Institut für Informatik
Lehr- und Forschungseinheit
für Datenbanksysteme

Diplomarbeit

Graph-based Functional Classification of Proteins using Kernel Methods

Karsten Borgwardt

Aufgabensteller:	Prof. Dr. Hans-Peter Kriegel, LMU München
Betreuer:	Dr. Stefan Schönauer, LMU München
	Dr. Alexander Smola, NICTA Canberra
Abgabetermin:	1. Dezember 2004

Chapter 2

Protein Graphs and Kernels

Overview of this Project To combine structural and feature vector based functional prediction, the following steps had to be taken: We had to develop graph models of proteins that included structural and physical-chemical information (section 2.1). Then we had to implement graph kernel methods and run SVM functional classification on sets of proteins (section 2.2). Furthermore, we addressed two central questions when classifying graphs using a large set of attributes: How can more and less relevant attributes be distinguished (chapter 4)? How can we combine two relevant attributes (chapter 3)? Finally, we examined the effect of different parameter settings within our graph model and kernels on the prediction accuracy of our classification system (chapter 5).

2.1 Protein Graph Model

Basics from Graph Theory Let us first define the most important concepts of graph theory which we will need when designing graph models and graph kernels for proteins:

A *graph* $G = (V, E)$ consists of a set of *nodes* (or *vertices*) V and *edges* E. Every edge connects a pair of nodes. An *attributed* graph is a graph with labels on nodes and/or edges; we will often refer to labels as *attributes*. In our case, attributes will consist of pairs of the form (*attribute-name*, *value*); the notation *attribute-name(node)* shall stand for the value of the attribute with name *attribute-name* of this node.

The adjacency matrix A of Graph $G = (V, E)$ is defined as

$$[A]_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise} \end{cases},$$

where v_i and v_j are nodes in G. We will call v_j a *neighbour node* of v_i if $A(i, j) = 1$.

A *directed* graph is a graph where every edge has a direction, i.e. one of the nodes it is connecting is the *origin* (or *source*) node, the other is the *target* node. In *undirected* graphs, edges do not have a direction associated with them.

A *walk* in a graph is a sequence of nodes $v_1, v_2, \cdots, v_k$ where $(v_{i-1}, v_i) \in E$ $\forall i \in \mathbb{N}$ with $1 < i \leq k$. Its length is $k - 1$. It is closed if $v_1 = v_k$. A *cycle* is a walk where $v_1 = v_k$, but all other nodes are dissimilar. A *path* is a walk with no repeated nodes. A graph is *connected* if every pair of nodes is joined by a path. A graph is *acyclic* if it has no cycles. A *tree* is a connected acyclic graph, where one node is distinguished from the others; it is called the *root*. We define the height of a tree to be one plus the length of the longest path from the root to a node in the tree. These definitions will be sufficient to understand the elements of graph theory mentioned in this thesis.

Graph Structure When developing a graph representation for proteins, we designed our graph model to fulfil two competing requirements: computational feasibility and biological significance.

Nodes in our graph model represent secondary structure elements (SSEs) of proteins, i.e. helixes, strands of sheets and turns. Alternatively, one could have modelled nodes to be atoms or amino acid (AA) residues. Unfortunately, these models would have included thousands and tens of thousands of nodes, respectively. Calculating graph kernels on such huge graphs would have shifted the centre of this project from bioinformatics to computational feasibility.

Graph Information Not only the sequence of secondary structure elements, but also chemical and physical features encoded by these sequences might be hints at protein function. This assumption has been shown to be valid by successful experiments using chemical and physical feature vectors to classify proteins (CMY$^+$04, DMHK95). As many different features have been employed in these and similar studies and analyses on the significance of these attributes have been quite contradictory, we kept to the most commonly used set of four attributes, namely hydrophobicity, polarity, polarizability and normalised van der Waals volume, originally proposed by (DMHK95, DMM$^+$99). One total normalised van der Waals value was determined for each node individually. Additionally, each node was labelled by the total number of its residues with low (L), medium (M) or high (H) normalised van Der Waals volume separately; we will refer to this as the *three bin distribution*. Analogously, one total value and the three bin distribution were added to every node for hydrophobicity, polarity and polarizability.

Beside these physical features, the length of each SSE in AAs and the distance between the Carbon-alpha-atom (C_α) of its first and last residue

in Ångstroms (Å) became further node attributes, called *AA length* and *3-d length*, respectively.

Two types of edges were introduced: *sequential* edges connect nodes that are neighbours along the amino acid sequence, i.e. there is no other SSE between them along the amino acid chain; *structural* edges connect every SSE to its three nearest spatial neighbours. Every edge was labelled with its type, i.e. structural or sequential. Besides, sequential edges were labelled with their length in AA and structural edges with their length in Å. The length of a structural edge between two SSEs was calculated to be the distance between their centres; the centre of an SSE was defined as the centre of the line between the C_α atom of its first and the C_α atom of its last residue.

Graph Generation Protein graphs were generated from protein files, obtained from the RCSB Protein Data Bank (PDB) (BWF$^+$00). Providing the length and position of SSEs, the AA sequence of a protein and the coordinates of its atoms, the PDB made it possible to derive all data described above, except for the chemical and physical features. These were assigned to SSEs using amino acid indices from the AA Index Database (KOK99). An amino acid index is a table with one value for each amino acid characterising a chemical or physical feature of this AA. Normalised Amino Acid indices for hydrophobicity (CBCG92), van der Waals Volume (FCK$^+$88), polarity (Gra74), and polarizability (CC82) were applied to the sequence of each SSE node to derive its chemical and physical properties (table 2.1).

Graph Format Graphs were stored in the Graph Modelling Language (GML) which is closely related to the Graphlet graph algorithm and modelling package developed at the University of Passau, Germany. A short example will show that the format is almost self-explanatory:

Amino acid	Hydrophobicity	Van der Waals Volume	Polarity	Polarizability
A	0.01 M	0.12 L	0.62 M	0.11 L
L	0.63 H	0.5 M	0.38 L	0.45 M
R	-0.23 L	0.76 H	0.81 H	0.71 H
K	-0.23 L	0.59 H	0.87 H	0.53 H
N	-0.43 L	0.37 M	0.89 H	0.33 M
M	0.55 H	0.55 H	0.44 L	0.54 H
D	-0.57 L	0.34 L	1 H	0.26 L
F	0.75 H	0.73 H	0.4 L	0.71 H
C	0.43 H	0.3 L	0.42 L	0.31 M
P	-0.05 M	0.34 L	0.62 M	0.32 M
Q	-0.61 L	0.49 M	0.81 H	0.44 M
S	-0.54 M	0.2 L	0.71 M	0.15 L
E	-0.63 L	0.47 M	0.95 H	0.37 M
T	-0.43 M	0.32 L	0.66 M	0.26 L
G	-0.44 M	0 L	0.69 M	0 L
W	0.94 H	1 H	0.42 L	1 H
H	0.14 M	0.58 H	0.8 H	0.56 H
Y	0.61 M	0.8 H	0.48 L	0.73 H
I	1 H	0.5 M	0.4 L	0.45 M
V	0.62 H	0.37 M	0.45 L	0.34 M

Table 2.1: Amino acid attributes and the division of the amino acids into three bins for each attribute (L = Low, M = Medium, H = High; adapted from(CBCG92, FCK$^+$88, Gra74, CC82, DMHK95)).

```
graph [
        name 228L_
        directed 0
        node [
                id 0
                attributes [
                        meanHydro -9.52
                        character 2
                ]
        ]
        node [
                id 1
                attributes [
                        meanHydro -16.3061
                        character 2
                ]
        ]
        edge [
                source 0
                target 1
                 attributes [
                        distance 10
                ]
        ]
]
```

The GML format is structured into units by opening [and closing] tags. Every unit has an identifier in front of the opening tag, which can be one of the following: *graph*, *node*, *edge*, *attributes*. Entries consist of "variablename value" pairs. A graph has an entry *name* and an entry *directed*, which is 1 for directed graphs and 0 otherwise. A graph consists of units of *nodes* and *edges*. A node contains an *id* entry and a unit of *attributes*. An edge has an entry source for its *source* node and an entry target for its *target* node; the values of both must refer to an id of a node within the same graph. Furthermore, an edge also contains a unit of attributes. Units of attributes consist of a list of entries.

Scripts were implemented to generate GML files from PDB entries and to then translate these GML files into MATLAB graph structures. GML files are also readable by the JAVA graph algorithm package by (Sch04). GML files were created as an intermediate product between PDB files and MATLAB structures to allow future use of these proteins graphs with other algorithms

and other programming languages.

2.2 Graph Kernels

The next step was to design kernel functions for our protein graph models.

2.2.1 The Type Graph Kernel

The first kernel implemented was a naive graph kernel, called the "type graph kernel". Every node n_1 from input graph G_1 was compared to every node n_2 in input graph G_2 using a Dirac kernel with

$$k_{type\ node}(n_1, n_2) = \begin{cases} 1 & \text{if type}(n_1) = \text{type}(n_2), \\ 0 & \text{otherwise.} \end{cases}$$

The graph kernel value for G_1 and G_2 was then the sum over all kernel values of pairs of nodes n_1 and n_2 from G_1 and G_2, respectively:

$$k_{type\ graph}(G_1, G_2) = \sum_{n_1 \in G_1, n_2 \in G_2} k_{type\ node}(n_1, n_2).$$

The type graph kernel is a convolution kernel over nodes and consequently positive definite. It is by far the simplest kernel used in this work. It mainly served as a comparator, to check whether more information about structure and features encoded in the graph would have an effect on classification accuracy.

2.2.2 The Random Walk Kernel

Originally, random walk kernels were proposed by (GFW03) and (KTI03). Given two graphs, a random walk kernel counts the number of *matching* labelled random walks. The match between two nodes or two edges is determined by using a Dirac kernel on their labels. The measure of similarity

between two random walks is then the product of the kernel values corresponding to the nodes and edges encountered along the walk. The kernel value of two graphs is then the sum over the kernel values of all pairs of walks within these two graphs:

$$k_{graph}(G_1, G_2) = \sum_{walk_1 \in G_1} \sum_{walk_2 \in G_2} k_{walk}(walk_1, walk_2).$$

An elegant approach for calculating all random walks within two graphs by (GFW03) uses direct product graphs:

Definition: Direct Product Graph The graph product of two graphs $G_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $G_2 = (\mathcal{V}_2, \mathcal{E}_2)$ shall be denoted by $G_1 \times G_2$. The node and edge set of the direct product are respectively defined as:

$$\begin{aligned} \mathcal{V}(G_1 \times G_2) &= \{(v_1, v_2) \in \mathcal{V}_1 \times \mathcal{V}_2 : (label(v_1) = label(v_2))\} \\ \mathcal{E}(G_1 \times G_2) &= \{((u_1, u_2), (v_1, v_2)) \in \mathcal{V}^2(G_1 \times G_2) : (u_1, v_1) \in \mathcal{E}_1 \\ &\quad \wedge (u_2, v_2) \in \mathcal{E}_2 \wedge (label(u_1, v_1) = label(u_2, v_2))\} \end{aligned}$$

Based on this direct product graph, the random walk kernel is defined as

Definition: Random Walk Kernel Let G_1, G_2 be two graphs, let $A_\times$ denote the adjacency matrix of their direct product $A_\times = A(G_1 \times G_2)$, and let $\mathcal{V}_\times$ denote the node set of their direct product $\mathcal{V}_\times = \mathcal{V}(G_1 \times G_2)$. With a weighting factor $\lambda \in \mathbb{R}$ and $\lambda \geq 0$ the random walk graph kernel is defined as

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [\sum_{n=0}^{\infty} \lambda^n A_\times^n]_{ij}$$

Nodes and edges in graph $G_1 \times G_2$ have the same labels as the corresponding nodes and edges in G_1 and G_2. Introducing λ means that random walks of length n get weight λ^n in the sum over all walks. λ must be chosen appropriately if the sum shall converge (GFW03). Alternatively to examining all walks up to length ∞, one can also calculate the random walk kernel for walks up to a fixed length only. All pairs of longer walks are then assigned kernel value zero. This zero extension preserves the positive definiteness of the random walk kernel (Hau99).

Our Modifications This graph kernel is designed for discrete attributes: Attributes of two nodes v_1 and v_2 are considered similar if they are completely identical, i.e. the kernel employed on the nodes is a Dirac kernel. The nodes in our protein graph, however, contain several continuous attributes; these are almost never completely identical between two nodes. For that reason, the Dirac kernel for nodes is not a good measure of similarity for the nodes in our protein graphs; we replaced the Dirac kernel by a more complex kernel which reflects knowledge about protein structure. The modified random walk graph kernel and biological justifications for our changes will be presented in the following:

Definition. Modified Random Walk Kernel Consider two walks, namely $walk_1 = (v_1, v_2, ..., v_{n-1}, v_n)$ in Graph $G_1 = (V, E)$ where v_i is a node in V for $i \in \{1, \ldots, n\}$ and $walk_2 = (w_1, w_2, ..., w_{n-1}, w_n)$ in Graph $G_2 = (W, F)$ where w_i is a node in W for $i \in \{1, \ldots, n\}$. Let pairs of nodes from the same graph represent the edge that is connecting them. Let x and x' be either both edges or both nodes.

Then the walk kernel shall be defined as:

$$k_{walk}(walk_1, walk_2) = k_{step}((v_1, v_2), (w_1, w_2)) * k_{step}((v_2, v_3), (w_2, w_3)) \\ * \ldots * k_{step}((v_{n-1}, v_n), (w_{n-1}, w_n))$$

with

$$k_{step}((v_i, v_{i+1}), (w_i, w_{i+1})) = k_{node}(v_i, w_i) * k_{edge}((v_i, v_{i+1}), (w_i, w_{i+1})) \\ * k_{node}(v_{i+1}, w_{i+1}),$$

for $i \in \{1, .., n-1\}$, and k_{node} is defined as

$$k_{node}(v_i, w_i) = k_{type}(v_i, w_i) * k_{node\ labels}(v_i, w_i) * k_{length}(v_i, w_i)$$

and k_{edge} as

$$k_{edge}((v_i, v_{i+1}), (w_i, w_{i+1})) = k_{type}((v_i, v_{i+1}), (w_i, w_{i+1})) * k_{length}((v_i, v_{i+1}), (w_i, w_{i+1})).$$

k_{type} is defined identically for both nodes and edges:

$$k_{type}(x, x') = \begin{cases} 1 & \text{if type}(x) = \text{type}(x'), \\ 0 & \text{otherwise} \end{cases}$$

k_{length} is defined identically for both nodes and edges, but c = 2 for edges and c = 3 for nodes:

$$k_{length}(x, x') = max(0, c - |length(x) - length(x')|).$$

The node labels kernel $k_{node\ labels}$ is a Gaussian RBF kernel over two vectors

representing labels of node v_i and node w_i:

$$k_{node\ labels}(v_i, w_i) = \exp\left(-\frac{\|labels(v_i) - labels(w_i)\|^2}{2\sigma^2}\right).$$

If we now use a slightly modified version of the adjacency matrix of the direct product graph, namely

$$[A_\times]_{ij} = \begin{cases} k_{step}((v_i, v_j), (w_i, w_j)) & \text{if } ((v_i, v_j), (w_i, w_j)) \in E_\times, \\ 0 & \text{otherwise} \end{cases},$$

where $E_\times$ is the set of edges in the direct product graph and and (v_i, v_j) is an edge from G_1 and (w_i, w_j) is an edge from G_2, then we can finally define the modified random walk kernel on graphs over the direct product graph as:

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [\sum_{n=0}^{\infty} \lambda^n A_\times^n]_{ij}$$

where λ is defined as above.

As it is essential to show that this modified graph kernel is still a valid kernel, we will prove its positive definiteness in the following lemma.

Lemma The modified random walk graph kernel is positive definite. *Proof*: In the first step we will show that the node and edge kernels defined above are positive definite. The type kernel is a Dirac kernel which is known to be positive definite. The length kernel is a Brownian bridge kernel which is also a known positive definite kernel (SS02). The same applies to the node labels kernel, because it is a Gaussian RBF kernel. Since point-wise multiplication of two positive kernel matrices preserves positive definiteness, node kernel and edge kernel and step kernel are consequently positive definite.

As a second step, the positive definiteness of the walk kernel k_{walk} will be proven (KMFI04). The set of all possible labelled walks $\mathcal{W}$ can be divided into subsets of fixed length as $\mathcal{W}_1, \mathcal{W}_2, \ldots$. We define $k^{(j)}_{walk}$ as k_{walk} whose domain is limited to the subset $\mathcal{W}_j \times \mathcal{W}_j$, then $k^{(j)}_{walk}$ is a valid kernel as it is described as the tensor product of edge and node kernels (SS02). Now let us expand the domain of $k^{(j)}_{walk}$ to the whole set $\mathcal{W} \times \mathcal{W}$ by assigning zero when one of the inputs is not included in $\mathcal{W}_j$, and call it $\bar{k}^{(j)}_{walk}$. This zero extension preserves positive definiteness (Hau99). Since k_z is the sum over all $\bar{k}^{(j)}_{walk}$, it turns out to be a valid kernel.

The last step is to demonstrate the positive definiteness of the modified random walk graph kernel itself. It follows directly from the definition of the random walk graph kernel, as it is a convolution kernel, proven to be positive definite by (Hau99). ■

This modified random walk kernel was designed to include biological meaning and to ease computation.

Biological Aspects of Kernel Design

Let us first consider the biological relevance of the kernels employed. The type kernel makes sure that a step in a random walk in two input graphs can only be performed if both edges are of the same type, i.e. both sequential or both structural, and both origin nodes and both target nodes are of the same type, i.e. helix, loop or turn. Identical motifs of SSEs both within protein structure and along the amino acid chain are strong hints at structural and functional relationship; in fact, most databases that group proteins into structural and functional families do so by secondary structure content analysis, both on sequence and structure level (SCOP (AHB$^+$04), CATH (OPT03)).

The length kernels ensure that we do not count SSEs or edges as being similar that differ a lot in size. Insertion and deletion of amino acid residues might change the length of SSEs or their distance towards each other, while the overall fold and function of the protein remains unchanged. For this reason, we employed the Brownian bridge kernel, that assigns the highest kernel value to SSEs and edges that are identical in length and zero to all SSEs and edges that differ in length more than a constant c. This maximum difference constant for sequential edges was set to 2 AA, for structural edges to 2 Å and for SSE nodes to 3 Å.

A Gaussian kernel was chosen as the node labels kernel, since these have shown the best performance in related successful studies of classifying proteins (CHJC04).

Computational Aspects of Kernel Design

Beside their biological interpretation, the chosen kernels also enable us to limit the computational costs of calculating kernel matrices for big sets of graphs. An easy example calculation can illustrate the complexity of the problem faced: If we are calculating a kernel matrix for 1000 proteins with only 25 SSEs and 100 edges each without any optimisations, then we have to determine kernel values for 10^6 pairs of graphs; each kernel value of two graphs is the sum over all entries of the kernel matrix of all 10^4 pairs of their edges. Consequently, we have to perform 10^{10} edge kernel value calculations to determine the graph kernel matrix for the 1000 proteins. The goal must be to minimise the cost of calculating an edge kernel value and to exclude all pairs of edges from the calculation that will give kernel value zero.

Symmetry of Kernel Function The first and simplest step is to remember that positive definite kernel functions are symmetric; this fact halves the number of calculations, both on graph and edge level.

Selectiveness Second, our edge and node kernel functions are products of kernel functions; if the value of one of the factors, i.e. one of the kernel functions, is 0, the total product will be 0 and the calculation of all other factors becomes needless. The direct product graph approach uses the adjacency matrix of the direct product graph to calculate the kernel value of two graphs. We modified the adjacency matrix for the modified random walk kernel by making its entries step kernel values, i.e. a product of node and edge kernels.

As a consequence, if either the kernel value of a pair of edges, of their two origin nodes or of their two target nodes becomes zero, we can conclude that the corresponding entry in the modified adjacency matrix is zero and can omit further calculations for this entry. The type kernel is a Dirac kernel and the length kernel is a Brownian bridge kernel; both kernel functions are highly selective, producing sparse kernel matrices. We can exploit this characteristic of our random walk kernel to reduce computational efforts: By checking first if type or length kernel are zero, we can avoid calculating the node labels kernel value if they are indeed zero.

Speeding up the Gaussian Kernel Third, we can save computational time on the Gaussian kernel. Its equation can be rewritten as:

$$k_{node\ labels}(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right) =$$
$$= \exp\left(-\frac{\|x\|^2}{2\sigma^2} - \frac{\|x'\|^2}{2\sigma^2} + \frac{2 * x * x'}{2\sigma^2}\right)$$

If we precalculate $-\frac{\|x\|^2}{2\sigma^2}$ and $-\frac{\|x'\|^2}{2\sigma^2}$ for all input nodes x and x', then the effort of determining the Gaussian kernel value for a pair of nodes is reduced to calculating $\frac{2*x*x'}{2\sigma^2}$ and to then exponentiating the sum of this and the two precalculated terms. If one tries to exponentiate each addend individually and to then multiply the results, one is likely to run into numerical problems, as MATLAB cannot calculate $\exp x$ if $x > 709$. The term $\frac{2*x*x'}{2\sigma^2}$ can easily exceed values of 709, while the overall sum $-\frac{\|x\|^2}{2\sigma^2} - \frac{\|x'\|^2}{2\sigma^2} + \frac{2*x*x'}{2\sigma^2}$ is less than $\frac{2*x*x'}{2\sigma^2}$ as the first two addends are negative.

Optimising MATLAB Code Fourth, the MATLAB software package used for the kernel computations has certain characteristics that can be exploited to speed up calculation, or in other terms, that must be avoided not to slow down the code. Calculating a kernel matrix using the modified random walk kernel involves thousands of comparisons of pairs of edges, nodes and graphs. The simplest and general solution is to generate graph, edge and node pairs by iterating over two graph, edge and node sets, respectively, using two for-loops; the simplest and general solution to compare edge and node labels is to use an if-statement.

However, MATLAB's inner implementation of for-loops and if-clauses is quite slow, while arithmetic matrix operations and the find-statement are much faster. The find-statement finds all entries in a matrix that match a certain logical expression. We therefore designed our kernel to contain as few for-loops and if-statements as possible and to calculate iterations and comparisons using matrix operations and the find-statement. The resulting implementation of the type and the length kernel will be described in the following, as they are good examples for optimised implementation in MATLAB.

Type Kernel Implementation As defined above, the type kernel is 1, if the type of nodes or edges is identical, otherwise it is zero. To implement this as a matrix operation, one takes a vector v_1 from Graph G_1 containing the types of all the edges in G_1; the same is done with a vector v_2 for the types of all edges in G_2. Let m be the length of v_1 and n be the length of v_2, i.e. m is the number of edges in G_1 and n the number of edges in G_2. To get a pairwise comparison, we concatenate n copies of v_1 horizontally to get a $m * n$ matrix M_1 of n copies of v_1. Analogously, we concatenate m copies of the transpose of v_2, v_2', vertically to produce a $m * n$ matrix M_2 of m copies of v_2'. All entries in matrix $M_1 - M_2$ that equal zero then represent a pair of edges from G_1 and G_2 with identical type. We can then get the position of these pairs within the matrix $M_1 - M_2$ using a find($M_1 - M_2 = 0$) statement in MATLAB. Completely analogously, one can calculate the type kernel for nodes.

Length Kernel Implementation The length kernel as a Brownian bridge kernel is zero, if the difference of two edge or node lengths is greater than a constant. We determine the nonzero pairs for the Brownian kernel by performing the same concatenation operations as in the type kernel case; the only difference is that we are dealing with vectors v_1 and v_2 that now contain length, not type information. We determine the difference matrix $D = C - |M_1 - M_2|$, where C is a matrix of the same size as M_1 and M_2 with all its entries equal to the length kernel constant c. If we then set all entries in D that are negative to zero using the find-statement $D[find(D < 0)] = 0$, we have calculated the Brownian bridge kernel for all pairs of edges.

Avoiding for-loops and if-statements reduced the computational time costs for kernel matrix calculations by a factor of more than 10.

Parallelisation Fifth, if one cannot further reduce the computation time for a pair of graphs, we can at least distribute the job of calculating a full kernel matrix over several workstations. Distribution scripts were implemented for the kernel matrix calculations which allow to run the calculation on several machines at the same time; maximum 16 workstations were used in parallel.

2.2.3 The Subtree Kernel

While the random walk kernel examines the similarity of walks within two graphs, one could also examine subtrees in graphs. Subtrees extended from the root node to its neighbours, then to the neighbours of its neighbours and so on[1]; this extension is multi-directional, while a random walk is always extended "linearly". This subtree graph kernel was proposed by (RG03).

Definition: Subtree Graph Kernel Let $G_1(\mathcal{V}_1, \mathcal{E}_1), G_2(\mathcal{V}_2, \mathcal{E}_2)$ be two graphs. Let r be a node with $r \in \mathcal{V}(G_1)$ and analogously, s be a node with $s \in \mathcal{V}(G_2)$. $\delta^+(n)$ shall denote the set of all neighbour nodes of node n. h is the height of the subtrees we examine. If $h = 1$ and label(r) = label(s), then $k_h(r, s) = 1$; if h = 1 and label(r)$\neq$label(s) then $k_h(r, s) = 0$. For h > 1, $k_h(r, s)$ is computed as follows: Let $M(r, s)$ be the set of all matchings from $\delta^+(r)$ to $\delta^+(s)$, i.e.

$$M(r,s) = \{R \subseteq \delta^+(r) \times \delta^+(s) | (\forall (a,b),(c,d) \in R : a = c \iff b = d) \\ \wedge (\forall (a,b) \in R : label(a) = label(b))\}$$

[1]The fact that we are recursively visiting neighbours of neighbours could lead to a cycle in our subtree which is not allowed by the definition of a tree as an acyclic graph; for our subtree kernel, it is however unavoidable to allow these repetitions of nodes in trees as we would otherwise lose the positive definiteness of our kernel (RG03).

Then $k_h(r,s)$ can be computed as

$$k_h(r,s) = \lambda_r \lambda_s \sum_{R \in M(r,s)} \prod_{(r',s') \in R} k_{h-1}(r',s'),$$

where λ_r and λ_s are positive values; if set to smaller than 1, they cause higher trees to have a smaller weight in the overall sum. The subtree graph kernel for for G_1 and G_2 can then be defined for a fixed height h as

$$k_{tree,h}(G_1, G_2) = \sum_{r \in \mathcal{V}_1} \sum_{s \in \mathcal{V}_2} k_h(r,s).$$

One can also define this graph kernel for h approaching infinity:

$$k_{tree}(G_1, G_2) = \lim_{h \to \infty} k_{tree,h}(G_1, G_2),$$

which will converge for suitable choice of λ_r and λ_s. Both versions of the subtree graph kernels have been shown to be positive definite (RG03), as they are convolution kernels on subtrees. As $k_{tree,h}$ is already a positive definite kernel, sufficiently large choice of h will provide a good approximation of k_{tree}.

The Modified Subtree Kernel The original subtree kernel by (RG03) does not fit our problem. Again, node labels are compared by a Dirac kernel, namely $k_1(r,s)$. We have earlier explained that a Dirac kernel is not suitable for our node labels that are hardly ever identical between two nodes. We therefore redefined $k_1(r,s)$ as

$$k_1(r,s) = k_{node}(r,s) = k_{type}(r,s) * k_{node\ labels}(r,s) * k_{length}(r,s)$$

where

$$k_{node\ labels}(r,s) = \exp\left(-\frac{\|labels(r) - labels(s)\|^2}{2\sigma^2}\right).$$

Furthermore, the original subtree kernel only compares node labels of neighbour nodes, not edge labels. It is essential for our protein comparison that we measure similarity between subtrees by comparing distances between nodes in these subtrees, i.e. by comparing distance edge labels.

Additionally, the original subtree kernel compares subtrees by comparing neighbours of two root nodes, but not the root nodes themselves. In our scenario, it is essential that we measure the similarity of the root nodes by a kernel first, as two subtrees with dissimilar roots should not be given a high kernel value. This can be achieved by multiplying a node labels kernel for the root nodes to the subtree kernel.

Consequently, we redefined for h $>$ 1:

$$k_h(r,s) = k_{node\ labels}(r,s) * \lambda_r\lambda_s \sum_{R\in M(r,s)} \prod_{(r',s')\in R} k_{h-1}(r',s') * k_{edge}((r,r'),(s,s'))$$

where

$$k_{edge}((r,r'),(s,s')) = k_{type}((r,r'),(s,s')) * k_{length}((r,r'),(s,s')).$$

$k_{node\ labels}$, k_{type} and k_{length} for nodes and edges are defined exactly in the same way as in section 2.2.2.

This modified subtree graph kernel $k_{modified\ subtree,h}$ is still positive definite, as

$$k_{modified\ subtree,h}(G_1,G_2) = \sum_{r\in V_1}\sum_{s\in V_2} k_h(r,s).$$

is a convolution kernel on subtrees, while $k_h(r,s)$ is a convolution kernel on nodes and edges which is multiplied pointwise to a Gaussian RBF kernel,

namely $k_{node\ labels}$. The kernels on nodes and edges are positive definite, as they are products of Brownian bridge, Dirac and Gaussian kernels.

Our modified subtree graph kernel is designed to measure subtree similarities between two graphs. It seems attractive to compare subtrees instead of walks in our protein case, as subtrees represent 3-d protein structures more directly than linear random walks do. However, determining the set $M(r, s)$, i.e. the set of all possible mappings between sets of neighbour nodes between r and s, is computationally expensive, as we have to consider all pairs of subsets of the neighbour sets of r and s. The number of these subsets is growing exponentially with the size of the neighbour set.

Additionally, all nodes in our protein graphs have at least 4 neighbours, namely its 3 nearest neighbour nodes in space and at least 1 neighbour node long the sequence. This means that the number of neighbour nodes grows exponentially in every recursion step, i.e. the higher our subtrees get. We experienced in our test runs, that this characteristic produces an enormous computational slow-down if we exceed height 3. That is why we chose height 3 as our default height of our modified subtree graph kernel subtrees.

2.2.4 Experiments

Experimental Setting After designing graph kernels under theoretical considerations, we tested their experimental performance. All the experiments in this thesis are based on the following setting: enzyme proteins with known structure are modelled as graphs and then classified into one of six EC top level classes, i.e. into one of the six classes of enzymes defined by the Enzyme Commission.

We chose enzymes for our protein function prediction and the EC hierarchy as our functional classification scheme for five reasons: First, enzymes

make up more than half of all known protein structures. Second, EC numbers of enough protein structures are known to give us sufficient numbers of training samples. Third, EC numbers are available for many proteins in almost every protein database, while GO annotations are still under construction. Fourth, many function prediction approaches have used the EC hierarchy as functional classification scheme (CHJC04, DD03) and therefore performance comparisons will be made easier if we use the same scheme. Fifth, the choice of EC does not mean that we cannot apply the same procedure to GO classes in absolutely analogous experiments.

Technical Note on Hardware and Software For all experiments in this project, Debian Linux workstations with Intel(R) Pentium(R) 4 CPU at 3.00 GHz and cache size 512 KB were employed. For SVM training and prediction we used the SVLAB package by Alexandros Karatzoglou, Alex Smola, Achim Zeileis, and Kurt Hornik. We extended this package to be able to handle our graph kernels and graph kernel matrices. SVLAB is written in MATLAB(R), a mathematical programming language by the MathWorks company. We coded all kernel functions developed in this project and accompanying data analysis scripts in MATLAB, release 13. Scripts handling text files for data analysis were coded in PERL, version 5.8.4.

Note on Statistics Let us define some statistical terms that will reoccur in all our experiments, to make their meaning clear for the rest of this thesis.

Performing *x-fold cross-validation* classification on a given data set means: We divide the data set into x subsets of equal size. Then we train our classifier on x-1 subsets and test it on the remaining subset. This step is repeated x times; every time another subset becomes the test set until all subsets have been the test set exactly once.

1-vs.-rest classification means that we are dealing with a dataset consisting of more than two classes and that we are trying to distinguishing members of one class from non-members of this class ("the rest").

Prediction *accuracy* is the number of test points that were correctly classified divided by the number of all test points. *Sensitivity* is the percentage of test points with positive label that were classified to be positive. *Specificity* is the percentage of test points with negative label that were classified to be negative. The *positive predictive value* is the percentage of test points with positive labels among all test points that were predicted to have positive labels. The *negative predictive value* is the percentage of test points with negative label among all test points that were predicted to have negative labels. We will report these classification quality measures as percentages throughout the thesis.

Now we have clarified all technical details necessary to follow our experiments.

Type vs. Random Walk vs. Subtree Graph Kernel We first conducted an experiment to compare the performance of our three graph kernels, namely the type graph kernel, the modified random walk graph kernel and the modified[2] subtree graph kernel.

From the BRENDA database (SCH+02), we obtained lists of PDB protein IDs, abbreviations that uniquely identify a protein structure within the PDB, associated with EC numbers. We randomly picked 60 proteins from each of the six EC top level classes, 360 proteins in total. We ran the type graph kernel, the random walk kernel and the subtree kernel on this set of 360 enzymes. In our experiments, we considered random walks of up to length

[2]We will skip 'modified' in the following in our experiments, as we never used the original, unchanged random walk and subtree kernel.

accuracy	type	random walk	subtree
EC 1	83.33 ± 0.00	90.83 ± 2.94	86.11 ± 2.27
EC 2	83.33 ± 0.00	89.72 ± 3.48	84.72 ± 1.46
EC 3	83.33 ± 0.00	86.11 ± 2.27	78.61 ± 16.92
EC 4	83.33 ± 0.00	88.33 ± 2.19	85.28 ± 2.94
EC 5	86.94 ± 4.15	90.28 ± 3.53	86.67 ± 2.55
EC 6	83.33 ± 0.00	89.72 ± 2.64	84.72 ± 1.46
average	83.93	89.17	84.35
SD	1.47	1.71	2.92

Table 2.2: Overall accuracy of 10-fold cross-validation on a set of 360 proteins for the type kernel, the random walk and the subtree kernel for all EC classes (1-vs.-rest; $\pm$ indicates standard deviation in cross-validation, SD is standard deviation among EC classes).

3 for default. These limits allowed us to avoid memory problems which we will explain in chapter 5. We set the λ parameter to 2 for the random walk graph kernel, and set $\lambda_r * \lambda_s = 2$ for the subtree graph kernel, so that both assign increasing weights to longer walks and higher subtrees, respectively. We determined the Gaussian σ parameter on a sample of nodes to be 1083, following a common rule of thumb explained in chapter 5.

On the three resulting kernel matrices, we performed C-SVM classification 1-vs.-rest 10-fold cross-validation. Throughout the thesis, we tested powers of 10^2 as values for the soft-margin classification parameter C, ranging from 10^{-6} to 10^6. The overall accuracies of our three kernels on our set of 360 proteins are given in table 2.2, detailed results are given in tables 2.3 and 2.4.

Results The random walk graph kernel performs best of all three graph kernels, across all six EC classes, with 89.17% in average (table 2.2). The subtree graph kernel gives non-random classification accuracies, but does

not reach the accuracy levels of the random walk kernel. For EC class 3, the subtree graph kernel can only classify 78.61% of proteins correctly, while its accuracies are better than a random classifier for the remaining classes.

Remember that we classify 1-vs.-rest and that we are dealing with 360 proteins, 60 from each of six classes. The number of non-class-members is therefore always 5times higher than the number of class-members, both in training and test sets. A naive classifier that declares all test set proteins to be class-members would therefore yield an accuracy of 16.67%, a naive classifier that declares all test set proteins to be non-class-members would yield 83.33%, and a random classifier that assigns function randomly with 50% probability for class-membership and 50% for non-class-membership would give 50% accuracy in average[3]. Both random walk and subtree graph kernel give accuracies higher than 83.33% (except for the subtree on EC class 3).

The type graph kernel only gives non-random results for EC class 5, for all other classes it just predicts all test set proteins to be non-members. Of our designed kernels, the random walk kernel shows consequently by far the best performance on our set of 360 enzymes.

While random walk and subtree graph kernel reach high specificity values, the random walk 99.67-100.00% and the subtree kernel 91.67-100.00%, sensitivity is much lower, namely 18.33-46.67% for the random walk and 8.33-20.00% for the subtree kernel (table 2.3 and 2.4).

[3]For simplicity of notation, we will refer to both, naive and random classifiers, as random classifiers in the following.

	type	random walk	subtree
EC 1			
best C	10^6	10^6	10^6
training error	16.67	0.00	0.00
sensitivity	0.00	46.67	16.67
specificity	100.00	99.67	100.00
ppv	-	96.67	100.00
npv	83.33	90.39	85.76
accuracy	83.33	90.83	86.11
EC 2			
best C	10^6	10^{-4}	10^6
training error	16.67	0.00	0.00
sensitivity	0.00	38.33	8.33
specificity	100.00	100.00	100.00
ppv	-	100.00	100.00
npv	83.33	89.13	84.52
accuracy	83.33	89.72	84.72
EC 3			
best C	10^6	10^{-4}	10^6
training error	16.67	0.00	0.00
sensitivity	0.00	18.33	13.33
specificity	100.00	99.67	91.67
ppv	-	95.00	91.94
npv	83.33	85.95	85.48
accuracy	83.33	86.11	78.61

Table 2.3: Results of 10-fold cross-validation 1-vs.-rest on a set of 360 proteins for the type kernel, the random walk and the subtree kernel for EC classes 1 to 3 (ppv = positive predictive value, npv = negative predictive value).

	type	random walk	subtree
EC 4			
best C	10^6	10^{-4}	10^6
training error	16.67	0.00	0.00
sensitivity	0.00	31.67	16.67
specificity	100.00	99.67	100.00
ppv	-	96.67	100.00
npv	83.33	87.98	85.06
accuracy	83.33	88.33	85.28
EC 5			
best C	10^6	10^{-4}	10^6
training error	11.76	0.00	0.00
sensitivity	31.67	43.33	20.00
specificity	98.00	99.67	100.00
ppv	80.00	96.67	100.00
npv	87.81	89.88	86.26
accuracy	86.94	90.28	86.67
EC 6			
best C	10^6	10^{-4}	10^{-4}
training error	16.67	0.28	0.28
sensitivity	0.00	38.33	8.33
specificity	100.00	100.00	100.00
ppv	-	100.00	100.00
npv	83.33	89.08	84.52
accuracy	83.33	89.72	84.72

Table 2.4: Results of 10-fold cross-validation 1-vs.-rest on a set of 360 proteins for the type kernel, the random walk and the subtree kernel for EC classes 4 to 6 (ppv = positive predictive value, npv = negative predictive value).

Evaluation on Independent Test Sets Our cross-validation results on a set of 360 proteins show that our classification system does not classify proteins randomly into functional classes. A statistically even more powerful result would be to demonstrate this not only via cross-validation, but also by evaluation on an independent set of proteins; this independent set shall consist of proteins that are neither training set nor test set members. The idea is to optimise the C-parameter by cross-validation on a given set of proteins and then to perform C-SVM classification on an independent evaluation set. To exclude the possibility that a lucky choice of the independent test set produced good results, the same process is repeated 10 times with different training, test and evaluation sets.

Again, we obtained lists of PDB protein IDs associated with EC numbers from the BRENDA database (SCH$^+$02). 120 proteins were chosen randomly from each of the six classes on the first level of the EC hierarchy, 720 in total. From these 720 proteins, 600 were randomly picked and then 10-fold cross-validation with a C-SVM was performed on them. Repeating the cross-validation with different values for C, the C value optimising the average test error was determined. Finally, this C-value was used to train a C-SVM on all 600 proteins and to predict the labels of the remaining 120 proteins. This whole procedure was repeated 10 times. We report classification accuracies and their standard deviation on the independent evaluation set as average values of these 10 runs (table 2.5).

Results The independent test set evaluation shows that the random walk graph kernel prediction accuracy from our first cross-validation experiments on 360 proteins can be reached on independent test sets as well. With an accuracy of 89.17-92.33%, a sensitivity of 36.12-52.42% and a specificity of 99.81-100% for the six EC top level classes on an independent test set in 10

Enzyme Class	EC 1	EC 2	EC 3	EC 4	EC 5	EC 6	AVG	SD
Independent Evaluation								
Accuracy avg	91.33	91.00	92.33	91.17	91.75	89.17	91.13	1.07
Accuracy $\pm$	2.43	2.00	3.04	2.49	1.44	2.42	-	-
Sensitivity avg	46.35	48.16	52.42	47.66	50.93	36.12	46.94	5.75
Sensitivity $\pm$	12.77	11.30	13.84	10.93	9.07	8.30	-	-
Specificity avg	100.00	100.00	99.81	99.90	99.90	100.00	99.94	0.08
Specificity $\pm$	0.00	0.00	0.39	0.32	0.32	0.00	-	-
Cross-Validation								
Accuracy avg	92.65	91.07	91.03	89.95	91.05	89.67	90.9	1.05
Accuracy $\pm$	0.61	0.58	0.49	0.4	0.49	0.45	-	-
Sensitivity avg	56.10	45.77	48.19	39.67	48.02	37.82	45.93	6.6
Sensitivity $\pm$	3.86	4.93	1.78	3.23	3.12	2.66	-	-
Specificity avg	100.00	100.00	99.72	99.94	99.62	100.00	99.88	0.17
Specificity $\pm$	0.00	0.00	0.11	0.10	0.15	0.00	-	-

Table 2.5: Results of evaluation on independent test set and in 10-fold cross-validation in percent. Parameters were optimised by 10-fold cross-validation on 600 proteins, then these parameters were used for prediction on an independent evaluation set of 120 proteins. Results are reported as averages of 10 iterations (avg is average of 10 iterations, $\pm$ is standard deviation in 10 iterations, AVG is average over all six EC classes, SD is standard deviation over all six EC classes).

iterations, the prediction accuracy of the random walk is shown to be reproducible on different evaluation sets.

In figure 2.1, we compare the overall accuracy for 10-fold cross-validation using the best C-parameter to the accuracy on the independent evaluation set for all 10 iterations, exemplary for EC class 1. On the independent evaluation set, the accuracy is in average only 1.32% lower than in the 10-fold cross-validation with optimal C-parameter for EC class 1.

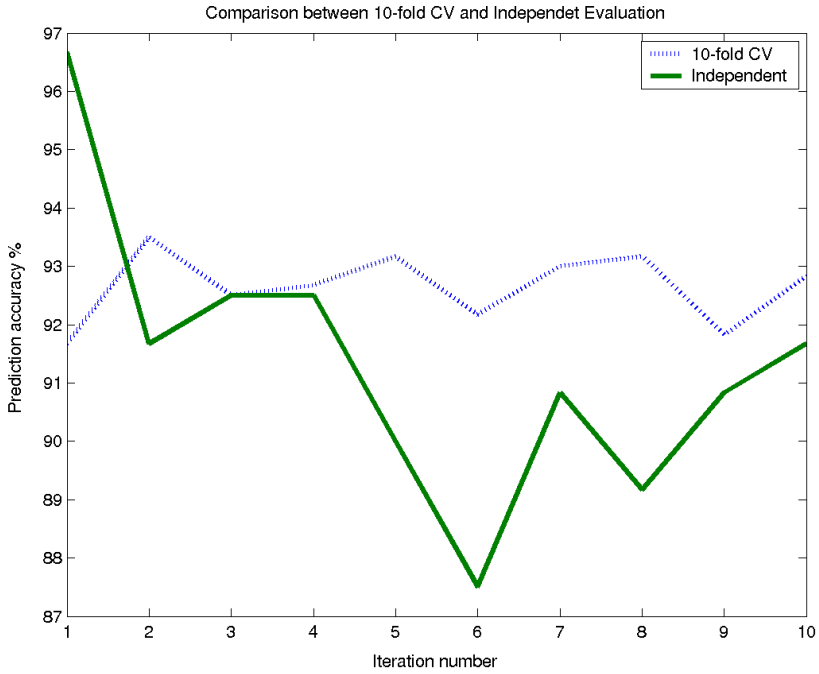

Figure 2.1: Comparison of 10-fold cross-validation and evaluation on independent test set for all 10 iterations, for EC class 1.

The picture is not different for EC classes 2-6. In figure 2.2, we can see that the average classification accuracy on the independent evaluation set is even higher than in 10-fold cross-validation for EC class 3, 4 and 5. Furthermore, in average of all 10 iterations and all EC classes, we reach 91.13 % $\pm$ 2.3% accuracy on the independent evaluation set and 90.9% $\pm$ 0.5 % for

the cross-validation. This is amazing as we had optimised the C-parameter for the cross-validation, but not on the independent evaluation set, i.e. we reached worse classification accuracies in the cross-validation although we could optimise over a free parameter. The obvious explanation is that we yield better classification accuracies in the independent evaluation because we are training on more proteins, namely 600 instead of 540 for the cross-validation.

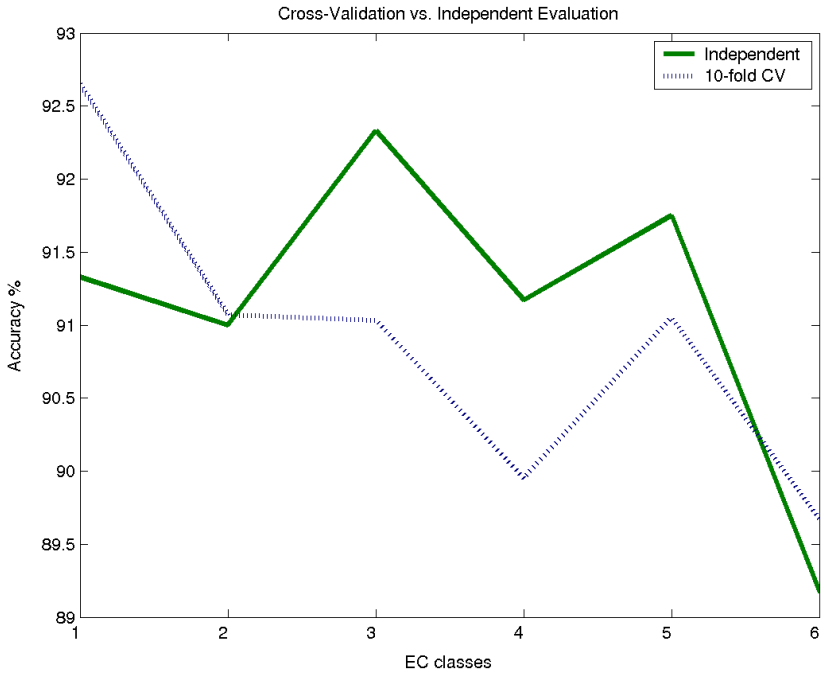

Figure 2.2: Comparison of 10-fold cross-validation and evaluation on independent test set as average of 10 iterations, for all six EC classes.

Comparison to Nearest Neighbour Classifiers Our protein classification approach uses protein graphs and SVMs to classify proteins functionally; our first two experiments show that our function prediction approach does not randomly assign function to query proteins. The next step is to compare the performance of our system to that of other protein models and classification techniques.

To assess the quality of our approach, we compared it to methods that use

a) our protein graphs, but a Nearest Neighbour[4] classifier instead of SVMs,

b) a different protein graph model and a Nearest Neighbour classifier instead of SVMs.

Approach a), a Nearest Neighbour classifier, can be computed on a given kernel matrix, as every kernel k induces a distance. The distance between two graphs G_1 and G_2 can be defined via

$$distance(G_1, G_2) \;=\; \sqrt{k(G_1, G_1) + k(G_2, G_2) - 2 * k(G_1, G_2)}.$$

Classifying a test set protein by the Nearest Neighbour approach then means finding the protein in the training set with minimum distance to the query protein. The EC class of this nearest neighbour is the predicted class of the query protein.

The same Nearest Neighbour technique was applied to results from the SSM Protein Structure comparison server (KH03). The SSM server allows the comparison of proteins against a pre-defined set of proteins. The SSM

[4]We will write nearest neighbour as "Nearest Neighbour" when we mean the classification algorithm and "nearest neighbour" when we mean the protein most similar to a query protein.

algorithm matches secondary structure elements and then performs structural alignment along the main chain of the protein to measure the similarity between two protein structures. This service is available via *http://www.ebi.ac.uk/msd-srv/ssm/ssmstart.html*. We chose SSM as a comparison method as its model is based on secondary structure protein graphs as well, but its measurement of similarity does not use kernel methods. We defined the nearest neighbour of a query protein as the protein SSM judges to be structurally most similar to the query protein. The predicted class of the query protein is -as before- the class of its nearest neighbour.

We have to be careful about one point when comparing Nearest Neighbour and our SVM results. Our SVM 1-vs.-rest classifiers give answers of the form "class-member($=1$)" or "not class-member($=-1$)". A Nearest Neighbour approach on six EC classes gives an answer of the form "member of class X". To compare both, we have to bring their answers into the same format.

For this reason, we transferred the Nearest Neighbour results into 1-vs.-rest classification results for class X by the following procedure: If the nearest neighbour of a query protein from EC class X had EC class X, this prediction was correct; if the nearest neighbour of a query protein from class X was not in class X, the prediction was incorrect. If the nearest neighbour is member of an EC class different from X and the query protein is not in X, this prediction is correct; if the query protein is not in X, but the nearest neighbour is, the prediction is incorrect. If a nearest neighbour could not be found, the prediction is generally incorrect. The latter case can appear as SSM requires a minimal percentage of matching for Nearest Neighbour comparisons. This procedure was repeated for all six EC classes to yield results for six 1-vs.-rest classifiers.

Accuracy	**SVM-PG**	**NN-PG**	**NN-SSM**
EC 1	89.00	88.00	80.00
EC 2	88.17	86.00	79.00
EC 3	90.50	89.00	80.00
EC 4	90.00	89.00	80.00
EC 5	90.00	87.00	79.00
EC 6	91.83	36.00	79.00
AVG	89.92	79.17	79.50
SD	1.26	21.18	0.55
Sensitivity	**SVM-PG**	**NN-PG**	**NN-SSM**
EC 1	33.95	26.00	66.00
EC 2	30.09	17.00	71.00
EC 3	44.06	31.00	86.00
EC 4	40.93	31.00	73.00
EC 5	40.01	20.00	87.00
EC 6	51.10	100.00	81.00
AVG	40.02	37.5	77.33
SD	7.42	31.14	8.59
Specificity	**SVM-PG**	**NN-PG**	**NN-SSM**
EC 1	100.00	100.00	82.00
EC 2	99.80	100.00	81.00
EC 3	99.80	100.00	78.00
EC 4	99.80	100.00	81.00
EC 5	99.80	100.00	77.00
EC 6	100.00	23.00	78.00
AVG	99.87	87.17	79.50
SD	0.1	31.44	2.07

Table 2.6: Comparison of Support Vector Machine Classification on our protein graphs (SVM-PG), Nearest Neighbour (NN-PG) on our data and on SSM structure comparisons (NN-SSM). Accuracy, sensitivity and specificity are reported as average of 6-fold cross-validation for each EC class individually (AVG is average across EC classes, SD is standard deviation across EC classes).

We performed 6-fold cross-validation on 600 randomly chosen proteins, 100 per EC top level class, classifying them into EC classes 1 to 6. First, our random walk graph kernel with C-SVM classification was applied. Then we classified our protein graphs using the Nearest Neighbour method on our random walk kernel matrix. Last, we classified the proteins via Nearest Neighbour using the SSM (NN-SSM) server for protein structure comparison. We report accuracy, sensitivity and specificity results in table 2.6.

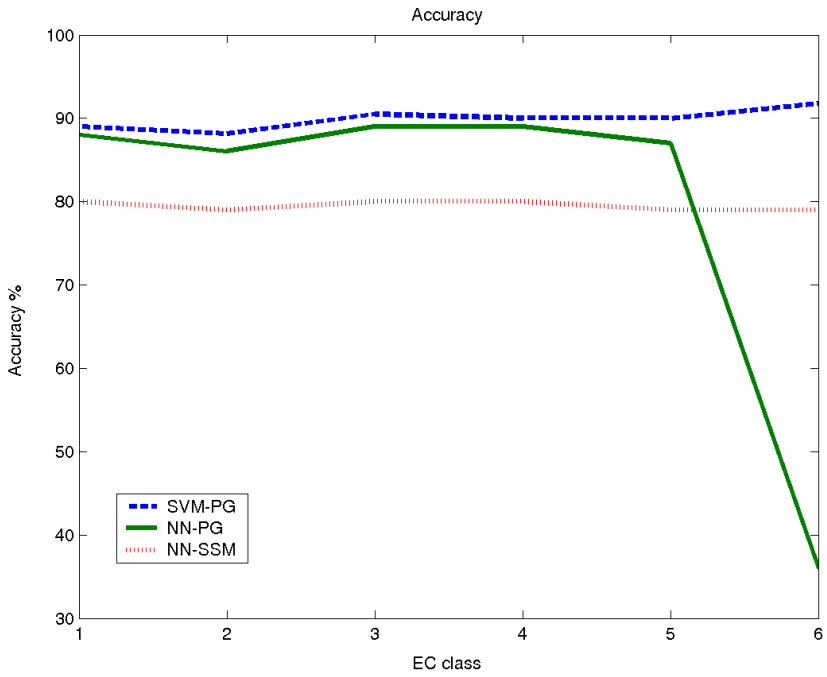

Figure 2.3: Average Accuracy of Support Vector Machine Classification on our protein graphs (SVM-PG), Nearest Neighbour (NN-PG) on our data and on SSM structure comparisons (NN-SSM).

Results Results show that overall accuracy is highest for the SVM approach (SVM-PG) on our protein graphs (figure 2.3). The Nearest Neighbour approach on our protein graphs (NN-PG) shows accuracy rates over 86% except for EC class 6, where accuracy sinks down to 36%. Nearest

Neighbour on SSM achieves 79.50% accuracy in average.

SSM does not find a nearest neighbour for every protein, as SSM requires a minimum matching threshold. If the sequences of the SSEs of two proteins match less than this threshold, the proteins are not considered similar at all.

If that threshold is set below 40%, comparison times grow extremely; the authors of SSM report this fact on their homepage. We started jobs for 0% and 10% and 30% threshold on the SSM, but they all crashed without finishing after 3 days. We obtained our results for a threshold of 50%. These jobs finished in approximately 4 hours. This threshold of 50% meant that for 19.83% of our proteins, no SSM-nearest neighbour was detected at all.

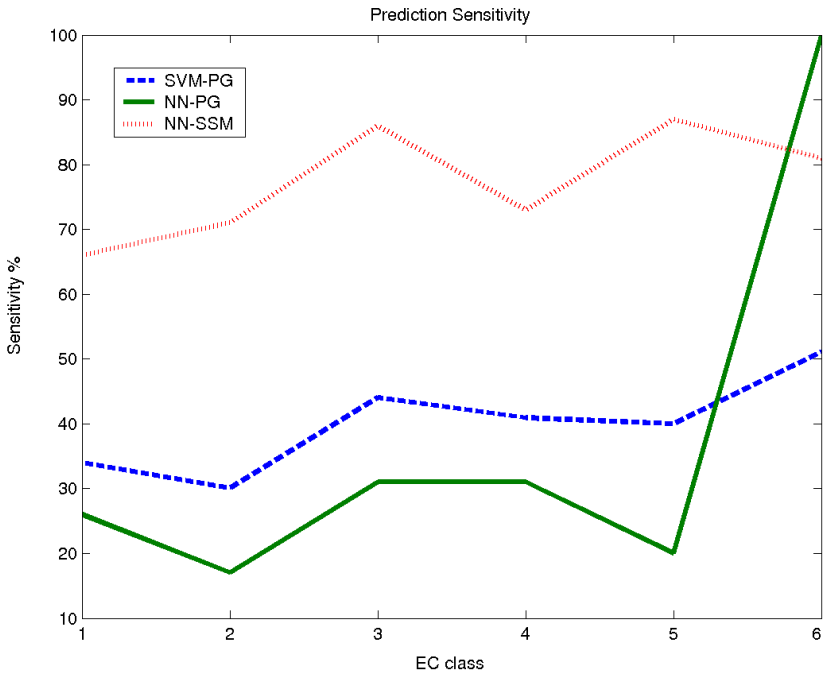

Figure 2.4: Average Sensitivity of Support Vector Machine Classification on our protein graphs (SVM-PG), Nearest Neighbour (NN-PG) on our data and on SSM structure comparisons (NN-SSM).

The accuracy results of the three methods studied are caused by great differences in specificity and sensitivity. The NN-SSM is the most sensitive

of the three techniques, finding in average 77% of all class-members in 1-vs.-rest classifications, while SVM-PG and NN-PG detect 40% and 38% in average, respectively (figure 2.4).

The higher accuracy results for SVM-PG are caused by its high specificity and the fact that in our 6-fold cross-validation with 1-vs.-rest classification the number of non-class-members is five times that of the class-members. While SVM-PG detects almost all negative samples (>99.8%), NN-PG finds 87% of non-members and NN-SSM classifies 80% of negative samples correctly.

SVM-PG turns out to be a very specific classification approach, but is much less sensitive than the nearest neighbour approach on SSM comparisons. This specificity result is encouraging, but the average specificity level of less than 50% in the experiments so far leaves much room for improvement.

The NN-PG approach assigns most proteins incorrectly to class 6. This is the reason why sensitivity for EC class 6 is so high, and specificity so low, and why sensitivity is much lower in the other five EC classes, while specificity is much higher. We checked manually why so many proteins had a nearest neighbour in EC class 6 and found out that one protein within class EC 6 was the nearest neighbour of more than half the data set. This protein consisted of two SSEs only, connected by edges. If this protein was compared to a protein with two SSEs that had identical types and similar distance, kernel values grew extremely; a possible explanation is that the random walk went back and forth, i.e. it ”tottered” between the two nodes in this small protein, resulting in large kernel values when compared to a bigger protein. To check if this is a single bad protein that ruins Nearest Neighbour on our protein graphs, we removed it from the data set and ran

NN again. The result was almost the same, but now a small protein from EC class 2 was nearest neighbour to half the dataset. We removed that protein, but then another protein took its place. This indicates that even 2-NN and 3-NN, and most likely k-NN, would not work much better on our protein graph kernel matrix due to tottering. We will discuss tottering and remedies in more detail in chapter 5.

Summary So far, we have established a secondary structure element based attributed graph model of proteins. We have designed three protein graph kernels, of which the modified random walk showed the best performance in an initial experiment. Its good performance with more than 89% accuracy could be confirmed on larger test sets and on independent evaluation sets. Compared to Nearest Neighbour approach on results by the SSM protein structure comparison server, our protein random walk graph kernel reached higher specificity, but lower sensitivity levels in enzyme class prediction. Nearest Neighbour classifiers using our protein graphs suffered from an effect called "tottering" that creates artificially high kernel values when comparing a set of proteins to a tiny protein.

In the next three chapters, we will present techniques that can enable us to improve the accuracy of our function prediction system and assess their impact on classification performance.