Master-Thesis

Studiengang Medizinische Informatik Master

Fakultät für Informatik

Hochschule Mannheim

15.12.2015

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

Betreuer: Prof. Ivo Wolf, Hochschule Mannheim

Zweitkorrektor: Jurgis Pods, Qudosoft GmbH & Co. KG

**Vergleich, Optimierung und Evaluation 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, 15.12.2015

Nicholas Pastuovic

Inhalt

[1 Einleitung 1](#_Toc437949239)

[1.1 Motivation und Zielsetzung 1](#_Toc437949240)

[1.2 Aufteilung der Arbeit 2](#_Toc437949241)

[2 Grundlagen 3](#_Toc437949242)

[2.1 Klassifikationsverfahren 3](#_Toc437949243)

[2.1.1 Random Forests 3](#_Toc437949244)

[2.1.2 Support Vector Machine 3](#_Toc437949245)

[2.1.3 Logistische Regression 4](#_Toc437949246)

[2.1.4 Künstliche Neuronale Netze 4](#_Toc437949247)

[2.2 Maschinelles Lernen 5](#_Toc437949248)

[2.2.1 Überwachtes Lernen 5](#_Toc437949249)

[2.2.2 Unüberwachtes Lernen 5](#_Toc437949250)

[2.2.3 Bestärkendes Lernen 5](#_Toc437949251)

[2.3 Bewertung 5](#_Toc437949252)

[2.3.1 Cross Validation 6](#_Toc437949253)

[2.3.2 ROC-Kurve 6](#_Toc437949254)

[2.3.3 F-Score 6](#_Toc437949255)

[2.4 Verwendete Frameworks 6](#_Toc437949256)

[2.4.1 Pandas 6](#_Toc437949257)

[2.4.2 Scikit-learn 6](#_Toc437949258)

[2.5 Gütemaße 6](#_Toc437949259)

[2.5.1 F1-Score 6](#_Toc437949260)

[2.5.2 Precision 6](#_Toc437949261)

[2.5.3 Recall 6](#_Toc437949262)

[2.5.4 Accuracy 7](#_Toc437949263)

[3 Bibliothekauswahl für neuronale Netze 8](#_Toc437949264)

[3.1.1 PyBrain 8](#_Toc437949265)

[3.1.2 Tensorflow 9](#_Toc437949266)

[3.1.3 Theano 11](#_Toc437949267)

[3.1.4 Lasagne 13](#_Toc437949268)

[3.1.5 Scikit-neuralnetwork 14](#_Toc437949269)

[3.2 Fazit 14](#_Toc437949270)

[4 Aktueller Stand der Entwicklung 16](#_Toc437949271)

[4.1 Erster Lösungsansatz 16](#_Toc437949272)

[4.2 Verwendeter Datensatz 16](#_Toc437949273)

[4.3 Ergebnis 17](#_Toc437949274)

[4.4 Unausgeglichene Daten 17](#_Toc437949275)

[4.4.1 Over- und Undersampling 17](#_Toc437949276)

[5 Umsetzung 20](#_Toc437949277)

[5.1 Verbesserungsansatz Random Forest 20](#_Toc437949278)

[5.2 Verwendung von neuronalen Netzen 20](#_Toc437949279)

[5.3 Verwendung von SVM 20](#_Toc437949280)

[5.4 Verwendung von Logistischer Regression 20](#_Toc437949281)

[6 Multiple Klassifikationsverfahren 21](#_Toc437949282)

[7 Evaluierung 22](#_Toc437949283)

[8 Fazit 23](#_Toc437949284)

[Abkürzungsverzeichnis iv](#_Toc437949285)

[Tabellenverzeichnis v](#_Toc437949286)

[Abbildungsverzeichnis vi](#_Toc437949287)

[Literaturverzeichnis vii](#_Toc437949288)

# 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 80% 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

In Kapitel 2 werden die Grundlagen verschiedener Klassifikations-Algorithmen, statistische Modelle und alle verwendeten Frameworks vorgestellt. In Kapitel 3 werden die verwendeten Daten und die charakteristischen Merkmale erläutert. Desweiteren wird auf den aktuellen Stand der Entwicklung eingegangen. Es wird genau beschrieben mit welcher prozentualen Genauigkeit eine Vorhersage getroffen wird.

Im vierten Kapitel wird die OTB-Problematik an verschiedenen Klassifikationsverfahren angewandt. Mithilfe von Kreuzvalidierungsverfahren sollen die einzelnen Parameter der Verfahren bestmöglich eingestellt werden. In Kapitel 5 werden die Verfahren mithilfe von einem Dashboard untereinander verglichen. Hierzu werden z.B. Receiver Operating-Characteristic-Kurven (ROC) zur Beurteilung der Klassifikatoren verwendet.

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

## 

# Grundlagen

## Klassifikationsverfahren

Klassifikationsverfahren beschreiben den Prozess gegebene Objekte in Klassen einzuteilen. Zu welcher Klasse ein Objekt gehört wird über dessen Merkmale entschieden. Im Weiteren werden verschiedene Verfahren zur Lösung des Problems vorgestellt.

### Random Forests

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

#### Bagging

### Support Vector Machine

Support Vector Machine (SVM) ist ein supervised learning Verfahren, welches versucht eine Menge von Daten in Klassen zu unterteilen. Dabei wird eine Hyperebene, die als Trennlinie dient, so gelegt, dass sie die verschiedenen Klassen optimal trennt.

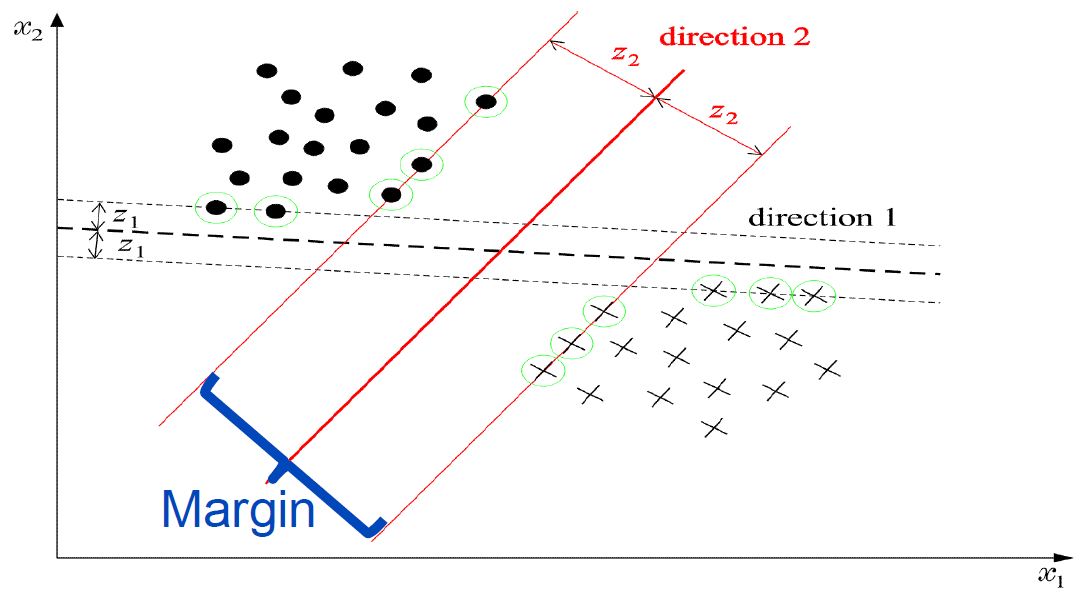


Abbildung : Support Vector Machine [Buchquelle]

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, werden Support Vectors genannt. In Abbildung 1 sind sie grün umkreist zu sehen. Um einen eine Optimale Trennung zu erreichen, wird der Abstand zwischen den Support Vectors der verschiedenen Klassen berechnet und die Hyperebene daraufhin so gelegt, dass der Abstand der Support Vectors zur Hyperebene möglichst groß ist.

#### Nicht linear trennbare Hyperebenen

### Logistische Regression

### Künstliche Neuronale Netze

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

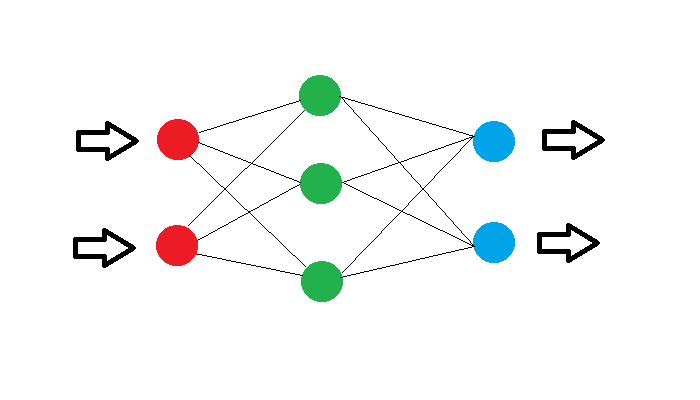


Abbildung : Neuronales Netz

## Maschinelles Lernen

### 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.

## Gütemaße

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

### Cross Validation

### ROC-Kurve

Mit ROC-Kurven lassen sich statistische Modelle Optimieren und Bewerten.

### F-Score

s

### Precision

### Recall

### Accuracy

## Verwendete Frameworks

In diesem Unterkapitel werden alle, für diese Arbeit relevanten Frameworks vorgestellt. Es wird beschrieben welche Anwendungsfälle die einzelnen Frameworks abdecken.

### Pandas

### Scikit-learn

Das

# Bibliothekauswahl für neuronale Netze

In diesem Kapitel wird die Auswahl der Bibliotheken zur Realisierung des neuronalen Netzes diskutiert. Mittels kleiner Code-Beispielen werden die gängigsten Bibliotheken vorgestellt und dessen Vor- und Nachteile diskutiert. Die Code-Beispiele beziehen sich alle zur Lösung der XOR-Problematik.

### PyBrain

PyBrain ist eine Deep-Learning-Bibliothek für Python, welche ausschließlich Modellierungen neuronaler Netze realisiert. Das Ziel der Bibliothek ist es möglichst einfach und flexibel neuronale Netze 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 neuronales 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 realisieren. 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 die 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 für die Definierung, 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 vorgegeben werden und wie sie jeweils verbunden sind. 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 neuronales 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, 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 dessen Aktivierungsfunktion. Ein großer Vorteil von Lasagne ist die große Auswahl von Kostenfunktionen und Propagierungsregeln.

#### Trainieren des neuronales 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 Funktionalitäten 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, Propagierungsregel und Lernrate festgelegt. Über die *fit()*-Methode wird das erstellte Netz trainiert. Durch die *predict*-Methode kann auf ungesehene 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. Unglücklicherweise wird an der Bibliothek seit 2009 nicht mehr weiterentwickelt, welches spürbar an der Performanz zu erkennen ist. Dadurch ist die Bibliothek trotz seiner Vorteile nicht für den Produktiven gebrauch geeignet.  
Tensorflow ist

Theano

Die Wrapper-Bibliothek *Scikit-neuralnetwork* bietet einen schnellen und Produktiven Einsatz von neuronalen Netzwerken.

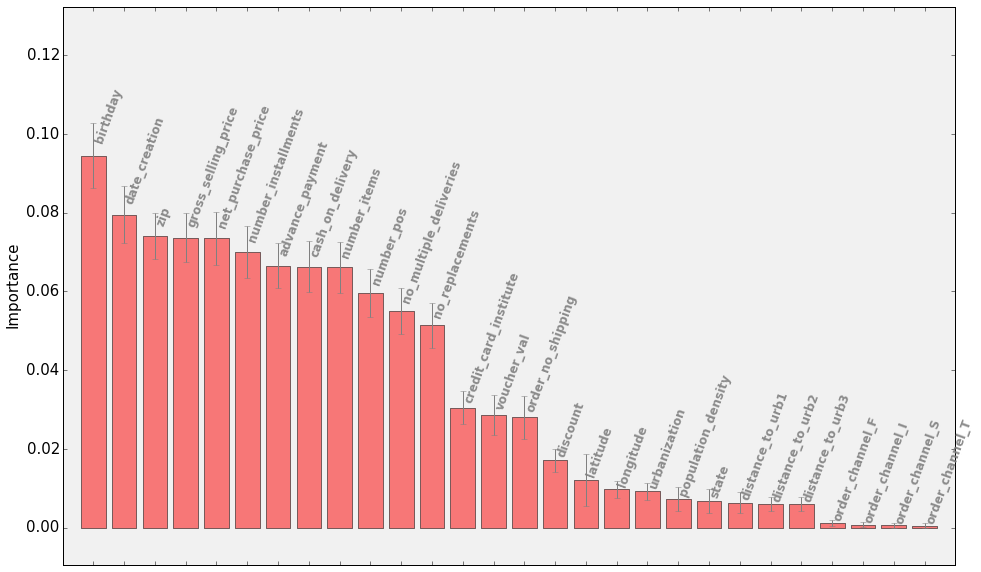
# Aktueller Stand der Entwicklung

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.

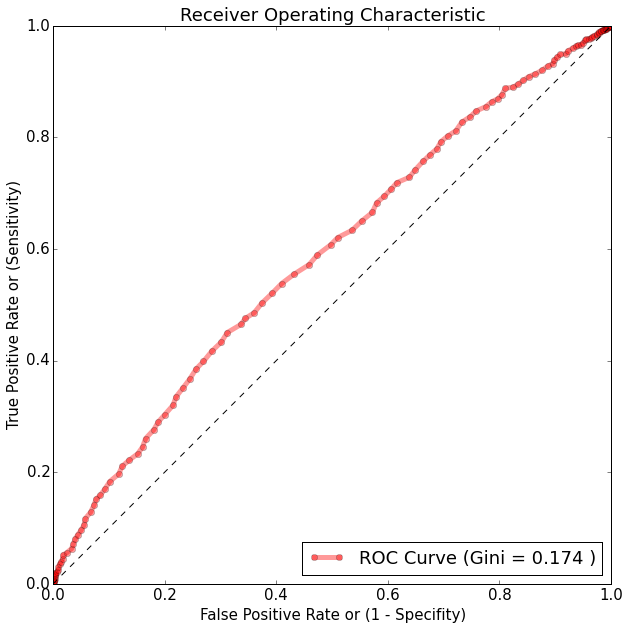
## Erster Lösungsansatz

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



## Unausgeglichene Daten

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 | Person ist OTB | Person ist MTB |
| Klassifikation OTB | 174 (true Positive) | 3523 (false Positive) |
| Klassifikation MTB | 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*(Kunde ist ein OTB wird aber als MTB eingestuft) sehr hoch. Dieser Fehler ist besonders schwerwiegend, da hohe Kosten ohne einen Nutzen entstehen. Es fallen für einen Kunden Werbekosten an, der nicht mehr im Versandhaus einkaufen wird. 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. die *Neighborhood-Cleaning-Rule[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.

#### SMOTE

SMOTE (Synthetic Monority 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-nearest neighbors* Algorithmus synthetische Daten erzeugt.

#### Verfahren

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.

#### Verfahren

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.

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

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

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

Zum Schluss werden die neuen synthetischen Daten wie beim SMOTE-Verfahren berechnet (1).

# 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. Danach werden die verschiedenen Klassifizierungsverfahren an das OTB-Problem angewandt. Zum Schluss werden die einzelnen Verfahren untereinander verglichen.

## Verbesserungsansatz Random Forest

## Verwendung von neuronalen Netzen

## Verwendung von SVM

## Verwendung von Logistischer Regression

# Multiple Klassifikationsverfahren

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

**ADASYN** Adaptive Synthetic Sampling Approach

**SMOTE** Synthetic Monority Oversampling Technique

Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

Abbildungsverzeichnis

[Abbildung 1: Support Vector Machine [Buchquelle] 2](#_Toc433632597)

Literaturverzeichnis

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