Master-Thesis

Studiengang Medizinische Informatik Master

Fakultät für Informatik

Hochschule Mannheim

11.01.2016

Durchgeführt bei der Firma Qudosoft GmbH & Co. KG, Karlsruhe

Betreuer: Prof. Dr. Ivo Wolf, Hochschule Mannheim

Zweitkorrektor: Dr. Jurgis Pods, Qudosoft GmbH & Co. KG

**Untersuchung verschiedener statistischer Modelle zur Vorhersage der Wiederkaufwahrscheinlichkeit von Neukunden im Versandhandel.**

Nicholas Pastuovic

**Erklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Karlsruhe, 11.01.2016

Nicholas Pastuovic

Inhalt

[1 Einleitung 1](#_Toc439752185)

[1.1 Motivation und Zielsetzung 1](#_Toc439752186)

[1.2 Aufbau der Arbeit 2](#_Toc439752187)

[2 Grundlagen 3](#_Toc439752188)

[2.1 Klassifikationsverfahren 3](#_Toc439752189)

[2.1.1 Entscheidungsbäume 3](#_Toc439752190)

[2.1.2 Random Forests 3](#_Toc439752191)

[2.1.3 Support Vector Machine 3](#_Toc439752192)

[2.1.4 Logistische Regression 4](#_Toc439752193)

[2.1.5 Künstliche Neuronale Netze 5](#_Toc439752194)

[2.2 Maschinelles Lernen 6](#_Toc439752195)

[2.2.1 Überwachtes Lernen 6](#_Toc439752196)

[2.2.2 Unüberwachtes Lernen 6](#_Toc439752197)

[2.2.3 Bestärkendes Lernen 6](#_Toc439752198)

[2.3 Gütemaße 7](#_Toc439752199)

[2.3.1 Wahrheitstabelle 7](#_Toc439752200)

[2.3.2 Gütemaße für die Wahrheitstabelle 8](#_Toc439752201)

[2.3.3 F1-Maß 9](#_Toc439752202)

[2.3.4 ROC-Kurve 9](#_Toc439752203)

[2.3.5 Matthews-Korrelations-Koeffizient 10](#_Toc439752204)

[2.4 Verwendete Frameworks 10](#_Toc439752205)

[2.4.1 Pandas 10](#_Toc439752206)

[2.4.2 Scikit-learn 10](#_Toc439752207)

[3 Bibliothekauswahl für neuronale Netze 11](#_Toc439752208)

[3.1 PyBrain 11](#_Toc439752209)

[3.2 Tensorflow 12](#_Toc439752210)

[3.3 Theano 14](#_Toc439752211)

[3.4 Lasagne 16](#_Toc439752212)

[3.5 Scikit-neuralnetwork 17](#_Toc439752213)

[3.6 Fazit 17](#_Toc439752214)

[4 Aktueller Stand der Entwicklung 19](#_Toc439752215)

[4.1 Erster Lösungsansatz 19](#_Toc439752216)

[4.2 Verwendeter Datensatz 20](#_Toc439752217)

[4.3 Ergebnis 20](#_Toc439752218)

[4.4 Unausgeglichene Daten 20](#_Toc439752219)

[4.5 Over- und Undersampling 21](#_Toc439752220)

[4.5.1 Tomek Links 22](#_Toc439752221)

[4.5.2 SMOTE 22](#_Toc439752222)

[4.5.3 ADASYN 22](#_Toc439752223)

[4.6 Kostenbasierte Methoden 23](#_Toc439752224)

[4.7 Vergleich der Sampling-Methoden 24](#_Toc439752225)

[5 Umsetzung 25](#_Toc439752226)

[5.1 Verbesserungsansatz Random Forest 25](#_Toc439752227)

[5.2 Verwendung von neuronalen Netzen 25](#_Toc439752228)

[5.3 Verwendung von SVM 25](#_Toc439752229)

[5.4 Verwendung von Logistischer Regression 25](#_Toc439752230)

[6 Multiple Klassifikationsverfahren 26](#_Toc439752231)

[7 Evaluierung 27](#_Toc439752232)

[8 Fazit 28](#_Toc439752233)

[Abkürzungsverzeichnis iv](#_Toc439752234)

[Tabellenverzeichnis v](#_Toc439752235)

[Abbildungsverzeichnis vi](#_Toc439752236)

[Literaturverzeichnis vii](#_Toc439752237)

# Einleitung

## Motivation und Zielsetzung

In den Big Data Labs der Firma Qudosoft wird derzeit u.a. ein Vorhersagemodell auf anonymisierten Kundendaten eines großen klassischen Versandhändlers erstellt. Eine besonders interessante Fragestellung betrifft die Identifizierung von "One-time buyers" (OTBs): Kunden, die einmalig bestellen und nicht wiederkehren. Diese Kunden stellen für das Unternehmen ein Verlustgeschäft dar, da sie Werbekosten verursachen, ohne jedoch Nachfrage zu generieren. Gleichzeitig ist das Problem aus Sicht des maschinellen Lernens ein extrem schwieriges, da die Einordnung eines Neukunden als OTB oder Wiederkäufer lediglich anhand der Signatur seines Ersteinkaufs erfolgen muss. In einem ersten Klassifikationsversuch wurde eine Genauigkeit von 65% erreicht. Dafür wurde ein 250000 großer Datensatz mit je 40-200 Features, 2-5 Parameter pro Modell und Random Forest als Klassifikationsverfahren verwendet.

Es ist daher wünschenswert, dieses Vorhersageproblem methodologisch eingehender zu untersuchen. Konkret werden verschiedene statistische Modelle wie Logistische Regression, Support Vector Maschines (SVM) und Neuronale Netze auf den vorhandenen Daten trainiert. Mittels Kreuzvalidierungsverfahren (engl. Cross Validation) sollen optimale Parameter für jedes Modell gefunden werden. Dazu wird die vorhandene Hadoop-Infrastruktur genutzt, um diese aufwendigen Berechnungen parallelisiert durchführen zu können. Die dabei derzeit verwendeten Technologien umfassen Python, Scikit-learn, Pandas, Hive und Apache Spark.

Als Ziel sollen alle optimiert berechneten Modelle auf ungesehenen Daten hinsichtlich verschiedener Zielkriterien wie Genauigkeit, Precision, Recall und F-score ausgewertet und verglichen werden. Desweiteren soll eine Genauigkeit von 75% erreicht werden. Eine derzeit in der Entwicklung befindliches Web-Interface könnte in diesem Zuge zu einem komfortabel zu bedienenden Dashboard erweitert werden, dass das Fitten der Modelle, deren Optimierung per Cross Validation und den abschließenden Performance-Vergleich von einer einzigen Oberfläche aus ermöglicht.

## Aufteilung der Arbeit

Im zweiten Kapitel werden alle relevanten Grundlagen für diese Arbeit erläutert. Es werden die verschiedenen Klassifikationsverfahren, Gütemaße für die Performanz sowie die verwendeten Python Frameworks beschrieben. Im nächsten Kapitel werden die relevantesten Frameworks für neuronale Netze untereinander verglichen. Anhand von Code-Beispiele werden kleinere neuronale Netze modelliert und darauf aufbauend ihre Vor- und Nachteile erörtert.

In Kapitel 4 wird der aktuelle Stand der Entwicklung beschrieben. Der verwendete Datensatz samt seiner Merkmale wird exakt beschrieben. Zusätzlich werden die Performanz sowie die momentanen Probleme der Klassifizierung erläutert. Zum Schluss des Kapitels werden verschiedene Lösungsansätze für unbalancierte Datensätze diskutiert. Im nächsten Kapitel wird der erste Ansatz durch eine Optimierung der Parametereinstellungen mittels „Grid-Search“ ermittelt einschließlich einer Selektierung der irrelevanten Merkmale innerhalb des Datensatzes. Anschließend werden verschiedene Klassifikationsverfahren durchgeführt und dessen Performanz hinsichtlich der One-Time-Buyer evaluiert. Im 6. Kapitel werden die gewonnen Wiedereinkaufswahrscheinlichkeiten als neue Merkmale genutzt, um so die Vorteile der einzelnen Klassifizierungsverfahren zu kombinieren. In Kapitel 7 werden die Vor- und Nachteile der Verfahren diskutiert. Insbesondere wird auf die Sampling- und Tuning-Methoden Bezug genommen. Desweiteren wird der Mehrwert der neuen Merkmale durch die Kombination der verschiedenen Klassifikationsverfahren gegenüber der „einfachen“ Verfahren analysiert.

Im letzten Kapitel werden die gewonnenen Erfahrungen diskutiert und daraus die entsprechenden Schlussfolgerungen gezogen.

## 

# Grundlagen

## Maschinelles Lernen

In diesem Unterkapitel werden die unterschiedlichen Arten des maschinellen Lernens beschrieben.

### Überwachtes Lernen

Beim Überwachten Lernen (engl. supervised learning) lernt der Algorithmus anhand einer Menge von Trainingsdaten, die aus Ein- und Ausgaben bestehen, Gesetzmäßigkeiten. Ein „Lehrer“ gibt dem Algorithmus die korrekten Ausgaben für jede Eingabe vor. Nach dem Lernvorgang ist das System in der Lage Gesetzmäßigkeiten nachzubilden.

Diese Arbeit wird ausschließlich Verfahren für das überwachte Lernen behandeln.

### Unüberwachtes Lernen

Beim Unüberwachten Lernen (engl. unsupervised learning) ist kein „Lehrer“ anwesend der die korrekten Antworten vorgibt. Der Algorithmus gruppiert selbstständig die Objekte anhand ihrer Merkmale in Klassen.

### Bestärkendes Lernen

Beim Bestärkendes Lernen (engl. Reinforcement Learning) ist eine Menge von Datensätzen vorhanden, jedoch gibt es keine Vorgabe der Zielwerte. Der Algorithmus versucht eine Gesetzmäßigkeit herzustellen und ein Bewerter bestimmt im Anschluss wie gut das Ergebnis ist. Anhand der Bewertung verfeinert der Algorithmus sein Ergebnis.

## Klassifikationsverfahren

Klassifikationsverfahren beschreiben den Prozess gegebene Objekte anhand ihrer Merkmale in Klassen einzuteilen. Im Folgenden werden verschiedene Verfahren zur Lösung des Klassifikationsproblems vorgestellt.

### Entscheidungsbäume

Durch den Einsatz von Entscheidungsbäumen werden die, in den Daten vorkommenden Variablen (bzw. Merkmale) iterativ partitioniert. Die bekannteste Implementierung für Entscheidungsbäume ist der von Breiman entwickelte CART-Algorithmus[xxx]. Dabei werden die Daten in jeder Iteration anhand des besten Merkmals binär geteilt.

Abbildung .: Beispiel eines Entscheidungsbaumes

### Random Forests

Random Forests ist ein Klassifikationsverfahren, welches zur Lösung ein Ensemble von Entscheidungsbäumen bildet.

### Support Vector Machine

Support Vector Machine (SVM) ist ein supervised learning Verfahren, welches eine Menge von Daten in Klassen unterteilt. Dabei wird eine Hyperebene bzw. Trennlinie so gesetzt, dass die Klassen optimal getrennt werden.

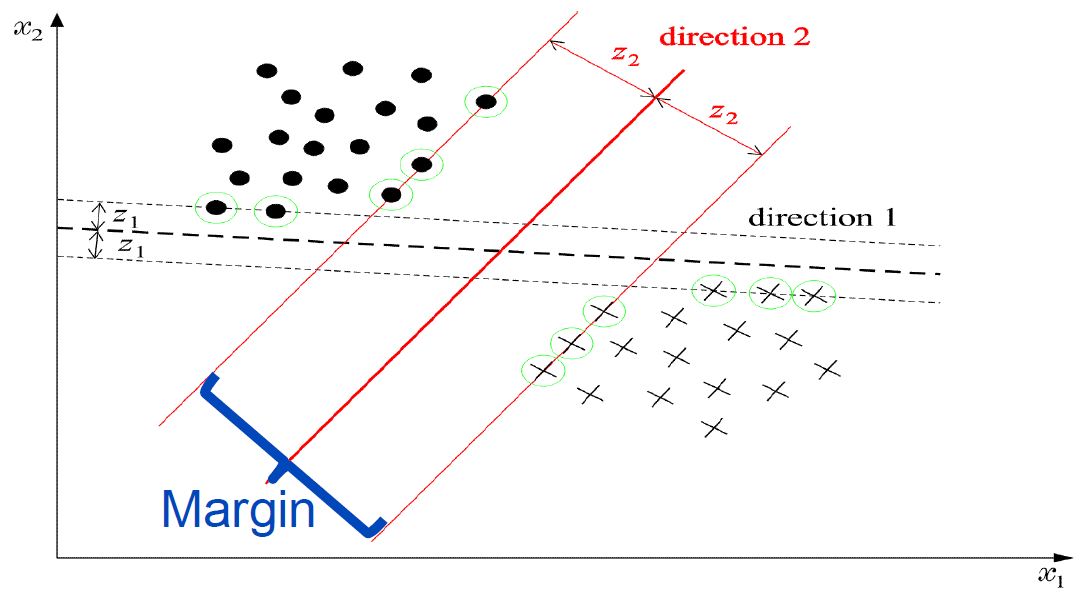


Abbildung .: Support Vector Machine [Buchquelle]

Wie man in Abbildung 1 sieht, gibt es mehrere Möglichkeiten eine Trennlinie zwischen zwei Klassen zu legen. Optimal wird die Hyperebene so gelegt, dass der Abstand der zwischen den Klassen maximal ist. Liegt sie zu nah an einer Klasse, ist die Klassifizierung sehr anfällig und eine korrekte Verallgemeinerung wird dadurch nicht gewährleistet. Die Objekte die der Hyperebene am nächsten liegen heißen Support Vectors. In Abbildung 1 sind diese grün umkreist zu sehen. Um einen eine Optimale Trennung zu erreichen wird der Abstand zwischen den Support Vectors der Klassen berechnet und die Hyperebene daraufhin so gelegt, dass der Abstand der Support Vectors zur Hyperebene möglichst groß ist. Um den Abstand zwischen den Objekten zu berechnen werden diese als Vektoren dargestellt.

#### Nicht linear trennbare Hyperebenen

In den meisten realen Anwendungen können die Datensätze nicht linear getrennt werden. Mithilfe von speziellen Funktionen können die Daten in einen höherdimensionalen Raum transformiert werden um dort eine lineare Separierung vorzunehmen. Da

### Logistische Regression

Die logistische Regression ist eines der bekanntesten Klassifikationsverfahren, welches die Eintrittswahrscheinlichkeit eines Ereignisses unter Beobachtung unabhängiger Variablen berechnet. Es wird mithilfe der linearen Regression (2.1) ein Modell erstellt, dass den linearen Zusammenhang mehrere erklärender Variablen (X1…Xn) auf eine Zielvariable Y beschreibt.

|  |  |
| --- | --- |
|  | (.) |

Nur die Verwendung der linearen Regression als Klassifikation ist nicht geeignet, da zum einen der lineare Zusammenhang beschrieben wird und der Wertebereich auch außerhalb des Wahrscheinlichkeitsintervalls [0,1] liegen kann. Möchte man den Einfluss der erklärenden Variablen auf eine dichotome Zielvariable Y untersuchen, wie es bei der Klassifikation der Fall ist, kommt die logistische Regression zu Einsatz.

|  |  |
| --- | --- |
|  | (.) |

Durch die verwendete Sigmoide-Funktion 2.2 wird der Wertebereich von Y auf [0,1] beschränkt. Werte größer eines Schwellwertes e (meistens 0,5) werden der positiven Klasse und Werte unterhalb der negativen Klasse zugeordnet. Der Schwellwert kann jedoch beliebig an das jeweilige Klassifikationsproblem angepasst werden. Die Optimierung des Schwellwertes kann somit zu einer Verbesserung des Klassifikationsergebnisses führen.

|  |  |
| --- | --- |
| Abbildung .: Logistische Regression anhand eines sigmoiden Kurvenverlaufes | Abbildung .: Beispiel einer linearen Regression |

Anhand Abbildung 2.2 wird veranschaulicht, wie die charakteristischen Eigenschaften der sigmoiden-Funktion verwendet werden um den Wertebereich [0,1] zu realisieren. Die Verlauf bzw. die Verzerrung der Kurve wird über die Koeffizienten und bestimmt. Welchen Wert letztendlich die Koeffizienten annehmen wird mittels des Trainingsdatensatz und des überwachten Lernens festgelegt.

### Künstliche neuronale Netze

Künstliche neuronale Netze (KNN) sind informationsverarbeitende Systeme, die als Vorbild das menschliche Gehirn haben. Sie bestehen aus mehreren Einheiten bzw. Neuronen die untereinander Verbunden sind.

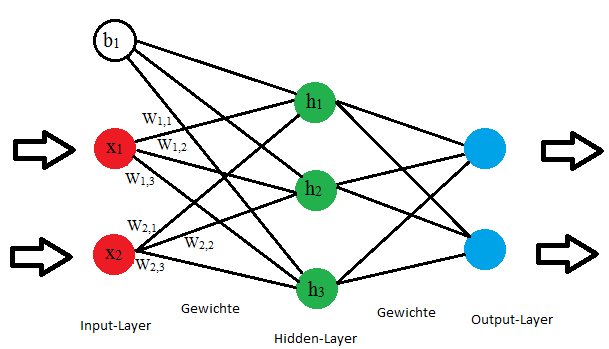


Abbildung .: Aufbau eines neuronalen Netzes

Wie in Abbildung 2.4 zu sehen, gibt es drei verschiedene Arten von Einheiten (engl. Units) und Schichten (engl. Layer). Informationen von der Außenwelt werden von der Input-Schicht entgegen genommen. Dabei hängt die Anzahl der Einheiten, die sich in der Input-Schicht befinden, von der Anzahl der Merkmale ab. Die verschiedenen Einheiten der Schichten sind miteinander über gewichtete Kanten verbunden. Dabei variieren die Verbindungen je nach Topologie des neuronalen Netzes. Zuerst kommen die Daten (bzw. Werte) in der Input-Schicht an und werden dort verarbeitet.

|  |  |
| --- | --- |
|  | (2.3) |

Die Werte werden mit den entsprechenden Gewichten multipliziert und die Produkte aufsummiert. Die Gewichte geben an, wie groß ihr Einfluss auf deren verbundenen Einheit ist, welches dem Aktionspotential dem Gehirn nachempfunden ist. Ein positives Gewicht hat einen erregenden und ein negatives Gewicht einen hemmenden Einfluss. Zusätzlich zu jeder Einheit in der Input- oder Hidden-Schicht kann ein sogenannter Bias hinzugefügt werden. Die Aufgabe des Bias ist es den Impuls eines Neurons zusätzlich zu verstärken oder zu hemmen. Dabei ist jede Bias-Verbindung mit Gewichten ausgestattet. Möchte man z.B. dafür sorgen, dass eine Einheit oft einen erregenden Einfluss hat, wird sie mit einem positiven Gewicht ausgestattet. Daraus resultierend entspricht der Bias einem zusätzlichen Schwellwert. Anschließend werden die bearbeiteten Daten *net* (s. 2.3) an die Hidden oder Output-Schicht weitergeleitet. Je nach Komplexität des Problems wird eine Hidden-Schicht zur Lösung benötigt. Diese kann aus einer oder mehreren Schichten bestehen. Dabei wird zwischen folgenden Aktivierungsfunktionen unterschieden:

|  |  |
| --- | --- |
| C:\Users\npastouvic\AppData\Local\Microsoft\Windows\INetCache\Content.Word\mculloch pitts.bmp  Abbildung .: Binäre Aktivierungsfunktion | C:\Users\npastouvic\AppData\Local\Microsoft\Windows\INetCache\Content.Word\lin aktfkt.bmp  Abbildung .: Lineare Aktivierungsfunktion |
| C:\Users\npastouvic\AppData\Local\Microsoft\Windows\INetCache\Content.Word\sigmoid.bmp  Abbildung .: Sigmoide Aktivierungsfunktion |  |

*Binäre Schwellenfunktion:*

Die binäre Aktivierungsfunktion entspricht dem „ganz oder gar nicht“ Prinzip und ist der Funktionsweise zum Weiterleiten von Signalen nachempfunden.

|  |  |
| --- | --- |
|  | (.) |

Wie man an der Funktion 2.4 sieht, wird bei einem positiven Wert von *net* das Neuron aktiviert und ansonsten gehemmt bzw. es wird kein Signal weitergeleitet.

*Lineare Aktivitätsfunktion:*

Bei der linearen Aktivierungsfunktion werden die gewichteten Daten nicht weiter verarbeitet. Bei einer Klassifizierung gewinnt das Neuron mit dem höchsten Wert.

|  |  |
| --- | --- |
|  | (.) |

*Sigmoide Aktivitätsfunktion:*

Bei der sigmoiden-Aktivierungsfunktion liegt das Ergebnis auf dem Intervall [0,1]. Sigmoide-Funktionen sind besonders für Klassifizierungen geeignet, da das Ergebnis Wahrscheinlichkeiten zurückliefert. Jedoch muss hier stark auf die Performanz geachtet werden, da sehr große Werte für *net* zu einem hohen Rechenaufwand führen können.

|  |  |
| --- | --- |
|  | (.) |

Zum Schluss werden die Daten zur Output-Schicht weitergeleitet. In dieser Schicht wird auf den Daten nochmals eine Aktivierungsfunktion ausgeführt, welches als Ergebnis die entsprechende Klassifikation liefert.

### Netztopologie von neuronalen Netzes

In diesem Unterkapitel werden die verschiedenen Topologien von neuronalen Netzwerken vorgestellt. Wobei die Rückgekoppelten Netzwerke (auch rekurrente Netzwerke) eine spezielle Funktion besitzen, mit der sie zeitlich codierte Informationen in den Daten finden können.

|  |  |  |
| --- | --- | --- |
| direktes feedback.png  Abbildung .: Direkte Rückkopplung eines Neurons | Abbildung .: Indirektes Feedback eines neuronales Netzes | seitliches feedback.png  Abbildung .: Seitliche Rückkopplung innerhalb einer Schicht |

#### Feedforward-Netze

Ein typisches Feedforward-Netzwerk ist in Abbildung 2.4 zu sehen. Die Daten werden nur zur nächsten Schicht weitergeleitet und nicht zu einer vorherigen Schicht zugegeführt.

#### Direkte Rückkopplung

Durch die direkte Rückkopplung wird der Ausgang eines Neurons mit sich selbst verbunden. Somit bildet sich ein zweiter Eingabewert für das gegebene Neuron.

#### Indirekte Rückkopplung

Mithilfe der indirekten Rückkopplung wird die Ausgabe eines Neurons an ein Neuron der vorherigen Schicht verknüpft.

#### Seitliche Rückkopplung

Bei der seitlichen Rückkopplung wird der Ausgabewert an ein Neuron derselben Schicht weitergeleitet.

#### Vollständige Verbindung

Die vollständige Verbindung ist eine Kombination von allen Rückkopplungsarten. So sind alle Neuronen miteinander Verbunden.

## Gütemaße

Im Folgenden werden verschiedene Verfahren zur Bewertung der verwendeten Modelle beschrieben.

### Wahrheitstabelle

Mithilfe einer Wahrheitstabelle(engl. Confusion Matrix) kann ein binärer Klassifikator bewertet werden. Dabei werden alle möglichen Vorhersagekombinationen in einer 2x2-Matrix festgehalten. Dabei sind vier Kombinationen möglich:

* Richtig positiv: Eine positive Klasse wird als solche erkannt
* Richtig negativ: Eine negative Klasse wird als solche erkannt
* Falsch negativ:  Eine positive Klasse wird als negative erkannt
* Falsch positiv:  Eine negative Klasse wird als positive erkannt

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Ergebnis Realität |  |
|  |  | Positiv | Negativ |
| Ergebnis Test | Positiv | richtig positiv (engl. true positive) | falsch positiv(engl. false positive) |
|  | Negativ | falsch negativ(engl. false negative) | richtig negativ(engl. true negativ) |

Tabelle : Wahrheitstabelle

Wie in der Tabelle zu sehen ist, werden die richtig klassifizierten Objekte entlang der Hauptdiagonalen eingetragen und die falsch klassifizierten Objekte entlang der Nebendiagonale. Durch die Matrix kann eine erste Beurteilung über den Klassifikator getroffen werden und die dient als Grundlage für weitere Gütemaße.

### Einfache Gütemaße

Im Folgenden werden die Grundgütemaße die mithilfe der Wahrheitsmatrix getroffen werden beschrieben. Der Wertebereich der Gütemaße beträgt [0,1]. Wobei der ideale Wert der Gütemaße 1 beträgt

#### Sensitivität

Mit der Sensitivität wird der Anteil der positiv richtig Klassen im Verhältnis zu allen positiven Klassen berechnet.

|  |  |
| --- | --- |
|  | (.) |

#### Spezifität

Die Spezifität berechnet den Anteil der richtig erkannten negativen Klassen im Verhältnis zu allen negativen Fällen.

|  |  |
| --- | --- |
|  | (2.8) |

#### Relevanz

Mithilfe der Relevanz wird der Anteil der richtig positiven erkannten Klassen im Verhältnis zu allen positiv klassifizierten Objekten berechnet.

|  |  |
| --- | --- |
|  | (2.9) |

#### Segreganz

Die Segreganz ist das Pendant zu der Relevanz. Es wird hier der Anteil der richtig negativen erkannten Klassen im Verhältnis zu allen negativ klassifizierten Objekten.

|  |  |
| --- | --- |
|  | (2.10) |

#### Genauigkeit

Die Genauigkeit berechnet den Anteil aller korrekt Klassifizierten Objekte im Verhältnis zu der gesamten Testmenge.

|  |  |
| --- | --- |
|  | (.) |

Die genannten Gütemaße besitzen alleine eine geringe Aussagekraft. Eine hohe Spezifität oder Genauigkeit hat zum Beispiel bei unausgeglichenen Datensätzen keine große Bedeutung. Sind zum Beispiel die positiven Klassen in der Mehrheit, üben die negativen Klassen als Faktor eine geringe Auswirkung bei der Performanz aus. Selbst wenn die negativen Klassen durchgehend falsch Klassifiziert werden, ist eine Genauigkeit über 90% möglich. Außerdem kann eine gegenseitige Beeinflussung der Gütemaße nicht ausgeschlossen werden. Zum Beispiel spricht sich ein konservativer Klassifikator selten für die positive Klasse aus. Dadurch entsteht eine hohe Spezifität, da die negativen Klassen weitgehend erkannt werden. Jedoch werden die positiven Klassen nicht berücksichtigt, welches zu einer niedrigen Sensitivität führt. Daher sollten, um eine gute Aussage über die Performanz des Klassifikators zu treffen, kombinierte Maße verwendet werden.

### F1-Maß

Das F-Maß bildet sich aus dem harmonischen Mittel der Relevanz und Sensitivität. Wie bei den Gütemaßen zuvor liegt der Wertebereich im Intervall [0,1], wobei 1 der ideale Wert ist. Zu beachten ist, dass die richtig negativen Klassifizierten Objekte nicht beachtet werden und somit das Augenmerkt auf den richtig positiven liegt.

|  |  |
| --- | --- |
|  | (2.12) |

### ROC-Kurve

Die Receiver-Operating-Characteristic-Kurve (ROC) wird verwendet um grafisch einen Klassifikator zu bewerten oder optimieren. Dabei wird die Relevanz gegen die Segreganz visuell aufgetragen. Verläuft die Kurve entlang der Diagonalen gleicht die Klassifikation einem Zufallsprozess.

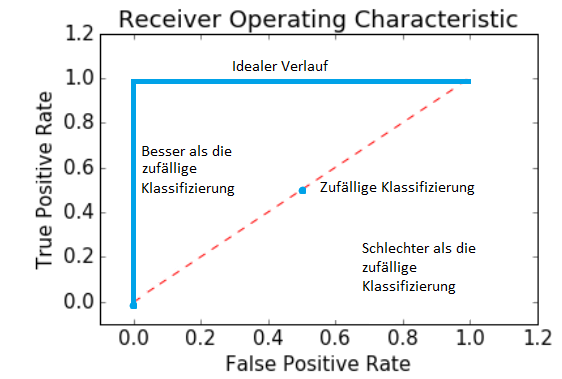


Abbildung .: ROC – Kurve

Die blaue Kurve in Abbildung 3 ist der optimale Verlauf. Zuerst steigt die Kurve senkrecht zur y-Achse, welches der richtigen Klassifizierung der positiven Klassen entspricht. Anschließend werden die korrekt Klassifizierten negativen Klassen aufgetragen, wobei die Kurve optimal waagerecht der x-Achse verläuft. Einen guten Einsatz liefert die ROC-Kurve bei der Betrachtung verschiedener Klassifikatoren. Verläuft eine der Kurven näher an dem Optimum ist diese besser. Problematisch sind Kurven die sich Überlappen bzw. die einen ähnlichen Kurvenverlauf besitzen. Um dieses Problem zu Umgehen kann die Fläche unter der Kurve(engl. Area Under the Curve) verwendet werden. Sie liefert einen eindeutigen Wert der im Intervall [0,1] liegt. Wobei 0,5 einer Zufallsklassifikation entspricht und somit dem schlechtesten Wert entspricht. Ein Vorteil der AUC ist, dass die Werte als Wahrscheinlichkeiten interpretiert werden können. Ist der Wert z.B. bei 0,7, entspricht dies einer 70%igen Wahrscheinlichkeit einer korrekten Klassifizierung.

### Metthews Korrelationskoeffizient

Ein weiteres Gütemaß für die Performanz eines Klassifikators ist der Metthews Korrelationskoeffizient. Der Fokus liegt hier wie gut die korrekte Klassifikation mit der Verteilung der Klassen korreliert.\*

|  |  |
| --- | --- |
|  | (2.13) |

Der Wertebereich des *mett* liegt im Intervall von [-1,1]. Beträgt der Wert 0, entspricht es einer Zufallsklassifizierung und der Idealwert liegt bei 1.

## Sampling-Methoden

Ein häufiges Problem bei realen Klassifizierungen ist ein unausgeglichenes Verhältnis der Klassen in den Datensätzen. Daraus resultiert ein erschwertes Erkennen der Klasse die in der Minderheit ist (*engl. Minority Class*), wobei die Klasse in Überzahl (*engl. Majority Class*) in den meisten Fällen richtig erkannt wird.

|  |  |  |
| --- | --- | --- |
| Konfusionsmatrix | Positives Objekt | Negatives Objekt |
| Klassifikation positiv | 174 (true Positive) | 3523 (false Positive) |
| Klassifikation negativ | 251 (false Negative) | 10976 (true Negative) |

Tabelle : Wahrheitsmatrix der OTB-Klassifizierung

In Tabelle 1 sieht man die Konfusionsmatrix der ersten Klassifizierung des OTBs mittels Random Forest. Wie man anhand der Tabelle erkennen kann, sind die *false Positives*(Objekt ist positiv wird aber negativ Klassifiziert) sehr hoch. Dieser Fehler ist, je nach Fragestellung besonders schwerwiegend. Zum Beispiel bei der Fragestellung ob eine Person „Krank“ oder „Gesund“ ist, kann eine Person als gesund eingestuft wird obwohl sie eigentlich an einer Krankheit leidet. Um diese Minderheitsklassen bei der Generalisierung in ein Gleichgewicht zu stellen, werden verschiedene *Sampling-*Methoden im Folgenden vorgestellt.

## Over- und Undersampling

Bei Undersampling-Verfahren werden Datenpunkte aus der *Majority-Class* entfernt um somit ein Gleichgewicht herzustellen. Ein grundlegendes Verfahren hierbei ist das *Random Undersampling*. Hierbei wird eine zufällige Anzahl von Datenpunkte der Majority-Class entfernt. Die Anzahl der zu entfernenden Punkte kann frei gewählt werden. Ein Nachteil des Oversampling-Verfahrens ist der Verlust von Informationen beim entfernen der Datenpunkte. Es können wichtige Daten entfernt werden, die für eine Generalisierung der *Majority-Class* wertvoll sind. Um dieses Problem zu vermeiden gibt es verschiedene Ansätze des Verfahrens, wie z.B. *Tomek-Links[xxx]*. Hierbei werden die Daten entfernt, die für ein Rauschen verantwortlich sind. Somit wird der Informationsverlust beim zufälligen Entfernen der Daten minimiert.

Das Pendant zum Undersampling ist das Oversampling. Bei diesem Verfahren werden neue *Minority-Class* Daten erstellt um für ein Gleichgewicht zwischen den Klassen zu sorgen. Ein grundlegendes Verfahren hierbei ist das *Random Oversampling.* Bei diesem Verfahren werden zufällig Datenpunkte aus der Minority-Class dupliziert, um somit ein Gleichgewicht herzustellen. Dabei besteht die Gefahr, dass verrauschte Daten verdoppelt werden und sich somit die allgemeine Generalisierung verschlechtert. Ein weiterer Nachteil ist die signifikante Vergrößerung des Datensatzes, welches zu einer längeren Trainingsphase des Klassifikators führt.

Besonders beim Testen des Klassifikators auf ungesehene Daten muss darauf geachtet werden, dass diese nicht durch die Sampling-Verfahren manipuliert werden. Zum Beispiel kann es beim Random Oversampling vorkommen, dass duplizierte Daten in den Testsatz gelangen und somit der Klassifikator die Daten erneut sieht. Darauf resultiert eine Verfälschung der Vorhersagegenauigkeit. Im Folgenden werden verschiedene Over- und Undersampling verfahren vorgestellt.

### Tomek Links

Tomek Links gehört zu den Undersampling Verfahren, welches Datenpunkte „intelligent“ entfernt. Daten die stark unter rauschen werden vom Algorithmus entfernt. Somit kann der Klassifikator die Generalisierungsregel für die Minderheitsklasse erlernen. Um korrekte Datenpunkte zu entfernen werden zuerst zwei Datenpunkte xi und xj von unterschiedlichen Klassen analysiert und die Distanz d(xi,xj) berechnet. Eine paar (xi,xj) ist ein Tomek Link wenn es keinen weiteren Datenpunkt xk vorhanden ist, so dass d(xk, xi) < d(xi, xj) oder d(xk, xj) < d(xi, xj) gilt. Entspricht das Datenpaar (xi,xj) einem Tomek Link, so leidet einer der Punkte unter rauschen oder beide Punkte befinden sich auf der Entscheidungslinie.

### SMOTE

SMOTE (Synthetic Minority Oversampling Technique) ist eine Oversampling Variante, bei der synthetisch „neue Daten“ erstellt werden. Anstatt wie bei dem klassischen Oversampling vorhandene Daten zufällig zu duplizieren und somit den Datensatz zu balancieren, werden durch Verwendung des *k-nächsten-Nachbarn* Algorithmus synthetische Daten erzeugt.

Zuerst werden zufällig Nachbarn aus dem *k-nächsten-Nachbar* Verfahren ausgewählt. Danach werden wie folgt neue synthetische Daten berechnet:

|  |  |
| --- | --- |
|  | (.) |

Wobei die Differenz der Merkmalsvektoren des Datenpunktes und dessen nächster Nachbar ist und [0,1] eine Zufallszahl.

### ADASYN

*Adaptive Synthetic Sampling Approach* (ADASYN) ist ein Oversampling Verfahren, welches auf dem SMOTE-Algorithmus aufbaut. Besonders hervorzuheben ist der Adaptive Ansatz. Dieser besagt, dass aus keinen Daten synthetische hergestellt werden, die unter rauschen leiden.

Zuerst wird der Grad *d* der Ungleichheit berechnet zwischen den Klassen berechnet.

|  |  |
| --- | --- |
|  | (.) |

Wobei d [0,1] .

Als nächstes wird der G-Wert bestimmt. Dieser sagt aus wie viele Sampling-Daten berechnet werden sollen.

|  |  |
| --- | --- |
|  | (4.3) |

Wobei [0,1]. Ist =1, sind die Daten komplett ausgeglichen.  
Anschließend wird für jeden Datenpunkt xiin der Unterklasse der k-nächsten Nachbar gefunden.

|  |  |
| --- | --- |
|  | (4.4) |

Wobei die Anzahl der Klassen in der Minderheit. Danach wird ri normalisiert, um die Verteilungsfunktion zu bestimmen.

|  |  |
| --- | --- |
|  | (4.4) |

Darauf folgend wird die Anzahl der synthetischen Daten in xi Smin berechnet.

|  |  |
| --- | --- |
|  | (.) |

Zum Schluss werden die neuen synthetischen Daten wie beim SMOTE-Verfahren berechnet.

## Kostenbasierte Methoden

Bei den kostenbasierten Methoden wird nicht wie zuvor der vorhandene Datensatz manipuliert, sondern den jeweiligen Klassen wird ein Gewicht zugeordnet. Die Gewichte werden beim erlernen der Generalisierungsregel berücksichtigt. Dadurch gewinnen die Klassen, trotz der ungleichen Verteilung, an gleicher Bedeutsamkeit. Ein großer Vorteil des Verfahrens ist, dass keine Daten entfernt werden und somit kein Informationsverlust beim Lernen entsteht.

# Bibliothekauswahl

In diesem Kapitel wird eine Auswahl von Bibliotheken zur Realisierung von neuronalen Netzen in Python diskutiert. Mittels kleiner Code-Beispielen werden die gängigsten Bibliotheken vorgestellt und deren Vor- und Nachteile diskutiert. Die Code-Beispiele beziehen sich auf die Lösung des oft verwendeten einfachen XOR-Problems, bei dem die korrekte Lösung einfach die XOR-Funktion ist.

## Verwendete Bibliotheken

In diesem Unterkapitel werden die Unternehmensinternen verwendeten Bibliotheken vorgestellt.

### Pandas

### Scikit-learn

## Anforderungen zur Realisierung von neuronalen Netzen

Die wichtigste Anforderung für die Auswahl der Bibliotheken zur Realisierung von neuronalen Netzen ist die Nutzung in Python, da im Unternehmen diese als Programmiersprache zur Realisierung von statistischen Modellen verwendet wird. Python bietet die Möglichkeit des „Rapid Prototyping“ und ist gut mit „Apache Hadoop“ integrierbar. Eine einfache und verständliche Nutzung der Bibliotheken Lernen muss gegeben sein, da auch Entwickler ohne tiefgehende Erfahrungen in Maschinellem Lernen daran arbeiten. Ein weiterer wichtiger Faktor ist die Wartung der Bibliothek. Um zum einen mit Techniken auf aktuellen Stand und mit möglichst fehlerfreier Software zu arbeiten. Außerdem muss die Bibliothek über genügend Einstellungsmöglichkeiten verfügen. Besonders bei der Realisierung von neuronalen Netzen sollte es möglich sein verschiedene Aktivierungsfunktionen, Netztopologien und Lernregeln zu verwenden.

## PyBrain

PyBrain ist eine Deep-Learning-Bibliothek für Python, welche ausschließlich Modellierungen neuronaler Netze realisiert. Das Ziel der Bibliothek ist es, neuronale Netze möglichst einfach und flexibel zu bilden und einzusetzen.

#### Aufbau des neuronalen Netzes

Wie in Listing 1 zu sehen, lässt sich unkompliziert ein Netz mit der *buildNetwork()*-Klasse erzeugen.

1. n = buildNetwork(2,6,3, outclass=SoftmaxLayer )

Listing : Erstellung eines Netzes mit einem Hidden-Layer

Anhand der Übergabeparameter können Größe, Aktivierungsfunktion, usw. der einzelnen Schichten definiert werden.

#### Trainieren des neuronalen Netzes

Nachdem das Netz erstellt wurde, kann mithilfe der *BackpropTrainer*-Klasse die Lernregel und Lernrate zur Generalisierung angegeben werden.

1. t = BackpropTrainer(n, dataset=trndata,learningrate=0.01, momentum=0.1)
2. t.trainUntilConvergence(maxEpochs=200)

Listing :Trainieren eines Netzes in Pybrain

Ist der Trainer eingestellt, kann das eigentliche Lernen beginnen. Mit der *trainUnitlConvergence()*-Methode wird der Lernvorgang gestartet. Es kann die Anzahl der Epochen sowohl die verwendeten Daten bestimmt werden.

#### Auswertung der Daten

Ist das Lernen beendet, kann mit der *activateOnDataset()*-Methode auf ungesehene Daten eine Vorhersage getroffen werden. Die Methoden liefert die Zugehörigkeit einer Klasse als Wahrscheinlichkeit zurück.

1. activatedNetwork = n.activateOnDataset(evaldata)

Listing : Ausgabe der Klassifikationswahrscheinlichkeit in Pybrain

## Tensorflow

Tensorflow ist eine Deep-Learning-Bibliothek, die von Google entwickelt wurde. Mit der Bibliothek lassen sich numerische Berechnungen als Datenflussgraphen darstellen. Die Knoten des Graphen dienen als mathematische Operationen und entlang der Kanten fließen Daten als Tensoren.

#### Aufbau des neuronalen Netzes

Um ein Netz mit Tensorflow zu erstellen, müssen zuerst *Placeholder* deklariert werden, welche die In- und Output Schichten repräsentieren.

1. x = tf.placeholder("float", [None, n\_input])
2. y = tf.placeholder("float", [None, n\_output])

Listing : Erstellen von Placeholdern in Tensorflow

Über die *Placeholder* fließen später die Daten zu den entsprechenden Knoten bzw. Operationen. Zur Deklaration werden Datentyp und Dimension der Ein- und Ausgabedaten benötigt. Die Dimensionen sind abhängig von der Anzahl der Merkmale und Klassen. Sind die Schichten mit ihrer jeweiligen Größe initialisiert, müssen die Gewichte und die Bias als *Variable* definiert werden. *Variablen* sind feste Werte innerhalb des Datenflussgraphen.

1. weights = {
2. 'in': tf.Variable(tf.random\_normal([n\_input, n\_hidden\_1])),
3. 'hidden': tf.Variable(tf.random\_normal([n\_hidden\_1, n\_hidden\_2])),
4. 'out': tf.Variable(tf.random\_normal([n\_hidden\_2, n\_output]))
5. }
7. biases = {
8. 'in': tf.Variable(tf.random\_normal([n\_hidden\_1])),
9. 'hidden': tf.Variable(tf.random\_normal([n\_hidden\_2])),
10. 'out': tf.Variable(tf.random\_normal([n\_output]))
11. }

Listing : Einstellen der Gewichte und Bias in Tensorflow

Um die grundlegende Struktur des Netzes zu vervollständigen, müssen die Schichten, Gewichte, Bias und Aktivierungsfunktionen festgelegt werden. Tensorflow bietet eine Vielzahl an Aktivierungs-, Kosten- und Optimierungsfunktionen, welche in der Tensorflow-API nachgelesen werden können. Für das Beispiel wird eine Sigmoide-Aktivierungsfunktion in der Hidden-Schicht verwendet und in der Output-Schicht eine Softmaxfunktion.

1. **def** multilayer\_perceptron(\_X, \_weights, \_biases):
2. layer\_1 = tf.nn.sigmoid(tf.matmul(\_X, \_weights['h1']) + \_biases['b1'])
3. layer\_2 = tf.nn.softmax(tf.matmul(layer\_1, \_weights['h2']) + \_biases['b2'])
4. **return** tf.matmul(layer\_2, weights['out']) + biases['out']
6. pred = multilayer\_perceptron(x, weights, biases)

Listing : Erstellen der Schichten in Tensorflow

Nachdem alle Knoten und Aktivierungsfunktionen verbunden sind, muss das Netz durch die *initialize\_all\_variables()*-Methode initialisiert werden.

1. init = tf.initialize\_all\_variables()

Listing : Initialisierung des Netzes in Tensorflow

#### Trainieren des neuronales Netzes

Zum Trainieren des Modells muss für den Graph eine *Session* erstellt werden, welche den Graphen, die dazugehörigen Daten und Optimierungsfunktion als Übergabeparameter entgegen nimmt.

1. with tf.Session() as sess:
2. sess.run(init)
3. **for** epoch **in** range(training\_epochs):
4. avg\_cost = 0.
5. **for** i **in** range(total\_batch):
6. sess.run(optimizer, feed\_dict={x: data, y: label})

Listing : Trainieren des Netzes in Tensorflow

Um die *Placeholder* mit richtigen Daten zu „füttern“ wird der *run()-*Methode ein *feed\_dict* mit den Trainingsdaten und der entsprechenden Zielvariable übergeben.

#### Auswertung der Daten

Um in Tensorflow das Netz auf ungesehene Daten zu testen oder sich die Wahrscheinlichkeiten einer Klassenzugehörigkeit auszugeben, eignet sich die *eval()-*Methode. Dieser wird wie beim Training ein *feed\_dict* mit Daten übergeben und dazu die entsprechende Session, damit die Daten auf das richtige Netz angewendeten werden.

1. probabilities=y
2. **print** "probabilities", probabilities.eval(feed\_dict={x: test\_data}, session=sess)

Listing : Ausgabe der Wahrscheinlichkeiten in Tensorflow

## Theano

Theano ist eine Python Bibliothek zur Definition, Optimierung und Evaluierung mathematischer Ausdrücke, welche sich ideal für maschinelles Lernen eignet[].

#### Aufbau des Neuronalen Netzes

Um ein neuronales Netz in Theano zu erstellen, müssen zunächst die Ein- und Ausgangsschichten definiert werden. Dafür können von Theano gestellte Variablen bzw. Datentypen verwendet werden.

1. x = T.dvector()
2. y = T.dscalar()

Listing : Initialisierung Ein- und Ausgangsschicht

In diesem Codebeispiel ist die Variable x ein Vektor für die Eingabeschicht und die Variable y ein Skalar für die Ausgabeschicht.

1. **def** layer(x, w):
2. b = np.array([1], dtype=theano.config.floatX)
3. new\_x = T.concatenate([x, b])
4. m = T.dot(w.T, new\_x)
5. h = nnet.sigmoid(m)
6. **return** h
8. **def** grad\_desc(cost, theta):
9. alpha = 0.1
10. **return** theta - (alpha \* T.grad(cost, wrt=theta))
12. theta1 = theano.shared(np.array(np.random.rand(3,3), dtype=theano.config.floatX))
13. theta2 = theano.shared(np.array(np.random.rand(4,1), dtype=theano.config.floatX))
15. hidden\_layer = layer(x, theta1)
17. output\_layer = T.sum(layer(hid1, theta2))
18. fc = (output\_layer - y)\*\*2

Listing : Einstellen der Schichten, Aktivierungs- und Kostenfunktion in Theano

Sind die Variablen definiert, muss fest vorgebeben werden, wie viele Schichten, Merkmale und Klassen vorhanden sind. Für die jeweiligen Schichten muss die Aktivierungsfunktion und die Konnektivität vorgegeben werden. Zur Einfachheit dient dafür die *layer()-*Methode, welche die Eingabedaten und Gewichte entgegen nimmt. In der Methode wird zuerst ein Bias erzeugt und mit den Gewichten verbunden. Danach wird das Produkt aus den Gewichten und den Eingabedaten inklusive des Bias erzeugt und der Aktivierungsfunktion übergeben. Die Methode liefert zum Schluss die initialisierte Schicht zurück.

#### Trainieren des neuronales Netzes

Nachdem alle Schichten initialisiert wurden, kann das Netz trainiert werden. Hierfür muss eine Funktion generiert werden, welche den Trainingsvorgang beschreibt.

1. cost = theano.function(inputs=[x, y], outputs=fc, updates=[
2. (theta1, grad\_desc(fc, theta1)),
3. (theta2, grad\_desc(fc, theta2))])
5. **for** i **in** range(10000):
6. **for** k **in** range(len(inputs)):
7. cur\_cost = cost(inputs[k], output[k])
8. **if** i % 500 == 0:
9. **print**('Cost: %s' % (cur\_cost,))

Listing : Training des neuronalen Netzes in Theano

Theano bietet zur leichteren Beschreibung die *function*-Methode. Diese nimmt die am Anfang definierten Ein- und Ausgabeschichten, welche die Dimensionen der Schichten beschreibt. Dazu muss noch der Ausdruck der Ausgabeschicht und die Propagierungsregel übergeben werden. Zum trainieren des Netzwerkes wird die erstellte Funktion mit den Trainingsdaten aufgerufen.

#### Auswertung der Daten

Um anhand von ungesehenen Daten eine Vorhersage zu treffen, muss wieder eine Theano *function()* definiert werden. Dieser werden die ungesehenen Daten und der Ausdruck der Ausgabeschicht übergeben.

1. run\_forward = theano.function(inputs=[x], outputs=output\_layer)
3. **print**(run\_forward([0,1]))

Listing : Ausgabe der Klassifikation in Theano

Danach kann die Funktion mit den entsprechenden Daten aufgerufen werden. Sie liefert als Rückgabewert die Klassifikationswahrscheinlichkeit.

## Lasagne

Lasagne ist eine Python Bibliothek, welche Theano um relevante Funktionen erweitert und den Aufbau von neuronalen Netzen vereinfacht.

#### Aufbau des Neuronalen Netzes

Anstatt wie in Theano die verschiedenen Schichten selbstständig zu schreiben, können die vorgefertigten Schichten von Lasagne verwendet werden.

1. l\_in = lasagne.layers.InputLayer(shape=input.shape)
3. l\_hidden = lasagne.layers.DenseLayer(
4. l\_in,
5. num\_units=5,
6. nonlinearity=lasagne.nonlinearities.tanh)
7. l\_output = lasagne.layers.DenseLayer(
8. l\_hidden, num\_units=N\_CLASSES, nonlinearity=lasagne.nonlinearities.softmax)
10. net\_output = lasagne.layers.get\_output(l\_output)
11. true\_output = T.ivector('true\_output')
12. loss = T.mean(lasagne.objectives.categorical\_crossentropy(net\_output, true\_output))
14. all\_params = lasagne.layers.get\_all\_params(l\_output)
16. updates = lasagne.updates.sgd(loss, all\_params, learning\_rate=0.5)

Listing : Aufbau eines Netzes in Lasagne

Wie in dem Listing zu sehen ist, muss zur Initialisierung der Eingabeschicht nur die Anzahl der Merkmale übergeben werden. Um die Hidden-Schicht zu initialisieren, muss zuerst die vorherige Schicht übergeben werden, um diese miteinander zu verbinden. Außerdem die Anzahl der Neuronen sowie die Aktivierungsfunktion notwendig. Ein großer Vorteil von Lasagne ist die große Auswahl von Kostenfunktionen und Propagierungsregeln.

#### Trainieren des neuronalen Netzes und Auswerten der Daten

Wurde das Netz erstellt, muss eine *Theano-function()* definiert werden, in der die Daten, Kostenfunktion und die Propagierung übergeben wird.

1. train = theano.function([l\_in.input\_var, true\_output], loss, updates=updates)
2. get\_output = theano.function([l\_in.input\_var], net\_output)
4. **for** n **in** xrange(1000):
5. train(input, target)
6. y\_predicted = np.argmax(get\_output(input), axis=1)

Listing : Training und Vorhersage in Lasagne

Wie schon bei Theano kann dann zum Trainieren des Netzes die Funktion aufgerufen werden. Das selbige gilt beim Vorhersagen der Klassen.

## Scikit-neuralnetwork

Da die verschiedenen Bibliotheken wie Theano oder Tensorflow eine gewisse Lernkurve besitzen, um sie auf echte Projekte anzuwenden, gibt es diverse Wrapper, die die Nutzung erleichtern sollen ohne dabei Funktionalität einzubüßen.

1. nn = Classifier(
2. layers=[
3. Layer("Rectifier", units=100),
4. Layer("Softmax")],
5. learning\_rate=0.02,
6. n\_iter=10)
7. nn.fit(X\_train, y\_train)
9. y\_valid = nn.predict(X\_valid)

Listing : Erstellen, Trainieren und Vorhersage in Scikit-neuralnetwork

Das Listing zeigt ein simples erstellen und trainieren eines Netzes. Besonders hervorzuheben ist die einfache *Scikit-Learn-*ähnliche Syntax. Ohne großes Wissen über die Bibliotheken können diese direkt verwendet werden. Über die *Classifier*-Klasse werden die Schichten, Aktivierungsfunktionen, Propagierungsregeln und die Lernrate festgelegt. Über die *fit()*-Methode wird das erstellte Netz trainiert. Durch die *predict*-Methode kann auf ungesehenen Daten eine Vorhersage getroffen werden.

## Fazit

PyBrain bietet einen enorm schnellen Einstieg in die Nutzung von neuronalen Netzen in Python. Besonders die *buildNetwork()*-Methode lässt ziemlich simpel ein Netzwerk durch eine Zeile erstellen. Komplexere Netzwerke lassen sich durch ebenso geringen Aufwand erstellen. Leider wird an der Bibliothek seit 2009 nicht mehr weiterentwickelt, welches spürbar an der Wartbarkeit zu erkennen ist. Dadurch ist die Bibliothek trotz seiner Vorteile nicht für den Produktiven gebrauch geeignet. Tensorflow besitzt eine ausführliche Dokumentation die einen einfachen Einstieg ermöglicht. Auch für Nutzer die keine große Erfahrung in maschinellem Lernen besitzen, sind ausreichende Dokumentationen verfügbar. Ein weiter Vorteil von Tensorflow ist, dass Berechnungen auf der GPU ausgeführt werden können und somit die Performanz stark verbessert wird.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Einfachheit | Wartbarkeit | Performanz | Einstellungsmöglichkeiten |
| Theano | 3 | 5 |  | 5 |
| Tensorflow | 3-4 | 5 | 5 | 5 |
| Lasagne | 4 | 5 |  | 5 |
| Scikit-neualnetwork | 5 | 4 |  |  |
| PyBrain | 5 | 0 (Wird nicht mehr aktiv Entwickelt) |  | 4 |

Tabelle : Bewertungstabelle der verschiedenen Bibliotheken für neuronale Netze

# Aktueller Stand der Entwicklung

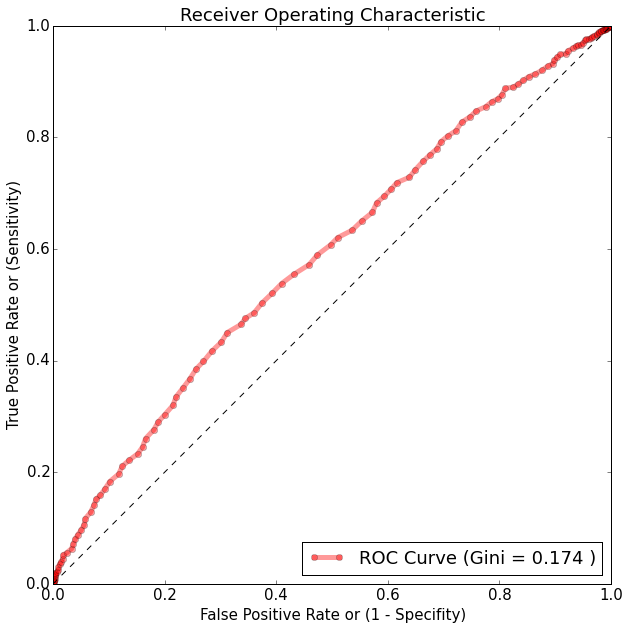
In diesem Kapitel wird der aktuelle Stand der Entwicklung beschrieben. Es wird gezeigt welches Klassifizierungsverfahren zur Lösung der OTB-Problematik verwendet wurde. Genauer werden die verwendeten Parameter und Merkmale beschrieben, sowie den vorhandenen Datensatz. Danach wird der aktuelle Stand mithilfe von verschiedenen statistischen Gütekriterien bewertet. Zum Schluss des Kapitels werden die Probleme der Klassifizierung diskutiert.

## Erster Lösungsansatz

Für die Lösung der OTB-Problematik wird als Programmierungssprache Python verwendet, da sie viele statistische Open-Source Bibliotheken frei zu Verfügung stellt. Als Klassifikationsverfahren wird der Random Forests Algorithmus gewählt. Ausschlaggebend für die Verwendung des Verfahrens ist seine Parallelisierbarkeit der Evaluierung, seine Effizienz in der Trainingszeit hinsichtlich der größer der Datenmenge. Zur Realisierung des Problems wird das Python-Paket „Scikit-learn„ verwendet, welche eine Implementierung des Random Forests bereitstellt.

## Verwendeter Datensatz

## Ergebnis



# Umsetzung

In Kapitel 4 wird die OTB-Klassifizierung mit verschiedenen Klassifizierungsverfahren angewandt und verglichen. Es wird beschrieben wie die einzelnen Verfahren aufgebaut und parametrisiert werden. Dabei wird die bereits vorhandene Random Forest Klassifizierung mittels Oversampling und Anpassung der Hyperparameter optimiert. Desweiteren werden die gewonnen Ergebnisse mit dem alten Stand verglichen. Zum Schluss werden die verschiedenen Klassifizierungsverfahren an das OTB-Problem angewandt und die einzelnen Verfahren untereinander verglichen.

## Verbesserungsansatz Random Forest

### Sampling

Um die verschiedenen Sampling Methoden möglichst aussagekräftig zu Evaluieren werden verschiedene Gütemaße verwendet. Für die Evaluierung werden die Ersteinkaufsdaten von 2014 genommen. Der Datensatz wird in einen Trainingsdatensatz und Evaluierungsdatensatz die im Verhältnis 80:20 aufgeteilt werden. Um den Trainingsdatensatz in ein ausgewogenes Gleichgewicht zu bringen, werden auf diesen die vorgestellten Sampling-Methoden angewendet. Der Random Forest wird mittels Kreuz-Validierung trainiert. Der unveränderte Evaluierungsdatensatz wird zum Testen und Evaluieren des Klassifizierer verwendet.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ACC | F1 | Precision | METT | Recall | AUC |
| Random Forest | 0,74 | 0,66 | 0,67 |  | 0,75 |  |
| RF mit SMOTE | 0,73 | 0,68 | 0,67 |  | 0,74 |  |
| RF mit ADASYN | 0,73 | 0,69 | 0,68 |  | 0,73 |  |
| RF mit Tomek Links | 0,74 | 0,68 | 0,69 |  | 0,75 |  |
| RF mit SMOTE und Tomek Links | 0.708 | 0.69 | 0.68 |  | 0.71 |  |
|  |  |  |  |  |  |  |

Tabelle : Ergebnisse der verschiedenen Sampling-Methoden

## Verwendung von neuronalen Netzen

## Verwendung von SVM

## Verwendung von Logistischer Regression

# Ensemble Klassifikation

Im diesem Kapitel werden die verschiedenen Modelle des 6. Kapitels zur Vorhersage der Wiedereinkaufwahrscheinlichkeit kombiniert.

# Evaluierung

# Fazit

Abkürzungsverzeichnis

**ABK** Abkürzung

**OTB** One-time-buyers

**SVM** Support Vector Maschines

**KNN** Künstliche Neuronale Netze

**ROC** Receiver Operating Characteristic

**AUC** Area Under the Curve

**ADASYN** Adaptive Synthetic Sampling Approach

**SMOTE** Synthetic Monority Oversampling Technique

Tabellenverzeichnis

[Tabelle 1: Wahrheitstabelle 10](#_Toc440025067)

[Tabelle 2: Wahrheitsmatrix der OTB-Klassifizierung 23](#_Toc440025068)

[Tabelle 3: Ergebnisse der verschiedenen Sampling-Methoden 27](#_Toc440025069)

Abbildungsverzeichnis

[Abbildung 2.1: Support Vector Machine [Buchquelle] 4](#_Toc440025056)

[Abbildung 2.2: Logistische Regression anhand eines sigmoiden Kurvenverlaufes 6](#_Toc440025057)

[Abbildung 2.3: Beispiel einer linearen Regression 6](#_Toc440025058)

[Abbildung 2.4: Aufbau eines neuronalen Netzes 7](#_Toc440025059)

[Abbildung 2.5: Binäre Aktivierungsfunktion 8](#_Toc440025060)

[Abbildung 2.6: Lineare Aktivierungsfunktion 8](#_Toc440025061)

[Abbildung 2.7: Sigmoide Aktivierungsfunktion 8](#_Toc440025062)

[Abbildung 2.8: Direkte Rückkopplung eines Neurons 9](#_Toc440025063)

[Abbildung 2.9: Indirektes Feedback eines neuronales Netzes 9](#_Toc440025064)

[Abbildung 2.10: Seitliche Rückkopplung innerhalb einer Schicht 9](#_Toc440025065)

[Abbildung 2.11: ROC – Kurve 13](#_Toc440025066)

Literaturverzeichnis

|  |  |
| --- | --- |
| [1] | M. Kornmeier, Wissenschaftlich schreiben leicht, 4. Hrsg., UTB, 2011. |