Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2004-12

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Kernbasierte Verfahren zur TIS-Erkennung bei Prokaryoten

Thomas Brodag, Nico Pfeifer

am Institut für

Numerische und Angewandte Mathematik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

13. August 2004

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 02
Fax +49 (5 51) 39-1 44 03
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Wir erklären hiermit, daß wir die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Göttingen, den 13. August 2004

Bachelorarbeit

Kernbasierte Verfahren zur TIS-Erkennung bei Prokaryoten

Thomas Brodag, Nico Pfeifer

13. August 2004

Betreut durch Prof. Dr. Waack
Institut für Numerische und Angewandte Mathematik
Georg-August-Universität Göttingen

Inhaltsverzeichnis

1 Einleitung

Das Verarbeiten großer Mengen von genomischen Daten ist ein Hauptbereich der Bioinformatik. Da viele Genome verschiedener Spezies bereits sequenziert sind, gibt es einen großen Bedarf an Methoden, um diese weiter verarbeiten zu können. Hat man das Genom sequenziert, so weiß man noch nicht viel über die Information, die in diesen Daten verborgen ist. Die Aufgabe ist, nun weitere Information aus den reinen DNA- Sequenzen zu gewinnen, also zunächst einmal Gene zu annotieren. Dabei ergibt sich das Problem, die Anfänge der Gene zu bestimmen.
Man sucht also die Stellen von wo an die genetische Information des Gens codiert ist. Diese Startstellen werden im Weiteren auch als TIS (Translation Initiation Sites) bezeichnet.

Um eine Antwort für dieses Problem zu finden, gibt es grundsätzlich unterschiedliche Herangehensweisen, wie, unter anderem, probabilistische Modelle oder maschinelle Lernverfahren.
Bei Eukaryoten gibt es zu diesem Thema diverse Arbeiten, wie zum Beispiel, auf Hidden Markov Modellen beruhende, Methoden wie GeneMark[1], Genscan [2] und EuGène [3].
Bei maschinellen Lernverfahren sind insbesondere Zien et al. zu erwähnen [4]. Sie benutzten Supportvektormaschinen, um die Genstarts von Wirbeltieren vorherzusagen. Zudem verglichen sie ihre Verfahren mit den besten bis dahin bekannten Methoden und erzielten deutlich bessere Resultate.
Bei Prokaryoten gibt es bisher wenige Ansätze zum Thema TIS- Erkennung. Hier sei das Programm Glimmer [5] erwähnt, welches ein interpoliertes Markov Modell verwendet. In unserer Arbeit geht es darum, ein Verfahren von Zien et al.[4], den Locality- Kern, und ein neues Verfahren von Dr. Peter Meinicke [6], den Oligokern, auf Prokaryoten anzuwenden und die Performanz der jeweiligen Methoden zu vergleichen. Unsere Evaluation der Lernmaschinen zeigt, dass der Oligokern noch bessere Ergebnisse erzielt, als der Locality-Kern.
Als Ausblick wäre es also denkbar, den Oligokern im Bereich der TIS- Erkennung einzusetzen. Man könnte ihn mit bestehenden Verfahren kombinieren.
Bisherige Verfahren für die Genvorhersage bei Prokaryoten suchen lange offene Leserahmen und deklarieren dann das erste Startcodon im Leserahmen als Genstart. In diesen Fällen sind die Gene oft zu lang. Man könnte also nun den Oligokern einsetzen, um die in dem Bereich liegenden potentiellen TIS zu bewerten und dann die Beste auszuwählen. Hierdurch könnte man die Performanz der bestehenden Verfahren vermutlich deutlich verbessern.

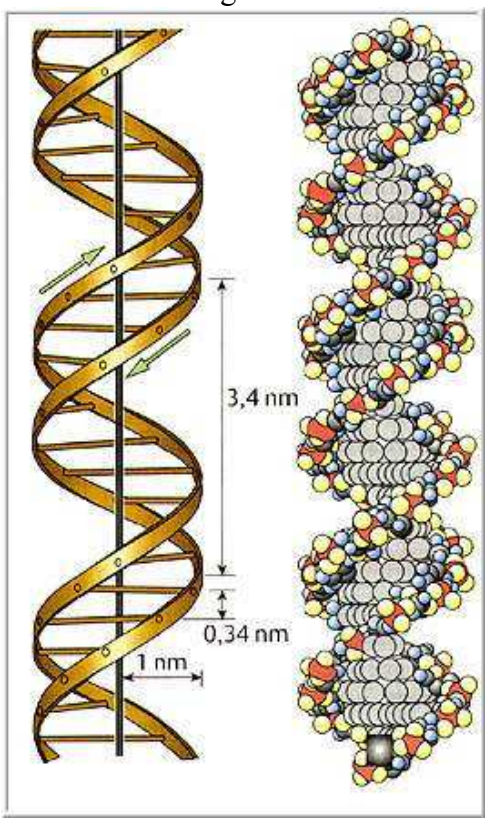
Im folgenden Kapitel werden zuerst die biologischen Grundlagen erläutert. In Kapitel drei gehen wir auf einfache Lernverfahren ein um dann in Kapitel vier zu den kernbasierten Verfahren zu kommen. In Kapitel fünf wird beschrieben, wie unsere Validierung der einzelnen Kerne aus Kapitel vier durchgeführt wurde. Außerdem werden die Ergebnisse der Validierung dargestellt. Danach wird in Kapitel sechs ein abschließender Performanztest der verschiedenen Kerne durchgeführt, woraufhin in Kapitel sieben die algorithmische Umsetzung besprochen wird.

2 Biologische Grundlagen

2.1 Aufbau der DNA

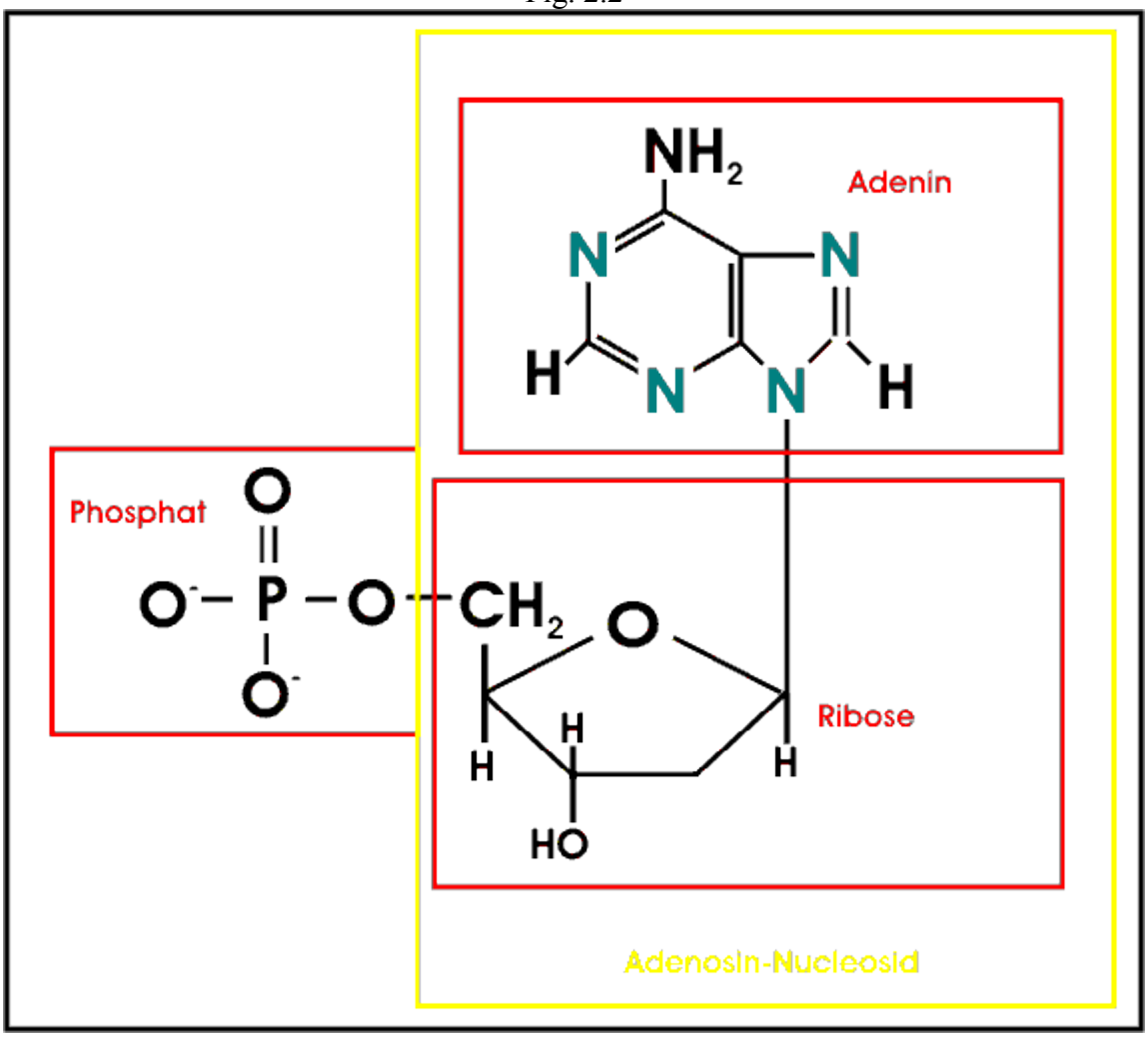
Die DNA (**D**esoxyribo**n**ucleic **A**cid) ist der Träger der genetischen Information. Sie besteht aus zwei Strängen (Doppelstrang), die in Form einer Doppel-Helix (Siehe Fig. 2.1) ineinander verdreht sind.

Fig. 2.1



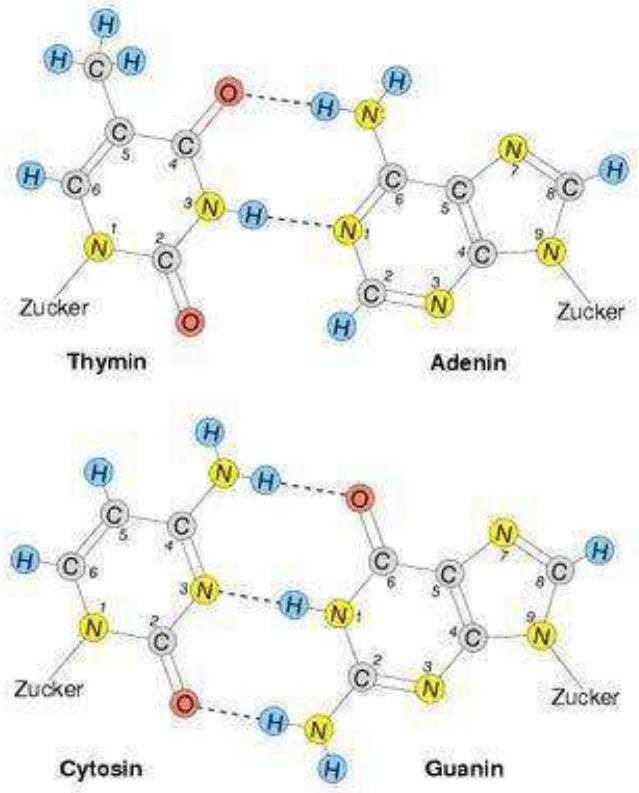
Ein Strang besteht aus einer Menge von sogenannten Nukleotiden.
Ein Nukleosid besteht aus einem Zucker (Desoxyribose), der am C_1' – Atom eine, von 2 Pyrimidinbasen (Cytosin, Thymin) oder eine von 2 Purinbasen (Adenin, Guanin) trägt, die gemeinsam eine glycosidische Bindung ausbilden. Wenn der Zucker nun zusätzlich einen Phosphat-Rest am C_5'-Atom trägt, so spricht man von einem Nukleotid (Siehe Fig. 2.2).

Fig. 2.2



Mehrere Nukleotide bilden eine Kette, indem sie eine Phosphor-Diester-Bindung ausbilden, die durch Reaktion zwischen dem sogenannten 3'-OH-Ende (Also die OH-Gruppe am C_3'-Atom des Zuckers) und dem 5'-Phosphat-Rest unter der Abgabe von H_2O entsteht.
Wie entsteht nun die Doppelstrang-Bildung ?
Nach J. Watson und F. Crick bilden die Basen gegenüberliegender Nukleotide Wasserstoff-Brückenbindungen aus, die für die Stabilität der Doppelstrang-Helix verantwortlich sind (Siehe Fig. 2.3). Da Thymin mit Adenin eine Paarung ausbildet, und Cytosin mit Guanin, kann der eine Strang als komplementärer Strang zu dem anderen betrachtet werden. Diese Eigenschaft ist vor allem wichtig bei der Replikation; ein einzelner Strang kann so von der DNA-Polymerase entsprechend der komplementären Basen-Paarungen vervollständigt werden.

Fig. 2.3



Zu Beginn wurde beschrieben, dass die DNA genetische Information trägt. Wie wird diese nutzbar gemacht ? Mit Hilfe von weiteren Enzymen in der Zelle wird die genetische Information in ein Protein umgesetzt. Die hierfür erforderlichen Schritte sind die Transkription und die Translation.

2.2 Transkription

Zunächst erfolgt eine Transkription (Umschreibung) von Teilen der DNA in eine Form, die als mRNA (messenger **R**ibo**n**ucleic **A**cid) bezeichnet wird. Diesen Vorgang übernimmt die RNA-Polymerase. RNA ist der DNA sehr ähnlich, mit den Unterschieden, dass der Zucker an dem C_2'-Atom noch seine OH-Gruppe besitzt. Zudem wird die Base Thymin durch die Base Uracil ersetzt.

2.3 Translation und der genetische Code

Man konnte nachweisen, dass jeweils Codons einer Länge von 3 Basen für eine von 20 Amino-Säuren kodieren. Da es $4^3 = 64$ mögliche Codons gibt, liegt eine Redundanz im genetischen Code vor, die auch als Degeneration bezeichnet wird. Die Umsetzung der Codons zu einer Kette von passenden Aminosäuren wird als Translation bezeichnet und vom Ribosom durchgeführt. Eine unfertige Kette von Aminosäuren bezeichnet man allgemein als Polypeptid, da eine Aminosäure über das N-Terminale Ende mit dem Carboxy-Terminalen Ende einer zweiten Aminosäure unter der Abgabe von Wasser eine Peptidbindung ausbildet.

Die entscheidende Frage stellt sich nun nach dem Startpunkt, an dem das Ribosom die Translation beginnt.

2.4 Die Genauigkeit des Starts

Man fand heraus, dass der Beginn der Translation von einem universellen Start-Codon (AUG) bestimmt wird. Weniger häufig, aber dennoch möglich sind UUG und GUG. Die Termination der Translation wird durch eines, von drei möglichen Stopp-Codons (UAA, UAG und UGA) bestimmt. Der Abschnitt zwischen Start-Codon und Stopp-Codon wird auch als „Open Reading Frame“, kurz ORF bezeichnet und stellt den kodierenden Bereich für ein Protein dar. Das Start-Codon AUG kodiert für die Amino-Säure Methionin, die demnach zu Beginn eines Proteins zu finden ist.
Allerdings ist dieses Start-Codon nicht immer eindeutig, da das Codon AUG auch an anderen Stellen im Genom vorkommen kann und dann keinen Translations-Start darstellt. Daher stellt sich die Frage, wie das Ribosom in der Lage ist, Translations-Starts von einfachen Vorkommen eines Start-Codons zu unterscheiden.. Die Vermutung liegt daher nahe, dass die umliegenden Nukleotide eine wichtige Information dafür liefern. Eine wahre Start-Site heißt im folgenden TIS (Translation Initiation Site). Es gibt verschiedene Ansätze, echte TIS zu finden. Im folgenden werden Support-Vektor-Maschinen aus dem Bereich des kern-basierten, maschinellen Lernens auf dieses Problem angewendet.

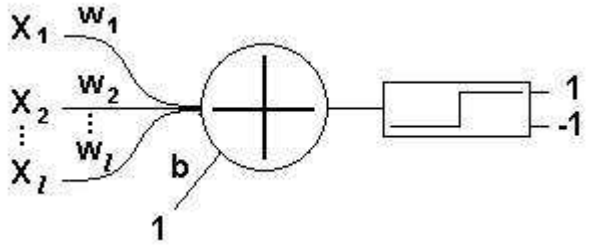
3 Lineare Klassifikation

3.1 Definitionen

Sei $\vec{x}_i \in R^l$ ein *l*-dimensionaler Spaltenvektor.
Ferner sei $y_i \in \{1,-1\}$ das zugehörige Label, welches die Klassenzugehörigkeit angibt.
Dann heißt das Paar $(\vec{x}_i, y_i)$ ein Trainingsbeispiel,
und die Menge $S = \{(\vec{x}_1, y_1),...,(\vec{x}_n, y_n)\}$ eine Lernstichprobe.

3.2 Lineare Klassifikation mit dem Perzeptron

Fig. 3.1

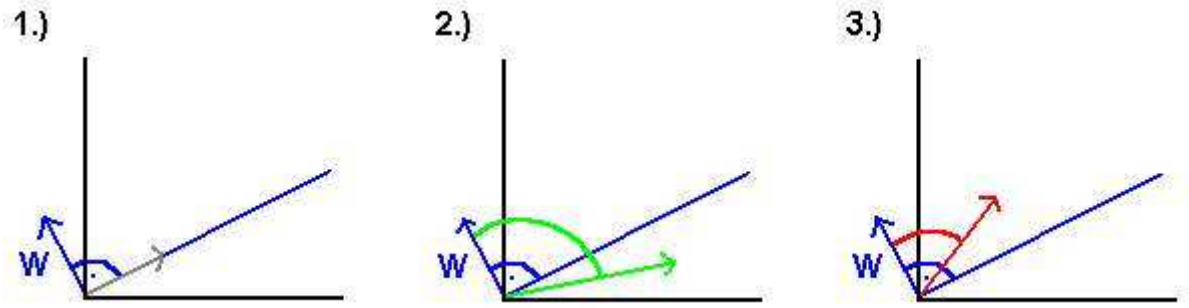


In dem Modell des Perzeptrons (Siehe Fig. 3.1) werden die Eingaben $x_1,...,x_l$ mit den Gewichten $w_1,...,w_l$ multipliziert und anschließend aufsummiert. Es folgt daraus die lineare Diskriminante $f(\vec{x}) = \vec{w} \cdot \vec{x} + b$.
Das Ergebnis wird an eine Aktivierungsfunktion weitergegeben, die in diesem Falle entscheidet, ob die angelegten Werte 1 oder –1 entsprechen. Dieser Abschnitt entspricht dem Klassifikator.
Die Gleichung $f(\vec{x}) = \vec{w} \cdot \vec{x} + b$ definiert, wenn sie gleich 0 gesetzt wird, eine Ebene, die für hohe Dimension (*l* > *3*) Hyperebene genannt wird. Der Vektor $\vec{w}$ heißt dabei Normalenvektor oder, vom allgemeinen Standpunkt des maschinellen Lernens, auch Gewichtsvektor. Die Konstante *b* bestimmt die Verschiebung gegenüber dem Koordinatenursprung.
Da eine Hyperebene in einem *l*-dimensionalen Raum (*l*-1)-dimensional ist, unterteilt sie den Raum in zwei Halbräume. Ziel ist es nun, eine Hyperebene zu finden, die die Trainingsmenge S so teilt, dass alle Trainingsbeispiele, für die $y_i = 1$ gilt, in einem der beiden Halbräume liegen, und demnach, alle Trainingsbeispiele, mit $y_i = -1$, im anderen Halbraum.
Zunächst stellt sich die Frage nach einer Entscheidungsregel, die eine Antwort auf die Frage liefert, auf welcher Seite der Hyperebene, im *l*-dimensionalen Raum, ein beliebiger Punkt $\vec{x}$ liegt.

3.3 Entscheidungsregel: Auf welcher Seite liegt ein Punkt ?

3.3.1 Anschaulich: Skalarprodukt als Winkelmaß

Fig. 3.2



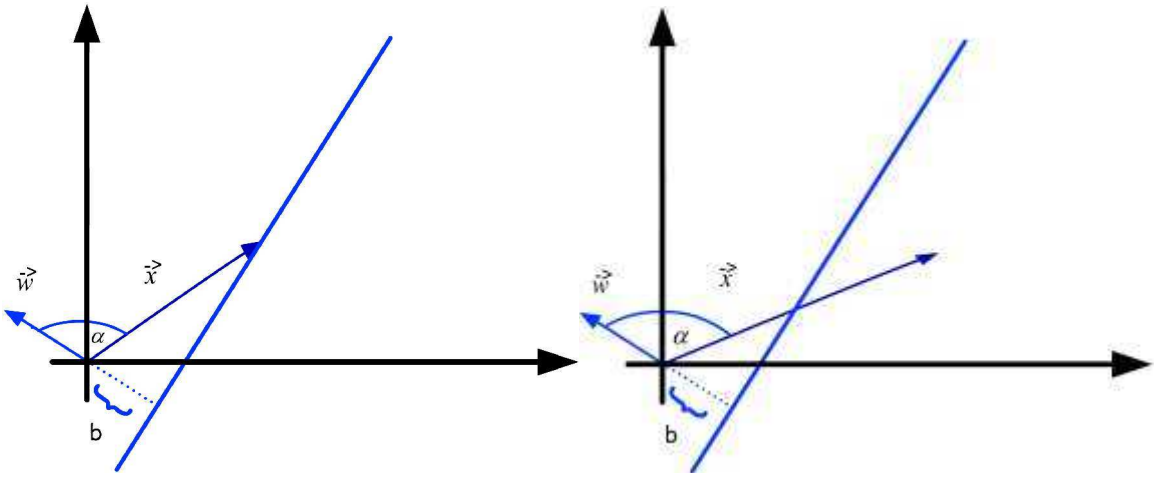
Man betrachte das Skalarprodukt $\frac{\vec{w}}{\|\vec{w}\|} \cdot \vec{x}$. Eine mögliche, geometrische Deutung ist der Winkel zwischen den beiden Vektoren. Stehen beide senkrecht aufeinander, so ist das Skalarprodukt *0*. Zeigen beide Vektoren in die annähernd gleiche Richtung, also besitzt der eine Vektor eine Ausrichtung innerhalb eines 180 Grad-„Blick"-Winkels des anderen, so ist das Skalarprodukt positiv; andernfalls negativ.

Betrachtet man den Fall im R^2 für $\frac{b}{\|\vec{w}\|} = 0$, so besitzt ein Punkt auf der Trenngeraden einen Ortsvektor $\vec{x}$, der senkrecht zum Normalenvektor $\frac{\vec{w}}{\|\vec{w}\|}$ steht. (Siehe Fig. 3.2.1). $\vec{x}$ zeigt also entlang eines gedachten Richtungsvektors der Geraden. Folglich ist das Skalarprodukt zwischen beiden Vektoren *0*.

Schiebt man nun den Punkt $\vec{x}$ von der Geraden herunter, so verändert sich der Winkel zu $\frac{\vec{w}}{\|\vec{w}\|}$. Wie zuvor erklärt, führt ein Winkel zwischen 2 Vektoren der größer ist als 90 Grad zu einem negativen (Siehe Fig. 3.2.2), und bei einem Winkel kleiner als 90 Grad, zu einem positiven Skalarprodukt (Siehe Fig. 3.2.3). Hierbei ist zu bemerken, dass eine skalare Multiplikation von $\frac{\vec{w}}{\|\vec{w}\|}$ mit –1 zu einer Vertauschung der Halbräume führt, da $\vec{w}$ anschließend in die entgegengesetzte Richtung zeigt.

Die Verschiebung durch $\frac{b}{\|\vec{w}\|}$ fügt sich in diese Überlegung ein. Anschaulich wird einer der Halbräume vergrößert, wenn *b* von *0* verschieden ist. Für $b > 0$ wird auf das Skalarprodukt noch ein konstanter Wert addiert, wodurch ein Punkt einen hinreichend großen Winkel zum Normalenvektor haben muss, um die „Verschiebung zu kompensieren", um im „weiter entfernten" Halbraum zu liegen (Siehe Fig.3.3). In diesem Fall ist der Winkel α größer als 90 Grad, wenn der Punkt $\vec{x}$ auf der Ebene liegt. Analog gilt dies auch für negatives *b*.

Fig. 3.3



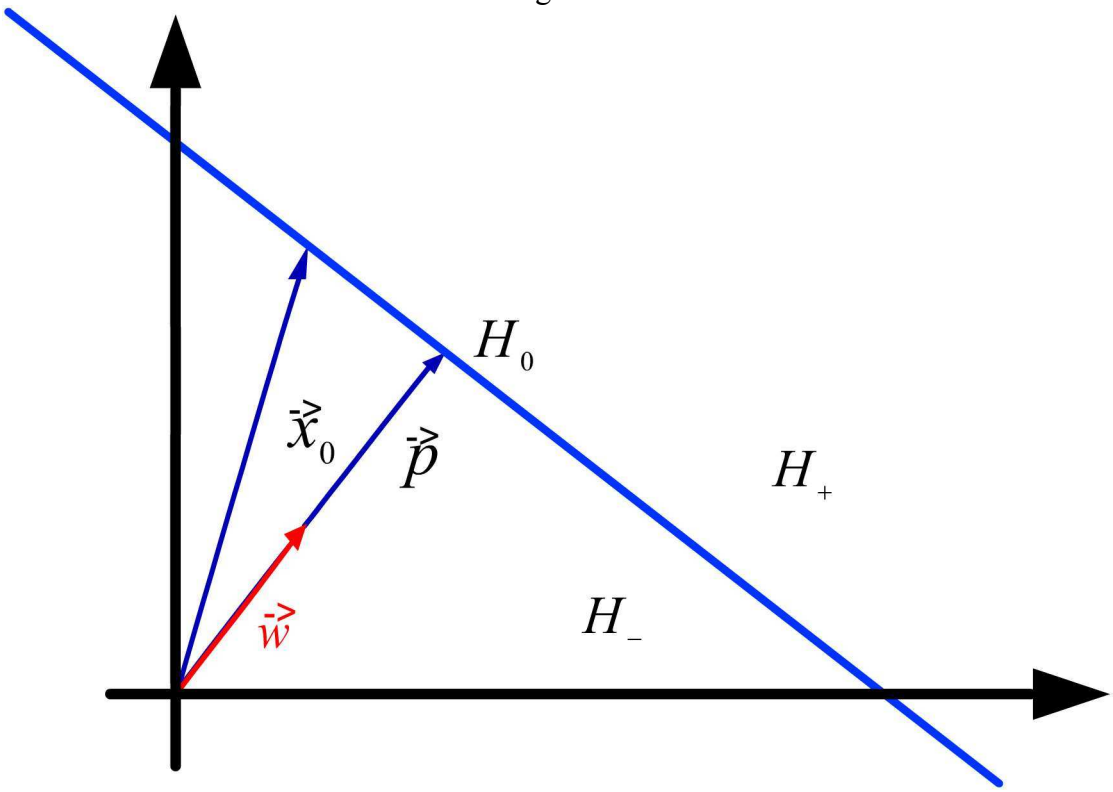
3.3.2 Formaler Ansatz

Sei $H_0 = \{\vec{x} \mid \vec{w} \cdot \vec{x} + b = 0\}$ die Trennebene.

Sei $\|\vec{p}\|$ der Abstand von H_0 zum Koordinatenursprung, also ist $\vec{p} = \lambda \vec{w}$ mit $\lambda \geq 0$ der Fußpunkt der Ebene. Dies kann ohne Beschränkung der Allgemeinheit angenommen werden, da für $\lambda \leq 0$ stets $\vec{w}$ durch skalare Multiplikation mit –1 invertiert werden kann, und so der Fall identisch zu $\lambda \geq 0$ ist.

Für $\vec{x}_0 \in H_0$ gilt $\frac{\vec{w}}{\|\vec{w}\|} \cdot \vec{x}_0 = \|\vec{p}\|$ (Projektion von $\vec{x}_0$ auf Normalenrichtung).

Fig. 3.4



Daraus folgt:

$$\frac{\vec{w}}{\|\vec{w}\|} \cdot \vec{x}_0 = \|\vec{p}\|$$

$$\Leftrightarrow \vec{w} \cdot \vec{x}_0 = \|\vec{p}\| \cdot \|\vec{w}\|$$

$$\Leftrightarrow \vec{w} \cdot \vec{x}_0 - \|\vec{w}\| \cdot \|\vec{p}\| = 0$$

$$\Leftrightarrow \vec{w} \cdot \vec{x}_0 - \lambda \cdot \|\vec{w}\| \cdot \|\vec{w}\| = 0$$

$$\Leftrightarrow \vec{w} \cdot \vec{x}_0 - \lambda \cdot \|\vec{w}\|^2 = 0$$

Dabei entspricht $-\lambda \cdot \|\vec{w}\|^2$ einem $b < 0$ (Abstand vom Koordinatenursprung).
Daraus folgen die beiden Halbräume

$$H_+ = \{\vec{x} \mid \vec{w} \cdot \vec{x} - \lambda \cdot \|\vec{w}\|^2 > 0\}$$

$$H_- = \{\vec{x} \mid \vec{w} \cdot \vec{x} - \lambda \cdot \|\vec{w}\|^2 < 0\}$$

Demnach besitzt $\vec{x}_+ \in H_+$ einen größeren Abstand als $\|\vec{p}\|$ entlang der Richtung von $\vec{w}$.
Analoges gilt für $\vec{x}_- \in H_-$.

Es lässt sich nun der folgende Klassifikator definieren:

$$c(f(\vec{x})) = \begin{cases} 1, falls \quad f(\vec{x}) > 0 \\ -1, sonst \end{cases}$$

Nun stellt sich die Frage nach einem geeigneten Algorithmus, der in der Lage ist, geeignete Werte für $\vec{w}$ und *b* zu finden.

3.4 Algorithmus: Perzeptron Lernregel

Eingabe: Linear separable Lernstichprobe S.
Ausgabe: $(\vec{w}, b)$, so dass alle Trainingsbeispiele durch Diskriminante $f(\vec{x}) = \vec{w} \cdot \vec{x} + b$ richtig klassifiziert werden.
Konstanten:
$\eta > 0$ // Lernschrittweite
Initialisierung:
$\vec{w}_0 = \vec{0}$
$b_0 = 0$
$k = 0$ // Fehlerzähler
$R = \max_i \|\vec{x}_i\|$ // Radius der Kugel die alles umfasst. „Länge des längstes Ortsvektors aller // Trainingspunkte“
Hauptteil:

```
Do
{
        m = k
        For (i = 1 to n)
        {
```

If ($y_i \cdot f_k(\vec{x}_i) \leq 0$) // Fehler ! Diskriminante weicht vom Label ab.
{ // Daher negativer Wert, sonst positiv.
$\vec{w}_{k+1} = \vec{w}_k + \eta \cdot y_i \cdot x_i$ ***(1)***
$b_{k+1} = b_k + \eta \cdot y_i \cdot R^2$ ***(2)***
k++
}
}
}
While ($m \neq k$)

Return ($\vec{w}_k, b_k$)

Zu (1) Die Anpassung des gegenwärtigen Gewichtsvektors $\vec{w}_k$ erfolgt durch Addition des fehlklassifizierten Beispiels, falls es als negativ klassifiziert wurde, jedoch eigentlich positiv ist bzw. durch Subtraktion, falls es als positiv klassifiziert wurde, jedoch eigentlich negativ ist. Die Lernschrittweite η gewichtet dabei die Größe der Modifikation, verlängert oder verkürzt den Vektor durch skalare Multiplikation. Anschaulich lässt sich dieser Vorgang als Linearkombination zwischen fehlklassifiziertem Trainingsbeispiel und dem aktuellen Gewichtsvektor darstellen.

Zu (2) Die Anpassung der gegenwärtigen Verschiebung b_k gegenüber dem Koordinatenursprung erfolgt wie bei **(1)** durch Addition oder Subtraktion des Radius der Kugel, die alles umfasst. Dies ist daher eine sinnvolle Anpassung, da keine Verschiebung der Ebene hinter das am weitesten außerhalb liegende Trainingsbeispiel zu einem fehlerfreien Klassifikationsergebnis führen kann, da in diesem Fall alle Trainingsbeispiele in einem Halbraum liegen würden.

Falls für optimale Werte $\vec{w}_{opt}$ und b_{opt} die Ungleichung $y_i(\vec{w}_{opt} \cdot \vec{x}_i + b_{opt}) \geq \gamma > 0$ für alle $1 \leq i \leq n$ erfüllt ist, so ist γ der kleinste Abstand der $\vec{x}_i$ zur Trennebene.
Nach dem Perzeptron-Konvergenzsatz gilt dann, dass nach Abarbeitung des Algorithmus, also bis alle Trainingsbeispiele richtig klassifiziert werden, höchstens $\left(\frac{2R}{\gamma}\right)^2$ Trainingsfehler auftreten. Dieses Konvergenzkriterium garantiert den Abbruch des Algorithmus und qualifiziert die Perzeptron-Lernregel als Verfahren zur Bestimmung von $\vec{w}$ und *b*.

4 Kernbasierte Verfahren

4.1 Motivation

Schon bei relativ einfachen Klassifikationsaufgaben kann man mit linearen Klassifikatoren nicht die optimale Lösung erreichen.
Angenommen, man habe zwei Klassen, deren Eingabevektoren jeweils normalverteilt seien. Die Mittelwertvektoren der Klassen bezeichne man mit $\vec{m}_+$, bzw. $\vec{m}_-$. Die dazugehörigen Kovarianzmatrizen seien C_+, bzw. C_-, die symmetrisch sind und die Form der elliptischen Verteilung charakterisieren. Die Quantile der Verteilung werden durch Ellipsoide eingeschlossen.
Nun weiß man, dass unter allen möglichen Klassifikatoren der *Bayes- Klassifikator* mit Diskriminante $f_B(\vec{x})$ den erwarteten Fehler minimiert. Für die Normalverteilung gibt es zwei verschieden Fälle:

1. $C_+ = C_- = \mathrm{C}$ (also sind die Kovarianzmatrizen identisch)
   Dann ist $f_B(\vec{x}) = \left[C^{-1}(\vec{m}_+ - \vec{m}_-)\right] \cdot \left(\vec{x} - \frac{\vec{m}_+ + \vec{m}_-}{2}\right)$
   Mann könnte also einen linearen Klassifikator angeben, der die gleichen Ergebnisse, wie der *Bayes- Klassifikator* liefert, mit Diskriminante:
   $\vec{w} = C^{-1}(\vec{m}_+ - \vec{m}_-)$, $b = \frac{1}{2}(\vec{m}_+ + \vec{m}_-)^T C^{-1}(\vec{m}_- - \vec{m}_+)$ und
   $f(\vec{x}) = \vec{w}\vec{x} + b$

2. $C_+ \neq C_-$
   Dann ist die Diskriminante des *Bayes- Klassifikators* gegeben durch:
   $f_B(\vec{x}) = (\vec{x} - \vec{m}_+)^T C_+^{-1}(\vec{x} - \vec{m}_+) - (\vec{x} - \vec{m}_-)^T \vec{C}_-^{-1}(\vec{x} - \vec{m}_-) + \text{konst.}$

Man sieht also, dass im zweiten Fall - beide Klassen jeweils normalverteilt, unterschiedliche Kovarianzmatrizen - schon kein linearer Klassifikator angegeben werden kann, der das gleiche leistet, wie der *Bayes- Klassifikator*. Da also lineare Klassifikatoren schon bei diesen, im Verhältnis zu anderen, recht einfachen Klassifikationsaufgaben nicht optimale Lösungen liefern, ist es sinnvoll, sich komplexere Klassifikations- Funktionen zu überlegen. Zu nennen wären hier zum Beispiel „Neuronale Netze“ mit verdeckten Schichten, auf die in dieser Arbeit aber nicht weiter eingegangen werden soll. Hier geht es vielmehr um kernbasierte Verfahren, die mit Hilfe einer Kernfunktion eine Trennebene in einem höherdimensionalen Feature-Raum bestimmen.

4.2 Kernbasierte Vorhersage

Wie bei der linearen Klassifikation hat man wieder eine Lernstichprobe
S = $\{(\vec{x}_1, y_1),(\vec{x}_2, y_2),...,(\vec{x}_n, y_n)\}$ mit $\vec{x}_i \in R^m$ und $y \in R$. Zudem hat man eine Kernfunktion: $k(\vec{x}, \vec{x}') : X \times X \to R$. Damit lässt sich folgende Vorhersagefunktion konstruieren:

$$f(\vec{x}) = \sum_{i=1}^{n} a_i \cdot k(\vec{x}, \vec{x}_i) + b \,,\; a_i \in R \,.$$

Diese Form der Vorhersagefunktion wird auch als *duale Form* bezeichnet. Die Idee hierbei ist, die Kernfunktion so zu designen, dass sie dem inneren Produkt in einem abstrakten Merkmalsraum F entspricht. Wenn man also eine Abbildung in diesen Raum definiert mit $\phi : X \to F$, dann muss für die Kernfunktion gelten $k(\vec{x}, \vec{x}') = \phi(\vec{x}) \cdot \phi(\vec{x}')$ und weil die Kernfunktion dem inneren Produkt entspricht, gelten auch: $k(\vec{x}, \vec{x}') = k(\vec{x}', \vec{x})$ und $k(\vec{x}, \vec{x}) \geq 0$. Die *Kernmatrix* $K = (k(\vec{x}_i, \vec{x}_j))_{n,n}$ sei positiv semidefinit, was bedeutet, dass alle Eigenwerte größer gleich null sind. Nun kann man auch die *primale Form* der Vorhersagefunktion konstruieren:

$$f(\vec{x}) = \vec{w}_F \cdot \phi(\vec{x}) + b \,,\; mit\; \vec{w}_F = \sum_{i=1}^{n} a_i \phi(\vec{x}_i)$$

Der Grund für die Abbildung in einen höherdimensionalen Merkmalsraum ist, dass man erwartet, dass dort eine lineare Trennung der Daten möglich ist und einer komplizierteren Funktion im Eingaberaum entspricht. Ist die Abbildung allerdings in einen Merkmalsraum, der die gleiche Dimension hat wie der Eingaberaum, so hat man wieder einen linearen Klassifikator.
Dadurch dass die Kernfunktion dem inneren Produkt in F entspricht, kann man jetzt die Trennhyperebene im Merkmalsraum verwenden, ohne die Abbildung explizit ausführen zu müssen.
Wenn die Kernfunktion effizient berechenbar ist, kann man mit ihr unter Umständen nicht linear trennbare Funktionen sehr effizient bestimmen. Dies wird in der Literatur oft als *„Kerneltrick“* bezeichnet.
Am besten lässt sich der *Kerneltrick* anhand des Polynomkerns illustrieren:

4.2.1 Polynomkern

Die Kernfunktion ist nach [9] definiert als

$k_p(\vec{x}, \vec{x}') = (\vec{x} \cdot \vec{x}')^p$, wobei p $\geq$ 1 eine natürliche Zahl ist.

Die Vorhersagefunktion mit dem Polynomkern als Kernfunktion repräsentiert im Fall p = 1 einen linearen Klassifikator. Für p > 1 ist der Merkmalsraum F die Menge aller Monome vom Grad p.
Sei p = 2 und $x \in R^2$. Sei die Abbildung ϕ_2 definiert als

$$\phi_2 : (x_1, x_2) \to (x_1^{\,2}, x_2^{\,2}, x_1 x_2, x_2 x_1)$$

Es ist also

$$\phi_2(\vec{x}) \cdot \phi_2(\vec{y}) = x_1^{\,2} y_1^{\,2} + x_2^{\,2} y_2^{\,2} + 2 x_1 x_2 y_1 y_2 = \langle x \cdot y \rangle^2 = k_2(\vec{x}, \vec{y})^2$$

Man definiert also allgemein ϕ_p als Abbildung von $x \in R^n$ auf den Vektor $\phi_p(\vec{x})$, der geordnet alle möglichen Monome von x mit Grad p enthält. Dann ist:

$$\phi_p(\vec{x}) \cdot \phi_p(\vec{y}) = \sum_{i_1,\ldots,i_p=1}^{n} x_{i1} \cdot \ldots \cdot x_{ip} \cdot y_{i1} \cdot \ldots \cdot y_{ip} = (\sum_{i=1}^{n} x_i \cdot y_i)^p = \langle x \cdot y \rangle^p = k_p(\vec{x}, \vec{y})^p$$

An dieser Kernfunktion lässt sich wieder gut der *Kerneltrick* verdeutlichen. Der Merkmalsraum F kann, gerade bei großen p, sehr hochdimensional werden. Ein explizites Ausrechnen der Abbildung und der Produkte der abgebildeten Vektoren wäre sehr aufwendig. Da man dies aber nur implizit durchführt, hängt die Rechenzeit nur von der Effizienz der Kernfunktion ab. Diese lässt sich in diesem Fall sehr schnell berechnen, selbst für sehr große p.

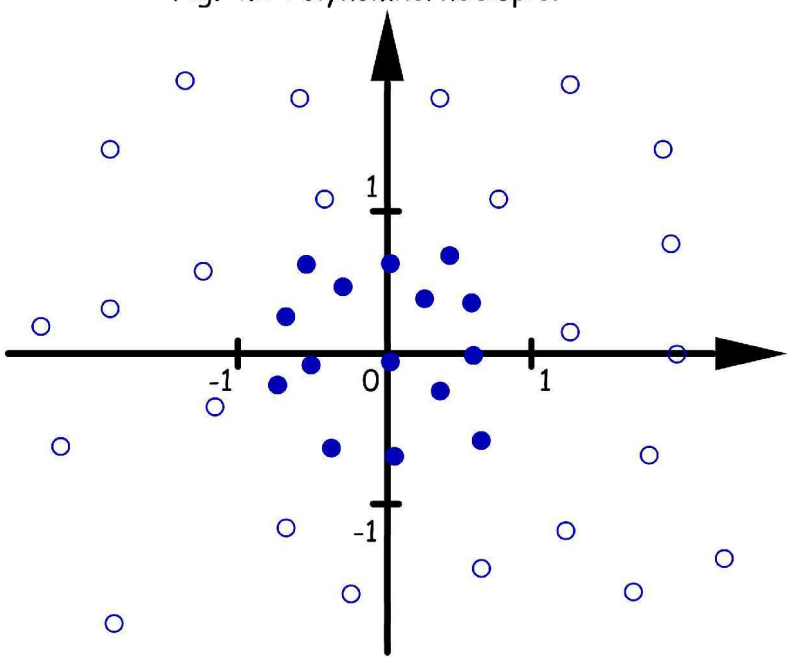
Wenn $x \in \{0,1\}^n$, dann stehen die Monome für alle möglichen Kombinationen von p Positionen. Durch die Kernfunktion tragen nur die Monome zum Wert bei, die in beiden Sequenzen gleich sind. Das heißt zum Beispiel für p = 3, dass das Monom $x_1 x_2 x_3 y_1 y_2 y_3$ nur zu $k_p(\vec{x}, \vec{y})^p$ beiträgt, also ungleich null ist, wenn $x_1 = x_2 = x_3 = y_1 = y_2 = y_3 = 1$.

Sind $\vec{x}$ und $\vec{y}$ also Sequenzen in spärlicher Kodierung (siehe 7.3.1), so symbolisiert das Monom $x_1 x_2 x_3 y_1 y_2 y_3$ einen Gleichheitstest der beiden Sequenzen an den Positionen eins, zwei und drei. Der Wert ist nur ungleich null, wenn die beiden Sequenzen an den drei Positionen identisch sind.

Man kann anhand des Polynomkerns noch einmal zeigen, wie die lineare Trennung im Hyperraum einer komplizierteren Funktion im Eingaberaum entspricht.

Hat man zum Beispiel die Trainingsbeispiele folgendermaßen verteilt,
(ausgefüllte Kreise Negativbeispiele, leere Kreise Positivbeispiele):

Fig. 4.1: Polynomkernbeispiel

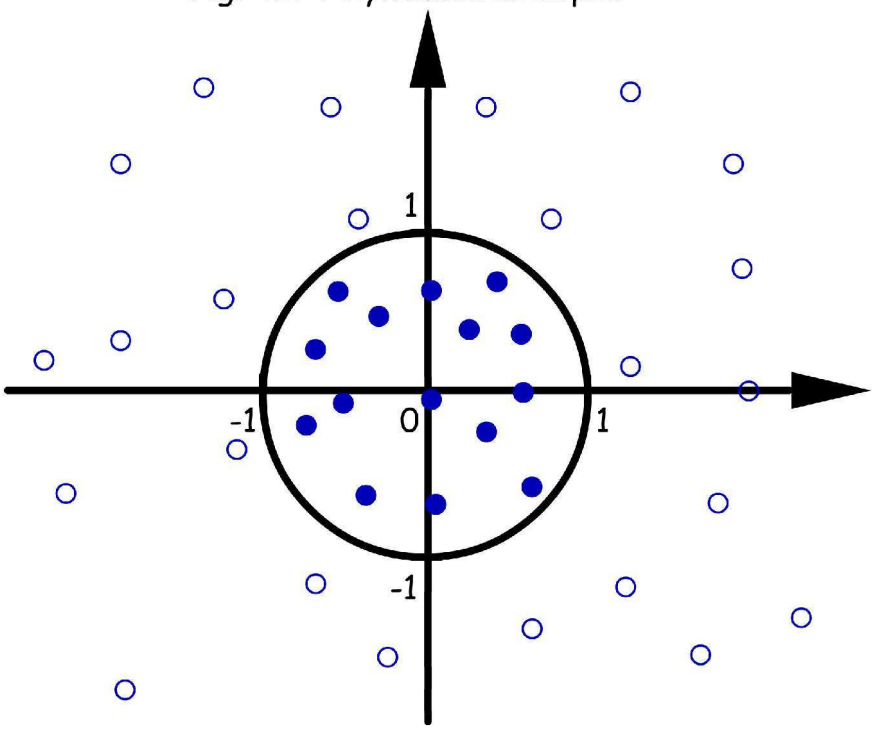


, dann setzt man den Normalenvektor in der primalen Form folgendermaßen:
$\vec{w}_F = (1, 1, 0, 0)^T$ und b = -1. Daraufhin erhält man bei dem Polynomkern vom Grad zwei:

$$f(\vec{x}) = \vec{w}_F \cdot \phi(\vec{x}) + b = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} x_1^{\,2} \\ x_2^{\,2} \\ x_1 x_2 \\ x_2 x_1 \end{pmatrix} - 1 = x_1^{\,2} + x_2^{\,2} - 1$$

Dies entspricht einem Kreis im Eingaberaum, wie in Fig. 4.2 zu sehen ist.

Fig. 4.2: Polynomkernbeispiel



Die lineare Trennung im Featureraum entspricht also dem Einheitskreis im Eingaberaum.

4.2.2 Locality-improved Kern

Bei dem Polynomkern werden Korrelation über die gesamte Sequenz zwischen allen möglichen Positionen betrachtet. Dabei wird die Lage der Positionen zueinander nicht weiter berücksichtigt. Alternativ kann man Korrelationen in lokalen Umgebungen, von zusammenhängenden Positionen betrachten. Dies ist der Ansatz des Locality-improved Kerns [4].
Man betrachte zunächst folgende Funktion:

$$win_p(x, y) = \left(\sum_{j=-l}^{+l} w_j \cdot match_{p+j}(x, y) \right)^{d_1}$$

mit

$$match_k(x,y) = \begin{cases} 1 & falls \quad x[k] = y[k] \\ 0 & sonst \end{cases}$$

Die Funktion win_p bildet einen Score innerhalb eines Fensters der Länge $2l+1$ um die Position *p*. Dabei werden übereinstimmende Basen mit größerer Entfernung zur Position p als weniger relevant betrachtet und entsprechend mit einem kleineren Gewicht w_j gewichtet. Der Grad d_1 induziert genau wie bei dem Polynomkern einen Feature-Space, der aus allen Monomen vom Grad d_1 besteht, die nun noch zusätzlich gewichtet sind.

Der resultierende Kern summiert die Fenster win_p über alle Positionen *p*

$$k(x,y) = \left(\sum_{p=1}^{l} win_p(x,y) \right)^{d_2}$$

Der Grad d_2 zeigte bei Experimenten eher nachteilige Wirkung, weshalb $d_2 = 1$ in allen Experimenten gewählt wurde.

4.2.3 Oligokern

Im Gegensatz zu anderen Kernen, wie zum Beispiel dem Polynomkern, beschränkt sich die Kernfunktion des Oligokerns [6] nicht auf exakte, positionsabhängige Vergleiche, sondern bewertet auch sehr nahe beieinanderliegende Vorkommen positiv. Den Grad dieser Nähe kann man durch einen Parameter der Funktion beeinflussen.

4.2.3.1 Repräsentation der Oligos

Betrachtet werden Vorkommen von K- Meren, die im folgenden auch als Oligos bezeichnet werden. Als Repräsentation dieser Wörter der Länge K könnte man pro Wort eine Kombination von Deltapeaks an den Positionen, an denen das Wort vorkommt, verwenden. Wenn man ein Alphabet A hat, ergibt dies dann für jedes $w \in A^K$ eine Funktion der Form:

$$\hat{f}_w(t) = \sum_{i \in I_w} \delta(t-i)$$

Hierbei ist I_w die Menge aller Positionen, an denen das Wort w vorkommt und t die Positionsvariable der Funktion. Dies ist allerdings eine vollkommen positionsabhängige Repräsentation der Wörter.
Will man einen gewissen Grad an Unsicherheit in das Modell integrieren, so kann man eine Funktion verwenden, die auch neben der Position des Vorkommens einen Wert größer null liefert. Sie glättet also in gewisser Weise die Deltapeaks. Aufgrund seiner Integraleigenschaften bietet sich hier die Gaußfunktion sehr stark an, wie man später noch genauer erkennen wird. Die Glättungsfunktion ist dann:

$$g(t;\mu,\sigma^2) = \exp(-\frac{1}{2\sigma^2}(t-\mu)^2)$$

Die Kombination der Deltapeaks wird jetzt also eine Kombination von Gaußfunktionen, die auch im Folgenden als Repräsentation der Oligovorkommen verwendet wird:

$$f_w(t) = \sum_{i \in I_w} g(t;i,\sigma^2)$$

4.2.3.2 Featuremapping und Kernfunktion

Da der Oligokern im Bereich der TIS- Erkennung eingesetzt wird, wird im Folgenden anstelle von $\vec{x}$ die Notation s verwendet, weil Gensequenzen, bzw. die Oligos der Gensequenzen, betrachtet werden.
Der Featurevektor enthält für jedes verschiedene K- Mer w die Repräsentation des Oligos, also $f_w(t)$. Man erhält somit für das Mapping:

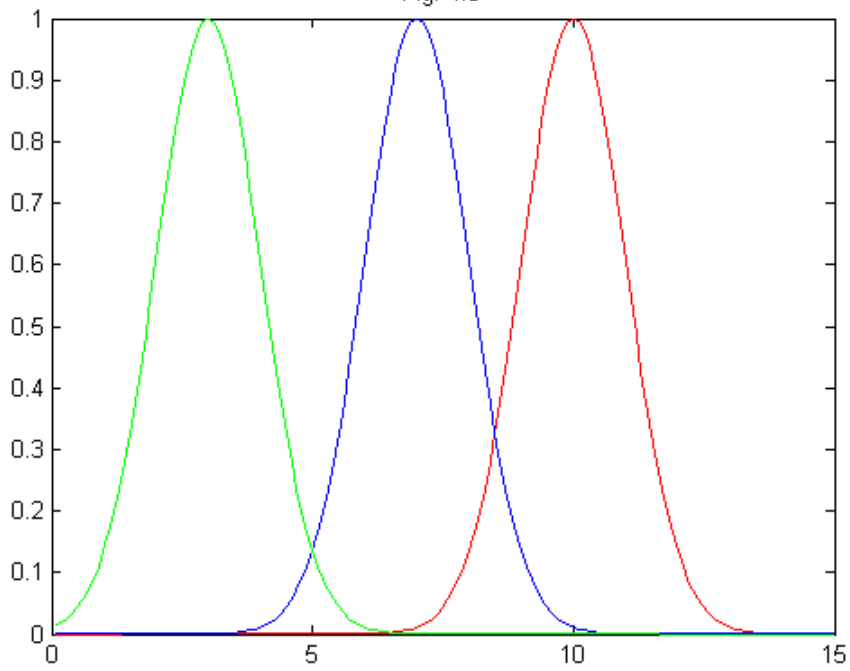
$$\phi(s) = [f_{w_1}(t), f_{w_2}(t), ..., f_{w_d}(t)]^T$$

Hier steht d für die Anzahl der verschiedenen K- Mere. Im Folgenden sei $I_w^{(i)}$ die Menge der Startpositionen des Wortes w der i-ten Sequenz. Dann ist das innere Produkt $\phi(s_1) \cdot \phi(s_2)$ zweier Sequenzen bestimmt durch die Kernfunktion:

$$k(s_1, s_2) = \sum_{w \in A^K} \sum_{i \in I_w^{(1)}} \sum_{j \in I_w^{(2)}} \int g(t;i,\sigma^2) g(t;j,\sigma^2) dt$$

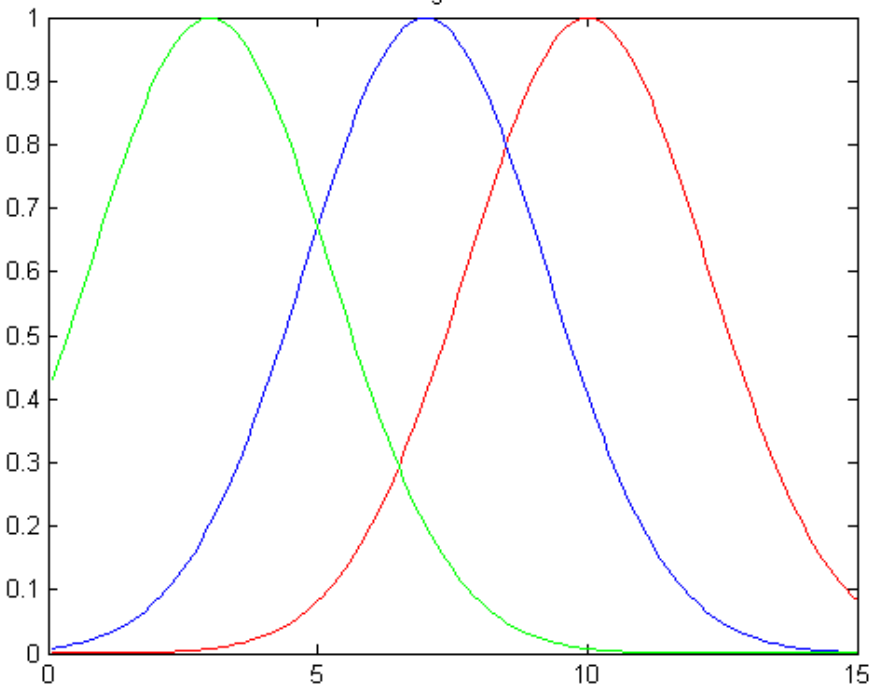
Man kann an dem Produkt der Gaußfunktionen noch einmal das Prinzip des Oligokerns verdeutlichen. Wenn eine der Funktionen an einer Stelle fast null ist, ist auch das Produkt sehr klein. Sind die Positionen des Vorkommens aber nahe genug, so überlagern sich die Gaußfunktionen, was zu einem höheren Wert der Kernfunktion führt. Dies ist in Fig. 4.3 zu sehen:

Fig. 4.3



Betrachtet man die rote und die blaue Kurve, so gibt es Überlagerungen der beiden Funktionen. Die Positionen des Vorkommens der Oligos (hier sieben und zehn) sind dicht genug beieinander. Die rote und die grüne Kurve überlagern sich nicht so gut. In diesem Fall sind die beiden Merkmale nicht ähnlich genug, was hier bedeutet, dass sie zu weit voneinander entfernt sind. Hier spielt aber auch noch die Breite der Kurve eine Rolle. Will man zum Beispiel weniger Positionsabhängigkeit haben, also auch noch die Startposition des Oligos der grünen Sequenz mit der Startposition des Oligos der roten Sequenz positiv bewerten, so kann man das σ der Gaußfunktion vergrößern. Das Ergebnis sieht man in Fig. 4.4.

Fig. 4.4



Man kann die Kurven natürlich auch schmaler gestalten, was im Extremfall wieder eine absolut positionsabhängige Repräsentation erzeugen würde, wie bei den Deltapeaks.

Ruft man sich jetzt noch einmal die Kernfunktion ins Gedächtnis,

$$k(s_1, s_2) = \sum_{w \in A^K} \sum_{i \in I_w^{(1)}} \sum_{j \in I_w^{(2)}} \int g(t; i, \sigma^2) g(t; j, \sigma^2) dt$$

so kommt man zu der Ansicht, dass diese Funktion, zumindest in dieser Form, nicht effizient berechenbar ist. Deshalb betrachtet man das Integral der Funktion noch einmal genauer. Anstelle von $\int g(t; i, \sigma^2) g(t; j, \sigma^2) dt$ kann man auch $\int g(t; 0, \sigma^2) g(t; |i - j|, \sigma^2) dt$ berechnen, da dies einer Verschiebung beider Gaußfunktionen entspricht, wobei ihr Abstand gleich bleibt. Definiert man also u = |i – j|, so bekommt man für das Integral:

$$\begin{aligned}
&\int \exp\left(-\frac{1}{2\sigma^2} t^2\right) \exp\left(-\frac{1}{2\sigma^2}(t-u)^2\right) dt \\
&= \int \exp\left(-\frac{1}{2\sigma^2} t^2 - \frac{1}{2\sigma^2}(t-u)^2\right) dt \\
&= \int \exp\left(-\frac{1}{2\sigma^2}[t^2 + (t-u)^2]\right) dt \\
&= \int \exp\left(-\frac{1}{2\sigma^2}[2t^2 - 2tu + u^2]\right) dt \\
&= \int \exp\left(-\frac{1}{\sigma^2}\left[t^2 - tu + \frac{1}{4}u^2 + \frac{1}{4}u^2\right]\right) dt \\
&= \int \exp\left(-\frac{1}{\sigma^2}\left[t - \frac{1}{2}u\right]^2\right) \cdot \exp\left(-\frac{1}{4\sigma^2}u^2\right) dt \\
&= \int \exp\left(-\frac{1}{\sigma^2}\left[t - \frac{1}{2}u\right]^2\right) dt \cdot \exp\left(-\frac{1}{4\sigma^2}u^2\right)
\end{aligned}$$

Das uneigentliche Integral hat den Wert $\sqrt{2\pi\sigma^2/2} = \sqrt{\pi}\sigma$ ist also eine Konstante. Man kann somit $\exp\left(-\frac{1}{4\sigma^2}u^2\right)$ als Wert der Kernfunktion verwenden. Die Berechnung dieses Wertes geht viel effizienter als die Berechnung von $\int g(t; i, \sigma^2) g(t; j, \sigma^2) dt$. Man kann insbesondere die Werte der Gaußfunktion schon vorher berechnen und tabellieren, um dann, wenn man die Kernfunktionen berechnet, nur noch in einer Tabelle nachsehen zu müssen.

4.2.3.3 Oligokombinationen

Bisher waren die K-Mere beschränkt auf ein bestimmtes K, was bedeutet, dass man zum Beispiel nur Oligos der Länge drei betrachtet. Es mag aber auch sinnvoll sein, eine Kombination von Oligos verschiedener Längen zu betrachten. Hierbei ist allerdings zu

beachten, dass man die einzelnen Kerne normiert addiert, weil ansonsten die kleineren Längen ein stärkeres Gewicht hätten, da hier viel größere Werte entstehen.
Man berechnet also anstelle von

$$k(s_1, s_2) = \sum_{w \in A^K} \sum_{i \in I_w^{(1)}} \sum_{j \in I_w^{(2)}} \exp\left(-\frac{1}{4\sigma^2}(i-j)^2 \right)$$

nun zum Beispiel für eine Kombination der Längen eins bis n,

$$k_{combined_n}(s_1, s_2) = \sum_{i=1}^{n} \frac{k_i(s_1, s_2)}{\sqrt{k_i(s_1, s_1)} \cdot \sqrt{k_i(s_2, s_2)}}$$

wobei das i hier für die Länge der betrachteten K- Mere steht. Der Nenner sorgt für die Normierung der einzelnen Kernlängen, da: $\sqrt{k_i(s_1, s_1)} \cdot \sqrt{k_i(s_2, s_2)} = \|\phi_i(s_1)\| \cdot \|\phi_i(s_2)\|$.
Durch Addition zweier Kernfunktionen, die wie in 4.2 definiert sind, bekommt man wieder eine Kernfunktion.

$$\mathrm{k(x, y)} = k_1(x, y) + k_2(x, y)$$

Dies lässt sich folgendermaßen erklären:
Angenommen, man hat eine endliche Menge an Punkten $\{x_1, ..., x_r\}$. Dann seien K_1 und K_2 die entsprechenden Kernmatrizen zu den Kernfunktionen k_1 und k_2 .Da diese Beiden Matrizen nach Definition positiv semidefinit sind, gilt also für jeden reellen Vektor α der Länge r

$$\alpha' K_1 \alpha + \alpha' K_2 \alpha \geq 0$$

Da

$$\alpha'(K_1 + K_2)\alpha = \alpha' K_1 \alpha + \alpha' K_2 \alpha ,$$

gilt dann auch

$$\alpha'(K_1 + K_2)\alpha \geq 0$$

Somit ist also $K_1 + K_2$ positiv semidefinit und $k_1 + k_2$ eine Kernfunktion.
Der Kern mit den Kombinationen von eins bis sechs wurde von uns auch programmiert und erzielte bessere Resultate als alle anderen Kerne, die nur eine Länge verwendeten.

4.3 Bestimmung der Hyperebene

4.3.1 Linear separable Trainingsbeispiele

In der Praxis wird man selten linear separable Trainingssätze bekommen. Dieser Fall eignet sich allerdings gut, um die Idee zum Finden der Trennebene zu illustrieren. Für den allgemeinen Fall bedarf es dann nur einer kleinen Ergänzung.
Die Trainingsmenge sei S = $\{(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_n, y_n)\}$ mit $\vec{x}_i \in R^m$ und $y \in \{-1, 1\}$.
Angenommen, man hat eine separierende Hyperebene H, die die Positivbeispiele von den Negativbeispielen trennt. Wie man aus dem vorigen Kapitel weiß, gilt für die Punkte, die auf der Hyperebene liegen $\vec{w}_F \cdot \phi(\vec{x}) + b = 0$. Hierbei ist $\vec{w}_F$ die Normale von H, $|b| / \|\vec{w}_F\|$ der

Abstand der Ebene vom Ursprung und $\|\vec{w}_F\|$ die Euklidische Norm von $\vec{w}_F$. Man definiert nun den minimalen Abstand der positiven Beispiele von der Trennebene als d_+ und den der negativen Beispiele als d_-. Ferner sei $d_+ + d_-$ der Rand oder auch *margin* von H.
Der Algorithmus sucht nun nach der separierenden Hyperebene mit größtem Rand.
Angenommen, alle Trainingsbeispiele erfüllen folgende Bedingungen:

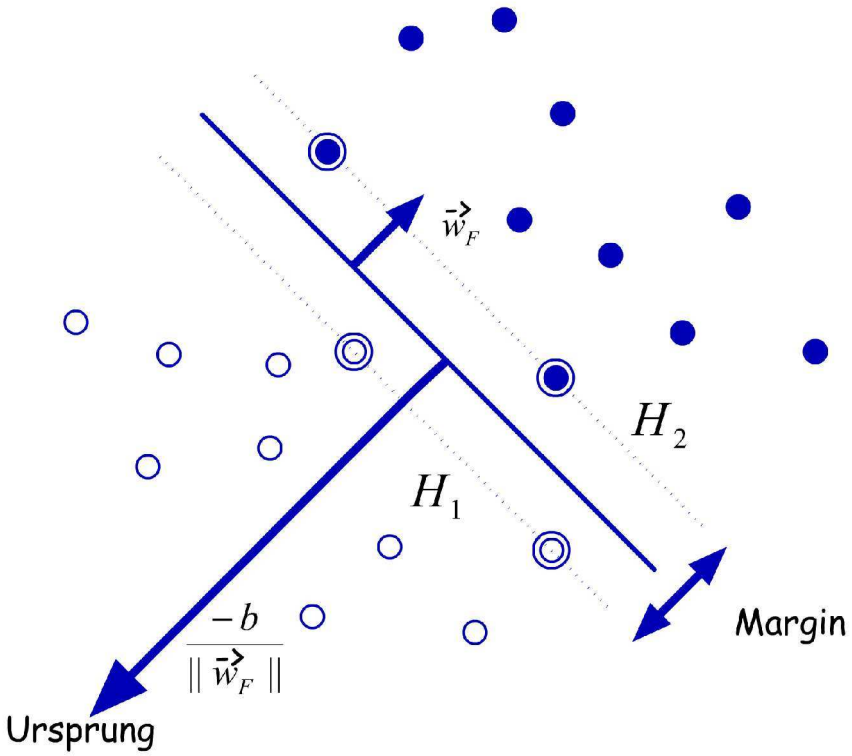
$$\vec{w}_F \cdot \phi(\vec{x}_i) + b \geq 1, \quad \textit{für } y_i = 1$$
$$\vec{w}_F \cdot \phi(\vec{x}_i) + b \leq -1, \textit{für } y_i = -1$$

Hieraus kann man folgende Ungleichung bilden:

$$y_i(\vec{w}_F \cdot \phi(\vec{x}_i) + b) \geq 1 \quad ,i = 1,...,n \qquad (*)$$

Diese Bedingungen kann man auch folgendermaßen interpretieren. Sei H_1 die Hyperebene $\vec{w}_F \cdot \phi(\vec{x}) + b = 1$. Wenn die Bedingungen für alle Positivbeispiele erfüllt sind, so heißt das, dass alle Punkte entweder auf H_1 oder auf der H abgewandten Seite von H_1 liegen. Analoges gilt für die Negativbeispiele und $H_2 : \vec{w}_F \cdot \phi(\vec{x}) + b = -1$. Da der Abstand von H_1 zum Ursprung $|1-b| / \|\vec{w}_F\|$ beträgt und der Abstand von H_2 zum Ursprung $|-1-b| / \|\vec{w}_F\|$, gilt $d_+ = d_- = 1/\|\vec{w}_F\|$. Folglich ist der Rand (*margin*) $2/\|\vec{w}_F\|$. Will man also die separierende Hyperebene mit maximalem Rand bestimmen, so muss man $\|\vec{w}_F\|$ minimieren, wobei (*) weiterhin gelten muss.
Die Punkte, für die $y_i(\vec{w}_F \cdot \phi(\vec{x}_i) + b) = 1$ gilt und deren Entfernen eine andere Trennebene als Lösung zur Folge hätte, werden als *Supportvektoren* bezeichnet. In Fig. 4.5 haben die Supportvektoren einen Extrakreis als Begrenzung.

Fig.4.5: Linear separable Trainingsmenge mit Trennhyperebene



4.3.2 Linear nicht trennbare Trainingsbeispiele

Wenn die Trainingsbeispiele nicht linear trennbar sind, so kann man keine Hyperebene finden, so dass (*) für alle Beispiele gilt. Folglich muss man Fehlklassifikationen zulassen, um die bestmögliche Trennung finden zu können. Hierzu weicht man die Bedingung (*) ein bisschen auf, indem man positive S*lack- Variablen* $\xi_i,\ i = 1,...,l$ einführt, so dass man neue Bedingungen bekommt:

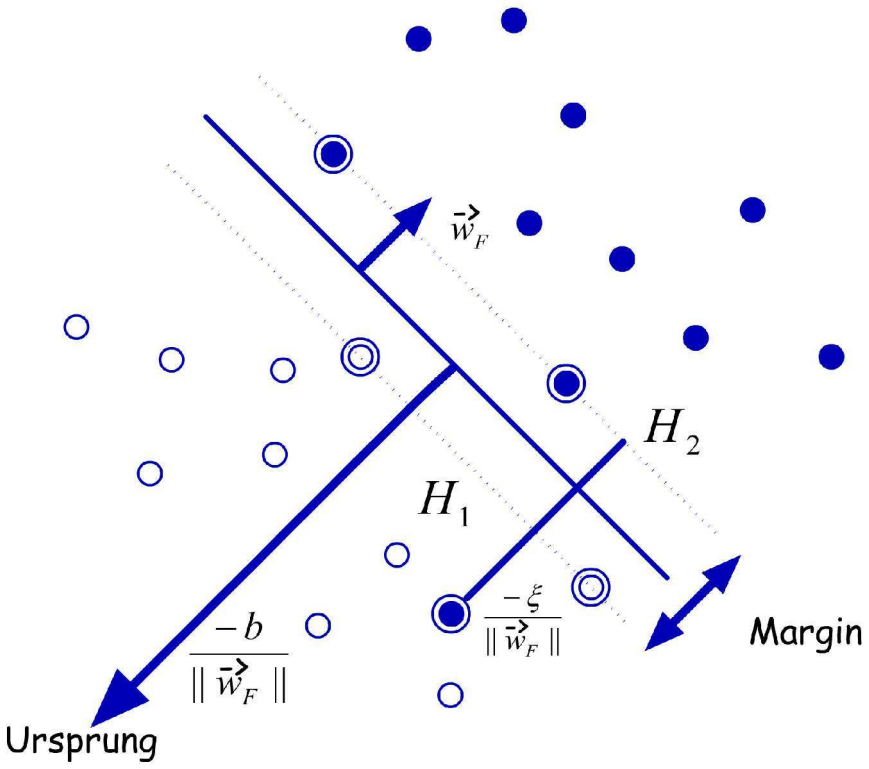
$$\vec{w}_F \cdot \phi(\vec{x}) + b \geq 1 - \xi_i, \quad \textit{für}\ y_i = 1$$
$$\vec{w}_F \cdot \phi(\vec{x}) + b \leq -1 + \xi_i, \textit{für}\ y_i = -1$$
$$\xi_i \geq 0 \quad \forall i.$$

Folglich wird aus (*):

$$y_i(\vec{w}_F \cdot \phi(\vec{x}_i) + b) \geq 1 - \xi_i \quad \forall i$$

Man kann also $\sum_i \xi_i$ als eine obere Schranke für die Anzahl der Fehlklassifikationen ansehen, da die jeweilige Slack- Variable dann größer gleich eins ist, wenn das Trainingsbeispiel auf der anderen Seite der Ebene liegt. Folglich minimiert man nicht mehr nur $\| \vec{w}_F \|^2 / 2$, sondern $\| \vec{w}_F \|^2 / 2 + C(\sum_i \xi_i)$. Das C kann man hier beliebig wählen. Je größer man den Wert setzt, desto stärker werden Fehlklassifikationen bestraft. Man kann somit zum Beispiel die Hyperebene für die Trainingsbeispiele in Fig. 4.6 finden:

Fig.4.6: Nicht linear separable Trainingsmenge mit Trennhyperebene



5. Validierung

5.1 Einführung

Die vorgestellten Kerne weisen eine Menge von Hyperparametern auf, deren Werte nicht nach einer Regel oder willkürlich bestimmt werden können. Hinzu kommt, dass es gänzlich unklar ist, worauf genau in den Sequenzbeispielen geachtet werden muss. Daher ist ein Verfahren erforderlich, das es ermöglicht, anhand einer gegebenen Trainingsmenge sinnvolle Werte für diese Hyperparameter zu bestimmen.

5.2 Hyperparameter der Kerne (Überblick)

Polynomkern:

$p = Degree$

Locality-improved Kern:

$d_1 = Degree$

$l = Window - Half$

Oligo Kern

$\sigma = Sigma$

5.3 Hyperparameter C der libsvm und Slack-Variablen

Zusätzlich zu den Hyperparametern der Kernfunktionen gibt es den Parameter C innerhalb des Moduls libsvm. Der Parameter gewichtet die Fehlklassifikation von einzelnen Trainingsbeispielen während des Trainings.

Bereits im Kapitel „Lineare Klassifikation“ wurde gezeigt, dass für richtig klassifizierte Beispiele der Abstand zur Hyperebene gegeben ist durch:

$$y_i(\vec{w}_{opt} \cdot \vec{x}_i + b_{opt}) \geq \gamma > 0$$

Wobei γ dabei den kürzesten Abstand zu der Hyperebene darstellt. Ausgehend von der primalen Form eines kernbasierten Verfahrens erhält man die Gleichung:

$$y_i(\vec{w} \cdot \Phi(\vec{x}_i) + b) \geq \gamma$$

Nun wird $\gamma = 1$ gewählt und so $\frac{1}{\|\vec{w}\|}$ als Mindestabstand vorgegeben, was zur Folge hat, dass alle Punkte, die näher an der Ebene liegen, als Fehlklassifikation betrachtet werden. Wenn es Beispiele gibt, die sogar auf der falschen Seite der Ebene liegen, so führt die Multiplikation von Diskriminante und Label y_i zu einem negativen Wert, der die Bedingung ebenfalls nicht erfüllt. Mit der Einführung einer Slack-Variablen ξ_i pro Trainingsbeispiel, kann nun in dieses Kriterium eingegriffen werden:

$$y_i(\vec{w} \cdot \Phi(\vec{x}_i) + b) \geq 1 - \xi_i \ , \ \xi_i \geq 0$$

Gerät ein Trainingsbeispiel zu nah an die Hyperebene, so kann das zugehörige ξ_i so gewählt werden, dass die Ungleichung erfüllt ist. Dies ist für $\xi_i > 1$ auch für Beispiele möglich, die auf der falschen Seite der Ebene liegen.
Da für ξ_i nach Definition letztlich eine komplette Fehlklassifikation aller Beispiele möglich ist, sollen große Werte für die Slack-Variablen vermieden werden. Wie zuvor erläutert gilt es folgenden Term für die Maximierung des Margins zu minimieren, bei dem große Werte für die ξ_i bestraft werden:

$$\min_{w,b,\xi} \ \frac{1}{2}\|\vec{w}\|^2 + C \cdot \sum_{i=1}^{n} \xi_i$$

und dabei $y_i(\vec{w} \cdot \Phi(\vec{x}_i) + b) \geq 1 - \xi_i$

$\xi_i \geq 0 \quad i = 1,...,l$

Demnach gewichtet die Konstante C das Auftreten großer Slack-Variablen ξ_i. Daher führt ein großer Wert für C zu einer Hyperebene, die wenige Trainingsbeispiele fehlklassifiziert.

5.4 Verfahren: Kreuz-Validierung

5.4.1 Definitionen

TP: True Positive – Ein richtig klassifiziertes positiv-Beispiel. (y = 1, c(f(x)) = 1)
FP: False Positive – Ein falsch klassifiziertes positiv-Beispiel. (y = 1, c(f(x)) = -1)
TN: True Negative – Ein richtig klassifiziertes negativ-Beispiel (y = -1, c(f(x)) = -1)
FN: False Negative – Ein falsch klassifiziertes negativ-Beispiel (y = -1, c(f(x)) = 1)

Dann heißt

$E = \frac{FP + FN}{TP + FP + TN + FN}$ der Fehler der Vorhersage.

Entsprechend heißt dann

$C = \frac{TP + TN}{TP + FP + TN + FN}$ die Klassifikationsrate der Vorhersage.

Einfacher ausgedrückt werden die falsch klassifizierten Beispiele (Siehe E) bzw. die richtig klassifizierten Beispiele (Siehe C) aufsummiert und durch die Gesamtanzahl an Beispielen geteilt.

Sei S eine gegebene Trainingsmenge.
Seien $H_1, H_2, ..., H_l$ Mengen aus Werten für jeweils einen von insgesamt *l* Hyperparametern.
Dann ist $d = |H_1| \cdot |H_2| \cdot ... \cdot |H_l|$ die Anzahl an Hyperparameter-Kombinationen.
Die Menge M bestehe aus allen Kombinationen der Werte aus $H_1, H_2, ..., H_l$.

Also $M = H_1 \times H_2 \times ... \times H_l$ mit $|M| = d$.

5.4.2 Algorithmus: KreuzValidierung

Eingabe: Trainingsmenge S, Mengen $H_1, H_2, ..., H_l$ von Werten für Hyperparameter.
Ausgabe: Die vermutlich beste Kombination der Parameter.

Ordne die Trainingsbeispiele in S zufällig.

Teile die Trainingsbeispiele aus S in n gleich große Blöcke P_i auf, sodass:
$P_1 \cup P_2 \cup ... \cup P_n = S$ mit $P_i \cap P_j = \varnothing \quad \forall i \neq j$

For (j = 1 to d):
{
Setze die Hyperparameter auf ein zufälliges $m \in M$.

For (i = 1 to n)
{
Trainiere auf $S \setminus P_i$ und führe eine Vorhersage auf der Partition P_i durch.
Speichere die jeweilige Klassifikationsrate in C_i.
}

Setze $\overline{C}_j = \frac{1}{n} \cdot \sum_{i=1}^{n} C_i$

Setze $A_j = m$ // Halte diese Kombination fest für die spätere Auswertung.

$M = M \setminus m$
}

Return A_k mit $k = \arg\max_{1 \leq i \leq d} \overline{C}_i$

Bei n Blöcken spricht man auch von einer n-Fach-Kreuz-Validierung.

5.5 Durchführung

5.5.1 Datenauswahl

Das „Department of Biochemistry and Molecular Biology“ der Universität Miami bietet auf der eigenen Website unter dem Namen „Ecogene” eine Menge verifizierter Genstarts aus E.coli an. Der Nachweis erfolgte durch biochemische Experimente und korrigiert so einige Einträge in der Genebank-Datei U00096.gbk, in der die bisherige Annotation von E.coli zu finden ist. „Ecogene“ bietet daher eine neue Datei im FASTA-Format an, in dem die experimentell nachgewiesen Gene zu finden sind.

5.5.2 Testdaten

Aus den oben beschriebenen Dateien werden alle Gene extrahiert. Dabei werden die experimentell nachgewiesen Gene aus „Ecogene“ als „Verified“ bezeichnet, alle verbleibenden Gene aus der Genebank-Datei U00096.gbk, die nicht verifiziert wurden, als „Remain“.
Obwohl es eigentlich um die RNA und das Ribosom geht, unterscheiden wir diese Fälle nicht, da in der DNA-Sequenz die gleiche Information zu finden ist.
Man muss sich nun auf eine Länge für die TIS-Sites einigen, die zunächst auf 200 gesetzt wird. Dabei beinhaltet die rechte Hälfte dieses Fensters das Start-Codon.

Es ist nun erforderlich, negative Beispiele zu generieren. Zu den „Verified“ Daten wird wie folgt vorgegangen:
Ausgehend von einem wahren Start-Codon, werden in einem Fenster der Länge 60, im gleichen Leserahmen weitere Vorkommen von Start-Codons gesucht. Um jedes dieser „falschen“ Start-Codons wird ein Fenster der Länge 200 gelegt.
Die „Remain“ Daten werden etwas anders generiert, da sie auch als Trainingsmenge verwendet werden sollen. Es hat sich bei SVMs gezeigt, dass es das Beste ist, eine ausgewogene Trainingsmenge, im Hinblick auf positive und negative Beispiele zu haben. Daher wird wie folgt vorgegangen:
Ausgehend von einem wahren Gen-Start wird zunächst Upstream nach einem weiteren Start-Codon im gleichen Lesenrahmen gesucht. Trifft man auf ein Stopp-Codon, so wird die Suche Downstream fortgesetzt. Um ein gefundenes Start-Codon wird dann wieder ein Fenster der Länge 200 gelegt.
Mit diesem Verfahren gelang es, zu jedem positiven Beispiel genau ein Negatives zu generieren.

5.5.3 Validierung der Kerne

Zunächst werden 2/3 der Sequenzen aus den Verified Daten zufällig gezogen und als Trainingsmenge verwendet. Auf dieser Menge wird eine 5-Fach-Kreuz-Validierung durchgeführt. Anschließend werden diese Daten zusammen mit den Parametern der Kreuz-Validierung zum Training verwendet, um die verbleibenden 1/3 der Verified Daten vorherzusagen. Da die Auswahl der Mengen zufällig ist, wird das Experiment 20 mal wiederholt.

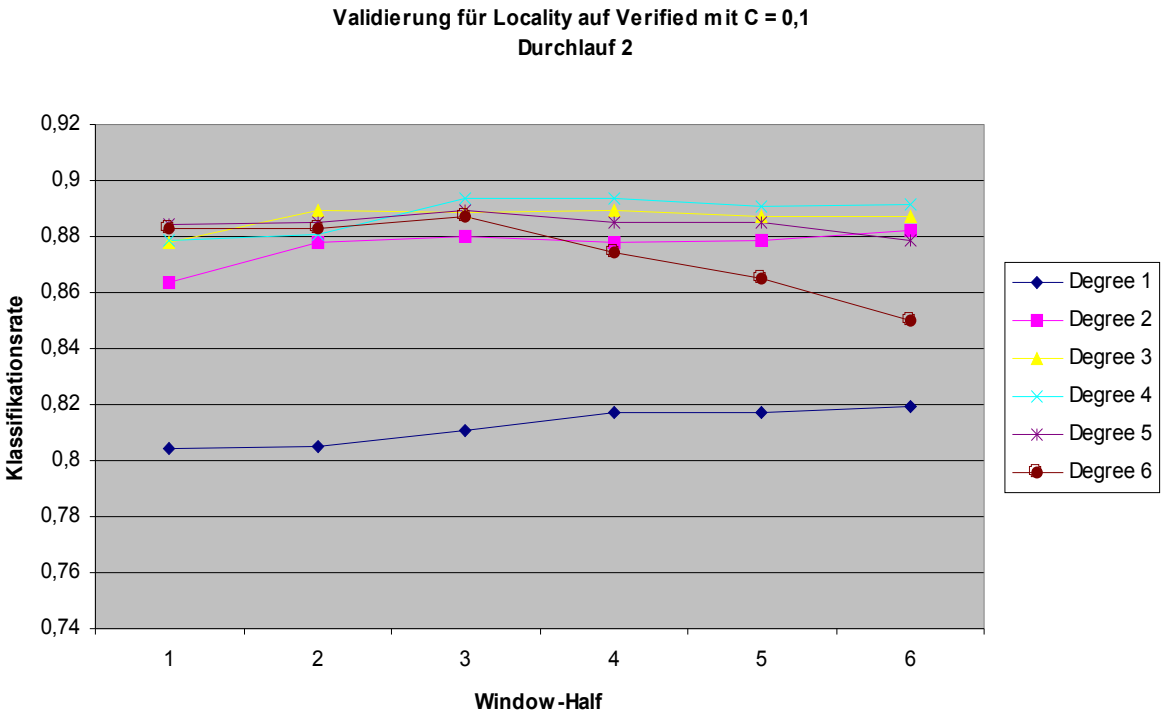
Der Hyperparameter C wurde durch vorherige Tests großflächig abgetastet, zeigte aber nur geringfügigen Einfluss auf die Vorhersage. Der entscheidende Bereich konnte auf das Intervall von 0 bis 1 eingegrenzt werden.

5.5.3.1 Locality-improved Kern

Die grafische Auswertung zeigt exemplarisch das Ergebnis von einer Validierung für den Locality-improved Kern (Siehe Fig. 5.1). Man erkennt, dass für einen Grad von 1 die Performanz deutlich schlechter ist, als für größeren Grad. Dies ist nicht weiter verwunderlich, da aus dem Kapitel 4.2.2 die Funktion des Grades bekannt ist. Bei einem Grad von 1 liegt ein Feature-Space vor, der aus allen Monomen vom Grad 1 besteht, also letztlich identisch mit dem Eingaberaum ist. Der einzige Unterschied besteht in der Gewichtung dieser Monome. Da der Feature-Space unter anderem eine lineare Trennung erleichtern soll, ist bereits intuitiv klar, dass eine Identität (aus der Sicht der induzierten Monome und der Dimension) diese

Anforderungen nicht gut erfüllen kann. Die beste Kombination von Degree und Window-Half in diesem Durchlauf lässt sich der Tabelle (Siehe Tab. 5.1) entnehmen.

Fig. 5.1



Tab. 5.1 Ergebnis der Validierung von Durchlauf 2

Degree	Window-Half	C
4	3	0,1

Im Verlauf der 20 Validierungen ergaben sich je nach der gezogenen Trainingsmenge unterschiedliche Parameter, deren Median in der Tabelle (Siehe Tab. 5.2).

Tab. 5.2 Median über alle Durchläufe

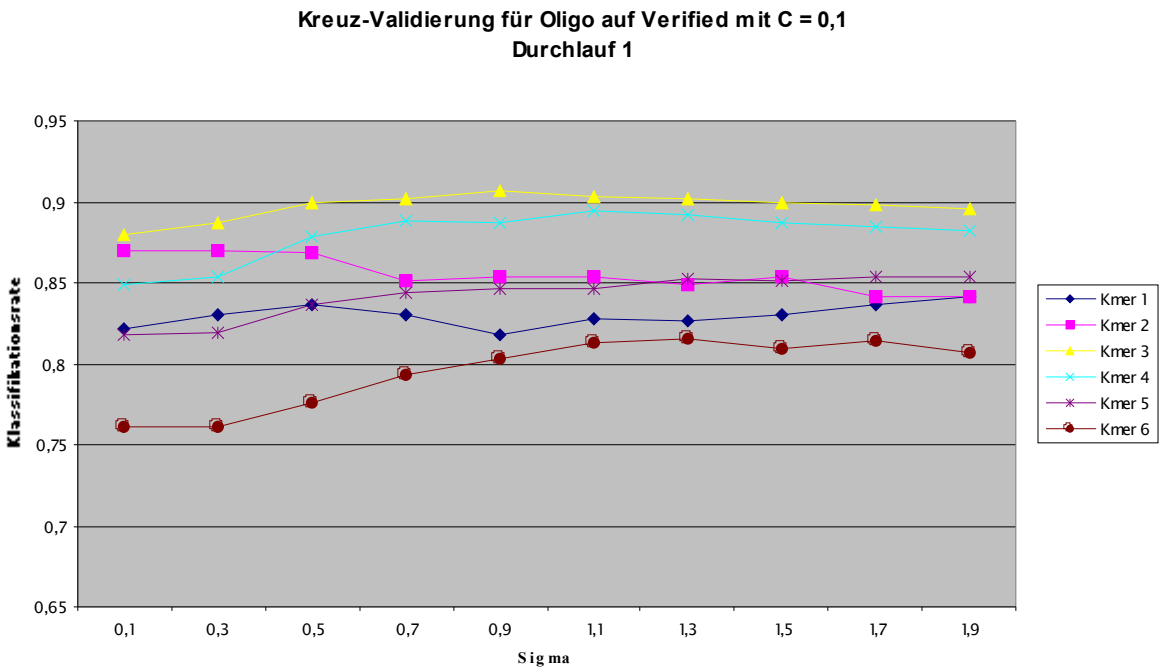
Degree	Window-Half	C
4	3	0,1

5.5.3.2 Oligo Kern

Für den Oligo Kern zeigt sich eine annähernd lineare Abhängigkeit zwischen Kmer-Länge und Sigma. Die Grafik (Siehe Fig. 5.2) zeigt dieses Verhalten exemplarisch für eine Validierung. Für größere Kmer-Längen ergibt sich ein größerer Wert für Sigma, der eine optimale Vorhersage gewährleistet. Dies stimmt mit der Anschauung überein; vergleicht man längere Wörter, so sollten diese positionsunabhängiger sein, als bei einem Vergleich einzelner

Basen, da die Wahrscheinlichkeit, ein langes Wort an der gleichen Stelle in zwei Sequenzen zu finden, sehr gering ist. In der Tabelle (Siehe Tab. 5.3) zeigen sich bei diesem zufällig gewählten Beispiel leichte Abweichungen.

Fig. 5.2



Tab. 5.3 Ergebnis der Validierung von Durchlauf 1

Kmer	Sigma	C
1	0,5	0,1
2	0,3	0,1
3	0,9	0,1
4	1,1	0,1
5	1,7	0,1
6	1,3	0,1

Die Auswertung der Validierung (Siehe Tab 5.4 und Tab 5.5) bestätigt den annährend linearen Zusammenhang im Median und im Mittel. Eine Ausnahme stellt hier die Kmer-Länge 1 dar, die auch für ein größeres Sigma eine (relativ) gute Klassifikationsrate liefert. Die Standardabweichung ist der letzten Tabelle (Siehe Tab 5.6) zu entnehmen.

Tab. 5.4 Median über alle Durchläufe

Kmer	Sigma	C
1	1,5	0,1
2	0,5	0,1
3	0,8	0,1
4	1,1	0,1
5	1,3	0,1
6	1,7	0,1

Tab. 5.5 Mittelwert über alle Durchläufe

Kmer	Sigma	C
1	1,3	0,1
2	0,6	0,1
3	1,0	0,1
4	1,2	0,1
5	1,3	0,1
6	1,6	0,1

Tab 5.6 Standardabweichung über alle Durchläufe

Kmer	Sigma	C
1	0,648	0,1
2	0,352	0,1
3	0,452	0,1
4	0,236	0,1
5	0,224	0,1
6	0,301	0,1

6 Performanzvergleich

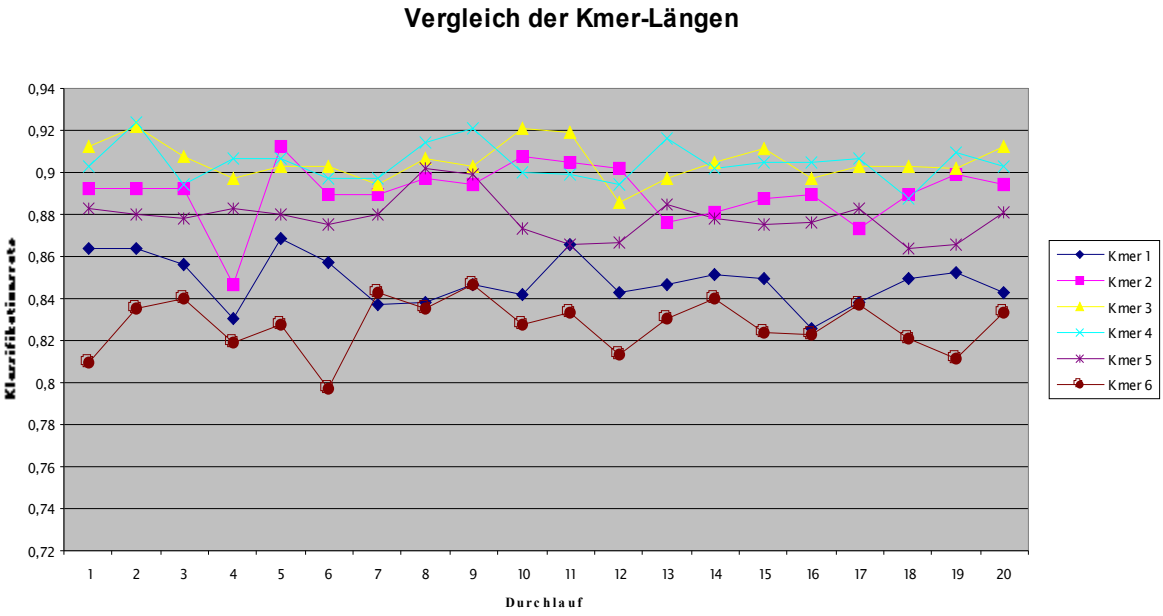
Zum Vergleich der Kerne wurde nach jeder der 20 Validierungen ein Training auf der Trainingsmenge (2/3 der Verified Daten, auf denen validiert wurde) mit den besten Hyperparametern durchgeführt. Anschließend wurde eine Vorhersage durchgeführt auf:

1.) 1/3 der Verified Daten auf denen nicht trainiert wurde
2.) Remain Daten

6.1 Oligo Kern

Betrachtet man die Vorhersage der Verified Daten, so zeigt sich für die verschiedenen Kmer-Längen des Oligo-Kerns (Siehe Fig. 6.1), ein starker Abfall der Vorhersagegenauigkeit zu den Rändern. Die Kmer-Längen 1 und 6 schneiden relativ schlecht ab. Die Kmer-Längen 3 und 4 hingegen zeigen sehr gute Vorhersagegenauigkeit und im Durchschnitt schneidet die Kmer-Länge 3 am Besten ab. Eine mögliche Erklärung hierfür ist, dass die TIS-Sites durch kurze Signale (einzelne Nukleotide) geprägt sind, aber auch längere Wörter eine Rolle spielen. Der Oligo-Kern mit Kmer-Länge 3 stellt daher einen Kompromiss zwischen beiden Extrema dar. Er ist in der Lage, längere Wörter zu erkennen, entfernt sich aber auch nicht allzu sehr von der Fähigkeit, kürzere Signale zu erkennen. Daher kann der Kern als eine Art Kompromiss zwischen allen Kmer-Längen gesehen werden.

Fig. 6.1

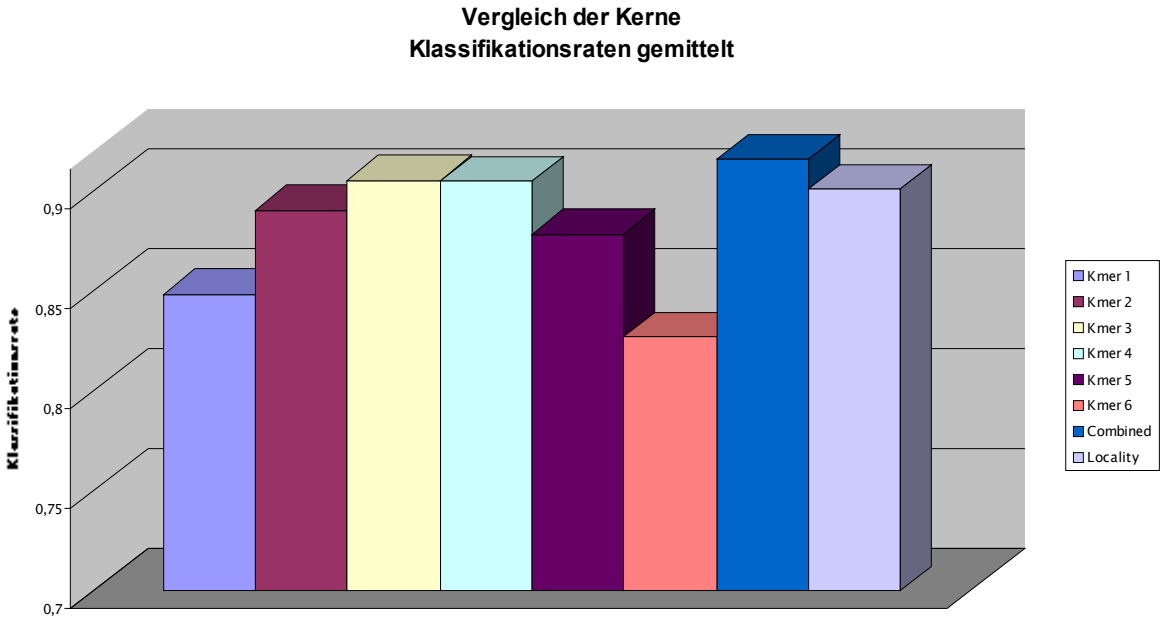


6.2 Vergleich im Mittel

Wie zuvor bereits erwähnt, liegen in den TIS-Sites vermutlich zwei Arten von Informationen vor, die von den Oligo Kernen mit unterschiedlicher Kmer-Länge unterschiedlich gut erkannt werden. Daher stellt dies eine gute Möglichkeit für den Einsatz des zuvor vorgestellten Combined Oligo-Kerns dar, der alle Kmer-Längen miteinander vereinigt. Hierfür werden die am Häufigsten aufgetretenen Sigma-Werte aus den Validierungen für den Combined Oligo-Kern verwendet. Die Grafik der gemittelten Klassifikationsraten für alle Kerne (Siehe Fig. 6.2) zeigt, dass der Combined Oligo Kern um 1,5 Prozentpunkte vor den sehr guten Werten

des Oligo Kerns mit Kmer-Länge 3 und des Locality-improved Kerns liegt. Es bleibt fraglich, ob die Vorhersagegenauigkeit überhaupt noch weiter verbessert werden kann, oder ob in den Sequenzen keine ausreichende Information zu finden ist, um den Vorhersagefehler weiter zu minimieren. Anhand dieser Grafik bestätigt sich erneut die Überlegung des Oligo Kerns mit Kmer-Länge 3 als Kompromiss-Kern, da er auch im Mittel der Beste ist. Da ein Abfall der Vorhersagegenauigkeit zu den Rändern der Kmer-Längen erkennbar ist, wäre es weniger sinnvoll, noch weitere Kmer-Längen in den Combined Oligo Kern einzubeziehen.

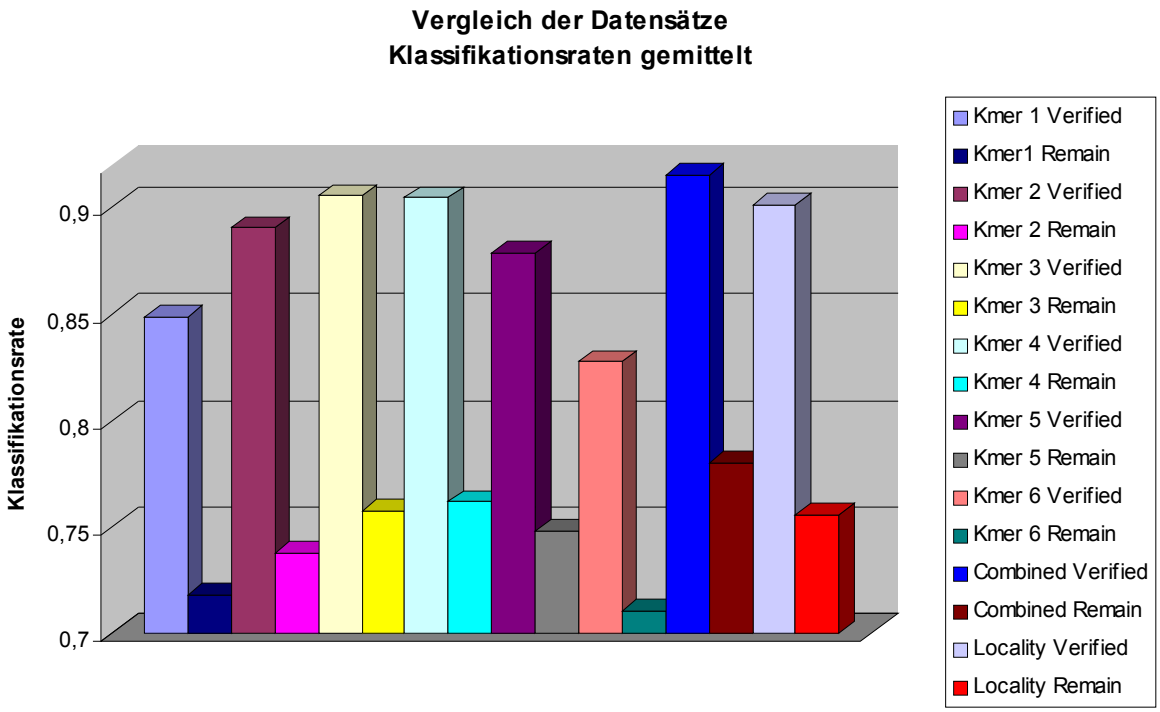
Fig. 6.2



6.3 Vergleich der Datensätze

Zum Vergleich und für eine Qualitätsanalyse, wurde eine Vorhersage der Remain Daten durchgeführt. Wie zuvor erläutert, ist auf die Annotation dieser Daten weniger Verlass, als auf die der verifizierten Daten. Die Grafik (Siehe Fig. 6.3) zeigt einen sehr starken Einbruch von durchschnittlich 15 Prozentpunkten an Vorhersagegenauigkeit. Da frühere Experimente mit diesem Datensatz als Trainingsmenge zu widersprüchlichen Ergebnissen geführt haben, ist die schlechte Performanz, nach einem Training mit sicheren Daten ein Hinweis darauf, dass eine größere Anzahl von Annotationsfehlern existiert. Zumindest lässt sich sicher annehmen, dass die Verteilung der beiden Mengen unterschiedlich ist.

Fig. 6.3

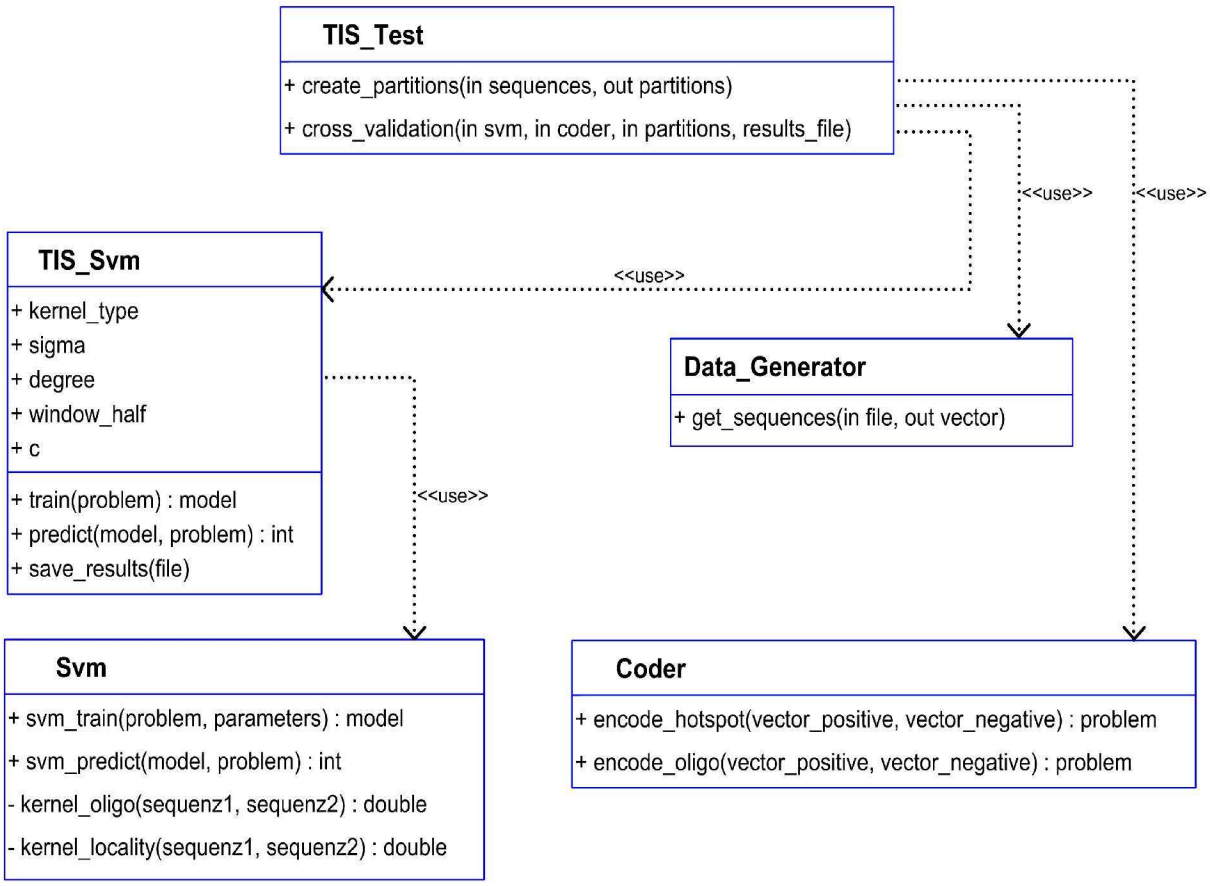

Vergleich der Datensätze
Klassifikationsraten gemittelt
Klassifikationsrate
0,9
0,85
0,8
0,75
0,7
Kmer 1 Verified
Kmer1 Remain
Kmer 2 Verified
Kmer 2 Remain
Kmer 3 Verified
Kmer 3 Remain
Kmer 4 Verified
Kmer 4 Remain
Kmer 5 Verified
Kmer 5 Remain
Kmer 6 Verified
Kmer 6 Remain
Combined Verified
Combined Remain
Locality Verified
Locality Remain

7 Implementationsdetails

7.1 Einführung

Zur Durchführung der Experimente wurde die Bibliothek **libsvm** in der **Version 2.5** verwendet [11]. Diese Bibliothek bietet ein Modul an, mit dem man Support Vektor Klassifikation betreiben kann. Die erforderlichen Funktionen wie zum Beispiel der Localitykern, sowie der Oligokern wurden von uns hinzugefügt. Des weiteren haben wir Klassen geschrieben, die DNA Sequenzen einlesen, für die libsvm kodieren und groß angelegte Tests ermöglichen.
Die vereinfachte Struktur dieser Klassen sieht man in Fig. 7.1 in UML- Notation [12]. Hierbei sind nur die Klassen, Operationen und Attribute aufgeführt, die für das grundlegende Verständnis notwendig sind.

Fig. 7.1: vereinfachtes Klassendiagramm in UML- Notation



Wie man in Fig. 7.1 sieht, benutzt die Klasse TIS_Test die Klasse Data_Generator, um die Sequenzen zu extrahieren. Außerdem verwendet sie die Klasse Coder, um die Sequenzen zu kodieren und die Klasse TIS_Svm um die Lernmaschine zu trainieren und Vorhersagen zu initiieren. Die Klasse TIS_Svm seinerseits verwendet die Klasse Svm der libsvm, um die Berechnungen auszuführen.
Wenn man sich mit der libsvm beschäftigt, so fällt das Augenmerk zuerst auf das verwendete Datenformat, weswegen es hier kurz beschrieben werden soll. Danach gehen wir auf die

Kodierung der DNA- Sequenzen ein, um dann die Implementation des Oligokerns zu erläutern.

7.2 Das Datenformat der libsvm

Sequenzen werden in der libsvm als ein Feld von Knoten dargestellt. Hierbei enthält jeder Knoten den Wert der Sequenz an der Stelle i und den Index i. Dies ist so implementiert, da bei der Kodierung oft viele Nullwerte entstehen und diese so weggelassen werden können. Wenn man also einen Index nicht findet, so heißt dies, dass der Wert der Sequenz an dieser Stelle null ist.

7.3 Kodierung von DNA- Sequenzen

7.3.1 Spärliche Kodierung

Für DNA- Sequenzen wird oft das Alphabet A = {A, C, G, T, N} verwendet. Hierbei steht N für unbekannt und die anderen Buchstaben für die jeweilige Base, wie in den biologischen Grundlagen erläutert.
Jeder Buchstabe der Sequenz wird nun durch einen Vektor mit fünf Einträgen dargestellt. Hierbei sind vier Einträge null und einer eins. Die eins steht an der Stelle, die dem Buchstaben der Sequenz entspricht, also zum Beispiel an der zweiten Stelle bei einem C oder an der fünften Stelle, falls der Buchstabe ein N ist.
Die Vektoren für die einzelnen Buchstaben werden nun hintereinandergeschrieben. Es entsteht ein Vektor, der fünfmal so lang ist, wie die ursprüngliche Sequenz und nur Nullen und Einsen enthält.
An dieser Kodierung sieht man, warum das Datenformat der libsvm sinnvoll ist, da 80 Prozent der Sequenz ansonsten aus Nullen besteht, wenn man diese mitspeichern würde. Die Sequenzen im libsvm- Datenformat haben somit die gleiche Länge, wie die ursprüngliche DNA- Sequenz.

7.3.2 Oligo- Kodierung

7.3.2.1 Allgemeines

Wie in Abschnitt 4.2.3.1 beschrieben, wird für jedes Oligovorkommen seine Position gespeichert. Man braucht also eine Struktur, in der man die Position und die Art des Vorkommens speichern kann. Hierzu bietet sich die Knotenstruktur der libsvm an.
Für die Art des Oligovorkommens speichert man einen Zahlenwert. Dieser wird ähnlich einer Binärdarstellung berechnet. Da man die vier Basen A, C, G und T zur Verfügung hat, ordnet man ihnen also Zahlenwerte zu. A steht für null, C für eins, G für zwei und T für drei.

Wenn man also zum Beispiel CTG repräsentieren möchte, so berechnet sich der Wert folgendermaßen: $C \cdot 4^2 + T \cdot 4^1 + G \cdot 4^0 = 1 \cdot 4^2 + 3 \cdot 4^1 + 2 \cdot 4^0 = 30$. Am Ende wird aus notationstechnischen Gründen zu dem Wert noch eins addiert.

Man speichert nun die Oligos geordnet nach diesem Zahlenwert zusammen mit der Position, an der sie vorkommen, ab. Damit dies möglichst effizient durchgeführt werden kann, wurde folgender Algorithmus von uns entwickelt, der einen binären Suchbaum als Hilfsdatenstruktur verwendet:

7.3.2.2 Der Kodierungsalgorithmus

Funktionsname: encode_Oligo

Eingabe: DNA- Sequenz ’sequenz’, K- Mer Länge ’k_merLaenge’

Ausgabe: Die codierte DNA- Sequenz

Initialisierung: oligoWert = 0
faktor = 1
Erzeuge neuen binären Suchbaum ’baum’
Erzeuge neuen Iterator ’iterator’ für den binären Suchbaum
Erzeuge Knotenfeld ’knoten’

Hauptteil:

```
{

   Für k = k_merLaenge – 1 bis 0
   {
         betrachte sequenz[k]
         {

            falls es ein 'A' ist:    mache nichts

            falls es ein 'C' ist:   oligoWert = oligoWert + faktor

            falls es ein 'G' ist:   oligoWert = oligoWert + faktor * 2

            falls es ein 'T' ist:   oligoWert = oligoWert + faktor * 3

         }
         faktor = faktor * 4
   }

   Füge baum das Paar (oligoWert + 1, 1) hinzu

   faktor = faktor / 4

   Für j = 1 bis sequenz.laenge() - k_merLaenge
   {
         betrachte sequenz[j - 1]
         {

            falls es ein 'A' ist:    mache nichts

            falls es ein 'C' ist:   oligoWert = oligoWert  - (faktor * 1)

            falls es ein 'G' ist:   oligoWert = oligoWert  - (faktor * 2)
```

```
            falls es ein 'T' ist:   oligoWert = oligoWert  - (faktor * 3)

        }

        betrachte sequenz[j + k_merLaenge - 1]
        {

            falls es ein 'A' ist:   oligoWert = oligoWert * 4

            falls es ein 'C' ist:   oligoWert = oligoWert * 4 + 1

            falls es ein 'G' ist:   oligoWert = oligoWert * 4 + 2

            falls es ein 'T' ist:   oligoWert = oligoWert * 4 + 3

        }

        Füge baum das Paar (oligoWert + 1, j + 1) hinzu          // j + 1, da mit 1 begonnen wird

    }

    setze den Iterator auf das kleinste Element des Baumes
    Für j = 0 bis sequenz.laenge() - k_merLaenge
    {
        knoten[j].index = iterator.schluesselwert()
        knoten[j].wert = iterator.wert()
        setze den Iterator ein Element weiter
    }

    gib knoten zurück
}
```

7.3.2.3 Erläuterung des Kodierungsalgorithmus

Die Idee des Algorithmus ist es, linear durch die Sequenz zu gehen und die Zahlenwerte dabei iterativ zu berechnen.
Hierbei wird ausgenutzt, dass man den Nachbarn eines Wertes in konstanter Zeit berechnen kann. Man muss lediglich von dem alten Wert den Summanden mit der größten Base abziehen, den Wert mit vier multiplizieren (entspricht einem Shift nach links) und kann dann den neuen Buchstaben addieren. Dies wird zur Verdeutlichung an einem Beispiel illustriert:

Angenommen man hat die Sequenz GCGTAGGATAGCGATAG und man betrachtet Oligos der Länge 4:

GCGTAGGATAGCGATAG

Der Algorithmus hat den ersten Oligowert berechnet:

$$G \cdot 4^3 + C \cdot 4^2 + G \cdot 4^1 + T \cdot 4^0 =$$

$$2 \cdot 4^3 + 1 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 155$$

Nun muss man den rot markierten Wert abziehen $155 - G \cdot 4^3 = 155 - 2 \cdot 4^3 = 27$, den Wert mit vier multiplizieren (entspricht dem Shift nach links) und den grün markierten Wert addieren (in diesem Fall 0).

Jetzt hat man den zweiten Wert (108), ohne die Positionen zwei, drei und vier noch einmal betrachtet zu haben.

Dieses Verfahren lässt sich iterativ so fortführen.
Wie bereits oben erwähnt, addiert man vor dem Speichern immer eins. Dies wird nur durchgeführt, damit die Nummerierung bei Eins anfängt, also AA...A den Wert Eins bekommt.
Die errechneten Werte werden in einen binären Suchbaum eingefügt, damit man nachher mit einem Iterator die Oligowerte geordnet ausgeben kann. Man spart sich so die Sortierung.

7.4 Implementation des Oligokerns

7.4.1 Der Oligokern- Algorithmus

Funktionsname:	kernel_Oligo	
Eingabe:	kodierte DNA- Sequenz x am Ende der Sequenz ist ein Knoten mit negativem Oligowert, kodierte DNA- Sequenz y am Ende der Sequenz ist ein Knoten mit negativem Oligowert, Tabelle mit Werten der Gaußfunktion gauß_Tabelle	
Ausgabe:	Der Wert k(x, y) des Oligokerns	
Initialisierung:	kern_Wert	= 0
	index1	= 0
	index2	= 0
	zähler	= 0

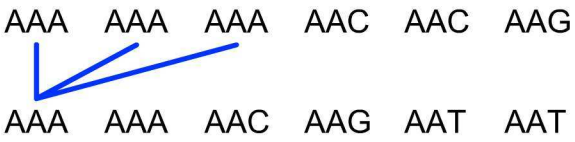
Hauptteil:

```
solange der Oligowert von x an der Stelle index1 größer null ist
    und der Oligowert von y an der Stelle index2 größer null ist

{
     falls x an Stelle index1 den gleichen Oligowert hat wie y an der Stelle index2
     {

        kern_Wert = kern_Wert + gauß_Tabelle[|(x[index1].position - y[index2].position)|]

        falls das nächste Element von x den gleichen Oligowert hat
        {
           inkrementiere index1
           inkrementiere zähler
        }
        andernfalls falls das nächste Element von y den gleichen Oligowert hat
        {
           inkrementiere index2
           vermindere index1 um zähler
           setze zähler auf 0
        }
        andernfalls
        {
           inkrementiere index1
           inkrementiere index2
           setze zähler auf 0
        }

   }
   andernfalls
   {
      falls der Oligowert von x kleiner ist als der von y
      {
         inkrementiere index1
      }
      andernfalls
      {
         inkrementiere index2
      }
   }
}
gib kern_Wert zurück
```

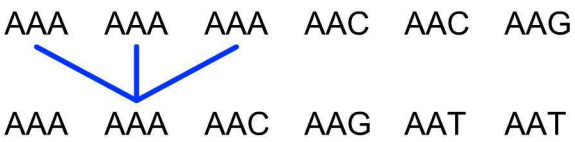
7.4.2 Erläuterung des Oligokern- Algorithmus

Die Sequenzen enthalten die Oligovorkommen aufsteigend sortiert nach ihrem Wert, der wie in Abschnitt 7.3.2 erläutert ist, berechnet wurde. Man muss nun jedes Oligovorkommen mit allen gleichen Oligovorkommen in der anderen Sequenz verrechnen.
Hierzu versucht man immer erst alle Vorkommen in der ersten Sequenz mit dem Vorkommen in der zweiten Sequenz zu verarbeiten. Dabei merkt man sich in einem Zähler, wie viele gleiche Oligos in der ersten Sequenz vorkamen. Man kann also dann den Index der ersten Sequenz wieder um die Anzahl der Vorkommen minus eins zurücksetzen und nun die weiteren Vorkommen der zweiten Sequenz verrechnen. Hierzu ein Beispiel:

Man verrechnet zuerst die Vorkommen der ersten Sequenz mit denen der Zweiten.



Nun verrechnet man die weiteren Vorkommen der zweiten Sequenz mit der Ersten.



Ist der Oligowert der einen Sequenz kleiner als der der anderen Sequenz, so wird der Index um eins vergrößert. Falls eine der Sequenzen einen negativen Oligowert liefert, so ist man am Ende angekommen und kann den Algorithmus beenden.

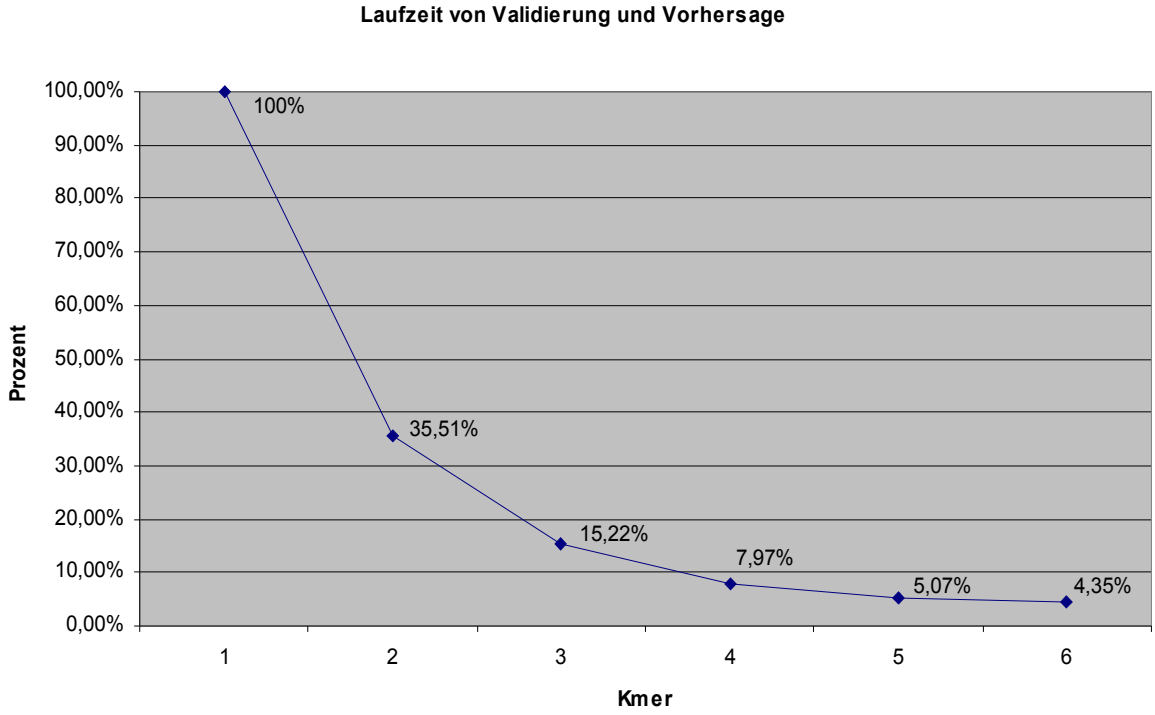
7.4.3 Laufzeit des Oligokern- Algorithmus

Der *worst- case* für die Laufzeit des Algorithmus wäre, wenn alle Oligos den gleichen Wert hätten. Man müsste also jedes Oligo der einen Sequenz mit jedem Oligo der anderen Sequenz verrechnen und hätte somit n mal n Schritte. Die O Abschätzung des Algorithmus ist folglich $O(n^2)$.
Die Frage für den Anwender ist nun, wann man nah an diesen *worst- case* herankommt. Ist die betrachtete K- Mer Länge klein, im Extremfall also eins, gibt es nur sehr wenige mögliche verschiedene Oligowerte. Da man im Fall der K- Mer Länge eins nur vier verschiedene Oligowerte bekommen kann, ist es wahrscheinlich, dass viele Werte der einen Sequenz auch mehrmals in der anderen Sequenz vorkommen. Je größer man die K- Mer Länge wählt, desto mehr verschiedene Oligowerte kann man bekommen. Die Wahrscheinlichkeit, dass es viele gleiche Werte in beiden Sequenz gibt, wird folglich immer kleiner. Es ist daher für den Anwender zu erwarten, dass der Algorithmus umso schneller läuft, je größer die K- Mer Länge gewählt wird.

7.4.4 Praktisch ermittelte Laufzeit des Oligokern- Algorithmus

Die Überlegungen aus 7.4.3 werden auch durch praktische Analysen bestätigt. Das Laufzeit-Diagramm (Siehe Fig. 7.2) zeigt das Verhalten der Laufzeit in Abhängigkeit der Kmer-Länge. Die Angaben beziehen sich auf den Zeitbedarf eines Laufs für ein festes C, ohne Vorhersage der Remain Daten.
Die prozentualen Angaben beziehen sich auf den rechenaufwendigsten Oligo-Kern mit Kmer-Länge 1. So verbraucht zum Beispiel der Oligo-Kern mit Kmer-Länge 2 lediglich 35,51 % der Zeit, die der Oligo-Kern mit Kmer-Länge 1 verbraucht.

Fig. 7.2



Die Tatsache, dass ein so umfassender Lauf bei dem Oligo-Kern mit Kmer-Länge 6 nur einen Bruchteil der Zeit benötigt, wird die Einbindung von noch größeren Kmer-Längen ohne merklichen Zeitverlust ermöglichen. Zudem zeigt sich für den Combined Oligo-Kern, dass der Hauptteil der Zeit für die Berechnungen der Werte für die Kmer-Länge 1 verloren geht.
Der Verbrauch an Zeit dieses Laufs auf einem durchschnittlichen System (Pentium 4 Prozessor mit 384 MB PC-133 SDRAM) kann der Tabelle entnommen werden (Siehe Tab. 7.1).

Tab 7.1

Kmer	Zeitverbrauch
1	02:18
2	00:49
3	00:21
4	00:11
5	00:07
6	00:06

8 Fazit und Ausblick

Unsere ausführlichen Tests zeigen, dass kernbasierte Verfahren sehr gut geeignet sind, um für die TIS- Erkennung eingesetzt zu werden. Ferner sieht man, dass der Combined- Oligo- Kern noch bessere Resultate liefert, als der Locality- Improved- Kern. Es wäre also sinnvoll, ihn in ein Tool zur Genvorhersage einzubinden.

Die Oligo-Grundidee zeigt auch weitere Verbesserungsmöglichkeiten auf. So wäre es möglich, biologisches Wissen über bestimmte Sequenz-Signale in den Kern zu integrieren. Eine praktische Anwendung des Kerns wäre in Form eines Tools für Gen-Erkennungslabore denkbar, die bereits einige Gen-Starts eines Genoms richtig bestimmen konnten, somit also eine gesicherte Trainingsmenge zu Verfügung haben. Das vorhandene Wissen kann dann zur Verbesserung der Vorhersagegenauigkeit für das restliche Genom eingesetzt werden. Dazu wäre es interessant, zunächst Mindestgrößen von Stichproben zu kennen, die eine akzeptable Vorhersage gewährleisten.

Anhang A

A.1 Literaturverzeichnis

[1] Borodovsky,M. and McIninch,J. (1993) "*GENMARK: parallel gene recognition for both DNA strands*". Comput. Chem., 17, 123±133.

[2] Burge,C. and Karlin,S. (1997) "*Prediction of complete gene structures in human genomic DNA*". J. Mol. Biol., 268, 78±94.

[3] Schiex,T., Moisan,A. and Rouzé,P. (2001) "*EuGène: an eukaryotic gene finder that combines several sources of evidence"*. In Gascuel,O. and Sagot,M.-F. (eds), Lecture Notes in Computer Science, Vol. 2006, First International Conference on Biology, Informatics and Mathematics, JOBIM 2000. Springer-Verlag, Germany, pp. 111±125.

[4] Zien, A., Rätsch, G., Mika, S., Schölkopf, B., Lengauer, T. and Müller, K.-R. "*Engineering support vector machine kernels that recognize translation initiation sites*," Bioinformatics, vol. 16, pp. 799--807, 2000

[5] Salzberg,S., Delcher,A., Kasif,S. and White,O. (1998) "*Microbial gene identification using interpolated Markov models. Nucleic Acids*" Res., 26, 544±548.

[6] Meinicke, P., Tech, M., Morgenstern, B. and Merkl, R. "*Oligo kernels for datamining on biological sequences: A case study on prokaryotic translation initiation sites*". BMC Bioinformatics, 2004

[7] Burges, C. J. "*A tutorial on support vector machines for pattern recognition*, " in Data mining and knowledge discovery, U. Fayyad, Ed. Kluwer Academic, 1998, pp. 1--43.

[8] Christianini, N., Shawe- Taylor J. (2000), "*Support vector machines and other kernel-based learning methods*", Camebridge University Press

[9] Schölkopf, B.(1997) "*Support vector learning*", Dissertation, Fachbereich 13-Informatik der technischen Universität Berlin

[10] Smola A., J., Schölkopf, B. "*Learning with kernels*", GMD, Forschungszentrum Informationstechnik, GmbH

[11] Chang, C.-C. und Lin, C.-J. "*LIBSVM: a Library for Support Vector Machines*", http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[12] Jeckle, M., Rupp, C., Hahn, J., Zengler, B., Queins, S.(2004), „*UML 2 glasklar*", Carl Hanser Verlag

A.2 Hilfsmittel

Die benötigten Klassen wurden in C++ unter Linux nach dem Standard ISO/IEC 14882 programmiert. Für die Programmierung und das Testen haben wir unsere privaten Rechner verwendet. Die Grafiken Fig 2.1 und Fig 2.3 wurden der Internetseite http://www.drd.de/helmich/bio/gen/reihe2/karte212.html entnommen.
Die Grafik Fig 2.2 stammt von der Internetseite
http://www.guidobauersachs.de/oc/nukleo.html.
Alle anderen Abbildungen wurden von uns selbst mit Flash, Paint, Excel und Matlab erstellt.

Danksagung

Unser besonderer Dank gilt Herrn Professor Dr. Waack und Herrn Dr. Meinicke für ihre ausführliche Betreuung und die Möglichkeit ein interessantes Thema im Rahmen dieser Bachelorarbeit bearbeiten zu dürfen. Weiterhin wollen wir Dr. Rainer Merkl für seine Unterstützung und Motivation am Anfang des Projekts danken.

Besonders bedanken wir uns auch bei unseren jeweiligen Eltern und unserer jeweiligen Freundin, Christin, bzw. Ina für ihre Unterstützung und Geduld.

Göttingen, den 13. August 2004 Thomas Brodag, Nico Pfeifer