Humboldt Universität zu Berlin

Hausarbeit

Vorhersage von Kredit-Ausfallwahrscheinlichkeiten mit neuronalen Netzen

Thomas Siskos (580726)

Datenanalyse II

Dozent:

Dr. Sigmund Klinke

30. März 2018

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Ausfallwahrscheinlichkeit ist ein Begriff der Finanzen und beschreibt die Wahrscheinlichkeit, dass ein Kreditnehmer, innerhalb eines Zeitrahmens, nicht in der Lage sein wird seine Verpflichtungen zum vorher festgelegten Termin einzuhalten. Die möglichst genaue Schätzung und Vorhersage eines solchen Ausfalls ist von entscheidender Bedeutung. Die Bepreisung von Vermögenswerten, das Einschätzen der Risiken von Krediten und Kreditportfolios sowie die Wertschätzung anderer Finanz-Produkte hängen maßgeblich von der Präzision der geschätzten Ausfallwahrscheinlichkeiten ab (Miao et al., 2008). Es gibt zwei wesentliche Denkschulen bei der Analyse von Kreditausfällen, der markt-basierte und der statistische Ansatz. Die markt-basierte Art modelliert Kreditausfälle mithilfe struktureller Modelle, wohingegen der statistische Ansatz empirische Methoden bemüht, historische Daten auszuwerten. Diese Daten können beispielsweise aus der Buchhaltung, beziehungsweise aus den Bilanzen von Unternehmen gewonnen werden (Härdle et al., 2012).

Zur Quantifizierung der Ausfallwahrscheinlichkeit werden in der Literatur oft Kennzahlen verwendet, die vornehmlich eine oder mehrere Bilanzpositionen gegen eine oder mehrere andere ins Verhältnis setzen. Diese Kennzahlen haben sich in vergangenen Studien oft als hilfreich erwiesen (Altman, 1968). In dieser Arbeit werden wir die 28 finanziellen Kennzahlen bestimmen, die von Zhang und Härdle (2010) verwendet wurden. Wir werden mithilfe dieses transformierten Datensatzes neuronale Netze unterschiedlicher Architekturen trainieren und miteinander vergleichen. Insbesondere verwenden wir klassische neuronale Netze mit einer versteckten Schicht, Netze mit zwei versteckten Schichten und Netze mit fünf versteckten Schichten. Außerdem werden wir versuchen die Entscheidungsprozesse eines simplen neuronalen Netzes mithilfe von Konturenplots zu visualisieren und zu interpretieren. Die verschiedenen Architekturen wurden mithilfe der Programmiersprache `Python`, genauer mit dem Modul `tensorflow`, erzeugt. Der Quellcode für die neuronalen Netze ist auf github.com/thsis/DAII einsehbar.

Der folgende Abschnitt beschreibt den Datensatz der Creditreform Datenbank, sowie die Art und Weise wie die Daten bereinigt und transformiert wurden um die 28 Finanz-Kennzahlen zu bestimmen. Abschnitt 3 erläutert die Theorie und Anwendung neuronaler Netze auf die Daten. Der vierte Abschnitt präsentiert die Ergebnisse. Der letzte Abschnitt enthält eine abschließende Zusammenfassung und kritische Würdigung.

Tabelle 1: Variablen des Creditreform Datensatzes

Variable	Bedeutung
ID	Kennnummer jedes Unternehmens
T2	Solvenz-Status (solvent:0, insolvent:1)
JAHR	Jahr
VAR1	Scheck, Kassenbestand
VAR2	Vorräte - Gesamt
VAR3	Umlaufvermögen
VAR4	Sachanlagen - Gesamt
VAR5	Immaterielle Vermögensbestände
VAR6	Gesamtvermögen
VAR7	Forderungen aus Lieferung und Leistung
VAR29	Forderungen ggü. Unternehmen mit Beteiligungsverhältnis
VAR8	Grundstücke und Bauten
VAR9	Eigenkapital
VAR10	Gesellschafterdarlehen
VAR11	Pesionsrückstellungen
VAR12	kurzfristige Verbindlichkeiten - Gesamt
VAR13	langfristige Verbindlichkeiten - Gesamt
VAR14	Bankschulden
VAR15	Verbindlichkeiten aus Lieferung und Leistung
VAR30	Verbindlichkeitern ggü. Unternehmen mit Beteiligungsverhältnis
VAR16	Umsätze
VAR17	Vertriebs- / Verwaltungsaufwand
VAR18	Abschreibungen
VAR19	Zinsaufwendungen
VAR20	Gewinn vor Zins und Steuern (EBIT)
VAR21	Betriebsgewinn
VAR22	Jahresüberschuss
VAR23	(Lager-) Bestandsveränderungen
VAR24	Veränderungen der Verbindlichkeiten ggü. Vorjahr
VAR25	Veränderungen Bargeld/Kassenbestand/flüssige Mittel
VAR26	Branchenzugehörigkeit
VAR27	Rechtsform
VAR28	Anzahl Mitarbeiter

2 Die Creditreform Datenbank

Die Creditreform Datenbank enthält Daten für 20.000 solvente und 1.000 insolvente deutsche Firmen aus den Jahren 1997 bis 2007. Der Datensatz wurde durch das Labor für empirische und quantitative Forschung (LEQR) der Humboldt Universität zu Berlin bereitgestellt. Die enthaltenen Variablen stammen aus den Bilanzen der Unternehmen und stellen für potentielle Investoren die Hauptgrundlage für Analysen dar. Ein Unternehmen wird entweder mit sich selbst verglichen, indem der zeitliche Verlauf der Bilanzposten untersucht wird oder das Unternehmen wird mit ähnlichen Firmen verglichen, indem eine Auswahl finanzieller Kennzahlen betrachtet wird (Berk & DeMarzo, 2016).

Rund die Hälfte der Daten stammt aus den Jahren 2001 und 2002. Da 1996 keine insolventen Unternehmen vorliegen, werden alle Beobachtungen dieses Jahres gelöscht. Der Großteil der verbleibenden Unternehmen ist entweder im Baugewerbe, im Handel, in der Industrue oder im Immobilienwesen tätig. Andere Kategorien umfassen beispielsweise Branchen wie Landwirtschaft und Bergbau, Elektrizität-, Gas- und Wasserversorgung, die Gastronomie, Logistik und soziale Dienstleistungen. Alle Unternehmen die zu diesen Kategorien gehören werden von der folgenden Analyse ausgeschlossen, um die Schichtung des Trainingsdatensatzes nicht unnötig zu komplizieren. Außerdem werden sowohl die kleinsten, als auch die größten Unternehmen entfernt. Betrachtet werden nur Unternehmen, deren Gesamtvermögen zwischen 10.000 und 10.000.000 Euro liegen. Die kleinsten Unternehmen werden entfernt, da deren finanzielle Lage oft von den Finanzen einer einzelnen verantwortlichen Person, typischerweise die Eigentümerin oder der Eigentümer, abhängt. Die größten Unternehmen werden hingegen entfernt, da sie in Deutschland nur in den allerseltensten Fällen Gefahr laufen in die Zahlungsunfähigkeit zu geraten. Des Weiteren werden Unternehmen entfernt, bei denen während der Berechnung der finanziellen Kennzahlen Nullen im Nenner auftreten (Chen, 2010).

Die verbleibenden Unternehmen gliedern sich in verschiedene Sektoren, von denen die vier häufigsten im Folgenden analysiert werden. Von den 9567 solventen Unternehmen sind 35,9% in der Industrie, 34,1% im Handel, 19,5% im Baugewerbe und 10,4% in der Immobilienbranche tätig.

Tabelle 2: Definitionen der finanziellen Kennzahlen

Name	Formel	Kennzahl
x1	VAR22/VAR6	Gesamtkapitalrentabilität (ROA)
x2	VAR22/VAR16	Nettogewinnmarge
x3	VAR21/VAR6	
x4	VAR21/VAR16	Betriebsgewinnmarge
x5	VAR20/VAR6	
x6	(VAR20+VAR18)/VAR6	EBITDA
x7	VAR20/VAR16	
x8	VAR9/VAR6	Eigenmittelquote (einfach)
x9	(VAR9-VAR5)/(VAR6-VAR5-VAR1-VAR8)	Eigenmittelquote (angepasst)
x10	VAR12/VAR6	
x11	(VAR12-VAR1)/VAR6	Nettoverschuldung
x12	(VAR12+VAR13)/VAR6	
x13	VAR14/VAR6	Schuldenquote
x14	VAR20/VAR19	Zinsdeckungsgrad
x15	VAR1/VAR6	
x16	VAR1/VAR12	Liquiditätsgrad
x17	(VAR3-VAR2)/VAR12	Quick Ratio
x18	VAR3/VAR12	Current Ratio
x19	(VAR3-VAR12)/VAR6	
x20	VAR12/(VAR12+VAR13)	
x21	VAR6/VAR16	Kapitalumschlag
x22	VAR2/VAR16	Lagerumschlag
x23	VAR7/VAR16	Forderungsumschlag
x24	VAR15/VAR16	Verbindlichkeitenumschlag
x25	log(VAR6)	Proxy für die Unternehmensgröße
x26	VAR23/VAR2	Gestaffelte Prozentuale Lagerbestandsänderung
x27	VAR24/(VAR12+VAR13)	Gestaffelte Prozentuale Forderungsänderung
x28	VAR25/VAR1	Gestaffelte Prozentuale änderung des Cash Flow

Von den 782 insolventen Unternehmen sind 45,0% im Baugewerbe, 28,2% in der Industrie, 21,6% im Handel und 5,1% in der Immobilienbranche tätig.

Anschließend werden mithilfe der verbleibenden Unternehmen die finanziellen Kennzahlen ermittelt. Die Variablen `x1-x7` gehören zu den sogenannten Rentabilitätsverhältnissen. Rentabilitätsverhältnisse haben sich in der Vergangenheit als besonders starke Prädiktoren für Kreditausfälle erwiesen. Zum Beispiel gewährt die Gesamtkapitalrentabilität (return on assets, ROA) `x1` einen Einblick in die Umsatzstärke eines Unternehmens im Vergleich zu dessen Kosten. So signalisiert ein höherer Wert der Kennzahl, dass ein Unternehmen in der Lage ist mehr Geld mit weniger Mitteln zu verdienen. Die Nettogewinnmarge `x2` hingegen veranschaulicht den Anteil des Umsatzes, den das Unternehmen als Einnahmen einbehält. Ein hoher Wert geht mit einem profitablen Unternehmen einher, das seine Kosten zu kontrollieren versteht (Chen et al., 2011).

Eine weitere Reihe der Kennzahlen beschreibt die sogenannte Hebelwirkung. Damit ist das Ausmaß gemeint, in dem ein Unternehmen auf Schulden als Finanzierungsquelle angewiesen ist (Berk & DeMarzo, 2016). Da Unternehmen Schulden und Eigenkapital kombinieren, um ihre Aktivitäten zu finanzieren, erweisen sich Kennzahlen über ebenjene Hebelwirkung als hilfreiche Werkzeuge um

die Zahlungsfähigkeit eines Unternehmens einzuschätzen. Zu den Kennzahlen der Hebelfinanzierung gehören die Variablen `x8-x14`. Beispielsweise misst die Nettoverschuldung `x11` die Höhe der kurzfristigen Verpflichtungen, welche nicht durch die liquidesten Vermögensbestände gedeckt sind, als prozentualer Anteil des Gesamtvermögens. Somit misst diese Kennzahl auch die kurzfristige Liquidität eines Unternehmens (Chen et al., 2011).

Die sechs Folgenden Verhältnisse `x15-x20` gehören in den Bereich der Liquiditäts-Kennzahlen. Liquidität ist eine weit verbreitete Variable, die in vielen Kreditentscheidungen eine wichtige Rolle spielt. Liquidität beschreibt die Möglichkeiten eines Unternehmens Vermögensbestände in kurzer Zeit in Bargeld umzuwandeln. Die wohl wichtigste Kennzahl für die Liquidität ist der Anteil der Kassenbestände am Gesamtvermögen `x15`. Ein weiterer wichtiger Gradmesser für die Liquidität eines Unternehmens ist der sogenannte Quick-Ratio `x17`. Mithilfe des Quick-Ratio versucht man einzuschätzen, ob ein Unternehmen über ausreichend liquide Mittel verfügt um insbesondere kurzfristige Zahlungen zu decken. Ein hoher Quick-Ratio indiziert, dass es für das Unternehmen unwahrscheinlich ist, kurzfristig in Zahlungsnot zu geraten (Berk & DeMarzo, 2016).

Einen weitereren wichtigen Typus von betriebswirtschaftlichen Kennzahlen stellen die Aktivitätskennzahlen `x21-x24` dar. Aktivitätskennzahlen messen die Effizienz mit der ein Unternehmen eigene Ressourcen aufwendet, um Umsatz mithilfe seiner Vermögensbestände zu generieren (Chen et al., 2011).

Zusätzlich berechnen wir den Logarithmus des Gesamtvermögens `x25`. Dieser Risikoindikator stellt die Größe eines Unternehmens dar und versetzt uns in die Lage große, mittlere und kleine Unternehmen miteinander in Beziehung zu setzen. Als letzte Gruppe betriebswirtschaftlicher Kennzahlen berechnen wir die gestaffelten prozentualen änderungen des Cash-Flow, des Lagerbestandes und der Forderungen im Vergleich zum Vorjahr `x26-x28`. (Chen et al., 2011).

Um Einflüsse von Ausreißern auf die neuronalen Netze zu eliminieren, werden extreme Werte für die verschiedenen Verhältnisse durch das 0.05- bzw. das 0.95-Quantil ersetzt. Präziser ausgedrückt, folgen wir der Regel, wenn $x_{ij} < q_{0.05}(x_j)$, dann setze $x_{ij} \stackrel{!}{=} q_{0.05}(x_j)$. Beziehungsweise, wenn $x_{ij} > q_{0.95}(x_j)$, dann setze $x_{ij} \stackrel{!}{=} q_{0.95}(x_j)$. Wobei $x_i, i \in \{1, \vdots, N\}$ für jeden einzelnen Wert einer Kennzahl x_j und $q_k(x_j), j \in \{1, \vdots, 28\}, k = 0.05, 0.95$ für die jeweiligen Quantile der Kennzahl x_j des Datensatzes steht.

3 Methoden

3.1 Neuronale Netze

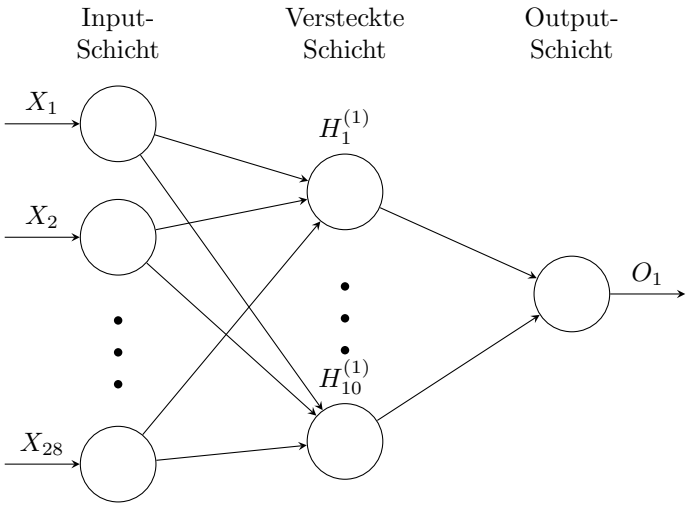
Künstliche neuronale Netze versuchen die Struktur von Gehirnen nachzubilden. Stark vereinfacht enthält ein jedes Gehirn sogenannte Neuronen, welche über zwei Zustände verfügen können. Ein Neuron ist entweder aktiviert oder nicht. In einem Gehirn verändern Neuronen ihren Zustand als Reaktion auf einen chemischen oder elektrischen Stimulus. Das Netz, das die Neuronen innerhalb des Gehirns eines Menschen bilden ist ein enormes Gespinst, in dem der Zustand eines Neurons das Ergebnis tausender anderer Neuronen sein kann. Führt ein Stimulus dazu, dass bestimmte Verbindungen wiederholt aktiviert werden, verfestigt sich deren Verbindung. Das führt dazu, dass bei einem ähnlichen Stimulus ebenfalls dieselben Verbindungen aktiviert werden und zu demselben Outputzustand führen. Dieses Verhalten nennen wir Lernen (Duda, 2012).

Künstliche neuronale Netze vereinfachen die Vorgänge des Gehirns stark. Ein künstliches Neuron wird durch eine Aktivierungsfunktion simuliert, deren Bildmenge das Verhalten eines Schalters emuliert. Wir verlangen von der Aktivierungsfunktion, dass sie über mindestens zwei deutlich verschiedene Zustände verfügt. Typischerweise ist der Output einer Aktivierungsfunktion zum Beispiel entweder Null oder Eins, Null oder größer Null, Minus Eins oder Eins oder eine sigmoidale Funktion, welche auf dem Intervall $[0, 1]$ beschränkt ist. Die Aktivierungsfunktion die in der folgenden Analyse verwendet wird besitzt die Form

$$\phi(z) = \frac{1}{1 + \exp(-z)}. \tag{1}$$

Wie bereits erläutert wurde, sind biologische Neuronen Teil eines hierarchischen Netzwerkes, in dem das Signal von manchen in andere Neuronen eingespeist wird. Im Allgemeinen wird diese Struktur durch miteinander verbundene *Knoten* einer *Schicht* repräsentiert. Ein Netzwerk besteht aus einer Input, ein oder mehrerer versteckter Schichten und einer Output-Schicht. Der Name der mittleren Schichten trägt der Tatsache Rechnung, dass das Wirken der versteckten Schichten zu einem gewissen Grad nur schwer zu beobachten, zuweilen sogar komplett unbeobachtbar ist. Für gewöhnlich sieht der Benutzer eines neuronalen Netzes nur das, was eingespeist wird, sowie dessen Ergebnis. Innerhalb eines *Knoten* wird die Aktivierungsfunktion des Neurons auf die gewichtete

Abbildung 1: Neuronales Netz mit einer versteckten Schicht.



Summe aller Kanten des Graphen angewandt. Der Output h_l^k eines *Knoten* der ersten versteckten Schicht $H_l^{(1)}$ ist somit als

$$\begin{aligned} h_l^{(k)} &= \phi(\beta_0 + x_1\beta_1 + \cdots + x_{28}\beta_{28}) \\ &= \frac{1}{1 + \exp(-\beta_0 - \beta_1 w_1 - \cdots - \beta_{28} x_{28})}. \end{aligned} \tag{2}$$

Das Signal o_1 des Output-Knotens O_1 ist wiederum eine gewichtete Summe der Outputs der versteckten Schicht (Duda, 2012).

$$o_1 = \phi\left(\sum_{q=0}^{u} \gamma_q h_q^{(1)}\right) \tag{3}$$

Mithilfe dieser Gleichungen lassen sich die Inputs vorwärts durch das Netzwerk propagieren, vorausgesetzt man kennt die Gewichte β und γ. Im Allgemeinen muss man die Gewichte zunächst ermitteln. Dies geschieht durch die Optimierung einer Kostenfunktion. Im Falle binärer Zustandsvariablen bietet sich die Kreuzentropie an (Duda, 2012).

$$J = -\frac{1}{n} \sum \left(y \log a + (1-y) \log(1-a)\right), \tag{4}$$

wobei a für die jeweilige Aktivierungsfunktion steht (Duda, 2012).

Die Kreuzentropie aus Gleichung 4 vereint zwei Eigenschaften, die sie als Kostenfunktion besonders auszeichnen. Sie ist größer Null und wenn der Output a des Endknotens nahe am tatsächlichen Zustand der Beobachtung liegt, ist der Wert des Summanden an dieser Stelle ebenfalls sehr gering. Für die optimalen Gewichte gilt es, J zu minimieren. Das Verfahren dazu nennt sich in der Literatur *backpropagation* und ist eine Anwendung der Kettenregel des Ableitens.

$$\begin{aligned} \frac{\partial J}{\partial \beta_j} &= -\frac{1}{n} \sum \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial \beta_j} \\ &= -\frac{1}{n} \sum \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \sigma(z) \left(1-\sigma(z)\right) x_j. \end{aligned} \tag{5}$$

Wobei wir die sigmoidale Aktivierungsfunktion $\sigma(z)$ für a in Gleichung 4 eingesetzt haben. Gleichung 5 lässt sich weiter vereinfachen,

$$\begin{aligned} \frac{\partial J}{\partial \beta_j} &= \frac{1}{n} \sum \frac{\sigma(z)\left(1-\sigma(z)\right) x_j}{\sigma(z)\left(1-\sigma(z)\right)} \left(\sigma(z) - y\right) \\ &= \frac{1}{n} \sum x_j \left(\sigma(z) - y\right) \stackrel{!}{=} 0. \end{aligned} \tag{6}$$

Der Ausdruck in 6 besagt, dass die Rate mit der das Gewicht gelernt wird, von dem Fehler in $(\sigma(z) - y)$ abhängt. Je größer dieser Fehler, desto schneller lernt das Neuron.

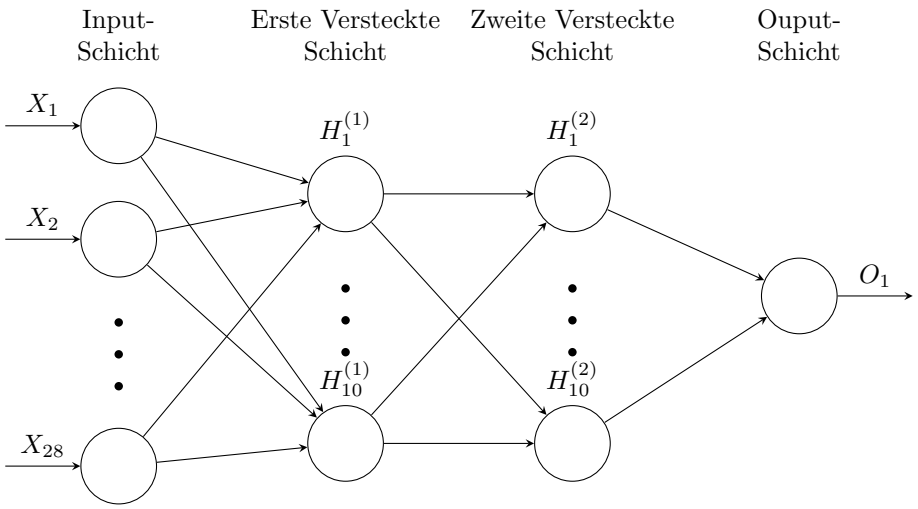
3.2 Architekturen und Hyperparameter

Im Folgenden stellen wir kurz den Aufbau der verwendeten neuronalen Netze vor. Es wurden drei verschiedene Architekturen implementiert, die erste enthält genau eine versteckte Schicht, genau wie in Abbildung 1. Die zweite Art Netz, dargestellt in Abbildung 2, enthält zwei versteckte Schichten und die dritte Architektur, dargestellt in Abbildung 3, beherbergt fünf versteckte Schichten. Bei allen neuronalen Netzwerken handelt es sich um sogenannte *feed-forward-networks*. Das bedeutet, dass die beobachteten Daten von links nach rechts durch das Netzwerk propagiert werden. Alle Neuronen sind vollständig miteinander verbunden. Falls mehrere Schichten vorhanden sind, sind diese jeweils symmetrisch aufgebaut, das heißt alle versteckten Schichten enthalten jeweils dieselbe Anzahl an Neuronen.

Tabelle 3: Verwendete Hyperparameter

Hyperparamter	Verwendete Werte		
Anzahl Neuronen	1	10	20
Lern-Rate	0.01	0.005	
Epochen	10.000	100.000	

Abbildung 2: Neuronales Netz mit zwei versteckten Schichten.

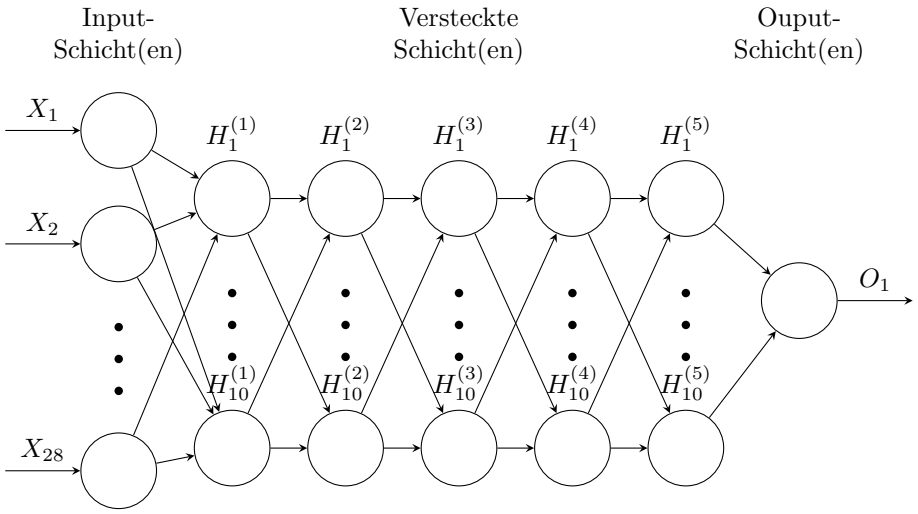


Es werden drei Hyperparameter verwendet. Der markanteste ist sicherlich die Anzahl an Neuronen, da davon die gesamte Struktur der jeweiligen neuronalen Netze abhängt. Für alle drei Architekturen wird zuerst ein Netz trainiert, das jeweils ein Neuron pro versteckter Schicht enthält, anschließend werden Netze trainiert deren Schichten zehn Neuronen enthalten und abschließend werden Netze trainiert deren Schichten zwanzig Neuronen enthalten.

Um die Kostenfunktion zu optimieren, wurde ein Gradientenverfahren angewandt, dessen Lern-Rate einmal 0.01 und 0.005 betrug. Es wurden jeweils einmal 10.000 und 100.000 Trainingsschritte unternommen. Insgesamt wurden somit 36 Modelle trainiert. Um die Güte der Vorhersagen zu quantifizieren wurden sogenannte *Receiver-Operator-Characteristic (ROC)* Kurven gezeichnet sowie die Fläche unter den jeweiligen Kurven (AUC) berechnet.

Die ROC-Kurve wird ermittelt, indem man auf Basis der geschätzten Wahrscheinlichkeiten eines Test-Datensatzes Vorhersagen trifft. Diese Vorhersagen werden in die vier Felder einer sogenannten Konfusionsmatrix eingeordnet. Die Vorhersage kann entweder zutreffen, das heißt einem

Abbildung 3: Neuronales Netz mit fünf versteckten Schichten.



Unternehmen wurde vorhergesagt, dass es nicht in der Lage sein wird einen Kredit zu bedienen und es ist tatsächlich nicht in der Lage (TP). Beziehungsweise es wird korrekterweise vorhergesagt, dass es solvent sein wird (TN). Oder die Vorhersage trifft nicht ein, einem solventen Unternehmen wurde die Insolvenz vorhergesagt (FP), beziehungsweise einem insolventen Unternehmen wurde fälschlicherweise die Kreditwürdigkeit vorhergesagt (FN). Damit sind alle möglichen Fälle erschöpft. Aus dieser Einteilung lassen sich die Falsch-Positiv-Rate (FPR), der Anteil aller fälschlichen positiven Vorhersagen von allen eingetroffenen negativen Ausgängen

$$FPR = \frac{FP}{FP + TN} \tag{7}$$

und die Sensitivität (TPR), das Verhältnis aller wahren positiven Vorhersagen und aller positiven Ausgänge berechnen

$$TPR = \frac{TP}{TP + FN}. \tag{8}$$

Die Fläche unter der Kurve ist ein Maß für die Vorhersagekraft eines Modells, wobei ein perfektes Modell einen AUC-Wert von eins annimmt. Ein AUC-Wert von 0.5 hingegen besagt, dass das Modell dieselbe prädiktive Kraft besitzt, wie der Wurf einer fairen Münze (Winkelmann, 2006).

4 Ergebnisse

Die einzelnen Modelle wurden anhand eines Testdatensatzes beurteilt. Vor dem Training wurde der gesamte Datensatz nach dem Solvenzstatus der Beobachtungen getrennt. Der gesamte Datensatz enthält eine viel größere Anzahl solventer, als insolventer Unternehmen. Daher wurde zunächst der kleinere Teildatensatz der insolventen Unternehmen im Verhältnis vier zu eins aufgeteilt. Diese beiden Datensätze stellen jeweils die Hälfte des Trainings- und Testdatensatz dar. Anschließend wurde aus dem Teildatensatz der solventen Unternehmen zufällig Beobachtungen gezogen, so dass das Verhältnis von solventen und insolventen Firmen in Trainings- und Testdatensatz eins zu eins besteht.

4.1 Eine versteckte Schicht

Die AUC-Werte der verschiedenen Netze sind weitestgehend mittelmäßig, wie Abbildung 5 zeigt. Die meisten liegen im Intervall zwischen 0.74 und 0.76. Dabei gibt es ein Modell, welches einen Wert von 0.5 aufweist und somit über überhaupt keine Vorhersagekraft verfügt. Das schlechteste Modell enthielt ein einziges Neuron in seiner Schicht und wurde mit der höheren Lern-Rate von 0.01 über 100.000 Epochen trainiert. Ein Blick auf die Vorhersagen von Modell `nn1l8` verrät, dass es lediglich den Mittelwert vorhersagt. Die besten aller einfachen Modelle, `nn1l6` und `nn1l10` enthielten jeweils 20 und 10 Neuronen in ihrer versteckten Schicht und wurden 100.000 Epochen lang trainiert. Modell `nn1l6` wurde mit der geringeren Lern-Rate von 0.005 gefittet, wohingegen die Lern-Rate für Modell `nn1l10` 0.01 betrug.

Abbildung 4: Entscheidungsgrenzen des einfachen Netzwerkes

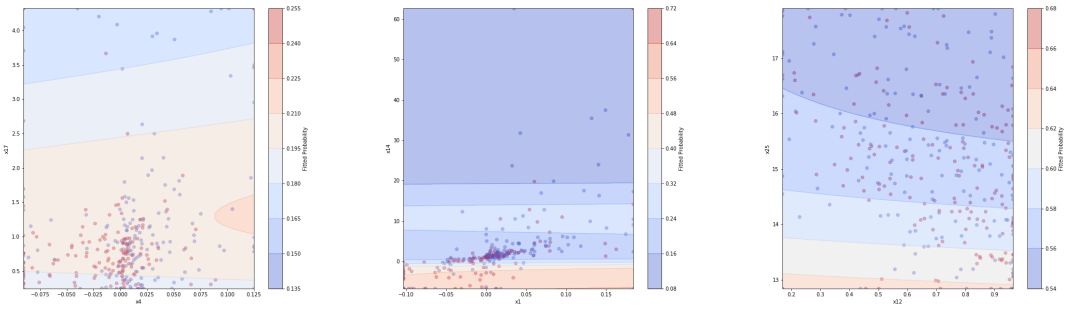
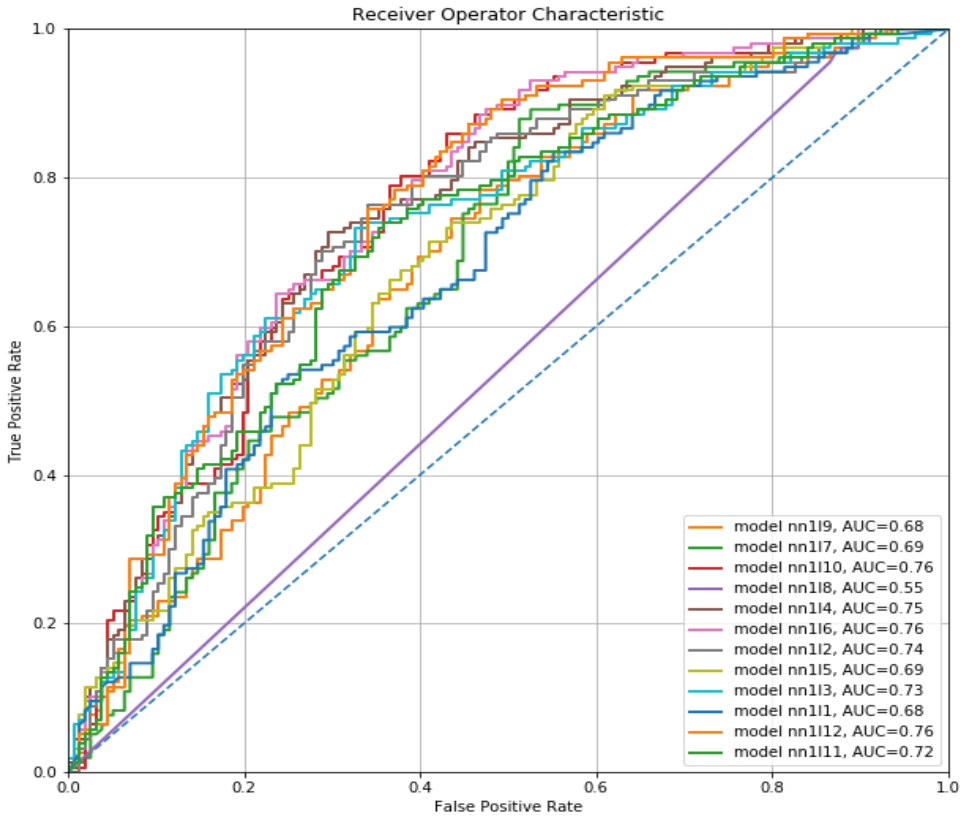


Abbildung 5: ROC-Kurve Neuronaler Netze mit einer versteckten Schicht



Exemplarisch stellen wir in Abbildung 4 einen Blick in das Innenleben der einfachen neuronalen Netze vor. Die Grafiken wurden erstellt, indem Vorhersagen für fiktive Datensätze generiert wurden, in denen lediglich zwei Variablen manipuliert wurden. Für diesen Datensatz wurden die Ausfallwahrscheinlichkeiten mithilfe des Modells `nn1l9` berechnet und die Kontouren der entstandenen dreidimensionalen Mannigfaltigkeit eingefärbt und eingetragen.

In der linken Abbildung wurde die Betriebsgewinnmarge `x4` auf der Abszisse und der Quick-Ratio `17` auf der Ordinate abgetragen. Die Färbung suggeriert, dass es ein Intervall innerhalb der

Werte des Quick-Ratio gibt, welches das Risiko eines Kreditausfalls erhöht und außerhalb dessen es sich verringert. Dieser Effekt sollte jedoch nicht überbewertet werden, da die Spannweite der vorhergesagten Ausfallwahrscheinlichkeiten lediglich von ca. 13 bis 25 Prozent reicht. Vielmehr soll mit dieser Abbildung gezeigt werden, dass selbst das simple neuronale Netzt in der Lage ist höchst nichtlineare Zusammenhänge aufzugreifen.

In der mittleren Abbildung wurde die Gesamtkapitalrentabilität `x1` gegen die Zinsdeckung `x14` abgetragen. Kurioserweise scheint die Betriebsgewinnmenge keinen Einfluss auf das vorhergesagte Ausfallrisiko zu haben, was man an den fast horizontal verlaufenden Konturen ablesen kann. Hingegen verringert sich das Ausfallrisiko je höher die Zinsdeckung eines Unternehmens ausfällt.

Die rechte Abbildung enthält die Konturen des Ausfallsrisikos, die man erhält, wenn man den Proxy für die Größe eines Unternehmens `x25` gegen die Variable `x12` abträgt. Der Verlauf der Konturen legt Nahe, dass kleinere Unternehmen anfälliger für Kreditausfälle sind als große, was auch der oben beschriebenen Intuition entspricht.

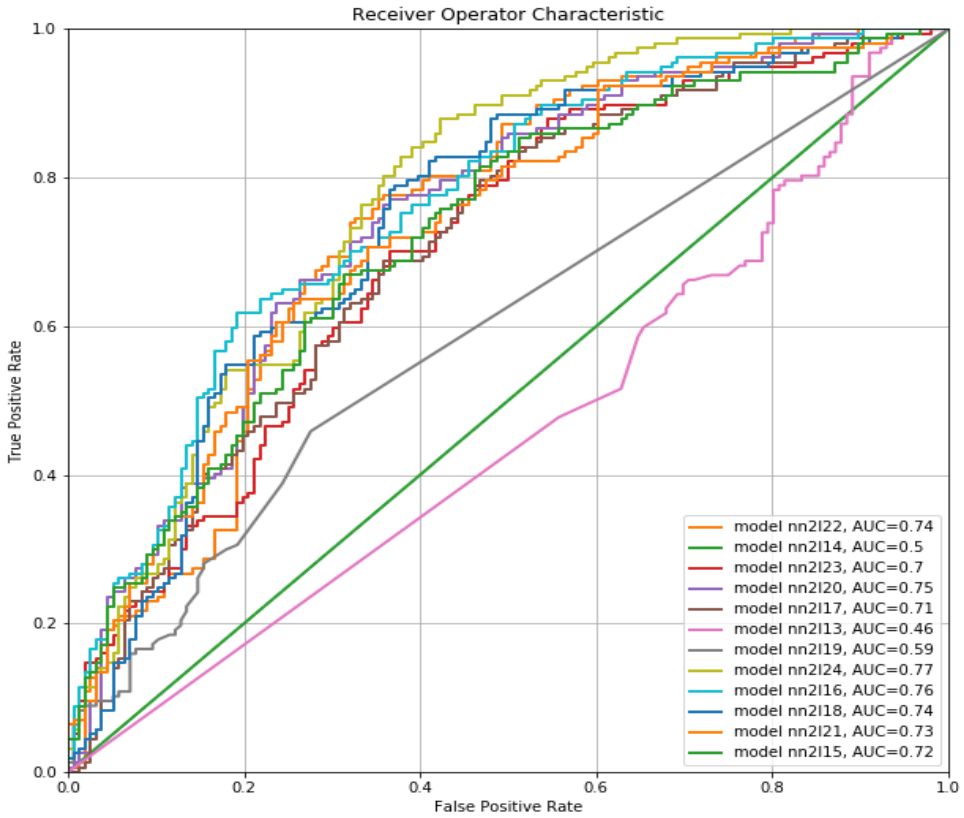
4.2 Zwei versteckte Schichten

Auch für zwei versteckte Schichten sehen die Werte der AUC-Metrik bestenfalls mittelmäßig aus. Im Vergleich mit den einfachen Netzen fällt jedoch auf, dass anteilig mehr Modelle die 0.7-Marke überschreiten. Allerdings gibt es auch mehr Modelle, die durch eine schlechte Performanz auffallen. Das schlechteste aller Modelle `nn2l13` produziert Vorhersagen, bei denen es ratsam wäre das genaue Gegenteil der Vorhersage anzunehmen. Das Modell enthielt ein Neuron pro versteckter Schicht und wurde 10.000 Epochen lang mit einer besonders geringen Lern-Rate trainiert. Die Vermutung liegt nahe, dass der Trainingsprozess vorzeitig abgebrochen wurde und dass somit ein Fall von *underfitting* vorliegt. Das beste Modell enthielt je zwanzig Neuronen pro Schicht. Die Trainings-Schleife wurde mit einer Lern-Rate von 0.01 über 100.000 Epochen durchlaufen.

4.3 Fünf versteckte Schichten

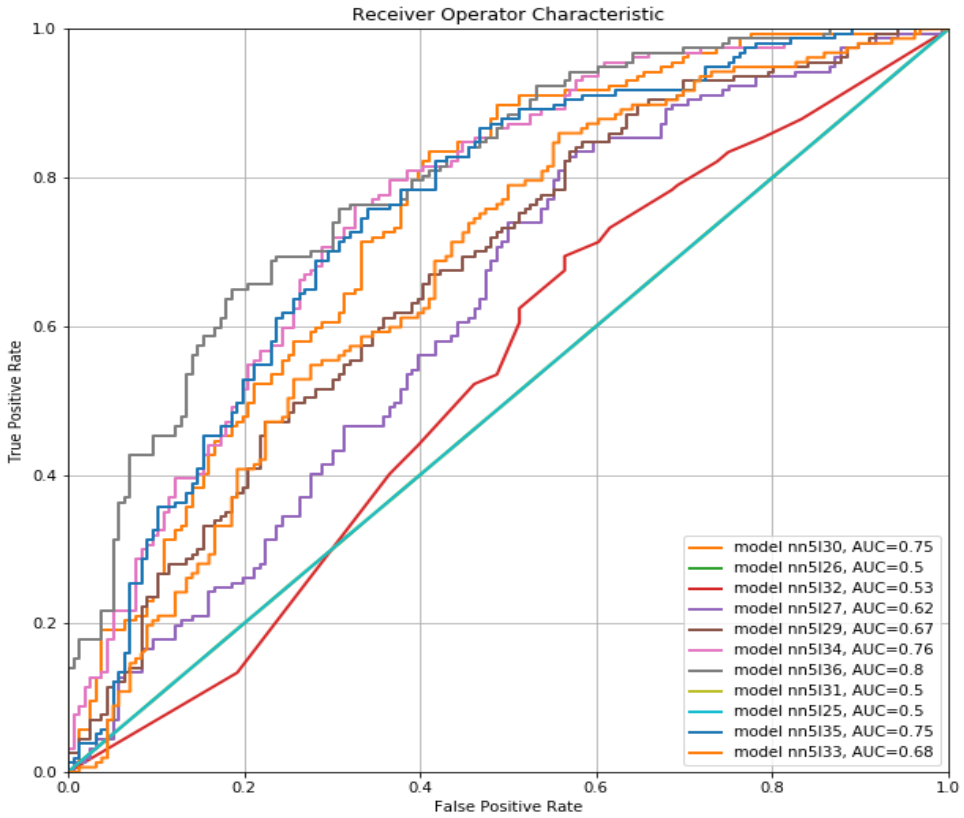
Die Netze mit fünf versteckten Schichten produzierten die weiteste Bandbreite an AUC-Werten. Einerseits produzieren sie den besten Wert, andererseits generiert diese Architektur die meisten

Abbildung 6: ROC-Kurve Neuronaler Netze mit zwei versteckten Schichten



uninformativen Modelle. Vier der Modelle weisen einen AUC-Wert nahe bei 0.5 auf. Jedes dieser Modelle enthielt ein einziges Neuron pro Schicht. Das beste der Modelle, `nn5l36` enthielt fünf mal zwanzig Neuronen und produziert einen Wert von 0.8 für die AUC-Metrik. Die Trainingsparameter bestanden aus der geringeren der beiden Lern-Raten (0.005) in Kombination mit einer hohen Anzahl an Epochen.

Abbildung 7: ROC-Kurve Neuronaler Netze mit fünf versteckten Schichten



5 Zusammenfassung und kritische Würdigung

Im Verlauf dieser Arbeit haben wir den Datensatz der Creditreform-Datenbank aufbereitet, die betrieblichen Kennzahlen berechnet, verschiedene Architekturen neuronaler Netze vorgestellt und angewandt sowie ausgewertet. Für ein simples neuronales Netz haben wir außerdem die Einflüsse einiger ausgewählter Variablen auf die Ausfallwahrscheinlichkeit visualisiert und interpretiert. Wir haben verschiedene Hyperparameter ausprobiert und die resultierenden Prognosen der neuronalen Netze anhand ihrer AUC-Werte miteinander verglichen. Dabei konnten wir verifizieren, dass die Qualität der Vorhersagen maßgeblich von der Anzahl der Neuronen pro versteckter Schicht abhängen. Je mehr Neuronen pro Schicht dem Netz zur Verfügung stehen, desto besser sind die

potentiellen Ergebnisse. Ist die Anzahl der Neuronen hingegen zu gering, besteht die Gefahr des *underfittings*. Erhöht man hingegen die Anzahl der Schichten, gestaltet sich der Trainingsprozess zunehmend komplizierter. Die neuronalen Netze mit den fünf versteckten Schichten bargen das höchste Risiko während der Optimierung der Kostenfunktion in einem lokalen Minimum hängen zu bleiben und somit unbrauchbare Prognosen zu liefern. Allerdings lieferten ebenjene Netze auch die besten Ergebnisse, falls der Trainingsprozess erfolgreich durchlaufen wurde.

Eine Bemerkung sei der Generierung von Trainings- und Validierungsdatensatz gegönnt. Viele der Beobachtungen innerhalb des Rohdatensatzes wurden nicht verwendet. Eine Alternative zu unserer Vorgehensweise bestünde darin den Datensatz nach dem Jahr der Beobachtung einzuteilen und den späteren Datensatz zur Validierung zu verwenden(vgl. Chen, 2011). Diese Design-Methode zwingt dem Datensatz jedoch eine zeitliche Struktur auf und wurde hier zugunsten eines zufälligeren Verfahrens verworfen.

6 Anhang

6.1 Code: Hilfsfunktionen

```
"""Script that generates config-jsons."""
import os
import json

learn = [0.005, 0.01]
nodes = [1, 10, 20]
epochs = [10000, 100000]
architectures = ["nn1l", "nn2l", "nn5l"]

i = 0
for a in architectures:
    for l in learn:
        for n in nodes:
```

```
        for e in epochs:
            i += 1
            if a == "nn1l":
                architecture = n
            elif a == "nn2l":
                architecture = (28, n, n)
            elif a == "nn5l":
                architecture = (28, n, n, n, n, n)

            name = os.path.join("configs", a, "config" + str(i) + ".json")

            config = {
                "name": a + str(i),
                "init": {"units": architecture},
                "train": {
                    "learn": l,
                    "logdir": os.path.join("models", a, a+"_"+str(i)),
                    "epochs": e}}

            with open(name, 'w') as f:
                json.dump(config, f)
```

"""

Utilities for plotting the decision boundary of a neural network.

Compare code with:

http://scikit-learn.org/stable/auto_examples/svm/plot_iris.html

First: Find the two combination of ratios that looks the nicest.

Second: Create a numpy meshgrid & obtain predictions from the trained network.

Third: Plot the predictions by using a contour plot.

```
"""

import numpy as np
from matplotlib import pyplot as plt


def make_meshgrid(x, y, h=0.02):
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy


def setup_contours(data, x, y, model):
    xpos = data.columns.get_loc(x)
    ypos = data.columns.get_loc(y)

    xx, yy = make_meshgrid(data[x].values,
                           data[y].values)

    xflat, yflat = xx.ravel(), yy.ravel()

    rep_data = np.ones((len(xflat), data.shape[1]-1))

    rep_data[:, xpos] = xflat
    rep_data[:, ypos] = yflat

    Z = model.predict(rep_data).reshape(xx.shape)
```

```
    return xx, yy, Z


def plot_decision_boundary(data, x, y, labels, model, **kwargs):
    xx, yy, Z = setup_contours(data=data, x=x, y=y, model=model)

    x0, x1 = data[x].values, data[y].values
    x0lim = x0.min(), x0.max()
    x1lim = x1.min(), x1.max()

    col = data[labels].values
    plt.figure(figsize=(10, 10))

    plt.scatter(x0, x1, c=col, **kwargs)
    CS = plt.contourf(xx, yy, Z, **kwargs)
    CS2 = plt.contour(CS, CS.levels[::2], **kwargs)
    cbar = plt.colorbar(CS, **kwargs)
    cbar.ax.set_ylabel('Fitted Probability')
    # Add the contour line levels to the colorbar
    cbar.add_lines(CS2)

    plt.xlim(x0lim)
    plt.ylim(x1lim)
    plt.xlabel(x)
    plt.ylabel(y)
    plt.legend()

"""
Define function that calculates and plots ROC-Metrics/Curves.
```

```
Compare with:
http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html
"""


from sklearn import metrics
from matplotlib import pyplot as plt


def roc(preds, labels, names=None):
    """
    Calculate ROC-metrics and plot curve for binary classifiers.
    ------------------------------------------------------------

    Parameters:
        - preds: 2D array of predictions. Values should be inside [0 ,1].
        - labels: 1D array of labels.
    Calculate True Positive Rate (tpr), False Positive Rate (fpr) and Area
    under curve (AUC).
    """


    _, m = preds.shape
    if not names:
        names = range(1, m+1)

    fpr = dict()
    tpr = dict()
    auc = dict()

    fig = plt.figure(figsize=(10, 10))
```

```
    ax = fig.add_subplot(111)
    ax.plot([0, 1], [0, 1], linestyle='--')

    for i in range(m):
        fpr[i], tpr[i], _ = metrics.roc_curve(labels, preds[:, i])
        auc[i] = metrics.auc(fpr[i], tpr[i])
        plt.plot(fpr[i], tpr[i],
                 label='model {name}, AUC={auc:.2}'.format(name=names[i],
                                                           auc=auc[i]), lw=2)

    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("Receiver Operator Characteristic")
    plt.legend(loc="lower right")
    plt.grid()
    return fpr, tpr, auc
```

6.2 Code: Datenaufbereitung

```
"""
Clean Creditreform Dataset.

Following steps are undertaken:
    - Select only observations between 1997-2002.
    - Select only observations from the main five industries.
    - remove observations with 0 values in variables used for denominator.
"""
```

```
import pandas as pd
import os
import collections

data_path = os.path.join('data', 'credit.csv')
named_cols = ['ID', 'T2', 'JAHR']
var_cols = ['VAR' + str(i) for i in range(1, 29)]

credit = pd.read_csv(data_path,
                     sep=';',
                     usecols=named_cols + var_cols,
                     low_memory=False)

credit[['VAR26']] = credit[['VAR26']].astype(str)

# Filter by year (i.e. take observations from 1996 to 2002).
credit = credit[(credit['JAHR'] > 1996) & (credit['JAHR'] <= 2002)]

# Filter by asset size (VAR6).
credit = credit[(credit['VAR6'] >= 10 ** 5) & (credit['VAR6'] <= 10 ** 8)]

# Filter out companies with 0 values in variables used in the denominator.
credit = credit[
    (credit['VAR6'] != 0) &
    (credit['VAR16'] != 0) &
    (credit['VAR1'] != 0) &
    (credit['VAR2'] != 0) &
    (credit['VAR12'] != 0) &
    (credit['VAR12']+credit['VAR13'] != 0) &
```

```
    (credit['VAR6']-credit['VAR5']-credit['VAR1']-credit['VAR8'] != 0) &
    (credit['VAR19'] != 0)]


# Generate a defaultdict ind, which defaults to other.
ind = collections.defaultdict(lambda: 'other')


# Update defaultdict with the 4 industry categories.
def add_category(cat, iterable):
    global ind
    for id in iterable:
        ind[str(id)] = cat


categories = ['manufacturing', 'wholesale', 'construction', 'real_estate']

add_category('real_estate', range(70, 75))
add_category('wholesale', range(50, 53))
add_category('construction', range(45, 46))
add_category('manufacturing', range(15, 37))

# Match category id with industry class from WZ 93 by 2 digits and store it.
credit['category'] = credit[['VAR26']].apply(lambda l: [ind[s[:2]] for s in l])

# Remove 'other' category.
credit = credit[credit['category'] != 'other']

# Print summary
```

```
ind_cat = credit.groupby(['T2']).category.value_counts(normalize=True)
print(ind_cat)
print(credit.groupby(['T2'])['ID'].nunique())



# Write to csv.
data_out = os.path.join('data', 'credit_clean.csv')
cols = [c for c in credit.columns if c not in ['VAR26', 'VAR27']]
credit.to_csv(data_out, sep=';', columns=cols, index=False)

"""
Define Finantial Ratios.abs

Following steps are undertaken:
    - Read cleaned data.
    - Define finantial ratios.
    - Censor values for ratios if they fall outside of
      [0.05-percentile, 0.95-percentile] by the respective value.
"""

import pandas as pd
import numpy as np
import os

data_path = os.path.join("data", "credit_clean.csv")
data = pd.read_csv(data_path, sep=';')

# Define ratios.
data['x1'] = data['VAR22'] / data['VAR6']
data['x2'] = data['VAR22'] / data['VAR16']
```

```
data['x3'] = data['VAR21'] / data['VAR6']
data['x4'] = data['VAR21'] / data['VAR16']
data['x5'] = data['VAR20'] / data['VAR6']
data['x6'] = (data['VAR20'] + data['VAR18']) / data['VAR6']
data['x7'] = data['VAR20'] / data['VAR16']
data['x8'] = data['VAR9'] / data['VAR6']
data['x9'] = (data['VAR9'] - data['VAR5']) / \
    (data['VAR6']-data['VAR5']-data['VAR1']-data['VAR8'])
data['x10'] = data['VAR12'] / data['VAR6']
data['x11'] = (data['VAR12'] - data['VAR1']) / data['VAR6']
data['x12'] = (data['VAR12'] + data['VAR13']) / data['VAR6']
data['x13'] = data['VAR14'] / data['VAR6']
data['x14'] = data['VAR20'] / data['VAR19']
data['x15'] = data['VAR1'] / data['VAR6']
data['x16'] = data['VAR1'] / data['VAR12']
data['x17'] = (data['VAR3'] - data['VAR2']) / data['VAR12']
data['x18'] = data['VAR3'] / data['VAR12']
data['x19'] = (data['VAR3'] - data['VAR12']) / data['VAR6']
data['x20'] = data['VAR12'] / (data['VAR12'] + data['VAR13'])
data['x21'] = data['VAR6'] / data['VAR16']
data['x22'] = data['VAR2'] / data['VAR16']
data['x23'] = data['VAR7'] / data['VAR16']
data['x24'] = data['VAR15'] / data['VAR16']
data['x25'] = np.log(data['VAR6'])
data['x26'] = data['VAR23'] / data['VAR2']
data['x27'] = data['VAR24'] / (data['VAR12'] + data['VAR13'])
data['x28'] = data['VAR25'] / data['VAR1']
```

```
# Censor outliers: replace values lower than 0.05-quantile by 0.05-quantile,
# and values higher than 0.95-quantile by 0.95-quantile.
ratios = data[['x' + str(i) for i in range(1, 29)]]

perc05 = ratios.quantile(0.05)
lower = (ratios < perc05)
ratios = ratios.mask(lower, perc05, axis=1)

perc95 = ratios.quantile(0.95)
higher = (ratios > perc95)
ratios = ratios.mask(higher, perc95, axis=1)

# Stitch dataframes back together.
data[['x' + str(i) for i in range(1, 29)]] = ratios

# Write data to disk
ratios["T2"] = data["T2"]
ratios_out = os.path.join('data', 'ratios.csv')
ratios.to_csv(ratios_out, sep=';', index=False)

full_out = os.path.join('data', 'full.csv')
data.to_csv(full_out, sep=';', index=False)
```

6.3 Code: Modelle

```
"""
Set up a function class for feed-forward-neural-networks.

Compare with code on:
https://github.com/nlintz/TensorFlow-Tutorials/blob/master/03_net.py
```

```
    Parameters are:
        - learn : learn rate
        - epochs: number of training epochs
        - units: tuple specifying the size of each layer (input, hidden1, ...)
    """
    import tensorflow as tf
    from tqdm import tqdm


    class NN1L(object):
        def __init__(self, data, labels, units):
            self.units = units
            self.data = data
            self.labels = labels
            self.rows, self.cols = data.shape

        def train(self, epochs, learn, logdir="."):
            train_graph = tf.Graph()

            with train_graph.as_default():
                with tf.name_scope('inputs'):
                    X = tf.placeholder(dtype=tf.float32,
                                       shape=self.data.shape,
                                       name='X')
                    Y = tf.placeholder(dtype=tf.float32,
                                       shape=(self.labels.shape[0], 1),
                                       name='Y')
                with tf.name_scope('hidden_1'):
```

```
            w1 = tf.Variable(
                tf.random_normal(shape=(self.cols, self.units)),
                name='weights_h')
            h1 = tf.nn.sigmoid(tf.matmul(X, w1),
                               name='sigmoid_nodes')
        with tf.name_scope('output'):
            wo = tf.Variable(tf.random_normal(shape=(self.units, 1)),
                             name='weights_o')
            # We don't apply the sigmoid yet. This is done later
            # by the cost-function.
            out = tf.matmul(h1, wo, name='out')
        with tf.name_scope('update'):
            cost = tf.reduce_mean(
                tf.nn.sigmoid_cross_entropy_with_logits(logits=out,
                                                        labels=Y),
                name='cost_cross_entropy')
            train = tf.train.GradientDescentOptimizer(learn).minimize(cost)

        with tf.name_scope('summaries'):
            tf.summary.scalar('cross_entropy', cost)

        with tf.name_scope('global_ops'):
            init = tf.global_variables_initializer()
            merged = tf.summary.merge_all()

    with tf.Session(graph=train_graph) as sess:
        init.run()
        writer = tf.summary.FileWriter(logdir=logdir, graph=train_graph)
        for epoch in tqdm(range(epochs)):
```

```
            c, log, w_1, w_o = sess.run(fetches=[train, merged, w1, wo],
                                        feed_dict={X: self.data,
                                                   Y: self.labels})
            writer.add_summary(log, global_step=epoch)
        writer.close()

        self.w1, self.wo = w_1, w_o

def predict(self, data):
    val_graph = tf.Graph()

    with val_graph.as_default():
        with tf.name_scope('validation'):
            with tf.name_scope('weights'):
                w1 = tf.constant(self.w1, name='hidden_1')
                wo = tf.constant(self.wo, name='output')

            with tf.name_scope('inputs'):
                X = tf.placeholder(shape=(None, self.cols),
                                   dtype=tf.float32,
                                   name='features')

            with tf.name_scope('forward_pass'):
                h1 = tf.nn.sigmoid(tf.matmul(X, w1))
                predictions = tf.nn.sigmoid(tf.matmul(h1, wo))

    with tf.Session(graph=val_graph) as sess:
        tf.global_variables_initializer().run()
```

```
            preds = sess.run(predictions,
                             feed_dict={X: data})

        return preds


class NN2L(object):
    def __init__(self, data, labels, units):
        self.inp, self.h1, self.h2 = units
        self.data = data
        self.labels = labels

    def train(self, epochs, learn, logdir="."):
        train_graph = tf.Graph()

        with train_graph.as_default():
            with tf.name_scope('inputs'):
                X = tf.placeholder(dtype=tf.float32,
                                   shape=self.data.shape,
                                   name='X')
                Y = tf.placeholder(dtype=tf.float32,
                                   shape=(self.labels.shape[0], 1),
                                   name='Y')
            with tf.name_scope('hidden_1'):
                w1 = tf.Variable(
                    tf.random_normal(shape=(self.inp, self.h1)),
                    name='weights_h')
                h1 = tf.nn.sigmoid(tf.matmul(X, w1),
                                   name='sigmoid_nodes')
```

```
        with tf.name_scope('hidden_2'):
            w2 = tf.Variable(
                tf.random_normal(shape=(self.h1, self.h2)),
                name='weights_h')
            h2 = tf.nn.sigmoid(tf.matmul(h1, w2),
                               name='sigmoid_nodes')
        with tf.name_scope('output'):
            wo = tf.Variable(tf.random_normal(shape=(self.h2, 1)),
                             name='weights_o')
            # We don't apply the sigmoid yet. This is done later
            # by the cost-function.
            out = tf.matmul(h2, wo, name='out')
        with tf.name_scope('update'):
            cost = tf.reduce_mean(
                tf.nn.sigmoid_cross_entropy_with_logits(logits=out,
                                                        labels=Y),
                name='cost_cross_entropy')
            train = tf.train.GradientDescentOptimizer(learn).minimize(cost)

        with tf.name_scope('summaries'):
            tf.summary.scalar('cross_entropy', cost)

        with tf.name_scope('global_ops'):
            init = tf.global_variables_initializer()
            merged = tf.summary.merge_all()

    with tf.Session(graph=train_graph) as sess:
        init.run()
        writer = tf.summary.FileWriter(logdir=logdir, graph=train_graph)
```

```
        for epoch in tqdm(range(epochs)):
            c, log, w_1, w_2, w_o = \
             sess.run([train, merged, w1, w2, wo],
                      feed_dict={X: self.data,
                                 Y: self.labels})
            writer.add_summary(log, global_step=epoch)
        writer.close()

        self.w1, self.w2, self.wo = w_1, w_2, w_o

def predict(self, data):
    val_graph = tf.Graph()

    with val_graph.as_default():
        with tf.name_scope('validation'):
            with tf.name_scope('weights'):
                w1 = tf.constant(self.w1, name='hidden_1')
                w2 = tf.constant(self.w2, name='hidden_2')
                wo = tf.constant(self.wo, name='output')

            with tf.name_scope('inputs'):
                X = tf.placeholder(shape=(None, self.inp),
                                   dtype=tf.float32,
                                   name='features')
            with tf.name_scope('forward_pass'):
                h1 = tf.nn.sigmoid(tf.matmul(X, w1))
                h2 = tf.nn.sigmoid(tf.matmul(h1, w2))
                predictions = tf.nn.sigmoid(tf.matmul(h2, wo))
```

```
        with tf.Session(graph=val_graph) as sess:
            tf.global_variables_initializer().run()

            preds = sess.run(predictions,
                             feed_dict={X: data})

        return preds


class NN5L(object):
    def __init__(self, data, labels, units):
        self.inp, self.h1, self.h2, self.h3, self.h4, self.h5 = units
        self.data = data
        self.labels = labels

    def train(self, epochs, learn, logdir="."):
        train_graph = tf.Graph()

        with train_graph.as_default():
            with tf.name_scope('inputs'):
                X = tf.placeholder(dtype=tf.float32,
                                   shape=self.data.shape,
                                   name='X')
                Y = tf.placeholder(dtype=tf.float32,
                                   shape=(self.labels.shape[0], 1),
                                   name='Y')
            with tf.name_scope('hidden'):
                w1 = tf.Variable(
                    tf.random_normal(shape=(self.inp, self.h1)),
```

```
            name='weights_h1')
        w2 = tf.Variable(
            tf.random_normal(shape=(self.h1, self.h2)),
            name='weights_h2')
        w3 = tf.Variable(
            tf.random_normal(shape=(self.h2, self.h3)),
            name='weights_h3')
        w4 = tf.Variable(
            tf.random_normal(shape=(self.h3, self.h4)),
            name='weights_h4')
        w5 = tf.Variable(
            tf.random_normal(shape=(self.h4, self.h5)),
            name='weights_h5')
        h1 = tf.nn.sigmoid(tf.matmul(X, w1),
                           name='sigmoid_h1')
        h2 = tf.nn.sigmoid(tf.matmul(h1, w2),
                           name='sigmoid_h2')
        h3 = tf.nn.sigmoid(tf.matmul(h2, w3),
                           name='sigmoid_h3')
        h4 = tf.nn.sigmoid(tf.matmul(h3, w4),
                           name='sigmoid_h4')
        h5 = tf.nn.sigmoid(tf.matmul(h4, w5),
                           name='sigmoid_h5')

    with tf.name_scope('output'):
        wo = tf.Variable(tf.random_normal(shape=(self.h5, 1)),
                         name='weights_o')
        # We don't apply the sigmoid yet. This is done later
        # by the cost-function.
```

```
            out = tf.matmul(h5, wo, name='out')
        with tf.name_scope('update'):
            cost = tf.reduce_mean(
                tf.nn.sigmoid_cross_entropy_with_logits(logits=out,
                                                        labels=Y),
                name='cost_cross_entropy')
            train = tf.train.GradientDescentOptimizer(learn).minimize(cost)

        with tf.name_scope('summaries'):
            tf.summary.scalar('cross_entropy', cost)

        with tf.name_scope('global_ops'):
            init = tf.global_variables_initializer()
            merged = tf.summary.merge_all()

    with tf.Session(graph=train_graph) as sess:
        init.run()
        writer = tf.summary.FileWriter(logdir=logdir, graph=train_graph)
        for epoch in tqdm(range(epochs)):
            c, log, w_1, w_2, w_3, w_4, w_5, w_o = \
             sess.run([train, merged, w1, w2, w3, w4, w5, wo],
                      feed_dict={X: self.data,
                                 Y: self.labels})
            writer.add_summary(log, global_step=epoch)
        writer.close()

        self.w1, self.w2, self.w3, self.w4, self.w5, self.wo =\
            w_1, w_2, w_3, w_4, w_5, w_o
```

```
def predict(self, data):
    val_graph = tf.Graph()

    with val_graph.as_default():
        with tf.name_scope('validation'):
            with tf.name_scope('weights'):
                w1 = tf.constant(self.w1, name='hidden_1')
                w2 = tf.constant(self.w2, name='hidden_2')
                w3 = tf.constant(self.w3, name='hidden_3')
                w4 = tf.constant(self.w4, name='hidden_4')
                w5 = tf.constant(self.w5, name='hidden_5')
                wo = tf.constant(self.wo, name='output')

            with tf.name_scope('inputs'):
                X = tf.placeholder(shape=(None, self.inp),
                                   dtype=tf.float32,
                                   name='features')
            with tf.name_scope('forward_pass'):
                h1 = tf.nn.sigmoid(tf.matmul(X, w1))
                h2 = tf.nn.sigmoid(tf.matmul(h1, w2))
                h3 = tf.nn.sigmoid(tf.matmul(h2, w3))
                h4 = tf.nn.sigmoid(tf.matmul(h3, w4))
                h5 = tf.nn.sigmoid(tf.matmul(h4, w5))
                predictions = tf.nn.sigmoid(tf.matmul(h5, wo))

    with tf.Session(graph=val_graph) as sess:
        tf.global_variables_initializer().run()
        tf.local_variables_initializer().run()
```

```
            preds = sess.run(predictions,
                             feed_dict={X: data})

        return preds
```

6.4 Code: Training

```
"""Main script for analysis."""
import os
import json
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd

from glob import glob
from models import ann
from utils import graphs, roc
from matplotlib import pyplot as plt

np.random.seed(42)

# Set up train data
data_path = os.path.join("data", "ratios.csv")
ratios = pd.read_csv(data_path, sep=';', index_col=False)

insolvent = ratios.loc[ratios['T2'] == 1, :]
solvent = ratios.loc[ratios['T2'] == 0, :]

data = pd.concat([insolvent, solvent.sample(insolvent.shape[0])])
```

```
train, test = train_test_split(data, test_size=0.2, stratify=data["T2"])
X_train = train.loc[:, train.columns != "T2"].values
y_train = train.loc[:, "T2"].values.reshape((1251, 1))

X_test = test.loc[:, train.columns != "T2"].values
y_test = test.loc[:, "T2"].values.reshape((313, 1))

# Initialize Validation dataframe
Validate = test.loc[:, :]

# Set up Variables for contour-plots.
plot_x = ["x" + str(i) for i in range(1, 14)]
plot_y = ["x" + str(i) for i in range(14, 27)]
plot_vars = zip(plot_x, plot_y)

print("Single Hidden Layer Neural Nets.")
nn1l_configs = glob(os.path.join("configs", "nn1l", "*.json"))

for config in nn1l_configs:
    with open(config, 'r') as model_pars:
        params = json.load(model_pars)

    nn1l = ann.NN1L(data=X_train,
                    labels=y_train,
                    **params["init"])
    nn1l.train(**params["train"])
    validation = nn1l.predict(X_test)
    Validate[params["name"]] = validation.flatten()
```

```
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    for i, (x, y) in enumerate(plot_vars):
        fig_path = os.path.join(params["train"]["logdir"],
                                params["name"] + "_" + str(i) + "_.png")
        graphs.plot_decision_boundary(test.iloc[:, :29], x, y, "T2", nn1l,
                                      cmap=plt.cm.coolwarm, alpha=0.4)
        plt.savefig(fig_path)
        plt.clf()

print("Two Hidden Layers Neural Nets.")
nn2l_configs = glob(os.path.join("configs", "nn2l", "*.json"))

for config in nn2l_configs:
    with open(config, 'r') as model_pars:
        params = json.load(model_pars)

    nn2l = ann.NN2L(data=X_train,
                    labels=y_train,
                    **params["init"])
    nn2l.train(**params["train"])
    validation = nn2l.predict(X_test)
    Validate[params["name"]] = validation.flatten()

    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    for i, (x, y) in enumerate(plot_vars):
        fig_path = os.path.join(params["train"]["logdir"],
                                params["name"] + "_" + str(i) + "_.png")
```

```
        graphs.plot_decision_boundary(test.iloc[:, :29], x, y, "T2", nn2l,
                                      cmap=plt.cm.coolwarm, alpha=0.4)
        plt.savefig(fig_path)
        plt.clf()

print("Five Hidden Layers Neural Nets.")
nn5l_configs = glob(os.path.join("configs", "nn5l", "*.json"))

for config in nn5l_configs:
    with open(config, 'r') as model_pars:
        params = json.load(model_pars)

    nn5l = ann.NN5L(data=X_train,
                    labels=y_train,
                    **params["init"])
    nn5l.train(**params["train"])
    validation = nn5l.predict(X_test)
    Validate[params["name"]] = validation.flatten()

    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    for i, (x, y) in enumerate(plot_vars):
        fig_path = os.path.join(params["train"]["logdir"],
                                params["name"] + "_" + str(i) + "_.png")
        graphs.plot_decision_boundary(test.iloc[:, :29], x, y, "T2", nn5l,
                                      cmap=plt.cm.coolwarm, alpha=0.4)
        plt.savefig(fig_path)
        plt.clf()
```

```
Validate.to_csv(os.path.join("data", "validation.csv"), sep=';')

# Roc for one Hidden Layer
fpr_1l, tpr_1l, auc_1l = roc.roc(preds=Validate.iloc[:, 29:41].values,
                                 labels=Validate.iloc[:, 28].values,
                                 names=list(Validate.iloc[:, 29:41].columns))
plt.savefig("rocNN1L.png")
plt.show()
plt.clf()
# Roc for two Hidden Layers
fpr_2l, tpr_2l, auc_2l = roc.roc(preds=Validate.iloc[:, 41:53].values,
                                 labels=Validate.iloc[:, 28].values,
                                 names=list(Validate.iloc[:, 41:53].columns))
plt.savefig("rocNN2L.png")
plt.show()
plt.clf()

# Roc for five Hidden Layers
fpr_5l, tpr_5l, auc_5l = roc.roc(preds=Validate.iloc[:, 54:].values,
                                 labels=Validate.iloc[:, 28].values,
                                 names=list(Validate.iloc[:, 54:].columns))
plt.savefig("rocNN5L.png")
plt.show()
plt.clf()
```

Literatur

[1] Altman, E., (1968), Finantioal Ratios, Discriminant Analysis and the Prediciton of Corporate Bankruptcy. The Journal of Finance, Vol. 23, 4: 589-609.

[2] Berk, J., DeMarzo, P. (2016), Corporate Finance, 4th edition, Harlow, Pearson Education

[3] Chen, S., Härdle, W.K., Moro, A.R (2011), Modeling default risk with support vector machines, Quantitative Finance, 11(1), 135-154.

[4] Duda, R. O., Hart, P. E., Stork, D. G. (1973). Pattern Classification. New York: Wiley-Interscience.

[5] Miao, H., Ramchander, S., Ryan, P., Wang, T. (2018), Default prediction models: The role of forward-looking measures of returns and volatility, Journal of Empirical Finance, 46: 146-162

[6] Winkelmann, R., Boes, S. (2009): Analysis of Microdata, 2nd edition, 124-128.

[7] Zhang, J.L., Härdle, W.K. (2010). The Bayesian Additive Classification Tree Applied to Credit Risk Modelling. Computational Statistics and Data Analysis, 54: 1197-1205

[8] Jones, E, Oliphant, T., Peterson, P. (2001), Scipy: open source scientific tools for Python.

[9] Hunter, J. D., (2007) Matplotlib: A 2D graphics environment

[10] Abadi M., Agarwal A., Barham, P., et al. (2015): Large-Scale Machine Learning on Heterogeneous Systems

[11] McKinney, W. (2010): ata Structures for Statistical Computing in Python.