Erweiterung und Evaluierung des modularen Perzeptron-Frameworks Perceptrovement

Extension and evaluation of the modular perceptron-framework Perceptrovement

Master-Thesis von Alexander Heinz aus Darmstadt

Mai 2014

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Erweiterung und Evaluierung des modularen Perzeptron-Frameworks Perceptrovement
Extension and evaluation of the modular perceptron-framework Perceptrovement

Vorgelegte Master-Thesis von Alexander Heinz aus Darmstadt

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Eneldo Loza Mencía

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 27. Mai 2014

(Alexander Heinz)

Inhaltsverzeichnis

Erklärung zur Master-Thesis **1**

Inhaltsverzeichnis **2**

1. Einleitung **4**

1.1. Stand der Technik und Related Work 4

1.2. Überblick 6

2. Der Perzeptron-Algorithmus **7**

2.1. Primalform, Dualform und Kerneltrick 9

3. Varianten des Perzeptron-Algorithmus **10**

3.1. Perceptron with (uneven) Margin 10

3.2. Passive-Aggressive 11

3.3. Pegasos 13

3.4. MICRA 14

3.5. Shrinking Perceptron 16

3.6. Perceptron with Dynamic Margin 18

3.7. Modellauswahl 19

3.8. Reduzierte Epochen 20

3.9. Budget 21

3.10. Unlearning 22

3.11. α -Bound 23

3.12. λ-Trick 24

4. Experimentaufbau und Hyperparametersuche **25**

4.1. Datengenerator 25

4.2. Algorithmusgenerator 26

4.3. Experimentierframework 30

4.4. Lernen von Hyperparametern 32

5. Ergebnisse **33**

5.1. Kombinationen 33

5.2. Instanzenanzahl 35

5.3. Clusterradien 36

5.4. Attributanzahl 37

5.5. Rauschen 38

5.6. Iterationen 39

5.7. Reduzierte Epochen 40

5.8. α-Bound 40

5.9. λ-Trick 41

5.10. Vorhersage 41

5.11. Bias 42

5.12. Budget....42
5.13. Margin- und Updateparameter....43
5.13.1. Dynamic Margin....43
5.13.2. MICRA Margin....43
5.13.3. Shrinking Margin....44
5.13.4. Uneven Margin....44
5.13.5. Constant Update....45
5.13.6. MICRA Update....46
5.13.7. Passive-Aggressive Update....46
5.13.8. Pegasos Update....47
5.13.9. Shrinking Update....47
5.14. Gelernte Hyperparameter....48
5.14.1. Modell der erreichbaren Genauigkeit....48
5.14.2. Modell der Hyperparameter....50
6. Fazit und Ausblick....53
7. Verzeichnisse....54
7.1. Tabellenverzeichnis....54
7.2. Algorithmenverzeichnis....54
7.3. Abbildungsverzeichnis....54
7.4. Literaturverzeichnis....56

1. Einleitung

Der Perzeptron-Algorithmus, in seiner ursprünglichen Form 1958 von Frank Rosenblatt vorgestellt, gehört zur Familie der linearen Klassifizierer. Er arbeitet auf Datensätzen bestehend aus Instanzen, die definiert sind durch ihrer Werte bezüglich einer Reihe festgelegter Attribute, sowie ihrer Zugehörigkeit zu einer von zwei Klassen, in der Regel einer positiven und einer negativen. Die Aufgabe des Perzeptron-Algorithmus besteht darin, eine Hyperebene zu finden, welche die Instanzen beider Klassen voneinander trennt. Mithilfe dieser Hyperebene lassen sich daraufhin neue Instanzen mit unbekannter Klassenzugehörigkeit klassifizieren, indem überprüft wird, auf welcher Seite der Ebene sie sich befinden, und ihnen schließlich diejenige Klasse zugeordnet wird, welche durch diese Seite repräsentiert wird.
Da der Algorithmus jede Instanz einzeln und nacheinander bearbeitet, ist er relativ effizient, erreicht aber nicht die qualitativ besseren Generalisierungseigenschaften von „Large Margin Classifier" wie beispielsweise der Support Vector Machine (SVM), die zusätzlich zu einer korrekt trennenden Hyperebene auch noch deren Abstand zu den ihr nächsten Instanzen maximiert. Dazu hat sie jedoch ein quadratisches Optimisierungsproblem zu lösen, dessen Aufwand sich der Perzeptron-Algorithmus erspart. Erstrebenswert wäre also ein Algorithmus, welcher die Effizienz des Perzeptron-Algorithmus mit den Generalisierungseigenschaften der SVM vereint. Unter anderem mit diesem Ziel vor Augen wurden im Laufe der Jahre viele Variationen und Erweiterungen des Perzeptron-Algorithmus entwickelt. Weitere Aspekte sind zum Beispiel die Behandlung von verrauschten Datensätzen, die Reduzierung der Anzahl der Durchläufe und die Performanz auf Datensätzen mit stark unterschiedlich ausgeprägter Klassenhäufigkeit. Jede dieser Varianten bringt eine Reihe von einstellbaren Parametern mit, durch deren Wahl die Effizienz des jeweiligen Algorithmus entscheidend beeinflusst werden kann. Während bei einigen Parametern die Wahl passender Werte logisch erscheint, ist es bei anderen jedoch schwierig, klare Kriterien für deren Einstellung zu erkennen.
2011 wurde an der Technischen Universität Darmstadt das Framework „Perceptrovement" entwickelt, welches einige dieser Algorithmusvariationen implementiert und sie miteinander kombinierbar macht [K11]. Mithilfe dieses Frameworks lassen sich die entsprechenden Parameter leicht einstellen und austesten. Aufgabe dieser Arbeit ist es nun, Perceptrovement um weitere Perzeptron-Varianten zu erweitern, diese miteinander zu kombinieren und die daraus entstehenden Algorithmen sowie ihre Hyperparameter zu evaluieren. Damit soll überprüft werden, welchen Einfluss einzelne Parameter ausüben, auf welchen Datensätzen welche Parameterkonstellationen die besten Ergebnisse liefern, und wie man daher die jeweiligen Hyperparameter an die Daten anpassen sollte. Mithilfe dieser Ergebnisse sollen daraufhin Modelle erstellt werden, die bereits auf Basis eines Datensatzes bestimmen können, wie die jeweiligen Hyperparameter des anzuwendenden Algorithmus eingestellt werden sollten, um die Genauigkeit der entstehenden Hyperebene zu maximieren.

1.1. Stand der Technik und Related Work

Das „Library for Online Learning Algorithms" LIBOL [HWP12] ist eine Open Source Ansammlung von online arbeitenden Lernalgorithmen, welche sich mit den in dieser Arbeit behandelten Algorithmen teilweise überschneidet. Es bietet eine breite Auswahl an Klassifikationsalgorithmen und ermöglicht deren Vergleich. Da in dieser Arbeit allerdings auch die Kombination einzelner Module verschiedener Algorithmen evaluiert werden und speziell Perzeptron-ähnliche Algorithmen untersucht werden sollen, wurde stattdessen das Framework „Perceptrovement" verwendet. Für eine Einordnung der Ergebnisse

in ein generelleres Spektrum an Lernalgorithmen oder eine Untersuchung weiterer Algorithmen ist LIBOL zu empfehlen.
Das Framework „Perceptrovement“ wurde 2011 im Rahmen einer Bachelorarbeit an der Technischen Unversität Darmstadt entwickelt. Es ermöglicht das Konfigurieren und Evaluieren einiger Perzeptron-ähnlicher Algorithmen mithilfe der maschinellen Lernplattform WEKA [HFH+09]. Es ist modular aufgebaut und erlaubt damit auch das Kombinieren einzelner Module verschiedener Algorithmen. Zu Beginn dieser Arbeit enthielt Perceptrovement die folgenden Algorithmen und Techniken:

- Pegasos (Primal Estimated sub-GrAdient SOlver for SVM): Ein Perzeptronalgorithmus, der durch das Angehen des Optimierungsproblem der SVM mithilfe des stochastischen Subgradienten entsteht
- MICRA (Mistake-controlled rule algorithm**):** Ein Perzeptronalgorithmus, bei dem Lernrate und Margin von der Anzahl der bisherigen Updates abhängen
- Reduzierte Epochen, in denen der jeweilige Algorithmus exklusiv zuvor missklassifizierte Instanzen behandelt
- PA (Passive Aggressive): Ein Perzeptronalgorithmus, bei dem sich der Gewichtsvektor neuen Instanzen anpasst, dabei aber nur minimal geändert wird
- PAUM (Perceptron with uneven margins): Ein Perzeptronalgorithmus, der unterschiedliche Margins für positive und negative Instanzen verwendet
- Budget für Supportvektoranzahl, wobei eine Obergrenze der Anzahl an Supportvektoren definiert wird und überschüssige entfernt oder auf die restlichen projeziert werden
- Longest surviving und Voting perceptrons: Techniken, bei denen für die Vorhersage der Klasse nicht mehr (nur) die zuletzt gefundene Hyperebene verwendet wird
- Primal- und Dualform: Zwei Varianten, den Gewichtsvektor darzustellen
- Kernel: Die Möglichkeit, durch die Verwendung von Kernelfunktionen auch hochdimensionelle Featurefunktionen ohne stark steigende Rechenkosten zu ermöglichen

Eine ausführliche Auflistung findet sich unter
http://www.ke.tu-darmstadt.de/resources/perceptrovement.
Folgende Algorithmen und Techniken wurden im Rahmen dieser Arbeit Perceptrovement hinzugefügt:

- Shrinking Perceptron: Ein Perzeptronalgorithmus, welcher den Gewichtsvektor vor jedem Update schrumpft und somit das jeweils aktuelle Update gegenüber vergangenen höher gewichtet
- Perceptron with Dynamic Margin: ein Perzeptronalgorithmus, welcher den Marginparameter dynamisch an Gewichtsvektor und Updateanzahl anpasst, um sich der maximal möglichen Margin anzunähern
- Unlearning: Eine Technik, bei der bereits dem Gewichtsvektor hinzugefügte Supportvektoren wieder entfernt werden können, sobald sie von der jeweiligen Hyperebene genügend Abstand besitzen

Alle aufgelisteten Algorithmen und Techniken werden in dieser Arbeit erläutert, um schließlich ausgiebig evaluiert zu werden.

1.2. Überblick

Kapitel 2 wird zunächst die Grundform des Perzeptron-Algorithmus vorstellen und dessen Eigenschaften besprechen. In Kapitel 3 werden daraufhin die neueren Varianten erklärt. Kapitel 4 behandelt den Aufbau der Experimente zur empirischen Evaluation der Hyperparameter sowie das Verfahren, geeignete Hyperparameter automatisch zu bestimmen. Die dabei entstandenen Ergebnisse werden in Kapitel 5 vorgestellt. Schließlich enthält Kapitel 6 ein Fazit sowie einen Ausblick auf mögliche weitere Betätigungsfelder auf diesem Gebiet.

2. Der Perzeptron-Algorithmus

Der Perzeptron-Algorithmus in seiner Ursprungsform arbeitet im Kontext des überwachten Lernens auf Instanzen, deren Klassenzugehörigkeit während des Lernprozesses bekannt sind. Ein Trainingsdatensatz besteht damit aus einer Menge an Instanzen der Form (**x**, y), mit Attributwerten **x** aus der Menge aller möglichen Attributwerte $\mathbb{X}$[1] sowie Klasse $y \in \{1, -1\}$. Oftmals werden für den Algorithmus die Attribute nicht direkt, sondern mithilfe einer Featurefunktion $\Phi : \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$ aus ihnen extrahierte Feature verwendet. Damit ist es möglich, die Instanzen in einen höherdimensionalen Raum zu überführen, um lineare Separierbarkeit zu ermöglichen. Als anzupassendes Modell verwendet der Algorithmus eine gewichtete lineare Kombination dieser Feature:

$$f(x) = w_1 \phi_1(\boldsymbol{x}) + w_2 \phi_2(\boldsymbol{x}) + w_3 \phi_3(\boldsymbol{x}) + \ldots = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) \tag{2.1}$$

Durch ein zusätzliches konstantes Feature $\Phi_0 = 1$ kann mit w_0 ein Bias hinzugefügt werden. Modellanpassungen entstehen durch Veränderungen des Gewichtsvektors **w**. Ziel ist es, einen Gewichtsvektor zu finden, der für jede Instanz ($\mathbf{x}_n$, y_n) des Trainingsdatensatz die Forderung

$$y_n(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_n)) > 0 \tag{2.2}$$

erfüllt. Eine durch einen solchen Gewichtsvektor aufgespannte Hyperebene trennt die Trainingsinstanzen gemäß ihrer Klassenzugehörigkeit korrekt. Zur Messung der Distanz des aktuellen Modells zu einem solchen korrekten **w** dient das „Perzeptron Kriterium“ [B09], eine Fehlerfunktion, welche für alle missklassifizierten Instanzen, gesammelt in der Indexmenge $\mathbb{M}$, ihre Abweichung von (2.2) aufaddiert:

$$E_P(\boldsymbol{w}) = -\sum_{n \in \mathbb{M}} \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_n) y_n \tag{2.3}$$

Um diese Fehlerfunktion zu minimieren wird mithilfe des stochastischen Gradientenabstiegs eine Instanz ($\mathbf{x}_n$, y_n) betrachtet und folgender Updateschritt des Gewichtsvektors zum Zeitpunkt t durchgeführt:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla E_P(\boldsymbol{w}) = \boldsymbol{w}_t + \eta \boldsymbol{\phi}(\boldsymbol{x}_n) y_n \tag{2.4}$$

Die Lernrate η wird dabei im originalen Perzeptron-Algorithmus auf eins gesetzt. Dieser Updateschritt verringert den Anteil des Fehlers, für den die Instanz ($\mathbf{x}_n$, y_n) verantwortlich ist, aufgrund der Beobachtung

$$-\boldsymbol{w}_{t+1}^T \boldsymbol{\phi}(\boldsymbol{x}_n) y_n = -\boldsymbol{w}_t^T \boldsymbol{\phi}(\boldsymbol{x}_n) y_n - (\boldsymbol{\phi}(\boldsymbol{x}_n) y_n)^T \boldsymbol{\phi}(\boldsymbol{x}_n) y_n < -\boldsymbol{w}_t^T \boldsymbol{\phi}(\boldsymbol{x}_n) y_n \tag{2.5}$$

Dies ist allerdings nur eine lokale Optimierung, da nur der Fehleranteil der aktuell betrachteten Instanz ($\mathbf{x}_n$, y_n) garantiert verringert wird. Der Fehleranteil anderer Instanzen könnte bei einem solchen Updateschritt sogar wieder erhöht werden. Dass der Algorithmus auf linear separierbaren Datensätzen mit diesem Update trotzdem zu einem Modell konvergiert, welches alle Trainingsdaten korrekt separiert, besagt Novikoffs Theorem [FS99]:

[1] In der Regel handelt es sich um numerische oder binärische Attribute, nominale sollten vorher in numerische umgewandelt werden.

Novikoffs Theorem: *Sei $\{(\mathbf{x}_1,y_1),\ldots,(\mathbf{x}_n,y_n)\}$ eine Menge an linear separierbaren Instanzen mit Attributwerten $\mathbf{x}_i$ und Klasse y_i mit $\|\mathbf{x}_i\| \leq R$. Aufgrund der linearen Separierbarkeit existiert ein Vektor $\mathbf{u}$ mit $\|\mathbf{u}\|=1$ und $y_i(\mathbf{u} * \mathbf{x}_i) \geq \gamma$ für alle Instanzen. Ist dies der Fall, so beträgt die Anzahl der Updateschritte bis zum Erreichen einer korrekt trennenden Hyperebene maximal $(R/\gamma)^2$.*

Damit ergibt sich folgendes Prozedere des Algorithmus: Zunächst wird der Gewichtsvektor auf den Nullvektor initialisiert. Die Instanzen werden einzeln durchgegangen und jeweils auf die Bedingung (2.2) überprüft. Wird sie eingehalten, findet keine Modellanpassung statt, wird sie verletzt, so wird mithilfe des Updateschritts (2.4) die aktuelle Instanz gemäß ihrer korrekten Klassifikation auf den Gewichtsvektor aufaddiert. Wurden alle Trainingsinstanzen behandelt ist eine sogenannte „Epoche" abgeschlossen und die nächste beginnt wieder mit der ersten Instanz. Dies wiederholt sich so lange, bis schließlich eine Epoche ohne Anpassung des Gewichtsvektors ausgeführt wurde, da für alle Instanzen die Bedingung (2.2) erfüllt ist und die Fehlerfunktion (2.3) damit auf null minimiert wurde. Der zu diesem Zeitpunkt gefundene Gewichtsvektor spannt eine Hyperebene auf, welche alle Trainingsinstanzen korrekt klassifiziert.
Im ursprünglichen Perzeptron-Algorithmus sind zwei zentrale Aspekte zu finden, welche auch alle hier vorgestellten Varianten in abgewandelter Form implementieren: Die Margin und das Update. Die Margin, hier die Bedingung (2.2), steuert die Missklassifikationskondition und bestimmt damit, wann es zu einer Modellanpassung kommt. Das Update beschreibt die Änderungen am Gewichtsvektor, die bei Missklassifikation vorzunehmen sind, was Updateschritt (2.4) entspricht. Da in Perceptrovement diese beiden Module aus verschiedenen Algorithmen miteinander kombiniert werden können, wurden die dort implementierten Bereiche im Code farblich markiert: Blau für die Margin, rot für das Update. Zusätzlich wird als Kommentar noch der Name des jeweiligen Moduls in Perceptrovement genannt.

Algorithmus 2.1 Perzeptron-Algorithmus nach Rosenblatt

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\})$, linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$
Ausgabe: Gewichtsvektor **w**, welcher korrekt separierende Hyperebene aufspannt
Initialisierung: $\mathbf{w}_0 := \mathbf{0}$, $t := 0$

1: **repeat**
2: **for** $i := 1$ to N **do**
3: **if** $y_i(\mathbf{w}^T \Phi(\mathbf{x}_i)) \leq 0$ **then**
4: $\mathbf{w}_{t+1} := \mathbf{w}_t + y_i \Phi(\mathbf{x}_i)$ *// ConstantUpdate (hier Spezialfall mit Lernrate 1)*
5: $t := t+1$
6: **end if**
7: **end for**
8: **until** kein update in for-Schleife
9: **return** $\mathbf{w}_t$

2.1. Primalform, Dualform und Kerneltrick

Anstatt den Gewichtsvektor wie bisher in seiner sogenannten Primalform darzustellen, lässt er sich alternativ auch in der Dualform ausdrücken. Dabei wird von der Beobachtung Gebrauch gemacht, dass er aus der Summe der bei Updates ihm hinzugefügten Instanzen besteht. Speichert man also die Koeffizienten in einem Vektor **a**, wobei a_n die Anzahl der Updateschritte auf Instanz n zählt, lässt sich der Gewichtsvektor folgendermaßen darstellen:

$$\boldsymbol{w} = \sum_{n=1}^{N} a_n \boldsymbol{\Phi}(\boldsymbol{x_n}) \tag{2.1.1}$$

Dies hat den Vorteil, dass nun nur die Koeffizienten im Speicher gehalten werden müssen, statt eines abhängig von der Dimension der Trainingsinstanzen potentiell sehr langen Gewichtsvektoren. Ersetzt man beim Testen der Missklassifikationskondition wie in Zeile 3 von Algorithmus 2.1 den Gewichtsvektor durch seine duale Darstellung (2.1.1), dann werden die Instanzen nicht mehr direkt, sondern nur noch als Skalarprodukt zweier Featurevektoren behandelt:

$$y_i(\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{\Phi}(\boldsymbol{x_i})) = y_i\left(\sum_{n=1}^{N} a_n \left(\boldsymbol{\Phi}(\boldsymbol{x_n}) \boldsymbol{\Phi}(\boldsymbol{x_i})\right)\right) \tag{2.1.2}$$

Ist dies der Fall, dann erlaubt der Kerneltrick laut [B09] dieses Skalarprodukt zu ersetzen durch eine Kernelfunktion k, welche die eingegebenen Instanzen direkt mit dem Ergebnis des Skalarproduktes ihrer Featurevektoren verknüpft:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\Phi}(\boldsymbol{x}) \boldsymbol{\Phi}(\boldsymbol{x}') \tag{2.1.3}$$

Durch Verwenden einer Kernelfunktion wird das bei hoher Dimensionalität von Φ aufwendige Mapping einer Instanz in den Featureraum gespart und das Nutzen von Featurefunktionen beliebiger Dimensionalität ermöglicht.

3. Varianten des Perzeptron-Algorithmus

Nachdem der Perzeptron-Algorithmus in seiner Ursprungsform vorgestellt wurde, folgt nun eine Auswahl an Erweiterungen und Variationen, die über die Jahre zur Verbesserung des Algorithmus vorgeschlagen und in Perceptrovement implementiert wurden. Dabei werden zunächst vollständige auf Basis des Perzeptron entwickelte Algorithmen behandelt, die jeweils durch die zur Veränderung gegenüber des ursprünglichen Algorithmus führenden Motivation und den veränderten Algorithmus selbst in Pseudocode erklärt werden. Schließlich folgen kleinere Modifikationen, die mit den vorgestellten Algorithmen kombiniert werden können.

3.1. Perceptron with (uneven) Margin

Ein klarer Unterschied des Perzeptron-Algorithmus gegenüber der SVM liegt in seiner Akzeptanz jeder Hyperebene, welche die Trainingsdaten korrekt trennt. Bei der SVM herrscht eine striktere Anforderung: Die Hyperebene muss zusätzlich zur korrekten Separation auch noch den Abstand zu den ihr am nächsten gelegenen Instanzen, ihren namensgebenden „Supportvektoren“, maximieren. Aus dieser Forderung erwachsen die verbesserten Generalisierungseigenschaften der SVM gegenüber des Perzeptron-Algorithmus. Es wäre also sinnvoll, den Perzeptron-Algorithmus so zu modifizieren, dass er eine ähnliche Eigenschaft aufweist. Dies geschieht beim „Perceptron with Margin“ [LZH+02]. Um einen Mindestabstand zur Hyperebene zu realisieren, wird die Forderung

$$y_i\left(\boldsymbol{w}^T \boldsymbol{\Phi}(\boldsymbol{x}_i)\right) > 0 \qquad (3.1.1)$$

welche eine korrekte Klassifizierung der Instanz $(\mathbf{x}_i, y_i)$ repräsentiert, modifiziert zu

$$y_i\left(\boldsymbol{w}^T \boldsymbol{\Phi}(\boldsymbol{x}_i)\right) > b \qquad (3.1.2)$$

mit $b \geq 0$. Eine Instanz gilt durch diese Modifikation erst dann als korrekt klassifiziert, wenn sie einen durch b festgelegten Mindestabstand zur Hyperebene einhält.
Als weitere Spezialisierung dieser Festlegung eines Mindestabstandes wurde in [LZH+02] gezeigt, dass bei Datensätzen mit einer stark ungleichen Verteilung ihrer Instanzen auf die beiden Klassen der Perzeptron-Algorithmus von unterschiedlichen Margins für positive und negative Instanzen profitiert. Dabei werden statt eines klassenunabhängigen b die beiden Mindestabstände τ_{+1} und τ_{-1} zur Klassifizierung von positiven bzw. negativen Instanzen festgelegt. Dies führt zum „perceptron algorithm with uneven margins (PAUM)“.

Algorithmus 3.1 PAUM

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),...,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\}$), linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$

Eingabe: Lernrate $\eta \in \mathbb{R}^+$

Eingabe: Margin Parameter $\tau_{+1}, \tau_{-1} \in \mathbb{R}^+$

Ausgabe: Gewichtsvektor **w**, welcher korrekt separierende Hyperebene aufspannt

Initialisierung: $\mathbf{w}_0 := \mathbf{0}$, $t := 0$

1: **repeat**
2: **for** i := 1 to N **do**
3: **if** $y_i (\mathbf{w}^T \Phi(\mathbf{x}_i)) \leq \tau_{y_i}$ **then** *// UnevenMargin*
4: $\mathbf{w}_{t+1} := \mathbf{w}_t + \eta y_i \Phi(\mathbf{x}_i)$
5: $t := t + 1$
6: **end if**
7: **end for**
8: **until** kein update in for-Schleife
9: **return** $\mathbf{w}_t$

3.2. Passive-Aggressive

Beim „Passive-Aggressive (PA)“ Algorithmus [CDK+06] handelt es sich um eine Perzeptron Variante, bei welcher der Update Schritt, also die Veränderung des Gewichtsvektors aufgrund einer fehlerhaft klassifizierten Instanz, modifiziert wurde. Basierend auf dem Grundgedanken, dass die bei einem Update entstehende Hyperebene die aktuelle Instanz korrekt klassifizieren, sich aber dabei minimal von der bisherigen unterscheiden sollte, entsteht folgendes Optimierungsproblem:

$$\boldsymbol{w}_{t+1} = \underset{\boldsymbol{w} \in \mathbb{R}^N}{\operatorname{argmin}} \frac{1}{2} \left\| \boldsymbol{w} - \boldsymbol{w}_t \right\|^2 \quad \textit{so dass} \quad \mathcal{L}(\boldsymbol{w}; (\boldsymbol{x}_t, y_t)) = 0 \tag{3.2.1}$$

$\mathcal{L}$ steht hierbei für die „Hinge-Loss“-Funktion

$$\mathcal{L}(\boldsymbol{w}; (\boldsymbol{x}, y)) = \begin{cases} 0 & y(\boldsymbol{w} \cdot \boldsymbol{x}) \geq 1 \\ 1 - y(\boldsymbol{w} \cdot \boldsymbol{x}) & \textit{andernfalls} \end{cases} \tag{3.2.2}$$

welche null ergibt, wenn die betrachtete Instanz sich auf der korrekten Seite der Hyperebene befindet und zu ihr einen Mindestabstand einhält. Ansonsten gibt sie die Entfernung zu diesem Zustand wieder. Als Lösung dieses Optimierungsproblems erhält man die Updateregel

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \tau_t y_t \boldsymbol{x}_t \quad \textit{mit} \quad \tau_t = \frac{\mathcal{L}_t}{\left\| \boldsymbol{x}_t \right\|^2} \tag{3.2.3}$$

mit $\mathcal{L}_t = \mathcal{L}(\boldsymbol{w}_t; (\boldsymbol{x}_t, y_t))$. Damit entsteht ein Update, welches bei korrekter Klassifikation und eingehaltenem Abstand den alten Gewichtsvektor beibehält (Passiv), andernfalls jedoch den Gewichtsvektor gerade soweit ändert, dass diese Bedingungen eingehalten werden (Aggressiv). Durch diese aggressive Forderung einer korrekten Klassifikation des aktuellen Beispiels haben allerdings auch fehlerhafte Instanzen aus verrauschten Datensätzen großen Einfluss auf das Update. Um dieses Problem zu lindern werden in [CDK+06] die Varianten „PA-I“ und „PA-II“ vorgeschlagen, die, ähnlich zur SVM, eine „slack variable“ ξ einführen. Die dadurch erreichte Lockerung der Forderung des aggressiven Updates lässt sich mithilfe des Aggressivitätsparameters C steuern.

$$\text{PA-I:} \quad \boldsymbol{w}_{t+1} = \underset{\boldsymbol{w} \in \mathbb{R}^N}{\operatorname{argmin}} \frac{1}{2} \|\boldsymbol{w} - \boldsymbol{w}t\|^2 + C\zeta \quad so\, dass \quad \mathcal{L}(\boldsymbol{w}; (\boldsymbol{x}_t, y_t)) \leq \zeta \quad und \quad \zeta \geq 0 \tag{3.2.4}$$

$$\text{PA-II:} \quad \boldsymbol{w}_{t+1} = \underset{\boldsymbol{w} \in \mathbb{R}^N}{\operatorname{argmin}} \frac{1}{2} \|\boldsymbol{w} - \boldsymbol{w}t\|^2 + C\zeta^2 \quad so\, dass \quad \mathcal{L}(\boldsymbol{w}; (\boldsymbol{x}_t, y_t)) \leq \zeta \tag{3.2.5}$$

Daraus resultieren die folgende Updates für PA-I und PA-II:

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t + \tau_t y_t \boldsymbol{x}_t \\ \tau_t &= min\left\{C, \frac{\mathcal{L}_t}{\|\boldsymbol{x}\|^2}\right\} \quad (PA-I) \\ \tau_t &= \frac{\mathcal{L}_t}{\|\boldsymbol{x}_t\|^2 + \frac{1}{2C}} \quad (PA-II) \end{aligned} \tag{3.2.6}$$

Algorithmus 3.2 Passive-Aggressive Algorithmus

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),...,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\})$, linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$

Eingabe: Aggressivitätsparameter $C \in \mathbb{R}^+$

Ausgabe: Gewichtsvektor **w**, welcher korrekt klassifizierende Hyperebene aufspannt

Initialisierung: $\mathbf{w}_0 := \mathbf{0}$, $t := 0$

1: **repeat**
2:     **for** i := 1 to N **do**
3:         $\mathcal{L}_t := \max\{0, 1 - y_t(\mathbf{w}_t * \Phi(\mathbf{x}_i)\}$ *// PAUpdate*
4:         $\mathbf{w}_{t+1} := \mathbf{w}_t + \tau_t y_i \Phi(\mathbf{x}_i)$
5:         $t := t + 1$
7:     **end for**
8: **until** kein update in for-Schleife
9: **return** $\mathbf{w}_t$

3.3. Pegasos

Anstatt wie bei den bisherigen Varianten sich ausgehend vom ursprünglichen Perzeptron-Algorithmus den Eigenschaften der SVM anzunähern, setzt der „Primal Estimated sub-GrAdient Solver for SVM (Pegasos)“ [SSS07] direkt bei der SVM an und erhält schließlich einen Perzeptron-ähnlichen Algorithmus. Er approximiert das Minimierungsproblem der SVM

$$min_w \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{m}\sum_{(\boldsymbol{x},y)\epsilon\mathbb{Z}} \mathcal{L}(\boldsymbol{w};(\boldsymbol{x},y)) \tag{3.3.1}$$

mit der in (3.2.2) beschriebenen „Hinge-Loss“-Funktion $\mathcal{L}$ mithilfe des stochastischen Subgradienten, betrachtet also jeweils nur eine einzelne Instanz ($\mathbf{x}_i$, y_i) zum Zeitpunkt t:

$$min_w \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \mathcal{L}(\boldsymbol{w};(\boldsymbol{x}_i,y_i)) \tag{3.3.2}$$

Zur Minimierung wird der Subgradient dieser Funktion verwendet:

$$\nabla_t = \lambda\boldsymbol{w}_t - \mathbb{1}[y_i(\boldsymbol{w}_t\boldsymbol{x}_i) < 1]\, y_i\boldsymbol{x}_i \tag{3.3.3}$$

Hierbei steht $\mathbb{1}$ für die Indikatorfunktion, welche dem Wahrheitswert ihres Argumentes eine 1 bzw. 0 zuweist:

$$\mathbb{1}[b] = \begin{cases} 1 & \text{falls } b = \text{wahr} \\ 0 & \text{falls } b = \text{falsch} \end{cases} \tag{3.3.4}$$

Um die Minimierung nun voranzutreiben, erfährt der Gewichtsvektor ein Update in Gegenrichtung des Subgradienten (3.3.3), wobei die Schrittgröße von der Anzahl der bisherigen Updates t sowie einem angebbaren Parameter λ abhängt:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{1}{\lambda t}\nabla_t = \left(1 - \frac{1}{t}\right)\boldsymbol{w}_t + \frac{1}{\lambda t}\mathbb{1}[y_i(\boldsymbol{w}_t\boldsymbol{x}_i) < 1]\, y_i\boldsymbol{x}_i \tag{3.3.5}$$

Um den Gewichtsvektor schließlich in den Raum der optimalen Lösung, einem Ball mit Radius $1/\sqrt{\lambda}$, zu überführen, schlägt [SSS07] nach einem Update den folgenden Projektionsschritt vor:

$$\mathbf{w}_{t+1} = \min\left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|}\right\}\mathbf{w}_{t+1} \tag{3.3.6}$$

Dies führt zu folgendem Algorithmus:

Algorithmus 3.3 Pegasos

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\})$, linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$

Eingabe: Schrittgröße $\lambda \in \mathbb{R}^+$

Ausgabe: Gewichtsvektor **w**, welcher korrekt klassifizierende Hyperebene aufspannt

Initialisierung: $\mathbf{w}_0 := \mathbf{0}$, $t := 0$

1: **repeat**
2: **for** i := 1 to N **do**
3: $\eta_t := \frac{1}{\lambda t}$
4: **if** $y_i(\mathbf{w}^T \Phi(\mathbf{x}_i)) < 1$ **then**
5: $\mathbf{w}_{t+1} := (1 - \eta_t \lambda)\mathbf{w}_t + \eta y_i \Phi(\mathbf{x}_i)$ *// Pegasos*
6: **else**
7: $\mathbf{w}_{t+1} := (1 - \eta_t \lambda)\mathbf{w}_t$
8: **end if**
9: $\mathbf{w}_{t+1} := \min\left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+1}\|}\right\}\mathbf{w}_{t+1}$
10: $t := t + 1$
11: **end for**
12: **until** kein update in for-Schleife
13: **return** $\mathbf{w}_t$

3.4. MICRA

Ein Perzeptron-Algorithmus mit einer konstanten Lernrate η sorgt aufgrund des Wachstums der euklidischen Norm des Gewichtsvektors $\|\mathbf{w}\|$ im Laufe der Ausführung dafür, dass der Effekt eines Updates des Gewichtsvektors mit der Zeit immer weiter abnimmt. In [TST07] wird dies beschrieben mit einer von der maximalen euklidischen Norm R der Trainingsdaten abhängigen „effektiven Lernrate“ η_{eff}

$$\eta_{eff_t} = \eta_t \frac{R}{\|\boldsymbol{w}_t\|} \tag{3.4.1}$$

die den tatsächlichen Einfluss eines Updates auf den Gewichtsvektor nach t Updates misst. Das Sinken dieses Einflusses soll nun im „Mistake-Controlled Rule Algorithm (MICRA)“ von der Anzahl der Fehler und damit der Anzahl der Updates abhängen. Um dies zu erreichen wird die effektive Lernrate auf

$$\eta_{eff_t} = \frac{\eta}{t^\zeta} \tag{3.4.2}$$

gesetzt. Parameter ζ ermöglicht dabei eine Steuerung des Verfalls der Lernrate in Abhängigkeit von t. Durch Gleichsetzen von (3.4.1) und (3.4.2) erhält man eine Updateregel für die aktuelle Lernrate η_t:

$$\eta_t = \|\boldsymbol{w}_t\| \frac{\eta}{R\, t^{\zeta}} \tag{3.4.3}$$

Ähnlich dazu soll auch die geforderte Margin von der Anzahl der Updates abhängen, mit ϵ als Parameter, der die Verfallsrate bestimmt:

$$\beta_t = \|\boldsymbol{w}_t\| \frac{\beta}{t^{\epsilon}} \tag{3.4.4}$$

Wird die Lernrate abhängig von Startmargin β gewählt

$$\eta = \eta_0 \left(\frac{\beta}{R}\right)^{-\delta} \tag{3.4.5}$$

und erfüllen die Parameter die Forderung

$$0 < \epsilon\,\delta + \zeta < 1 \tag{3.4.6}$$

dann strebt MICRA für $\frac{\beta}{R} \to \infty$ laut [TST07] zur Lösung mit maximaler Margin.

Algorithmus 3.4 MICRA

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),...,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\})$, linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \to \mathbb{K} \subseteq \mathbb{R}^D$

Eingabe: Parameter ϵ, δ, ζ, η_0, β

Ausgabe: Gewichtsvektor $\mathbf{w}$, welcher korrekt klassifizierende Hyperebene aufspannt

Definiere: $R = \max_k \|x_k\|$

Initialisierung: $\mathbf{w}_1 := \Phi(\mathbf{x}_1)$, $t := 1$, $\eta_1 := \|w_1\| \frac{\eta}{R}$, $\beta_1 := \|w_1\| \beta$, $\eta := \eta_0 \left(\frac{\beta}{R}\right)^{-\delta}$

```
 1: repeat
 2:   for i := 1 to N do
 3:     if y_i (w^T Φ(x_i)) < β_t then                 // MicraMargin
 4:       w_{t+1} := w_t + η_t y_i Φ(x_i)              // MicraUpdate
 5:       t := t + 1
 6:       η_t := ||w_t|| η / (R t^ζ)
 7:       β_t := ||w_t|| β / t^ε
 8:     end if
 9:   end for
10: until kein update in for-Schleife
11: return w_t
```

3.5. Shrinking Perceptron

Geht man das Minimierungsproblem der SVM mithilfe des stochastischer Subgradienten an, wie es beispielsweise Pegasos (3.3.) macht, erhält man eine Updateregel, welche den Gewichtsvektor vor dem Update zunächst schrumpft:

$$\boldsymbol{w}_{t+1} = (1-\eta_t \lambda)\boldsymbol{w}_t + \eta\, y_i \boldsymbol{x}_i \tag{3.5.1}$$

Der Effekt dieses Schrumpfens bei jedem Updateschritt ist eine relative Verringerung des Einflusses zurückliegender Updates auf den Gewichtsvektor verglichen mit dem Einfluss des aktuellen. [PT12] greift diese Idee auf und betrachtet sie allgemeiner. Der Effekt verschieden starker Schrumpfungen wird hierbei untersucht, indem Parameter eingeführt werden, die diese Stärke kontrollieren. Dies führt zur Updateregel

$$\boldsymbol{w}_{t+1} = c_t^{(w)} \boldsymbol{w}_t + \eta\, y_i \boldsymbol{x}_i \tag{3.5.2}$$

und Missklassifikationskondition

$$y_i \left(c_t^{(m)} \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}_i \right) \leq b \tag{3.5.3}$$

mit separaten Schrumpffaktoren $0 < c^{(w)}_t, c^{(m)}_t \leq 1$ für Update und Margin, die entweder konstant sind oder von der Anzahl der bisherigen Updates abhängen können. Im zugehörigen Algorithmus werden diese Schrumpfungen äquivalent durch Vergrößerung der Lernrate bzw. Vergrößerung des Margin Thresholds b implementiert, was weniger Rechenaufwand erfordert, als die Elemente des Gewichtsvektoren zu verkleinern. Dabei werden zwei Varianten unterschieden. Beim „Constant Shrinking“ bleiben die Schrumpffaktoren während des Durchlaufs konstant, wie in Algorithmus 3.5 zu sehen.
Alternativ schlägt [PT12] vor, das Schrumpfen in Form von

$$\eta_t = \eta(t+1)^n \tag{3.5.4}$$

bzw.

$$b_t = b(1+n)^n \tag{3.5.5}$$

beim „Variable Shrinking“ von der bisherigen Updateanzahl abhängig zu machen werden, wie Algorithmus 3.6 verdeutlicht.

Algorithmus 3.5 Margin Perceptron with Constant Shrinking

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\})$, linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$

Eingabe: Startlernrate η, Startmargin b, Schrumpffaktoren $c^{(w)}$, $c^{(m)}$

Ausgabe: Gewichtsvektor **w**, welcher korrekt klassifizierende Hyperebene aufspannt

Initialisierung: $\mathbf{w}_0 := \mathbf{0}$, $t := 0$, $\eta_0 := \eta$, $b_0 := c^{(m)} b$

1: **repeat**
2:   **for** i := 1 to N **do**
3:     **if** $y_i (\mathbf{w}^T \Phi(\mathbf{x}_i)) \leq b_t$ **then**   *// ShrinkingMargin (constant)*
4:       $\mathbf{w}_{t+1} := \mathbf{w}_t + \eta_t y_i \Phi(\mathbf{x}_i)$ *// ShrinkingUpdate (constant)*
5:       $\eta_{t+1} := \frac{\eta_t}{c^{(w)}}$
6:       $b_{t+1} := \frac{b_t}{c^{(m)}}$
7:       $t := t+1$
8:     **end if**
9:   **end for**
10: **until** kein update in for-Schleife
11: **return** $\mathbf{w}_t$

Algorithmus 3.6 Margin Perceptron with Variable Shrinking

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\})$, linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$

Eingabe: Startlernrate η, Startmargin b, Exponent n

Ausgabe: Gewichtsvektor **w**, welcher korrekt klassifizierende Hyperebene aufspannt

Initialisierung: $\mathbf{w}_0 := \mathbf{0}$, $t := 0$

1: **repeat**
2:   **for** i := 1 to N **do**
3:     $t_n := (t+1)^n$
4:     $\eta_t := \eta\, t_n$
5:     $b_t := b\, t_n$
6:     if $y_i (\mathbf{w}^T \Phi(\mathbf{x}_i)) \leq b_t$ **then**   *// ShrinkingMargin (variable)*
7:       $\mathbf{w}_{t+1} := \mathbf{w}_t + \eta_t y_i \Phi(\mathbf{x}_i)$ *// ShrinkingUpdate (variable)*
8:       $t := t+1$
9:     **end if**
10:   **end for**
11: **until** kein update in for-Schleife
12: **return** $\mathbf{w}_t$

3.6. Perceptron with Dynamic Margin

Betrachtet man einen Perzeptron-Algorithmus, welcher in seiner Missklassifikationskondition einen konstanten, mit der Norm des Gewichtsvektors normalisierten Mindestabstand zur Hyperebene fordert, gilt die Forderung, dass dieser Abstand kleiner als die maximal erreichbare Margin sein sollte, um Konvergenz zu gewährleisten. In den meisten praktischen Fällen ist die maximal erreichbare Margin aber zu Beginn nicht bekannt. Um dieses Problem zu umgehen verwendet der Algorithmus „Perceptron with Dynamic Margin“ [PT11] die Beobachtung, dass die maximal erreichbare Margin γ zu einem beliebigen Zeitpunkt des ursprünglichen Perzeptron-Algorithmus nach oben durch die euklidische Norm des Gewichtsvektors und die Updateanzahl t begrenzt wird:

$$\gamma \leq \frac{\|\boldsymbol{w}_t\|}{t} \tag{3.6.1}$$

Dies kann genutzt werden, um bei einer Missklassifikationskondition, die das Einhalten eines durch ϵ bestimmten Teilabstands der maximal möglichen Margin γ fordert

$$y_i(\boldsymbol{w}^T \boldsymbol{\Phi}(\boldsymbol{x}_i)) \leq (1-\epsilon)\gamma\|\boldsymbol{w}_t\| \tag{3.6.2}$$

dieses unbekannte γ zu ersetzen durch seine dynamische Obergrenze (3.6.1):

$$y_i(\boldsymbol{w}^T \boldsymbol{\Phi}(\boldsymbol{x}_i)) \leq (1-\epsilon)\frac{\|\boldsymbol{w}_t\|^2}{t} \tag{3.6.3}$$

Zusammen mit der Beobachtung, dass $\frac{\|\mathbf{w}_t\|}{t}$ sich bei steigender Anzahl der Updates langfristig an γ annähert und somit tatsächlich eine Annäherung an die bestmögliche Margin stattfindet, stellt [PT11] den Algorithmus „Perceptron with Dynamic Margin“ (Algorithmus 3.7) vor.

Algorithmus 3.7 Perceptron with Dynamic Margin

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\})$, linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$

Eingabe: Genauigkeitsparameter ϵ

Ausgabe: Gewichtsvektor **w**, welcher korrekt klassifizierende Hyperebene aufspannt

Initialisierung: $\mathbf{w}_0 := \mathbf{0}$, $t := 0$, $\beta_0 := 0$

1: **repeat**
2:   **for** $i := 1$ to N **do**
3:     if $y_i (\mathbf{w}^T \Phi(\mathbf{x}_i)) \leq \beta_t$ **then**   *// DynamicMargin*
4:       $\mathbf{w}_{t+1} := \mathbf{w}_t + y_i \Phi(\mathbf{x}_i)$
5:       $t := t + 1$
6:       $\beta_t := (1 - \epsilon) \frac{\|\mathbf{w}\|^2}{t}$
7:     **end if**
8:   **end for**
9: **until** kein update in for-Schleife
10: **return** $\mathbf{w}_t$

3.7. Modellauswahl

Der ursprüngliche Perzeptron-Algorithmus und auch die hier vorgestellten Varianten verwerfen bei jeder Modellanpassung das vorherige Modell und geben schließlich nur den finalen Gewichtsvektor zurück. Doch auch in den verworfenen Modellen befinden sich Informationen, welche bei der Vorhersage der Klasse einer neuen Instanz verwendet werden können.
Anstatt des letzten Modells gibt zum Beispiel die Variante „Longest Survivor“ [KW07] denjenigen Gewichtsvektor aus, welcher während des Durchlaufs des Algorithmus die meisten Instanzen korrekt klassifiziert, bevor er aufgrund einer falschen Klassifizierung verändert wird. Dazu muss neben dem aktuellen Gewichtsvektor auch der derzeit führende sowie dessen Überlebensdauer im Speicher gehalten werden.
Bei der „Voted“-Variante [FS99] werden statt eines Modells alle während eines Durchlaufs entstandenen Gewichtsvektoren gespeichert und zur Vorhersage verwendet. Um der variierenden Vorhersagequalität der einzelnen Modelle Rechnung zu tragen werden diese unterschiedlich gewichtet. Als Qualitätsmerkmal gilt auch hier die Überlebensdauer, gemessen an der Anzahl der Instanzen, die korrekt klassifiziert werden, bis es zur nächsten Modellanpassung kommt. Um die Klasse einer neuen Instanz vorherzusagen findet eine gewichtete Abstimmung aller gespeicherter Modelle statt. Algorithmus 3.8 zeigt beide Varianten.

Algorithmus 3.8 Voted/Longest Survivor Perceptron

Training

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\}$), linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$

Ausgabe: Liste aus Gewichtsvektoren und deren Gewicht $<(\mathbf{w}_1,c_1),\ldots,(\mathbf{w}_t,c_t)>$

Initialisierung: $t := 0$, $\mathbf{w}_0 := 0$, $c_1 := 0$

1: **repeat**
2: **for** i := 1 to N **do**
3: **if** $y_i (\mathbf{w}^T \Phi(\mathbf{x}_i)) > 0$ **then**
4: $c_t := c_t + 1$
5: **else**
6: $c_{t+1} := 1$
7: $\mathbf{w}_{t+1} := \mathbf{w}_t + y_i \Phi(\mathbf{x}_i)$
8: $t := t + 1$
9: **end if**
10: **end for**
11: **until** kein update in for-Schleife
12: **return** $<(\mathbf{w}_1\, c_1),\ldots,(\mathbf{w}_t,\, c_t)>$

Vorhersage Longest Survivor:

Eingabe: Liste aus Gewichtsvektoren und deren Gewicht $<(\mathbf{w}_1,c_1),\ldots,(\mathbf{w}_t,c_t)>$, Instanz **x**

Ausgabe: Vorhergesagte Klasse $\hat{y}$

$$\hat{y} = sign(\boldsymbol{w}_s \cdot \boldsymbol{\Phi}(\boldsymbol{x})) \qquad mit\; s = argmax_{k \in \{1,\ldots,t\}} c_k$$

Vorhersage Voted:

Eingabe: Liste aus Gewichtsvektoren und deren Gewicht $<(\mathbf{w}_1,c_1),\ldots,(\mathbf{w}_t,c_t)>$, Instanz **x**

Ausgabe: Vorhergesagte Klasse $\hat{y}$

$$s = \sum_{i=1}^{k} c_i\, sign(\boldsymbol{w}_i \cdot \boldsymbol{\Phi}(\boldsymbol{x}))$$
$$\hat{y} = sign(s)$$

3.8. Reduzierte Epochen

Neben MICRA (3.4.) wird in [TST07] auch eine Modifikation der Reihenfolge vorgeschlagen, in welcher Instanzen dem Algorithmus vorgelegt werden. Dabei wird nach jeder vollen Epoche mit allen Instanzen der Fokus auf die Problemfälle, also die missklassifizierten Instanzen gelegt, die im sogenannten „active set" $\mathfrak{M}$ gesammelt werden. In einer „reduzierten" Epoche werden dann lediglich diese Instanzen betrachtet und dem Algorithmus vorgelegt. Dies kann mit weiteren reduzierten Epochen auf den jeweils in der vorherigen reduzierten Epoche missklassifizierten Instanzen theoretisch beliebig oft wiederholt werden, solange das active set nicht leer ist. Um den Einfluss von verrauschten Instanzen zu verringern, die sehr lange in $\mathfrak{M}$ verbleiben und die Updates dabei in eine falsche Richtung

lenken könnten, wird hierbei jedoch eine Obergrenze für die Anzahl der reduzierten Epochen festgelegt, welche jeder vollen Epoche folgen dürfen. Ein auf solche Weise modifizierter Perzeptron-Algorithmus terminiert, wenn eine volle Epoche ohne Missklassifizierung auskommt. Algorithmus 3.9 zeigt die Einbettung eines Perzeptron-Algorithmus in ein solches Framework.

Algorithmus 3.9 Perzeptron-Algorithmus mit Reduzierte Epochen

Eingabe: Perzeptron-Algorithmus, Trainingsdatensatz mit N Instanzen und Parameter wie von Perzeptron-Algorithmus gefordert, Obergrenze für die Anzahl reduzierter Epochen R

Ausgabe: Gewichtsvektor **w**, welcher korrekt klassifizierende Hyperebene aufspannt

```
 1: repeat
 2: 𝔐 = {}
 3:    for i := 1 to N do
 4:       Algorithmusspezifischer Margintest und Update, sammle missklassifizierte Instanzen in 𝔐
 5:    end for
 6:    k := 0;
 7:    while (𝔐 nicht leer and k < R) do
 8:       for j:= 1 to size(𝔐) do
 9:          Algorithmusspezifischer Margintest und Update, entferne korrekt klassifizierte Instanzen aus 𝔐
10:       end for
11:    k := k + 1;
12:    end while
13: until kein update in for-Schleife aus Zeile 2
```

3.9. Budget

Eine Eigenschaft von SVMs besteht darin, dass ihre Supportvektoren lediglich aus denjenigen Instanzen bestehen, welche der Hyperebene am nächsten liegen. Bei Perzeptron-Algorithmen bestehen sie dagegen in der Regel aus sämtlichen Instanzen, welche den Gewichtsvektor ausmachen, ihm also bei Updates hinzugefügt wurden[1]. Dies kann vor allem bei verrauschten Datensätzen zu einer hohen Anzahl an Supportvektoren führen, was den zu erbringenden Rechenaufwand während eines Algorithmusdurchlaufs vergrößert. Es ist also auch hier erstrebenswert, sich der SVM anzunähern. Dies geschieht in [WCV10] durch Techniken zum Einhalten eines Budgets für Supportvektoren: Sobald eine festgelegt Obergrenze für deren Anzahl überschritten wurde, wird ein Supportvektor wieder entfernt. Dies wird implementiert indem folgende Prozedur direkt nach einem Update eingeschoben wird:

Prozedur 3.1 Budget

Eingabe: Anzahl bisheriger SV b, Maximal erlaubte Anzahl SV B

1: $b := b + 1$
2: **if** $b > B$ **then**
3: $\quad \mathbf{w}_{t+1} := \mathbf{w}_{t+1} - \Delta \mathbf{w}_t$
4: $\quad b := b - 1$
5: **end if**

[1] In der Regel multipliziert mit der Lernrate

Hierbei muss zunächst geklärt werden, welcher Supportvektor bei einer solchen Überschreitung des Budgets zu entfernen ist. Vorgeschlagen und in Perceptrovement implementiert sind dabei die folgenden Heuristiken:

- SV mit geringstem Einfluss auf den Gewichtsvektor
- SV mit geringstem Koeffizienten[1] innerhalb des Gewichtsvektors
- Ältester SV
- Neuester SV
- Per Zufall bestimmter SV
- SV mit geringster Norm
- SV mit größtem Abstand zur Hyperebene (auf der jeweils korrekten Seite)

Weiterhin ist die Art des Entfernens zu wählen. Hierbei sind zwei Varianten aus [WCV10] in Perceptrovement implementiert: Beim einfachen **Entfernen** wird der Koeffizient des ausgewählten Supportvektoren auf Null gesetzt und damit der Einfluss dieser Instanz auf den Gewichtsvektor vollständig rückgängig gemacht. Das gleiche geschieht auch bei der **Projektion**, wobei dabei jedoch zusätzlich durch Anpassen der Koeffizienten der übrigen SV das Entfernen so gut wie möglich kompensiert wird.

3.10. Unlearning

Wie beim Einhalten eines Budgets wird auch beim „Margin Perceptron with Unlearning“ [PT10] der Fokus darauf gelegt, einmal dem Gewichtsvektor hinzugefügte Instanzen im Laufe des Algorithmus wieder zu entfernen. Statt eine Obergrenze der Anzahl an SV festzulegen wird hier jedoch eine andere Technik verwendet: Ein SV sollte entfernt werden, wenn er von der aktuellen Hyperebene „gut genug“ klassifiziert wird. Als Messgrundlage für die Qualität der Klassifizierung dient dabei der Abstand zur Hyperebene. Dies entspricht einer der Heuristiken beim Einhalten eines Budgets aus (3.9.), wird hier jedoch nicht erst beim Überschreiten einer Obergrenze der Supportvektorenanzahl eingesetzt, sondern bei jeder zu behandelnden Instanz. Auch bei dieser Technik erkennt man wieder einen Annäherungsversuch zur SVM, bei der nur die der Hyperebene nächsten Instanzen diese beeinflussen.
Für die Implementation muss neben dem Margin Parameter b, welcher bestimmt, wann eine korrekte Klassifizierung vorliegt, noch der Parameter δb festgelegt werden, sodass ab einem Abstand von $b + \delta b$ zur Hyperebene eine Instanz als „gut genug“ klassifiziert gilt, um sie zu entfernen. [PT10] weist darauf hin, dass $\delta b > R^2$ gelten sollte, um sich wiederholende Update- und Entfernschritte auf der gleichen Instanz zu vermeiden, und zeigt dies an folgendem Beispiel: Sei $(\mathbf{x}_i, y_i)$ die Instanz mit der größten euklidischen Norm R im Featureraum. Angenommen diese Instanz wird dem Algorithmus vorgelegt und erfüllt gerade noch die Missklassifikationskondition durch

$$y_i(\boldsymbol{w}_t^T \boldsymbol{\Phi}(\boldsymbol{x}_i)) = b \tag{3.10.1}$$

was zu einem Update führt

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + y_i \boldsymbol{\Phi}(\boldsymbol{x}_i) \tag{3.10.2}$$

[1] siehe Dualform in Kapitel 2.1

Wird nun sofort die gleiche Instanz wieder dem Algorithmus vorgelegt, so gilt

$$y_i(\boldsymbol{w}_{t+1}^T \boldsymbol{\Phi}(\boldsymbol{x}_i)) = y_i(\boldsymbol{w}_t^T \boldsymbol{\Phi}(\boldsymbol{x}_i)) + \|\boldsymbol{\Phi}(\boldsymbol{x}_i)\|^2 = b + R^2 \quad (3.10.3)$$

und es kommt zu einem Unlearning Schritt, der die Instanz aus dem Gewichtsvektor entfernt, bis sie dem Algorithmus wieder vorgelegt wird.
Im „Margin Perceptron with Unlearning“ - Algorithmus aus [PT10] ist zusätzlich noch eine Obergrenze für die Anzahl der Updateschritte pro Instanz angebbar. Diese Technik wird hier jedoch ausgelagert und im folgenden Kapitel „ α-Bound“ (3.11.) behandelt.

Algorithmus 3.10 Margin Perceptron with Unlearning

Eingabe: Trainingsdatensatz $\mathbb{Z} = \{(\mathbf{x}_1, y_1),...,(\mathbf{x}_N, y_N)\} \subseteq \mathbb{X} \times \{1, -1\}$), linear separierbar unter Featurefunktion $\Phi: \mathbb{X} \rightarrow \mathbb{K} \subseteq \mathbb{R}^D$

Definiere: $R := \max_k \|x_k\|$, I_k^t : Gewichtsvektor Koeffizient der Instanz k zur Zeit t

Ausgabe: Gewichtsvektor **w**, welcher Hyperebene mit Mindestabständen aufspannt

Initialisierung: $\mathbf{w}_0 := \mathbf{0}$, $t := 0$, $I_k^0 = 0 \; \forall k$

1: **repeat**
2: **for** i = 1 to N **do**
3: **if** $y_i(\mathbf{w}^T \Phi(\mathbf{x}_i)) \leq b$ **then**
4: $\mathbf{w}_{t+1} := \mathbf{w}_t + y_i \Phi(\mathbf{x}_i)$
5: $I_i^t := I_i^t + 1$
6: $t := t + 1$
7: **else if if** $y_i(\mathbf{w}^T \Phi(\mathbf{x}_i)) \geq b + \delta b$ **and** $I_i^t > 0$
8: $\mathbf{w}_{t+1} := \mathbf{w}_t - y_i \Phi(\mathbf{x}_i)$
9: $I_i^t := I_i^t - 1$
10: $t := t + 1$
11: **end if**
12: **end for**
13: **until** kein update in for-Schleife
14: **return** $\mathbf{w}_t$

3.11. α-Bound

In verrauschten Datensätzen kann es vorkommen, dass einzelne Trainingsinstanzen, die der falschen Klasse zugeordnet wurden, zu häufigen Updates des Gewichtsvektors führen: Eine ansonsten korrekt liegende Hyperebene klassifiziert diese Instanzen immer wieder falsch. Erschwerend kommt hinzu, dass die dadurch erzwungenen Updates die Qualität der Hyperebene in der Regel verschlechtern. Um diesen Einfluss zu mindern legt der sogenannte „α -Bound“ aus [KW07] eine Obergrenze für die Anzahl der Updates fest, welche pro Instanz durchgeführt werden dürfen. Ist diese Grenze überschritten, wird die jeweilige Instanz während des Algorithmus ignoriert, indem sie immer als korrekt klassifiziert angesehen wird. Bei ausreichend langer Laufzeit soll dies dazu führen, dass die Updates durch verrauschte Instanzen an ihre Grenzen stoßen, bis schließlich die Updates durch korrekte Instanzen den Gewichtsvektor dominieren.

3.12. λ-Trick

Eine weitere Möglichkeit, den Einfluss verrauschter Trainingsinstanzen zu verkleinern, bietet der sogenannte „λ-Trick“ aus [KW07]. Hierbei wird für einen Datensatz von N Instanzen die Anzahl der Dimensionen des Featurevektors von ursprünglich D auf D + N erhöht. Die dadurch entstehende Featurefunktion Φ^{λ} stimmt auf den ersten D Dimensionen mit der alten Featurefunktion Φ überein. Für die weiteren N Dimensionen gilt:

$$\begin{aligned} \boldsymbol{\Phi}^{\lambda}_{D+i}(\boldsymbol{x}_n) &= 0 & \forall\, i \neq n \\ \boldsymbol{\Phi}^{\lambda}_{D+i}(\boldsymbol{x}_n) &= \sqrt{\lambda} & i = n \end{aligned} \tag{3.12.1}$$

Betrachtet ein Perzeptron-Algorithmus eine Instanz $(\mathbf{x}_i, y_i)$zum ersten mal, so unterscheiden sich die Ergebnisse der Verwendung von alter und neuer Featurefunktion noch nicht:

$$y_i(\boldsymbol{w}^T \boldsymbol{\Phi}^{\lambda}(\boldsymbol{x}_i)) = y_i(\boldsymbol{w}^T \boldsymbol{\Phi}(\boldsymbol{x}_i)) \tag{3.12.2}$$

Durch jedes durch diese Instanz ausgelöste Update wächst jedoch $\mathbf{w}_{D+i}$ um $\sqrt{\lambda}$ [1], sodass schon bei der nächsten Betrachtung dieser Instanz gilt

$$y_i(\boldsymbol{w}^T \boldsymbol{\Phi}^{\lambda}(\boldsymbol{x}_i)) = y_i(\boldsymbol{w}^T \boldsymbol{\Phi}(\boldsymbol{x}_i) + \lambda) \tag{3.12.3}$$

wodurch die Forderung des Einhaltens der Margin für diese Instanz gelockert wird. Dies führt dazu, dass die Missklassifikationskondition für eine bestimmte Instanz weniger streng wird, je öfter diese Instanz bereits Teil des Gewichtsvektors wurde.

Neben dem geringeren Einfluss verrauschter Daten erlangt ein Datensatz durch diese künstlich hinzugefügten zusätzlichen Dimensionen lineare Separierbarkeit, was Voraussetzung einiger hier behandelter Algorithmen und Techniken ist.

[1] Es wird ein konstantes Update mit Lernrate 1 angenommen

4. Experimentaufbau und Hyperparametersuche

Zur Evaluation der verschiedenen Perzeptron-Algorithmen, Techniken und ihrer jeweiligen Parameter wurden ein Datengenerator und ein Algorithmusgenerator erstellt, welche in einem Experimentierframework eingebundene Experimente der generierten Algorithmen auf den generierten Daten ermöglicht. Diese Experimente fanden in WEKA [HFH+09] statt. Die dabei verwendeten Module werden in den folgenden Unterkapiteln erläutert. Schließlich wird noch der Weg zum Finden von Modellen erklärt, die mithilfe der gesammelten experimentellen Datenreihen auf Basis der Datensatzparameter möglichst gute Hyperparameter vorschlagen sollen.

4.1. Datengenerator

Um Daten zu erstellen, auf denen die Algorithmen getestet werden können, wurde ein Datengenerator entwickelt, welcher anhand einiger Parameter künstliche Datensätze erstellt. Es handelt sich dabei um eine Abwandlung des in WEKA bereits implementierten Clustergenerator BIRCHcluster [ZRL96]. Dabei war es wichtig, möglichst viele die Qualität und Performanz der Algorithmen beeinflussende Aspekte in ausreichender Bandbreite konstruieren zu können. Der Generator erstellt dafür jeweils einen Datensatz bestehend aus zwei Clustern, die für die positive bzw. negative Klasse stehen. Tabelle 4.1.1 zeigt die dabei verwendeten Parameter mit den Ober- und Untergrenzen ihrer Wertebereiche, aus denen die entsprechende Werte gemäß einer Gleichverteilung unabhängig voneinander gezogen werden können.

Parameter	Grenzen
M, N: Anzahl Instanzen (pro Cluster)	$2 \leq M, N \leq 1000$
S, T: Clusterradien (Varianz)	$0.1 \leq S,T \leq 1$
P: Klassenrauschen	$0 \leq P \leq 0.25$
A: Attributanzahl	$1 \leq A \leq 30$

Tabelle 4.1.1: Parameter und Parametergrenzen des Datengenerators

In späteren Experimenten wurden die Instanzenanzahlen als Verhältnis des größeren zum kleineren Cluster und die Radien als Summe der Radien beider Cluster ausgedrückt, um die Anzahl der Parameter zu verringern und einen Vergleich mit neuen Datensätzen zu vereinfachen. Es folgen Erläuterungen zu den gewählten Ober- und Untergrenzen der Parameter.

Instanzenanzahl

Eine Mindestanzahl von zwei Instanzen pro Cluster ist nötig, um im Fall eines Aufspalten des Datensatzes in Trainings- und Testset mindestens eine Instanz pro Klasse zu garantieren. Die Obergrenze von 1000 wurde gewählt, um auch bei sehr rechenaufwändigen Algorithmen noch eine zeitlich angemessene Rechenzeit zu gewährleisten.

Clusterradien

Die Parameter der Clusterradien bestimmen hauptsächlich die Separierbarkeit der beiden Klassen. Dabei wird pro Cluster für jedes Attribut ein Zentrum zufällig aus einer Gleichverteilung im Bereich

[0,2] gewählt und die Instanzen mit diesem Zentrum sowie Varianz $S^2/2$ bzw. $T^2/2$ generiert. Dies führt dazu, dass die Wahrscheinlichkeit der Existenz eines Attributes, mithilfe dessen die beiden Klassen linear separierbar sind, durch Verringern der Varianzen gesteigert wird. Abbildung 4.1.1 zeigt zur Verdeutlichung drei Datensätze mit jeweils vier Attributen und Radien von 0,1, 0,5 sowie 0,9.

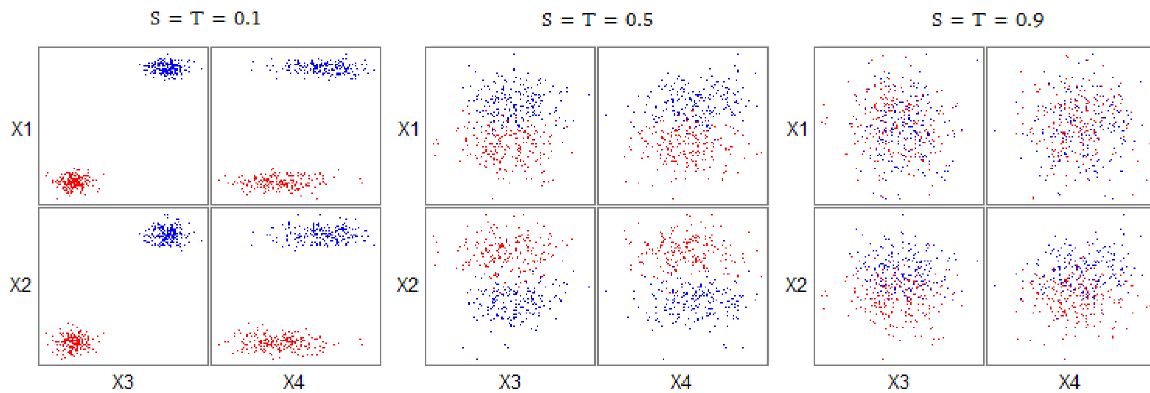

Abbildung 4.1.1: Drei Datensätze mit jeweils vier numerischen Attributen (X1 bis X4) und Radien von 0.1, 0.5 und 0.9. Je höher die Radien, desto unwahrscheinlicher die Existenz eines Attributes, durch das sich beide Klassen linear separieren lassen

Klassenrauschen

Um verrauschte Datensätze zu emulieren kann durch den Parameter P ein Klassenrauschen zwischen 0% und 25% eingestellt werden. Dadurch erhält dieser Anteil an Instanzen den Klassenattributswert der jeweils anderen Klasse.

Attribute

Parameter A steuert die Gesamtanzahl der Attribute, zu denen noch das Klassenattribut hinzukommt.

4.2. Algorithmusgenerator

Ähnlich wie die Daten werden auch die darauf zu testenden Algorithmen und ihre Hyperparameter durch einen Generator erstellt, welcher Zufallswerte zwischen definierten Ober- und Untergrenzen gemäß einer Gleichverteilung wählt. Dabei werden nicht nur die behandelten Algorithmen in ihrer Vollständigkeit betrachtet, sondern auch alle möglichen Kombination ihrer Margin- und Updatemodule. Tabelle 4.2.1 zeigt eine Auflistung der möglichen Komponenten eines Durchlaufs des Algorithmusgenerators und der Ober- sowie Untergrenzen der jeweils zugehörigen Parameter.

Schritt	*Parameter*	*Grenzen*
Iterationen	I: Anzahl Iterationen	$I \leq I \leq 1000$
Reduzierte Epochen	N: Anzahl reduzierter Epochen	$0 \leq N \leq \log_2$(#Instanzen)
α-Bound	α: Maximale Anzahl der Updateschritte pro Instanz	$1 \leq \alpha \leq I * (N + 1)$
λ-Trick	λ: Lambda Wert	$0 < \lambda \leq$ minNorm $0 < \lambda \leq 1$ falls minNorm = 0
Vorhersage Variante	v	$v \in \{0,1,2\}$ 0 := klassisch, 1 := Longest Survivor, 2 := Voted
Margin	**Dynamic** ϵ: Genauigkeit **MICRA** ϵ: Margin Verfallsrate β: Startmargin **Shrinking** b: Startmargin e: Exponent für Variable Shrinking c: Konstanter Shrinkingfaktor v: Variante **Uneven** n/p: Wert für negative / positive Margin scale: Margin skaliert auf Trainingsdaten	 $0 \leq \epsilon < 1$ $0 < \varepsilon \leq 1$ $0 \leq \beta \leq$ maxNorm $0 < b \leq$ maxNorm $0 \leq e \leq 1$ $0.99 \leq c < 1$ $v \in \{0,1\}$, 0 := Konstant, 1 := Variabel $0 \leq n/p \leq$ maxNorm scale $\in$ {true, false}
Update	**Constant** η: Lernrate t: Unlearning Threshold u: Benutze Unlearning **MICRA** η: Startlernrate β: Startmargin δ: Lernrate abhängig von β ζ: Verfallsrate der Lernrate **PA** C: Aggressivität V: Variante **Pegasos** λ: Projektionswert **Shrinking** e: Exponent für variables Shrinking η: Startlernrate c: Konstanter Shrinkingfaktor v: Variante	 $0 < \eta <$ maxNorm - minNorm $\text{maxNorm}^2 < t \leq \text{maxNorm}^5$ $u \in$ {true, false} $0 < \eta \leq 1$ Aus Marginschritt übernehmen, min(n,p) bei Uneven Margin $0 \leq \delta < 1$ $0 < \zeta \leq 1 - \varepsilon * \delta$ ($\varepsilon = 1$ falls nicht MICRA Margin benutzt) $10^{-3} \leq C < 10^{3}$ (oder C = infinity für normales PA) $V \in \{0,1\}$, 0 := PA-I, 1:= PA-II 10 ^ (-10) < λ < 10 ^ (-1) $0 \leq e \leq 1$ $0 < \eta <$ maxNorm - minNorm $0.99 \leq c < 1$ $v \in \{0,1\}$, 0 := Konstant, 1 := Variabel
Bias	N: Bias Wert	$0 < N \leq$ maxNorm
Budgetvariante	**JustRemove** B: Maximale Anzahl erlaubter SV H: Heuristik zur Auswahl des zu entfernenden SV **LeastSquaresProjection** B: Maximale Anzahl erlaubter SV H: Heuristik zur Auswahl des zu entfernenden SV Q: Maximale Anzahl SV um den zu entfernenden zu ersetzen R: Ridge Regression Regularisierungsfaktor (R)	 $1 \leq B \leq$ #Instanzen $H \in \{0,1,2,3,4,5,6\}$, 0 := Kleinster Einfluss, 1 := Kleinster Koeffizient, 2 := Ältester, 3: Zufall, 4 := Kleinste Norm, 5 := größter Abstand zur Trennebe, 6 := Neuester wie bei „JustRemove“ wie bei „JustRemove“ $1 \leq Q \leq \log_2(B)$ $0 < R < 1$

Tabelle 4.2.1: Parameter und Parametergrenzen des Algorithmusgenerators

Auch hier sollen die gewählten Grenzen kurz erläutert werden.

Iterationen

Die Obergrenze der Anzahl der maximalen Iterationen ist inspiriert von Noffikovs Theorem zum ursprünglichen Perzeptron-Algorithmus [Nov62], welches besagt, dass die maximale Anzahl der Updateschritte T durch die maximale Norm R und die maximal erreichbare Margin γ nach oben begrenzt ist:

$$T \geq \left(\frac{R}{\gamma}\right)^2 \qquad (4.2.1)$$

Da die generierten Datensätze aufgrund von Klassenrauschen und geringen Clusterabständen bei gleichzeitig relativ großen Radien in der Regel nicht linear separierbar sind, wird die maximal erreichbare Margin explizit durch den angegebenen Wert des λ-Tricks bestimmt. Wie die erreichbare Margin aus λ abgeleitet werden kann zeigt Abbildung 4.2.1, welche zwei Instanzen i und j auf den durch den λ-Trick für sie hinzugefügten Dimensionen D + i und D + j abbildet. Durch Anwendung des Satzes von Pythagoras lässt sich

$$\gamma = \sqrt{\frac{\lambda}{2}} \qquad (4.2.2)$$

zeigen.

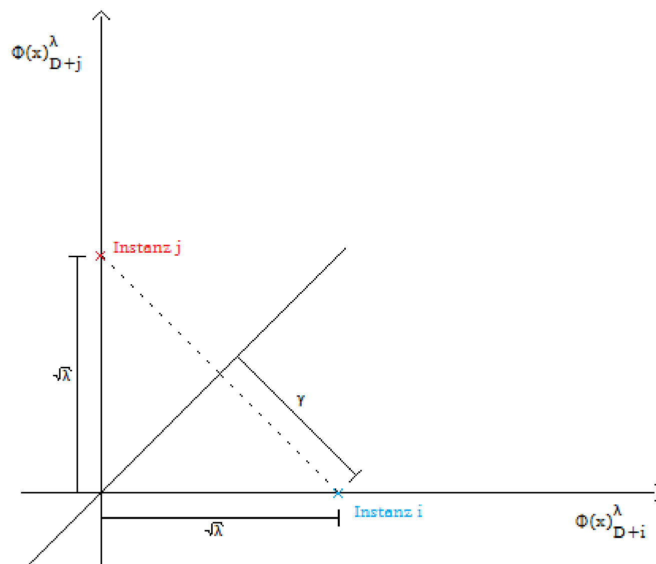

Abbildung 4.2.1: Instanz i und Instanz j, geplottet auf den durch den λ-Trick hinzugefügten Dimensionen D + i und D + j

Mit der aus Kombination von (4.2.1) und (4.2.2) entstandenen Obergrenze

$$T \geq 2\frac{R^2}{\lambda} \tag{4.2.3}$$

wurden daraufhin einige Testdurchläufe durchgeführt, bei denen zu beobachten war, dass die Werte dieser Obergrenze bis auf sehr wenige dafür aber relativ rechenaufwändige Fälle unterhalb von $I = 1000$ lagen, so dass dieser Wert übernommen wurde. Hierbei ist jedoch anzumerken, dass Novikoffs Theorem nur auf den ursprünglichen Perzeptron-Algorithmus anzuwenden ist, welcher hier einem Constant Update mit Lernrate $\eta = 1$ sowie Uneven Margin mit $n = p = 0$ ohne weitere in dieser Arbeit vorgestellte Techniken entspricht. Die Verwendung dieser Obergrenze für alle anderen Kombinationen ist als heuristische Wahl anzusehen.

Reduzierte Epochen
Die Anzahl der reduzierten Epochen wurde aus Berechenbarkeitsgründen stark nach oben durch $\log_2$(Instanzenanzahl) begrenzt. Für jede reduzierte Epoche kann sich im schlechtesten Fall die Zahl der Algorithmendurchläufe während eines Experiments um die Anzahl der angegebenen Iterationen erhöhen, wobei in jedem Durchlauf im schlechtesten Fall alle Instanzen zu Modellupdates führen. Da mit steigender Instanzenzahl auch die erwartete Anzahl an falsch klassifizierten und damit in den reduzierten Epochen behandelten Instanzen steigt, skaliert diese Obergrenze mit der Instanzenanzahl. Mit den angegebenen Grenzen für Instanzenanzahl und einem 66% Trainingssplit sind maximal 10 reduzierte Epochen sowie 13 Millionen Modellupdates möglich.

α-Bound
Die Obergrenze des α-Bound von Iterationenanzahl * (Anzahl Reduzierte Epochen + 1) erfolgt aus der Beobachtung, dass jede Instanz maximal in jeder Iteration sowie jeder einzelnen darauf folgenden reduzierten Epoche zu einem Update führen kann.

λ-Trick
Der Wert des λ-Tricks wurde auf $\lambda > 0$ beschränkt, um in jedem Fall eine lineare Separierbarkeit der Datensätze zu garantieren, um allen Algorithmuskombinationen die Möglichkeit zu geben, zu einer Lösung zu konvergieren, bevor die maximale Anzahl an Iterationen erreicht ist. Gleichzeitig sollten die Algorithmen sich aber nicht übermäßig auf die durch den λ-Trick künstlich hinzugefügten Dimensionen fokussieren, weshalb als Obergrenze die minimale Norm gewählt wurde.

Margins
Die jeweiligen Startmargins wurden nach oben durch die maximale Norm begrenzt, um sie mit der Größe der Instanzen zu skalieren. Der Shrinkingfaktor des konstanten Shrinkings wurde auf mindestens 0.99 gesetzt, da bei Shrinking Margin bzw. Shrinking Update die Margin bzw. die Lernrate bei jedem Update durch diesen Faktor dividiert wird, was bei einer hohen Anzahl an Updates dazu führt, dass die im Perceptrovement Framework verwendeten Werte vom Datentype double über ihre darstellbaren Grenzen wachsen. Dieses Problem tritt zwar auch bei Shrinkingfaktoren über 0.99 auf, jedoch weitaus seltener als bei niedrigeren Werten. Die weiteren Grenzen aus dem Bereich Margin sind mathematische Notwendigkeiten und aus den zugehörigen Papern übernommen.

Updates
Als Wert für die Lernrate zu Beginn der verschiedenen Algorithmen wurde als Obergrenze die Differenz zwischen maximaler und minimaler Norm gewählt, die als Annäherung der Streuung der Instanzen beider Klassen dienen soll, welche den Raum bestimmt, in dem sich die Hyperebene bewegen sollte. Nur bei MICRA Updates wurde die Startlernrate auf maximal 1 begrenzt, da es hier bei höheren Werten und einer hohen Anzahl an Updates Probleme gab mit nicht mehr vom Datentyp double darstellbaren Werten. Der Aggressivitätsparameter C des Passive-Aggressive Updates wurde ähnlich begrenzt wie in den Experimenten des zugehörigen Papers, die auf Datensätzen einer ähnlichen Größenenordnung stattfanden.
Die Untergrenze des Unlearning Thresholds ergibt sich aus der Beobachtung, dass es bei einem Wert unterhalb R^2 zu einem Kreislauf kommen kann, bei dem eine Instanz abwechselnd gelernt und entfernt wird (siehe 3.10.).
Die Grenzen des MICRA Updates für δ und ζ folgen aus der Eigenschaft des MICRA Algorithmus, für die Werte $0 < \epsilon\, \delta + \zeta < 1$ und für $\beta/R \rightarrow \infty$ zu einer korrekten Lösung zu konvergieren, vorausgesetzt die Daten sind linear separierbar, was mit dem λ-Trick garantiert wird.
Die Grenzen des Projektionsparameters λ von Pegasos sind an die Experimente von [SSS+07] angelehnt. Da dort jedoch auf sehr unterschiedlichen Datensätzen gearbeitet wurde sind die Grenzen in dieser Arbeit erweitert worden.

Bias
Um zu verhindern, dass der Wert des Bias bei einem Update die Veränderung des Gewichtsvektors dominiert, wurde er auf maximal R begrenzt.

Budget
Der Parameter Q der LeastSquaresProjection Budgetmethode wurde nach anfänglichen Experimenten mit theoretisch möglichen Grenzen $1 \leq Q \leq B$ als in vielen Fällen zu rechenintensiv erkannt. Dies liegt an der bei hoher Instanzenanzahl und hohem Wert Q sehr aufwendigen Regression, welche zur Anpassung der Gewichtskoeffizienten durchzuführen ist. Daher wurde die Obergrenze dieses Parameters auf $Q \leq \log_2(B)$ reduziert.

4.3. Experimentierframework

Die Verknüpfung von Daten- und Algorithmusgenerator mit anschließender Ausgabe der Ergebnisse ergibt das Experimentierframework, mithilfe dessen die Algorithmen und ihre Hyperparameter evaluiert wurden. Tabelle 4.3.1 zeigt dabei alle Parameter, welche während eines Durchlaufs des Frameworks einstellbar sind.

Datensatz

M,N,S,T,P,A

Algorithmus

I, N, α, λ, v, N, B, H(,Q,R)

+

Update \ Margin	Dynamic	MICRA	Shrinking	Uneven
Constant	ε, β, η, t, u	ε, β, η, t, u	b, e, c, v, η, t, u	n, p, scale, η, t, u
MICRA	ε, β, η, ζ, β ,δ	ε, β, η, ζ, β ,δ	b, e, c, v, η, ζ, β ,δ	n, p, scale, η, ζ, β ,δ
PA	ε, β, C, V	ε, β, C, V	b, e, c, v, C, V	n, p, scale, C, V
Pegasos	ε, β, λ	ε, β, λ	b, e, c, v, λ	n, p, scale, λ
Shrinking	ε, β, e, η, c, v	ε, β, e, η, c, v	b, e, c, v, e, η, c, v	n, p, scale, e, η, c, v

Tabelle 4.3.1: Zu bestimmende Hyperparameter bei Durchlauf des Experimentierframeworks

Damit ergibt sich pro Durchlauf des Experimentierframeworks je nach Update- und Marginwahl eine Liste aus minimal 17, maximal 24 größtenteils reellwertigen Parameterwerten. Bei der Auswahl dieser Werte wurde eine Zufallsauswahl einem Grid, also der Wahl von Werten mit gleichem Abstand zueinander, vorgezogen. Dies folgt aus der Vermutung, dass der Einfluss der verschiedenen Parameter auf die Performanz des Algorithmus und die Genauigkeit der letztendlich gelernten Hyperebene sehr unterschiedlich ausfällt. Ist dies der Fall, so ist laut [BB12] eine Zufallsauswahl im Vorteil, da bei ihr eine größere Zahl an Werten der relevanten Parameter untersucht wird, wie Grafik 4.3.1 verdeutlicht.

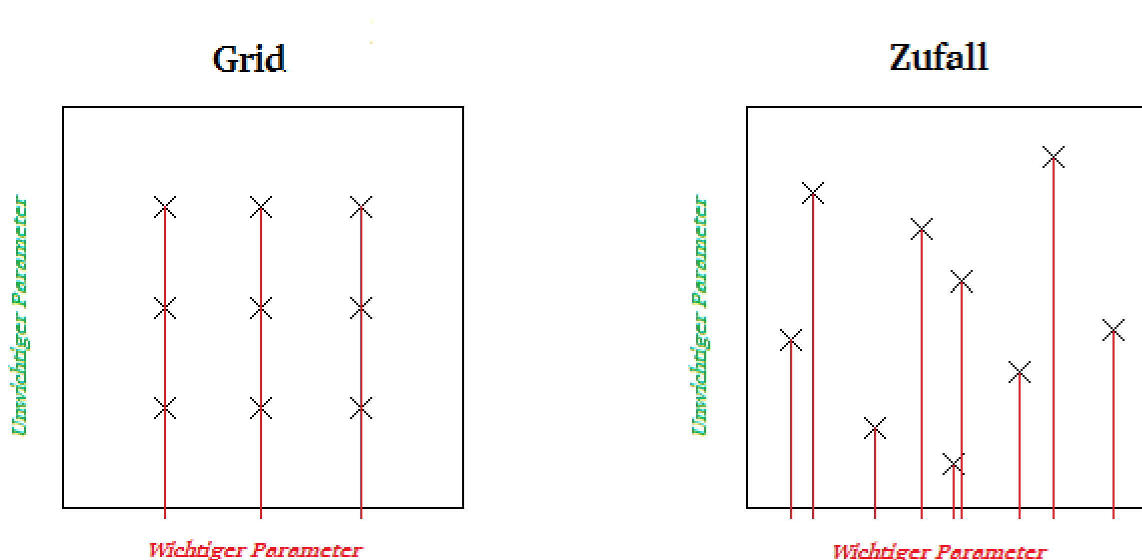

Abbildung 4.3.1: Auswahl von neun Versuchen eines Systems mit einem entscheidenden und einem unwichtigen Parameter, links per Grid, rechts per Zufallsauswahl. Bei der Wahl eines Grids wird der wichtige Parameter nur an drei Stellen betrachtet, bei der Zufallsauswahl dagegen an neun. In Anlehnung an Figure 1 aus [BB12]

Zum Testen der entstandenen Perzeptronvarianten wurden die jeweils generierten Daten in 66% Trainingsdaten, auf denen das Perzeptron gelernt wurde, und 34% Testdaten, auf denen es daraufhin auf seine Genauigkeit hin getestet wurde, aufgeteilt.

4.4. Lernen von Hyperparametern

Zur Wahl passender Hyperparameter, die die Genauigkeit auf einem gegebenen Eingabedatensatz maximieren, wäre es erstrebenswert, eine Funktion der Form $g(\mathbf{x}) = \mathbf{y}$ zu lernen, die aus den Datensatzparametern **x** die zu wählenden Hyperparameter **y** bestimmt, welche die Genauigkeit auf diesem Datensatz maximieren. Mithilfe des vorgestellten Experimentierframeworks ist es aber auf direktem Weg nur möglich, für Paare aus Datensatz- und Hyperparametern die erreichte Genauigkeit a zu bestimmen, also Daten einer Funktion $f(\mathbf{x},\mathbf{y}) = a$ zu erstellen. Aus diesen Daten lässt sich die gewünschte Funktion nicht lernen, da diese Datensatz- und Hyperparameter per Zufall gewählt werden und zu einem breiten Spektrum an Genauigkeiten und nicht nur deren Maximum führen.
Um die gewünschte Funktion abzuleiten, wurden zunächst Modelle der Form $f(\mathbf{x},\mathbf{y}) = a$ gelernt, die es erlauben, bei gegebenen Datensatz- und Hyperparametern effektiver die erreichbare Genauigkeit zu errechnen als sie experimentell zu bestimmen. Zum Lernen dieser Modelle wurden vier Lernalgorithmen verwendet: Lineare Regression, M5-Rules, SVM und kNN. Die genauen Konfigurationen sind Tabelle 5.14.1.1 zu entnehmen.
Mit diesen Modellen wurden daraufhin für zufällig gewählte Eingabeparameterkonstellationen jeweils 1000 zufällig gewählte Hyperparameterkonstellationen evaluiert, und dasjenige Set aus Hyperparametern, welches die Genauigkeit maximiert ($a_{max,}$), extrahiert. Insgesamt wurden mit diesem Verfahren pro Update- Marginvariante 5000 Datenreihen aus Eingabeparametern und die Genauigkeit für diese Eingabeparameter maximierenden Hyperparametern erstellt, die daraufhin als Trainingsdaten der gewünschten Funktionen $g(\mathbf{x}) = \mathbf{y}$ dienten. Abbildung 4.4.1 zeigt den Datenfluss dieses Vorgehens.

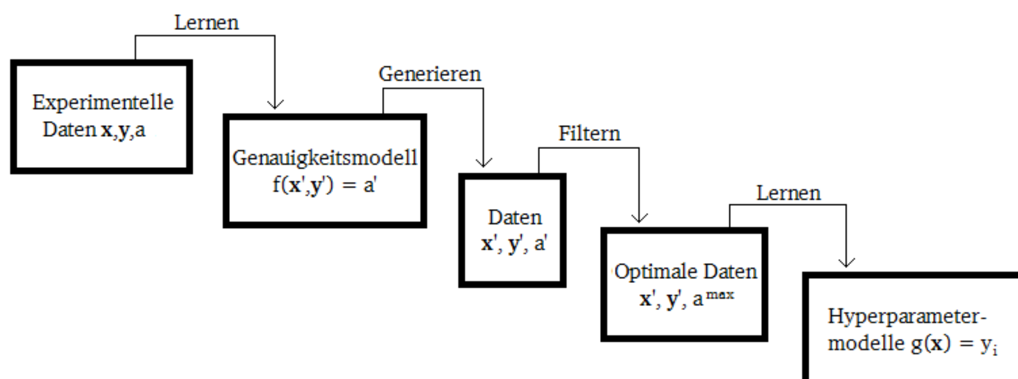

Abbildung 4.4.1: Datenfluss bei Lernen der Hyperparametermodelle $g(\mathbf{x}) = y_i$

Nicht gelernt wurden die Parameter der Vorhersagevariante und des Budgets, da diese nominale Werte beinhalten. Zur deren Wahl sei auf Kapitel 5.10 (Vorhersagevariante) bzw. 5.12 (Budget) verwiesen. Weiterhin ist es mit diesen Modellen nicht möglich, etwaige Abhängigkeiten einzelner Hyperparameter untereinander abzubilden. Da es sich jedoch herausstellte, dass diese simplen Modelle bereits gute Ergebnisse liefern, wurde auf die Untersuchung komplexerer verzichtet.

5. Ergebnisse

Nach 500.000 Durchläufen des Experimentierframeworks wurden die verschiedenen Kombinationen des Perzeptron-Algorithmus und die zugehörigen Parameter auf Basis der dabei erhaltenen Ergebnisse evaluiert. Es folgen nun zunächst Betrachtungen über die allgemeine Performanz der Kombinationen aus Margin- und Updatevarianten. Dabei ist zu beachten, dass hier alle in Tabelle 8.1 beschriebenen Parameterwerte einfließen, sodass die Vermutung nahelag, einzelne Kombinationen mit geschickter Parameterwahl noch entscheidend verbessern zu können.
Darauf folgen Ergebnisse zum Einfluss der einzelnen Parameter auf die Genauigkeit. Dabei werden zunächst die globalen Parameter behandelt, welche in jedem Experimentdurchlauf gewählt wurden, wie beispielsweise die Instanzenanzahl, und schließlich die Hyperparameter der einzelnen Update- und Marginmodule. Die zugehörigen Graphen zeigen in der Regel die durchschnittliche Genauigkeit, mit der die entstandenenen Hyperebenen die Instanzen der Testsets klassifizierten, in Abhängigkeit der jeweiligen Parameterwahl sowie die Standardabweichung der Genauigkeitswerte als symmetrische horizontale Balken. In einigen Graphen wurde allerdings auf die Darstellung der Standardabweichung verzichtet, um die Unterschiede der Durchschnittswerte besser hervorzuheben.
Schließlich werden die Ergebnisse bezüglich des Lernens von Hyperparametermodellen $g(\mathbf{x}) = \mathbf{y}$ vorgestellt.

5.1. Kombinationen

Um die Performanz der verschiedenen Algorithmuskombinationen zu testen wurden die Ergebnisse gemäß ihrer Update- und Marginvarianten gruppiert. Um die gleiche Anzahl aller Kombinationen zu gewährleisten wurden aus den ca. 25000 Experimentdurchläufen pro Kombination die ersten 20000 ausgewählt. Der Bereich der möglichen Genauigkeitsergebnisse wurde in zehn gleich große Bereiche aufgeteilt. Die folgenden Histogramme zeigen dabei gruppiert nach dem Update die Anzahl der Ergebnisse der Update-Margin-Kombinationen, die in einen solchen Bereich fielen. Die gestrichelten Linien markieren den Durchschnittswert. Bei einigen Kombinationen kam es, wie in Kapitel 4.2. beschrieben, zu Genauigkeitsangaben von 0 aufgrund zu hoher Werte. In diesen Fällen wurden in einem zweiten Histogramm diese Experimentierergebnisse herausgefiltert und die Anzahl auf 18000 geschrumpft.

Constant Update

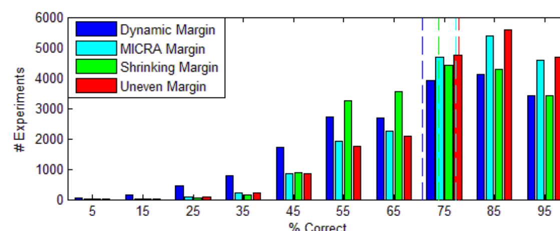

Abbildung 5.1.1: Constant Update und Margin Varianten

MICRA Update

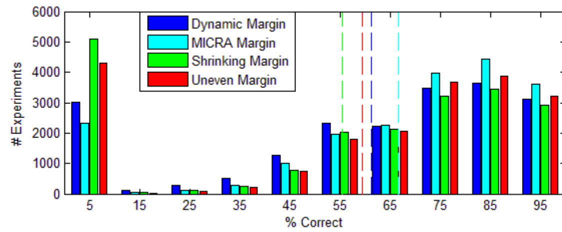

Abbildung 5.1.2: MICRA Update und Margin Varianten

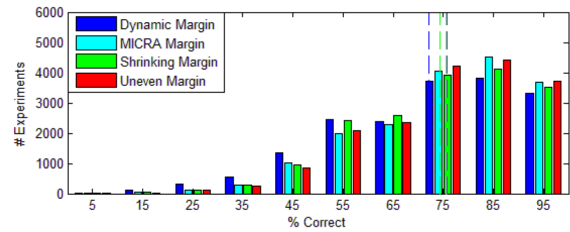

Abbildung 5.1.3: MICRA Update und Margin Varianten gefiltert

Passive-Aggressive Update

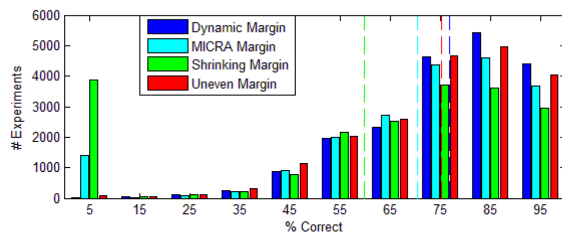

Abbildung 5.1.4: Passive-Aggressive Update und Margin Varianten

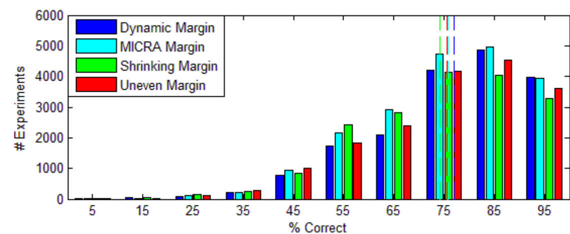

Abbildung 5.1.5: Passive-Aggressive Update und Margin Varianten gefiltert

Pegasos Update

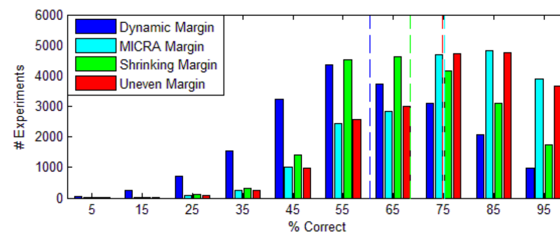

Abbildung 5.1.6: Pegasos Update und Margin Varianten

Shrinking Update

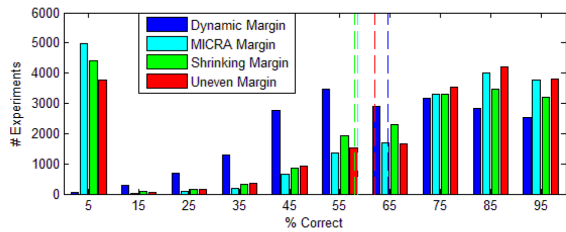

Abbildung 5.1.7: Shrinking Update und Margin Varianten

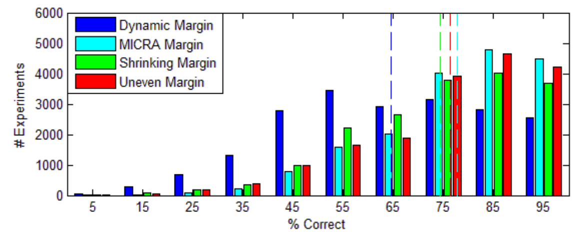

Abbildung 5.1.8: Shrinking Update und Margin Varianten gefiltert

Auffallend ist hierbei die hohe Varianz in den Ergebnissen der Dynamic Margin: Während sie mit dem Passive-Aggressive Update gut kombinierbar zu scheint liegt sie beim Pegasos und Shrinking Update sehr weit hinter ihren alternativen Margin Varianten zurück. Außerdem ist sie die einzige Margin, die beim Shrinking Update den Gewichtsvektor stabil hält und zu keiner Überschreitungen des Maximalwertes des double-Datentyps führt, dafür aber auch die vergleichsweise schlechtesten Ergebnisse liefert. Erstaunlich ist auch, dass die Shrinking Margin bei ihrer zugehörigen Update Variante zwar nur knapp, aber dennoch hinter MICRA- und Uneven Margin liegt, die bei allen Varianten nahezu identische Durchschnittswerte aufzeigen.

5.2. Instanzenanzahl

Bei der Anzahl der Instanzen, die das Trainings- und das Testset ausmachen, zeigt sich, dass die Gesamtanzahl (Abbildung 5.2.1) kaum Einfluss auf die Genauigkeit hat. Allenfalls ist ab etwa 1000 Instanzen ein leichter Abwärtstrend erkennbar, was vermutlich damit zusammenhängt, dass die Obergrenzen einiger der anderen Werte von der Instanzenanzahl abhängen und höhere Werte dort zu einer negativen Beeinflussung der Genauigkeit führen könnten. Es sollte beachtet werden, dass die Extrembereiche nahe 0 und 2000 Instanzen hoher Varianz unterliegen, da hier nur vergleichsweise wenige Ergebnisse vorliegen. Im Gegensatz zur Gesamtanzahl hat die Differenz der Anzahl der Instanzen beider Klassen spürbare Auswirkung auf die Genauigkeit (Abbildung 5.2.2).

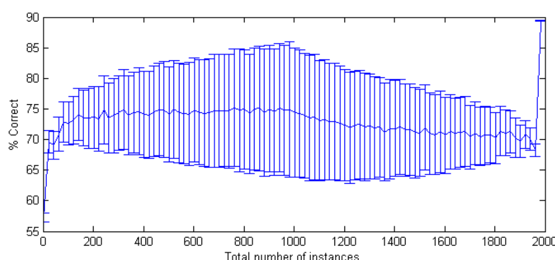

Abbildung 5.2.1: Genauigkeit bei Gesamtanzahl der Instanzen

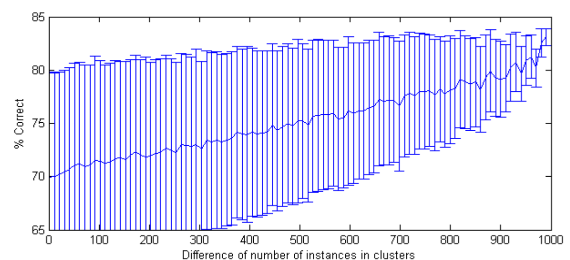

Abbildung 5.2.2: Genauigkeit bei Differenz der Instanzenanzahl beider Cluster

Hierbei ist interessant, ob diese Verbesserung in der Genauigkeit bei steigender Differenz bei allen Kombinationen vorkommt, oder lediglich die Uneven Margin wie in [LZH+02] beschrieben mit entsprechender Parameterwahl den Durchschnitt hebt.

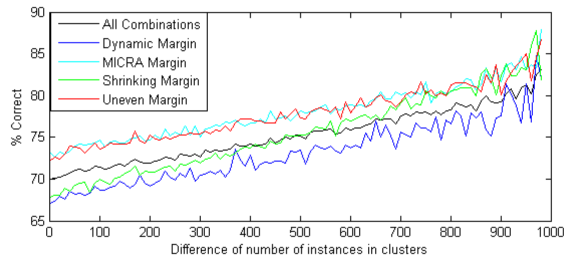

Abbildung 5.2.3: Genauigkeit der Margin Varianten bei Differenz der Instanzenanzahl beider Cluster

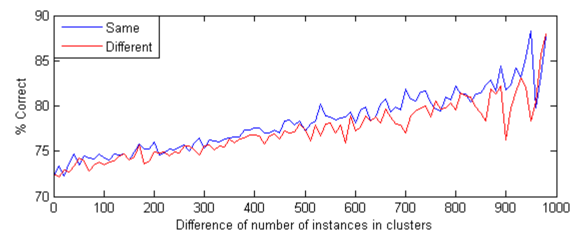

Abbildung 5.2.4: Genauigkeit der Uneven Margin bei Differenz der Instanzenanzahl beider Cluster, in blau zu Instanzenverhältnis passende Parameterordnung, in rot unpassende

Bei genauerer Betrachtung und Vergleich der verschiedenen Margins zeigt sich, dass dies nicht der Fall ist und die Uneven Margin in ihrer Steigung dem Durchschnitt ähnelt (Abbildung 5.2.3). Vermutlich gleicht hier eine überdurchschnittliche Performanz bei passender Parameterwahl eine unterdurchschnittliche bei unpassender Parameterwahl aus. Um dies zu untersuchen, wurden die Daten der Uneven Margin aufgeteilt in Fälle, in denen die Marginparameter n und p einer Ordinalskala entsprechend die gleiche Reihenfolge wie die Verteilung von positiven und negativen Instanzen aufweisen, also $n > p$ nur dann auftrat, wenn mehr negative als positive Instanzen vorhanden sind, und umgekehrt (Abbildung 5.2.4). Hier zeigt sich eine leichte Verbesserung der Genauigkeit durch die Wahl passender Parameter.
Auffallend ist weiterhin das überdurchschnittliche Ausnutzen stark unterschiedlicher Instanzenanzahlen durch die Shrinking Margin.

5.3. Clusterradien

Da die Radien der beiden Cluster eines Datensatzes die Wahrscheinlichkeit beschreiben, dass deren Werte sich für ein beliebiges Attribut überschneiden, war zu erwarten, dass mit steigenden Radien die Genauigkeit sinkt. Die folgenden Graphen zeigen die erreichte Genauigkeit als Funktion der Summe der beiden Radien, wie in 5.1 gruppiert nach dem Update. Zum Vergleich ist in jedem Graphen auch noch die Kurve des Durchschnitts aller Kombinationen in schwarz zu sehen.

Constant Update

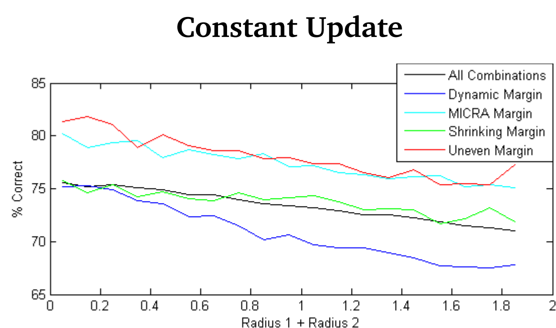

Abbildung 5.3.1: Genauigkeit bei Summe der Radien unter Constant Update

MICRA Update

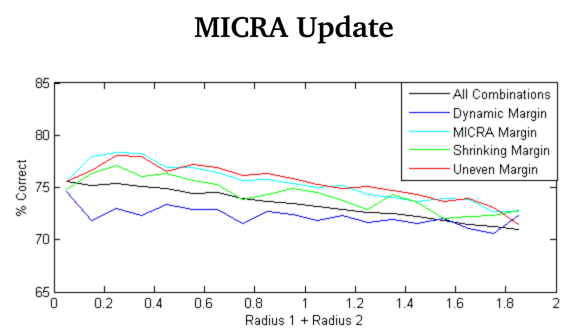

Abbildung 5.3.2: Genauigkeit bei Summe der Radien unter MICRA Update

Passive-Aggressive Update

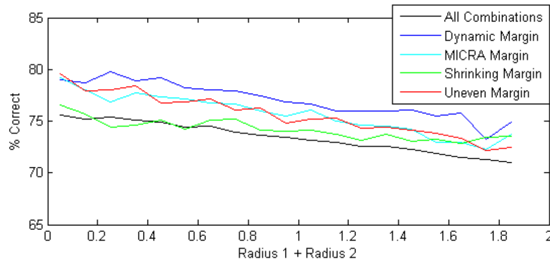

Abbildung 5.3.3: Genauigkeit bei Summe der Radien unter Passive-Aggressive Update

Pegasos Update

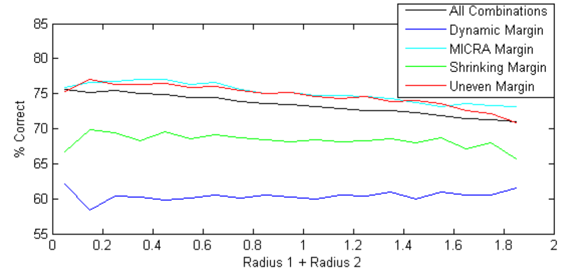

Abbildung 5.3.4: Genauigkeit bei Summe der Radien unter Pegasos Update

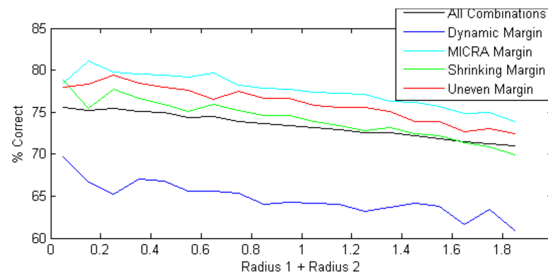

Abbildung 5.3.5: Genauigkeit bei Summe der Radien unter Shrinking Update

Auch hier zeigt sich die Dynamic Margin wieder auffällig. Während sie beim MICRA- und Pegasos Update beinahe unabhängig von den Radien ist, wird sie beim Constant Update überdurchschnittlich von ihnen beeinflusst.

5.4. Attributanzahl

Durch zusätzliche Attribute lassen sich die beiden Klassen potenziell besser separieren. Dies zeigen die folgenden Graphen, wieder gruppiert nach Update-Variante und mit Durchschnitt aller Kombinationen in schwarz.

Constant Update

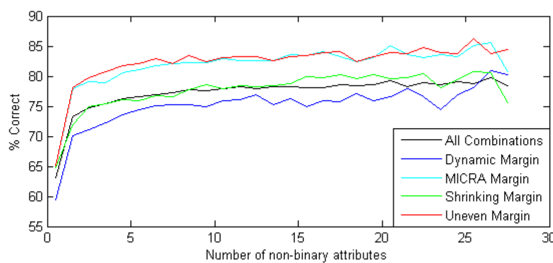

Abbildung 5.4.1: Genauigkeit bei Anzahl Attribute unter Constant Update

MICRA Update

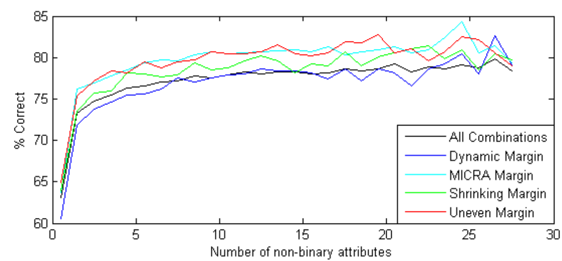

Abbildung 5.4.2: Genauigkeit bei Anzahl Attribute unter MICRA Update

Passive-Aggressive Update

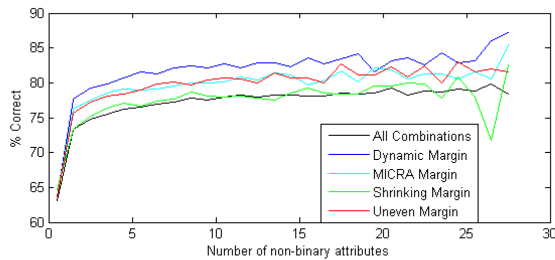

Abbildung 5.4.3: Genauigkeit bei Anzahl Attribute unter Passive-Aggressive Update

Pegasos Update

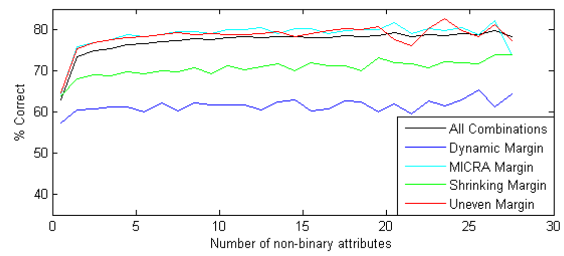

Abbildung 5.4.4: Genauigkeit bei Anzahl Attribute unter Pegasos Update

Shrinking Update

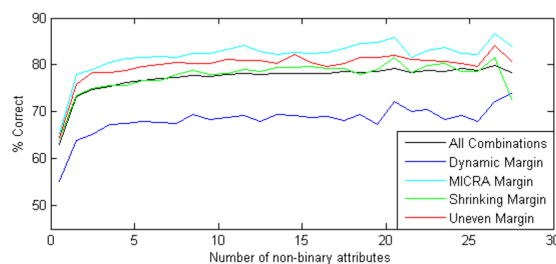

Abbildung 5.4.5: Genauigkeit bei Anzahl Attribute unter Shrinking Update

Einer rasanten Verbesserungen der Genauigkeit durch die ersten zwei bis drei Attribute folgt ein abgeschwächter, aber dennoch deutlicher Anstieg durch weitere Attribute in beinahe allen Kombinationen. Die bereits in Abschnitt 5.1-5.3 festgestellte Beobachtung einer relativen Unvereinbarkeit von Dynamic- und Shrinking Margin mit dem Pegasos Update zeigt sich auch hier wieder: Diese Kombinationen gewinnen durch weitere Attribute kaum an Genauigkeit hinzu.

5.5. Rauschen

Durch Klassenrauschen erhalten Instanzen einer Klasse den Klassenattributswert der anderen. Dies führt zum einen zu kontraproduktiven Updates des Gewichtsvektors während eines Algorithmus Durchlaufs, zum anderen werden diese Instanzen im Testset durch eine korrekte Hyperebene falsch klassifiziert. In jedem Fall sinkt die Genauigkeit bei steigendem Rauschen.

Constant Update

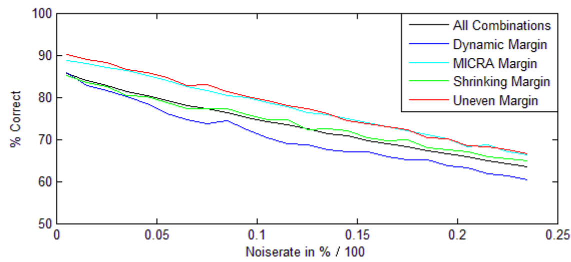

Abbildung 5.5.1: Genauigkeit bei Rauschen unter Constant Update

MICRA Update

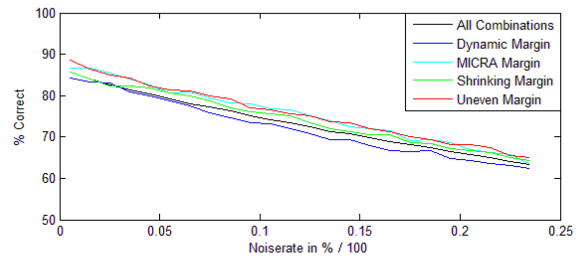

Abbildung 5.5.2: Genauigkeit bei Rauschen unter MICRA Update

Passive-Aggressive Update

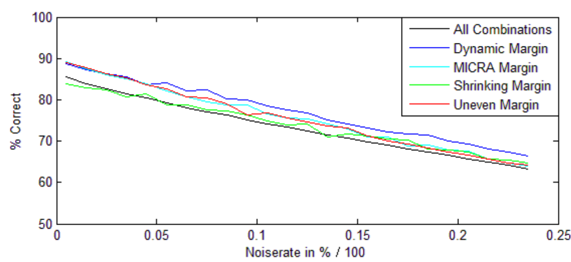

Abbildung 5.5.3: Genauigkeit bei Rauschen unter Passive-Aggressive Update

Pegasos Update

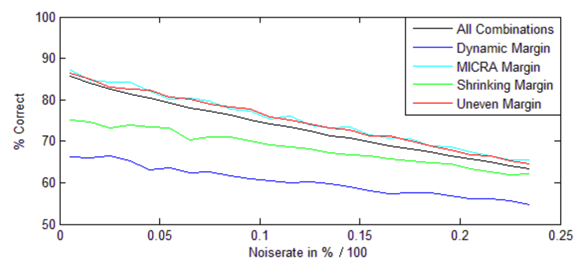

Abbildung 5.5.4: Genauigkeit bei Rauschen unter Pegasos Update

Shrinking Update

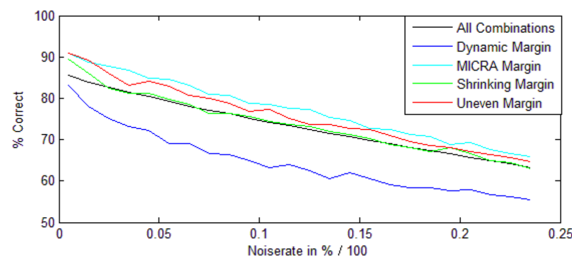

Abbildung 5.5.5: Genauigkeit bei Rauschen unter Shrinking Update

5.6. Iterationen

Beschränkt man die Anzahl der Iterationen eines Perzeptron Algorithmus, sollte seine Genauigkeit sinken. Hier verhielten sich die Experimente gemäß dieser Erwartung, auch wenn ab ca. 150 Iterationen die Verbesserung der Genauigkeit durch Erhöhen der Iterationsanzahl merklich abnimmt.

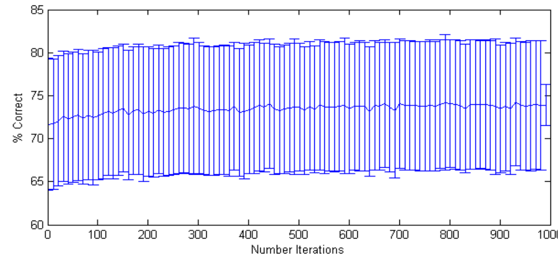

Abbildung 5.6.1: Genauigkeit bei maximal erlaubter Iterationen

5.7. Reduzierte Epochen

Bei der Betrachtung des Einflusses der reduzierten Epochen auf die Genauigkeit zeigt sich, dass diese auf den vorliegenden Daten zu einer leichten Verringerung der Genauigkeit führen (Abbildung 5.7.1).

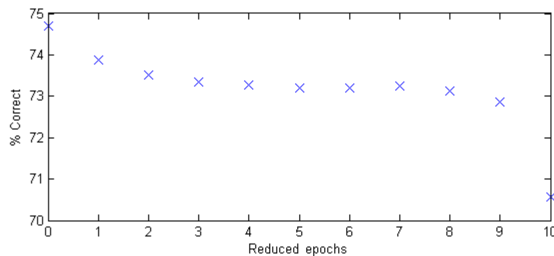

Abbildung 5.7.1: Genauigkeit bei Anzahl reduzierter Epochen

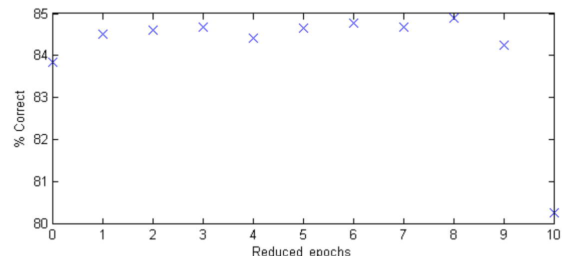

Abbildung 5.7.2: Genauigkeit bei Anzahl reduzierten Epochen, Daten gefiltert auf unter 5% Rauschen und Radien kleiner als 0,5

Hier liegt die Vermutung nahe, dass bei den mit dem hier verwendeten Datengenerator erstellten Datensets, die durch Rauschen und geringe Radien viele kontraproduktive Instanzen beinhalten, diese Instanzen die reduzierten Epochen dominieren und zu schlechten Updates führen. Um dies zu überprüfen wurden die Daten gefiltert, sodass nur noch diejenigen Experimentierreihen mit Rauschen von unter fünf Prozent und Radien unterhalb 0,5 übrigblieben (Abbildung 5.7.2).
Dabei steigt zwar die Genauigkeit bei den ersten drei reduzierten Epochen, bleibt dann allerdings relativ konstant und fällt schließlich wieder ab. Da die Anzahl der reduzierten Epochen abhängig ist von der Gesamtanzahl der Instanzen (hohe Anzahl reduzierter Epochen sind nur möglich bei hoher Instanzenanzahl), hat der in 5.2 etablierte Abwärtstrend der Genauigkeit bei steigender Instanzenanzahl auch hier Einfluss und dient als mögliche Erklärung des Abfalls gegen Ende.

5.8. α-Bound

Ein Begrenzen der Updates auf einzelnen Instanzen durch den α-Bound führt zu einer Verbesserung der Genauigkeit. Dabei werden die besten Werte erreicht bei einem möglichst kleinem α-Bound.

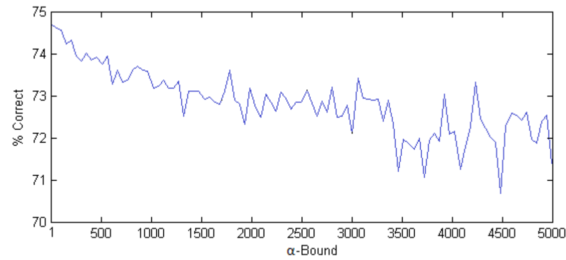

Abbildung 5.8.1: Genauigkeit bei α-Bound

5.9. λ-Trick

Beim λ-Trick zeigt sich, dass für λ ein Wert von mindestens ein Zehntel der maximalen Norm gewählt werden sollte. Für die hohen Werte liegen relativ wenige Beispiel vor, sodass der Graph an diesen Stellen aufgrund der hohen Varianz nicht interpretiert werden sollte.

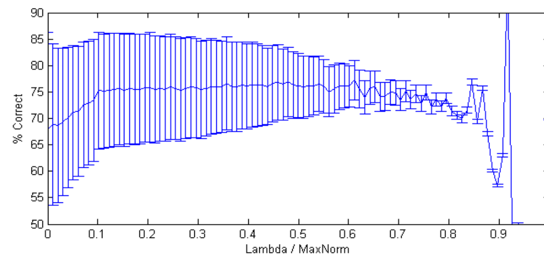

Abbildung 5.9.1: Genauigkeit bei λ-Trick, skaliert mit maximaler Norm

5.10. Vorhersage

Bei den Vorhersagevarianten zeigt sich eine Verbesserung der Genauigkeit durch Wechsel von der klassischen Variante zu der Variante „Longest Survivor". Weitere Verbesserung erreicht man mit der „Voted" Variante, erkauft sie sich jedoch mit stark steigendem Speicher- und Rechenaufwand, da dafür alle zwischenzeitlich erreichten Gewichtsvektoren gespeichert und ausgewertet werden müssen.

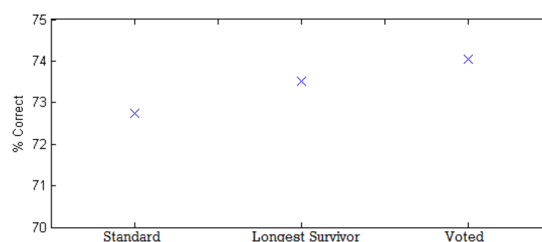

Abbildung 5.10.1: Durchschnittliche Genauigkeit bei Wahl der Vorhersagemethoden

5.11. Bias

Als Bias-Wert sollte ein eher kleiner Wert gewählt werden, allerdings ist die im zugehörigen Graphen zu erkennende Tendenz relativ klein und die Varianz hoch. Hier wäre eine genauere Untersuchung mit mehr Daten als dieser Arbeit vorliegen wünschenswert.

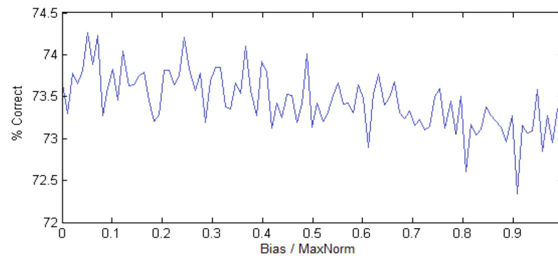

Abbildung 5.11.1: Genauigkeit bei Bias, skaliert mit maximaler Norm

5.12. Budget

Im Gegensatz zum Begrenzen der Updates auf einzelnen Instanzen durch den α-Bound (5.8) zeigt das Einhalten eines Budgets der maximalen Supportvektorenanzahl keine positiven Auswirkungen für die Genauigkeit (Abbildung 5.12.1). Im Gegenteil, bei kleinerem Budget sinkt diese sogar. Verglichen mit dem simplen Entfernen der das Budget überschreitenden Supportvektoren hebt das Projizieren auf die verbleibenden den durchschnittlichen Genauigkeitswert von 72,75% auf 74,14%.
Bei den Heuristiken scheint die Entfernung der Supportvektoren mit kleinstem Koeffizient am erfolgreichsten zu verlaufen, dicht gefolgt von der Wahl der Supportvektoren mit kleinstem Einfluss und der Wahl des neuesten (Abbildung 5.12.2). Noch schlechter als eine Zufallsauswahl zeigt sich die Wahl des ältesten und desjenigen Supportvektoren, der den größten Abstand zu Hyperebene einhält.
Durch Erhöhen der Anzahl der Supportvektoren, die einen entfernten ersetzen, ist zunächst eine Verbesserung der Genauigkeit erreichbar, die aber ähnlich zur Anzahl der Instanzen und der reduzierten Epochen gegen Ende abfällt (Abbildung 5.12.3).
Die Wahl des Regularisierungsfaktors hat keinen erkennbaren Einfluss auf die Genauigkeit (Abbildung 5.12.4).

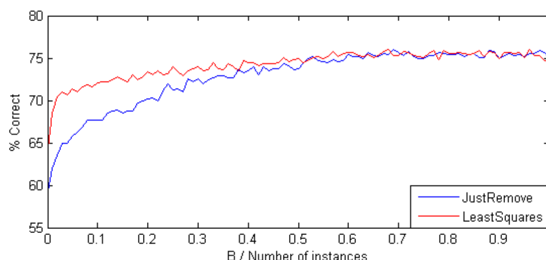

Abbildung 5.12.1: Genauigkeit bei Budgetlimit, skaliert mit Instanzenanzahl

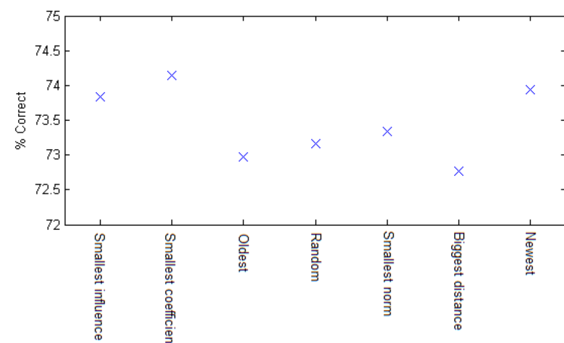

Abbildung 5.12.2: Durchschnittliche Genauigkeit der Budget Heuristiken

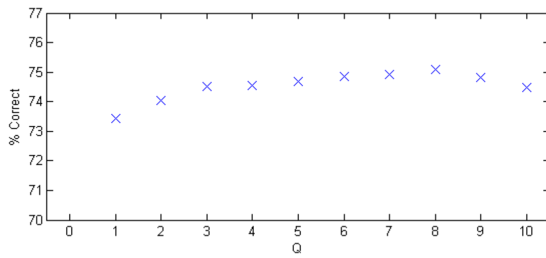

Abbildung 5.12.3: Genauigkeit bei Anzahl der Supportvektoren, die einen entfernten ersetzen

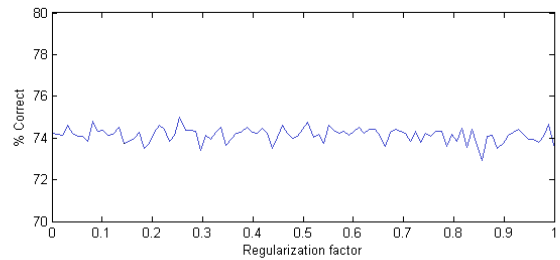

Abbildung 5.12.4: Genauigkeit bei Ridge Regression Regularisationsfaktor

5.13. Margin- und Updateparameter

5.13.1. Dynamic Margin

Bei der Wahl des Parameters ϵ verhält sich die Genauigkeit wie erwartet: Je mehr Nähe zur maximal möglichen Margin man einfordert, desto höher steigt die Genauigkeit. Die Kosten liegen bei der höheren Anzahl an Updates, die bei hohen Werten von ϵ benötigt werden.

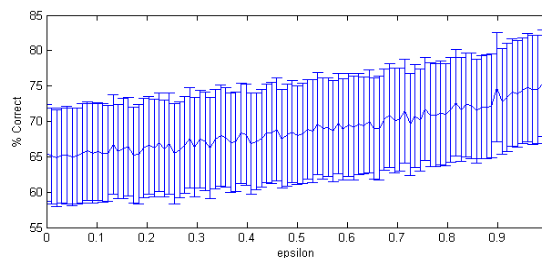

Abbildung 5.13.1.1: Genauigkeit bei Genauigkeitsforderung ϵ unter Dynamic Margin

5.13.2. MICRA Margin

Die Wahl der Margin zu Beginn hat keinen Einfluss auf die Genauigkeit (Abbildung 5.13.2.2), wohl aber die Wahl des Verfallsparameters ϵ (Abbildung 5.13.2.1). Ein Wert von ca. 0,2 scheint hier das Optimum zu sein.

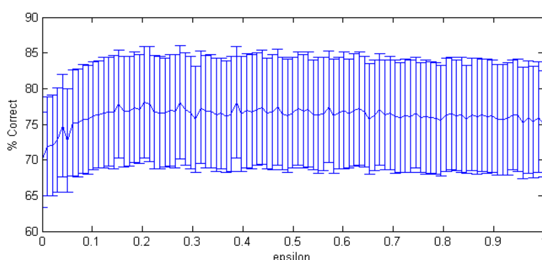

Abbildung 5.13.2.1: Genauigkeit bei Verfallsparameter ϵ unter MICRA Margin

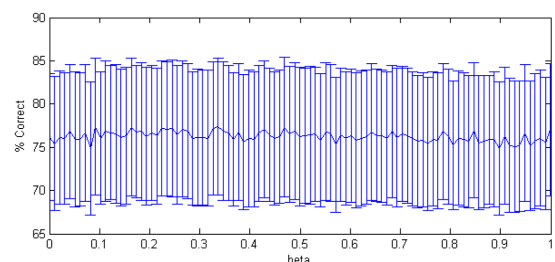

Abbildung 5.13.2.2: Genauigkeit bei Startmargin unter MICRA Margin, skaliert mit maximaler Norm

5.13.3. Shrinking Margin

Auch bei der Shrinking Margin ist die Wahl der Startmargin irrelevant (Abbildung 5.13.3.1). Das variable Shrinking erweist sich als im Schnitt besser als das konstante (Abbildung 5.13.3.2). Die das Shrinking beeinflussenden Parameter zeigen unterschiedliche Eigenschaften: Der Shrinkingfaktor c für das konstante Shrinking sollte möglichste hoch gewählt werden (Abbildung 5.13.3.3), während der Exponent e der variablen Variante eher bei niedrigeren Werten für höhere Genauigkeit sorgt (Abbildung 5.13.3.4).

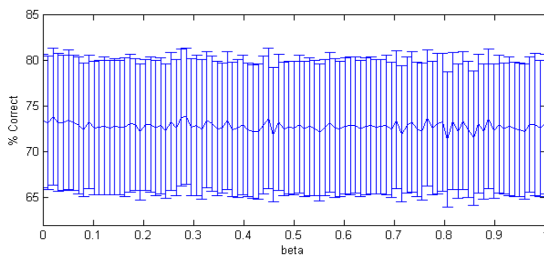

Abbildung 5.13.3.1: Genauigkeit bei Startmargin unter Shrinking Margin, skaliert mit maximaler Norm

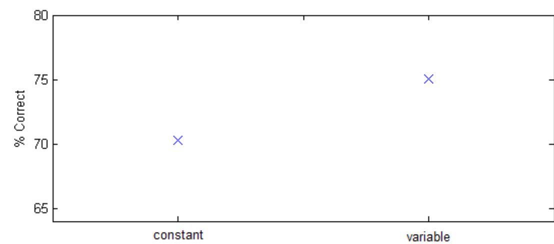

Abbildung 5.13.3.2: Durchschnittliche Genauigkeit bei Wahl der Shrinking Margin Varianten

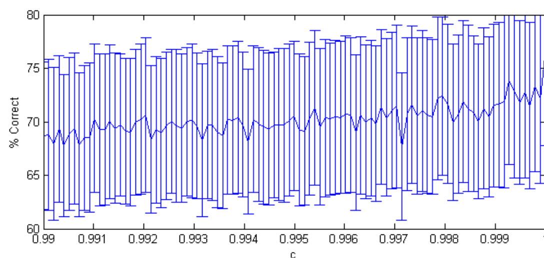

Abbildung 5.13.3.3: Genauigkeit bei konstantem Shrinkingfaktor c unter konstanter Shrinking Margin

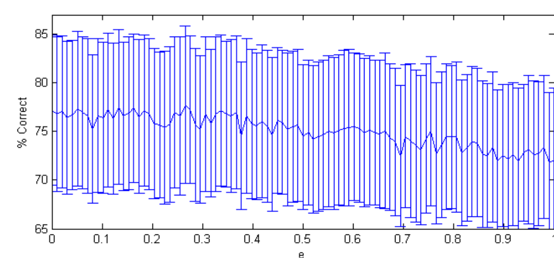

Abbildung 5.13.3.4: Genauigkeit bei Exponent e unter variabler Shrinking Margin

5.13.4. Uneven Margin

Bei steigender Differenz der unterschiedlichen Margins der Variante Uneven Margin fällt im Schnitt die Genauigkeit leicht (Abbildung 5.13.4.1). Daraus lässt sich schließen, dass der positive Effekt von passend zur Instanzenanzahl gewählten Parametern n und p von einem negativen Effekt bei unpassender Wahl mehr als ausgeglichen werden. Ein Skalieren der Margins mit der durchschnittlichen quadratischen Norm der Trainingsinstanzen sorgt für eine minimale Verschlechterung der Genauigkeit (Abbildung 5.13.4.2).

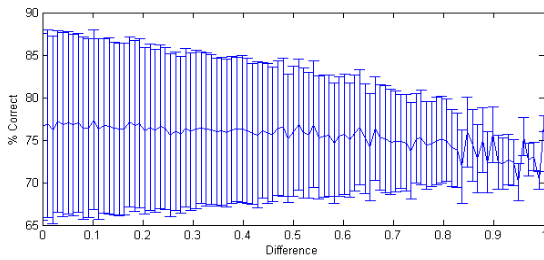

Abbildung 5.13.4.1: Genauigkeit bei Differenz der beiden Marginparameter n und p unter Uneven Margin

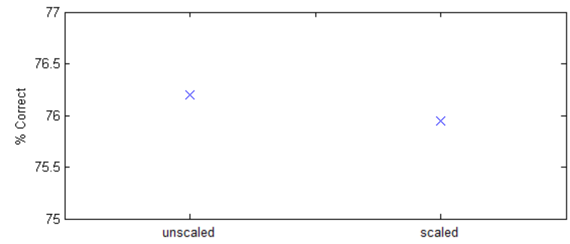

Abbildung 5.13.4.2: Durchschnittliche Genauigkeit bei Skalieren der Marginparameter mit der durchschnittlichen Norm der Trainingsdaten unter Uneven Margin

5.13.5. Constant Update

Bei Verwendung des Constant Update empfiehlt sich laut der Ergebnisse eine Lernrate im Bereich von 0,1 bis 0,2 (Abbildung 5.13.5.1). Eine Verbesserung der Genauigkeit durch die Technik des Unlearnings trat nicht ein (Abbildung 5.13.5.2). Dass der Threshold bei höheren Werten zu besserer Genauigkeit führt, lässt sich damit verbunden dadurch erklären, dass es bei höheren Werten kaum zu Unlearning Schritten kommt (Abbildung 5.13.5.3).

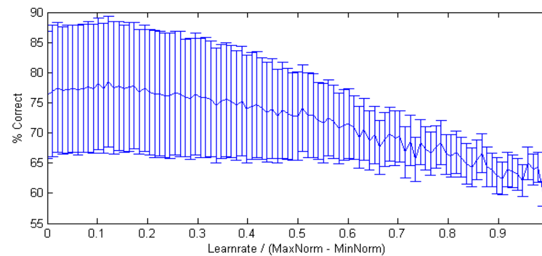

Abbildung 5.13.5.1: Genauigkeit bei Lernrate unter Constant Update, skaliert mit Differenz von maximaler und minimaler Norm der Trainingsdaten

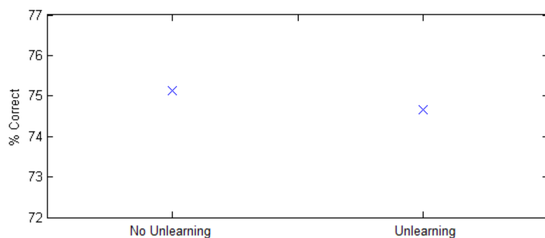

Abbildung 5.13.5.2: Durchschnittliche Genauigkeit bei Verwendung der Unlearning Technik und ohne Verwendung

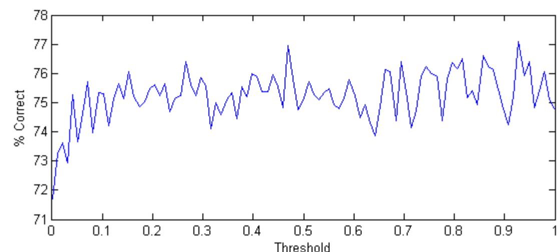

Abbildung 5.13.5.3: Genauigkeit bei Unlearning Threshold, skaliert mit ($maxNorm^5$-$maxnorm^2$)

5.13.6. MICRA Update

Das MICRA Update profitiert von einer Startlernrate von mindestens 0,1 (Abbildung 5.13.6.1). Wählt man sie abhängig von der Startmargin, dann sollte der dafür zuständige Parameter δ auf einen möglichst niedrigen Wert, allerdings höher als 0,1 gesetzt werden (Abbildung 5.13.6.2). Die Verfallsrate der Lernrate, gesteuert von ζ, zeigt keinen Einfluss auf die Genauigkeit (Abbildung 5.13.6.3).

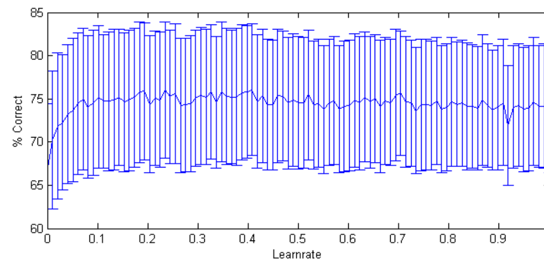

Abbildung 5.13.6.1: Genauigkeit bei Lernrate unter MICRA Update

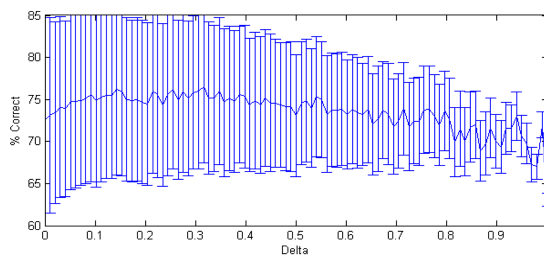

Abbildung 5.13.6.2: Genauigkeit bei Wahl der Lernrate abhängig von Startmargin

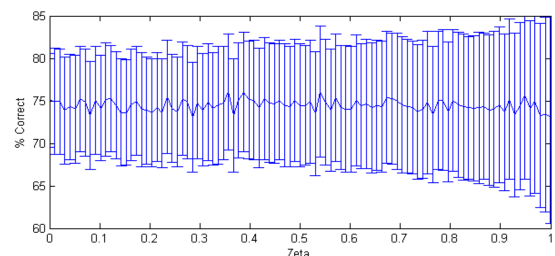

Abbildung 5.13.6.3: Genauigkeit bei Verfallsrate der Lernrate

5.13.7. Passive-Aggressive Update

Beim Passive-Aggressive Update schlägt sich die Variante PA-II auf unseren Daten am erfolgreichsten (Abbildung 5.13.7.1). Auffallend ist, dass PA-I, das wie PA-II für verrauschte Datensätze konzipiert wurde, weniger korrekte Ergebnisse liefert als das ursprüngliche PA Update.
Für den Aggressivitätsparameter C sollte in Verbindung mit den hier verwendeten Datensätzen ein Wert von mindestens 0,1 gewählt worden, bei höheren Werten bleibt die Genauigkeit relativ konstant (Abbildung 5.13.7.2).

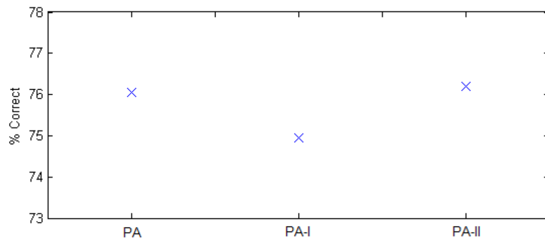

Abbildung 5.13.7.1: Durchschnittliche Genauigkeit bei Wahl der PA Variante

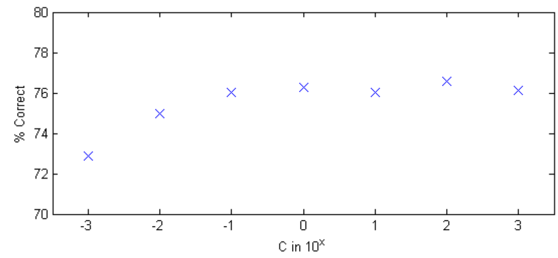

Abbildung 5.13.7.2: Genauigkeit bei Wahl des Aggressivitätsparameters C

5.13.8. Pegasos Update

Das Pegasos Update zeigt eine klare Verbesserung der Genauigkeit bei Wahl eines höheren Projektionswerts.

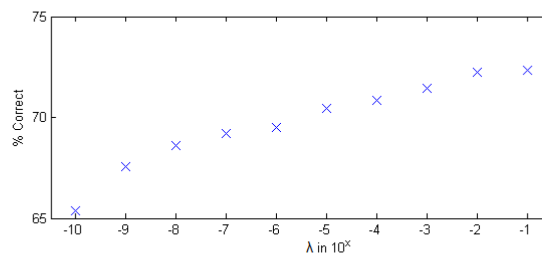

Abbildung 5.13.8.1: Genauigkeit bei Projektionsparameter λ

5.13.9. Shrinking Update

Die Lernrate des Shrinking Updates hat kaum Einfluss auf die Genauigkeit, allenfalls ist eine marginale Tendenz zu einer höheren Genauigkeit bei niedrigerer Lernrate auszumachen (Abbildung 5.13.9.1). Bei der zu wählenden Variante und dem konstanten Shrinkingfaktor c ähnelt das Shrinking Update der Shrinking Margin: Die variable Variante zeigt eine besserer Performanz (Abbildung 5.13.9.2), wählt man aber die konstante, so sollte der zugehörige Shrinkingfaktor c möglichst hoch gewählt werden (Abbildung 5.13.9.3). Im Gegensatz zur Shrinking Margin lässt sich mit dem Exponenten e bei der variablen Variante kein Einfluss auf die Genauigkeit nehmen (Abbildung 5.13.9.4).

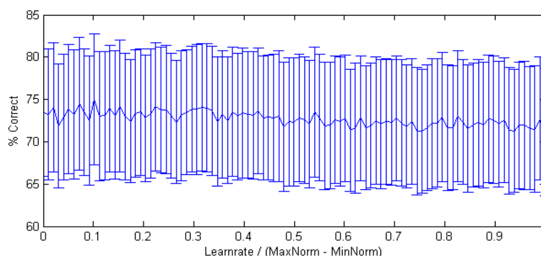

Abbildung 5.13.9.1: Genauigkeit bei Lernrate

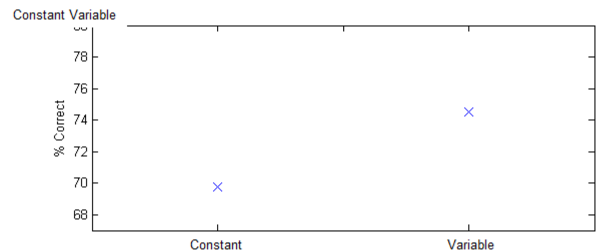

Abbildung 5.13.9.2: Durchschnittliche Genauigkeit bei Wahl der Shrinking Variante

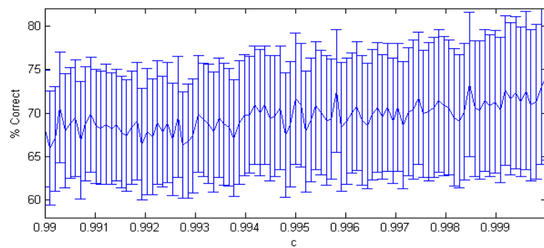

Abbildung 5.13.9.3: Genauigkeit bei Shrinkingfaktor c des konstanten Shrinkings

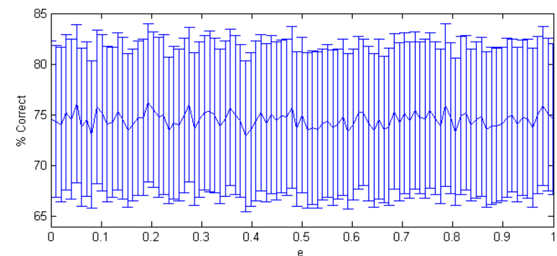

Abbildung 5.13.9.4: Genauigkeit bei Exponent e des variablen Shrinkings

5.14. Gelernte Hyperparameter

5.14.1. Modell der erreichbaren Genauigkeit

Zum Lernen der Funktionen f(**x**,**y**) = a wurden vier Lernalgorithmen ausgewählt: Lineare Regression, M5-Rules, Regression-SVM sowie k-NearestNeighbor. Tabelle 5.14.1.1 zeigt den root mean squared error (5.14.1.1) der dabei erstellten Modelle, der den auf einem Testdatensatz ermittelten Unterschied zwischen den Ausgaben der Modelle und den wahren Zielwerten t berechnet.

$$\sqrt{\frac{\sum (f(\mathbf{x},\mathbf{y}) - t)^2}{n}} \tag{5.14.1.1}$$

M5-Rules erreicht hier mit im Schnitt 12.0859 den niedrigsten Wert. Tabelle 5.14.1.2 setzt anhand des root relative squared errors

$$\sqrt{\frac{\sum (f(\mathbf{x},\mathbf{y}) - t)^2}{\sum (t - \bar{t})^2}} \tag{5.14.1.2}$$

die errechneten Modelle in Relation zu einem Modell, welches immer den Durchschnittswert der Zielwerte vorhersagt. Auch hier schneidet M5-Rules mit im Schnitt 73.1683 % am besten ab.

Update \ Margin	**Dynamic**	**MICRA**	**Shrinking**	**Uneven**
Constant	LinearReg.:15.3766 M5-Rules:14.647 SVM:19.3742 kNN:15.2533	LinearReg.:12.1638 M5-Rules:10.3516 SVM:15.3977 kNN:11.9896	LinearReg.:12.6442 M5-Rules:10.8264 SVM:15.3101 kNN:12.7133	LinearReg.:11.7884 M5-Rules:9.7258 SVM:15.2706 kNN:11.5606
MICRA	LinearReg.:15.7912 M5-Rules:14.902 SVM:18.8197 kNN:15.7071	LinearReg.:10.4261 M5-Rules:12.1822 SVM:16.6448 kNN:13.6664	LinearReg.:10.8768 M5-Rules:12.8132 SVM:16.6856 kNN:14.0748	LinearReg.:10.1286 M5-Rules:12.0474 SVM:16.0689 kNN:13.4617
PA	LinearReg.:12.552 M5-Rules:10.6069 SVM:15.8285 kNN:12.2761	LinearReg.:12.437 M5-Rules:10.3966 SVM:15.3403 kNN:12.0785	LinearReg.:13.7383 M5-Rules:12.0683 SVM:16.2012 kNN:13.8279	LinearReg.:12.8617 M5-Rules:11.276 SVM:16.3355 kNN:12.789
Pegasos	LinearReg.:16.6326 M5-Rules:15.2734 SVM:17.899 kNN:16.789	LinearReg.:13.3401 M5-Rules:10.6113 SVM:15.7957 kNN:12.9209	LinearReg.:13.6043 M5-Rules:11.3273 SVM:15.0189 kNN:13.922	LinearReg.:13.1421 M5-Rules:11.1247 SVM:15.7003 kNN:13.2205
Shrinking	LinearReg.:17.9278 M5-Rules:16.3378 SVM:20.2912 kNN:17.2079	LinearReg.:12.3768 M5-Rules:10.3955 SVM:15.695 kNN:12.2387	LinearReg.:14.5881 M5-Rules:12.7397 SVM:17.3451 kNN:14.6835	LinearReg.:14.0141 M5-Rules:12.0642 SVM:17.4264 kNN:13.9511
Ø:	LinearReg.:13.32053 M5-Rules:12.0859 SVM:16.6224 kNN:13.7166	***Konfigurationen:***	*LinearReg:*	*attributeSelectionMethod = M5, ridge = 1.0E-8*
			M5-Rules:	*minNumInstances = 4, pruned, smoothed*
			SVM:	*epsilonSVR, epsilon = 0.001, cost = 1.0, gamma = 1/maxIndex, loss = 0.1, normalized*
			kNN:	*k = 23, distanceWeighting = 1/distance*

Tabelle 5.14.1.1: Root mean squared error der Modelle f(**x**,**y**) = a der verschiedenen Update-Margin-Kombinationen

Update \ Margin	**Dynamic**	**MICRA**	**Shrinking**	**Uneven**
Constant	LinearReg.:81.3105 % M5-Rules:77.1872 % SVM:86.2706 % kNN:80.3827 %	LinearReg.:81.1664 % M5-Rules:68.0486 % SVM:86.4596 % kNN:78.8164 %	LinearReg.:83.1053 % M5-Rules:71.1002 % SVM:86.2649 % kNN:83.4914 %	LinearReg.:78.3702 % M5-Rules:64.4998 % SVM:82.7853 % kNN:76.6680 %
MICRA	LinearReg.:86.0277 % M5-Rules:80.4980 % SVM:89.6875 % kNN:84.8527 %	LinearReg.:84.3816 % M5-Rules:74.4593 % SVM:90.12184 % kNN:83.5310 %	LinearReg.:85.7839 % M5-Rules:77.8090 % SVM:90.7020 % kNN:85.4701 %	LinearReg.:84.3324 % M5-Rules:75.74086 % SVM:89.7752 % kNN:84.6328 %
PA	LinearReg.:80.697 % M5-Rules:68.04218 % SVM:86.5708 % kNN:78.7502 %	LinearReg.:81.0558 % M5-Rules:68.2913 % SVM:86.0604 % kNN:79.3385 %	LinearReg.:84.6718 % M5-Rules:75.1465 % SVM:88.2959 % kNN:86.1031 %	LinearReg.:79.9887 % M5-Rules:69.9890 % SVM:84.7675 % kNN:79.3798 %
Pegasos	LinearReg.:92.4824 % M5-Rules:85.3519 % SVM:93.04223 % kNN:93.8218 %	LinearReg.:84.8149 % M5-Rules:67.8761 % SVM:90.4551 % kNN:82.6497 %	LinearReg.:90.4585 % M5-Rules:75.63200 % SVM:91.7119 % kNN:92.5668 %	LinearReg.:85.1675 % M5-Rules:71.3958 % SVM:89.9632 % kNN:84.8465 %
Shrinking	LinearReg.:87.0695 % M5-Rules:80.4636 % SVM:87.7299 % kNN:84.7488 %	LinearReg.:78.8489 % M5-Rules:66.9791 % SVM:85.1444 % kNN:78.8550 %	LinearReg.:85.4960 % M5-Rules:74.2061 % SVM:90.9257 % kNN:85.5284 %	LinearReg.:81.7556 % M5-Rules:70.6488 % SVM:88.4703 % kNN:81.6986 %
Ø:	LinearReg.: 83.8488 % M5-Rules: 73.1683 % SVM: 88.2602 % kNN: 83.30660 %		***Konfigurationen siehe Tabelle 8.2***	

Tabelle 5.14.1.2: Root relative squared error der Modelle f(**x**,**y**) = a der verschiedenen Update-Margin-Kombinationen

5.14.2. Modell der Hyperparameter

Da die mit M5-Rules erstellten Modelle sich als die vergleichsweise genauesten ergaben, wurden sie für das weitere Vorgehen des Lernens passender Hyperparameter verwendet. Mit ihnen wurden, wie in Kapitel 4.4 beschrieben, Daten für Modelle der Form $g(\mathbf{x}) = y_i$ für jeden in den Perzeptronvarianten vorkommenden numerischen Hyperparameter y_i erstellt. Eine Übersicht der root relative squared errors befindet sich in Tabelle 5.14.2.1. Diese Modelle wurden daraufhin wieder auf Zufallsdatensätzen angewandt, und die dabei entstehenden Perzeptron-Algorithmen experimentell evaluiert. Abbildung 5.14.2.1 zeigt dabei die Verbesserung in der Genauigkeit der Perzeptron-Algorithmen mit nach diesen Modellen gewählten Hyperparametern im Vergleich zu einer Zufallsauswahl. Da eines der Ziele der meisten in dieser Arbeit vorgestellten Algorithmen darin liegt, die Genauigkeit von SVMs zu erreichen, ist als Referenzwert die erreichte Genauigkeit von mit LibSVM [CL011] erstellten SVMs auf den durch unseren Generator erstellten Datensätzen angegeben (rot). Zusätzlich wurde noch ein kostspieliger experimenteller RandomSearch nach guten Hyperparameterwerten durchgeführt, der ohne Modelle $f(\mathbf{x},\mathbf{y})$ und $g(\mathbf{x})$ auskommt (grün). Dafür wurden 1000 zufällig generierte Datensätze mit jeweils 100 zufällig generierten Hyperparameterkonstellationen kombiniert, experimentell evaluiert und die besten Konstellationen für jeden Datensatz gespeichert. Da dieses Vorgehen sehr rechenintensiv ist konnten dafür nur diese 1000 Datenreihen gesammelt werden, sodass lediglich ein verlässlicher Durchschnittswert aller Varianten ermittelt werden konnte anstatt nach Varianten getrennte Werte.

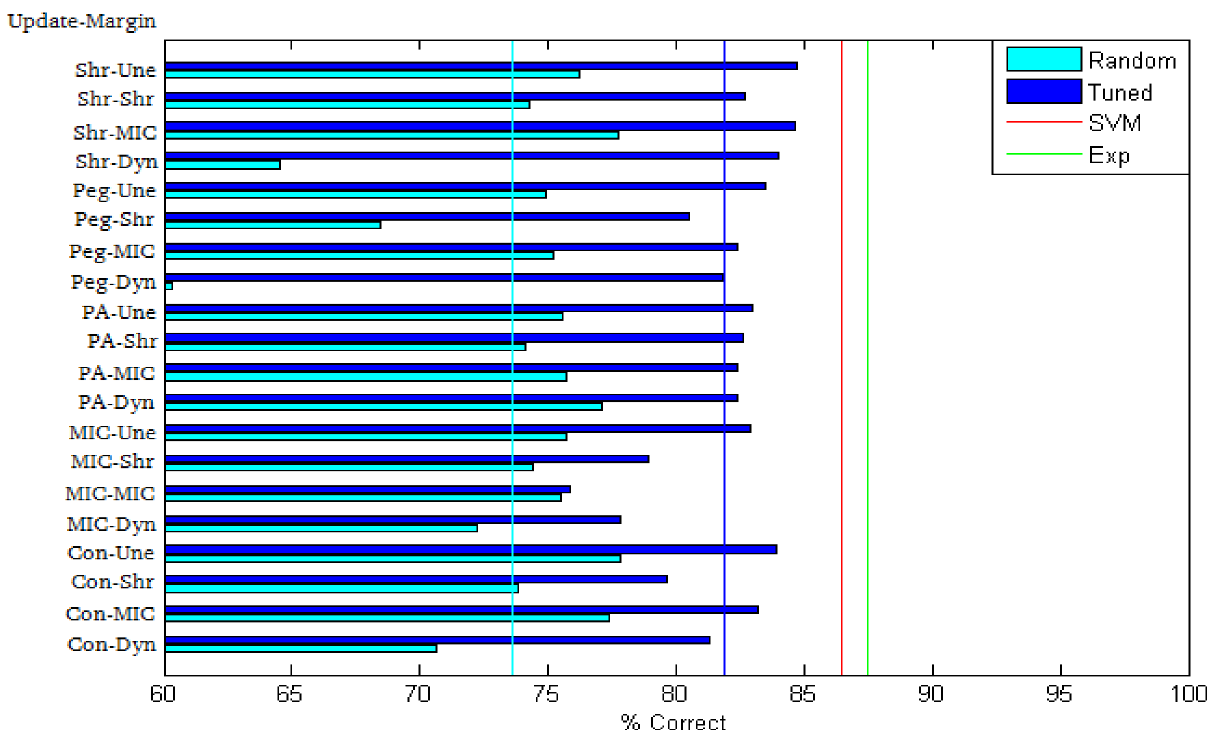

Abbildung 5.14.2.1: Genauigkeit der Perceptron Algorithmen bei zufälliger Hyperparameterwahl (Random), per g(x) bestimmter Hyperparameter (Tuned) sowie durchschnittliche Genauigkeit von SVMs und experimenteller Zufallssuche nach guten Hyperparametern. Vertikale Balken geben Durchschnittswerte an.

Es zeigt sich, dass die bei Zufallswahl der Hyperparameter als schwer kombinierbar angesehenen Update- und Marginvarianten, wie das Pegasos- oder ShrinkingUpdate in Kombination mit der DynamicMargin, bei guter Hyperparameterwahl ebenfalls die Genauigkeitswerte der anderen Varianten erreichen. Hier gab es wohl ungünstige Bereiche innerhalb der Parametergrenzen, welche die Durchschnittsgenauigkeit bei Zufallsauswahl gesenkt hatten und dann bei gezielter Parameterwahl nicht mehr ausgewählt wurden.

Potentiell sind für die Perzeptronvarianten SVM-ähnliche Werte um 87% erreichbar, wie der experimentelle RandomSearch zeigt. Durch Parameterwahl per $g(\mathbf{x})$ ist immerhin eine Steigung der durchschnittlichen Genauigkeit von 73,58% bei Zufallswahl auf 81,91% zu verzeichnen. Die fehlenden 5,59 Prozentpunkte bis zum Optimum von 87,50% der experimentellen Zufallssuche sind dabei den Ungenauigkeiten der Modelle $f(\mathbf{x},\mathbf{y})$ und $g(\mathbf{x})$ anzulasten. Der Flaschenhals für weitere Verbesserung sind hier höchstwahrscheinlich die Modelle $f(\mathbf{x},\mathbf{y})$ mit ihren 12% root mean squared error.

Update \ Margin	Dynamic	MICRA	Shrinking	Uneven
Constant	I: 76.7991 % N: 72.1481 % α: 86.4111 % λ: 53.2496 % ϵ: 79.542 % β: 96.8339 % η: 71.9978 % t: 72.0518 % bias: 81.1406 %	I:71.3488 % N: 92.3836 % α:89.9404 % λ:78.1024 % ϵ: 55.4727 % β:87.9732 % η: 83.9081 % t: 77.3275 % bias: 91.6054 %	I:82.451 % N:93.7849 % α:94.9095 % λ: 69.4942 % b: 71.6494 % e: 67.5519 % c: 92.4406 % v: 69.2169 % η: 83.4087 % t: 75.0537 % bias: 93.4246 %	I: 76.4842 % N: 99.6822 % α: 103.2889 % λ: 44.9408 % n: 72.5049 % p: 62.9356 % scale: 100.0938 % η: 98.6376 % t: 75.1947 % bias: 93.4515 %
MICRA	I: 99.9537 % N: 90.3478 % α: 97.7879 % λ: 82.028 % ϵ: 100.0873 % β: 55.765 % η: 98.885 % ζ: 100.0671 % startmargin: 55.765 % δ: 96.4632 % bias: 99.2607 %	I: 93.7975 % N: 74.0257 % α: 86.4539 % λ: 82.3881 % ϵ: 80.2306 % β: 72.4653 % η: 50.159 % ζ: 85.267 % startmargin: 72.4653 % δ: 90.7221 % bias: 95.1444 %	I: 88.4566 % N: 76.7626 % α: 90.0944 % λ: 79.7748 % b: 59.398 % e: 92.2511 % c: 89.5516 % v: 55.7666 % η: 85.0244 % ζ: 62.1909 % startmargin: 59.398 % δ: 93.4588 % bias: 91.1307 %	I: 96.4916 % N: 95.3643 % α: 90.9333 % λ: 85.5972 % n: 46.9687 % p: 48.3373 % scale: 100 % η: 94.3505 % ζ: 68.1441 % startmargin: 48.8356 % δ: 64.6577 % bias: 91.1193 %
PA	I: 61.4794 % N: 59.0619 % α: 75.4058 % λ: 64.4964 % ϵ: 83.2849 % β: 64.006 % C: 64.619 % V: 88.3811 % bias: 59.8235 %	I: 78.955 % N: 86.0499 % α: 77.1886 % λ: 77.8004 % ϵ: 57.9429 % β: 63.8557 % C: 86.6546 % V: 87.3691 % bias: 87.9048 %	I: 79.0656 % N: 66.3374 % α: 87.3569 % λ: 83.0857 % b: 93.8363 % e: 84.1102 % c: 84.4896 % v: 58.9851 % C:88.8642 % V:88.5505 % bias: 84.9134 %	I: 82.4891 % N: 69.0547 % α: 92.5597 % λ: 73.7815 % n: 50.0402 % p: 62.9497 % scale: 100.1243 % C: 99.3412 % V: 99.9462 % bias: 80.4618 %
Pegasos	I: 97.4569 % N: 70.2703 % α: 81.3712 % λ: 83.2024 % ϵ: 78.3577 % β: 86.4108 % peg-λ: 85.5275 % bias: 91.7865 %	I: 78.8239 % N: 81.4148 % α: 86.3877 % λ: 81.4985 % ϵ: 88.399 % β: 82.8156 % peg-λ: 70.62 % bias: 95.1351 %	I: 75.957 % N: 85.1703 % α: 87.8217 % λ: 81.1897 % b: 82.5071 % e: 66.8698 % c: 95.8846 % v: 81.8317 % peg-λ: 96.8106 % bias: 89.9832 %	I: 81.9568 % N: 85.2756 % α: 82.6651 % λ: 86.7773 % n: 92.9742 % p: 84.6178 % scale: 100 % peg-λ: 89.1138 % bias: 86.5216 %
Shrinking	I: 99.1576 % N: 92.6542 % α: 92.7268 % λ: 70.4234 % ϵ: 80.4208 % β: 90.592 % e: 97.9325 % η: 99.4605 % c: 93.8349 % v: 71.2332 % bias: 94.7131 %	I: 99.5121 % N: 94.9074 % α: 93.2677 % λ: 72.4549 % ϵ: 81.6458 % β: 90.3142 % e: 97.6328 % η: 99.5838 % c: 94.2501 % v: 72.6079 % bias: 95.0397 %	I: 90.0605 % N: 94.2855 % α: 89.703 % λ: 56.9028 % b: 92.5701 % e: 98.3465 % c: 96.0295 % v: 86.2227 % e: 94.7641 % η: 83.9856 % c: 92.0532 % v: 81.4661 % bias: 93.5307 %	I: 88.9998 % N: 78.4338 % α: 90.143 % λ: 71.7104 % n: 83.6801 % p: 91.4948 % scale: 100 % e: 98.7426 % η: 75.2825 % c: 76.8893 % v: 57.8448 % bias: 92.2146 %
Ø:	82,5612 %			

Tabelle 5.14.2.1: Root relative squared error der Modelle $g(\mathbf{x}) = y_i$ der verschiedenen Update-Margin-Kombinationen

6. Fazit und Ausblick

Diese Arbeit hat verschiedene Perzeptron-ähnliche Algorithmen, Algorithmenkombinationen sowie mit diesen Algorithmen verbindbare Techniken hinsichtlich ihrer Effektivität untersucht. Es zeigt sich, dass die Wahl korrekter Hyperparameter der untersuchten Algorithmen und Techniken einen entscheidenden Einfluss auf die Genauigkeit des entstehenden Perzeptron-Algorithmen hat. Zwar gibt es auch einige Parameter, bei denen keine Korrelation zwischen ihren Werten und der Genauigkeit festgestellt wurde, der Großteil der Parameter bietet aber die Möglichkeit, die Genauigkeit durch geschickte Parameterwahl zu erhöhen.
Es ist möglich, Modelle zu lernen, die auf Basis der Eingabedaten Werte zur Einstellung der Hyperparameter vorschlagen, welche die Genauigkeit verbessern. Diese Verbesserungen reizen allerdings noch nicht die potentiell erreichbaren Genauigkeitswerte aus, die durch rechenintensives Ausprobieren von Hyperparameterkombination oder durch die ebenfalls rechenintensive SVM erreicht werden. Insgesamt zeigt sich jedoch, dass neuartige Perzeptron-Algorithmen bei guter Parameterwahl der SVM Konkurrenz machen können, was die Genauigkeit anbelangt.
Für weitere Untersuchungen empfiehlt sich das Finden von Modellen zum Zweck des Lernens von Hyperparametervorschlägen, welche genauer als die in dieser Arbeit untersuchten die Daten abbilden, um weitere Verbesserungen der Perzeptron-Algorithmen durch noch bessere Hyperparameter zu ermöglichen. Neben besseren Modellen könnten auch die Hyperparametermodelle $g(\mathbf{x})$ auf Basis von experimentellen Daten gelernt werden, anstatt wie in dieser Arbeit die Modelle $f(\mathbf{x},\mathbf{y})$ für die Generierung dieser Daten zu verwenden. Damit würde die Ungenauigkeit der Modelle $f(\mathbf{x},\mathbf{y})$ nicht mehr ins Gewicht fallen, allerdings müssen dafür höhere zeitliche bzw. rechnerische Ressourcen aufgewandt werden. Sind diese vorhanden, könnten damit auch diejenigen Bereiche der Parameterwerte untersucht werden, die in dieser Arbeit nicht behandelt werden konnten, wie zum Beispiel höhere Werte für die Anzahl der Iterationen und reduzierten Epochen.
Neben der in dieser Arbeit behandelten Genauigkeit der verschiedenen Perzeptronvarianten ist für den praktischen Gebrauch dieser Algorithmen auch die Effizienz ein wichtiges Kriterium. Ein Update des Gewichtsvektors und der Margin ist je nach gewählter Variante unterschiedlich rechenintensiv und neben der Genauigkeit hängt auch die Anzahl der durchgeführten Epochen von der Hyperparameterwahl ab. Hier bietet sich eine Untersuchung des Trade-Offs zwischen Genauigkeit und Effizienz verschiedener Perzeptron-Algorithmen an.
Die aktuellste hier behandelte Perzeptronvariante stammt aus dem Jahr 2012 und ist damit zum Zeitpunkt dieser Arbeit zwei Jahre alt, was nahegelegt, dass auf diesem Gebiet weiterhin geforscht und neue Algorithmen und Techniken entwickelt werden. Auch diese neueren Perzeptron-Algorithmen könnten in Perceptrovement aufgenommen und daraufhin untersucht werden, ob sie zu einer weiteren Verbesserung von Genauigkeit des Lernens eines Perzeptrons führen können.

7. Verzeichnisse

7.1. Tabellenverzeichnis

Tabelle 4.1.1: Parameter und Parametergrenzen des Datengenerators....25
Tabelle 4.2.1: Parameter und Parametergrenzen des Algorithmusgenerators....27
Tabelle 4.3.1: Zu bestimmende Hyperparameter bei Durchlauf des Experimentierframeworks....31
Tabelle 5.14.1.1: Root mean squared error der Modelle $f(\mathbf{x},\mathbf{y}) = a$....49
Tabelle 5.14.1.2: Root relative squared error der Modelle $f(\mathbf{x},\mathbf{y}) = a$....49
Tabelle 5.14.2.1: Root relative squared error der Modelle $g(\mathbf{x}) = y_i$....52

7.2. Algorithmenverzeichnis

Algorithmus 2.1 Perzeptron-Algorithmus nach Rosenblatt....8
Algorithmus 3.1 PAUM....11
Algorithmus 3.2 Passive-Aggressive Algorithmus....12
Algorithmus 3.3 Pegasos....14
Algorithmus 3.4 MICRA....15
Algorithmus 3.5 Margin Perceptron with Constant Shrinking....17
Algorithmus 3.6 Margin Perceptron with Variable Shrinking....17
Algorithmus 3.7 Perceptron with Dynamic Margin....19
Algorithmus 3.8 Voted/Longest Survivor Perceptron....20
Algorithmus 3.9 Perzeptron-Algorithmus mit Reduzierte Epochen....21
Prozedur 3.1 Budget....21
Algorithmus 3.10 Margin Perceptron with Unlearning....23

7.3. Abbildungsverzeichnis

Abbildung 4.1.1: Veranschaulichung der Radienparameter....26
Abbildung 4.2.1: Veranschaulichung der durch λ-Trick hinzugefügten Dimensionen....28
Abbildung 4.3.1: Vergleich GridSearch und RandomSearch....31
Abbildung 4.4.1: Datenfluss bei Lernen der Hyperparametermodelle $g(\mathbf{x}) = y_i$....32
Abbildung 5.1.1: Constant Update und Margin Varianten....33
Abbildung 5.1.2: MICRA Update und Margin Varianten....34
Abbildung 5.1.3: MICRA Update und Margin Varianten gefiltert....34
Abbildung 5.1.4: Passive-Aggressive Update und Margin Varianten....34
Abbildung 5.1.5: Passive-Aggressive Update und Margin Varianten gefiltert....34
Abbildung 5.1.6: Pegasos Update und Margin Varianten....34
Abbildung 5.1.7: Shrinking Update und Margin Varianten....35
Abbildung 5.1.8: Shrinking Update und Margin Varianten gefiltert....35
Abbildung 5.2.1: Genauigkeit bei Gesamtanzahl der Instanzen....35
Abbildung 5.2.2: Genauigkeit bei Differenz der Instanzenanzahl beider Cluster....35
Abbildung 5.2.3: Genauigkeit der Margin Varianten bei Differenz der Instanzenanzahl....36
Abbildung 5.2.4: Genauigkeit der Uneven Margin bei Differenz der Instanzenanzahl....36
Abbildung 5.3.1: Genauigkeit bei Summe der Radien unter Constant Update....36
Abbildung 5.3.2: Genauigkeit bei Summe der Radien unter MICRA Update....36

Abbildung 5.3.3: Genauigkeit bei Summe der Radien unter Passive-Aggressive Update....................37
Abbildung 5.3.4: Genauigkeit bei Summe der Radien unter Pegasos Update....................................37
Abbildung 5.3.5: Genauigkeit bei Summe der Radien unter Shrinking Update.................................37
Abbildung 5.4.1: Genauigkeit bei Anzahl Attribute unter Constant Update....................................37
Abbildung 5.4.2: Genauigkeit bei Anzahl Attribute unter MICRA Update..37
Abbildung 5.4.3: Genauigkeit bei Anzahl Attribute unter Passive-Aggressive Update........................38
Abbildung 5.4.4: Genauigkeit bei Anzahl Attribute unter Pegasos Update.......................................38
Abbildung 5.4.5: Genauigkeit bei Anzahl Attribute unter Shrinking Update....................................38
Abbildung 5.5.1: Genauigkeit bei Rauschen unter Constant Update..39
Abbildung 5.5.2: Genauigkeit bei Rauschen unter MICRA Update...39
Abbildung 5.5.3: Genauigkeit bei Rauschen unter Passive-Aggressive Update..................................39
Abbildung 5.5.4: Genauigkeit bei Rauschen unter Pegasos Update...39
Abbildung 5.5.5: Genauigkeit bei Rauschen unter Shrinking Update...39
Abbildung 5.6.1: Genauigkeit bei maximal erlaubter Iterationen..40
Abbildung 5.7.1: Genauigkeit bei Anzahl reduzierter Epochen...40
Abbildung 5.7.2: Genauigkeit bei Anzahl reduzierten Epochen, Daten gefiltert...............................40
Abbildung 5.8.1: Genauigkeit bei α-Bound...41
Abbildung 5.9.1: Genauigkeit bei λ-Trick, skaliert mit maximaler Norm..41
Abbildung 5.10.1: Durchschnittliche Genauigkeit bei Wahl der Vorhersagemethoden.......................41
Abbildung 5.11.1: Genauigkeit bei Bias, skaliert mit maximaler Norm...42
Abbildung 5.12.1: Genauigkeit bei Budgetlimit, skaliert mit Instanzenanzahl..................................42
Abbildung 5.12.2: Durchschnittliche Genauigkeit der Budget Heuristiken.......................................42
Abbildung 5.12.3: Genauigkeit bei Anzahl der Supportvektoren, die einen entfernten ersetzen.........43
Abbildung 5.12.4: Genauigkeit bei Ridge Regression Regularisationsfaktor....................................43
Abbildung 5.13.1.1: Genauigkeit bei Genauigkeitsforderung ϵ unter Dynamic Margin......................43
Abbildung 5.13.2.1: Genauigkeit bei Verfallsparameter ϵ unter MICRA Margin...............................43
Abbildung 5.13.2.2: Genauigkeit bei Startmargin unter MICRA Margin, skaliert mit maxNorm.........43
Abbildung 5.13.3.1: Genauigkeit bei Startmargin unter Shrinking Margin, skaliert mit maxNorm.....44
Abbildung 5.13.3.2: Durchschnittliche Genauigkeit bei Wahl der Shrinking Margin Varianten..........44
Abbildung 5.13.3.3: Genauigkeit bei konstantem Shrinkingfaktor c unter konstanter Shrink.Margin 44
Abbildung 5.13.3.4: Genauigkeit bei Exponent e unter variabler Shrinking Margin..........................44
Abbildung 5.13.4.1: Genauigkeit bei Differenz der Parameter n und p unter Uneven Margin............45
Abbildung 5.13.4.2: Genauigkeit bei Skalieren der Marginparameter unter Uneven Margin.............45
Abbildung 5.13.5.1: Genauigkeit bei Lernrate unter Constant Update, skaliert................................45
Abbildung 5.13.5.2: Durchschnittliche Genauigkeit bei Verwendung der Unlearning........................45
Abbildung 5.13.5.3: Genauigkeit bei Unlearning Threshold..45
Abbildung 5.13.6.1: Genauigkeit bei Lernrate unter MICRA Update..46
Abbildung 5.13.6.2: Genauigkeit bei Wahl der Lernrate abhängig von Startmargin..........................46
Abbildung 5.13.6.3: Genauigkeit bei Verfallsrate der Lernrate...46
Abbildung 5.13.7.1: Durchschnittliche Genauigkeit bei Wahl der PA Variante.................................47
Abbildung 5.13.7.2: Genauigkeit bei Wahl des Aggressivitätsparameters C......................................47
Abbildung 5.13.8.1: Genauigkeit bei Projektionsparameter λ...47
Abbildung 5.13.9.1: Genauigkeit bei Lernrate...47
Abbildung 5.13.9.2: Durchschnittliche Genauigkeit bei Wahl der Shrinking Variante........................47
Abbildung 5.13.9.3: Genauigkeit bei Shrinkingfaktor c des konstanten Shrinkings...........................48
Abbildung 5.13.9.4: Genauigkeit bei Exponent e des variablen Shrinkings......................................48
Abbildung 5.14.2.1: Genauigkeit bei Zufallsauswahl, $g(\mathbf{x})$, SVMs und experimenteller Suche...........50

7.4. Literaturverzeichnis

[B09] Christopher M. Bishop: *Pattern Recognition and Machine Learning*, Springer, 2006.

[BB12] James Bergstra and Yoshua Bengio, *Random search for hyper-parameter optimization,* Journal of Machine Learning Research, 3:281-305, 2012

[CDK+06] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, Yoram Singer: *Online Passive-Aggressive Algorithms*, Journal of Machine Learning Research, 7:551–585, 2006.

[CL011] Chih-Chung Chang and Chih-Jen Lin, *LIBSVM : a library for support vector machines.* ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[FS99] Yoav Freund, Robert E. Schapire: *Large Margin Classification Using the Perceptron Algorithm*, Machine Learning, 37(3):277-296, 1999.

[HFH+09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann und I. H. Witten: *The WEKA Data Mining Software: An Update*. Special Interest Group on Knowledge Discovery and Data Mining Explorer Newsletter, 11(1):10–18, 2009.

[HWP12] Steven C.H. Hoi, Jialei Wang, and Peilin Zhao. *LIBOL: A Library for Online Learning Algorithms*. Nanyang Technological University, 2012. Software available at http://libol.stevenhoi.org/

[K11] Tobias Krönke: *Verbesserungen des Perceptron-Algorithmus*, http://www.ke.tu-darmstadt.de/lehre/arbeiten/bachelor/2011/Kroenke_Tobias.pdf/view, 2011.

[KW07] Roni Khardon, Gabriel Wachman: *Noise Tolerant Variants of the Perceptron Algorithm*, Journal of Machine Learning Research, 8:227-248, 2007.

[LZH+02] Yaoyong Li, Hugo Zaragoza, Ralf Herbrich, John Shawe-Taylor, Jaz Kandola: *The Perceptron Algorithm with Uneven Margins*, Proceedings of ICML, 2002.

[PT10] Constantinos Panagiotakopoulos, Petroula Tsampouka: *The Margin Perceptron with Unlearning,* ICML (2010) 855-862 .

[PT11] Constantinos Panagiotakopoulos, Petroula Tsampouka: Th*e Perceptron with Dynamic Margin*, ALT (2011) 204-218.

[PT12] Constantinos Panagiotakopoulos, Petroula Tsampouka: The Role of Weight Shrinking in Large Margin Perceptron Learning. arXiv:1205.4698 2012.WV

[SSS+07] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, Andrew Cotter: *Pegasos: Primal Estimated sub-GrAdient SOlver for SVM*, Proceedings of the 24th Annual International Conference on Machine Learning 807–814, 2007.

[TS07] Petroula Tsampouka, John Shawe-Taylor: *Approximate Maximum Margin Algorithms with Rules Controlled by the Number of Mistakes*, Proceedings of the Twenty-Fourth International Conference on Machine Learning(ICML 2007) 903-910.

[WCV10] Zhuang Wang, Koby Crammer, Slobodan Vucetic: *Multi-Class Pegasos on a Budget*, Proceedings of the 27th International Conference on Machine Learning 1143-1150, 2010.

[WV10] Zhuang Wang, Slobodan Vucetic: *Online Passive-Aggressive Algorithms on a Budget*, Int. Conf. on Artificial Intelligence and Statistics (AISTATS) 908-915, 2010.

[ZRL96] Tian Zhang, Ragh Ramakrishnan, Miron Livny: *BIRCH: An Efficient Data Clustering Method for Verly Large Databases*, Proceedings of the 1996 ACM SIGMOD international conference on Management of data 103-114.