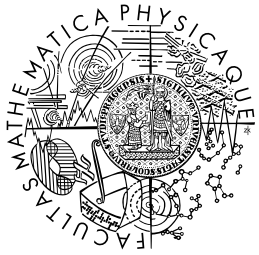

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Samuel Mešša

Machine learning with applications to finance

Department of Probability and Mathematical Statistics

Supervisor of the bachelor thesis:	doc. RNDr. Jan Hurt, CSc.
Study programme:	Mathematics
Study branch:	Financial Mathematics

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Here i would like to thank my thesis supervisor doc. RNDr. Jan Hurt, CSc. for his time, advice and guidance in the process of making this thesis.

Title: Machine learning with applications to finance

Author: Samuel Mešša

Department: Department of Probability and Mathematical Statistics

Supervisor: doc. RNDr. Jan Hurt, CSc., Department of Probability and Mathematical Statistics

Abstract: The impact of data driven, machine learning technologies across a wide variety of fields is undeniable. The financial industry, which relies heavily on predictive modeling being no exception. In this work we summarize two widely used machine learning models: support vector machines and neural networks, discuss their limitations and compare their performance to a more traditionally used method, namely logistic regression. Evaluation was done on two real world datasets, which were used to predict default of loan applicants and credit card holders formulated as a binary classification task. Neural networks and support vector machines either outperformed or showed comparable results to logistic regression with performance measured in receiver operator characteristic area under curve. In the second task neural networks outperformed both other models by a significant margin.

Keywords: machine learning classification prediction finance

Contents

Introduction

The financial industry has, for a long time, relied on garnering insight from data to make decisions. Both expert systems and more traditional statistical models were used to predict creditworthiness, trade securities and forecast macroeconomic indicators. Due to inherent limitations in these approaches, namely normality assumptions, inability to capture non-linear relationships and difficulty scaling to large datasets, novel machine learning techniques amassed great interest both in the research and industry communities. In contrast with statistical models, machine learning techniques make limited assumptions in attempting to extract predictive power from observations. The desire to automate decision making, combined with abundance of data sources and advancements in high performance computing make machine learning methods ideal candidates to substitute traditional statistical or human decision making processes.

The goal of this work is to introduce support vector machines and neural network models as forms of supervised learning as well as briefly discuss unsupervised learning techniques. To illustrate the applications of these methods in finance, we train, optimize and evaluate these models on two real world datasets concerning predicting loan and credit card default.

This works draws mainly from An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods (Cristianini and Shawe-Taylor, 2000) and Deep Learning (Goodfellow et al., 2016). In Chapter 1 we build up the theory underpinning support vector machines and neural networks. We describe the model architectures, learning process and some regularization techniques. Chapter 2 briefly discusses unsupervised learning methods. Chapter 3 compares traditional methods to machine learning techniques in binary classifications tasks and discusses their advantages and limitations in both training and hyperparameter optimization.

1. Supervised Learning

Neither supervised or unsupervised learning have a precise definition. Rather they provide a broad and informal classification of the wide range of tasks, for which machine learning algorithms are used. Before attempting to define them, we should ask what exactly we mean by machine learning. One fairly general definition by Mitchell (1997) defines learning as follows. "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". Experience E loosely defines a necessary component of every machine learning algorithm, a set of training examples. In supervised learning, training examples $\mathbf{x}$ are associated with labels or targets $\mathbf{y}$. These form a training set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$. Here N is the number of training examples. We usually represent a single example as a D-dimensional vector $\mathbf{x}_i = (x_{1,1}, x_{1,2}, \ldots, x_{1,D})$, with each entry $x_{i,j}$ of the vector being an individual feature or attribute of the particular example. In the context of finance one can imagine $\mathbf{x}$ being a collection of information about a loan applicant such as age, education, income, size of loan requested etc. In this case the label $\mathbf{y} = (y)$ will be one dimensional with values being either 1 for an applicant who successfully paid their loan or 0 if the applicant was unable to do so. The possible values of $\mathbf{y}$ narrow the range of tasks T. We discuss the two main types below.

Classification In case of one dimensional $\mathbf{y} = (y)$, with $y \in \{1, 2, \ldots, K\}$ we are dealing with a classification problem. Sometimes the algorithm is tasked to learn a mapping $f : \mathbb{R}^D \to \{1, \ldots, K\}$, which assigns an input $\mathbf{x}$ to one of the categories described by the numeric code $\{1, \ldots, K\}$. In other variants of the classification task, we try to learn a function f which outputs the entire probability distribution over the classes. In the simplest case $K = 2$ we are talking about binary classification.

Regression In regression problems, we are tasked to predict a numerical output $y \in \mathbb{R}$ given an input $\mathbf{x}$. Therefore, we typically task the algorithm to learn a mapping $f : \mathbb{R}^D \to \mathbb{R}$, which assigns an input $\mathbf{x}$ to a real valued output y.

1.1 Support Vector Machines

A support vector machine (SVM) is a supervised learning technique with applications in both classification and regression tasks. Its roots are in the Statistical Learning Theory developed by Boser et al. (1992); Cortes and Vapnik (1995). Even though SVMs can be used for both classification and regression, in this text we will focus on classification only. We start by describing a simple binary classifier technique called the maximal margin classifier, which requires a linearly separable boundary between classes. We extend this concept to problems without this property using a soft margin classifier. Finally, we introduce support vector machines as a way to capture non-linearity in class boundaries.

1.1.1 Maximum Margin Classifier

As discussed above the maximum margin classifier has a strict requirement on the training data, namely linear separability. In order to define what it means we first introduce the concept of a hyperplane.

Definition 1. *Let $\boldsymbol{a} \in \mathbb{R}^D$, $\boldsymbol{a} \neq \boldsymbol{0}$, $b \in \mathbb{R}$ and $\boldsymbol{x} \in \mathbb{R}^D$. The set $\{\boldsymbol{x} \mid \boldsymbol{a}^T\boldsymbol{x} = b\}$ is called a hyperplane.*

Definition 2. *Let $\boldsymbol{a} \in \mathbb{R}^D$, $b \in R$ and $\mathcal{D}_{train} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, with target labels $y \in \{1, -1\}$ and feature vectors $\boldsymbol{x}_i \in \mathbb{R}^D$ be a training set. $\mathcal{D}_{train}$ is called linearly separable with respect to a hyperplane $\{\boldsymbol{x} \mid \boldsymbol{a}^T\boldsymbol{x} = b\}$, if the following holds for $i = 1, 2, \ldots, N$.*

$$b + a_1x_{i1} + a_2x_{i2} + \ldots + a_Dx_{iD} > 0 \text{ iff } y_i = +1,$$

$$b + a_1x_{i1} + a_2x_{i2} + \ldots + a_Dx_{iD} < 0 \text{ iff } y_i = -1$$

or equivalently

$$y_i(b + a_1x_{i1} + a_2x_{i2} + \ldots + a_Dx_{iD}) > 0.$$

Given linear separability a simple classifier can be created. We find a separating hyperplane defined by the vector $\mathbf{h} = (b, a_1, \ldots, a_D)^T$ and classify any new example $\mathbf{x}_{new}$ as

$$y_{new} = \begin{cases} +1, & \text{if } \mathbf{x}_{new}^T\mathbf{a} + b > 0 \\ -1, & \text{if } \mathbf{x}_{new}^T\mathbf{a} + b < 0. \end{cases}$$

Since our data is linearly separable there exists an infinite number of hyperplanes $\mathbf{h}$ we could create. We would like our hyperplane to be "optimum" in some sense. An intuitive solution to this problem is creating a hyperplane, which creates the largest possible "gap" or margin between the classes. We define the notion of margin more formally.

Definition 3. *The functional margin of a labeled example $\{\boldsymbol{x}_i, y_i\}$ with respect to a hyperplane $\boldsymbol{h} = (b, a_1, \ldots, a_D)^T$ is defined as*

$$\gamma_i = y_i(b + a_1x_{i1} + a_2x_{i2} + \ldots + a_Dx_{iD}).$$

Definition 4. *The geometric margin of a labeled example $\{\boldsymbol{x}_i, y_i\}$ with respect to a hyperplane $\boldsymbol{h} = (b, a_1, \ldots, a_D)^T$ is defined as*

$$\overline{\gamma}_i = \frac{\gamma_i}{\sqrt{a_1^2 + a_2^2 + \cdots + a_D^2}} = \frac{\gamma_i}{\|\mathbf{a}\|},$$

where γ_i is the functional margin of $\{\boldsymbol{x}_i, y_i\}$ with respect to $\boldsymbol{h}$.

The geometric margin represents the Euclidean distance of the labeled example from the hyperplane. The functional margin is just an unscaled geometric margin. We should note that a positive margin implies correct classification of the example $\{\mathbf{x}_i, y_i\}$. We generalize the concept of a margin with respect to one labeled example to the entire training set.

Definition 5. *We define the functional margin of a training set $\mathcal{D}_{train}$ with respect to the hyperplane $\boldsymbol{h}$ as*

$$M(\boldsymbol{h}) = \min_{i \in \{1,2,\ldots,n\}} \gamma_i,$$

where γ_i is the functional margin of the labeled example $\{\boldsymbol{x}_i, y_i\}$ with respect to $\boldsymbol{h}$.

Definition 6. *The geometric margin of a training set $\mathcal{D}_{train}$ with respect to the hyperplane $\boldsymbol{h}$ is defined as*

$$\overline{M}(\boldsymbol{h}) = \min_{i \in \{1,2,\ldots,n\}} \overline{\gamma}_i = \frac{M(\boldsymbol{h})}{\|\mathbf{a}\|},$$

where $\overline{\gamma}_i$ is the geometric margin of the labeled example $\{\boldsymbol{x}_i, y_i\}$ with respect to $\boldsymbol{h}$.

Now we can formulate the optimization problem that is to be solved in order to find a hyperplane which maximizes the margin across the entire training set:

$$\begin{aligned} \underset{\mathbf{h}}{\text{maximize}} \quad & \overline{M}(\mathbf{h}) && (1.1)\\ \text{subject to} \quad & \|\mathbf{a}\| = 1 && (1.2)\\ & \gamma_i \geq \overline{M}(\mathbf{h}), \quad i = 1, 2, \ldots, N. && (1.3) \end{aligned}$$

We can understand the optimization problem as finding a hyperplane which maximizes the margin of the training set under the constraints 1.3. These constraints make sure that all points are correctly classified and their Euclidean distance from the hyperplane is at least the margin, which we are trying to maximize. Since $M(\mathbf{h}) = \overline{M}(\mathbf{h}) / \|\mathbf{a}\|$, the constraint 1.2 can be removed by altering the problem:

$$\begin{aligned} \underset{\mathbf{h}}{\text{maximize}} \quad & \frac{M(\mathbf{h})}{\|\mathbf{a}\|} && (1.4)\\ \text{subject to} \quad & \gamma_i \geq M(\mathbf{h}), \quad i = 1, 2, \ldots, N. && (1.5) \end{aligned}$$

The definition of the hyperplane allows it to be scaled by an arbitrary constant $k \neq 0$. More specifically, a hyperplane $\mathbf{h}$ is the same hyperplane as the one defined by $\mathbf{h}/k$. We use this property to further simplify the problem by conditioning $M(\mathbf{h}) = 1$, or equivalently multiplying $\mathbf{h}$ by $1/M(\mathbf{h})$, which is a constant given by hyperplane $\mathbf{h}$ and training set $\mathcal{D}$. Furthermore, maximizing $M(\mathbf{h}) / \|\mathbf{a}\| = 1 / \|\mathbf{a}\|$ is equivalent to minimizing $\frac{1}{2} \|\mathbf{a}\|^2$. The following optimization problem has a quadratic objective function and linear constraints:

$$\begin{aligned} \underset{\mathbf{a}}{\text{minimize}} \quad & \frac{1}{2} \|\mathbf{a}\|^2 && (1.6)\\ \text{subject to} \quad & \gamma_i \geq 1, \quad i = 1, 2, \ldots, N.^{1} && (1.7) \end{aligned}$$

Next we use the Lagrange dual formulation of the problem. This will be useful in introducing the notion of kernels. We formulate the constraints 1.7 as

$$h_i(\mathbf{a}) := 1 - \gamma_i = 1 - y_i(\mathbf{a}^T \mathbf{x}_i + b) \leq 0, \quad i = 1, 2, \ldots, N.$$

[1] This is known as a problem of quadratic programming

The Lagrangian for the optimization 1.6 problem takes the form

$$\mathcal{L}(\mathbf{a}, b, \boldsymbol{\alpha}) = \frac{1}{2} \|\mathbf{a}\|^2 - \sum_{i=1}^{N} \alpha_i [y_i(\mathbf{a}^T \mathbf{x}_i + b) - 1]. \tag{1.8}$$

Let $\mathcal{P}(\mathbf{a})$ be the objective function of the primal problem and $\mathcal{D}(\boldsymbol{\alpha})$ be the objective function of the dual problem. The objective 1.6 and the constraints 1.7 are convex in $\mathbf{a}$. Due to the conditions of linear separability, we know that the set $\{\mathbf{a} : h_i(\mathbf{a}), \forall i \in 1, 2, \ldots, n\} \neq \emptyset$, i.e., the constraints are feasible. Under these assumptions, there exists a solution $(\mathbf{a}^*, b^*, \boldsymbol{\alpha}^*)$, where $\mathbf{a}^*$ is the solution to $\mathcal{P}(\mathbf{a})$ and $\boldsymbol{\alpha}^*$ is the solution to $\mathcal{D}(\boldsymbol{\alpha})$. Additionally, the following holds: $\mathcal{P}(\mathbf{a}^*) = \mathcal{L}(\mathbf{a}^*, b^*, \boldsymbol{\alpha}^*) = \mathcal{D}(\boldsymbol{\alpha}^*)$. Furthermore, the solution $(\mathbf{a}^*, b^*, \boldsymbol{\alpha}^*)$ satisfies the Karush-Kuhn-Tucker (KKT) conditions (Cristianini and Shawe-Taylor, 2000, chapter 7.1):

$$\nabla_{\mathbf{a}} \mathcal{L}(\mathbf{a}^*, b^*, \boldsymbol{\alpha}^*) = \mathbf{0} \tag{1.9}$$

$$\frac{\partial \mathcal{L}(\mathbf{a}^*, b^*, \boldsymbol{\alpha}^*)}{\partial b} = 0 \tag{1.10}$$

$$\alpha_i^* h_i(\mathbf{a}^*) = 0, \quad i = 1, 2, \ldots, N \tag{1.11}$$

$$h_i(\mathbf{a}^*) \leq 0, \quad i = 1, 2, \ldots, N \tag{1.12}$$

$$\alpha_i^* \geq 0, \quad i = 1, 2, \ldots, N \tag{1.13}$$

In order to find the dual form, we set the derivatives of $\mathcal{L}(\mathbf{a}, b, \boldsymbol{\alpha})$ with respect to $\mathbf{a}$ and b equal to 0.

$$\nabla_{\mathbf{a}} \mathcal{L}(\mathbf{a}, b, \boldsymbol{\alpha}) = \mathbf{a} - \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i = \mathbf{0} \tag{1.14}$$

which implies that

$$\mathbf{a} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i. \tag{1.15}$$

For b we have

$$\frac{\partial \mathcal{L}(\mathbf{a}, b, \boldsymbol{\alpha})}{\partial b} = \sum_{i=1}^{N} \alpha_i y_i = 0. \tag{1.16}$$

Next we use 1.15 in the definition of the Lagrangian 1.8 and simplify it to get the following expression:

$$\mathcal{L}(\mathbf{a}, b, \boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j - b \sum_{i=1}^{N} \alpha_i y_i. \tag{1.17}$$

From 1.16 we see that the last term in the previous equation must be 0. Now, using the above, we formulate the dual optimization problem:

$$\underset{\boldsymbol{\alpha}}{\text{maximize}} \quad \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \tag{1.18}$$

$$\text{subject to} \quad \alpha_i \geq 0, \quad i = 1, 2, \ldots, N \tag{1.19}$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0. \tag{1.20}$$

Solving the dual problem allows us to find the optimum $\mathbf{a}^*$ using equation 1.15. In order to find the optimum b^*, we note that the functional margin of the closest examples for both classes must be equal and therefore

$$b^* = -\frac{\max\limits_{i:y_i=-1} \mathbf{a}^* \cdot \mathbf{x}_i + \min\limits_{i:y_i=+1} \mathbf{a}^* \cdot \mathbf{x}_i}{2} \tag{1.21}$$

1.1.2 Soft Margin Classifier

Since real-world training sets might not be linearly separable, we relax our requirements. We allow points to be within the margin, or even on the wrong side of the hyperplane. Thus we are making an implicit trade-off, we accept some misclassifications in order to get a better classification on the vast majority of examples and therefore a more robust classifier in general. We do this by adding n non-negative variables $\epsilon_1, \epsilon_2, \ldots, \epsilon_N$. They will control the extent to which the i-th example can violate the margin. We formalize this concept in the following optimization problem.

$$\underset{\mathbf{a}}{\text{minimize}} \quad \frac{1}{2} \|\mathbf{a}\|^2 + C \sum_{i=1}^{N} \epsilon_i \tag{1.22}$$

$$\text{subject to} \quad \gamma_i \geq 1 - \epsilon_i, \quad i = 1, 2, \ldots, N \tag{1.23}$$

$$\epsilon_i \geq 0, \quad i = 1, 2, \ldots, N. \tag{1.24}$$

C is referred to as a penalization constant. The above optimization problem can modified and expressed in its dual form the same way as we did in the previous section to get

$$\underset{\boldsymbol{\alpha}}{\text{maximize}} \quad \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \tag{1.25}$$

$$\text{subject to} \quad 0 \leq \alpha_i \leq C, \quad i = 1, 2, \ldots, N \tag{1.26}$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0. \tag{1.27}$$

Added variables $\epsilon_1, \epsilon_2, \ldots, \epsilon_N$ are often called *slack variables*. As mentioned earlier, they define a way to measure the individual violation of the i-th example. If $\epsilon_i = 0$, then the i-th example is correctly classified and does not violate the

margin. In the case of $\epsilon_i \in (0, 1)$ the example violates the margin but is still classified correctly. $\epsilon_i > 1$ represents a misclassification. Earlier we mentioned an implicit trade-off in allowing misclassifications to increase our ability to generalize. This trade-off is made explicit by the hyperparameter C. It determines the severity of penalizing the violations. If C is close to 0, our margin will be wide as the severity of margin violation penalization is low. Conversely, if C is large, the margin will be narrow as we impose a high penalty for observations which either violate the margin or are misclassified. In practice C is tuned on a validation set or using other techniques such as cross-validation.

1.1.3 Kernels

In practice we often encounter classification problems which are not susceptible to being modeled by a linear boundary. We could simply map our feature space to a new one by applying a function $\varphi : \mathbb{R}^D \to R^P$ to each observation in our training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ to get $\varphi(\mathcal{D}) = \{(\varphi(\mathbf{x}_i), y_i)\}_{i=1}^N$. However, this approach would be computationally expensive. Since the concepts described above may be used for the transformed feature space $\varphi(\mathcal{D})$, we can utilize the insights gained in formulating the training process in terms of its dual. Since both the optimization task and predicting new examples can be expressed in terms of the dot product of $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$, then simply replacing the dot products with $\langle \varphi(\mathbf{x}_i), \varphi(\mathbf{x}_j) \rangle$ would be equivalent to training the classifier in the new feature space $\varphi(\mathcal{D}) = \{(\varphi(\mathbf{x}_i), y_i)\}_{i=1}^N$. We could further simplify this task by finding a function $K(x_i, x_j) = \langle \varphi(\mathbf{x}_i), \varphi(\mathbf{x}_j) \rangle$, which will enable us to compute the inner product in the new feature space directly. We define the notion of this function K more formally.

Definition 7. *Let $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^N$ be an unlabeled training set with $\boldsymbol{x}_i \in \mathbb{R}^D$ and $\varphi : R^D \to F$ be a feature mapping. We define a kernel as a function K such that, $K(x_i, x_j) = \langle \varphi(\boldsymbol{x}_i), \varphi(\boldsymbol{x}_j) \rangle$, for all $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{D}$.*

Finding a kernel function can be done by first finding a feature mapping φ, its inner product and finally finding a way to compute it using the original features. In practice the kernel function is often defined directly and thus the new feature space is defined implicitly. For details on the necessary and sufficient conditions for kernel to be valid as well as commonly used kernels see Cristianini and Shawe-Taylor (2000). We now formulate the final version of our optimization task, using the dual form, kernel function and a regularization parameter C.

$$\underset{\boldsymbol{\alpha}}{\text{maximize}} \quad \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{1.28}$$

$$\text{subject to} \quad 0 \leq \alpha_i \leq C, \quad i = 1, 2, \ldots, N \tag{1.29}$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0. \tag{1.30}$$

After training on $\mathcal{D}$ and thus finding $(\mathbf{a}^*, b^*, \boldsymbol{\alpha}^*)$ we classify a new example $\mathbf{x}_{new}$

as

$$y_{new} = \text{sgn}\left(\sum_{i=1}^{N} \alpha_i^* y_i K(\mathbf{x}_i, \mathbf{x}_{new}) + b^*\right)$$

Using kernels we both efficiently train the model and classify new examples in high dimensional feature space. Additionally we gain the ability to model non-linear relationships in our features.

1.2 Artificial Neural Networks

The field of research regarding Neural Networks has its roots in attempts to model how biological systems process information in works like Rosenblatt (1962); McCulloch and Pitts (1988). Often neural networks are introduced using this biological perspective. While it is certainly a useful one to have, in the context of statistical learning it imposes some unnecessary limitations. The objective of a neural network in most settings is not to perfectly model the brain. Therefore, we will focus on neural networks as a useful model for learning from data. Neural networks have been used for a variety of problems both in the supervised and unsupervised learning domains. In this section we will focus on the most widely used type of neural network.

1.2.1 Feedforward Neural Networks

Feedforward Neural Networks, also called multilayer perceptrons (MLPs) are the most widely used neural network architecture in supervised learning. As in any type of supervised learning task our goal is to find a function $f^*(\mathbf{x}, \boldsymbol{\theta})$ which maps our feature space to an output $\mathbf{y}$. In the learning process the network learns the parameters or weights $\boldsymbol{\theta}$, so that the performance of the network increases with respect to some performance measure P. Feedforward neural networks compose many different functions together.In doing so they create an acyclic directed graph, which describes how the functions are composed together. The depth of the model L is given by the number of layers the model has. The L-th layer, is called the output layer. The layers $1, 2, \ldots, L-1$ are known as hidden, because the desired output of these is not in the training data. We already briefly discussed the neurological inspiration for these models. This inspiration shows up once more, in the way the individual layers are composed of multiple units which loosely resemble neurons in the brain working in parallel. The number of these units in the individual layers determines the width of the model. To describe this model more formally, we first introduce previously unused notation to describe the various parts of the network.

D_x the number of features of one example, i.e., the input size

D_y the output size

$D_h^{(\ell)}$ the number of hidden units in the ℓ-th layer

L the number of layers in the network

$\mathbf{b}^{(\ell)}$ the bias vector in the ℓ-th layer

$g^{(\ell)}$ the activation function used in the ℓ-th layer

$\mathbf{X} \in \mathbb{R}^{D \times N}$ the input matrix, with the i-th example being the i-th column

$\mathbf{W}^{(l)} \in \mathbb{R}^{D_h^{(l)} \times D_h^{(l-1)}}$ the weight matrix of the ℓ-th layer

$\mathbf{A}^{(l)} = g^{(l)}(\mathbf{W}^{(l)}\mathbf{A}^{(l-1)} + \mathbf{b}^{(l)}) \in \mathbb{R}^{D_h^{(l)} \times N}$ the activation of the ℓ-th layer

As the notation suggests the networks usually processes multiple examples or even the entire training set at the same time. We can define the function which

processes an arbitrary amount of examples as

$$f^*(\mathbf{X}, \boldsymbol{\theta}) = f^*(\mathbf{X}, \mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}) =$$
$$= g^{(L)}(\mathbf{W}^{(L)} g^{(L-1)}(\mathbf{W}^{(L-1)} \dots g^{(1)}(\mathbf{W}^{(1)} X + \mathbf{b}^{(1)}) + \mathbf{b}^{(L-1)}) + \mathbf{b}^{(L)}).$$

Cost Functions

The performance metric mentioned above is called a cost or a loss function in the context of neural networks. Most basic cost functions of modern neural networks are designed using the principle of maximum likelihood. Usually, our model defines a distribution with a probability density function $p(\mathbf{y}|\mathbf{x}; \boldsymbol{\theta})$. As we assume that our labeled examples are i.i.d. we therefore find our estimate of $\boldsymbol{\theta}$ by maximizing the log-likelihood, or equivalently minimizing the negative log-likelihood:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i; \boldsymbol{\theta}).$$

The form of the cost function depends on the distribution defined by $\log p_{model}$. This is closely associated with the choice of output layer of our network. We show how we derive the specific cost functions for a few widely used models.
If we set the distribution defined by $p_{model}(y|\mathbf{x})$ to be $\mathrm{N}(f^*(\mathbf{x}, \boldsymbol{\theta}), \sigma^2)$ for a given fixed σ^2, the resulting loss function takes the form:

$$\arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i; \boldsymbol{\theta}) =$$
$$\arg\min_{\boldsymbol{\theta}} -\sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i; \boldsymbol{\theta}) =$$
$$\arg\min_{\boldsymbol{\theta}} -\sum_{i=1}^{N} \log \left(\sqrt{\frac{1}{2\pi\sigma^2}} \exp -\frac{(y_i - f^*(\mathbf{x}_i; \boldsymbol{\theta}))^2}{2\sigma^2} \right) =$$
$$\arg\min_{\boldsymbol{\theta}} N \log \sigma + \frac{N}{2} \log 2\pi + \sum_{i=1}^{N} \frac{(y_i - f^*(\mathbf{x}_i; \boldsymbol{\theta}))^2}{2\sigma^2} =$$
$$\arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \frac{(y_i - f^*(\mathbf{x}_i; \boldsymbol{\theta}))^2}{2\sigma^2}.$$

Now comparing the last sum above with mean squared error (MSE) $MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - f^*(\mathbf{x}_i; \boldsymbol{\theta}))^2$, we immediately see that although they result in different values, minimizing both with respect to $\boldsymbol{\theta}$ leads to the same estimate. The above justifies its use as a cost function derived using maximum likelihood estimation.
Next we will define a cost function known as cross entropy which is used in binary classification problems. Let $z = \mathbf{w}^T \mathbf{h} + b$ be the output of the last layer without an activation function for our network. If we set the distribution defined by $p_{model}(y|\mathbf{x})$ to be $\mathrm{Alt}(\mathrm{sigm}(z), \boldsymbol{\theta})$, where $\mathrm{sigm}(x) = 1/(1 + e^{-x})$ is the logistic

sigmoid function, we can derive our cost function:

$$\begin{aligned}
&\arg\min_{\boldsymbol{\theta}} \; -\sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i;\boldsymbol{\theta}) = \\
&\arg\min_{\boldsymbol{\theta}} \; -\sum_{i=1}^{N} \log \left[sigm(z_i)^{I(y_i=1)}(1-sigm(z_i))^{I(y_i=0)} \right] = \\
&\arg\min_{\boldsymbol{\theta}} \; -\sum_{i=1}^{N} y_i \log\left[sigm(z_i)\right] + (1-y_i)\log\left[1-sigm(z_i)\right].
\end{aligned}$$

Learning Process

We have defined the basic architecture of the network as well as how cost functions are usually derived. Now we will describe the process by which the network learns. Most modern neural networks are trained using some variation of stochastic gradient descent (SGD). SGD and its variations require computing the gradient of the cost function at some point $\mathbf{x}$ with respect to the parameters of our model $\boldsymbol{\theta}$. The method for computing the gradient is called back-propagation. Evaluation the cost function at some point $\mathbf{x}$ is done through a process called forward-propagation. We describe both.

Algorithm 1 Forward propagation through a typical deep neural network and the computation of the cost function. The cost $J(\hat{\mathbf{y}}, \mathbf{y})$ depends on the output $\hat{\mathbf{y}}$ and on the target $\mathbf{y}$. For simplicity, the algorithm uses only a single input example $\mathbf{x}$.

Require: Network Depth L
Require: Weight matrices of the model $\mathbf{W}^{(i)}, \quad i = 1, 2, \ldots, L$
Require: The bias parameters of the model $\mathbf{b}^{(i)}, \quad i = 1, 2, \ldots, L$
Require: the example input $\mathbf{x}$
Require: the target output $\mathbf{y}$
  $\mathbf{h}^{(0)} = \mathbf{x}$
  for $k = 1, 2, \ldots, L$ **do**
    $\mathbf{z}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}$
    $\mathbf{h}^{(k)} = g^{(k)}(\mathbf{z}^{(k)})$
  end for
  $\hat{\mathbf{y}} = \mathbf{h}^{(L)}$
  Cost $= J(\hat{\mathbf{y}}, \mathbf{y})$
  return $\hat{\mathbf{y}}$, Cost

Backpropagation is a simple extension of the chain rule of calculus. Originally introduced in the 1970s, its full potential for training networks was shown by E. Rumelhart et al. (1986). In combination with stochastic gradient descent and its modifications, backpropagation forms the basis for modern neural network learning. To simplify the notation, we define the derivative of the cost function with respect to a non-activated j-th neuron in the ℓ-th layer $z_j^{(\ell)}$ as $\delta_j^{(\ell)} = \frac{\partial L}{\partial z_j^{(\ell)}}$. Ultimately, we would like to get the partial derivatives with respect to the weights and biases in the respective neurons and layers. Using the chain rule we derive

the following:

$$\delta_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}} = \frac{\partial J}{\partial g_j^{(L)}} \frac{\partial g_j^{(L)}}{\partial z_j^{(L)}} \tag{1.31}$$

$$\delta_j^{(\ell)} = \frac{\partial J}{\partial z_j^{(\ell)}} = \sum_{k=1}^{D_h^{(\ell+1)}} \frac{\partial J}{\partial z_k^{(\ell+1)}} \frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}} = \sum_{k=1}^{D_h^{(\ell+1)}} \delta_k^{(\ell+1)} \frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}}. \tag{1.32}$$

Furthermore,

$$z_k^{(\ell+1)} = \sum_{i=1}^{D_h^{(\ell)}} w_{ki}^{(\ell+1)} g^{(\ell)}(z_j^{(\ell)}) + b_k^{(\ell+1)}$$

$$\frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}} = w_{kj}^{(\ell+1)} \delta_k^{(\ell+1)} g^{(\ell)\prime}(z_j^{(\ell)}). \tag{1.33}$$

If we substitute 1.33 into 1.32 we get:

$$\delta_j^{(\ell)} = \sum_{k=1}^{D_h^{(\ell+1)}} w_{kj}^{(\ell+1)} \delta_k^{(\ell+1)} g^{(\ell)\prime}(z_j^{(\ell)}). \tag{1.34}$$

Finally, the derivatives of the cost function with respect to the weights and biases are:

$$\frac{\partial J}{\partial b_j^{(\ell)}} = \frac{\partial J}{\partial z_j^{(\ell)}} \frac{\partial z_j^{(\ell)}}{\partial b_j^{(\ell)}} = \delta_j^{(\ell)} \tag{1.35}$$

$$\frac{\partial J}{\partial w_{jk}^{(\ell)}} = \frac{\partial J}{\partial z_j^{(\ell)}} \frac{\partial z_j^{(\ell)}}{\partial w_{jk}^{(\ell)}} = \delta_j^{(\ell)} g^{(\ell-1)}(z_k^{(\ell-1)}) \tag{1.36}$$

Equation 1.31 shows us how to calculate the derivative at the output layer L and 1.34, shows how to express the derivative of the ℓ-th in terms the derivatives of the layer $\ell+1$. Using the above we can backpropagate to get the derivatives with respect to our weight and bias parameters. Next we formulate the backpropagation algorithm.

Algorithm 2 Backward propagation algorithm for a deep neural network. This computation yields the derivatives of the weights and biases for each layer k, starting from the output layer and going backwards to the first hidden layer.

Require: Network Depth L
Require: Weight matrices of the model $\mathbf{W}^{(i)}, \quad i = 1, 2, \ldots, L$
Require: The bias parameters of the model $\mathbf{b}^{(i)}, \quad i = 1, 2, \ldots, L$
Require: the example input $\mathbf{x}$
Require: the target output $\mathbf{y}$
  Compute the forward pass for the input $\mathbf{x}$
  Compute the derivatives without activation functions on the last layer
  for $j = 1, \ldots, D_h^{(L)}$ **do**
    $\delta_j^{(L)} = \frac{\partial J}{\partial g_j^{(L)}} g^{(L)\prime}(z_j^L)$
  end for
  Compute the derivative of the loss function with respect to all weights and biases in all layers of the network
  for $\ell = L-1, L-2, \ldots, 1$ **do**
    for $j = 1, 2, \ldots, D_h^{(\ell)}$ **do**
      $\delta_j^{(\ell)} = \sum_{k=1}^{D_h^{(\ell+1)}} w_{kj}^{(\ell+1)} \delta_k^{(\ell+1)} g^{(\ell)\prime}(z_j^{(\ell)})$
      $\frac{\partial J}{\partial b_j^{(\ell)}} = \delta_j^{(\ell)}$
      $\frac{\partial J}{\partial w_{jk}^{(\ell)}} = \delta_j^{(\ell)} g^{(\ell-1)}(z_k^{(\ell-1)})$
    end for
  end for
  return $\frac{\partial J}{\partial \boldsymbol{\theta}} = \nabla \boldsymbol{\theta}$ (The derivatives of the cost function with respect to all parameters in the network)

Implementations of both forward and backward propagation are usually vectorized and process a batch of training examples at once to speed up the process. Lastly, we formulate stochastic gradient descent, an algorithm used to minimize our cost function. Since our cost function is expressed as a sum across all of the training data, taking the gradient would require computing the derivatives for every example in our training set. The key insight in SGD is that at each step of the algorithm we can sample a minibatch of labeled training examples drawn uniformly from the training set to get an approximation of the gradient. A crucial parameter of the algorithm is the earning rate ϵ. It represents the rate, at which we allow $\boldsymbol{\theta}$ to be adjusted.

Algorithm 3 Stochastic Gradient Descent

Require: Learning Rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
  while convergence criterion not met **do**
    Sample a minibatch of m labeled examples from the training set $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^m$
    Compute gradient estimate $\hat{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^m J\left(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i\right)$
    Apply update $\boldsymbol{\theta} = \boldsymbol{\theta} - \epsilon \hat{g}$
  end while

Since the function defined by our network is non-convex, gradient descent

algorithms are not guaranteed to converge to a global minimum. In fact models with many layers have an extremely large number of local minima. This can be problematic for gradient based optimization algorithms if they have a high cost compared to the global minimum.The question whether neural networks used in practice encounter this problem is an area of active research. Recent studies Goodfellow and Vinyals (2014); Dauphin et al. (2014) suggest that for deep and wide enough networks most local minima have a low cost function value and finding the global minimum is not as important as it was previously considered.

Activation Functions

A critical part of the network architecture are activation functions used in the hidden layers. The rationale behind their use is the requirement that the network is able to model non-linear functions. If no activation functions or linear ones would be used, the resulting model would be a composite of linear functions and thus linear itself. The design of activation functions is actively pursued in research and there do not exist many guiding theoretical principles. Historically, neural networks were trained using the previously defined sigmoid function $\text{sigm}(x)$. In practice, sigmoid is not used anymore (as an activation function for hidden layers, it is still used as the activation for the ouput layer) due to some undesirable properties. Most importantly, sigmoid units saturate most of their domain i.e. they are sensitive only when the input is close to 0. This characteristic makes gradient based learning difficult as the gradient tends to be almost zero for most inputs. A different activation function which tends to outperform sigmoid in most problems is hyperbolic tangent, $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$. It saturates the same way as the sigmoid function, but unlike the sigmoid, its output is zero-centered. We can think think of the hyperbolic tangent as a scaled sigmoid since $\tanh(x) = 2\,\text{sigm}(2x) - 1$. Nowadays the default choice of activation function is the rectified linear unit(ReLu) defined as $\text{ReLu}(z) = \max(0, z)$. It has several useful properties, which make it a suitable choice. Namely, it greatly accelerates the convergence of stochastic gradient descent as shown by Krizhevsky et al. (2017). This is due to its almost linear, non-saturating form. In addition, both the derivative and the function itself is much easier to compute. Note, that unlike the first two activation functions described ReLu is not differentiable at every point in its domain. Specifically, ReLu is not differentiable at 0. In practice software implementations usually return either the left or right side derivative.

1.2.2 Regularization

Machine learning differs from optimization problems in that, the goal is not to only perform well only on training data but also on new, previously unseen, inputs. The collection of strategies devised to increase the generalization performance, often time at the expense of training error, are known as regularization. When training a machine learning model, we attempt to minimize the training error defined by some cost function J. In addition we would like the generalization error also known as test error to be low as well. We estimate the generalization error of the model by its performance on a test set of labeled examples (drawn from the same distribution as the training set) previously unseen during the training process. This leads to a central concepts in machine learning: underfitting

and overfitting. Underfitting happens when the model is unable to achieve low enough cost on the training set. Overfitting, on the other hand, occurs when the difference between the training error and test error is too large. Naturally, these notions are problem specific, i.e. test error for one problem setting can be state of art for that specific task but subpar for a different one. In this section we briefly discuss strategies to prevent overfitting used across most machine learning models and some strategies mostly specific to training neural networks. Most machine learning models have hyperparameters, which are parameters of the model that are not learned during the training process. Rather they adjust the settings of the model. Previously, we described the notion of the test set, which can be used to estimate the generalization error of the model. In order to choose the optimal hyperparemeters we divide the set of labeled examples at our disposal further to get a validation set. We then train multiple models on the training set with different hyperparameter values and observe the validation set error to choose the best model.

L^2 Regularization

A straightforward way to regularize our model is to augment the cost function J by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$. The resulting cost function then takes the form $\tilde{J}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta})$. The hyperparameter $\alpha \in [0, \infty)$ controls how much the weight penalty contributes to the overall cost function. In the context of neural networks we usually penalize only the weights $\mathbf{w}$ of the model and leave the biases $\mathbf{b}$ unregularized. By specifying the norm penalty Ω, we arrive at different solutions of optimum $\boldsymbol{\theta}$. One of the most common choices is L^2 regularization. If we regularize the weights $\mathbf{w}$ only, then it corresponds to the choice $\Omega(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2$. The resulting cost function then takes the form

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

If we were to take the derivative of the added regularization term with respect to the weights $\mathbf{w}$ the result would be $\alpha\mathbf{w}$. Therefore, at each step of the gradient descent algorithm we decrease the weights by a factor of $\alpha\mathbf{w}$ in addition to decreasing the weights by the original cost function J.

L^1 Regularization

Choosing $\Omega(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_i |w_i|$, sum of the absolute values of the weights we arrive at L^1 regularization. Analogous to L^2 regularization, the cost function takes the form

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) + \alpha\|\mathbf{w}\|_1$$

Since the gradient is not defined at $\mathbf{w} = \mathbf{0}$ we take the subgradient of the regularization term with respect to $\mathbf{w}$ to get $\alpha\,\mathrm{sign}(\mathbf{w})$, where sign is applied elementwise. Adding L^1 penalty yields solutions that are more sparse, which means that some parameters have an optimum value of 0.

Early Stopping

A different strategy to prevent overfitting used mostly in deep learning is so called *early stopping*. When training a large model with enough capacity to overfit, we can reliably observe the training error decreasing as a function of time trained, however the validation error decreases for a certain amount of time and then begins to increase. This suggests we could get a better model using parameters from an earlier point in the training. Early stopping can be formalized by using a parameter p, which represents the number of steps in the training process we allow the validation error to increase before we stop the process and use the last best performing learned parameters $\boldsymbol{\theta}$. We described early stopping as a strategy to prevent overfitting and its relation to minimizing validation set error shows how it can increase the performance of the model. How exactly does early stopping function as a regularization technique is not immediately clear, however. Bishop (1995) suggests that choosing to stop the learning process earlier corresponds to restricting the parameter space of $\boldsymbol{\theta}$ to a neighborhood of the initial value $\boldsymbol{\theta}_0$. Due to its simplicity and the fact that it can easily be used with other regularization strategies, earlier stopping is a popular and widely used technique.

Dropout

Last technique we describe is known as *dropout* Srivastava et al. (2014). The main idea is that during training we "drop" each hidden unit in the network with probability p, known as *dropout rate*. A dropped unit is simply multiplied by 0 and thus obviously outputs 0 to the next hidden layer. This is performed only during training of the network. When making predictions on new inputs we need to scale the activations down by a factor of $1 - p$ to account for more units than the network was trained on. This technique effectively introduces a source of noise to the model and forces the network not to explicitly rely on a single unit to recognize insights in the training data thus preventing overfitting. Srivastava et al. (2014) showed that dropout can lead to better results than more classical regularization strategies. In addition, similarly to early stopping, dropout can be used with other forms of regularization.

2. Unsupervised Learning

Unsupervised learning differs from supervised learning in the available set of data. In the first chapter we defined our training set as $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$. In the unsupervised setting we are limited to an unlabeled set of examples $\mathcal{D} = \{(\mathbf{x}_i\}_{i=1}^N$. It immediately follows, that whenever we are given labeled examples we can also perform unsupervised learning tasks. Since we do not have the correct labels for our data, the goal is usually to find some pattern in the data, which is useful for whatever end task is at hand. This is much less well defined problem since the question of what patterns we are looking for is not immediately clear. In addition there is no clear performance metric P, since we cannot simply compare our output $\mathbf{y}^*$ with the correct label $\mathbf{y}$. In the context of finance, unsupervised learning is used less frequently as supervised learning. The reason is that we are most often interested in predicting values rather than discovering structure in our data. Therefore unsupervised learning is usually used as a preprocessing step in supervised learning, allowing us to learn a representation of the data which both captures the information in it and reduces its complexity.

Principal Component Analysis

One of the most common approaches to this task is principal component analysis (PCA). In PCA we would like to arrive at a different representation of our data. A possible approach is one that reduces some notion of similarity across our features. This corresponds to removing redundancy in our data. In the context of PCA as a preprocessing step to a supervised learning algorithm, if we know that two features are highly similar, we could drop one of them without much loss in the predictive power of our model. In this task the new representation of our data is a linear combination of our original features.

K-Means Clustering

A typical example of unsupervised learning is clustering. In this task we attempt to find subgroups within our data. We do this by grouping observation which are similar or by seeking that observations between different clusters are dissimilar. K-Means Clustering is a simple approach for segmenting our data into predefined K non-overlapping groups. The algorithm tries to minimize the sum of within cluster variation across all clusters, which is simply a measure of dissimilarity among observations in a cluster. Dissimilarity is often measured in Euclidean distance. This problem is computationally difficult, however algorithms which quickly converge to a local minimum exist.

3. Empirical Results

3.1 Classifier Performance Evaluation

There are multiple ways to evaluate a binary classifier. In order to describe them we first introduce the two types of error a classifier can make: a false positive (FP), which arises when we estimate $y^* = 1$ but the correct label is $y = 0$ and a false negative (FN), which arises when we estimate $y^* = 0$ but the correct label is $y = 1$. It is not immediately clear what type of error should be preferred in predicting loan default. This is closely related to risk aversion of the investor, whether its an individual in peer to peer lending or a financial institution. A correct prediction can arise in the case of $y^* = y = 1$, known as a true positive (TP) or $y^* = y = 0$, known as a true negative (TN). These values can be summarized in a confusion matrix shown in Table 3.1.

		Correct Label	
		1	0
Classifier Prediction	1	TP	FP
	0	FN	TN

Table 3.1: Confusion Matrix

From these values we can compute performance measures of a classifier summarized in Table 3.2.

Accuracy	(TP+TN)/(TP+TN+FP+FN)
Recall	TP/(TP+FN)
Precision	TP/(TP+FP)
Specifity	TN/(TN+FP)
False Positive Rate (FPR)	FP/(TN+FP)
False Negative Rate (FNR)	FN/(TP+FN)

Table 3.2: Performance Measures

Both binary classifiers described in Chapter 1 provide a value $f^*(\mathbf{x})$. This can be understood as a measure of confidence that $y = 1$ which is monotonically related to $p(y = 1|\mathbf{x})$. Note that this does not necessarily need to be a probability as is the case in SVMs. To make a prediction y^* we establish a decision rule: if $f^*(\mathbf{x}) > \tau$ then $y^* = 1$ and set $y^* = 0$ otherwise. In the context of SVMs we instituted the default value $\tau = 0$. In neural networks, where we output a probability, the default value is usually 1/2.

Receiver Operating Characteristic

The performance of a classifier as measured by values in Table 3.2 is dependent on the choice of τ. A different measure of performance, independent of the choice

of τ, can be gained by running a classifier for a set of thresholds τ and plotting recall and FPR as a function of τ. This is called a receiver operating characteristic (ROC) curve. Any system can achieve the point (FPR, recall) = (0,0) by setting $\tau \to \infty$ and thus classifying all observations as negative. Analogously any classifier can achieve the point (FPR, recall) = (1,1) by setting $\tau \to -\infty$ and classifying all observations as positive. If a system is performing at chance level, it can achieve any point on the diagonal line Recall = FPR by choosing an appropriate threshold. The quality of a ROC curve can be summarized by the area under curve or AUC. Naturally, higher AUC scores are desirable with the maximum being 1 which signifies a classifier which correctly labels all observations at all thresholds levels τ.

Precision Recall Curve

In precision recall curves we plot precision and recall as a function of τ. In order to summarize the curve with a single value, the area under curve is used the same way as in a ROC. Precision recall curves can be more informative than ROC curves in unbalanced classification tasks where one class dominates the other.

3.2 Hyperparameter Optimization

There are multiple ways hyperparameters can be tuned. The most simple being manual optimization, i.e. running the model multiple times with different hyperparameters which are set manually while using both insight gained from previous runs and domain knowledge. This approach requires almost no technical overhead, however suffers both from lack of performance and reproducibility. Grid search consists of defining an H-dimensional grid, where H is the number of hyperparameters, with each point on the grid consisting of some combination of hyperparameters. These hyperparameter choices are then evaluated one by one. This technique benefits greatly from parallelization, however is computationally expensive if H is large. Additionally, continuous hyperparameters such as the learning rate need to be evaluated on a discrete grid which can have negative effects. Random search is similar to grid search, however instead of creating a finite list of tested hyperparameters we define a distribution for each. At every run we sample the distribution for all hyperparameters, train and evaluate the model. More complex methods for hyperparameter tuning like Bayesian search exist. These are more difficult to implement and require a more nuanced approach to parallelization Hutter et al. (2014). To tune the hyperparameters of trained models we choose to perform random search for neural networks as they have many more hyperparameters and choose to use grid search for other models.

3.3 Loan Default Prediction

3.3.1 Data Description

The data used in this example comes from Lending Club, a peer to peer lending platform which allows registered users to both apply for loans as well as finance loans of other users. During the application process the applicant gives over

personal data, which is combined with information gathered from external credit rating agencies. The dataset of funded loans from 2007 is available with up to date information about the current loan status. It consists of 1646801 observations with 150 variables. The dataset is limited to loan applicants in the United States. To demonstrate how machine learning can be used in finance we create binary classifiers which use a subset of the available features to predict whether the loan is going to be successfully paid. This can be used both during the approval process for new loans as well as in predicting the potential losses of a financial institution.

3.3.2 Data Preparation

Features containing information which was not available at the time of loan application as well as features which were updated throughout the process of paying the loan were removed. These leak information from the future and using them would result in an overestimation of the classifier's performance. Additionally, features which would require additional processing such as user entered description of the loan were removed. Furthermore, features which have more than 5 percent of missing values across the entire dataset are excluded and the remaining missing values are imputed using the mean of the training set. We limit the dataset to loans which have already reached their maturity as not doing so would overestimate the default rate. We are left with 298256 observations with 24 features and the outcome variable loan status. The dataset was split into 3 parts, a training, development and test using the ratio $8 : 1 : 1$. A development set was used to tune hyperparameters instead of cross validation because of the high number of observations available and due to the computational requirements of using cross validation to tune neural network hyperparameters.

3.3.3 Data Transformation

The remaining training set contains some features which need to be processed before use in machine learning models. Most importantly the outcome variable loan status is encoded as 1 for defaulted loans and 0 for paid off loans. The variables fico range high and fico range low are combined into fico range avg as the difference between them for all observations is 4. Variables representing the date a loan is issued and the date an applicant first drew credit are combined to a variable representing the number of months from the applicant's first credit use to loan issue date. Categorical variables are encoded using one hot encoding. For a complete list of included variables see A.1.

3.3.4 Model Performance Comparison

Neural Network

A deep learning model was implemented in the language Python using Tensorflow, an open source deep learning library. Due to the computational demands the training and hyperparameter optimization was done on Google Colab, a platform which offers free use of GPU virtual instances for non-commercial use. The GPU used is NVIDIA TESLA K80. A variation on SGD called Adam, introduced by

Kingma and Ba (2014), was used as the optimizer. The hyperparameters tuned were number of hidden layers, number of hidden units in each layer and learning rate. As mentioned above the hyperparameters were tuned using random search. Table 3.3 shows the performance evaluated on the development set after 30 epochs for 3 basic models with 1 hidden layer and increasing number of hidden units.

# of hidden units	**AUC (ROC)**	**AUC (PR)**
500	0.6668	0.2387
100	0.6576	0.2343
50	0.6529	0.2266

Table 3.3: Basic model performance

We can see that as we increase the capacity of the model the performance slightly increases. Best performance on the development set in terms of AUC ROC was achieved using a model with 4 hidden layers and the following number of hidden units 1784, 1065, 925, 891. Batch normalization, as introduced by Ioffe and Szegedy (2015), was used to both optimize the learning process as well as introduce some noise as a form of regularization, furthermore early stopping was used to select the best performing model. Using additional forms of regularization (dropout, l1 and l2 penalty) had negative effects on performance and thus were not used.

SVM

The complexity of the optimization problem associated with kernalized SVMs scales quadratically with the number of observations. This means that using SVMs with kernels for tasks with high number of observations is not practical. However linear SVMs can be trained much more efficiently. Due to the high number of observations in our training set only linear SVMs are used. The models were implemented using the Python machine learning library scikit-learn. They were trained using a CPU with hyperparameter optimization via grid search training 4 models in parallel. Both l1 and l2 penalty variations were evaluated on the development set with the best performing model using l2 regularization.

Test Set Performance

We evaluate the best performing models on the development set using held out test set. We add comparison with logistic regression.

Model	**Regularization**	**AUC (ROC)**	**AUC (PR)**
NN	Early stopping Batch Normalization	0.6825	0.2489
Linear SVM	l2	0.6733	0.2377
LR	l2	0.6793	0.2448

Table 3.4: Test set performance on loan dataset

A neural network model outperformed both linear SVM and logistic regression in both ROC and PR AUC. However, the differences in performance is small and the scores reflect comparable performance in all three models.

3.4 Credit Card Default Prediction

3.4.1 Data Description

The dataset used in the second example contains information on default payments, demographic factors, credit data, history of payment, and bill statements of credit card clients in Taiwan from April 2005 to September 2005. The task is formulated as predicting the probability a credit card holder will default in the next month. The dataset was used in Yeh and hui Lien (2009) and made available online. It consists of 30000 observations and 23 features and a binary label defining whether the holder defaulted next month. The dataset is split into training, validation and test sets using the ratio 8:1:1.

3.4.2 Data Preparation

Feature engineering was already performed on the dataset. We encode categorical variables using one hot encoding and keep other encoding choices consistent with the original work. The full dataset feature descriptions can be found in Yeh and hui Lien (2009).

3.4.3 Model Performance Comparison

Neural Network

A neural network model was trained and it's hyperparameters tuned using random search. Due to smaller number of available observations the network is much more likely to overfit to the training data and therefore dropout is added to early stopping and batch normalization as a regularization tool. Best performance on the development set in terms of AUC ROC was achieved using a model with 3 hidden layers and the following number of hidden units 400, 283, 166. Dropout rate of approximately 40% was added to all hidden layers of the network to increase generalization performance.

SVM

Due to the smaller number of observations training kernelized SVMs is more computationally feasible. We train and evaluate models using the widely used Radial Basis Function (RBF) kernel, polynomial kernels up to the 4th degree as well as previously used linear SVMs. As the table 3.5 shows, model using RBF kernel outperformed both linear and polynomial SVMs. We can also see that performance decreases as we increase the polynomial kernel degree.

Model	Degree	AUC (ROC)	AUC (PR)
RBF SVM		0.7224	0.5036
Polynomial SVM	2	0.7122	0.4811
Polynomial SVM	3	0.7002	0.4707
Polynomial SVM	4	0.6926	0.4425
Linear SVM		0.7046	0.4945

Table 3.5: Performance of various types of SVM models

Test Set Performance

The best performing SVM, neural network and logistic regression models were evaluated on the test set.

Model	Regularization	AUC (ROC)	AUC (PR)
NN	Batch Normalization, Dropout,Early stopping	0.78344436	0.5541372
RBF SVM	l2	0.719548408	0.4822387
LR	l2	0.718745928	0.4769602

Table 3.6: Test set performance on credit card dataset

RBF kernel SVM slightly outperformed logistic regression while as Table 3.6 suggests neural networks outperformed both models by a significant margin in both ROC and PR AUC.

Conclusion

In this work we gave a summary of machine learning, it's classification and the problem settings where it can be applied. Next the theory underpinning two widely used models was discussed.First we introduced support vector machines by building up the theory from using the model on linearly separable data, to a more general formulation where linear separability does not hold and finally extended the model to be able to capture non-linearity using kernels. Next basic neural network models, namely deep feedforward networks were described. We discussed model architecture, activation functions used in both older variants of neural networks as well as more modern default choices. We described the learning process namely forward and backward propagation and basic stochastic gradient descent algorithm. Both traditional regularization tools as well as techniques unique to neural networks were introduced as a method to increase generalization. The problem setting accompanying unsupervised learning tasks was formulated and brief summary of basic unsupervised learning techniques was given.

Lastly, we trained, optimized and evaluated the discussed models on real world data. We used the first dataset to create models for predicting loan default. Neural network, logistic regression and due to computational limitations of kernalized SVMs only linear SVM models were trained. We describe two basic strategies for hyperparameter tuning, random and grid search. These were used to optimize the model hyperparameters. The model performance was compared on a held out test set. While neural networks slightly outperformed logistic regression, linear SVMs yielded worse results to both other models.

The second example demonstrating machine learning applications in finance used data on credit card holders to predict their default. The lower size of the dataset allowed the training and optimization of kernalized SVM models which outperformed logistic regression. However, a neural network model yielded significantly better predictions to both other models.

Using real world datasets we demonstrated the viability of data driven machine learning models as a replacement to more traditional methods in finance related problem settings. Both the results and the inherent limitations in some of the discussed methods suggest that the choice of used model should be both task and data specific and further analysis is needed.

Bibliography

Bishop, C. M. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995. ISBN 0198538642.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM. ISBN 0-89791-497-X. doi: 10.1145/130385.130401.

Cortes, C. and Vapnik, V. Support-vector networks. *Machine Learning*, 20(3): 273–297, Sep 1995. ISSN 1573-0565. doi: 10.1007/BF00994018.

Cristianini, N. and Shawe-Taylor, J. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-78019-5.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 2933–2941. Curran Associates, Inc., 2014.

E. Rumelhart, D., E. Hinton, G., and J. Williams, R. Learning representations by back propagating errors. 323:533–536, 10 1986.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. ISBN 0262035618, 9780262035613.

Goodfellow, I. J. and Vinyals, O. Qualitatively characterizing neural network optimization problems. *CoRR*, abs/1412.6544, 2014.

Hutter, F., Hoos, H., and Leyton-Brown, K. An efficient approach for assessing hyperparameter importance. 2:1130–1144, 01 2014.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. 12 2014.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi: 10.1145/3065386.

Lending Club. Lending club loan statistics, 2018. URL `https://www.lendingclub.com/info/download-data.action`. Accessed: 2018-02-15.

McCulloch, W. S. and Pitts, W. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6.

Mitchell, T. M. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

Rosenblatt, F. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435.

Yeh, I.-C. and hui Lien, C. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications*, 36(2, Part 1):2473 – 2480, 2009. ISSN 0957-4174.

A. Attachments

A.1 Variables kept in Dataset

Abbreviated variable name	Description
loan_status	Current status of the loan
collections_12_mths_ex_med	Number of collections in 12 months excluding medical collections
revol_util	Revolving line utilization rate, or the amount of credit the borrower is using relative to all available revolving credit.
verification_status	Indicates if income was verified by LC, not verified, or if the income source was verified
open_acc	The number of open credit lines in the borrower's credit file.
total_acc	The total number of credit lines currently in the borrower's credit file
pub_rec	Number of derogatory public records
loan_amnt	The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.
delinq_2yrs	The number of 30+ days past-due incidences of delinquency in the borrower's credit file for the past 2 years
delinq_amnt	The past-due amount owed for the accounts on which the borrower is now delinquent.
inq_last_6mths	The number of inquiries in past 6 months (excluding auto and mortgage inquiries)
emp_length	Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years.
annual_inc	The self-reported annual income provided by the borrower during registration.

acc_now_delinq	The number of accounts on which the borrower is now delinquent.
revol_bal	Total credit revolving balance
int_rate	Interest Rate on the loan
dti	A ratio calculated using the borrower's total monthly debt payments on the total debt obligations, excluding mortgage and the requested LC loan, divided by the borrower's self-reported monthly income.
installment	The monthly payment owed by the borrower if the loan originates.
tax_liens	Number of tax liens
pub_rec_bankruptcies	Number of public record bankruptcies
fico_avg	Average of fico_range_low and fico_range_high
months_since_earliest_cr_line	The difference in months between the date of loan is issued and the date of earliest credit record
state	The state provided by the borrower in the loan application
home_ownership	The home ownership status provided by the borrower during registration or obtained from the credit report. Our values are: RENT, OWN, MORTGAGE, OTHER
purpose	A category provided by the borrower for the loan request.
term	The number of payments on the loan. Values are in months and can be either 36 or 60.

Table A.1: Variables used in loan default prediction models