

Protein Similarity Measures as Kernels for Proteochemometrics

Christoph Schwörer

1. November 2009

Inhaltsverzeichnis

1	Einleitung	2
1.1	Versuche	2
2	Methodik	3
2.1	SVM	3
2.2	Verwendete Kernel	4
2.2.1	Tanimoto Kernel	4
2.2.2	Missmatch Kernel	5
2.2.3	Gappy Kernel	5
2.2.4	Substitution Kernel	6
2.2.5	Alignment Kernel	7
2.3	Implementierung der Kernel	7
2.3.1	Tanimoto Kernel	7
2.3.2	Missmatch Kernel	8
2.3.3	Gappy Kernel	9
2.3.4	Substitution Kernel	10
2.3.5	Alignment Kernel	11
2.4	Die Daten	12
2.5	verwendete Programme	13
3	Ergebnisse	14
4	Diskussion	22

Kapitel 1

Einleitung

1.1 Versuche

Kapitel 2

Methodik

2.1 SVM

Eine Support Vektor Machine (SVM) ist ein Verfahren aus dem Bereich der Mustererkennung zur Klassifikation von Objekten. Diese Objekte werden hierbei durch ihre Eigenschaften (features) (zB. Länge, Gewicht oder Sequenzfolge) und ihre Klasse repräsentiert. Bei einer Gegebenen Anzahl d an Eigenschaften können diese als d -dimensionaler Vektor dargestellt werden. Der d -dimensionale Raum der Eigenschaftsvektoren wird Eigenschaftsraum (feature space) χ genannt.

Das Ziel einer SVM ist es nun anhand gegebener Trainingsvektoren deren Klasse bereits bekannt ist unbekannte Objekte korrekt zu klassifizieren. Man hat Beispielsweise 2 Klassen von l Objekten mit einer Anzahl d an Eigenschaften x die als Paare $(\mathbf{x}_i, y_i), i = 1, \dots, n$ mit $\vec{x}_i \in \mathbf{R}^d$ und $y_i \in \{-1, 1\}^n$ gegeben sind. Sind die Datenpunkte im Eigenschaftsraum χ linear separierbar durch eine Hyperebene so ist das Problem trivial (Siehe Abb 2.1A). Sind die Daten aber nicht linear separierbar (Siehe Abb 2.1B) so muss folgendes Optimierungsproblem

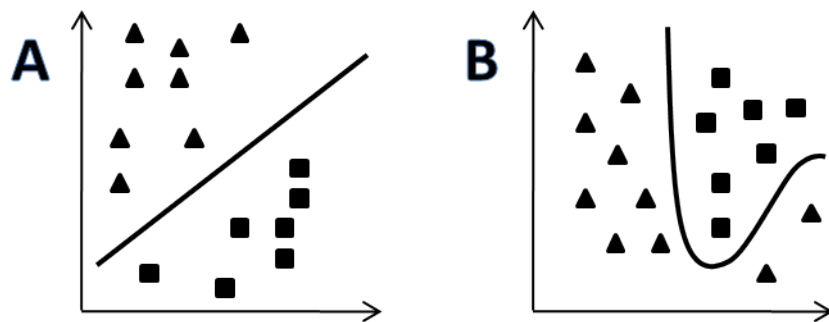


Abbildung 2.1: Beispiel für linear separierbare Daten (A) und nicht linear separierbare Daten (B)

gelöst werden (Boser et al. 1992; Cortes, C. and Vapnik, V., 1995):

$$\begin{aligned} \min_{\mathbf{w}, b, \xi} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^l \xi_i \\ \text{mit der Bedingung} \quad & y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) - 1 - \xi_i, \\ & \xi_i \geq 0. \end{aligned} \quad (2.1)$$

$C > 0$ ist eine positive Konstante die als Strafparameter dient. Die Trainingsvektoren \mathbf{w}_i werden zudem auf einen höher dimensionalen Vektorraum durch die Funktion $\phi : \mathbf{R}^{d_1} \rightarrow \mathbf{R}^{d_2}, \mathbf{w} \rightarrow \phi(\mathbf{w})$; $d_2 > d_1$ abgebildet um in diesem höher dimensionalen Raum eine Hyperebene zu finden die ihn linear separiert. Da die Daten \mathbf{x}_i im Algorithmus zur Lösung des oben genannten Problems nur in der Form eines Skalarproduktes $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ im Raum \mathbf{R}^{d_1} eingehen ist es möglich diese durch ein Skalarprodukt $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$ im Raum \mathbf{R}^{d_2} zu berechnen. Hierzu kann nun eine positiv-semidefinite Kernelfunktion verwendet werden mit:

$$k(x_i, x_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle \quad (2.2)$$

Die in dieser Arbeit verwendeten Kernelfunktionen werden im folgenden Abschnitt erläutert.

2.2 Verwendete Kernel

2.2.1 Tanimoto Kernel

Der einfachste implementierte Kernel ist der Tanimoto Kernel. Hierbei wird ein $|\Sigma|^k$ -dimensionaler Vektorraum über $\{0,1\}$ verwendet. Jede Koordinate wird durch ein mögliches k-mer α indexiert. Tritt das k-mer α auf, so wird der Wert der Koordinate 1 ansonsten bleibt sie 0. Dies führt zu folgender feature map:

$$\Phi_k^{Tanimoto}(x) = (\phi_\alpha(x))_{\alpha \in \Sigma^k} \quad (2.3)$$

wobei

$$\phi_\alpha(x) = \begin{cases} 1, & \text{falls } \alpha \text{ in } x \text{ vorkommt} \\ 0, & \text{sonst} \end{cases} \quad (2.4)$$

Für eine Sequenz x beliebiger Länge wird diese feature map nun über die Summation der einzelnen Vektoren für alle k-mere in x gebildet:

$$\Phi_k^{Tanimoto}(x) = \sum_{k\text{-mere } \alpha \text{ in } x} \Phi_k^{Tanimoto}(\alpha) = X \quad (2.5)$$

Der Tanimoto Koeffizient $T(X, Y)$ für zwei Sequenzen x und y wird nun errechnet durch den Tanimotokoeffizienten von X und Y

$$T(X, Y) = \frac{X \cdot Y}{\|X\|^2 + \|Y\|^2 - X \cdot Y} \quad (2.6)$$

Damit ergibt sich abschließend der Tanimoto Kernel

$$k_k^{Tanimoto}(x, y) = T(X, Y) = T(\Phi_k^{Tanimoto}(x), \Phi_k^{Tanimoto}(y)) \quad (2.7)$$

2.2.2 Mismatch Kernel

Zur Erhöhung des Realitätsgrads und der Annäherung an die natürlichen Gegebenheiten muss es jedoch möglich sein einen gewissen Grad von Ungenauigkeit zu ermöglichen. Ein Kernel der dies erreicht darf also nicht nur abhängig von genauen Vergleichen sein sondern muss ein Maß an *Ähnlichkeit* implementieren. Eine einfache Möglichkeit dieser Implementation ist es mismatches beim Vergleich von k-meren zu erlauben. In Leslie et al. (2003b) wird hierzu ein (k,m)-mismatch Kernel über eine feature map $\Phi_{(k,m)}^{Mismatch}$ realisiert. Für ein gegebenes k-mer $\alpha = \alpha_1\alpha_2\alpha_3\dots\alpha_k$, $\alpha_i \in \Sigma$ wird hierzu ein „mismatch neighborhood“ $N_{k,m}(\alpha)$ definiert. Dies ist die Menge aller k-mer die sich an maximal m Stellen vom k-mer α unterscheiden. Die featur map für α ist demnach definiert als:

$$\Phi_{(k,m)}^{Mismatch}(\alpha) = (\phi_\beta(\alpha))_{\beta \in \Sigma^k} \quad (2.8)$$

wobei

$$\phi_\beta(x) = \begin{cases} 1, & \text{falls } \beta \in N_{(k,m)}(\alpha) \\ 0, & \text{sonst} \end{cases} \quad (2.9)$$

Wie schon beim Tanimoto Kernel wird auch hier wieder für eine Sequenz x beliebiger Länge die map durch Addition der einzelnen feature Vektoren gebildet:

$$\Phi_{(k,m)}^{Mismatch}(x) = \sum_{k\text{-mere } \alpha \in x} \Phi_{(k,m)}^{Mismatch}(\alpha) \quad (2.10)$$

Im Gegensatz zum Tanimoto Kernel werden aber mehrfach vorkommende k-merere auch mehrfach gewertet. Jedes k-mer trägt somit zu allen Werten seines „mismatch neighborhood“ bei. In diesem Fall stellt die β Koordinate von $\Phi_{(k,m)}^{Mismatch}(x)$ also die Anzahl derjenigen k-merere in x dar, die maximal an m Stellen abweichen. Der (k, m) -mismatch Kernel $k_{(k,m)}^{Mismatch}(x, y)$ kann also dargestellt werden als das Skalarprodukt der feature Vektoren von x und y :

$$k_{(k,m)}^{Mismatch}(x, y) = \langle \Phi_{(k,m)}^{Mismatch}(x), \Phi_{(k,m)}^{Mismatch}(y) \rangle \quad (2.11)$$

2.2.3 Gappy Kernel

Alternativ zu mismatches müssen in einem biologisch motivierten Kernel auch Lücken erlaubt werden. Diese Möglichkeit ist mit dem Gappy Kernel gegeben. Wie auch die beiden vorhergehenden Kernel wird für den (g, l) -gappy string kernel (Leslie and Kuang, 2003) der gleiche $|\Sigma|^l$ -dimensionale Merkmalsraum verwendet. In diesem Fall aber basiert die feature map auf „gappy“ matches von g -meren zu l -meren (wobei $g > l$). Hierbei ist $G_{(g,l)}(\alpha)$ die Menge aller l -merere die als Teilfolgen der Länge l (mit $g - l$ Lücken) aus einem gegebenen g -mer $\alpha = \alpha_1\alpha_2\dots\alpha_g$, $\alpha_i \in \Sigma$ durch Konkatenation von Zeichen aus Σ gewonnen werden können. Wobei für alle Stringpositionen α_i, α_j gelten muß: $i < j$ falls $i < j$ in g . Somit ergibt sich die feature map:

$$\Phi_{(g,l)}^{Gappy}(\alpha) = (\phi_\beta(\alpha))_{\beta \in \Sigma^l} \quad (2.12)$$

wobei

$$\phi_\beta(\alpha) = \begin{cases} 1, & \text{falls } \beta \in G_{(g,l)}(\alpha) \\ 0, & \text{sonst} \end{cases} \quad (2.13)$$

Hierbei trägt wieder jede Teilfolge zum Wert aller feature Vektoren bei in denen sie vorkommt. Die feature map wird dann wieder erweitert auf eine beliebig lange Sequenz x indem über alle feature Vektoren aller g -mere in x summiert wird:

$$\Phi_{(g,l)}^{Gappy}(x) = \sum_{g\text{-mere } \alpha \in x} \phi_{(g,l)}^{Gappy}(\alpha) \quad (2.14)$$

Der (g, l) -gappy kernel $k_{(g,l)}^{Gappy}(x)$ wird wiederum erneut definiert als Skalarprodukt der feature Vektoren zweier Sequenzen x und y :

$$k_{(g,l)}^{Gappy}(x, y) = \langle \Phi_{(g,l)}^{Gappy}(x), \Phi_{(g,l)}^{Gappy}(y) \rangle \quad (2.15)$$

2.2.4 Substitution Kernel

Eine erweiterte Variante des mismatch Kernels ist der substitution kernel (Leslie and Kuang, 2003). Anstelle des mismatch neighborhood wird hier jedoch ein similarity neighborhood verwendet. Dieses basiert auf einem probabilistischen Model zum Austausch von Zeichen in den betrachteten Sequenzen. Hierzu werden paarweise Werte $S(a, b)$ verwendet die sich aus geschätzten evolutionären Austauschwahrscheinlichkeiten ableiten (Henikoff and Hennikoff, 1992; Schwartz and Dayhoff, 1978; Altschul et al., 1990). Um solch eine Matrix S zu generieren werden einzelne Blöcke von von Sequenzen homologer Proteine verglichen und ein \log odds-Ratio errechnet:

$$S(i, j) = \left(\frac{1}{\lambda} \right) \log \left(\frac{p_{ij}}{q_i * q_j} \right) \quad (2.16)$$

wobei p_{ij} die Wahrscheinlichkeit darstellt die Aminosäuren i und j in einem Alignment zu finden. q_i und q_j hingegen bezeichnen die Häufigkeiten der Aminosäuren. λ ist der Normalisierungsfaktor. Man definiert nun also den mutation neighborhood $M_{(k,\sigma)}(\alpha)$ eines k -mers $\alpha = a_1 a_2 \dots a_k$ folgendermaßen:

$$M_{(k,\sigma)}(\alpha) = \left\{ \beta = b_1 b_2 \dots b_k \in \Sigma^k : \sum S(a_k, b_k) \right\} \quad (2.17)$$

Dabei lässt sich $\sigma = \sigma(N)$ wählen, so dass $\max_{\alpha \in \Sigma^k} |M_{k,\sigma}(\alpha)| < N$. Dies ermöglicht eine Kontrolle über die Größe des mutation neighborhood. Die substitution feature map definiert sich nun wie folgt:

$$\Phi_{(k,\sigma)}^{Substitution} = \sum_{k\text{-mere } \alpha \in x} (\phi_{\beta}(\alpha)_{\beta \in \Sigma^k}) \quad (2.18)$$

wobei

$$\phi_{\beta}(\alpha) = \begin{cases} 1, & \text{falls } \beta \in M_{(k,\sigma)}(\alpha) \\ 0, & \text{sonst} \end{cases} \quad (2.19)$$

Der substitution kernel $k_{(k,\sigma)}^{Substitution}$ ist damit über das Skalarprodukt definiert als:

$$k_{(k,\sigma)}^{Substitution} = \langle \Phi_{(k,\sigma)}^{Substitution}(x), \Phi_{(k,\sigma)}^{Substitution}(y) \rangle \quad (2.20)$$

2.2.5 Alignment Kernel

Im Gegensatz zu den bisher angeführten Kernen stellt der Alignment Kernel keinen direkten Ähnlichkeitsvergleich zweier Sequenzen dar. Vielmehr wird dieser Kernel durch Faltung mehrere local alignments gebildet da ein einzelnes local alignment keinen gültigen Kernel darstellt. (Vert, Jean-Philippe; Siago, Hiroto; Akutsu, Tatsuya). Im folgenden wird nun ein gültiger local alignment Kernel definiert.

Gegeben sei hierzu eine Substitutionsmatrix S und eine gap penalty Funktion g . Zusätzlich werden drei Kernel auf Basis einer Funktion aus S und g definiert. Der erste Kernel k_0 ist hierbei eine konstante Abbildung von auf 1 welche für diejenigen Sequenzteile verwendet werden die außerhalb des matchings liegen:

$$k_0(x, y) := 1, \forall (x, y) \in \chi^2 \quad (2.21)$$

Der zweite Kernel k_a wird zur Berechnung der Ähnlichkeit von allinierten Symbolen mit Hilfe von S verwendet:

$$k_a^{(\beta)}(x, y) := \begin{cases} 0, & \text{falls } |x| \neq 1 \text{ oder } |y| \neq 1 \\ \exp(\beta S(x, y)), & \text{sonst} \end{cases}, \forall (x, y) \in \chi^2 \quad (2.22)$$

mit $\beta \geq 0$ als Parameter. Der dritte Kernel k_g dient abschließend zur Darstellung der gap penalty:

$$k(\beta)_g(x, y) := \exp[\beta(g(|x|) + g(|y|))] \quad (2.23)$$

wobei $\beta \geq 0$ den gleichen Parameter wie in (2.20) bezeichnet und g eine gültige gap penalty Funktion.

Diese 3 Kernel werden nun durch Faltung zu einem gültigen Kernel k_n zusammengefügt:

$$k_{(n)}^{(\beta)} := k_0 * \left(k_a^{(\beta)} * k_g^{(\beta)} \right)^{(n-1)} * k_a^{(\beta)} * k_0 \quad (2.24)$$

Dieser Kernel definiert nun die Ähnlichkeit von zwei Strings x und y mit einem local alignment der Länge n . Hierbei werden durch den Kernel alle möglichen Dekompositionen von x und y erfasst. Dabei ist k_0 der initiale Teil, $(k_a^{(\beta)} * (k_g^{(\beta)}))$ Die Verteilung aller local alignments von genau n Symbolen die durch $(n-1)$ gaps getrennt werden und das abschließende k_0 der finale Teil.

Um nun bei einem Vergleich zweier Strings alle möglichen lokalen alignments zu berücksichtigen ist es Notwendig über alle n zu summieren so dass sich der endgültige *local alignment kernel* $k_{LA}^{(\beta)}$ ergibt:

$$k_{LA}^{(\beta)} := \sum_{i=0}^{\infty} k^{(i)} \quad (2.25)$$

2.3 Implementierung der Kernel

2.3.1 Tanimoto Kernel

Um eine effiziente Berechnung zu gewährleisten wird eine Trie-Datenstruktur verwendet. Hierbei wird jeweils ein Trie für jede der Sequenzen x und y gebildet. Die Tiefe des Tries entspricht dem Parameter k der verwendeten k-mer. Jeder innere Knoten des Tries hat maximal $|\Sigma|$ (im Fall von Aminosäuren also

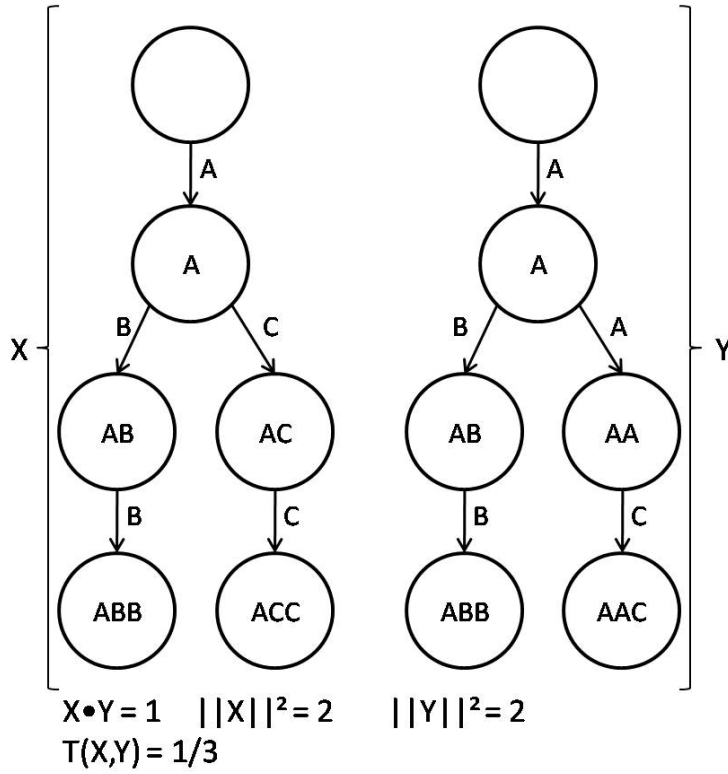


Abbildung 2.2: Beispiel für 2 Tries und deren Tanimoto Koeffizienzen

20) Äste. der Pfad von der Wurzel des Baumes zu einem Blatt entspricht einem in der zugehörigen Sequenz auftretenden k-mer. An jedem inneren Knoten wird beim Aufbau des Tries überprüft ob ein k-mer mit der entsprechenden Zeichenfolge erweitert um ein Symbol aus Σ in der Sequenz existiert. Falls ja wird der Trie um dieses Symbol erweitert. Die Blättern des Tries, also jeweils ein mögliches k-mer, entsprechen hierbei also den Koordinaten der Vektoren des Tanimoto Koeffizienten. Nach dem Aufbau der Tries wird der Tanimoto Koeffizient $T(X, Y)$ (Siehe Formel 2.4) der beiden Sequenzen anhand ihrer Tries errechnet. Mehrfach auftretende k-mere werden bei diesem Trie und so auch im Tanimoto Koeffizien auf eins reduziert.

2.3.2 Mismatch Kernel

Auch der mismatch Kernel nutzt zur Berechnung eine Trie Struktur ähnlich der des Tanimoto Kernels. Im Gegensatz zu dem beim Tanimoto Koeffizienten verwendeten Trie sollen aber bei dem hier verwendeten Trie auch alle mehrfach vorkommenden k-mere gewertet werden. Hierzu wird jedem Knoten (auch den Blättern) eine Liste mit Pointern aller n-mere (wobei n die Tiefe des aktuellen Knotens ist) zugewiesen, die dem Pfad des Knotens von der Wurzel aus entsprechen oder maximal m mismatches aufweisen. Es wird dazu bei jedem erweitern des Tries um ein Symbol am aktuellen Knoten für jedes n-mer der

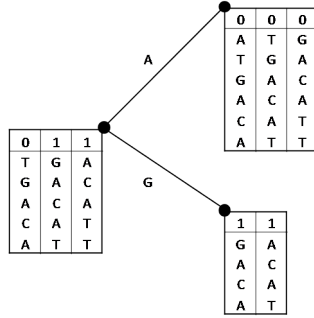


Abbildung 2.3: Teil des (6,1)-mismatch trees für die Sequenz ATGACATT. Es werden l -mere der Länge 6 mit max. 1 mismatch berechnet. Der hier dargestellte Pfad zeigt den Teilbaum aller mit l -mer features mit Präfix AL . In jedem Knoten werden zu allen gültigen Präfixen die Anzahl an mismatches zwischen dem Präfix einer l -mer Instanz und dem Präfix eines features gespeichert sowie ein Pointer zum Startpunkt des jeweiligen Präfixes.

Liste des Vorgängerknotens geschaut ob das $(n+1)$ -mer noch innerhalb der m mismatches liegt. Ist dies der Fall wird es in die Liste übernommen; ist dies nicht der Fall wird es nicht übernommen. Die Liste der $(n+1)$ -mere ist also in jedem Fall eine valide Teilmenge der Liste des Knotens des vorhergehenden n -mers. Erreicht man auf diese Weise ein Blatt so ist die Liste der l -mere also eine gültige Liste aller l -mere die maximal m mismatches zum gesuchten l -mer α aufweisen.

Für eine Sequenz x sind also alle gültigen l -mere die in $N_{(l,m)}(\alpha)$ liegen also äquivalent zu allen l -meren in den Listen des Blattes des Tries mit dem Pfad α . Alle l -mere der Liste tragen somit zur α Koordinate des feature vektors $\Phi(x)$ bei. Man kann also nun einfach die Beiträge aller auftretenden Instanzen summieren und somit den Wert des Kernels aktualisieren:

$$k(x, y) := k(x, y) + n_\alpha(x) * n_\alpha(y) \quad (2.26)$$

wobei $n_\alpha(x)$ und $n_\alpha(y)$ die Anzahl der Instanzen, einschließlich mismatches, eines l -mers α in x und y sind.

2.3.3 Gappy Kernel

Wie bei den beiden vorhergehenden Kernen wird auch für den (g, l) -gappy Kernel ein Baum mit Tiefe l verwendet bei dem jeder innere Knoten $|\Sigma|$ Äste hat. Der Aufbau des Baumes wird durch ein depth first traversal realisiert. Ähnlich dem Mismatch Kernel wird jedem besuchten Knoten eine Liste mit Pointern zu g -meren zugewiesen die dem aktuellen Präfix, mit maximal $g - l$ gaps, entsprechen. Für jedes g -mer wird hierbei zusätzlich ein Pointer zur letzten gültigen Position, also dem ersten Symbol nach letzten gültigen Position des Mutterknotens das der Bezeichnung des Astes entspricht, gespeichert. An der Wurzel sind diese Pointer also alle 0 da noch keine Symbole in den g -meren abgearbeitet wurden.

Bei jedem Schritt in den Baum hinein werden jeweils nur diejenigen g -mere

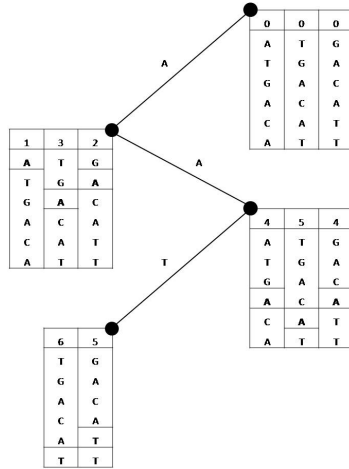


Abbildung 2.4: Teil eines (6,3)-gappy trees für die Sequenz ATGACATT. An jedem Knoten werden die noch gültigen g-mere gespeichert sowie die erste Stelle des Auftretens des aktuellen Symbols nach dem letzten gültigen Symbol. Im gezeigten Bsp wird der Baum für das l-mer *AAT* gezeigt.

weitergegeben bei denen die letzte gültige Position innerhalb des g-mers lag. Wird kein gültiges Symbol, das heißt ein Symbol das der Markierung des Astes entspricht, zwischen dem letzten gültigen und dem Ende des g-mers gefunden so wird dieses verworfen. Findet man jedoch ein gültiges Symbol so wird das g-mer zusammen mit dem neuen Pointer an den Kindknoten weitergegeben. Wird bei einem Schritt kein g-mer weitergegeben so muß dieser Teilbaum nicht weiter bearbeitet werden.

Zum update des Kernelwertes für x und y muß nun nur für jedes feature k -mer die Summe der gültigen Pointer am, dem k -mer entsprechenden Blatt, zum Kernelwert addiert werden.

2.3.4 Substitution Kernel

Die Berechnung des Substitution Kernels ähnelt der des mismatch Kernels. Auch hier wird ein trie der Tiefe l verwendet. An jedem Knoten der Tiefe d wird eine Liste mit Pointern zu allen l -meren gespeichert. Zudem wird noch zu jeder l -mer Instanz α die aktuelle mutation score $\sum_{i=1}^d S(a_i, b_i)$ im Verhältnis zum aktuellen Präfix des Pfades $b_1 b_2 \dots b_d$ gespeichert. Bei jedem Schritt in den Baum hinein wird an der Kante mit Beschriftung b der Tiefe $d+1$ zu jeder l -mer Instanz α der Wert $S(a, b)$ zur aktuellen mutation score addiert und zusammen mit der l -mer Instanz α an den Kindknoten weitergegeben. Wie bei den bisherigen Kernen wird nun der Kernel Wert für ein l -mer erneuert indem die Summe aller gültigen Instanzen (also mit mutation score $< \sigma$) von l -meren im trie an den Blättern zum Kernel Wert für x und y addiert wird.

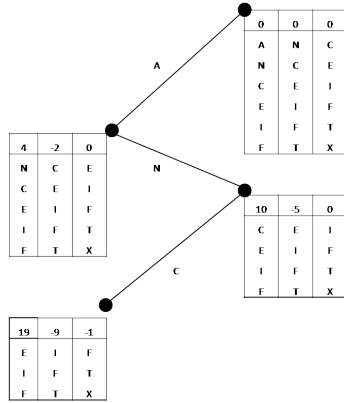


Abbildung 2.5: Beispiel für einen Substitution Kernel Trie der Tiefe 6 für das Präfix ANC. Die Werte für $S(x,y)$ sind aus der BLOSUM62 (Siehe Tabelle 2.1) entnommen.

2.3.5 Alignment Kernel

Da eine naive Berechnung des Kernels nach 2.23 zu einer exponentiellen Zunahme der Komplexität in Abhängigkeit von $|x|$ und $|y|$ führt wurde ein *dynamic programming* Ansatz gewählt. Hierbei handelt sich um eine Abwandlung des klassischen Smith-Waterman Algorithmus für affine gap penalties. Hierzu seien $(x, y) \in \chi^2$ zwei Sequenzen und g eine affine gap penalty Funktion mit

$$g(n) = \begin{cases} 0 & \text{falls } n = 0, \text{ oder} \\ d + e(n - 1) & \text{falls } n \geq 1 \end{cases} \quad (2.27)$$

dann ist der LA Kernel $k_{LA}^\beta(x, y)$ für x und y gleichwertig mit

$$k_{LA}^\beta(x, y) = 1 + X_2(|x|, |y|) + Y_2(|x|, |y|) + M(|x|, |y|) \quad (2.28)$$

wobei $M(i, j)$, $X(i, j)$, $Y(i, j)$, $X_2(i, j)$ und $Y_2(i, j)$ für $0 \leq i \leq |x|$, und $0 \leq j \leq |y|$ rekursiv definiert sind als

$$\begin{cases} M(i, 0) = M(0, j) = 0, \\ X(i, 0) = X(0, j) = 0, \\ Y(i, 0) = Y(0, j) = 0, \\ X_2(i, 0) = Y_2(0, j) = 0, \\ Y_2(i, 0) = Y_2(0, j) = 0, \end{cases} \quad (2.29)$$

und

$$\begin{cases} M(i, j) &= \exp(\beta S(x_i, y_j)) [1 + X(i - 1, j - 1) + Y(i - 1, j - 1) + M(i - 1, j - 1)], \\ X(i, j) &= \exp(\beta d) M(i - 1, j) + \exp(\beta e) X(i - 1, j), \\ Y(i, j) &= \exp(\beta d) [M(i, j - 1) + X(i, j - 1)] + \exp(\beta e) Y(i, j - 1), \\ X_2(i, j) &= M(i - 1, j) + X_2(i - 1, j), \\ Y_2(i, j) &= M(i, j - 1) + X_2(i, j - 1) + Y_2(i, j - 1), \end{cases} \quad (2.30)$$

β ist hierbei der frei wählbare Parameter, S ist die in Tabelle 2.1 gezeigte BLOSUM62 Matrix, d und e sind die gap open und gap extension penalties. Zur

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0	-4
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1	-4
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1	-4
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1	-4
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2	-4
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1	-4
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1	-4
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1	-4
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1	-4
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1	-4
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1	-4
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1	-3	-1	-1	-4
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1	-3	-3	-1	-4
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2	-2	-1	-2	-4
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2	0	0	0	-4
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0	-1	-1	0	-4
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3	-4	-3	-2	-4
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1	-3	-2	-1	-4
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4	-3	-2	-1	-4
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	-3	-3	-2	0	-1	-4	-3	-3	4	1	-1	-4
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1	-4
	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	1

Tabelle 2.1: Die BLOSUM62. Die Angegebenen Werte geben die *log odds* Ratio der Aminosäuren der Zeilen und Spalten an

Normierung der Ergebnisse und um das sogenannte diagonal dominance Problem zu vermeiden wird jeder Kernel wert $k_{LA}^{\beta}(x, y)$ durch folgende Formel aktualisiert

$$\tilde{k}_{LA}^{\beta}(x, y) = \frac{1}{\beta} \ln k_{LA}^{\beta}(x, y) \quad (2.31)$$

2.4 Die Daten

Der in dieser Arbeit verwendete Datensatz bezieht sich auf den in (Rausch, C.; Weber, T.; Kohlbacher, O; Wohleben, W. und Huson, D.; 2005) verwendeten Datensatz an NRPS Proteinen. NRPS steht für *nonribosomalproteinsynthetase* und Bezeichnet eine Familie von Proteinen in Bakterien und Pilzen die durch einzelnes Anfügen von Aminosäuren an eine Kette ein Protein erzeugen. In den meisten Fällen sind dies Peptidantibiotika die spezielle nicht kanonische Aminosäuren verwenden. Die NRPS Proteine sind nach der Art ihres spezifischen Substrates in 8 Klassen aufgeteilt:

- aliphatische Kettenenden mit Wasserstoffbrücken Donor
- apolare, aliphatische Seitenketten

- aromatische Seitenketten
- lange positiv geladene Seitenketten
- aliphate oder phenyle mit OH Gruppen
- polare ungeladene (Cys)
- zyklische Aliphate
- hydroxy benzoe Säuren und derivate

Der vollständige Datensatz enthält 339 Sequenzen.

2.5 verwendete Programme

Zu Analyse der berechneten Kerneldaten aus den vorgestellten Kernen wurde das Programm LibSVM (frei erhältlich unter: <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>) verwendet. Insbesondere der Programmteil svm-train der es ermöglicht sowohl vorberechnete Kernel zu verwenden als auch eine direkte n -fache Kreuzvalidierung ermöglicht. Hierzu müssen die Parameter $-t\ 4$ und $-v\ [n]$ verwendet werden. Bei der Auswertung der Kernel wurde im weiteren Verlauf der Parameter n immer mit 5 gewählt. Als Ausgabe erfolgt das Ergebnis der Kreuzvalidierung in % sowie eine Datei die das Model der SVM zur weiteren Verwendung innerhalb des Programms enthält.

Kapitel 3

Ergebnisse

Tanimoto Kernel

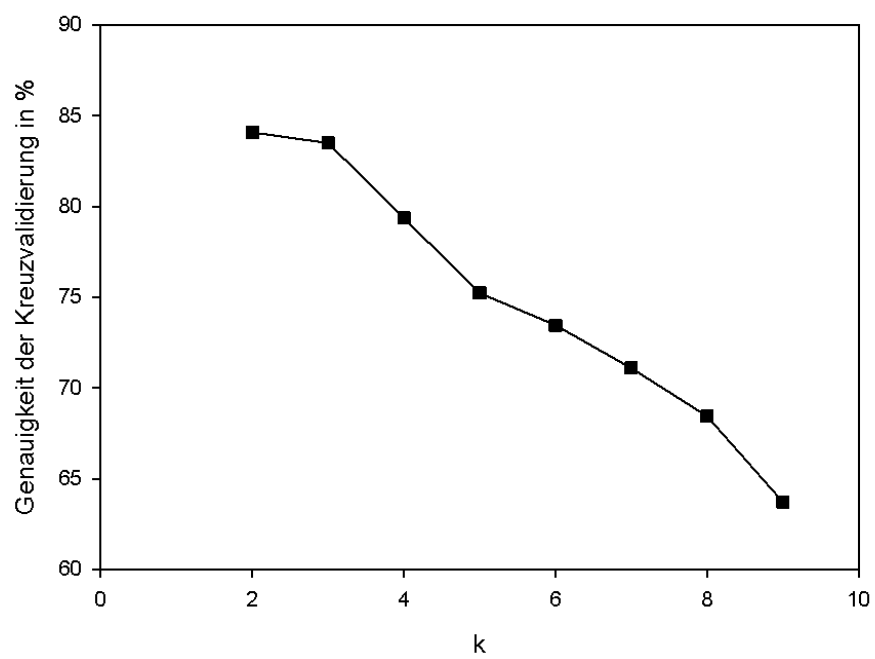


Abbildung 3.1: blabla

Missmatch Kernel

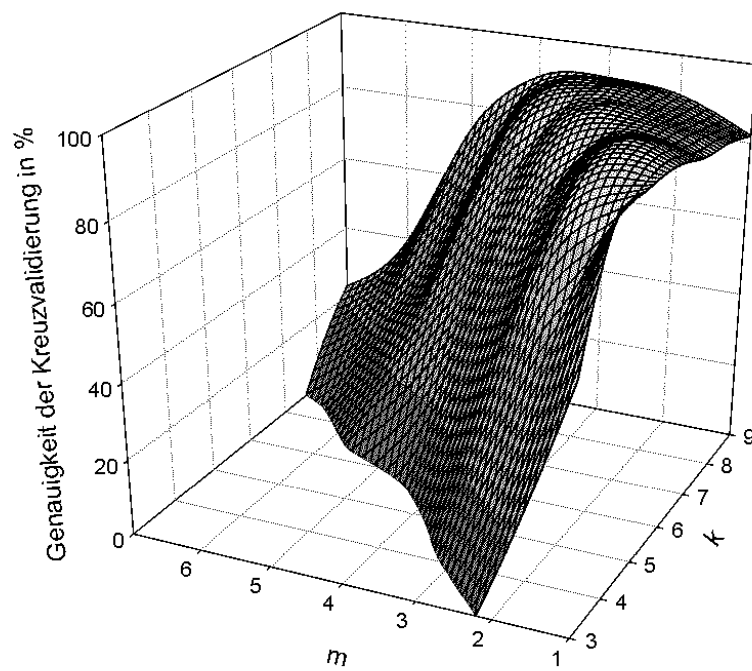


Abbildung 3.2: blabla

Missmatch Kernel

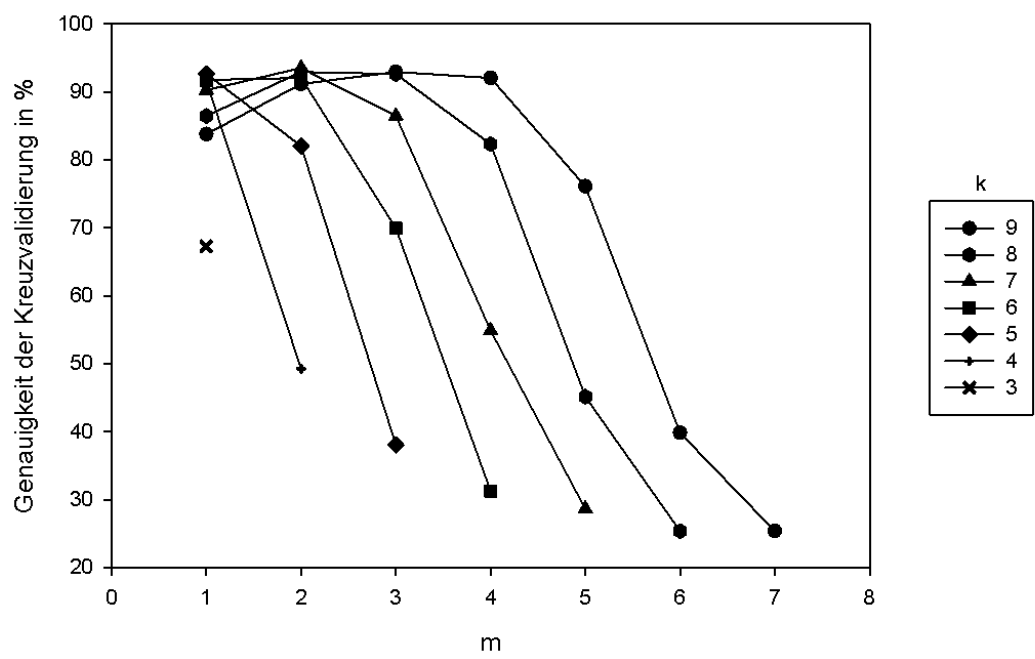


Abbildung 3.3: blabla

Gappy Kernel

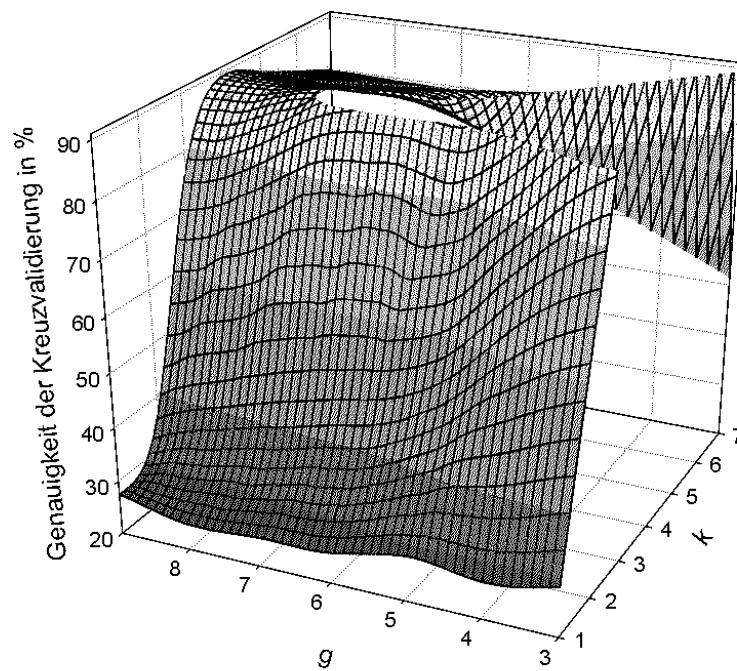


Abbildung 3.4: blabla

Gappy Kernel

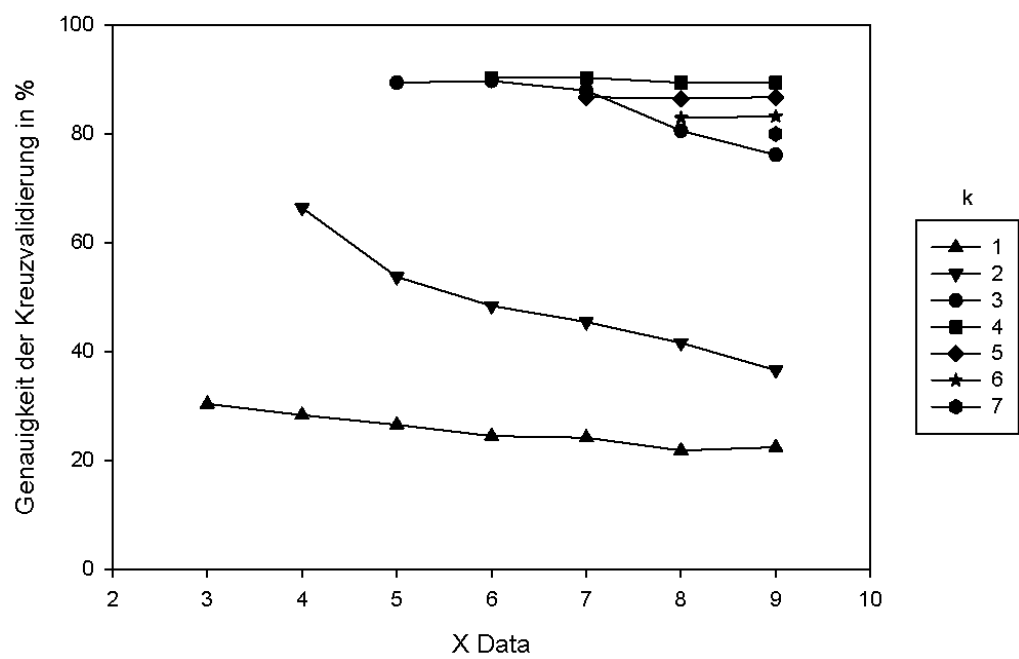


Abbildung 3.5: blabla

Substitution Kernel

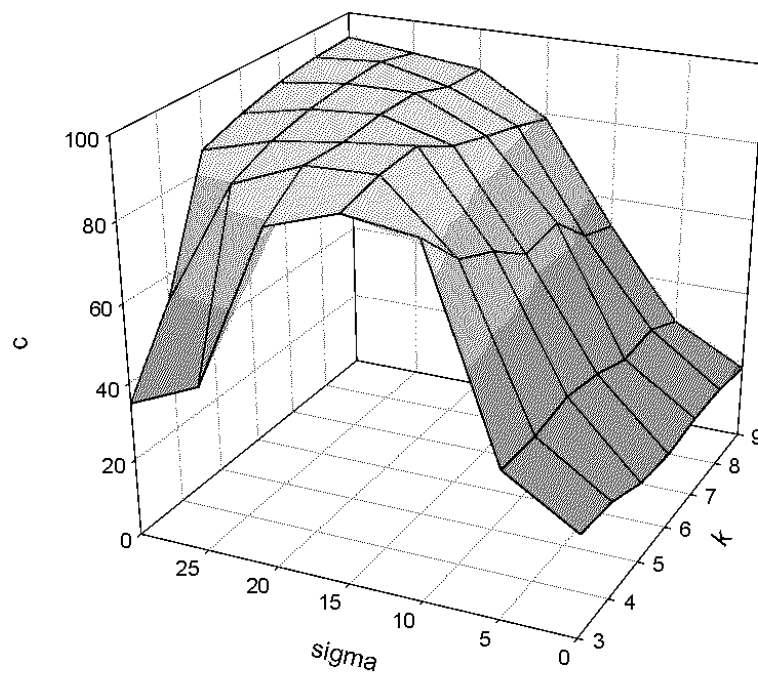


Abbildung 3.6: blabla

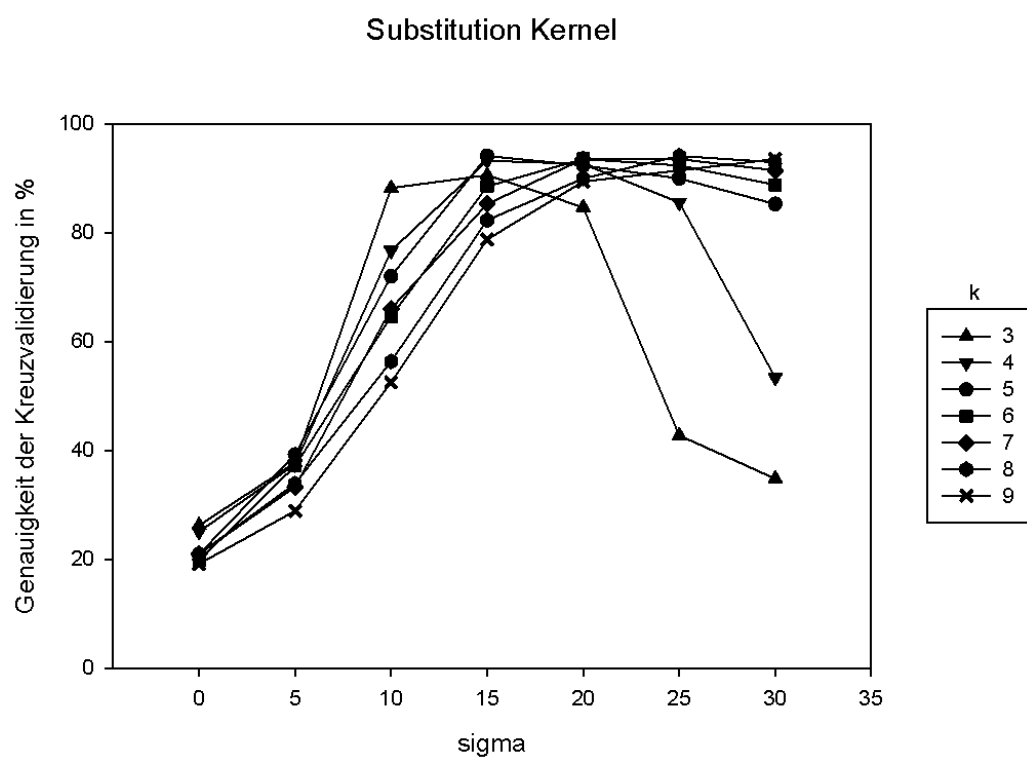


Abbildung 3.7: blabla

Kapitel 4

Diskussion