive to outliers. As a general rule, whenever you are confronted with a classification problem, you should at least try logistic regression before turning to methods that are more difficult to interpret and more costly to implement and evaluate.

Inference of model parameters

- we obtain model that assigns $P(Y = 1|x_i)$ for $x_i \in \mathbb{R}$
- for training data (x_i, y_i) we can compute likelihood of parameters $\Theta := (\beta_0, \beta_1)$ as

$$\mathcal{L}(\Theta) = \prod_i P(y_i|x_i) = \prod_i P(1|x_i)^{y_i} \left(1 - P(1|x_i)\right)^{1-y_i}$$

- **log-likelihood function** is

$$\log \mathcal{L}(\Theta) = \sum_i y_i \log P(1|x_i) + (1 - y_i) \log \left(1 - P(1|x_i)\right))$$

- with $P(1|x_i) = \sigma(\beta_0 + \beta_1 X)$ we have

$$\log \mathcal{L}(\Theta) = \sum_i y_i \log \sigma(\beta_0 + \beta_1 x_i) + (1 - y_i) \log \left(1 - \sigma(\beta_0 + \beta_1 x_i)\right)$$

- **maximum likelihood estimate** of coefficients

$$\hat{\Theta} := (\hat{\beta}_0, \hat{\beta}_1) := \underset{\Theta}{\operatorname{argmax}} \log \mathcal{L}(\Theta)$$

Notes:

- How can we estimate the coefficients of a logistic regression model? In (multiple) linear regression we compute the coefficients that minimize the squared error, which is equivalent to a maximum likelihood estimation of the model parameters.
- Since the logistic regression model provides us with conditional probabilities for the different classes, we can use training data to calculate model likelihood as the product of the class probabilities (conditional on the associated independent variable). Assuming that classes are encoded as zero and one, we can write the class probabilities based on a product of two terms, each being an expression of the positive class probability as shown above. For instances of the positive class $y_i = 1$ in our training data, the second term in the product is one and we are left with the probability of class $Y = 1$. For instances of the negative class $y_i = 0$ the first term is one and we are left with the probability of class $Y = 0$.
- We can now compute the log-likelihood function, which – since it is a monotonic transform – can be used to estimate the parameters that maximize the likelihood.

Gradient-based optimization

- we can use **partial derivatives** of $\log \mathcal{L}$ to compute **gradients** w.r.t. model parameters

$$\frac{\partial \log \mathcal{L}(\beta_0, \beta_1)}{\partial \beta_0} = \sum_i y_i - \sigma(\beta_0 + \beta_1 x_i)$$

$$\frac{\partial \log \mathcal{L}(\beta_0, \beta_1)}{\partial \beta_1} = \sum_i x_i \left[y_i - \sigma(\beta_0 + \beta_1 x_i) \right]$$

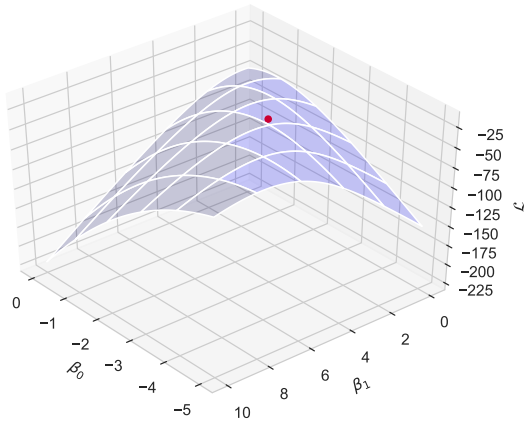
- **no analytical solution** for $\text{argmax}_\Theta(\mathcal{L}(\Theta))$

gradient ascent optimization

1. start at random point (β_0, β_1) in $\mathbb{R}^2$
2. move (β_0, β_1) along steepest **local gradient**, i.e.

$$\beta_0' := \beta_0 + \gamma \cdot \frac{\partial \log \mathcal{L}(\Theta)}{\partial \beta_0}, \beta_1' := \beta_1 + \gamma \cdot \frac{\partial \log \mathcal{L}(\Theta)}{\partial \beta_1}$$

3. repeat 2 until $\log \mathcal{L}(\beta_0', \beta_1') - \log \mathcal{L}(\beta_0, \beta_1) < \epsilon$



likelihood manifold of logistic regression model with coefficients β_0 and β_1 for example 1 shown on slide 9 (and slide 2)

Notes:

- In order to maximize the likelihood function we can calculate partial derivatives, which need to be zero at the maximum. To obtain the partial derivatives above, consider that $\frac{\partial \text{logistic}(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$.
- Unfortunately, the resulting expression does not have an analytical solution, i.e. different from the simple analytical solution for the parameters in linear regression we do not obtain a closed-form expression that allows us to directly calculate maximum likelihood estimates of parameters.
- However, we can use heuristic optimization methods to compute the maximum likelihood. In particular, the partial derivatives give us the **gradient** of the likelihood function in the direction of the two coefficients. We can use this to implement a gradient ascent algorithm. The idea behind this algorithm is really simple: consider you are on a mountain and it is foggy, i.e. you do not see in which direction the summit lies. You can just look at the terrain around you and move in the direction of the largest gradient, i.e. you simply move uphill. This necessarily takes you to a local maximum, at which point you can only walk downhill. For logistic regression, one can prove that if the likelihood function has a local maximum it is unique (see figure above), i.e. any local maximum is also the global maximum. This provides us with an efficient computational method to estimate model coefficients, even though we cannot derive an analytical solution.

Decision rule

- gradient ascent yields **maximum likelihood estimate** $\hat{\Theta} = (\hat{\beta}_0, \hat{\beta}_1)$
- for any given feature x we can calculate **class probability** as

$$P(Y = 1|x) = \sigma(\hat{\beta}_0 + \hat{\beta}_1 x)$$

- we define **odds-based classifier function** $C : \mathbb{F} \to \mathbb{C}$

$$C(x) = 1 \text{ iff } P(Y = 1|x) > P(Y = 0|x)$$

i.e.

$$C(X) = 1 \text{ iff } \frac{P(Y = 1|X)}{P(Y = 0|X)} > 1$$

- since we consider a linear model for logarithm of odds, this corresponds to **decision rule**

$$C(x) = 1 \text{ iff } \hat{\beta}_0 + \hat{\beta}_1 X > 0$$

i.e. **Heaviside step function**

Notes:

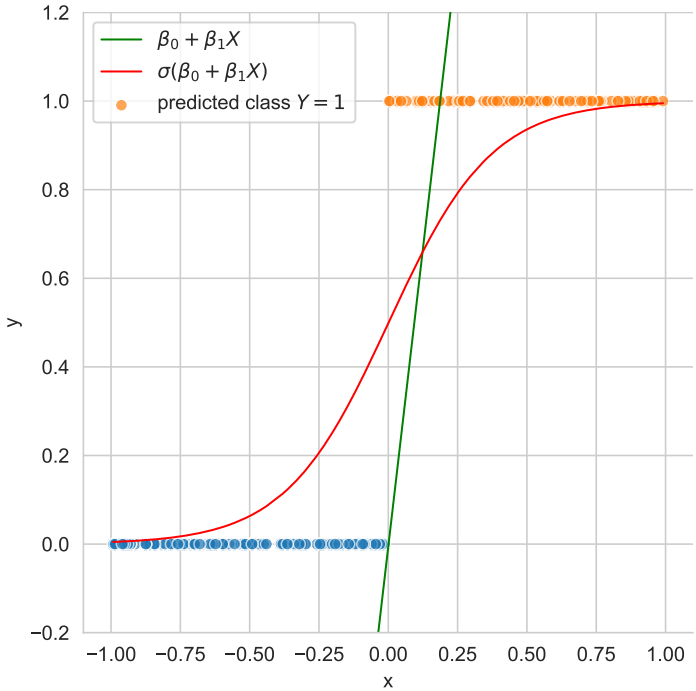
- We have found a maximum likelihood estimate for the model coefficients $\hat{\Theta} = (\hat{\beta}_0, \hat{\beta}_1)$. How can we use those coefficients to classify new instances of our data, e.g. in a test data set?
- With these optimal coefficients, we get an equation that allows us to compute the probability of class $Y = 1$ (as well as class $Y = 0$) for each value x_i of the independent variable X.
- Since we only have two classes, it is intuitive to assign the class for which the probability is higher, i.e. we use a decision rule that assigns the class for which the odds is larger than one. In logarithmic space, this corresponds to a decision rule that assigns class $Y = 1$ whenever the log-odds of this class is larger than zero. Considering that we have simply fitted a linear model for the logarithm of class odds, this is the case whenever the linear model assumes values larger than zero, while we assign class $Y = 0$ for values smaller or equal zero.
- We see that switching the coding of classes (i.e. exchanging the positive and negative class labels) simply changes the sign of the coefficients and thus inverts the decision rule.

Logistic regression: example

example 1

200 realisations of (X, Y) with

- $X \in$ Uniform on $[-1, 1]$ and
- $P(Y = 1) = 1$ for $X \geq 0$ and $P(Y = 1) = 0$ otherwise

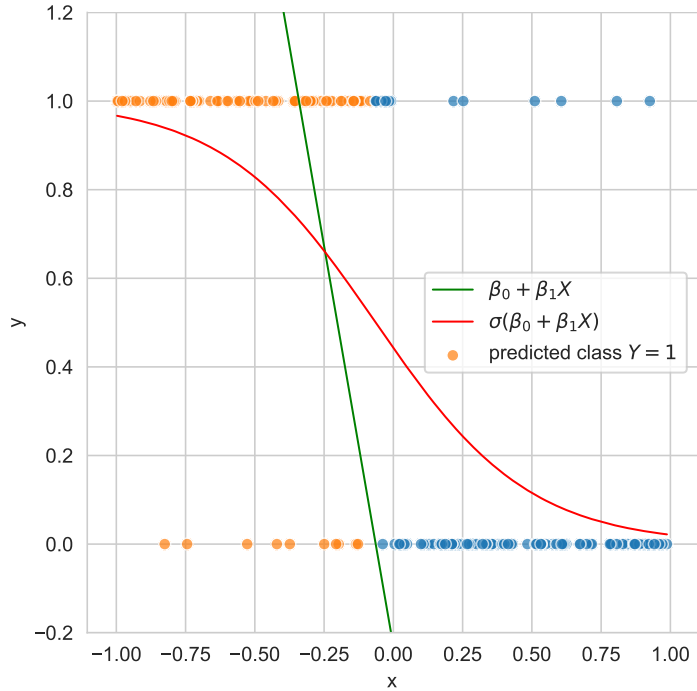


$\hat{\beta}_0 = -0.001, \hat{\beta}_1 = 5.37$

example 2

200 realisations of (X, Y) with

- $X \in$ Uniform on $[-1, 1]$ and
- $P(Y = 1) = 0.9$ for $X < 0$ and $P(Y = 1) = 0.1$ otherwise



$\hat{\beta}_0 = -0.20, \hat{\beta}_1 = -3.6$

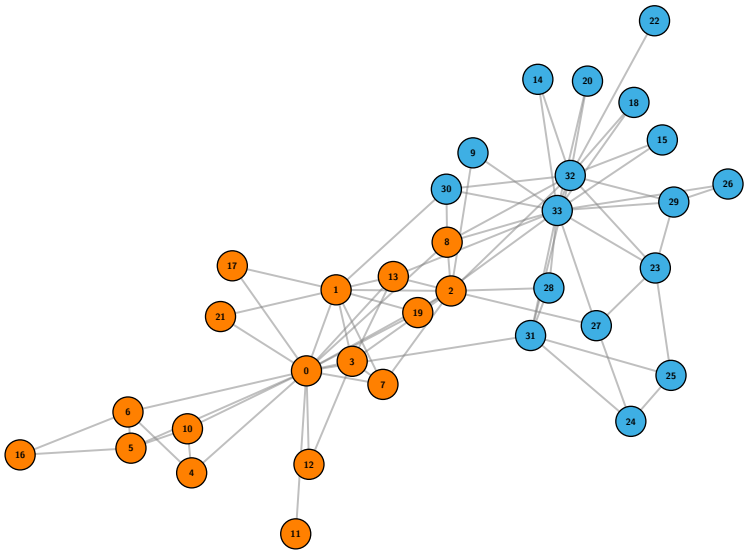
Notes:

- We illustrate logistic regression in two examples, where the independent variable X is uniformly distributed between -1 and 1. In the first example, there is a simple deterministic relationship, where class one is assigned whenever X is larger or equal than zero and class zero is assigned whenever X is smaller than zero. The fitted logistic regression model captures this relationship, which leads to a perfect prediction of classes in the training set. We plot the fitted logistic function (red) and the fitted linear equation for the log-odds (green). The green line crosses zero at $x = 0$ and the red line crosses 0.5 (at which point class one is more likely and thus the log-odds is larger than one) at $x = 0$. Due to the fact that the two classes are perfectly separable, we obtain a steep slope, which translates to a sharp transition of class probabilities around the decision boundary of $x = 0$.
- In the second example, we use a probabilistic class assignment, which results in an overlap between the two classes. We cannot get a perfect prediction since the decision rule is based on a threshold on x. We further changed the position of class one in terms of the independent variable, i.e. the majority of instances of class one have values smaller than zero. As a result, the sign of the fitted slope is reversed. The decision rule predicts class 1 for values of $X < 0$.
- While we introduced logistic regression with a single independent variable X, we can generalize it to **multiple logistic regression** by using a linear model $\beta_0 + \beta_1 X_1 + \beta_2 X_2$, i.e. we have a single intercept/bias β_0 and one slope parameter β_i for each feature dimension X_i. By combining decision boundaries of multiple one-vs-all classifiers, we can generalize logistic regression to non-binary classification.

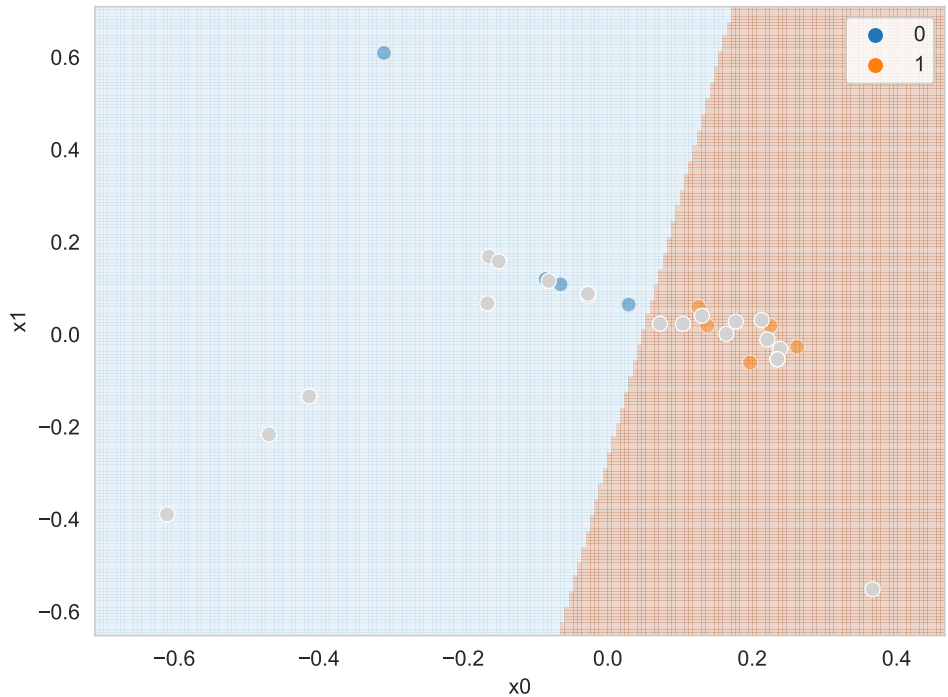
Supervised node classification

example

node classification in Karate club network with $n = 34$ nodes and $m = 77$ links, where ground truth node classes $\hat{y}$ are given by groups



Karate club network with ground truth node classes



decision boundary of logistic regression classifier using two-dimensional **Laplacian embedding** of nodes in test set (accuracy = 1.0)

Notes:

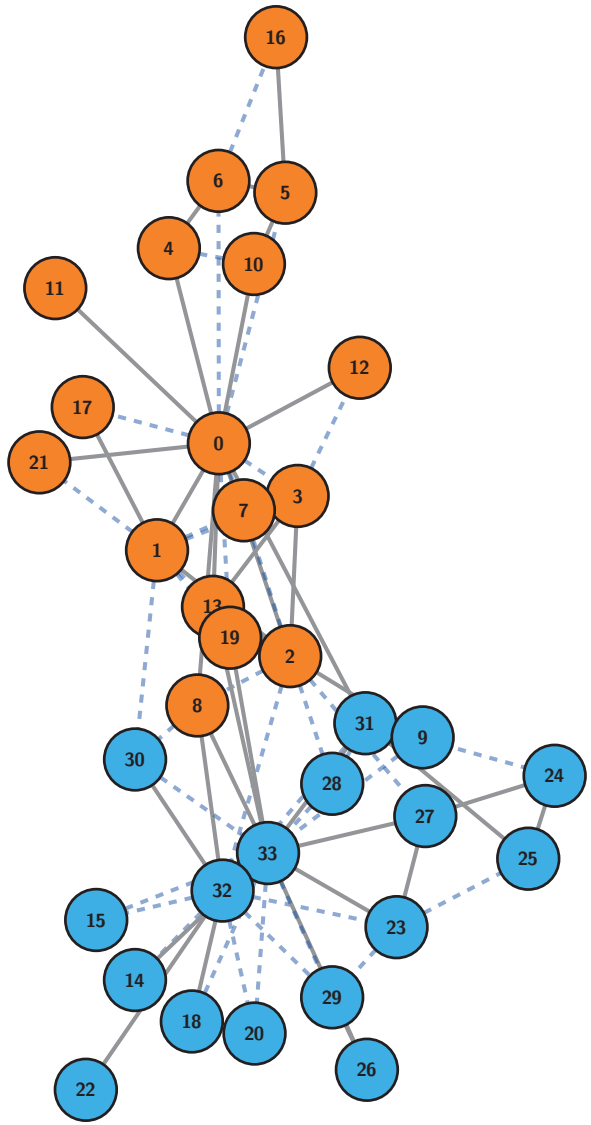
- We now apply logistic regression to a supervised graph learning tasks. We consider node classification in the Karate club network, where the two groups (or factions) of the club define the binary ground truth class label. We apply a Laplacian embedding of nodes into a two-dimensional Euclidean space, i.e. each node in the network has two features X_1 and X_2.
- We split the data into a training and test set and use the features and classes in the set of training nodes to fit a logistic regression model. This yields a linear decision boundary in our two-dimensional feature space, which we can use to assign class labels in the test set.
- In this example, we find that we can perfectly classify nodes in the test set based on logistic regression with a linear decision boundary and a two-dimensional Laplacian embedding. Naturally, many real-world examples are much more complicated and may require higher-dimensional embeddings or more advanced classification techniques with non-linear decision boundaries (more on this later).

Supervised link prediction

- in Lecture 07, we considered **unsupervised link prediction**
- ranking of node pairs based on **topological similarity** or **distance/cosine** between vector space embeddings
- useful if we want to identify **fixed small number of highest ranked node pairs**
- can we treat **link prediction as supervised binary classification problem**?

supervised link prediction

use training set of node pairs to learn **binary classifier** $f : V \times V \rightarrow \{0, 1\}$ that predicts links for node pairs in test set



Notes:

- As a second example, let us consider a supervised approach to link prediction. We first node that so far we have addressed link prediction as an unsupervised problem, as we only computed pair-wise similarity measures that we can use to rank node pairs according to their "probability" that a link between them exists. While this is sufficient to extract a small number of likely links (e.g. for a recommender system), we did not learn a decision boundary that could be used to exactly predict which links are likely to exist and which not. We rather evaluated our link prediction approaches based on the Receiver-Operator Characteristic and the AUC, which means we did not specify a specific discrimination threshold.
- We now consider how we can address link prediction as a binary classification problem. The idea is to use a set of training node pairs (and the links that may or may not exist between them) to train a binary classifier that can then be used to predict links in a test data set.

Using node features for link prediction

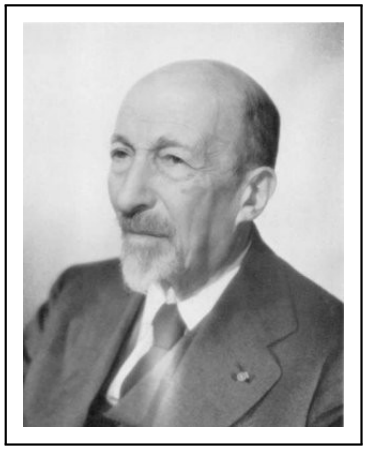
- graph representation learning techniques yield **node features** $x_i \in \mathbb{R}^d$ for $i \in V$
- to train classifier we need **features** $x_{ij} \in \mathbb{R}^d$ **for node pairs** $(i, j) \in V \times V$
- for $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times m}$ **Hadamard or Schur product** $A \circ B \in \mathbb{R}^{n \times m}$ is defined as **element-wise product**, i.e.

$$(\mathbf{A} \circ \mathbf{B})_{ij} = \mathbf{A}_{ij} \cdot \mathbf{B}_{ij}$$

- features $x_{ij} \in \mathbb{R}^d$ for node pairs (i, j) can be computed as

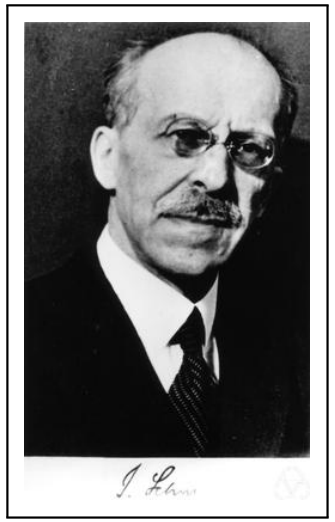
$$x_{ij} := x_i \circ x_j$$

based on node embeddings x_i, x_j



Jacques Hadamard
1865 – 1963

image credit: public domain



Issai Schur
1875 – 1941

image credit: https://opc.mfo.de/detail?photo_id=12209, CC BY-SA 2.0

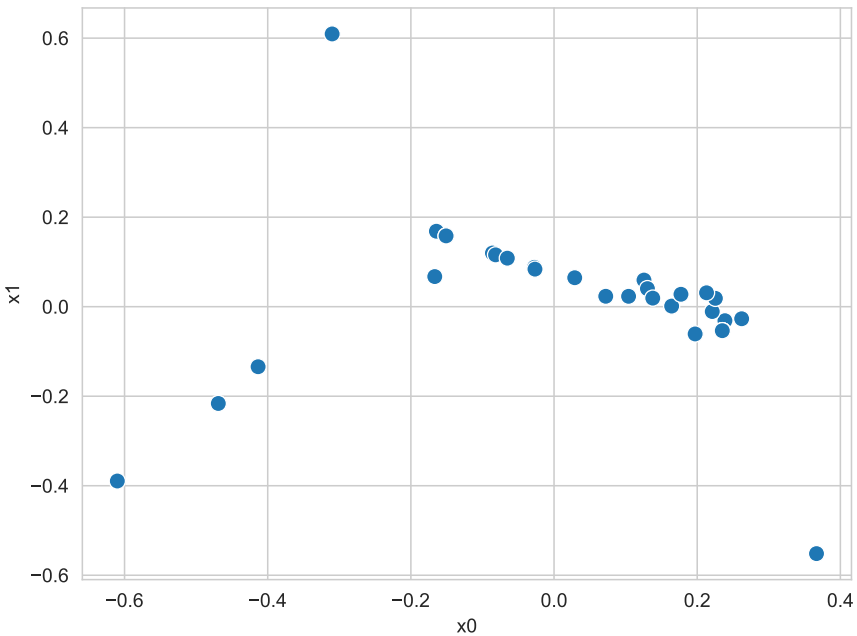
Notes:

- To apply binary classification algorithms like logistic regression to link prediction we must address one important gap: So far, we have used graph embedding techniques to calculate vector space embeddings of nodes. We can thus compute node features, but we need features of node pairs that allow us to classify them.
- Assuming that we have node features $x_i \in \mathbb{R}^d$ for nodes i we can address this by using a binary operator that maps pairs of node features $x_i, x_j \in \mathbb{R}^d$ to a new (node pair) feature $x_{ij} \in \mathbb{R}^{d'}$. While different operators have been tested in the literature, a simple yet powerful approach is to use the so-called Hadamard operator ir Schur product, which is defined as the element-wise product of two vectors or matrices.
- For a pair of nodes i, j we can thus assign the feature $x_{ij} = x_i \circ x_j \in \mathbb{R}^d$, which can now use for classification.

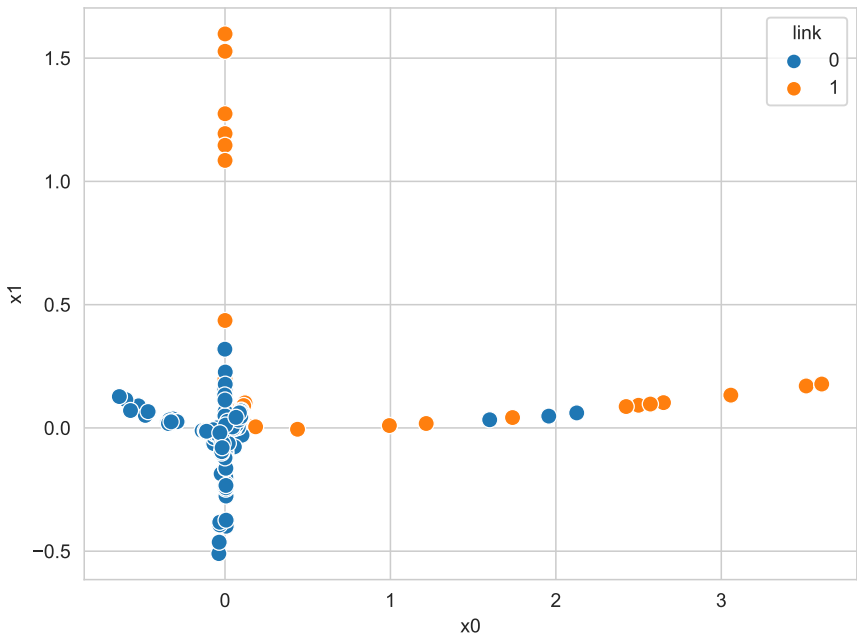
Supervised Link Prediction

empirical example

- network of homework cooperations between students at Ben-Gurion University
- $n = 185$ nodes, $m = 311$ links
- E_{train} = 90 % of links



representations $x_i \in \mathbb{R}^2$ of nodes i computed based on first two eigenvectors of graph Laplacian



Hadamard product $x_i \circ x_j \in \mathbb{R}^2$ for node pairs connected by link (orange) and randomly sampled unconnected node pairs (blue)

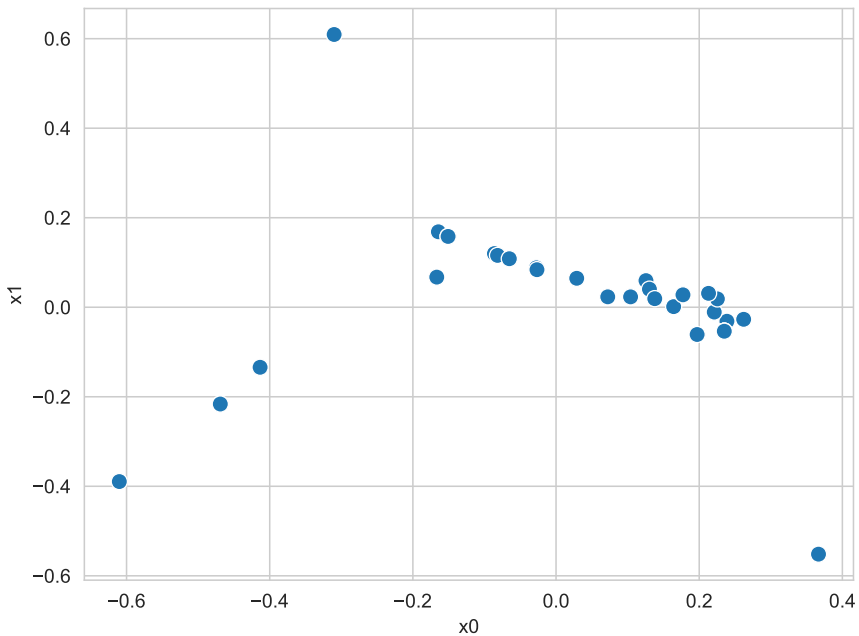
Notes:

- We can apply this to implement supervised link prediction in am empirical example, the network of homework cooperations between students at Ben Gurion University. For the purpose of illustration, we limit our analysis to a two-dimensional node embedding although we would probably achieve better results if we used a higher-dimensional embedding (you can test this as an exercise).
- Our approach is as follows: We first split the data into a training and test data set (by randomly selecting a fraction of the links). We then compute a two-dimensional Laplacian embedding of nodes in the training set and calculate the Hadamard product between the node features of all node pairs. This yields the feature space shown in the figure on the right.

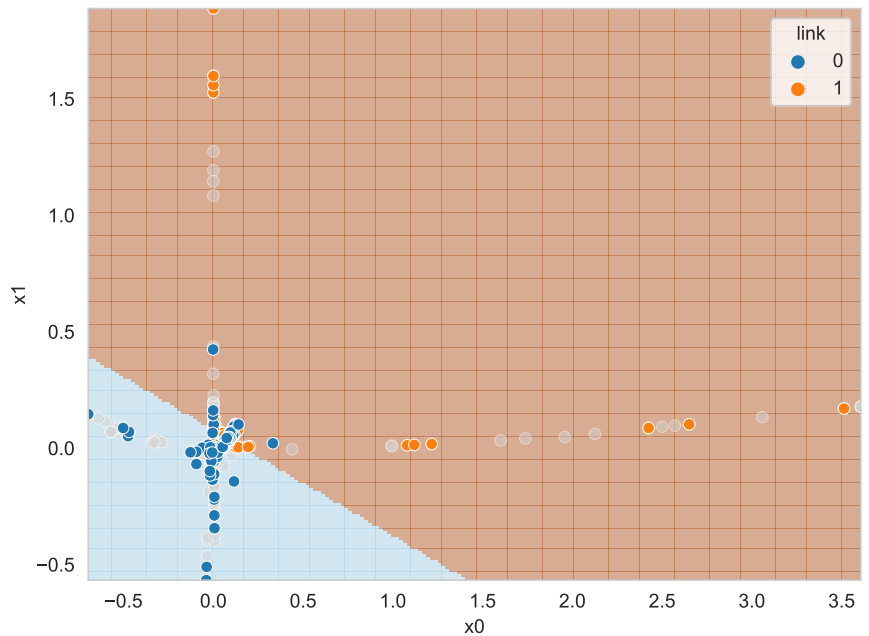
Supervised Link Prediction

empirical example

- network of homework cooperations between students at Ben-Gurion University
- $n = 185$ nodes, $m = 311$ links
- E_{train} = 90 % of links



representations $x_i \in \mathbb{R}^2$ of nodes i computed based on first two eigenvectors of graph Laplacian (colored by ground truth groups)



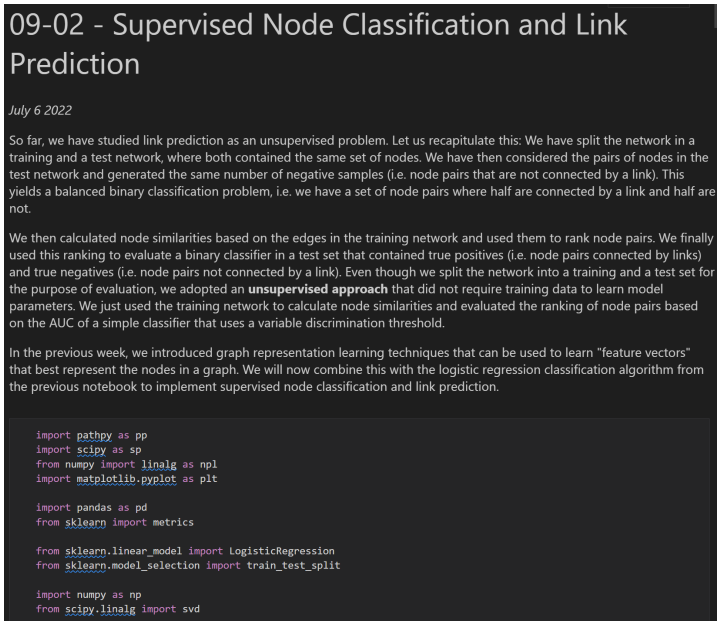
decision boundary of logistic regression using **Hadamard product** of Laplacian embedding in balanced test set (accuracy = 0.76)

Notes:

- Depending on whether a link exists or not for a given pair of nodes, we assign class labels to the node pairs and train a logistic regression classifier, which yields a linear decision boundary. We finally use this decision boundary to assign class labels (i.e. link or no link) to the node pairs in our test set. In this example, this simple approach yields an accuracy of 0.76, which we could probably improve if we use mode advanced or higher-dimensional node embedding techniques (again, try this as a self-study exercise).

Practice session

- we implement **logistic regression** and **gradient-based optimization** from scratch
- we use **logistic regression** for node classification in a social network
- we combine Laplacian embedding and logistic regression to implement **supervised link prediction**



09-02 - Supervised Node Classification and Link Prediction

July 6 2022

So far, we have studied link prediction as an unsupervised problem. Let us recapitulate this: We have split the network in a training and a test network, where both contained the same set of nodes. We have then considered the pairs of nodes in the test network and generated the same number of negative samples (i.e. node pairs that are not connected by a link). This yields a balanced binary classification problem, i.e. we have a set of node pairs where half are connected by a link and half are not.

We then calculated node similarities based on the edges in the training network and used them to rank node pairs. We finally used this ranking to evaluate a binary classifier in a test set that contained true positives (i.e. node pairs connected by links) and true negatives (i.e. node pairs not connected by a link). Even though we split the network into a training and a test set for the purpose of evaluation, we adopted an **unsupervised approach** that did not require training data to learn model parameters. We just used the training network to calculate node similarities and evaluated the ranking of node pairs based on the AUC of a simple classifier that uses a variable discrimination threshold.

In the previous week, we introduced graph representation learning techniques that can be used to learn "feature vectors" that best represent the nodes in a graph. We will now combine this with the logistic regression classification algorithm from the previous notebook to implement supervised node classification and link prediction.

```
import pathpy as pp
import scipy as sp
from numpy import linalg as npl
import matplotlib.pyplot as plt

import pandas as pd
from sklearn import metrics

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

import numpy as np
from scipy.linalg import svd
```

practice session

see notebook 09-01 – 09-02 in `gitlab` repository at

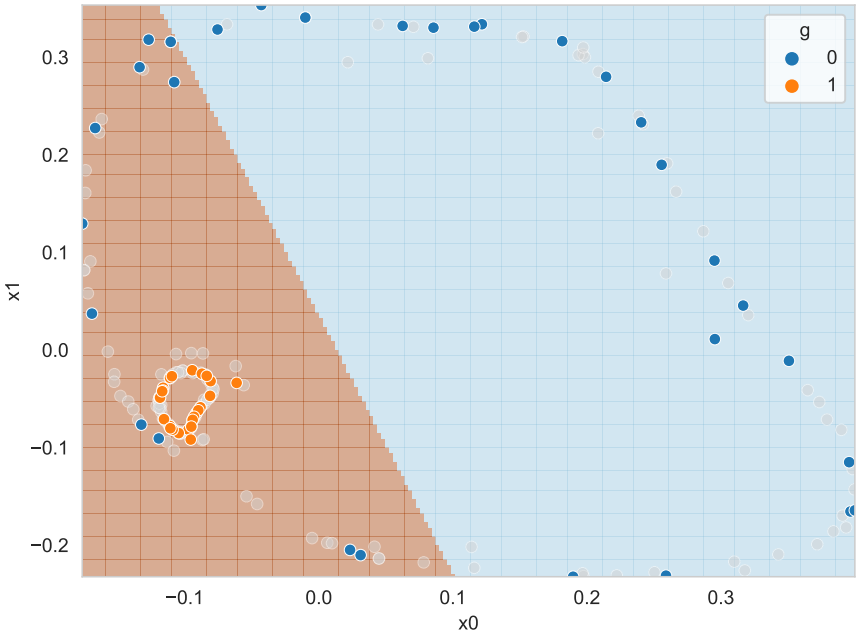
→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

Notes:

- In our first practice session, we implement a logistic regression classifier, as well as gradient-based optimization from scratch. We further apply the resulting classifier to address supervised node classification and link prediction in two empirical networks.

Graphs with non-linear patterns?

- logistic regression classifier has **linear decision boundary**
- how can we classify data with **non-linear patterns**?
- we introduce **non-linear classification with neural networks**
- basis for **walk-based embeddings** and generalization to **graph neural networks** → Lecture 10 and 11



linear **decision boundary** of logistic regression classifier in two-dimensional Laplacian feature space

Notes:

- In the final example of the practise session, we have generated a network with a complex topology that translates to a non-linear pattern in the Laplacian feature space. The linear model of the log-odds in logistic regression translates to a linear decision boundary (i.e. a point in $\mathbb{R}$, a line in $\mathbb{R}^2$, a hyperplane in $\mathbb{R}^3$, etc.) that cannot capture the pattern in the feature space.
- To address this issue, we need supervised learning techniques that are able to capture non-linear patterns, e.g., for node classification or link prediction. While we could use non-linear classification techniques like Random Forests or Support Vector Machines with non-linear kernels, here we introduce neural networks. This will allow us to address node classification in our example.
- Importantly, here we cannot broadly cover all of the interesting aspects of (deep) neural networks. We rather provide a basic introduction of key ideas, terms and concepts that are the necessary foundation to introduce walk-based embeddings and graph neural networks in the two subsequent lectures. For more details on deep neural networks, applications in computer vision and NLP and advanced optimization algorithms, we refer you to the respective specialized lectures at University of Würzburg (e.g. Programmieren mit neuronalen Netzen, Prof. Puppe, ML for Natural Language Processing, Prof. Glavas, Computer Vision, Prof. Timofte).

Neurons as binary classifier

- **neuron** is an electrochemically excitable nerve cell
- soma receives multiple **synaptic inputs** via dendrites
- electrochemical potential of dendrites and rest potential **accumulates** in soma
- if inputs exceed **threshold potential** neuron triggers **action potential** across **axon**
- simplified model of a single neuron can be seen as **binary classifier**
  1. inputs: voltage levels
  2. output: action potential

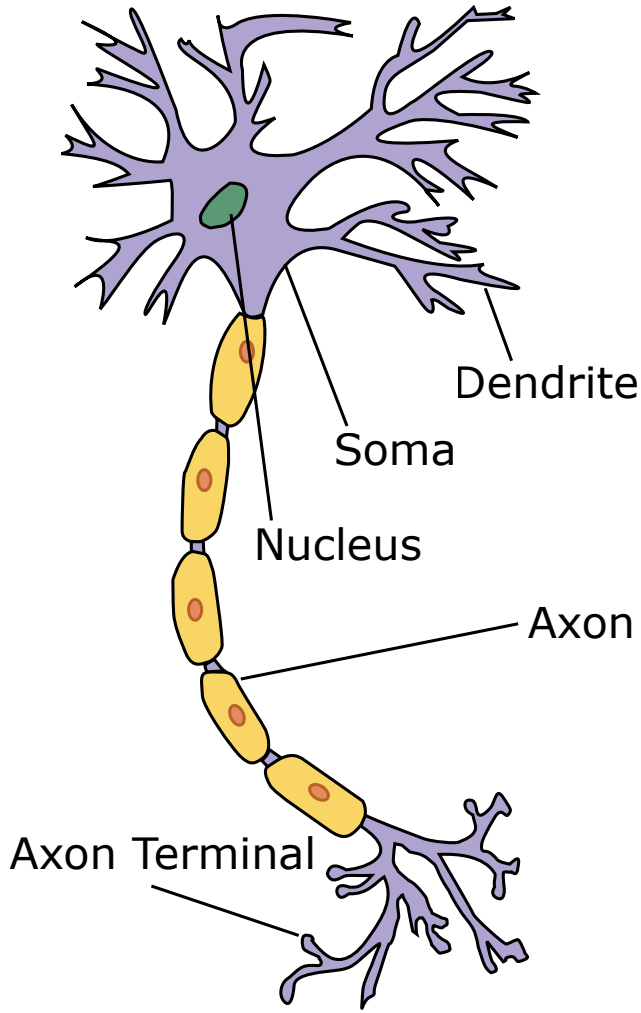


image credit: adapted from “Anatomy and Physiology”, SEER Program

Notes:

- We first introduce the basic building block of neural networks: the perceptron model, a binary classifier that is based on a (grossly simplified model) for a single neuron.
- Why can we use a model for a neuron as a binary classifier? Let us have a look at the physiology of a neuron, an electrochemically excitable nerve cell that is depicted above. It features a number of dendrites that join into the soma. These dendrites have an electrochemical potential that – together with the rest potential of the soma itself – accumulates up to a certain threshold value. Above this threshold, the neuron triggers a so-called action potential, which propagates across the axon. This axon can be close to a meter in length and it terminates in multiple axon terminals, which can connect to the dendrites of other neurons.
- We can now actually see each neuron as a basic binary classifier, which takes inputs (in the form of voltages) from the dendrites, and which generates a binary output (action potential or not) that can serve as input for other neurons.

Perceptron classifier

- **perceptron** is function y with input $\vec{x} \in \mathbb{R}^k$ and binary output $y(x) \in \{0, 1\}$ → F Rosenblatt, 1958
- **linear combination** $f : \mathbb{R}^k \to \mathbb{R}$ of inputs with bias $\beta_0 \in \mathbb{R}$ and weights $\beta_1, \ldots, \beta_k \in \mathbb{R}$, i.e.

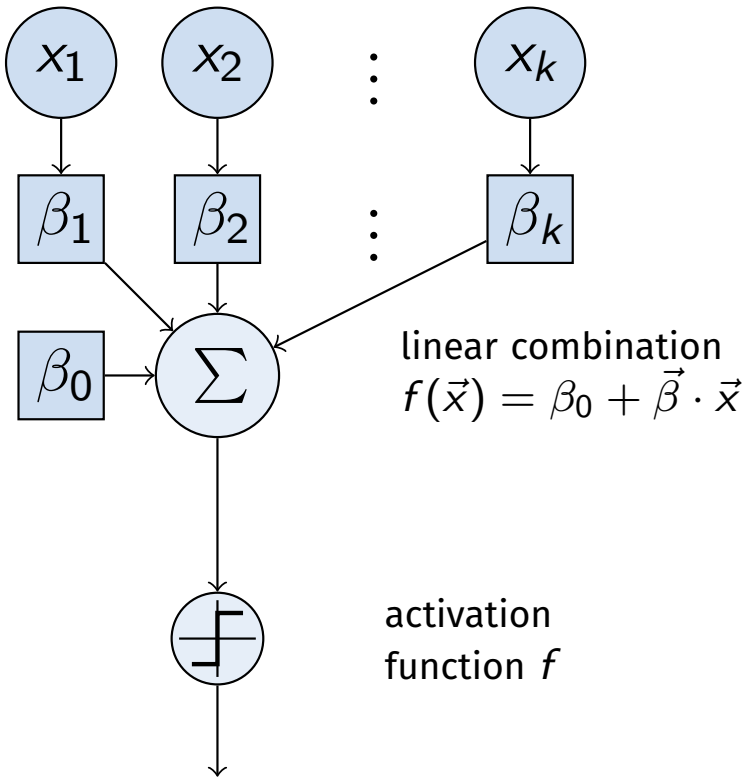
$$f(\vec{x}) := \beta_0 + \sum_{i=1}^{k} \beta_i \cdot x_i = \vec{\beta} \cdot (1, \vec{x})^T$$

with $\vec{\beta} \in \mathbb{R}^{k+1}$ and $(1, \vec{x}) := (1, x_1, \ldots, x_k)$

- application of **non-linear activation function** yields binary classifier, e.g.

$$y(\vec{x}) := \sigma(f(\vec{x})) = \frac{1}{1 + e^{-f(\vec{x})}} \in [0, 1]$$

where σ is **logistic function**



Notes:

- This can be modelled (in simplified form) as follows: We model the voltage inputs of k dendrites via real values x_i, which are multiplied by coefficients β_i. We further use a bias parameter β_0 which models the rest potential of the neuron. The accumulation of those inputs in the soma is modelled via a simple linear combination of the bias and the inputs. An additional non-linear activation function models the triggering of an action potential.
- Under certain conditions the perceptron classifier can actually be viewed as a different perspective on logistic regression. We first note that the function $f(\vec{x})$ yields a linear combination of features x_i, just like in logistic regression. We then use a **non-linear activation function** to map the function value $f(\vec{x}) \in \mathbb{R}$ to classes. We could use a simple Heaviside activation function to directly map all values $f(\vec{x}) > 0$ to class one and $f(\vec{x}) \leq 0$ to class zero. Or we can take the same approach as in logistic regression, i.e. we can use a logistic activation function to map function values $f(\vec{x}) \in \mathbb{R}$ to a continuous value range $[0, 1]$, i.e. we interpret the continuous output of the perceptron as class probability.

Gradient-based Perceptron Learning

- for $\vec{\beta} \in \mathbb{R}^{k+1}$ and training examples $(\vec{x}_s, \hat{y}_s)$ define **L2 loss function** as

$$L(\vec{\beta}) = \frac{1}{2}\sum_{s=1}^{n}(y_s - \hat{y}_s)^2 = \frac{1}{2}\sum_{s=1}^{n}(\sigma(f(\vec{x}_s)) - \hat{y}_s)^2$$

- we can move along **gradients** to find $\hat{\beta}$ that **minimizes loss function**
- for logistic function σ the contribution of training example $(\vec{x}_s, \hat{y}_s)$ to **partial derivatives of L2 loss function** is → exercise sheet

$$\frac{\partial L_s}{\partial \beta_j} = (y_s - \hat{y}_s) \cdot y_s \cdot (1 - y_s) \cdot x_{sj} \qquad (j = 1, \ldots, k)$$

$$\frac{\partial L_s}{\partial \beta_0} = (y_s - \hat{y}_s) \cdot y_s \cdot (1 - y_s)$$

perceptron learning algorithm (L2 loss, logistic activation function) → M Minsky and S Papert, 1969

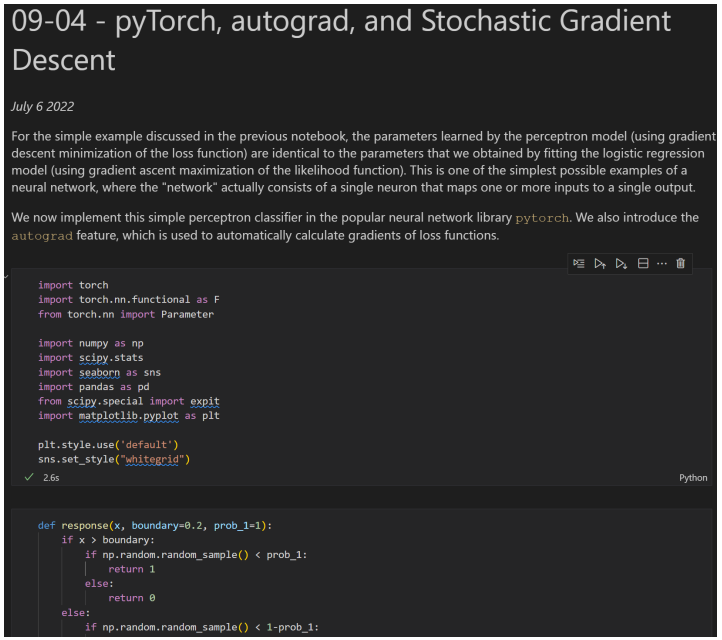
1. choose initial parameters $\beta_i = 0$ and learning rate $\eta \in [0, 1]$
2. for each $(\vec{x}_s, y_s)$ in **training batch** do:
   - $\beta_0 = \beta_0 - \eta(y_s - \hat{y}_s) \cdot y_s \cdot (1 - y_s)$
   - $\beta_j = \beta_j - \eta(y_s - \hat{y}_s) \cdot y_s \cdot (1 - y_s) \cdot x_{sj}$ for $j = 1, \ldots, k$
3. repeat 2 until $L(\vec{\beta}) \leq \epsilon$

Notes:

- Just like logistic regression, a perceptron classifier with input $\vec{x} \in \mathbb{R}^k$ has $k + 1$ parameters that we need to fit to the training data. In logistic regression we could use those parameters to calculate a probability of the data, which allows us to fit the parameters by maximizing the inverse probability, i.e. the likelihood function.
- For a perceptron model, we can define a **loss function** $L(\vec{\beta})$ that captures to what extent the outputs of our classifier match our training data. Minimizing this function to find the optimal parameters. In other words: rather than maximizing the likelihood function (which captures the plausibility of the model parameters in logistic regression given the training data), we minimize the loss function (which captures the implausibility of the model parameters of the perceptron model given the training data). In the exercise, you will see that the **perceptron and logistic regression are identical if we use a binary cross-entropy loss function and a logistic activation function**.
- Like in logistic regression, there is - in general - no analytical solution for this optimization problem. But we can take a similar approach as in logistic regression: We can use gradients of the loss function to find a parameter vector $\hat{\beta}$ that minimizes the loss function. The gradient actually depends on the loss function and the activation function. For the logistic activation function, we can calculate the partial derivatives in all directions β_i of the parameter space and obtain the expression above.
- This yields a greedy optimization algorithm that was first addressed (for a step activation function that yields a slightly different updating rule) by → M Minsky and S Papert, 1969

Practice session

- we implement **perceptron learning** from scratch in python
- we use **pytorch** to implement the perceptron model
- we apply the perceptron model to **node classification**



09-04 - pyTorch, autograd, and Stochastic Gradient Descent

July 6 2022

For the simple example discussed in the previous notebook, the parameters learned by the perceptron model (using gradient descent minimization of the loss function) are identical to the parameters that we obtained by fitting the logistic regression model (using gradient ascent maximization of the likelihood function). This is one of the simplest possible examples of a neural network, where the "network" actually consists of a single neuron that maps one or more inputs to a single output.

We now implement this simple perceptron classifier in the popular neural network library `pytorch`. We also introduce the `autograd` feature, which is used to automatically calculate gradients of loss functions.

```
import torch
import torch.nn.functional as F
from torch.nn import Parameter

import numpy as np
import scipy.stats
import seaborn as sns
import pandas as pd
from scipy.special import expit
import matplotlib.pyplot as plt

plt.style.use('default')
sns.set_style("whitegrid")
```

✓ 2.6s Python

```
def response(x, boundary=0.2, prob_1=1):
    if x > boundary:
        if np.random.random_sample() < prob_1:
            return 1
        else:
            return 0
    else:
        if np.random.random_sample() < 1-prob_1:
```

practice session

see notebook 09-03 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

Notes:

- In the second practice session, we implement the perceptron model. We further introduce `pytorch`, a popular machine learning framework that allows to efficiently implement (deep) neural networks. We show how pytorch's autograd feature allows to quickly implement gradients of loss functions and how we can use this to implement a stochastic gradient descent optimization of model parameters.

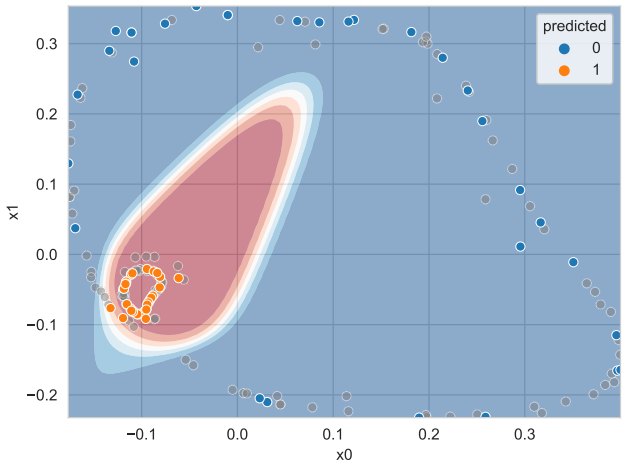
Feed-Forward Neural Networks

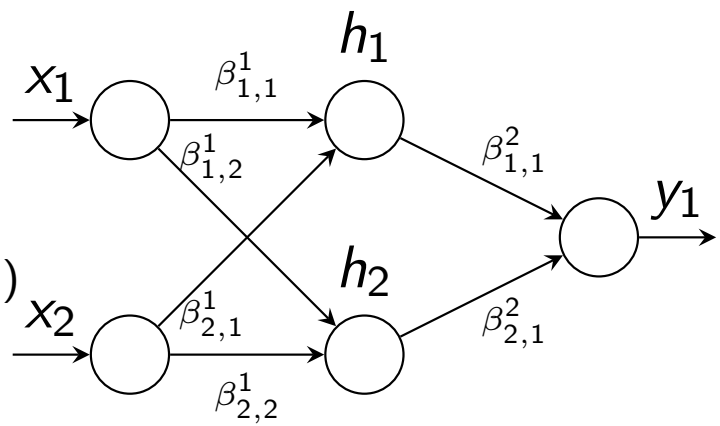
- like logistic regression, perceptron classifier has **linear decision boundary**
- idea: couple **multiple layers of perceptrons**
- neuron h_j in **hidden layer with width** d

$$h_j := h_j(\vec{x}) = \sigma\left(\beta^1_{0,j} + \sum_{i=1}^{k} \beta^1_{i,j} x_i\right) = \sigma(\vec{\beta}^1_j \cdot (1, \vec{x})^T)$$

- neuron y_i in **output layer**

$$y_i := y_i(\vec{x}) = \sigma\left(\beta^2_{0,i} + \sum_{j=1}^{d} \beta^2_{j,i} h_j(\vec{x})\right) = \sigma(\vec{\beta}^2_i \cdot (1, \vec{h})^T)$$





non-linear decision boundary of feed-forward network with one **hidden layer** with width $d = 2$

Universal Approximation Theorem

"arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single [...] hidden layer and any continuous sigmoidal nonlinearity"

→ G Cybenko, 1989

Notes:

- While the perceptron model is the fundamental building block of neural networks, it is not expressive enough to address classification in data with non-linear patterns. Due to the linear combination of input features, just like in logistic regression, we obtain a linear decision boundary. To address this limitation, we can couple multiple layers of perceptrons and build a **feed-forward neural network**. We call this feed-forward network because we can compute the outputs of the network by feeding the inputs through multiple layers of perceptrons and associated non-linear activation functions (note that there is no recurrence in the network). The inputs of the network are often called input layer (although those inputs are actually not perceptrons).
- Perceptrons that do not directly produce an output (e.g. h_0 and h_1 in the example) are called **hidden neurons**. They form the **hidden layers** of the model. Neurons in the output layer take the output of hidden neurons and generate an output (y_1 in the example above). One can show show that a neural network with two layers of perceptrons (i.e. with a single hidden layer) and a continuous non-linear activation function (such as the logistic function) can approximate arbitrary decision boundaries.
- The weight parameters β_{ij}^l can be interpreted as weights in layer l of a network of perceptrons, where β_{ij}^l denotes the weight of a directed edge that connects neuron h_i in layer $l-1$ to neuron h_j (or y_j) in layer l. We can, in principle, use different network topologies to connect those neurons. In the example we used a fully connected topology, where all neurons in layer $l-1$ are connected to all neurons in layer l. We further have two layers, i.e. each hidden neuron h_j has a weight vector β_j^1 and the single output neuron y_1 has a weight vector β_1^2. We call d the width of the hidden layer, while the number of hidden layers determines the depth of our neural network.

Gradient Optimization in Neural Networks

- for training samples $(\vec{x}_s, \hat{y}_s)$ consider **least square error (L2) loss function**

$$L(\beta^1, \beta^2) = \frac{1}{2}\sum_{s=1}^{n}(\hat{y}_s - y_s)^2 \quad \text{and} \quad L_s(\beta^1, \beta^2) = \frac{1}{2}(\hat{y}_s - y_s)^2$$

  where β^j is **weight matrix** of neurons in layer j and L_s is contribution of $\hat{y}_s$

- output y_s of feed-forward network with two layers and activation function σ is given by **composition of functions**, i.e. $y_s = \sigma(\beta^2 \cdot \sigma(\beta^1 \cdot (1, \vec{x}_s)^T))$

- for weight $\beta^2_{j,i}$ of **output neuron** y_i single application of chain rule yields

$$\frac{\partial L_s}{\partial \beta^2_{j,i}} = (\hat{y}_s - y_s)\sigma'(\underbrace{\vec{\beta}^2_i \cdot (1, \vec{h})^T}_{\text{input to } y_i})h_j(\vec{x}_s)$$

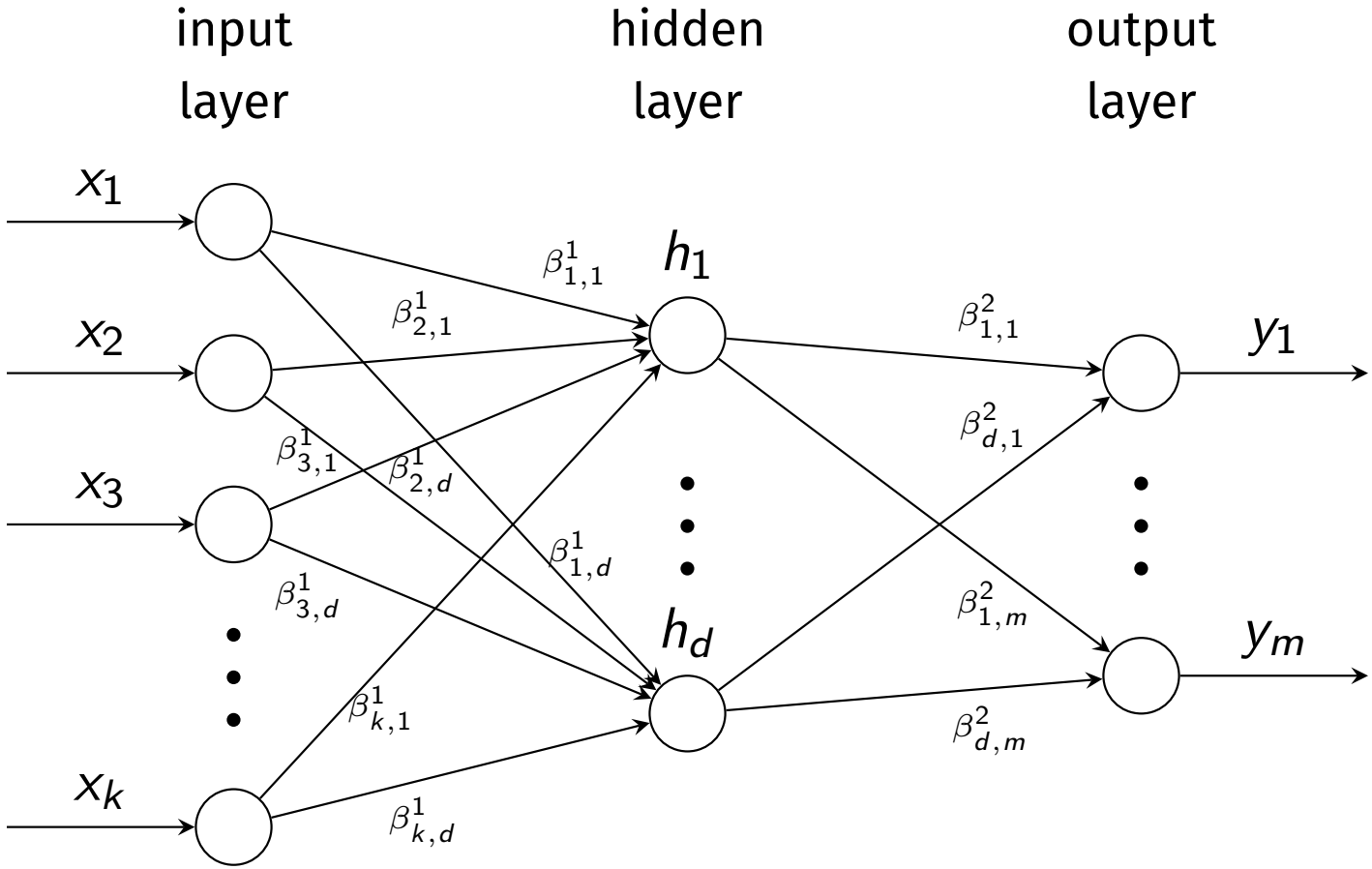
- for weight $\beta^1_{k,j}$ of **hidden neuron** h_j we apply chain rule once more and obtain

$$\frac{\partial L_s}{\partial \beta^1_{k,j}} = (\hat{y}_s - y_s)\sigma'(\underbrace{\vec{\beta}^2_i \cdot (1, \vec{h})^T}_{\text{input to } y_i})\beta^2_{i,j}\sigma'(\underbrace{\vec{\beta}^1_j \cdot (1, \vec{x}_s)^T}_{\text{input to } h_j}) \cdot x_{sk}$$

Notes:

- Given a training set, how can we find the optimal weight parameters in the different layers of a feed-forward neural network? For a perceptron model, we could easily calculate gradients based on the partial derivatives of a loss function (e.g. L2 loss). We found a simple updating rule that we can use for a gradient descent optimization. We can take a very similar approach in feed-forward networks, however we must consider the fact that the output of the neural network is generated by a composition of multiple functions that represent the layers of the network.
- Let us consider the same L2 loss function as before. Let us further consider that in each layer j of our neural network, we have weights for all directed edges that connect the output of neurons to the next layer. We can denote the weights of layer j in matrix form, i.e. $\beta^j \in \mathbb{R}^{m,k}$ where m is the number of neurons in the previous layer and d is the number of neurons in the next layer. We can now express the (possibly) vector output of a feed-forward neural network with l layers as a composition of l linear combinations and activation functions σ (assuming that the same activation function is used for all neurons).
- To compute the gradient, we calculate partial derivatives of the loss function w.r.t. to the weights. The output of the FF neural network is a composition of functions, i.e. we apply the chain rule. Different from a perceptron, which is a simple composition of a linear combination f and the activation function, we have multiple layers. For the weights of output neurons, we apply the chain rule a single time and obtain the expression for the contribution of samples s to the partial derivatives above. For the weight parameters of the first hidden layer, we apply the chain rule once more, i.e. we obtain an additional expression that we need to multiply with the partial derivatives for the weights in the output layer.

Neural networks as computation graph



- (deep) neural networks = **complex computation graphs**
- to calculate gradients of loss function w.r.t weights, we **recursively apply chain rule**, starting at outputs y_i until we reach the inputs x_i

Notes:

- This repeated application of the chain rule to the composition of functions that capture the output of the feed-forward network gives the general idea of the learning algorithm. We can actually see the neural network as a computation graph, i.e. we compute output values by feeding input values into the input nodes and calculating the composition of functions step-by-step. For a given input $\vec{x_s}$ this allows us to calculate the corresponding output y of the network. We can now use a loss function to calculate the error of the network for a ground truth value $\hat{y}$ in our training set. This error can then be used to calculate the gradients of all weight parameters, and we can calculate these gradients using the chain rule, by passing the error backwards through the neural network!

Differentiation via Backpropagation

- to calculate parameter gradients we **propagate model loss backwards** from output to input layer → DE Rumelhart et al., 1986

stochastic gradient descent optimization

1. choose initial parameters β^l_{ij} and learning rate $\eta \in [0, 1]$
2. for i in range(iterations) do:
3. batch = random subset of training examples
4. for each $(\vec{x}_s, \hat{y}_s)$ in **training batch** do:
5. update weights of output neurons y_i

$$\beta^2_{j,i} = \beta^2_{j,i} - \eta \frac{\partial L_s}{\partial \beta^2_{j,i}}(\vec{x}_s)$$

6. update weights of hidden neurons h_j

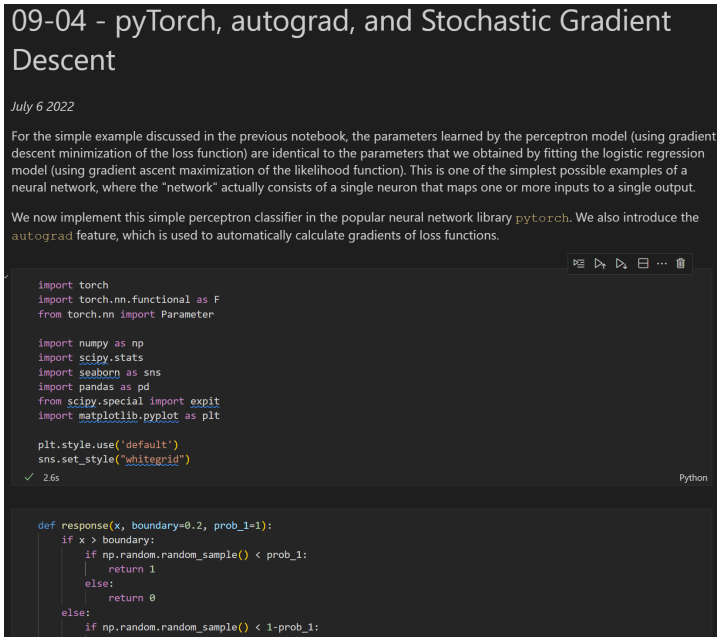
$$\beta^1_{k,j} = \beta^1_{k,j} - \eta \frac{\partial L}{\partial \beta^1_{k,i}}(\vec{x}_s)$$

Notes:

- We call this approach to automatically differentiate a loss function with respect to all model parameters **backpropagation**. We can use it to calculate gradients for all weight parameters in the output layer and all hidden layers. This then yields a modified (simple) update rule, which we can use for a gradient descent optimization of the model parameters.
- In principle, in each iteration of the gradient descent algorithm we could update the model parameters based on all examples in our training set. The stochastic version of gradient descent instead repeatedly updates model parameters based on a single randomly chosen example from the training set. We can instead use so-called batches, i.e. in each iteration we use a small set of randomly chosen training examples and update model parameters based on the examples in the batch. If the batch size is equal to the training set size, this corresponds to the standard gradient descent algorithm. However, we often have a large number of parameters and a large number of samples. In such cases, it is often more efficient to use smaller randomly chosen batches, i.e. in each iteration, we calculate the loss function for a small number of instances in a mini-batch and update the model parameters accordingly. We then repeat this for a number of iterations.

Practice session

- we introduce `pytorch` **and the autograd module** for automatic differentiation
- we implement a **multi-layer perceptron** and stochastic gradient descent optimization
- we use **neural networks** for non-linear node classification



09-04 - pyTorch, autograd, and Stochastic Gradient Descent

July 6 2022

For the simple example discussed in the previous notebook, the parameters learned by the perceptron model (using gradient descent minimization of the loss function) are identical to the parameters that we obtained by fitting the logistic regression model (using gradient ascent maximization of the likelihood function). This is one of the simplest possible examples of a neural network, where the "network" actually consists of a single neuron that maps one or more inputs to a single output.

We now implement this simple perceptron classifier in the popular neural network library `pytorch`. We also introduce the `autograd` feature, which is used to automatically calculate gradients of loss functions.

```
import torch
import torch.nn.functional as F
from torch.nn import Parameter

import numpy as np
import scipy.stats
import seaborn as sns
import pandas as pd
from scipy.special import expit
import matplotlib.pyplot as plt

plt.style.use('default')
sns.set_style("whitegrid")
```

✓ 2.6s Python

```
def response(x, boundary=0.2, prob_1=1):
    if x > boundary:
        if np.random.random_sample() < prob_1:
            return 1
        else:
            return 0
    else:
        if np.random.random_sample() < 1-prob_1:
```

practice session

see notebook 09-04 – 09-05 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

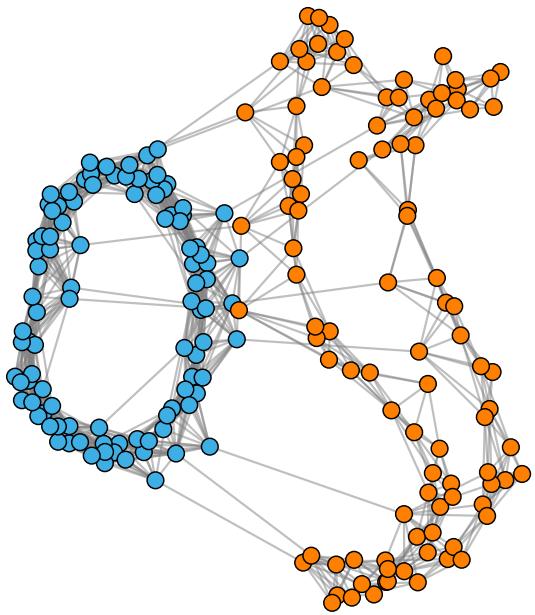
Notes:

- In the final practice session, we implement a feed-forward neural network in `pytorch` and introduce the `autograd` module, which can be used to efficiently calculate gradients of loss functions. We use a multi-layer neural network for node classification.

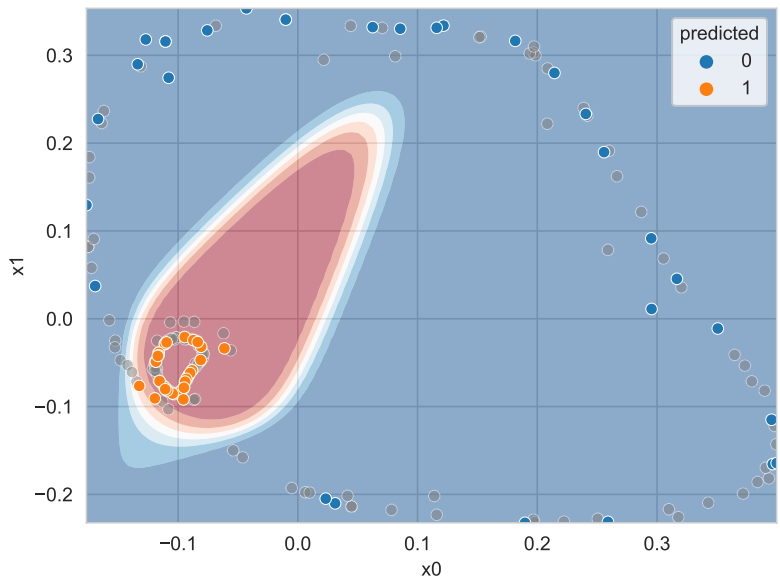
Node classification with neural networks

example

- synthetically generated network with two interconnected **ring topologies**
- ground truth node classes assigned based on rings



synthetically generated network with two interconnected ring topologies



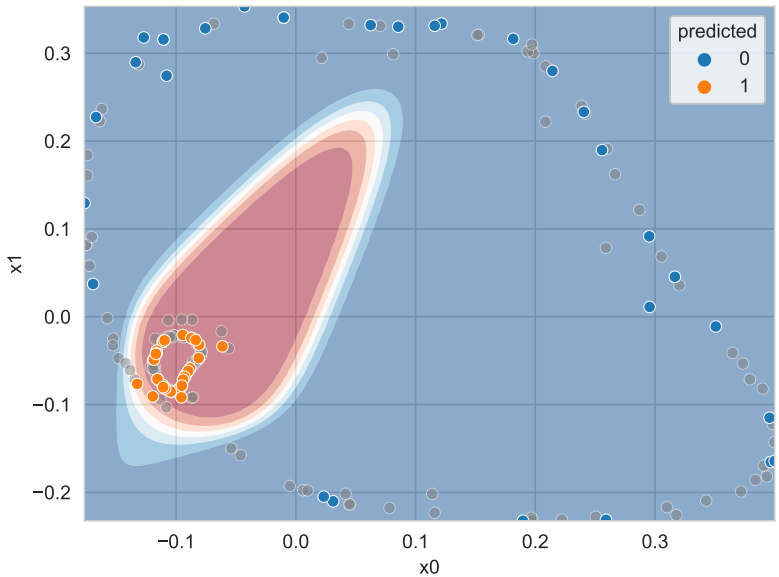
decision boundary of **feed-forward neural network** with one hidden layer in two-dimensional Laplacian node embedding

Notes:

- We can now reconsider our example for a network where node classification requires a non-linear decision boundary. In the example above, we are not able to correctly classify nodes by applying logistic regression or a simple perceptron model to a Laplacian embedding. The use of a neural network with a single hidden layer with two hidden neurons yields a non-linear decision boundary that captures the pattern in the data and thus allows to classify nodes with high accuracy.

In summary

- we introduced **supervised learning in complex networks**
- representation learning enables us to apply **standard machine learning algorithms**
- node classification: directly use **vector-space representations of nodes**
- supervised link prediction: we can use node representations to **generate vector representation of node pairs**
- **feed-forward neural networks** can learn arbitrary non-linear patterns

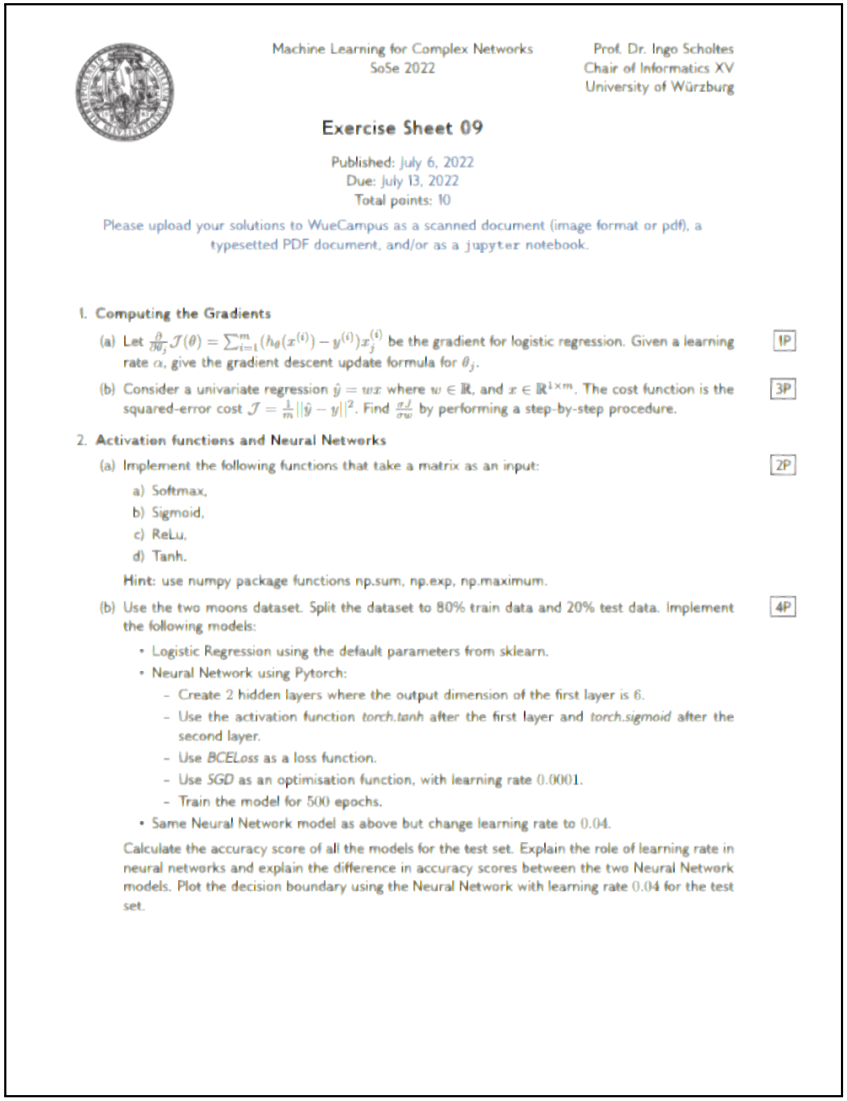


Notes:

- In summary, we introduced key supervised learning techniques and showed how we can apply them in complex networks. We specifically addressed node classification by applying logistic regression to a Euclidean representation of nodes. We further studied how we can use supervised classification algorithms to perform supervised link prediction.
- While logistic regression and the simple perceptron model yield a linear decision boundary, we have seen an example for a complex networks with a non-linear pattern. We introduced feed-forward neural networks and discussed how we can find optimal parameters by means of backpropagation and stochastic gradient descent. We then applied this approach to a complex network to address non-linear node classification.

Exercise sheet 09

- **ninth exercise sheet** will be released today
  - understand the relationship between logistic regression and perceptron classifier
  - explore backpropagation and stochastic gradient descent in feed-forward neural networks
  - explore gradient-based optimization in `python`
- solutions are due **July 3rd** (via Moodle)
- present your solution to earn bonus points



Machine Learning for Complex Networks
SoSe 2022

Prof. Dr. Ingo Scholtes
Chair of Informatics XV
University of Würzburg

Exercise Sheet 09

Published: July 6, 2022
Due: July 13, 2022
Total points: 10

Please upload your solutions to WueCampus as a scanned document (image format or pdf), a typesetted PDF document, and/or as a jupyter notebook.

1. **Computing the Gradients**

(a) Let $\frac{\partial}{\partial \theta_j}\mathcal{J}(\theta) = \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$ be the gradient for logistic regression. Given a learning rate α, give the gradient descent update formula for θ_j. 1P

(b) Consider a univariate regression $\hat{y} = wx$ where $w \in \mathbb{R}$, and $x \in \mathbb{R}^{1\times m}$. The cost function is the squared-error cost $\mathcal{J} = \frac{1}{m}||\hat{y} - y||^2$. Find $\frac{\sigma J}{\sigma w}$ by performing a step-by-step procedure. 3P

2. **Activation functions and Neural Networks**

(a) Implement the following functions that take a matrix as an input: 2P

a) Softmax,
b) Sigmoid,
c) ReLu,
d) Tanh.

Hint: use numpy package functions np.sum, np.exp, np.maximum.

(b) Use the two moons dataset. Split the dataset to 80% train data and 20% test data. Implement the following models: 4P

- Logistic Regression using the default parameters from sklearn.
- Neural Network using Pytorch:
  - Create 2 hidden layers where the output dimension of the first layer is 6.
  - Use the activation function *torch.tanh* after the first layer and *torch.sigmoid* after the second layer.
  - Use *BCELoss* as a loss function.
  - Use *SGD* as an optimisation function, with learning rate 0.0001.
  - Train the model for 500 epochs.
- Same Neural Network model as above but change learning rate to 0.04.

Calculate the accuracy score of all the models for the test set. Explain the role of learning rate in neural networks and explain the difference in accuracy scores between the two Neural Network models. Plot the decision boundary using the Neural Network with learning rate 0.04 for the test set.

Notes:

Questions

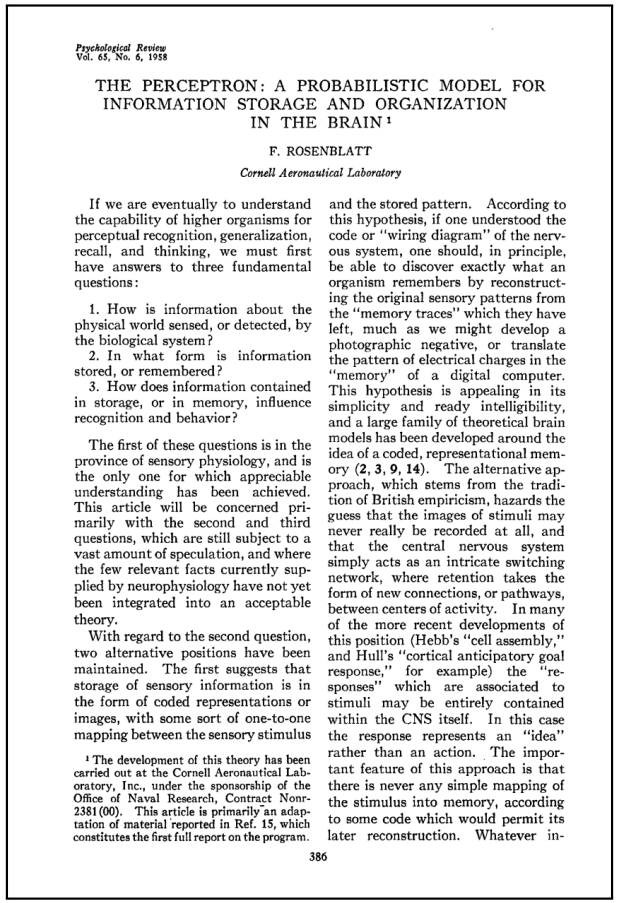
1. Explain the difference between supervised link prediction and unsupervised similarity-based link prediction.
2. Explain how the logit and the logistic function are used in logistic regression?
3. How can we estimate the parameters of a logistic regression model?
4. What is the Hadamard operator and how we can use it for link prediction?
5. What is the difference between the loss function in the perceptron classifier and the likelihood function in logistic regression?
6. How are the gradients of a perceptron affected by the choice of the loss/activation function?
7. Why is it convenient to include a factor $\frac{1}{2}$ in the L2 loss function?
8. Explain the perceptron learning algorithm. How are the gradients of the loss function calculated?
9. Calculate the update rule of the perceptron learning algorithm for a step activation function that assigns class 1 for $f(\vec{x}) > 0$ and class 0 else.
10. Explain how we can compute gradients of the loss function for feed-forward neural networks.
11. What is the difference between deterministic and stochastic gradient descent optimization?

Notes:

References

reading list

- F Rosenblatt: **The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain**, Psych. Review, 1958
- DE Rumelhart, GE Hinton, DJ Williams: **Learning representations by back-propagating errors**, Nature, 1986
- G Cybenko: **Approximation by superpositions of a sigmoidal function**, Mathematics of Control, Signals and Systems, 1989
- A Pinkus: **Approximation theory of the MLP model in neural networks**, Acta Numerica, 1999
- M Leshno et al.: **Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function**, Neural Networks, 1993
- Christopher M. Bishop: **Pattern Recognition and Machine Learning**, Springer, 2006 → Chapter 4
- J Hadamard: **Mémoire sur le problème d'analyse relatif à Véquilibre des plaques élastiques encastrées**, Mémoires présentés par divers savants éstrangers à l'Académie des Sciences de l'Institut de France, 1907
- T Parr, J Howard: **The Matrix Calculus You Need For Deep Learning**, arXiv:1802.01528, 2018



Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

If we are eventually to understand the capability of higher organisms for perceptual recognition, generalization, recall, and thinking, we must first have answers to three fundamental questions:

1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

The first of these questions is in the province of sensory physiology, and is the only one for which appreciable understanding has been achieved. This article will be concerned primarily with the second and third questions, which are still subject to a vast amount of speculation, and where the few relevant facts currently supplied by neurophysiology have not yet been integrated into an acceptable theory.

With regard to the second question, two alternative positions have been maintained. The first suggests that storage of sensory information is in the form of coded representations or images, with some sort of one-to-one mapping between the sensory stimulus and the stored pattern. According to this hypothesis, if one understood the code or "wiring diagram" of the nervous system, one should, in principle, be able to discover exactly what an organism remembers by reconstructing the original sensory patterns from the "memory traces" which they have left, much as we might develop a photographic negative, or translate the pattern of electrical charges in the "memory" of a digital computer. This hypothesis is appealing in its simplicity and ready intelligibility, and a large family of theoretical brain models has been developed around the idea of a coded, representational memory (**2**, **3**, **9**, **14**). The alternative approach, which stems from the tradition of British empiricism, hazards the guess that the images of stimuli may never really be recorded at all, and that the central nervous system simply acts as an intricate switching network, where retention takes the form of new connections, or pathways, between centers of activity. In many of the more recent developments of this position (Hebb's "cell assembly," and Hull's "cortical anticipatory goal response," for example) the "responses" which are associated to stimuli may be entirely contained within the CNS itself. In this case the response represents an "idea" rather than an action. The important feature of this approach is that there is never any simple mapping of the stimulus into memory, according to some code which would permit its later reconstruction. Whatever in-

[1] The development of this theory has been carried out at the Cornell Aeronautical Laboratory, Inc., under the sponsorship of the Office of Naval Research, Contract Nonr-2381(00). This article is primarily an adaptation of material reported in Ref. 15, which constitutes the first full report on the program.

386

Notes: