technische universität
dortmund

Bachelorarbeit

Erkennung ungewöhnlicher Textbelege in großen Textkorpora mittels minimum enclosing Balls

Jonas Langenberg
Juli 2015

Gutachter:
Prof. Dr. Katharina Morik
Dipl.-Inf. Christian Pölitz

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhlbezeichnung (LS-8)
http://www-ai.cs.uni-dortmund.de/index.html

In Kooperation mit:
Informatik
Lehrstuhl-08/Künstliche Intelligenz

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Die Disambiguierung von Wörtern ist ein wichtiges Thema im Data Mining in natürlichsprachlichen Umgebungen. Dabei werden vor allem besondere Eigenschaften der Sprache oft außer Acht gelassen. Darunter fallen unter anderem Metaphern. Als Beispiel seien folgende Sätze gegeben:

Das ist doch *Schnee* von *Gestern*.
Gestern lag aber eine Menge *Schnee*.

Beide Sätze haben einen ähnlichen Aufbau. In Beiden kommen die Wörter *Schnee* und *Gestern* vor und trotzdem haben sie eine ganz Unterschiedliche Bedeutung. Der Schnee von Gestern kann einerseits eine Neuigkeit sein, andererseits auch eine Beschreibung des Wetters vom Vortag. Nun ein Modell anzulernen das den Unterschied zwischen den beiden Anwendungen von *Schnee* versteht ist eine Herausforderung, denn durch eine reine Vorkommensanalyse der Wörter ist der Unterschied zwischen den Sätzen nicht zu erkennen. Dafür werden im Folgenden verschiedene Features von Texten untersucht, an denen Metaphern erkannt werden können.
Es ist also klar, dass Wörter und Präpositionen in ungewöhnlichen Bedeutungen existieren und dass diese ein Problem für die herkömmliche Analyse von Textdokumenten darstellen. Im Nachfolgenden soll ein Verfahren untersucht werden um diese ungewöhnlichen Textbelege effizient zu erkennen. Die in der aktuellen State of Art verwendete One-Class SVM ist für sehr große Textkorpora ineffizient. Das liegt vor Allem an der Auswirkung der verschiedenen Kernel Methoden und der gewählten Repräsentation der Daten [9] (Dazu später mehr). Deshalb wird in dieser Arbeit das Verhalten der Ball Vektor Maschine (im nachfolgenden auch BVM) bei einem Einklassenproblem, wie der Erkennung von Metaphern, untersucht. Diese ist um einiges effizienter auf großen Datenmengen als herkömmliche SVM Methoden. Es soll also untersucht werden ob sich die Ball Vektor Maschine auch für Ausreißererkennung eignet.

1.2 Aufbau der Arbeit

Um das im vorherigen Kapitel erwähnte Problem der Performance auf großen Dokumenten zu lösen, wird die Verwendung von *Minimum Enclosing Balls* vorgeschlagen und in verschiedenen Experimenten die Performance einer BVM mit der einer One Class SVMs verglichen. Zuerst wird ein Einblick in verwandte Arbeiten gegeben und eine Übersicht weiterer Verfahren um mit ungewöhnlichen Textbelegen und Metaphern im NLP zu arbeiten. Anschließend wird im 3 Kapitel ein Überblick über die Outlier Detection gegeben und anhand von kurzen Beispielen das Prinzip und die Verbindung zu den hier gesuchten Metaphern klar gemacht. In Kapitel 4 werden dann verschiedene Repräsentationsarten von Texten erklärt, aus denen dann die Eigenschaften besser herausgestellt werden können. Im Nachfolgenden soll klarer gemacht werden, welche Eigenschaften eines Textes für die vorgeschlagene Methode von Bedeutung sind. Anschließend werden im 5. Kapitel die verwendeten Kernel Methoden aufgezählt und erklärt. Dabei wird auch ein Augenmerk auf die im vorherigen Kapitel angedeuteten Eigenschaften der Texte und ihren Zusammenhang mit den verschiedenen Kernel Methoden gelegt. In Kapitel 6 wird ein Einblick in die Funktionsweise von Support Vektor Maschinen gegeben und ihr mathematischer Hintergrund erläutert sowie der Unterschied zwischen den verschiedenen SVM Arten erklärt. Außerdem wird noch einmal genauer die für die Methode relevante One Class SVM beschrieben. In Kapitel 7 wird das verwendete Verfahren der Enclosing Balls erläutert und ein Einblick in die Funktionsweise einer Core Vektor Maschine gegeben. In 8 Kapitel werden dann die durchgeführten Experimente beschrieben und ihre Ergebnisse aufgeführt. Im letzten Kapitel wird dann ein Fazit über die Methode, anhand der erzielten Ergebnisse gegeben. Darüber hinaus werden offen gebliebene Fragen geklärt und ein Ausblick auf weitere Arbeiten gegeben.

Kapitel 2

Verwandte Arbeiten

Das folgende Kapitel soll einen Überblick über ähnliche Arbeiten im selben Themenbereich geben. Die dabei zitierten Werke behandeln entweder ein verwandtes Thema und wurden für diese Arbeit verwendet oder bieten einen anderen Blickwinkel auf die Erkennung von ungewöhnlichen Textbelegen.

Zur Erkennung von ungewöhnlichen Textbelegen gibt es viele Arbeiten und Pilotstudien. In diesem Kapitel sollen einige ähnliche Arbeiten vorgestellt werden. Außerdem soll ein Einblick in andere Methoden und Forschungen in diesem Bereich gegeben werden. Eine der erwähnten Pilotstudien ist die Forschung von A. Feldman und J. Peng. In ihrer Arbeit [1] wird das Verhalten von Idiomen untersucht. Das dabei verwendete Verfahren ähnelt dem in dieser Arbeit vorgeschlagenem. In diesem Ansatz wird ein Text als *fehlerbehafteter* Datensatz aufgefasst. Es wird davon ausgegangen, dass in zusammenhängenden Textbelegen Wörter semantisch miteinander verbunden sind. Ähnlich wie in dem hier vorgestellten Ansatz werden diese Fehler dann durch eine Outlier Detection erkannt und somit als Idiome klassifiziert. Der Unterschied dabei ist, dass die in dieser Arbeit vorgestellten Vorgehensweise den Text noch durch Kernelmethoden auf verschiedene Features reduziert und außerdem eine Ball-SVM zur Klassifizierung einsetzt.
Eine etwas allgemeinere Arbeit ist *Learning to classify text using support vector machines* von T. Joachims [7]. In ihr wird eine Einführung in verschiedene Methoden und Algorithmen zur Klassifikation gegeben, wobei ein besonderes Augenmerk auf die Anwendbarkeit im Hinblick auf Textdokumente gegeben ist. Im Gegensatz zu dem hier vorgeschlagenen Verfahren werden aber besondere Belege wie Metaphern außen vorgelassen. Trotzdem stellt es einen guten Einführungstext in die allgemeinen Schwierigkeiten der Textklassifikation dar.
Eine weitere für diese Vorgehensweise interessante Arbeit ist *A Computational Model of Metaphor Interpretation* von James H. Martin [11]. Anders als in der oben erwähnten Arbeit geht es jetzt um die Interpretation von Metaphern. Natürlich ist eine Erkennung eben

dieser Belege in denen Metaphern verwendet werden die Grundlage für eine erfolgreiche Interpretation. Dennoch ist es interessant einmal zu sehen, was der nächste Schritt nach der erfolgreichen Erkennung beinhaltet. Dafür wird in der oben zitierten Arbeit das *MIDAS*(Metaphor Interpretation, Denotation, and Acquisition System) eingeführt, welches erfolgreich als Teil des UC consulting systems verwendet wird. Hierbei werden Metaphern mit bekanntem Wissen verknüpft und so für die Anwendung in einem NLP System nutzbar gemacht.
Etwas anders geht auch A. Kyriakopoulou vor. In ihrer Arbeit [8] wird die Anwendbarkeit von Clustering Methoden in der Textklassifikation untersucht. Ähnlich wie in der vorher genannten Arbeit werden auch hier keine besonderen Textbelege im speziellen untersucht. Clustering Methoden werden oft angewendet um Eigenschaften besser zusammenzufassen, was bei den umfangreichen Feature Räumen in der Text Klassifikation natürlich einen großen Vorteil mit sich bringt. Dabei wird die Performance von verschiedenen Methoden wie k-means und unterschiedlichen SVM Modellen an Textdokumenten untersucht.
Das Werk *Metapher Detection with Cross-Lingual Model Transfer* [21] befasst sich hingegen direkt mit der Problematik der Erkennung von Metaphern. Interessant ist hier vorrangig das Aufzeigen der auf die unterschiedlichen Verwendungen von Metaphern in verschiedenen Sprachen. Neben Englisch werden auch Testdatensätze aus dem spanischen, persischen und russischen für die Resultate benutzt. Dabei werden die semantischen Eigenschaften von Wörtern der englischen Sprache für das Modell verwendet und durch einen model transfer approach in ein zweisprachiges Dictionary übertragen. Im Gegensatz zu den bisher vorgestellten Methoden liegt das Augenmerk dieser Arbeit also auf den semantischen Eigenschaften der Sprachen. Zur Erstellung des Modells wird außerdem keine SVM sondern ein Random Forest benutzt.
Eine sehr ähnliche Arbeit ist *Catching Metaphors*[12]. In ihr wird eine Methode beschrieben um Metaphern aus bestimmten Wortbereichen zu erkennen. Dafür wird ein Maximum Entropy Klassifikator benutzt. Im Speziellen ist es der *Stanford conditional log linear (aka maxent) classifier* . Anders als in dieser Arbeit werden jedoch nur Metaphern aus bestimmten, vorher eingeteilten Wortbereichen gesucht. In diesem Falle *spatial motion*, *manipulation* und *health*. Zwar wird festgestellt das Wörter aus diesen Daten zu 90% metaphorisch benutzt werden, jedoch zählt dieses Ergebniss nur auf einem Textkorpora des *Wall Street Journals*. In dieser Arbeit sollen stattdessen Textbelege aus allen Bereichen gewählt werden.

Der Artikel *Metaphor Detection through Term Relevance* [10] erforscht hingegen die allgemeine Erkennung von Metaphern ohne vorheriges Wissen über Wortbereiche oder anderweitige Einschränkungen zu haben. Dabei gehen sie, ähnlich wie in dieser Arbeit vorgeschlagen, mit Hilfe der Ausreißererkennung vor. Metaphern sollen anhand einer ungewöhnlicher Benutzung gefunden werden. Im Rahmen dieses Artikels wird herausgefunden, dass die Metaphern immer schwerer als Ausreißer zu erkennen sind, desto größer die Traingingsdatenmenge wird. Dieses Fazit könnte für die weiteren Untersuchen in dieser Arbeit interessant sein.
Ein allgemeines Nachschlagewerk für das Verhalten von anderen Klassifizierungsmethoden ist die Website von Sebastian Raschka [15]. Auf ihr wird die Funktion des Naiven Bayes Klassifikators auf Textdokumenten erläutert. Der dort verwendete Bag of Words Ansatz ähnelt dem in dieser Arbeit vorgeschlagenem. Er wird besonders in der Spam Erkennung eingesetzt. Da aber kompliziere Features schlecht durch den Naiven Bayes bearbeitet werden können, bleibt die Verwendung eines Naiven Bayes Klassifikators für die Erkennung von Metaphern aus.
Die letzten beiden Arbeiten befassen sich mehr mit der Metapher Theorie. Der Artikel *Metaphern als strenge Wissenschaft* [6] versucht die Probleme der Metaphererkennung aufzuzählen. Vor allem die Eigenschaften einer Metapher spielen hier eine große Rolle. Metaphern können von einfachen Wortsubstitutionen bis hin zu semantischen Übertragungen ein breites Spektrum an Funktionen innehaben. Außerdem wird die Frage nach dem Ort der Informationen untersucht. Also in welchem Teil eines Textdokumentes die Informationen über den Gebraucht der Metapher zu finden sind und welcher Teil Noise darstellt. Des weiteren werden verschiedenste Probleme der Sprachentwicklung angesprochen. Darunter fallen unter anderem Bedeutungverschiebungen und Beziehungen zwischen einzelnen Wörtern. Insgesamt gibt diese Arbeit keine Lösung für die aufgeführten Probleme, viel mehr stellt sie eine Zusammenfassung über die Problematik von Metaphern im NLP dar.
Einen tieferen Einblick in die Theorie hinter Metaphern liefert das Buch *Analytische Theorien der Metapher* [13]. Es ist ein Sammelwerk der linguistischen Forschung im Hinblick auf Metaphern und ihre Bedeutung. Dabei werden sowohl die Entwicklung der Sprache, also auch verschiedene Interpretationsmodelle betrachtet.
Insgesamt bleibt anzumerken, dass die oben genannten Werke oft nur auf bestimmten Textkorpora arbeiten oder Probleme in den Bereichen der Performance und Accuracy haben. Deshalb ist diese Arbeit entstanden, in der ein neues Modell zur Erkennung ungewöhnlicher Textbelege vorgeschlagen werden soll. Dabei wird eine spezielle Art der Support Vektor Maschine, die Ball-SVM, mit der state of art One Class SVM verglichen.Im Besonderen wird dabei auch auf die Rolle verschiedener Kernel Methoden und Eigenschaften von Textdokumenen behandelt.

Kapitel 3

Outlier Detection

Ausreißererkennung oder Outlier Detection ist ein Sammelbegriff für verschiedene Verfahren zur Detektion von Anomalien. Darunter fallen sowohl Fehler in Datensätzen als auch besondere Belege und sporadisch vertretene Klassen. Beim Natural Language Processing (im Nachfolgenden NLP) werden sehr große Datenmengen verarbeitet. Im Speziellen sind die hier gesuchten Metaphern nur ein extrem kleiner Teil eines großen Datensatzes. Für standard Klassifikationsmodelle können eben diese kleinen Daten ein Problem darstellen. Oft sind genau diese wenigen Daten ein Ausreißer aus dem gesamten Datensatz, da sie sich stark von den anderen Daten unterscheiden. Meistens sind diese Daten Fehler oder *Noise*. Um also ein gutes Modell lernen zu können ist es wichtig diese Daten vor dem Training zu erkennen und auszusortieren. Im Falle des NLP können solche Daten aber auch Eigenheiten im Sprachgebrauch darstellen, wie z.B Metaphern und oder Redewendungen. Andere Bereiche in denen Outlier Detection angewendet werden sind die Erkennung von falschen Kreditkarten, Wettervorhersagen oder die Performance Analyse von Athleten, um nur einige Beispiele zu nennen. Um diese Daten also Effizient verarbeiten zu können wird das Prinzip der *Outlier Detection* angewendet.
Es wird davon ausgegangen, dass es für alle Daten einen Mechanismus gibt der sie erstellt hat. Da die Ausreißer sich stark von den anderen Daten unterscheiden wird also davon ausgehend das es für die Erstellung der Ausreißer einen anderen Mechanismus gibt [2]. Zur Erkennung von Ausreißern gibt es mehrere Methoden die sich in zwei Bereiche unterteilen lassen, *univariate* Methoden und *multivariate* Methoden. Darüber hinaus wird noch zwischen *parametrischen* (statistischen) und *nicht-parametrischen* Methoden unterschieden. Zum Bereich der *nicht-parametrischen* Methoden zählen insbesondere Data Mining Methoden wie SVM Klassifizierung. Diese werden auch als distanzbasierte Methoden bezeichnet und kommen besonders bei großen Datensätzen mit hochdimensionalen Feature Räumen zum Einsatz. Da im Natural Language Processing oft große Textkorpora als Datensätze benutzt werden, sind diese Methoden für die später vorgeschlagenen Experimente ideal. Dabei kann bei den Data Mining Methoden zwischen drei Methoden unterschieden

werden : Clustering, Raum- und distanzbasierte Methoden.
Beim Clustering werden auf Grundlage von Texteigenschaften Klassen(auch Cluster genannt) erstellt. Dabei werden Cluster mit geringem Datenaufkommen (auch einzelne Datenpunkte) als Ausreißer klassifiziert/gekennzeichnet. Oft benutzte Methoden aus dem Clustering Bereich sind *partitioning around medoids* (PAM) und *clustering large applications* (CLARA) [Kaufman und Rousseeuw, 1990). Da ihre Anwendungsfelder aber auf das Finden von Clustern spezialisiert sind, eignen sie sich meistens nicht optimal für Ausreißererkennung.
Eng damit verwand sind die Raum-Methoden. Ein Ausreißer in einer Raum-Methode ist ein Objekt dessen sekundäre Attribute sich stark von den anderen Objekten des Datensatzes unterscheiden [2]. Da zur Findung solcher Ausreißer oft graphische Methoden oder quantitative Tests verwendet werden, eignen sie sich auch nicht für die Ausreißererkennung in Textdokumenten.
Bei den distanzbasierten Methoden wird versucht einen Ausreißer anhand des Abstandes zum nächsten Nachbarn oder dem gesamten Datensatz zu ermitteln. Eng damit verwandt ist die Ausreißererkennung durch die Bildung eines Balls, welche in den später folgenden Experimenten untersucht werden soll.

Kapitel 4

Text Repräsentation

Dadurch, dass Metaphern selten auftreten und die untersuchten Textedaten meist sehr Umfangreich sind, spielen die Features der Daten eine große Rolle. Es ist wichtig, dass die Eigenschaften der Texte sinnvoll repräsentiert werden. Dafür kann ein Text auf verschiedene Arten dargestellt werden. Im Folgenden sollen 3 Darstellungsarten genauer vorgestellt werden.
Eine dieser Repräsentationen ist die Darstellung eines Textes als Baum. Dabei wird davon ausgegangen, dass jedem Text eine Struktur aus Regeln zugrunde liegt. Diese Struktur kann in einer Baumform dargestellt werden. Dafür können verschiedenste Werkzeuge zur Hilfe gezogen werden. Eines dieser Werkzeuge ist der Weblicht Service der Uni-Tuebingen. Mit seiner Hilfe können Textkorpora automatisch annotiert werden. Unter anderem ist er auch in der Lage Texte automatisch in eine Parse-Baum Darstellung zu überführen. Die Repräsentation des Satzes *Das ist doch Schnee von Gestern* durch den Weblicht Service sieht wie folgt aus:

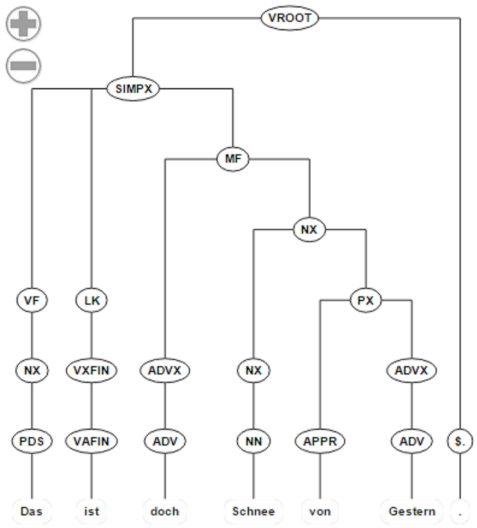


Abbildung 4.1: Beispielsatz als Parse-Baum

Dieser Parsetree stellt die dem Text zugrunde liegende Regelstruktur heraus. Das für die Analyse interessante Feature ist dabei die Anzahl und Art der Unterbäume. Jeder Unterbaum stellt ein eigenes Feature dar welches durch die SVM und Kernelmethoden analysiert werden kann.
Eine andere Darstellungsart eines Textes ist die Darstellung der einzelnen Wörter in einem Bag of Words. Es gilt: $w_1, w_2, ..., w_i \in T$ ist die Anzahl eines Wortes w_i in dem Textdokument T. Bei der Darstellung als Bag of Words werden alle Vorkommen eines Wortes w_i in T in einem Wortvektor zusammengefasst. Um dieses Maß auch in unterschiedlich großen Dokumenten anwenden zu können wird folgende Normalisierung der Häufigkeit vorgenommen:

$$tf(w,T) = \frac{f(w,T)}{max f(x,T) : x \in T}$$

Dabei steht w für das untersuchte Wort im Dokument T und $f(w,T)$ berechnet die Anzahl der Vorkommen von w in T. Dieser Wert wird dann durch die maximale Häufigkeit eines anderen Terms x aus T geteilt. Dieses Maß nennt man auch das Termfrequenzy Maß, oder Vorkommenshäufigkeit. Es wird gerade bei großen Textdokumenten benutzt um Wörtern ein Signifkanzmaß zuzuteilen. Das für die Analyse interessante Feature ist hier das Vorkommen der einzelnen Wörter.
Eine dritte Variante für die Darstellung von Texten ist die Darstellung von Substrings oder auch Subsequenzen. Dabei werden Strings in mehrere kleine Substrings zerlegt. Ein Beispiel für eine solche Darstellung ist :

Auto
Turbo

Die fett markierten Teile der Wörter sind die Substrings welche in beiden vorkommen. Dieses Feature ermöglicht einen schnellen Vergleich von mehreren Strings anhand ihrer Substrings. Desto mehr Substrings in beiden Strings auftauchen, umso ähnlicher sind sie sich.

In den nachfolgenden Experimenten werden alle 3 Repäsentationsarten und ihre zugehörigen Featues seperat voneinander untersucht. Die verschiedenen Darstellungen führen dazu, dass die Eigenschaften eines Textes in sehr großen Featurevektoren dargestellt werden. Um diese Effizient verarbeiten zu können bedienen sich die One Class Svm und die Ball Svm verschiedener Kernel Methoden, welche die inneren Produkte effizient bestimmen können. Dadurch lassen sich auch große Featurevektoren effizient von den SVM bearbeiten. Ein Einblick in die verwendeten Kernelmethoden wird im nächsten Kapitel gegeben.

Kapitel 5

Kernels

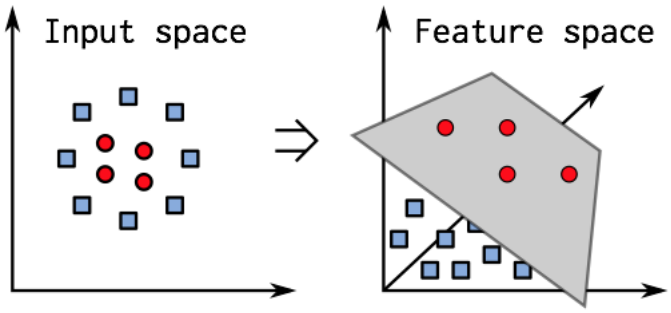
Kernel Methoden sind Mustererkennungsalgorithmen die auf vielen verschiedenen Datentypen arbeiten können. Sie sind in der Lage z.B Sequenzen, Texte, Vektoren, Graphen und Bilder zu verarbeiten. Dabei erkennen sie ein breites Spektrum an Mustern wie Cluster, Klassen und Rankings.
Da für diese Arbeit verschiedene SVM Algorithmen untersucht werden sollen, werden im diesen Kapitel mehrere Kernel Methoden vorgestellt. Ziel einer SVM ist es die Daten durch eine Hyperebene zu unterteilen. Dabei werden Daten auf jeder Seite der Hyperebene einer Klasse zugeteilt. Mehr dazu im Kapitel SVM und Enclosing Ball. Wie im vorherigen Kapitel erwähnt sind die Feature Vektoren unserer Daten besonders groß. Ein Beispiel für einen solchen Vektor ist folgender Satz in Parse-Baum Form (Dargestellt als String):

(ROOT (NUR (S (ADJD medien) ($, ,) (VP-SB (NP-OA (ADV sz) (CNP (NP (XY =) (CARD 234x60)) ($, ,) (AVP (ADV stile)) (NP (XY =) (CARD 1)))) (VVIZU -RSB-))) (NP (NN Nachrichten) (NP (NM (ADJA Aktuelle) (NN Ausgabe)) (NN Nachrichten)) ($. :)) (S (NP (NN Medien) (NM (CARD 15) ($. .) (CARD 09))) ($. .) (NP-SB (CARD 2000) (NN Fernsehmuseum) (MPN (NE TV-Koch) (NE Clemens) (NE Wilmenrod) (NE Harald) (NE Martenstein)))) (S (NP-SB (ART Das) (ADJA erste) (NN Menü) ($, ,) (S (PRELS-SB das) (NP-OA (ART der) (ADJA erste) (ADJA deutsche) (NN Fernsehkoch) (NP (PPOSAT seinem) (NN Publikum))) (VVFIN vorkochte))) ($, ,) (VVFIN ging) (ADV so) ($. :)) (S (NN-SB Fruchtsaft) (PP (APPRART im) (NN Glas) ($, ,) (NP (ADJA italienisches) (NN Omelette)) ($, ,) (NP (NE Kalbsniere) (CARD gebraten) (PP (APPR mit) (NN Mischgemüse) (PP (APPR aus) (ART der) (NN Dose)))) ($, ,) (NE Mokka))) ($. .)))

Deshalb kann es schwierig sein eine trennende Hyperebene im Raum zu finden. An dieser Stelle wird der *Kernel-Trick* angewendet. Da die Schranke des Fehlers nicht an die Dimension der Daten gebunden ist, können Kernel Methoden benutzt werden um die Daten in

einen höheren Dimensionsraum zu überführen. Dieser Feature Space ist entweder ein pre-Hilbert Raum oder ein innerer Produktraum. Durch die Überführung der Daten in einen der höherdimensionaleren Räume lässt sich leichter die Hyperebene bestimmen.

Abbildung 5.1: Beispiel für eine Transformation



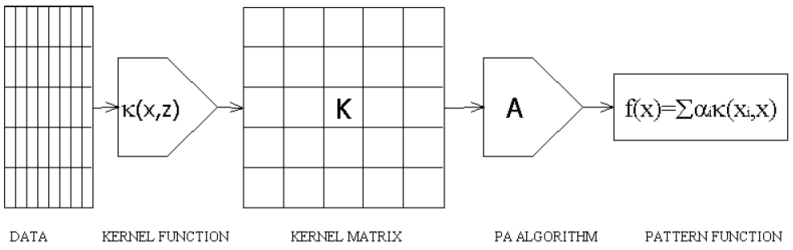
Quelle:
http://www.cg.cs.tu-bs.de/teaching/seminars/ss13/CG/webpages/SoerenPetersen/

Auf der linken Seite von Bild 1 erkennt man die Daten im zweidimensionalen Raum. Auf Anhieb ist zu sehen, dass es keine lineare Funktion gibt um die verschiedenen Klassen (hier Rot und Blau) zu unterteilen. Durch die Anwendung von Kernelmethoden lassen sich die Daten in einen Hilbertraum überführen. Dieser ist auf der Rechten Seite von Bild 1 zu erkennen. Dort lassen sich die Daten durch eine Hyperebene trennen.
Insgesamt gibt es viele verschiedene Kernel Methoden. Die am weitesten verbreiteten sind: *linear*, *sigmoid*, *polynomial* und *rbf*. Der Unterschied zwischen den Kernel Methoden liegt in den betrachteten Features. Beispielsweise vergleicht *polynomial* Kernel alle möglichen Kombinationen zwischen den verschiedenen Features. Aufgrund der von uns gewählten Feature benötigen wir außerdem noch spezielle Kernelmethoden welche auf diese zugeschnitten sind. Darunter fällt der Fast-String-Kernel, welcher sowohl Darstellungen von Substrings als auch Parse-Bäumen verarbeiten kann.

5.1 Mathematischer Hintergrund

Dabei funktionieren Kernel Methoden nach folgendem Grundgedanken [17]:
Zuerst werden die Daten χ durch Verwendung einer Funktion ϕ in einen höherdimensionalen Hilbertraum H_k übertragen, damit die SVM dann lineare Beziehungen und trennende Funktionale zwischen den Daten im Raum finden kann. Für die Funktion ϕ gilt also: $\phi : \chi \longrightarrow H_k$.

Abbildung 5.2: Idee einer Kernel Methode



Dabei wird die Funktion ϕ durch den gewählten Kernel vorgegeben. Die verwendeten Kernel Algorithmen arbeiten nicht direkt auf den Elementen im Raum, sondern berechnen die paarweisen inneren Produkte $< x_i, x_y >$ der Daten. Es gilt also für ein inneres Produkt aus V:

$$\begin{aligned} k(x,y) &= < \phi(x), \phi(y) > \\ \phi : x &\rightarrow \phi(x) \in V \end{aligned}$$

Wichtig ist vor allem die Auswahl des richtigen Kernels für das Problem, da der Vektorraum bei schlechter Wahl der Kernel Funktion wenig relevante Daten enthält. Durch eine Parameteroptimierung können dann die Kernel noch weiter verfeinert werden um die optimalen Hilberträume zu finden. Die verschiedenen Kernel Methoden und ihre Parameter werden im Folgenden noch einmal genauer betrachtet.

5.2 Verwendete Kernel Methoden

5.2.1 Subsequent String Kernel

Der *Subsequent String Kernel*, im Nachfolgenden auch SSK genannt, berechnet wie ähnlich sich zwei Strings sind. Dazu zählt er die Substrings welche in beiden Strings vorkommen. Je mehr Substrings in beiden Strings vorkommen, desto ähnlicher sind sie sich. Als Beispiel seien die 2 Strings **Baum** und T**raum** gegeben. In beiden Strings ist der Substring **aum** vorhanden, in diesem Fall sogar in der gleichen Gewichtung. Diese berechnet sich dadurch wie oft und wie nahe zusammen der gegebene Substring in den Strings vorkommt. Dafür wird ein Verfallsfaktor $\lambda \in (0, 1)$ eingeführt, welcher das Vorkommen einer bestimmten Eigenschaft eines Strings werten kann. Als Beispiel sei ein Dokument bestehend aus den Wörtern *cat*, *car*, *bat* und *bar* gegeben [5]:

	c-a	c-t	a-t	b-a	b-t	c-r	a-r	b-r
$\phi(cat)$	λ^2	λ^3	λ^2	0	0	0	0	0
$\phi(car)$	λ^2	0	0	0	0	λ^3	λ^2	0
$\phi(bat)$	0	0	λ^2	λ^2	λ^3	0	0	0
$\phi(bar)$	0	0	0	λ^2	0	0	λ^2	λ^3

Der Feature Vektor für dieses Dokument kann in der Tabelle oben eingesehen werden. Der unnormalisierte Kernel zwischen den Strings car und cat ist also das innere Produkt zwischen den Wörtern *car* und *cat*. Damit gilt für $k(car, cat) = \lambda^2 * \lambda^2 = \lambda^4$. Die normalisierte Version kann hingegen wie folgt berechnet werden: $k(car, car) = k(cat, cat) = 2\lambda^4 + \lambda^6$ und somit $k(car, cat) = \lambda^4/(2\lambda^4 + \lambda^6) = 1/(2 + \lambda^2)$. Natürlich haben die verwendeten Datensätze meist mehr als ein Wort pro String, weshalb die oben vorgeschlagene Berechnungsmethode sogar für durchschnittlich große Textdokumente unpraktisch ist. Mit Hilfe von dynamischer Programmierung lässt sich jedoch ein Kernel der die oben genannten Features verwendet effizient berechnen.

5.2.2 Tree Kernel

Tree Kernels basieren auf der Theorie, dass jeder Satz eine versteckte Struktur besitzt welche durch Regeln ausgedrückt werden kann. Diese Regeln können in Parse Bäumen dargestellt werden um eine leichtere Bearbeitung zu ermöglichen.
Um diese Bäume nun vergleichen zu können bedient sich der Tree Kernel eines ähnlichen Tricks wie der SSK. Anstatt die beiden Bäume zwischen 2 String direkt zu vergleichen, werden alle Unterbäume gefunden und miteinander verglichen. Dabei gilt für den Kernel die Funktion $K(T_1, T_2) = h(T_1) * h(T_2)$, für die Bäume T_1, T_2 [14]. Um K also zu berechnen werden die Bäume als Erstes als Menge von Knoten N1 und N2 aufgefasst. Die Funktion $I_i(n)$ gibt 1 zurück wenn der Unterbaum i eine Wurzel im Knoten n hat, ansonsten 0. Daraus folgt, dass $h_i(T_1) = \sum_{n_1 \in N_1} I_i(n_1)$ und $h_i(T_2) = \sum_{n_2 \in N_2} I_i(n_2)$. Um nun das innere Produkt effizient berechnen zu können wird folgende Eigenschaft eingeführt:

$$h(T_1) * h(T_2) = \sum h_i(T_1) h_i(T_2) = \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \sum_i I_i(n_1) I_i(n2) = \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} C(n_1, n_2)$$

Dabei gilt für $C(n_1, n_2) = \sum_i I_i(n_1) I_i(n_2)$. $C(n_1, n_2)$ kann in polynominieller Zeit berechnet werden da folgende Eigenschaften gelten:

- Wenn n_1 und n_2 unterschiedlich sind ist $C(n_1, n_2) = 0$.
- Wenn n_1 und n_2 gleich und Pre-Terminals sind, gilt für $C(n_1, n_2) = 1^2$.
- Wenn n_1 und n_2 gleich aber keine Pre-Terminals sind, gilt

$$C(n_1, n_2) = \prod_{j=1}^{nc(n_1)} (1 + C(ch(n_1, j), ch(n_2, j)))$$

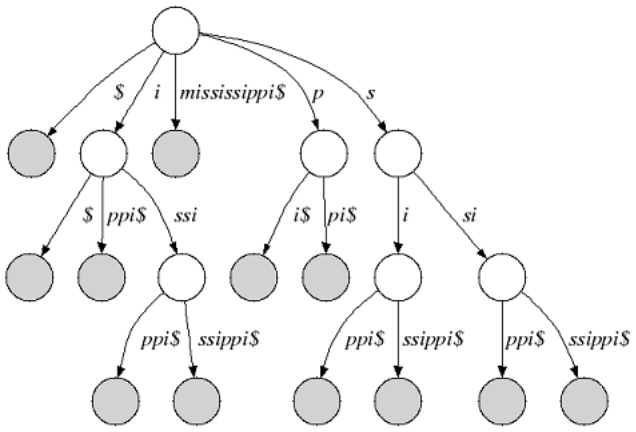
Pre-Terminals sind Knoten direkt über dem Blatt. Blätter sind in diesem Falle immer die Wörter des Ursprungsstrings. Außerdem ist $nc(n_1)$ die Anzahl der Kinder von n_1 und $ch(n_1, i)$ bestimmt das i'te Kind von n_1. In der obigen Variante des Tree-Kernels besteht das Problem, dass der Kernel vor allem vom größten Teilbaum bestimmt wird. Somit verhält er sich fast wie ein *Nearest-Neighbor* Kernel. Um diesem Problem entgegenzuwirken wird ein Parameter λ eingeführt, welcher die Relevanz eines Unterbaums direkt mit seiner größe skaliert. Somit gilt:

$$C(n_1, n_2) = \lambda \quad \text{und} \quad C(n_1, n_2) = \lambda \prod_{j=1}^{nc(n_1)} (1 + C(ch(n_1, j), ch(n_2, j)))$$

5.2.3 Fast String Kernel

Der in den nachfolgenden Experimenten verwendete Fast String Kernel benutzt die oberen beiden Kernelmethoden um in Linearzeit die Kernelmatrix zu berechnen. Da Bäume auch wie Strings dargestellt werden können, genügt für der Fast String Kernel um beide Dartesllungen zu implementieren. Im allgemeinen geht er dabei wie folgt vor: [16]
Wie bisher gezeigt gilt für einen Kernel $k(x, x') = \sum_{i \in I} \phi_i(x)\phi_i(x')$ das der Feature Vektor sehr groß ist, während die Anzahl der Einträge ungleich Null im Vergleich klein bleibt. Der Fast String Kernel sortiert die Einträge von $\phi(x)$ und $\phi(x')$ vor und zählt alle Einträge ungleich Null. Für die Berechnung des Kernels sind also nur die Vektoreinträge ungleich Null von Bedeutung. Durch eine vorherige Sortierung dieser Einträge kann Berechnungszeit gespart werden. Dieses Verfahren ist ähnlich dem Produkt von Sparse-Vektoren in der Numerik. Solange die Sortierung effizient ist, bleibt die Laufzeit von k linear. Die in der Einleitung erwähnten Repräsentationen als Substring und Tree werden in eine Reprrasentation als Suffix Baum überführt um die oben beschriebene Sortierung vorzunehmen. Ein Suffix Baum speichert alle Suffixe eines gegebenen Strings x. Im folgenden wird ein Suffix Baum für den String *Mississippi* gezeigt:

Abbildung 5.3: Beispiel für einen Suffix Baum



Quelle: http://seqan.readthedocs.org/en/latest/Glossary/SuffixTree.html

Das Zeichen \$ ist der sog. Sentinel. Für ihn gilt $\$ < y \in \sum$, wobei y ein beliebiges Zeichen aus dem Alphabet $\sum$ ist. Für jeden String x und einen Baum $S(t)$ kann nun entschieden werden ob $x \in words(S(t))$ gilt, indem die zugehörigen Kanten von $S(t)$ überprüft werden. Dies geschieht in der Funktion $lvs(t)$ und kann in $O(|t|)$ berechnet werden. Durch die Verwendung von Suffix Bäumen kann die Menge aller längsten Prefixe von $x[i : v_i]$ aus $x[i : n]$ einem String y in linearer Zeit berechnet werden. Dieser Trick wird für die

Berechnung des Kernels k zwischen den Strings x und y verwendet. Es gilt:

$$k(x,y) = \sum_{i=1}^{|x|} val(x[i:v_i)]$$

Hierbei ist $val(x) := lvs(ceil(x)) * W(y,x) + val(floor(x)) \quad and \quad val(root) := 0$. Für die Funktionen $W, lvs, ceil, floor$ gelten folgende Definitionen:

- $W(y,t) = \sum_{s\ inprefix(v)} w_{us} - w_u \quad wobei \quad u = floor(t) undt = uv$ für 2 Strings x, y und den Statistiken c, v von x.
- $ceil(t)$ gibt den Knoten w zurück für den $w = xu$ gilt, wobei u der kürzeste Substring ist, so dass $w \in nodes(S(x))$.
- $floor(t)$ gibt den Knoten w zurück für den $t = wu$ gilt, wobei u der kürzeste Substring ist, so dass $w \in nodes(S(x))$.

5.2.4 Linear

Der Lineare Kernel ist die einfachste Kernel Funktion und in fast jeder Bibliothek vorimplementiert. Er funktioniert wie folgt:

$$K(a,b) = a^T y + c$$

Wobei c auch hier ein freier Parameter ist.

5.2.5 Polynomial

Der Polynomial Kernel ist ein oft verwendeter Kernel für die SVM Klassifikation. Er eignet sich besonders für Probleme in denen alle Trainingsdaten normalisiert wurden. Er ist in vielen Bibliotheken wie LibSVM vorimplementiert und wie folgt definiert:

$$K(a,b) = (a^T y + c)^d$$

Dabei steht d für den Grad des Polynoms und C ist auch hier ein Kernelspezifischer Parameter welcher frei gewählt werden kann.

5.2.6 RBF

Der *Radial Basis Function* Kernel, oder auch Gaussian Kernel ist einer der am weitesten verbreiteten Kernel Methoden für SVM Klassifikation. Daher ist auch dieser in vielen Bibliotheken wie LibSVM vorimplementiert. Dabei ist der Kernel auf zwei Beispielen a und b wie folgt definiert:

$$K(a,b) = exp(-\frac{||a-b||^2}{2\sigma^2})$$

Dabei ist σ ein kernelspezifischer Parameter der frei gewählt werden kann. Die Wahl von Sigma spielt aber in der Performance des Kernels eine große Rolle, da sich der Kernel bei einem zu großem Wert ähnlich wie der lineare Kernel verhält. Bei einer zu kleinen Wahl des Parameters wird der Kernel anfällig für Noise in den Trainingsdaten.

5.2.7 Sigmoid

Der Sigmoid Kernel ist auch als MLP (Multilayer Perceptron) Kernel bekannt. Es hängt eng mit Neuralen Netzen zusammen, da die verwendete Sigmoid Funktion oft als Aktivator für künstliche Neuronen benutzt wird.

$$k(x, y) = tanh(\alpha x^T y + c)$$

Es ist interessant anzumerken, dass ein SVM Model, welches eine Sigmoid Kernel Methode benutzt, einem Neuralen Netzwerk mit 2 Schichten ähnelt. Die beiden Parameter α und c können frei gewählt werden. Ein oft benutzter Wert für Alpha ist aber $1/N$, wobei N für die Dimension der Daten steht.

Kapitel 6

SVM

Support Vektor Maschinen sind ein weit verbreitetes Model für maschinelles Lernen. Sie zählen zu den *überwachten* Lernvorgängen und können ein weites Spektrum an Problemen lösen, unter anderem Klassifikation, Clustering und Regression. Neben der linearen Klassifikation können Support Vektor Maschinen auch nicht lineare Probleme mittels verschiedener Kernel Methoden lösen. Dabei können sowohl Einklassen- als auch Multiklassenprobleme gelöst werden. Im folgenden werden zwar die Einklassenprobleme im Vordergrund stehen, trotzdem soll in diesem Kapitel ein kleiner Einblick in die allgemeine Funktionsweise von Support Vektor Maschinen gegeben werden. Die erste Arbeit deren Hauptthema eine SVM war wurde allerdings erst 1995 von *Vladimir Vapnik et al* herausgegeben.
Support Vektor Maschinen basieren auf dem *structural risk minimization* Prinzip [20]. Die Idee bei der *structural risk minimization* ist es, eine Hypothese h in einem Feature Raum H $H = \varphi : ||\varphi|| < \mathbb{R}$ zu finden für die der Fehler von $Error(h)$ für ein gegebenes Trainingsbeispiel S minimal ist [7].

$$(\vec{x}_1, y_1), ..., (\vec{x}_n, y_n)\ \vec{x}_i \in \Re^N, y_i \in \{-1, +1\}$$

Um eine falsche Klassifizierung durch overfitting zu vermeiden ist es wichtig das richtige Margin für das gegebene Problem zu finden. Dazu wird folgende Funktion als Schranke eingeführt:

$$h(\overrightarrow{x}) = sign\{\overrightarrow{w} * \overrightarrow{x} + b\} = \begin{cases} +1, if\, \overrightarrow{w} * \overrightarrow{x} + b > 0 \\ -1, else \end{cases}$$

Jede dieser Schranken gehört zu einer Hyperebene im Feature Raum. Die Funktion *sign* gibt 1 zurück für jedes positive Argument und -1 für jedes negative. In anderen Worten also, werden Daten anhand ihrer Position (auf welcher Seite der Hyperebene sie liegen) klassifiziert. Die inneren Produkte der Features werden durch die vorher gezeigten Kernel-Methoden bestimmt. Um nun die richtige Dimension zu finden führt Vapnik das Margin δ ein. Im Nachfolgenden wird die Funktionsweise einer SVM anhand der Hard- und Soft-Margin SVM erklärt.

6.1 SVM Arten

6.1.1 Hard-Margin SVM

Unter dem Margin δ versteht man die Distanz der Hyperebene zum nächstgelegenen Trainings Example. Bei einer Hard-Margin SVM wird solange die Hyperebene gesucht bis alle positiven und negativen Trainingsexample auf unterschiedlichen Seiten der Hyperebene liegen. Mathematisch gesehen ist das equivalent zu :

$$\forall_{i=1}^{n} : y_i[\overrightarrow{w}'\overrightarrow{x_i} + b'] >= 1$$

für jedes Trainingsexample $(\overrightarrow{x_i}, y_i)$. Diese Einschränkung sorgt dafür das jedes Trainings example auf der korrekten Seite der Hyperebene liegt. Die Hard-Margin SVM löst also das Problem:

$$V(\overrightarrow{w}, b) = \frac{1}{2}\overrightarrow{w}\overrightarrow{w}$$

unter obiger Einschränkung.
Natürlich gibt es im allgemeinen mehrere Hyperebenen. In diesem Fall sucht die SVM die Hyperebene mit dem größten Margin δ. Dabei kann davon ausgegangen werden das es für jedes Trainingszeit nur eine Hyperebene mit maximalem Margin δ gibt. Die Beispiele welche am nächsten zur Hyperebene liegen sind die Support Vektoren. Da das oben genannte Problem numerisch schwer zu lösen ist, wird stattdessen folgendes *Wolfe dual* der obigen Optimierung gelöst [4].

$$\text{minimize:} \quad W(\overrightarrow{\alpha}) = -\sum_{i=1}^{n}\alpha_i + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j (\overrightarrow{x_i} * \overrightarrow{x_j})$$

$$\text{subject to:} \quad \sum_{i=1}^{n} y_i \alpha_i = 0$$

$$\forall_i \in [1..n] : 0 <= \alpha_i$$

6.1.2 Soft-Margin SVM

Bei den Hard-Margin SVM ist natürlich das Problem gegeben, dass das SVM training fehlschlägt sobald Daten nicht linear aufteilbar sind. Zwar sind die meisten Probleme der Textklassifikation linear unterscheidbar, aber trotzdem kann es von Vorteil sein kleinere Fehler in der Klassifikation der Trainingsdaten zu erlauben. Die ergibt sich auch aus der *structural risk minimization*. Eine Lösung für dieses Dilemma ist die Soft-Margin SVM

[Cortes and Vapnik, 1995]. Diese führten für das Problem 4.1 eine obere Schranke für die Anzahl der Trainingsfehler ein und minimierten diese zusammen mit dem weight Vektor.

$$\begin{aligned} \text{minimize:} \quad & V(\overrightarrow{w}, b, \overrightarrow{\xi}) = \frac{1}{2}\overrightarrow{w}\overrightarrow{w} + C\sum_{i=1}^{n}\xi_i \\ \text{subject to:} \quad & \forall_{i=1}^{n} : y_i[\overrightarrow{w}'\overrightarrow{x_i} + b'] >= 1 - \xi_i \\ & \forall_{i=1}^{n} : \xi_i > 0. \end{aligned}$$

ξ ist hierbei eine *slack* Variable welche darstellt wie falsch unsere Vorhersage ist. Auch hier ist die Lösung numerisch wieder schwierig. Deshalb wird stattdessen das *Wolfe dual* der obigen Optimierung gelöst:

$$\begin{aligned} \text{minimize:} \quad & W(\overrightarrow{\alpha}) = -\sum_{i=1}^{n}\alpha_i + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j (\overrightarrow{x_i} * \overrightarrow{x_j}) \\ \text{subject to:} \quad & \forall_{i=1}^{n} : y_i[\overrightarrow{w}'\overrightarrow{x_i} + b'] >= 1 - \xi_i \\ & \forall_{i=1}^{n} : 0 <= \alpha_i <= C. \end{aligned}$$

Alle Trainingsbeispiele für die $\alpha_i > 0$ gilt sind die Support Vektoren. Dabei werden Support Vektoren für die $0 <= \alpha_i <= C$ gilt *ungebunden* genannt. Support Vektoren für die $\alpha_i = C$ gilt sind *gebunden*. Die hier gezeigten SVM Methoden sind natürlich nur linear. Durch Verwendung des Kernel Tricks aus dem Kapitel 5 können diese aber leicht nicht-lineare Probleme lösen. Die meisten Support Vektore Maschinen sind vom Typ der Soft-Margin SVM. Auch die im folgenden vorgestellte One-Class SVM besitzt die hier eingeführte Slack Variable ξ.

6.1.3 One Class SVM

Normalerweise werden SVM vor Allem für Zwei- oder Mehrklassenprobleme verwendet. Die Oneclass-SVM ist eine spezielle Variante der standard SVM, spezialisiert auf die Lösung von Problemen mit ungleich verteilten Klassen. Ein Beispiel für ein solches Problem ist eben die Metaphersuche in großen Textkorpora. Metaphern sind im Sprachgebrauch so selten, dass sie nur minimal oft auftreten, im Gegensatz zu normalen Anwendungen des Wortes. In den später verwendeten Datensätzen kommen z.B auf 1300 normale Anwendungen des Wortes *Ziege* nur 50 Anwendungen als Metapher. Es ist also schwierig aus diesen wenigen Beispielen eine trennende Hyperebene zu finden. Deshalb wird bei der Oneclass SVM nur aus einer Klasse gelernt. Alle anderen Daten die dann weit entfernt sind von der Hyperebene gelten als Ausreißer. Dadurch lassen sich Besonderheiten oder Fehler die nur spärlich in den Daten auftreten leichter finden.

Mathematischer Hintergrund

Schölkopf et al hat in seinem Paper *New Support Vector Algorithms* eine Oneclass-SVM wie folgt definiert [?]. Die Oneclass-SVM maximiert den Abstand der Hyperebene vom Ursprung (in einem Raum H). Die Funktion unterteilt die Beispiele also in verschiedene *Regionen*, +1 (nahe am Ursprung) oder -1 (weit Entfernt vom Ursprung). Das zu lösende Quadratische Programm (im nachfolgenden auch QP) sieht demnach wie folgt aus [3]:

$$min_{\omega,\xi_1,\rho}\frac{1}{2}||\omega||^2 + \frac{1}{\nu n}\sum_{i=1}^{N}\xi_i - \rho$$

subject to:

$$(\omega * \phi(x_i) >= \rho - \xi_i \; for \; all \; i = 1, ..., n$$

$$\xi_i >= 0 \; for \; all \; i = 1, ..., n$$

Im Gegensatz zur SVM aus dem vorherigen Kapitel gibt es bei einer Oneclass-SVM keinen Parameter C. Stattdessen wird ein Parameter ν eingeführt. Aufgrund der Relevanz dieses Parameters werden die, wie oben definierten SVM, in der Literatur auch gerne als ν - SVM bezeichnet. Für den Parameter ν gelten folgende Eigenschaften [3]:

1. *ν ist eine obere Schranke für die Ausreißer (Daten die nicht zur Klasse gehören)*
2. *ν ist eine untere Schranke für die Anzahl der Daten die als Support Vektor benutzt werden.*

Durch die Anwendung von Kernel Methoden k und des Lagrange Multipliers kann die Entscheidungsfunktion wie folgt festgelegt werden:

$$f(x) = sgn((\omega * \phi(x_i)) - \rho) = sgn(\sum_{i=1}^{n} a_i K(x, x_i - \rho)$$

Das zugrunde liegende Duale Problem ist:

$$min_\alpha \frac{1}{2}\sum_{ij}\alpha_i\alpha_j k(x_i, x_j)$$

$$\text{subject to} \quad 0 <= \alpha_i <= \frac{1}{\nu l}, \quad \sum_i \alpha_i = 1$$

Da die Lösung der One-Class SVM trotzdem aufwendig ist wird anschließend die Ball Vektor Maschine zur Lösung von Einklassenproblemen vorgeschlagen.

Kapitel 7

Enclosing Balls

Klassische Support Vektor Maschinen werden im Dualen als Lösungsansatz für viele Probleme im Maschinellen Lernen verwendet. Obwohl diese herausragende Ergebnisse in Hinsicht auf Treffergenauigkeit und Flexibilität geben, sind sie doch im Bereich der Performance auf großen Datensätzen meist zu langsam und sehr ineffizient. Normalerweise werden SVM Probleme als QP formuliert und dann mit Hilfe eines numerical Solver gelöst. Diese haben im Durchschnitt eine Trainingszeit von $O(m^3)$ und nutzen dabei $O(m^2)$ Speicher. Um die Schwäche in der Performance zu verbessern wird für die Experimente eine Ball Vektor Maschine [19] vorgeschlagen, welche eine Erweiterung der Core Vektor Maschine (im folgenden CVM; [18]) ist. Diese löst das effizientere Enclosing Ball Problem (im folgenden EB-Problem), welches eine Approximation des Minimum Enclosing Ball Problems darstellt (im folgenden MEB-Problem).
Ball Vektor Maschinen behandeln eigentlich Zweiklassenprobleme. Da die in dieser Arbeit untersuchten Metaphern als Ausreißerprobleme zu erkennen sind, ist für die Experimente vor Allem ihr Verhalten bei Einklassenproblemen von Interesse. Die Idee hinter der Anwendung der Ball Vektor Maschine ist jetzt, dass wir den Ball nur auf einer Klasse zu trainieren brauchen. Alle Metaphern unterscheiden sich dann dank ihrer Features so stark von den Trainingsdaten, dass sie außerhalb des Balls liegen.
Bevor es aber zu den Experimenten kommt, wird im folgenden ein kurzer Einblick in die CVM und die beiden Ball Probleme gegeben.

7.1 MEB-Problem

Das MEB-Problem ist auch als das *unweighted Euclidean 1-center-problem* bekannt. Es basiert auf dem *smallest circle problem*, welches 1857 von Sylvester das erste mal beobachtet wurde und gehört zur Familie der *Shape Fitting Problems*. Seitdem findet es Anwendung in vielen Bereichen wie Kollisionserkennung und im Maschinellen Lernen. Sei $S_\varphi = \varphi(x_1), ..., \varphi(x_n)$ wobei für jedes $\varphi(x_i) \in \mathbb{R}^2$ gilt. Der MEB ist der kleinste Ball der

alle Punkte aus S enthält. Da das finden eines exakten MEBs für größere Probleme nicht wirklich effizient ist, wird das MEB-Problem für die Daten durch eine $(1+\epsilon)$ Approximation vereinfacht. Außerdem wird bei der Suche nach dem MEB nur eine Submenge aus der gesamten Menge S verwendet. Diese wird auch die Coremenge oder das Coreset genannt. Es wird nur über das Coreset der Ball gesucht und sein Radius wird um $(1+\epsilon)$ erweitert, in der Hoffnung, dass auch die Punkte aus dem gesamten Set so vom Ball eingeschlossen werden. Dabei ist ϵ eine kleine positive Zahl. Es gilt also:
Sei $B(c, R)$ ein Ball mit dem Zentrum c und dem Radius R. Sei außerdem $\epsilon > 0$, dann ist $B(c, (1+\epsilon)R)$ eine Approximation des MEB(ς) wenn $R <= \gamma_{MEB(\varsigma)}$ und $\varsigma \subset B(c, (1+\epsilon)R)$ gilt. Dieses Problem wird von der Core Vektor Maschine gelöst.

7.2 CVM

Die CVM löst als Kernel Methode die oben beschriebene $(1 + \epsilon)$ Approximation des MEB-Problems. Im Nachfolgenden wird der CVM Algorithmus erläutert(Tsang et al 2007):
Sei φ die zum Kernel k zugehörige Mapping Funktion. Außerdem sei $S\varphi = \{\varphi(X1), ..., \varphi(Xn)\}$ eine Menge von Punkten über φ.
Der kleinste MEB mit dem Zentrum c^* und dem Radius R^* $(B(c^*, R^*))$ schließt folgende Punkte ein:

$$(c^*, R^*) = arg\min R^* : ||c - \varphi(xi)||^2 <= R^2$$

Das zugrunde liegende QP ist:

$$\max_{\lambda_i >= 0} \sum_{i=1}^{n} \lambda_i k_{ii} - \sum_{i,j=1}^{n} \lambda_i \lambda_j k_{ij} : \sum_{i=1}^{n} \lambda_i = 1$$

wobei $k_ij = k(x_i, x_j) = \varphi(x_i)'\varphi(x_j)$
Jedes QP dieser Form für das k^2 folgende Gleichung erfüllt

$$k(x, x) = \kappa$$

kann als MEB-Problem aufgefasst werden. In anderen Worten müssen die in den nachfolgenden Experimenten benutzten Kernelmethoden normalisiert sein.
Die CVM benutzt nun einen $(1 + \epsilon)$-Approximationsalgorithmus um eine möglichst optimale Lösung zu finden. Sei S das Set aller Daten und S_i das Coreset. Außerdem sei c_i das aktuelle Zentrum des Balls und R_i der aktuelle Radius. Dann beginnt die CVM damit, das Coreset, den Radius und das Zentrum zu Initialisieren. Dafür wird der erste Punkt aus der Feature Map φ gewählt. Im nächsten Schritt wird untersucht ob alle Punkte der Feature Map jetzt innerhalb des Balls liegen. Ist dies nicht der Fall wird der am weitesten

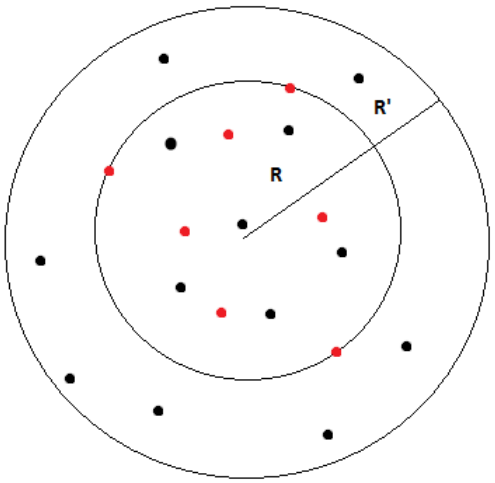


Abbildung 7.1: Der innere Kreis ist der MEB vom Set der roten Punkte und R' ist die $(1+\epsilon)$ Erweiterung. Da auch alle schwarzen Punkte enthalten sind sind die roten Punkte ein core Set.

Entfernte Punkt aus φ gewählt und ins Coreset hinzugefügt. Dies geschieht so lange bis alle Punkte in der Approximation des Radius liegen.

Algorithmus CVM [18]

- 1. Initialisiere $S_0 = \varphi(z_0), c_0 = \varphi(z_0)$ und $R_0 = 0$.
- 2. Stopp, wenn alle $\varphi(z)$ im Ball enthalten sind$B(c_i, (1+\epsilon)R_i)$.
  Sonst, finde $\varphi(z)$ das am weitesten Entfernt ist von c_i. Setze $S_{i+1} = S_t \cup \varphi(z_i)$.
- 3. Finde MEB(S_{i+1})
- 4. Erhöhe i um 1 und Spring zurück zu Schritt 2.

Wie oben gezeigt liegt dem 3. Schritt des Algorithmus immernoch ein QP zugrunde. Um also eine effiziente Berechnung zu ermöglichen wird die Ball Vektor Maschine als Vereinfachung der Core Vektor Maschine eingeführt.

7.3 Ball Vektor Maschine

Die von Tsang et al. vorgeschlagene Ball Vektor Maschine vereinfacht das MEB Problem indem ein fester Radius r gewählt wird. Außerdem gibt es kein Coreset mehr, sondern zur Betrachtung des Balls wird nur noch das aktuelle Zentrum benutzt. Der Algorithmus sieht wie folgt aus:

Algorithmus BVM [19]

- 1. Initialisiere $c_0 = \varphi(z_0)$
- 2. Stopp, wenn alle $\varphi(z)$ im Ball enthalten sind$B(c_i, (1+\epsilon)r)$.
  Sonst, sei $\varphi(z_i)$ solch ein Punkt
- 3. Finde das kleinste Update für das Zentrum, so dass $B(c_i, r)$ den Punkt $\varphi(z_i)$ berührt
- 4. Erhöhe i um 1 und Spring zurück zu Schritt 2.

Der größte Unterschied zwischen den beiden Algorithmen liegt im Schritt 3. Anstatt ein QP Problem zu lösen, sucht die Ball Vektor Maschine nach dem kleinsten Update für das Aktuelle Zentrum, so dass der neue Punkt aus der Feature Map ϕ den Ball berührt. Dieses Problem ist rein distanzbasiert und lässt sich effizient berechnen:

$$c_{i+1} = \varphi z_i + \beta_i(c_i - \varphi(z_i))$$

Für die Funktion β gilt dabei:

$$\beta_i = \frac{r}{||c_i - \varphi(z_i)||} \quad (>= 0)$$

Nach dem Update im Schritt 3 muss außerdem die Distanz zwischen dem neuen Zentrum c_{i+1} und jedem Punkt $\varphi(z)$ bestimmt werden ($O(|S_{i+1}|)$):

$$||c_{i+1}||^2 = ||\beta_i c_i + (1-\beta_i)\varphi(z_i)||^2$$
$$= \beta_i||c_i||^2 + (1-\beta_i)||\varphi(z_i)||^2 + (\beta_i^2 - \beta_i||c_i - \varphi(z_i)||^2$$

Im Nachfolgenden wird eine Implementierung der oben erklärten Ball Vektor Maschine auf verschiedenen Datensätzen benutzt. Es wird besonders ein Augenmerk auf die Accuracy der Metapher Klasse gelegt.

Kapitel 8

Experimente

Einleitung

Im folgenden wird der Aufbau, die Durchführung und die Ergebnisse der Experimente dokumentiert. Für die hier durchgeführten Experimente wurden die folgenden Datensätze benutzt. Die Datensätze enthalten Textbeispiele in denen die Vorkommen der Wörter *Hammer*, *Festplatte*, *Ziege* nach *Ungewöhnlich*, *Metapher*, *Eigenname* und *Sonstiges* klassifiziert wurden. Im Nachfolgenden werden die Datensätze nach ihren untersuchten Wörtern benannt. Die Text Attribute der Datensätze sind sehr groß. Als Beispiel seien

Nach zwei Monaten ging er pleite , es war ein schlechter Sommer . Er kehrte nach Wilhelmsburg zurück und fühlte sich so chancenlos wie damals , als Schüler , auf dem Gymnasium in Ankara . Neu war er dort , erstmals in einer Stadt aufgewachsen bei der Großmutter , in einem kleinen Dorf an der armenischen Grenze . Er konnte ***Ziegen****(keine Metapher) melken , Mehl mahlen , und er sprach Türkisch wie ein bayerischer Bauer Deutsch . In der Schule in Ankara wurde er wegen Dialekt ausgelacht . Sancar trug zu jener Zeit kaputte Hosen . Er sagt : Ich habe mich geschämt .*

und

Barbara Kolzer , schneeweiß , ist 72 Jahre alt . Ihre Freundin Jutta Hitzke , dunkelblond , ist ein Jahr jünger . Am 30. Dezember 1954 waren die beiden zum ersten Mal hier , sie verliebten sich in den Bariton , einen Dänen namens Herman Hansen . Hansen war mit einer Tänzerin liiert , die Kolzer und Hitzig die ***Ziege****(Metapher) nannten . Vor Hansens Haus sangen sie ein Liebesduett , sie waren jung , sie waren Groupies . Doch Hansen ließ sich nicht blicken , er blieb bei seiner* ***Ziege****(Metapher) . Heute hat das Landestheater keine Groupies mehr .*

gegeben. Auf den 3 Datensätzen wird ausschließlich die Klassifizierung nach Metaphern untersucht. Der Datensatz *Menü* enthält 500 Beispiele in denen das Wort Menü in 2 verschiedenen Bedeutungen vorkommt. Einmal als Speisekarte und einmal als Benutzerschnittstelle. Zwar handelt es sich dabei nicht um ungleich verteilte Klassen, trotzdem ist dieser Datensatz hinzugenommen worden um das verhalten der gewählten Methoden auf einem anderen Problem des NLP zu testen. Im Nachfolgenden wird ein kurzer Überblick über die benutzt Implementierungen und Werkzeuge geben.

8.0.1 Werkzeuge

Rapidminer

Rapidminer ist eine an der TU-Dortmund entwickelte Umgebung für maschinelles Lernen und Data Mining. Auf dem Rapdiminer Marketplace werden viele Plugins für verschiedenste Data Mining Probleme angeboten. Insbesondere *Text Processing*, ein Plugin für Textverarbeitung und Textmining, welches Operatoren zur Umwandlung und Verarbeitung von Textdokumenten stellt. Für die durchgeführten Experimente ist besonders der *Tokenize* Operator für die Erstellung von Wordbags von Interesse.

Weblicht

Weblicht ist eine Service-orientiere Webplattform zur Erstellung und Bearbeitung annotierter Textkorpora. Für die Experimente ist besonders die Erstellung von Parse-Bäumen in Strings Darstellung für den Tree Kernel mithilfe des NLP parsers von Bedeutung. Mithilfe von Operatoren in der Implementierung des *FastStringKernels* können Dokumente direkt aus dem Rapidminer auf Weblicht geladen, verarbeitet und in StringTrees für weitere Bearbeitung und Analyse gespeichert werden.

NLP-Parser

Für die Darstellung der Daten in Baumstruktur wurde außerdem eine Implementierung des NLP-Parsers aus dem Fachprojekt 2014 verwendet.

Implementierungen

Im Rahmen der Experimente werden Implementierungen eines *FastStringKernels* von *Marcel Fitzner* und einer *Ball-SVM* von Hendrik Bloom benutzt. Außerdem wird das Plugin *Text Processing* aus dem Rapidminer Marketplace verwendet.

8.0.2 LibSVM

LibSVM ist eine Open Source Bibliothek welche an der National Taiwan University entwickelt wurde. Sie ist in C++ geschrieben und als voll funktionsfähiger Operator im Ra-

pidminer in Java integriert. Dabei stellt sie verschiedene SVM Methoden mit vorimplementierten Kernels zur verfügung, inklusive der für diese Experimente benötigten One Class SVM. Außerdem besteht die Möglichkeit im Voraus berechnete Kernel durch die *precomputed* Kernel Methode mit den verschiedenen SVMs zu benutzen.

8.1 Oneclass-SVM

8.1.1 Parameter der Oneclass-SVM

Folgende Parameter der Oneclass-SVM wurden manuell optimiert:

- ϵ: Toleranzparameter der SVM
- ν: Schrankenparameter der SVM

Vorallem Parameter ν hat einen großen Einfluss auf die Performance der SVM. Wie in Kapitel 6 beschrieben ist er eine obere Schranke für die Ausreißer und eine untere für die Anzahl der verwendeten Support Vektoren.

8.1.2 Experiment I : Subsequent String Kernel

Das erste Experiment erforscht das Verhalten eines Subsequent String Kernels (Implementiert durch den *FastStringKernel*) und der OneClass SVM auf den 4 Datensätzen.

Aufbau

Für das erste Experiment gilt folgender Aufbau im Rapidminer:

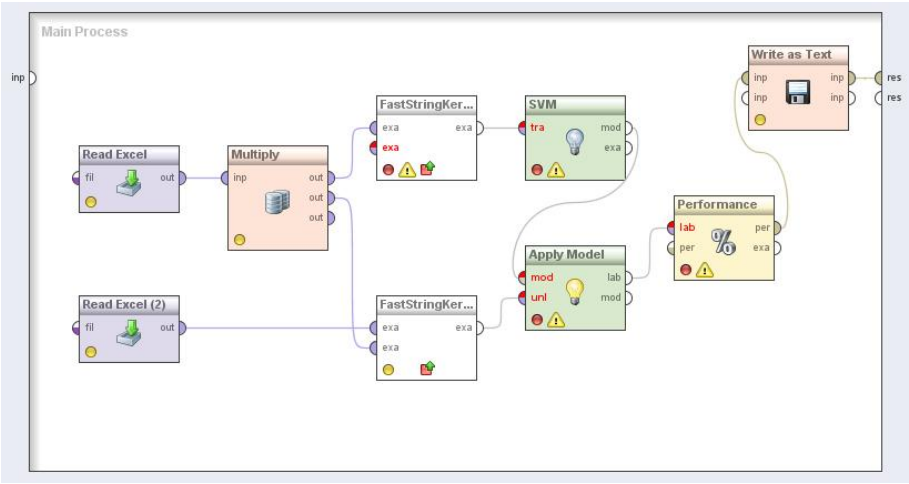


Abbildung 8.1: Aufbau Experiment I

Zuerst werden die Datensätze in den *read* Operator eingelesen. Dabei werden die Beispiele nach den Vorkommen von Metaphern unterteilt. Anschließend berechnet der *FastStringKernel* eine Kernelmatrix, die dann in der SVM als Modell angelernt zu werden. Diese erwartet eine fertige Kernelmatrix, welche von dem *FastStringKernel* bereitgestellt wird. Im letzten Schritt wird unser gelerntes Modell auf den Testdatensatz angewendet und die Performance durch den *Performance* Operator ausgegeben und durch den *Write* Operator in eine Logdatei geschrieben.

Durchführung

Vor der Durchführung des Tests wurden die Datensätze in 2 Klassen unterteilt. Einmal in die Beispiele ohne Metaphern und einmal in die Beispiele mit Metaphern. Aus den Daten ohne Metaphern wird das Modell gelernt um auf der Verknüpften Kernelmatrix angewendet zu werden. Für die optimalen Parameter der Oneclass-SVM gelten folgende Einstellungen:

- Parameter ν: 0.5
- Parameter ϵ: 0.001

Ergebnisse

Datensatz	True 0	True 1	False 0	False 1	Prec. 0	Prec. 1	Recall 0	Recall 1
Festplatte	159	34	340	4	96,54%	9%	31,7%	89,5%
Hammer	89	36	411	13	87,25%	8%	17,8%	73,5%
Ziege	13	54	487	3	81,25%	9,98%	2,6%	94,74%
Menü	83	35	66	29	74,1%	34,65%	55,7%	54,7%

True 0 steht in der Tabelle oben für die Anzahl der richtig Klassifizierten Daten welche keine Metapher sind. True 1 steht für die richtig Klassifizierten Metaphern und False 0/1 dementsprechend für die Falsch Klassifizierten Daten. Für die folgenden Tabellen gilt die gleiche Beschreibung.

8.1.3 Experiment II: Tree Kernel

Das zweite Experiment erforscht das Verhalten eines Tree Kernels (Implementiert durch den *FastStringKernel*) und der OneClass SVM auf den 4 Datensätzen.

Aufbau

Für das zweite Experiment gilt folgender Aufbau im Rapidminer:

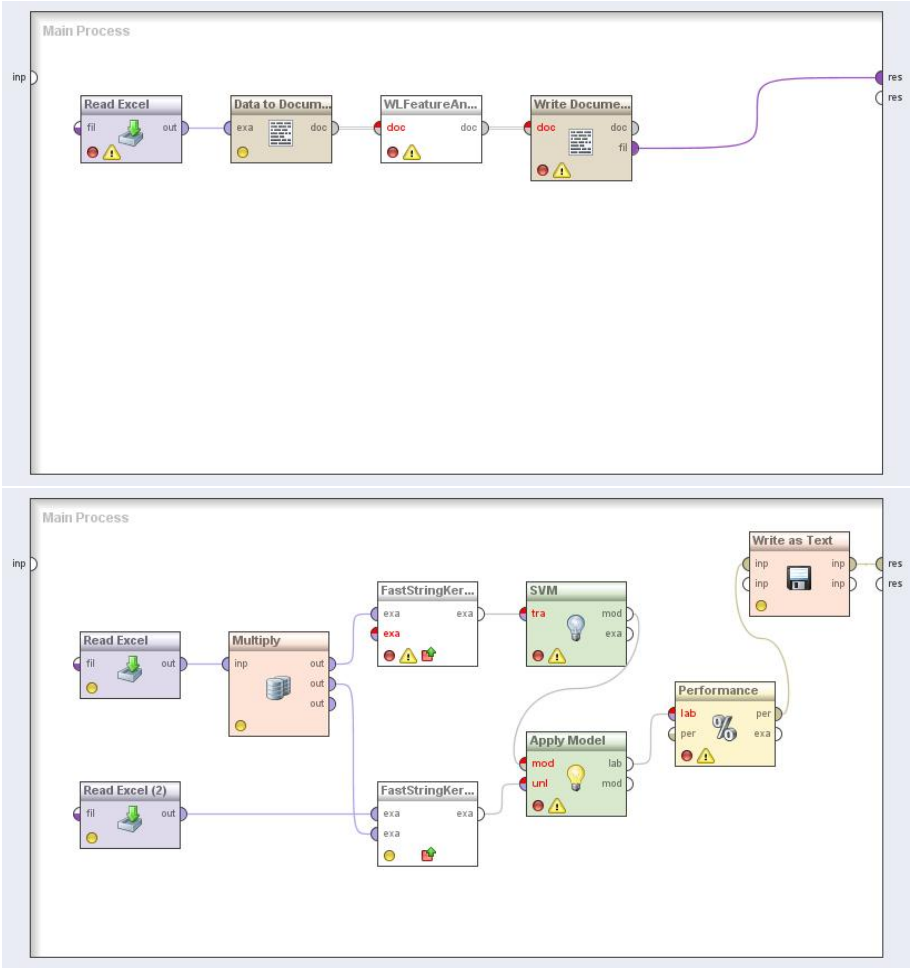


Abbildung 8.2: Aufbau Experiment II

Der Aufbau ist ähnlich wie im vorherigen Experiment. Der Unterschied besteht in der Vorbereitung der Daten. Diese werden in 2 vorherigen Aufrufen aus einer Excel Datei ausgelesen und mithilfe der Weblicht Plattform in eine Baumstruktur überführt. Diese wird wiederum in ein Document geschrieben welches später durch den *WLTFC2Exampleset* Operator in ein für den *FastStringKernel* verarbeitbares Exampleset umgewandelt wird. Die weitere Vorgehensweise erfolgt wie in Experiment I.

Durchführung

Vor der Durchführung des Tests wurden die Datensätze in 2 Klassen unterteilt. Einmal in die Beispiele ohne Metaphern und einmal in die Beispiele mit Metaphern. Aufgrund der Berechnungskosten der Parse-Bäume wurden nur 10% der Daten verwendet. Dies geschah durch einen *balanced sample operator*. Diese werden separat durch Weblicht in eine Baumstruktur überführt. Aus den Daten ohne Metaphern wird das Modell gelernt um auf der Verknüpften Kernelmatrix angewendet zu werden. Für die optimierten Parameter der Oneclass-SVM gelten folgende Einstellungen:

- Parameter ν: 0.5

Ergebnisse

Datensatz	True 0	True 1	False 0	False 1	Precision 0	Precision 1	Recall 0	Recall 1
Festplatte	32	2	18	1	96,97%	10%	64%	66,67%
Hammer	30	0	23	1	96,77%	0%	56,6%	0%
Ziege	28	1	20	6	82,35%	4,76%	58,33%	14,29%
Menü	7	1	8	5	58,33%	11,11%	46,67%	16,67%

8.1.4 Experiment III: Bag of Words mit versch. Kernelmethoden

Das dritte Experiment erforscht das Verhalten der Daten in Bag of Word Form auf verschiedenen vorimplementierten Standartkernels der Oneclass-SVM.

Aufbau

Für das dritte Experiment gilt folgender Aufbau im Rapidminer:

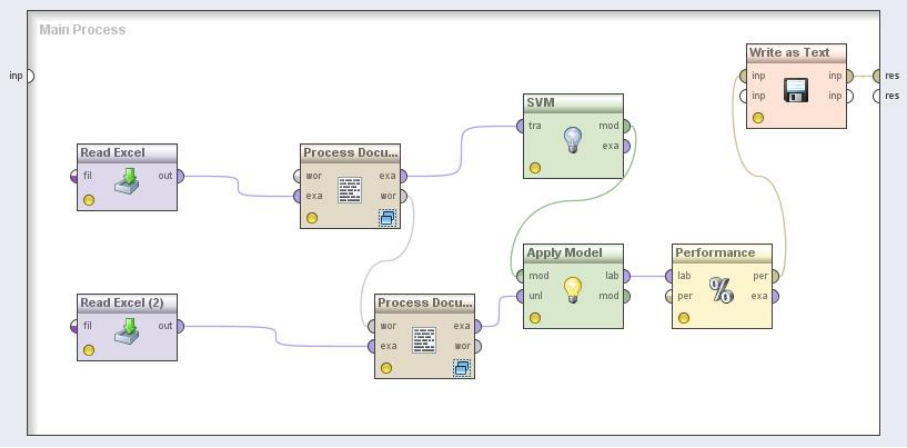


Abbildung 8.3: Aufbau Experiment III

Der Unterschied zu den vorherigen Experimenten besteht darin, dass vor dem Training in der SVM die Daten durch den *Process Documents* Operator in einen Bag of Words umgewandelt werden. Außerdem benutzt die SVM in diesem Experiment keinen precomputed kernel, sondern wendet verschiedene Kernel auf die Datensätze an.

Durchführung

Vor der Durchführung des Tests wurden die Datensätze in 2 Klassen unterteilt. Einmal in die Beispiele ohne Metaphern und einmal in die Beispiele mit Metaphern. Aus den Daten ohne Metaphern wird das Modell gelernt um auf der Verknüpften Kernelmatrix angewendet zu werden. Außerdem werden für jeden Datensatz verschiedene Kernelmethoden getestet. Für die optimierten Parameter der Oneclass-SVM gelten folgende Einstellungen:

- ν: 0.5

Ergebnisse

Datensatz	Kernel	Precision 0	Precision 1	Recall 0	Recall 1
Festplatte	Rbf	91,89%	2,15%	81,7%	5%
Festplatte	linear	97,72%	10,38%	42,9%	86,8 %
Festplatte	poly	100%	7,29%	3,2%	100%
Festplatte	sigmoid	97,73%	10,41%	43,1%	86,8%
Hammer	Rbf	90,65%	7,44%	76%	18,4 %
Hammer	linear	90,81%	8,66%	49,4%	49%
Hammer	poly	88,89%	8,89%	1,6%	97,8%
Hammer	sigmoid	90,74%	8,60%	49%	49%
Ziege	Rbf	89,53%	8,99%	83,8%	14%
Ziege	linear	89,63%	12,97%	45%	71,9%
Ziege	poly	100%	10,33%	1%	100%
Ziege	sigmoid	93,44%	13,10%	45,6%	72%
Menü	Rbf	70,76%	60,00%	99%	4,7%
Menü	linear	77,78%	31,18%	14%	90,6%
Menü	poly	0%	30,05%	0%	100%
Menü	sigmoid	77,78%	31,18%	14%	90,6%

Für eine besser Übersicht wurden bei diesen Ergebnissen die absoluten Werte der Experimente ausgelassen.

8.2 Ball Vektor Maschine

8.2.1 Parameter der Ball Vektor Maschine

Folgende Parameter der Ball Vektor Maschine wurden per Hand optimiert:

- Parameter C: Der Komplexitätsparameter der BVM.
- Parameter ϵ: Toleranzparameter der BVM.

8.2.2 Experiment I : Subsequent String Kernel

Das erste Experiment erforscht das Verhalten eines Subsequent String Kernels (Implementiert durch den *FastStringKernel*) und der Ball Vektor Maschine auf den 4 Datensätzen.

Aufbau

Für das erste Experiment gilt folgender Aufbau im Rapidminer:

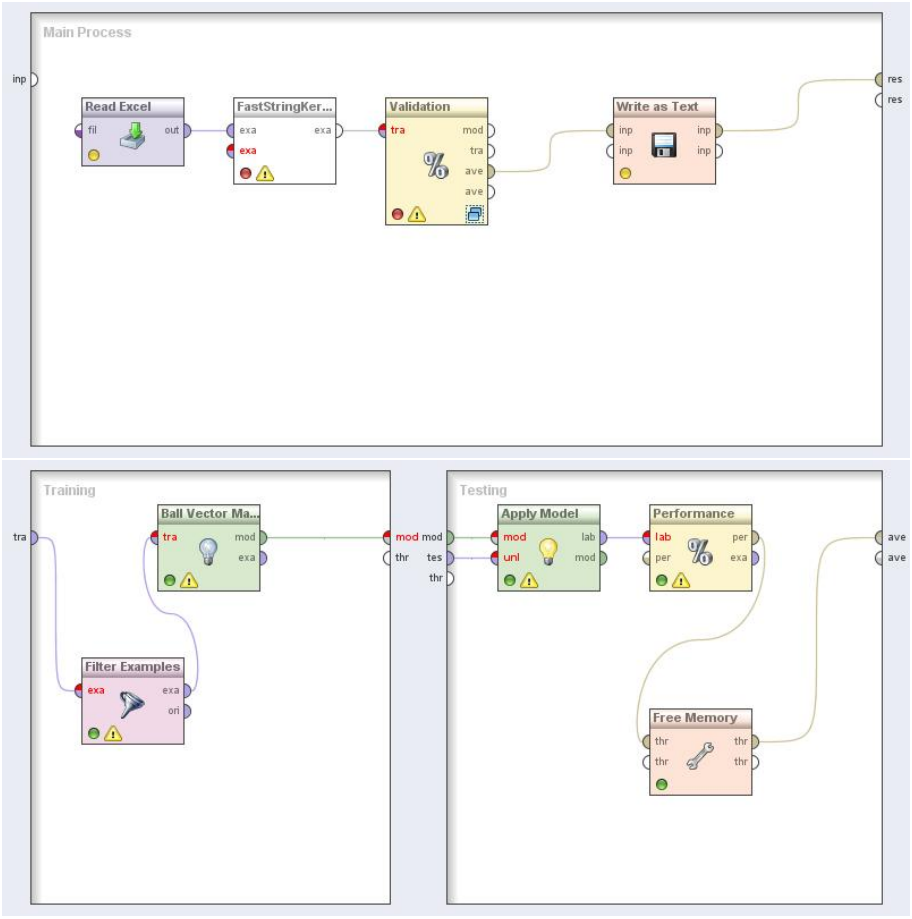


Abbildung 8.4: Aufbau Experiment I Ball

Zuerst werden die Datensätze in den *read* Operator eingelesen. Dabei werden die Beispiele nach den Vorkommen von Metaphern unterteilt. Anschließend werden die Daten im *FastStringKernel* in eine Kernelmatrix vorberechnet um dann in der SVM als Modell angelernt zu werden. Diese erwartet als Kernelmethode eine fertig Kernelmatrix, welche von dem *FastStringKernel* bereitgestellt wird. Dieser Vorgang findet im *Validation* Operator statt. Dabei werden die in einer Kreuzvalidierung als trainiert und anschließend getestet. Im letzten Schritt wird unser gelerntes Modell auf den Testdatensatz angewendet und die Performance durch den *Performance* Operator ausgegeben und durch den *Write* Operator in eine Logdatei geschrieben.

Durchführung

Durch die Kreuzvalidierung im *Validation* Operator werden die Daten zufällig unterteilt um dann als Test- und Trainingsdaten angewendet zu werden. Wichtig ist dabei, dass die Daten vor dem Training gefiltert werden. Nur Daten der Klasse 0 (keine Metapher) sollten zum Training verwendet werden. Dieser Vorgang wiederholt sich 10 mal. Das Ergebnis wird normalisiert und an den Performance Operator weitergegeben. Für die Parameter der Ball Vektor Maschine gelten folgende Einstellungen:

- Parameter C: 1.0
- Parameter ϵ: 0.001

Ergebnisse

Datensatz	True 0	True 1	False 0	False 1	Prec. 0	Prec. 1	Recall 0	Recall 1
Festplatte	1131	0	0	38	96,75%	0,00%	100%	0%
Hammer	1241	3	66	46	96,43%	4,35%	95%	4,3%
Ziege	1661	9	201	48	97,19%	4,29%	89,2%	15,9%
Menü	273	65	11	52	84,00%	14,47%	96,1%	56%

8.2.3 Experiment II: Tree Kernel

Das zweite Experiment erforscht das Verhalten eines Tree Kernels (Implementiert durch den *FastStringKernel*) und der Ball Vektor Maschine auf den 4 Datensätzen.

Aufbau

Für das zweite Experiment gilt folgender Aufbau im Rapidminer:

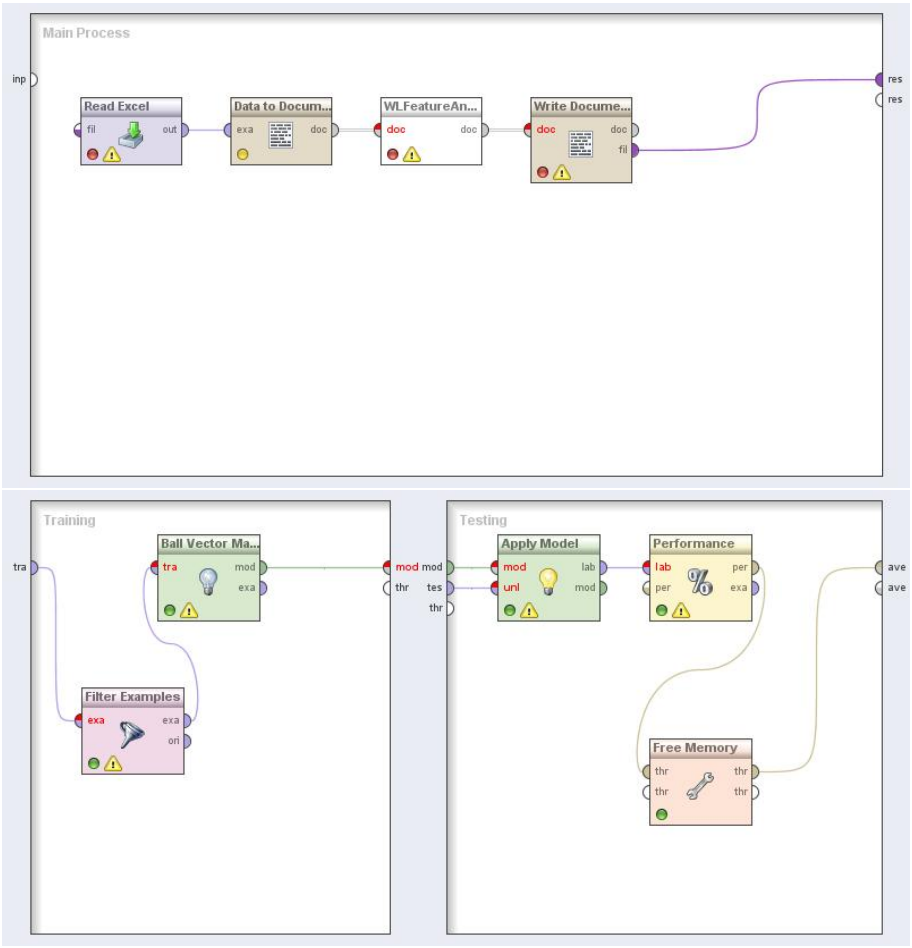


Abbildung 8.5: Aufbau Experiment II Ball

Der Aufbau ist ähnlich wie im vorherigen Experiment. Der Unterschied besteht in der Vorbereitung der Daten. Diese werden in 2 vorherigen Aufrufen aus einer Excel Datei ausgelesen und mithilfe der Weblicht Plattform in eine Baumstruktur überführt. Da dieser Vorgang sehr viele Ressourcen kostet wurden nur 10% der Daten verarbeitet. Die Baumstruktur wird wiederum in ein Document geschrieben welches später durch den *WLTFC2Exampleset* Operator in ein für den *FastStringKernel* verarbeitbares Exampleset umgewandelt wird. Die weitere Vorgehensweise erfolgt wie in Experiment I.

Durchführung

Vor der Durchführung der Experimentes wurden die Daten durch den Weblicht Operator in eine Parse-Tree Darstellung als String überführt. Die restliche Durchfuhren erfolgt wie im vorherigen Experiment. Für die optimierten Parameter der Ball Vektor Maschine gelten folgende Einstellungen:

- Parameter C: 1.0
- Parameter ϵ: 0.001

Ergebnisse

Datensatz	True 0	True 1	False 0	False 1	Prec. 0	Prec. 1	Recall 0	Recall 1
Festplatte	112	0	0	4	96,55%	0,00%	100%	0%
Hammer	128	0	0	7	94,81%	0,00%	100%	0%
Ziege	185	0	0	6	96,86%	0%	100%	0%
Menü	35	0	0	5	87,5%	0%	100%	0%

8.2.4 Experiment III: Bag of Words mit RBF Kernel

Das dritte Experiment erforscht das Verhalten der Daten in Bag of Words Form mit Verwendung des RBF Kernels. Dieser ist der einzige normalisierte Kernel der BVM-Implementierung.

Aufbau

Für das dritte Experiment gilt folgender Aufbau im Rapidminer:

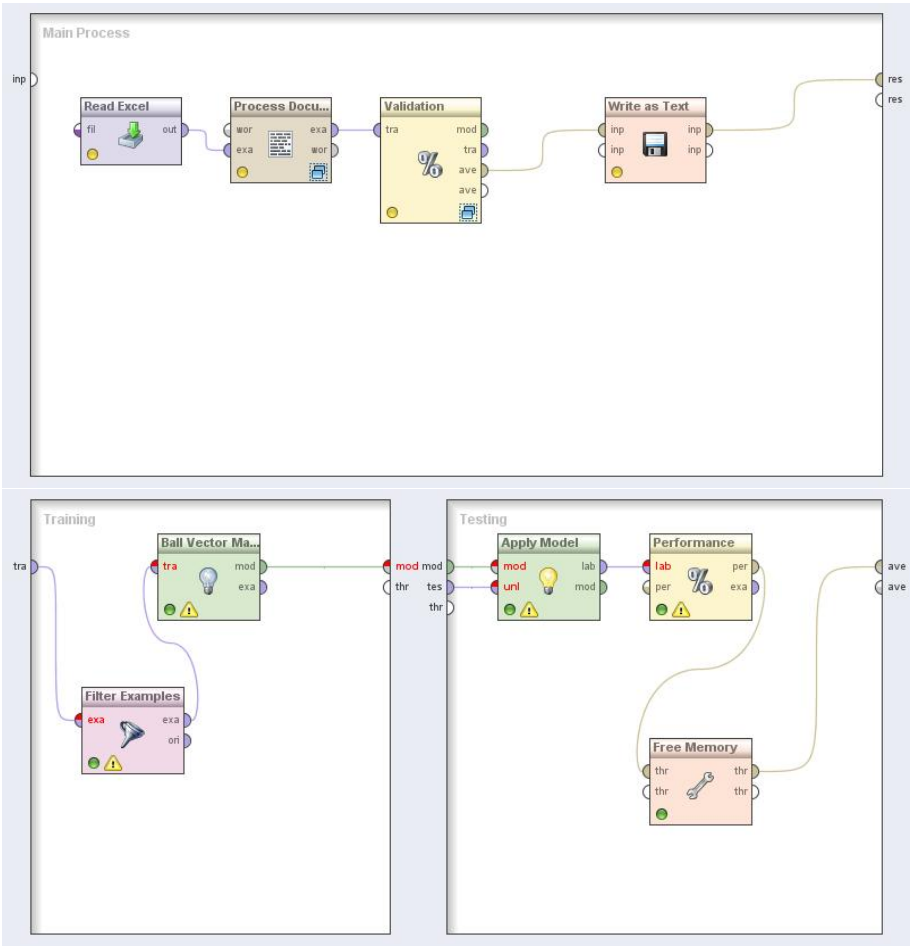


Abbildung 8.6: Aufbau Experiment III Ball

Der Unterschied zu den vorherigen Experimenten besteht darin, dass vor dem Training in der SVM die Daten durch den *Process Documents* Operator in einen Bag of Words umgewandelt werden. Außerdem benutzt die SVM in diesem Experiment keinen precomputed kernel, sondern wendet verschiedene Kernel auf die Datensätze an.

Durchführung

Die Durchführung erfolgtg wie im ersten Experiment. Für die optimierten Parameter der Ball Vektor Maschine gelten folgende Einstellungen:

- Parameter C: 1.0
- Parameter ϵ: 0.001
- Parameter γ: Parameter des RBF-Kernels : 1.0

Eine Veränderung der Parameter hat zu keiner Änderung der Ergebnisse geführt.

Ergebnisse

Datensatz	True 0	True 1	False 0	False 1	Prec. 0	Prec. 1	Recall 0	Recall 1
Festplatte	1131	0	0	38	96,75%	0,00%	100%	0%
Hammer	1307	0	0	49	96,39%	0,00%	100%	0%
Ziege	1862	0	0	57	97,03%	0,24%	100%	0%
Menü	338	0	0	63	84,29%	0,00%	100%	0%

Kapitel 9

Fazit

Aus den obigen Experimenten erkennt man, dass die Ball Vektor Maschine deutlich schlechtere Ergebnisse liefert. Die Wahl der Features und ihrer Repräsentation ist egal. Sowohl Substrings, als auch Bäume und der Bag of Words liefern keine zufriedenstellenden Ergebnisse. Metaphern werden meist nicht erkannt. Dies kann mehrere Möglichkeiten haben, welche im Folgenden untersucht werden sollen.

Möglichkeiten für schlechte Ergebnisse der Experimente:

- Features stellen den Unterschied zwischen Metaphern und normalen Anwendungen nicht heraus.
- Zu lange Text Attribute sind Noise und enhalten keine Informationen.
- Die Ball Vektor Maschine ist nicht für Ausreißererkennung geeignet.

Natürlich lassen sich am Ende dieser Arbeit nicht alle Möglichkeiten ausführlich überprüfen. Der genaue Grund für die schlechten Ergebnisse bildet eine Grundlage für weiter Untersuchungen in der Erkennung von Metaphern in großen Textkorpora. Trotzdem sollen im weiteren Verlauf die oben aufgezählten Möglichkeiten im Rahmen der durchgeführten Experimente überprüft werden.

9.1 Features

Die Aussagekraft der hier gewählten Features ist schwer einzuschätzen. Um die Relevanz der gewählten Features zu untersuchen wird am Beispiel des Datensatzes *Ziege* die Korrelation der Features berechnet.
Für den Bag of Words Ansatz sind in der folgenden Tabelle die 3 Attribute mit dem höchsten und dem niedrigsten Wert als Beispiel gegeben:

Attribut	Gewichtung
Minuten	0
Rücken	0
Gegend	0
weinend	0.726
blöde	0.856
dumme	1

Die gesamte Gewichtung kann in der Log Datei *BagLogsWeight* eingesehen werden. Ein unterschied zwischen den verschiedenen Wortarten und ihrer Gewichtung lässt sich auf Anhieb nicht herausstellen. Auffällig ist aber, dass zwar ein paar Attribute existieren die eine hohe Gewichtung haben, aber die meisten sich im Radius von 0.01- 0.1 bewegen. Das lässt darauf schließen, dass viele Daten im Bag of Words zu Noise führen [10]. Eine eventuelle Reduzierung der Daten vor den Experimenten könnte also die Ergebnisse beim Bag of Words Ansatz verbessern.
Für den SSK Ansatz ergaben sich folgende Gewichtungen:

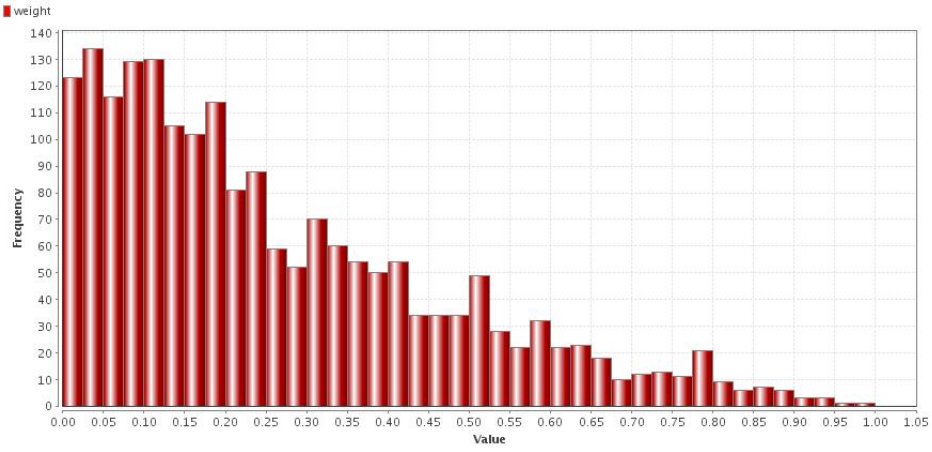


Abbildung 9.1: Verteilung der Gewichtungen beim SSK

Auch hier erkennt man, dass die meisten Attribute für die Klassifizierung unerheblich sind. Die Logs sind unter *SSKLogsWeight* zu finden.
Die Gewichtungen für die Parse-Baum Darstellung sind in der Log Datei *TreeLogsWeight* zu finden. Es ergaben sich folgende Gewichtungen:

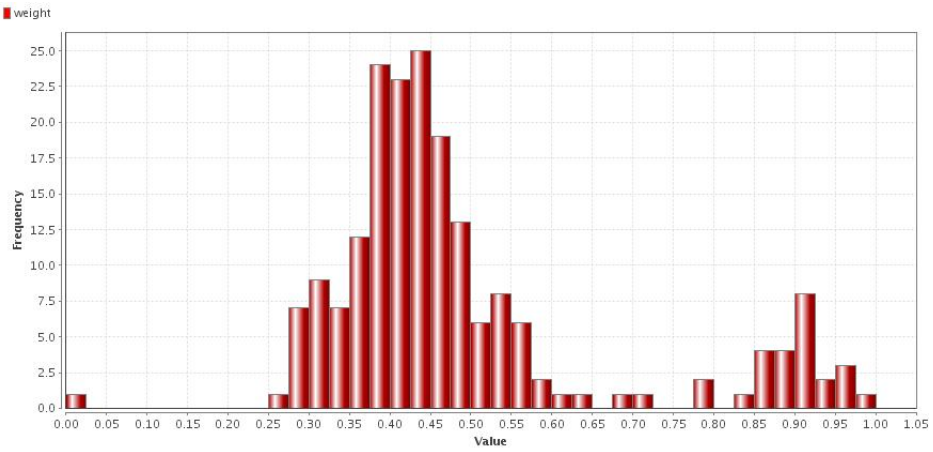


Abbildung 9.2: Verteilung der Gewichtungen bei der Parse-Baum Darstellung

Interessanterweise scheinen die Parsebäume mehr relevante Informationen zu enthalten als die anderen Darstellungen. Da die Ergebnisse aber besonders bei der Ball Vektor Maschine trotzdem schlecht sind, muss an dieser Stelle noch weiter nach der Ursache gesucht werden.

9.2 Noise im Text

Wie bei der Einführung der Datensätze erwähnt sind die Text Attribute der Daten sehr lang. Dazu zählen im Speziellen Sätze die vor und nach dem eigentlichen Auftreten der Metapher liegen. Da bei allen gewählten Features die Gewichtungen zur Klassifizierung nicht sonderlich gut verteilt sind, besteht die Möglichkeit, dass die Sätze vor und nach dem Auftreten der Metapher keine relevanten Informationen enthalten. Sie verfälschen das Ergebnis durch Noise. Um diese These zu stützen wurden im Datensatz *Ziege* die Text Attribute Testweise verkleinert. Der sonstige Aufbau der Experimente entspricht dem aus Kapitel 8. Für die Ball Vektor Maschine ergibt sich dabei:

Feature	True 0	True 1	False 0	False 1	Prec. 0	Prec. 1	Recall 0	Recall 1
SSK	16	0	0	3	84,21%	0%	100%	0%
Bag	16	0	0	3	84,21%	0%	100%	0%
Tree	16	0	0	3	84,21%	0%	100%	0%

Da immer noch keine besseren Ergebnisse erzielt wurden lässt sich daraus schließen, dass die Ball Vektor Maschine nicht für eine Ausreißererkennung geeignet ist.

9.3 Ball Vektor Maschine

Wie gut die Ball Vektor Maschine für Ausreißererkennung geeignet ist, lässt sich im Rahmen dieser Arbeit nicht weiter überprüfen. Um aber ein Vergleichsbeispiel mit einem Multiclass Problem zu bieten wurde der Datensatz *Menü* noch einmal als 2-Klassen Problem mit der Ball Vektor Maschine untersucht. Im Gegensatz zu den bisher getesteten Experimenten wurde hierbei die BVM auf beiden Klassen trainiert. Der sonstige Aufbau der Experimente gleicht dem aus Kapitel 8. Ein besonderes Augenmerk liegt auf der Darstellung der Daten als Parse-Bäume. Wie oben erwähnt scheinen diese die meisten relevanten Informationen zu enthalten.

Feature	True 0	True 1	False 0	False 1	Prec. 0	Prec. 1	Recall 0	Recall 1
SSK	71	50	267	13	84,52%	15,77%	21,01%	79.37%
Bag	338	0	0	57	97,03%	0%	100%	0%
Tree	18	3	17	2	90%	15%	51,43%	60%

Im Gegensatz zu den Ergebnissen aus Kapitel 8 erzielt die Ball Vektor Maschine auf dem gleichen Datensatz bessere Ergebnisse, sobald die Daten als Zwei Klassen Problem gelernt werden. Der Bag of Words scheint dabei mit ungekürzten Daten immer noch zu viel Noise zu enthalten. Das lässt darauf schließen, dass die Ball Vektor Maschine nicht für Ausreißererkennung geeignet ist.

Abschließen bleibt zu sagen, dass die in dieser Arbeit vorgeschlagene Methode der Ball Vektor Maschine keine Verbesserung gegenüber der herkömmlichen One-Class SVM bietet. Außerdem hat sich gezeigt, dass die Erkennung von ungewöhnlichen Textbelegen in großen Textkorpora ein schwieriges Thema ist, welches weiterer Forschung bedarf.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Anna Feldman, Jing Peng: *An approach to automatic figurative language detection: A pilot study.* Conference: Proceedings of the Corpus-Based Approaches for Figurative Language Colloquium.

[2] Ben-Gal, Irad: *Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers.* Kluwer Academic Publishers, 2005.

[3] Bernhard Schölkopf, Alex J. Smola, Robert C. Williams und Peter L. Bartlett: *New Support Vector Algorithms.* 2000.

[4] Fletcher, R.: *An Optimal Positive Definite Update for Sparse Hessian Matrices.* Seiten 192 – 218, 1992.

[5] Huma Lodhi, Craig Saunders, John Shawe-Taylor Nello Christianini und Chris Watkins: *Text Classification using String Kernels.* Journal of Machine Learning Research 2, 2002.

[6] Iryna Gurevych, Petra Gehring und: *Metaphern als Strenge Wisschenschaft.* 3. Phänomenal, Seiten 99 – 109, 2014.

[7] Joachims, Thorsten: *Learning to classify text using support vector machines.* Kluwer Academic Publishers, 2001.

[8] Kyriakopoulou, Antonia: *Text Classification Aided by Clustering: a Literature Review.* Tools in Artificial Intelligence.

[9] Manevitz, Larry M. und Malik Yousef: *One-Class SVMs for Document Classification.* Journal of Machine Learning Research 2, Seiten 139–154, 2001.

[10] Marc Schulder, Eduard Hovy: *Metaphor Detection through Term Relevance.* Proceedings of the Second Workshop on Metaphor in NLP, Seiten 18–26, 2014.

[11] Martin, James H.: *A Computational Model of Metaphor Interpreation.* Academic Press, Inc., 1990.

[12] Matt Gedigian, John Bryant, Srini Narayanan und Branimir Ciric: *Catching Metaphors.*

[13] Mácha, Jakub: *Analytische Theorien der Metapher.* Lit Verlag Dr. W. Hopf, 2010.

[14] Michael Collins, Nigel Duffy: *Convolution Kernels for Natural Language.*

[15] Raschka, Sebastian: *Naive Bayes and Text Classification I.*

[16] S.V.N. Vishwanathan, Alexander J. Smola: *Fast Kernels for String and Tree Matching.*

[17] Thomas Hofmann, Bernhard Schlkopf und Alexander J.Smola: *Kernel Methods in Machine Learining.* The Annals of Statistics, 2008.

[18] Tsang, Ivor W., Pak-Min Cheung und James T. Kwok: *Core Vector Machines: Fast SVM Training on Very Large Data Sets.* Journal of Machine Learning Reseach 6, Seiten 363–392, 2005.

[19] Tsang, Ivor W., Andras Kocsor und James T. Kwok: *Simpler Core Vector Machines with Enclosing Balls.* Proceedings of the 24th International Conference on Machine Learning, Corvallis, Seiten 1–8, 2007.

[20] Vladimir Vapnik, Corinna Cortes: *Support-Vector Networks.* Machine Learning, Seiten 273–297, 1995.

[21] Yulia Tsvetkov, Leonid Baytsov, Anatole Gerschman-Eric Nyberg und Chris Dyer: *Metaphor Detection with Cross-Lingua Model Transfer.*

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 16. Juli 2015

Jonas Langenberg