universität wien

DIPLOMARBEIT / DIPLOMA THESIS

Titel der Diplomarbeit / Title of the Diploma Thesis

„P-3: Paths in Protein Pockets“

verfasst von / submitted by

Michael Gostler

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Magister der Pharmazie (Mag.pharm.)

Wien, 2020 / Vienna, 2020

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 449
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Diplomstudium Pharmazie
Betreut von / Supervisor:	Univ.-Prof. Mag. Dr. Gerhard Ecker
Mitbetreut von / Co-Supervisor:	Mag. Dr. Lars Richter

Ich möchte mich sehr herzlich bei Herrn Prof. Gerhard Ecker, meinem Betreuer Dr. Lars Richter und Dr. Riccardo Martini bedanken, weil sie es mir durch ihre Unterstützung ermöglicht haben, diese Diplomarbeit zu verfassen und ich dadurch das Feld der Pharmakoinformatik kennen und schätzen lernen durfte, sowie meine Begeisterung für Mathematik und Programmieren (wieder)entdecken konnte.

Inhaltsverzeichnis

Zusammenfassung

Motivation

Die Abstraktion einer chemischen Substanz in Form von Zahlen oder Zahlenvektoren ermöglicht es, diese als Input für Machine Learning-Methoden zu verwenden. Dies wurde bereits erfolgreich verwendet, etwa, um Toxizität oder Bindungsaffinität von kleinen oder mittelgroßen Moleküle vorherzusagen. Das Ziel dieser Arbeit ist es, solch eine Abstraktion auch für Proteine zu ermöglichen, wodurch Modellierung verbessert werden könnte, da zusätzliche, relevante Informationen über Bindungsinteraktionen von Seiten der Targets, nicht nur der der Liganden, kommen könnte.

Methoden

Die Methode besteht, grob gesehen, aus zwei großen Teilschritten:
Datenvorbereitung – zuerst wurden die Bindungstaschen aus 3-dimensionalen Proteinmodellen extrahiert. Diese wurden standardisiert und durch statistische Methoden auf die relevanten funktionellen Gruppen (bzw. ihre physikochemischen Effekte, wie z.B. Aktivität als Wasserstoffbrückenakzeptor) der in ihnen enthaltenen Aminosäuren reduziert (als Punkte im 3-dimensionalen Raum) und von Artefakten gesäubert. Verwendet wurde dafür ein Datensatz aus Kinasen.
Abstraktion – die so erzeugten Punkte wurden, basierend auf ihrer (euklidischen) Entfernungen zueinander, zu Graphen verknüpft. Diese Graphen konnten zu Graph-Indices und Fingerprints reduziert und so die Proteinbindungstaschen analysiert und verglichen werden.

Resultate

Es konnten Ähnlichkeiten bei Kinasen der Src-Familie festgestellt werden, da diese aufgrund ihrer Fingerprints geclustert werden konnten. Jedoch waren solche Cluster nicht bei anderen Familien erkennbar. Deshalb bedarf dieser Ansatz weiterer Validierung, im besten Fall mit gelabelten Datensätzen, da dies einen objektiveren Maßstab für die Performance der Vorhersage bieten würde.

Abstract

Motivation

The abstraction of a chemical substance as a number or vector of numbers allows its use as input into a machine learning model. This has been done successfully for prediction of, for example, toxic effects or binding affinity to other substances of small to medium-sized molecules. The aim of this thesis is to allow this same abstraction for proteins, which could improve model performance, since relevant information regarding binding interactions could be extracted from targets, not just from ligands alone.

Methods

The method can be roughly described in two steps:

Data preparation – first, protein binding pockets were extracted from 3D protein models. These pockets were then standardized, the relevant functional groups (or rather, their underlying physicochemical effects, e.g. hydrogen bond acceptor) of their amino acids were calculated as a single spot in 3D space and artifacts were removed. This was done on a dataset of kinases.

Abstraction – the generated points were connected as a graph, based on their (Euclidean) distance to each other. These graphs could then be abstracted into indices and fingerprints, which allowed comparison of protein binding pockets.

Findings

Similarities were found in kinases of the Src family, which could be clustered based on their fingerprints. However, other closely related kinases did not show notable similarity. That is why this approach requires further validation, ideally using labelled datasets, since this would provide a more objective standard of measurement for predictive performance.

EINLEITUNG

GRAPHEN IN DER CHEMOINFORMATIK

Graphen haben in der Chemie eine sehr lange Geschichte, nämlich in Form von Molekülgraphen.

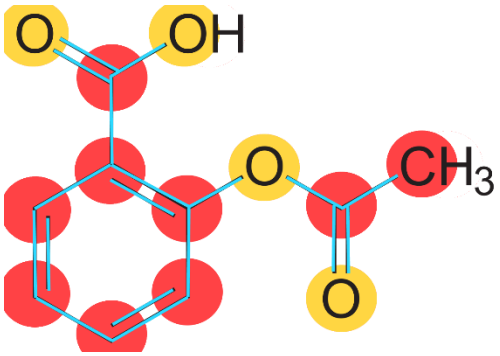


Abb. 1: Ein Molekülgraph von Acetylsalicylsäure, wobei die Knoten (engl. nodes) rot (C-Atome) bzw. gelb (O-Atome) (H-Atome werden vernachlässigt) und die Kanten (engl. edges) blau eingefärbt wurden. (Anmerkung: sämtliche in dieser Arbeit enthaltenen Bilder sind Originale des Autors)

Ein Graph kann mathematisch als $G = (V, E)$ definiert werden, wobei V eine Sammlung an Knoten (V wie „Vertices“) und E eine Sammlung von Kanten (E wie „Edges“) sind und jede Kante aus E mit zwei unterschiedlichen Knoten aus V assoziiert ist. Solche Graphen können, wie teilweise im oberen Bild zu sehen ist, noch komplexer gemacht werden, indem Mehrfachbindungen, unterschiedliche Knoten-Arten (z.B. verschiedene Atome), Direktionalität von Kanten (üblicherweise durch Darstellung als Pfeil) oder variable Kantenwerte (jede Kante wird zusätzlich mit einer Zahl assoziiert, die z.B. die Entfernung zweier Knoten zueinander angibt) eingebaut werden.

In der Chemoinformatik finden Graphen grundsätzlich die gleiche Anwendung: die abstrakte Repräsentation von Molekülen. Neben Graphen gibt es für diesen Zweck aber schon viele weitere Methoden, z.B. SMILES (Simplified Molecular Input Line Entry System)[1] oder InChIs (International Chemical Identifiers)[2], die, da sie relativ simple Zeichenketten sind, für Computer weitaus einfacher zu verarbeiten sind (z.B. für Speicherung und Datenbanksuchen), als Molekülgraphen (eine Abstraktion, die besser für Menschen geeignet ist).

Warum werden solche Molekülgraphen also dennoch in der Chemoinformatik verwendet? Unter Anderem, weil man mit Graphen Output (sog. Fingerprints) generieren kann, der für Machine Learning-Modelle (z.B. Random Forests, Support Vector Machines oder Neural Networks) eingesetzt werden kann. Solch ein Output ist z.B. der Extended Connectivity Fingerprint (ECFP), welcher für structure-activity modelling (d.h., Vorhersage von Molekülaktivität über seine Struktur) erfunden wurde.[3]

Über diese Methode

Es wurde hier ebenfalls versucht, über Graphen Fingerprints herzustellen. Jedoch unterscheiden sich sowohl die Grundlage der Graphen (was abstrahiert wird), als auch die produzierten Fingerprints von den in der Chemoinformatik gängigen. Hier wurden nämlich nicht Moleküle, sondern 3-dimensionale Proteinbindungstaschen zu Graphen verarbeitet und Fingerprints über deren verschiedene Nachbarschaften von „Feature Types“ (die Aktivität (lipophil, H-Brückendonor/-akzeptor, positiv/negativ ionisiert, arylisch, π-Aryl-Interaktion) der in den Aminosäuren der Proteinbindungstaschen vorliegenden funktionellen Gruppen) erzeugt.
Proteinbindungstaschen sind interessant, da ihre spezielle Faltung räumliche Nähe strukturell normalerweise weit auseinandergelegener Aminosäuren erzeugt, wodurch dreidimensionale Muster aus verschiedenen funktionellen Gruppen entstehen, die komplementär (bzw. komplementär genug) zu denen ihrer Liganden sind, sodass sie diese binden können. Wäre es möglich, diese Strukturen sinnvoll zu charakterisieren, könnte man z.B. über die Kombination der Fingerprints von Liganden und Targets bessere Voraussagen über Interaktionen treffen.
Um die Graphen der Proteinbindungstaschen weiter charakterisieren zu können, wurden außerdem sogenannte Graph Indices berechnet. Das sind Zahlen, welche auf der Basis von Graph-Eigenschaften berechnet werden (mathematisch gesehen: $f(Graph) = Index$) und unter Umständen mit Eigenschaften der zugrundeliegenden Strukturen korrelieren können, was bei molecular graphs bereits nachgewiesen wurde[4].

Kurz gesagt ist das Ziel dieser Arbeit also, die bereits etablierte Methodik der Verarbeitung von molecular graphs auf Proteinbindungstaschen umzulegen und in Zukunft auszuschöpfen, etwa, um das in silico-Screening von Arzneistoffen mit menschlichen Proteinen als Target zu unterstützen.

Existierende Ansätzen

Das computergestützte Wirkstoffdesign ist natürlich kein gänzlich neues Feld, weshalb es bereits andere Ansätze gibt, die in eine ähnliche Richtung gehen wie der hier beschriebene.

KRIPO

KRIPO (Key Representation of Interaction in Pockets)[5] wurde konzipiert, um bioisosteren Ersatz für Liganden für bestimmte Ziel-Proteine zu finden. Bioisostere sind Moleküle mit gleicher Anordnung der Atome aber unterschiedlicher Elektronenkonfiguration (z.B. durch Austausch einer OH- und einer Cl-Gruppe in einem Molekül), die eine vergleichbare in-vivo-Wirksamkeit haben.

Bei dieser Methode werden als Vorbereitung Feature Origins innerhalb einer Proteinbindungstasche auf Basis geometrischer Überlegungen und Berechnungen im Raum platziert. Die Pharmakophore werden dann mithilfe einer in-silico Platzierung von Liganden und Ligandenbruchstücken in den Proteinbindungstaschen generiert.
Dazu werden zunächst sämtliche Features, die mehr als 2,5Å (1Å = 10^{-10}m) zu einem beliebigen Teil der Liganden und Ligandenbruchstücke entfernt liegen, für die spezifische Fingerprint-Generierung ausgeschlossen. Aus den übrig gebliebene Features werden Fingerprints anhand des räumlichen Abstandes aller Features zu allen anderen berechnet.
Die Slots (Felder, die entweder den Wert 1 oder 0 haben können) der durch KRIPO erzeugten Fingerprints sind Permutationen (mit Wiederholung)[6] aus allen möglichen Feature-Permutationen (ebenfalls mit Wiederholung) und verschiedenen Klassen, in die die Abstände der Features eingeteilt werden.
Beispiel-Schema für ein Bit, das aus zwei Features besteht: {f1}{f2}{d12}, wobei f1 und f2 die Feature Types der beiden beteiligten Feature Origins und d12 die Distanzklasse der Distanz (in Å), die die beiden zueinander entfernt sind.

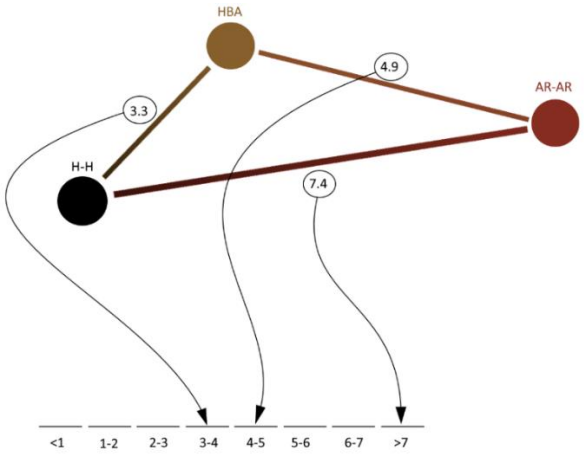


Abb. 1: Fingerprint-Generierung in KRIPO. Folgende Bits wären hier abzulesen:
{HBA}{H-H}{3-4}, {HBA}{AR-AR}{4-5}, {AR-AR}{HBA}{4-5},
{AR-AR}{H-H}{>7}, {H-H}{AR-AR}{>7}, {H-H}{HBA}{3-4}

Die Fingerprints wurden benutzt, um Bioisostere zu vergleichen (s. Kapitel „Vergleich der Bitvektoren“). Je höher der berechnete Koeffizient, desto höher die vorausgesagte Ähnlichkeit und desto höher die Wahrscheinlichkeit eines sinnvollen bioisosteren Ersatzes.

Mit KRIPO wurden für Liganden zweier verschiedener Proteine (eine MAP-Kinase- und ein Thrombin-Inhibitor) gefunden. Das zeigt, dass es möglich ist, Proteine zu abstrahieren und aus diesen Abstraktionen bedeutsame Schlüssen ziehen zu können.

FLAP

FLAP (Fingerprints for Ligands And Proteins)[7] erzeugt Fingerprints auf eine gänzlich andere Art, verglichen mit KRIPO.
Ein Protein (wenn man einen Fingerprint für eine Proteinbindungstasche erzeugen möchte) wird in einem Gitter platziert und an jedem Gitterpunkt wird die durch eine bestimmte Bindungsinteraktion (= Feature Type (H-H, HBA, etc.)) frei werdende Energie mathematisch berechnet[8].

An energetisch günstigen Positionen im Gitter, an denen durch eine Interaktion eine bestimmte Menge Energie frei werden würde (über einem festgelegten Threshold), wird angenommen, dass eine Interaktion eines bestimmten Types stattfinden wird. Diese wird als Punkt im Raum eingetragen. Da solche Interaktionen stark geclustert sind (d.h., wenn an einem Punkt im Raum eine ausreichend starke Interaktion stattfinden würde, ist die Wahrscheinlichkeit sehr hoch, dass z.B. 1Å entfernt davon ebenfalls eine solche Interaktion stattfinden würde), entstehen „Punktewolken“, aus denen auf verschiedene Arten Fingerprints erzeugt werden, z.B. wie bei KRIPO über Distanz-Feature Type-Kombinationen.

GRAPH-INDICES

Für alle in dieser Arbeit generierten Graphen wurden folgende Indices berechnet:

- Wiener- und Hyper-Wiener-Index[9]
- Randic-, erster Zagreb-, F- und Narumi-Katayama-Index[10] [11] [12] [13]
- Szeged-, Padmakar-Ivan und zweiter Zagreb-Index[14] [15]
- Balaban-J-Index[16]

Die Gruppierung erfolgt hier nach den zur Berechnung der Indices verwendeten Graph-Attributen.

WIENER- UND HYPER-WIENER-INDEX

Die Berechnung dieser beiden Indices beruhet auf den kleinstmöglichen Distanzen der einzelnen Knotenpunkten eines Graphen zueinander.
Der Wiener-Index kann auf mehrere Arten berechnet werden, eine gängige Formel dafür ist:

$$W(G) = \frac{1}{2} \sum_{v_i, v_j \in V(G)} d_G(v_i, v_j)$$

W(G)…Wiener-Index

V(G)…Knotenpunkte des Graphen G

$d_G(v_i, v_j)$…kürzeste Distanz zwischen Knoten v_i und v_j

Die Formel sagt aus, dass man die kleinsten Distanzen aller möglichen Knotenpunkt-Paar-Kombinationen aufsummieren und das Ergebnis halbieren muss. Die Halbierung erfolgt, damit Wege nicht doppelt gezählt werden (z.B. $d(v_1, v_2)$ nochmals als $d(v_2, v_1)$).

Der Hyper-Wiener-Index (WW) ist dem Wiener-Index sehr ähnlich. Er berechnet sich als

$$WW(G) = \frac{1}{2} * \left(\sum_{v_i, v_j \in V(G)} d_G(v_i, v_j) + \sum_{v_i, v_j \in V(G)} d_G^2(v_i, v_j) \right)$$

RANDIC-, ERSTER ZAGREB-, F- UND NARUMI-KATAYAMA-INDEX

Diese Gruppe an Indices basiert auf dem Grad der einzelnen Knoten in einem Graph. Der Grad eines Knotens ist die Anzahl der an ihm hängenden Verbindungen. Man kann ihn berechnen, indem man für jeden Knotenpunkt die jeweilige Zeile oder Spalte in der adjacency matrix (eine Matrix, die angibt, welche Knoten in einem Graph miteinander verbunden sind) aufsummiert.

Wenn d(v) der Grad eines Knoten v in einem Graph ist, errechnen sich für diesen Graphen der Randic- (R(G)), erste Zagreb (Z_2(G))-, F- (Z_3(G)) und Narumi-Katayama-Index (NK(G)) als

$$R(G) = \sum_{uv \in E(G)} \frac{1}{\sqrt{d(u)\, d(v)}}$$

$$\mathrm{Z}_2(\mathrm{G}) = \sum_{\mathrm{v} \in \mathrm{V(G)}} d_v^2$$

$$\mathrm{Z}_3(\mathrm{G}) = \sum_{\mathrm{v} \in \mathrm{V(G)}} d_v^3$$

$$\mathrm{NK(G)} = \prod_{v} d_v(G)$$

Szeged-, Padmakar-Ivan- und zweiter Zagreb-Index

Der Szeged-Index basiert auf dem Wiener-Index und ergibt für einen Graphen ohne Zyklen den gleichen Wert (ein Zyklus ist ein Ringschluss, wodurch ein Pfad entsteht, der den selben Anfangs- und Endpunkt haben kann), der Padmakar-Ivan-Index ist ein „Abkömmling“ des Szeged-Index.
Der Szeged- (Sz), Padmakar-Ivan- (PI) und der zweite Zagreb-Index (M_2) errechnen sich als

$$Sz(G) = \sum_{e \in E(G)} n_1(e|G) * n_2(e|G)$$

$$PI(G) = \sum_{e \in E(G)} n_1(e|G) + n_2(e|G)$$

$$M_2 = \sum_{edges} d_i * d_j$$

$e \in E(G)$ bzw. Kanten sind alle Verbindungen zwischen zwei Knoten im Graphen.
n_1 und n_2 sind die beiden durch eine gegebene Verbindung verknüpften Bindungspartner.
n_1(e|G) und n_2(e|G) geben die Anzahl der Knoten im Graphen wider, die näher an n_1 bzw. n_2 liegen.

Balaban-J-Index

Der Balaban-J-Index berechnet sich als

$$J(G) = \frac{m}{\mu + 1} \sum_{uv \in E} \frac{1}{\sqrt{D_u D_v}},$$

wobei m die Anzahl der Bindungen und μ die Anzahl der Ringe im Graph (berechnet als μ = m - n, wobei n die Anzahl der Knotenpunkte im Graph darstellt) bezeichnen.
D_u und D_v sind die Summen der kürzesten Verbindungen zu allen anderen Knotenpunkten ausgehend von u und v, wobei u und v immer benachbarte Knotenpunkte sind (die Knoten der Kante uv).

Vergleich der Bitvektoren

Eine schnelle Methode, um die aus Graphen generierten Bitvektoren zu verwenden, ist, ihre Ähnlichkeit zu vergleichen, etwa um Rückschlüsse auf Ähnlichkeit ihrer Bindungspartner zu ziehen. Dafür können der Tanimoto- und der modifizierte Tanimoto-Koeffizient[17] verwendet werden.

Tanimoto-Koeffizient

Der Tanimoto-Koeffizient wird berechnet als

$$S_T = \frac{c}{a + b - c}$$

Er drückt die die Ähnlichkeit zweier Bitvektoren als Zahl zwischen 0 bis 1 aus, wobei 1 die komplette Identität (BV1 = BV2) und 0 ein vollständiges Mismatch bedeuten.
a und b sind dabei die Summen der beiden Bitvektoren (= Anzahl der vorhandenen Muster) und c ist die Anzahl an Stellen, an denen sowohl a als auch b gleich 1 sind.

Modifizierter Tanimoto-Koeffizient

Der modifizierte Tanimoto-Koeffizient stellt eine Variation des Tanimoto-Koeffizienten dar. Er wurde entwickelt, um den „length bias“ des ursprünglichen Tanimoto-Koeffizienten (dieser tendiert dazu, bei länger werdenden Bitvektoren immer kleiner zu werden) auszugleichen[18] und berechnet sich als

$$T_{MT} = \left(\frac{2 - \rho_0}{3}\right) * S_T + \left(\frac{1 + \rho_0}{3}\right) * S_{T0}$$

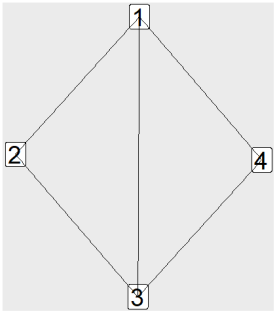
ρ_0 ist dabei die durchschnittliche Bit-Dichte des zugrundeliegenden Datensatzes, also, wie viele 1er ein Bitvektor im Verhältnis zu seiner Länge im Schnitt besitzt. S_{T0} ist der sogenannte „inverse“ Tanimoto-Koeffizient, der berechnet wird als

$$S_{T0} = \frac{n - a - b + c}{n - c}$$

n ist hier die Länge einer der beiden Bitvektoren inklusive 0er (da beide gleich lang sein müssen, ist es egal, welchen man verwendet).

Index-Berechnung anhand eines einfachen Beispiel-Graphen

Graph und notwendige Variablen




```
> distances(g)
     [,1] [,2] [,3] [,4]
[1,]    0    1    1    1
[2,]    1    0    1    2
[3,]    1    1    0    1
[4,]    1    2    1    0
```

```
> get.edgelist(g)
     [,1] [,2]
[1,]    1    2
[2,]    1    3
[3,]    1    4
[4,]    2    3
[5,]    3    4
```

```
> length(E(g))
[1] 5
> length(V(g))
[1] 4
```

```
> degree(g)
[1] 3 2 3 2
> d_sums
[1] 3 4 3 4
```

Abb. 2: Beispiel-Graph und für Berechnung der Graph-Indices notwendige Graph-Variablen. „d_sums“ sind die Summen der Distanzmatrix-Spalten.

Wiener- und Hyper-Wiener-Index

$$\frac{(0+1+1+1)+(1+0+1+2)+(1+1+0+1)+(1+2+1+0)}{2} = \frac{3+4+3+4}{2} = 7$$

$$\frac{(0+2+2+2)+(2+0+2+6)+(2+2+0+2)+(2+6+2+0)}{2} = \frac{6+10+6+10}{2} = 16$$

```
wieners <- function(g){
  d <- distances(g)
  wiener <- sum(d)/2
  hyper_wiener <- sum(d + d^2)/2
  return(c(wiener, hyper_wiener))
}
```

```
> wieners(g)
[1]  7 16
```

Abb. 3: Funktion zur Berechnung des Wiener- und Hyper-Wiener-Index und Ergebnisse für den Beispiel-Graph

Randic-, erster Zagreb-, F- und Narumi-Katayama-Index

Randic = $0{,}57735 + 0{,}70711 + 0{,}57735 + 0{,}70711 = \sim 2{,}57$

1. Zagreb = $9 + 4 + 9 + 4 = 26$

F = $27 + 8 + 27 + 8 = 70$

NK = $3 * 2 * 3 * 2 = 36$

```
randic_zagreb1_f_nk <- function(g){
  deg <- degree(g)
  randic <- sum(deg**-0.5)
  zagreb <- sum(deg**2)
  f <- sum(deg**3)
  nk <- prod(deg)
  return(c(randic, zagreb, f, nk))
}
```

```
> round(randic_zagreb1_f_nk(g), 2)
[1]  2.57 26.00 70.00 36.00
```

Abb. 4: Funktion zur Berechnung des Randic-, (1.) Zagreb-, F- und Narumi-Katayama-Index und Ergebnisse für den Beispiel-Graph.

Szeged- und zweiter Zagreb-Index

1) 1 und 4 sind näher an 1, 2 nur zu sich selbst
2) 1 und 3 sind sich nur selbst am nächsten
3) 1 und 2 sind näher an 1, 4 nur zu 4
4) 2 ist nur zu sich selbst am nächsten, 3 ist näher zu 3 und 4
5) 4 ist nur zu sich selbst am nächsten, 3 zu 2 und 3

Szeged-Index = $(2 * 1) + (1 * 1) + (2 * 1) + (1 * 2) + (1 * 2) = 9$

Padmakar-Ivan-Index = $(2 + 1) + (1 + 1) + (2 + 1) + (1 + 2) + (1 + 2) = 14$

zweiter Zagreb-Index = $(3 * 2) + (3 * 3) + (3 * 2) + (2 * 3) + (3 * 2) = 33$

```
szeged_padivan_zagreb2 <- function(g){
  szgd <- 0
  pdvn <- 0
  zgrb <- 0
  edgl <- get.edgelist(g)
  for(i in seq_along(edgl[,1])){
    row <- edgl[i,]
    node_one <- row[1]
    node_two <- row[2]
    path_one <- shortest.paths(g, node_one)
    path_two <- shortest.paths(g, node_two)
    degree_one <- degree(g, node_one)
    degree_two <- degree(g, node_two)
    number_one <- sum(path_one < path_two)
    number_two <- sum(path_one > path_two)
    szgd <- szgd + (number_one*number_two)
    pdvn <- pdvn + (number_one+number_two)
    zgrb <- zgrb + (degree_one*degree_two)
  }
  return(c(szgd, pdvn, zgrb))
}
```

```
> szeged_padivan_zagreb2(g)
[1]  9 14 33
```

Abb. 5: Funktion zur Berechnung des Szeged-, Padmakar-Ivan- und zweiten Zagreb-Index und Ergebnisse für den Beispiel-Graph.

Balaban-J-Index

Erster Teil der Formel

$$m = 5$$
$$n = 4$$
$$\mu = m - n = 1$$
$$\frac{q}{\mu + 1} = \frac{5}{2} = 2{,}5$$

Zweiter Teil der Formel

$$\frac{1}{\sqrt{3 * 4}} + \frac{1}{\sqrt{4 * 3}} + \frac{1}{\sqrt{3 * 4}} = \frac{3}{\sqrt{12}}$$

Balaban-J-Index

$$2{,}5 * \frac{3}{\sqrt{12}} = \sim 2{,}165$$

```
balaban_j <- function(g){
  q <- length(E(g))
  n <- length(V(g))
  u <- q - n + 1
  part_one <- q/(u+1)

  d <- distances(g)
  d_sums <- apply(d, 1, sum)
  part_two <- 0
  for(i in 1:(length(d_sums)-1)){
    part_two <- part_two + 1/sqrt(d_sums[i]*d_sums[i+1])
  }

  j <- part_one * part_two
  return(j)
}
```

```
> balaban_j(g)
[1] 2.165064
```

Abb. 6: Funktion zur Berechnung des Balaban-J-Index und Ergebnis für den Beispiel-Graph

Berechnung der Tanimoto-Indices

```
> bv1
[1] 1 0 1 1 0 0
> bv2
[1] 1 1 0 1 1 0
```

Die drei verschiedenen Tanimoto-Koeffizient der gezeigten Bitvektoren (bv1 und bv2) würden sich wie folgt berechnen

(ρ_0 muss angenommen werden, weil es keinen zugrundeliegenden Datensatz gibt. Deshalb wird für dieses Beispiel einfach die mittlere Bit-Dichte der beiden gegebenen Vektoren verwendet, welche $\frac{Summe(bv1) + Summe(bv2)}{Länge(bv1) + Länge(bv2)}$ ist)

$$\rho_0 = \frac{3 + 4}{6 + 6} = \frac{7}{12} = \sim 0{,}583$$
$$n = 6$$
$$a = 3$$
$$b = 4$$
$$c = 2$$

$$S_T = \frac{2}{3+4-2} = \frac{2}{5} = 0{,}4$$

$$S_{T0} = \frac{6-3-4+2}{6-2} = \frac{1}{4} = 0{,}25$$

$$S_{TM} = \left(\frac{2-0{,}583}{3}\right) * 0{,}4 + \left(\frac{1+0{,}583}{3}\right) * 0{,}25 = 0{,}1889 \ + \ 0{,}1319 \ = {\sim}0{,}3208$$

```
tanimotos <- function(bv1, bv2, p = NA){
  if(!is.numeric(bv1) || !is.numeric(bv2)){
    stop("input needs to be numeric")
  }else if(!all(bv1 == 0 | bv1 == 1) | !all(bv2 == 0 | bv2 == 1)){
    stop("input needs to be two bitvectors")
  }else if(length(bv1) != length(bv2)){
    stop("bitvectors need to be the same length")
  }else{
    n <- length(bv1)
    x <- which(bv1 == 1)
    y <- which(bv2 == 1)

    A <- sum(bv1)
    B <- sum(bv2)
    C <- sum(x%in%y)

    if(is.na(p)) p <- (A+B)/(length(bv1)+length(bv2))
    t <- C/(A+B-C)
    t_inv <- (n-A-B+C)/(n-C)
    t_mod <- ((2-p)/3)*t + ((1+p)/3)*t_inv
    tanims <- c(t, t_inv, t_mod)
    names(tanims) <- c("Tanimoto", "inverse Tanimoto", "modified Tanimoto")
    return(tanims)
  }
}
```

```
> tanimotos(bv1, bv2)
        Tanimoto  inverse Tanimoto modified Tanimoto
       0.4000000         0.2500000         0.3208333
```

Abb. 7: Funktion zur Berechnung der Tanimoto-Indices und Ergebnisse für die Beispiel-Bitvektoren.

Materialien

Datensatz

Der Datensatz umfasst kinetische Daten aus SPR Messungen von Kinasen und stammt aus der Studie „Binding Kinetics Survey of the Drugged Kinome“[19].

3D-Modelle der im Datensatz genannten Proteine

Die verwendeten 3D-Modelle wurden mithilfe von uniprot.org aus der Protein Data Bank[20] („PDB“) heruntergeladen und mit MOE strukturbereinigt.

Methoden zur Verarbeitung der 3D-Modelle

Die Interaktionsenergien der vorbereiteten 3D-Modelle wurden mithilfe von GRAIL[21] berechnet und durch Gaussian Mixture Modeling (GMM) geclustert und auf die Mittelpunkte der ellipsoid-förmigen Cluster - ab hier als „Feature Origins“ bezeichnet - reduziert.
Dann wurden die Daten mithilfe statistischer Methoden vorgefiltert, um Ausreißer zu entfernen.

Werkzeuge der Datenverarbeitung

Die zur Verarbeitung der Daten (weitere Filterung der Daten, Berechnung der Graphen, Selektion des größten connected component, …) verwendete Programmiersprache war R in der Version 3.6.1[22], inklusive folgender Module:
tidyverse, TSDist, igraph, arrangments, progress, readxl, hashmap

METHODEN

DATENVORBEREITUNG

BESORGUNG DER 3D-MODELLE

Die 3D-Modelle wurden aus der PDB („Protein Data Bank“) entnommen. Da jedoch die in der PDB verwendeten PDB-IDs nicht für Protein-Namen, sondern für bestimmte Einträge codieren und es für jedes Protein mehrere Einträge (und damit mehrere PDB-IDs) geben kann, musste zuerst für jedes Protein ein PDB-Eintrag selektiert werden.
Dafür wurde zuerst anhand des Namens der jeweiligen Proteine bei uniprot.org[23] die Uniprot-ID der humanen Variante herausgesucht.
Mit den Uniprot-IDs wurden danach für jedes Protein aus der PDB alle Einträge gesucht, die eine 3D-Struktur dieses Proteins codierten und nach bestimmten Regeln bzw. Guidelines eine PDB-ID pro Protein ausgesucht.

Die Regeln/Guidelines, nach denen vorgegangen wurde:

1) Die Proteine durften nicht phosphoryliert sein
2) Es durfte keine Mutation in der Bindungstasche vorliegen
3) Nur Sub-Domains des gewünschten Proteins waren erlaubt
4) Die Proteine mussten in aktiver Konformation, das heißt mit gebundenem Ligand, vorliegen
5) Der Ligand durfte nicht ADP sein
6) Der Ligand durfte kein Protein oder anderes Makromolekül sein

Unter den verbliebenen Kandidaten wurde der Eintrag mit der besten Auflösung in der 3D-Struktur ausgewählt, in eine Liste eingetragen und das 3D-Modell für weitere Verarbeitungsschritte heruntergeladen.

	A	B	C
1	**Kinase**	**Uniprot_ID**	**PDB_ID**
2	ABL	P00519	5HU9
3	AKT1	P31749	4GV1
4	ALK	Q9UM73	4Z55
5	AurA	O14965	6C2T
6	AXL	P30530	5U6B
7	BRAF	P15056	5ITA
8	BTK	Q06187	5P9J
9	CDK2	P24941	6Q4G
10	CDK9	P50750	3BLR
11	EGFR	P00533	3POZ
12	FAK	Q05397	4I4E
13	FGFR1	P11362	5EW8
14	FGFR3	P22607	---
15	FGFR4	P22455	4XCU
16	FLT3	P36888	5X02
17	FMS	P07333	2I1M

Abb. 8: Einige Beispiele für Protein-Namen und ihre Uniprot- & PDB-Äquivalente.
Eingekreist: manches Mal gibt es keine PDB-IDs, die alle Kriterien erfüllen. Diese Proteine können nicht verarbeitet werden.

Verarbeitung der 3D-Modelle

Die 3D-Modelle wurden mithilfe von MOE anhand der PDB-ID heruntergeladen und über den Structure Preparation Wizard mit den Default-Einstellungen präpariert, um etwaige Fehler auszugleichen und den Datensatz zu standardisieren.
Weiters wurden oligomere Strukturen zu Monomeren reduziert, da sich die hier beschriebene Abstraktionen nur auf einzelne Bindungstasche beziehen.

Generierung der Feature Origins

Die Daten wurden von meinem Betreuer Dr. Lars Richter und von Dr. Riccardo Martini weiterverarbeitet.
Zuerst wurden mithilfe von GRAIL die Interaktionsenergien der verschiedenen Feature Types innerhalb der vorliegenden Proteinmodellen berechnet. Dann wurden die daraus entstehenden Punktewolken durch Gaussian Mixture Modeling (kurz GMM, eine unsupervised Machine Learning-Methode, die Datensätze clustert) geclustert.
Die dadurch herausgerechneten Cluster sind geometrische Körper (Ellipsoide) in einem dreidimensionalen Raum, weshalb deren Mittelpunkte berechnet und als Vektor mit x-, y- und z-Koordinaten ausgegeben werden konnten. Diese Mittelpunkte stellen die bereits erwähnten Feature Origins dar, welche die Grundlage der Graph-Generierung waren.

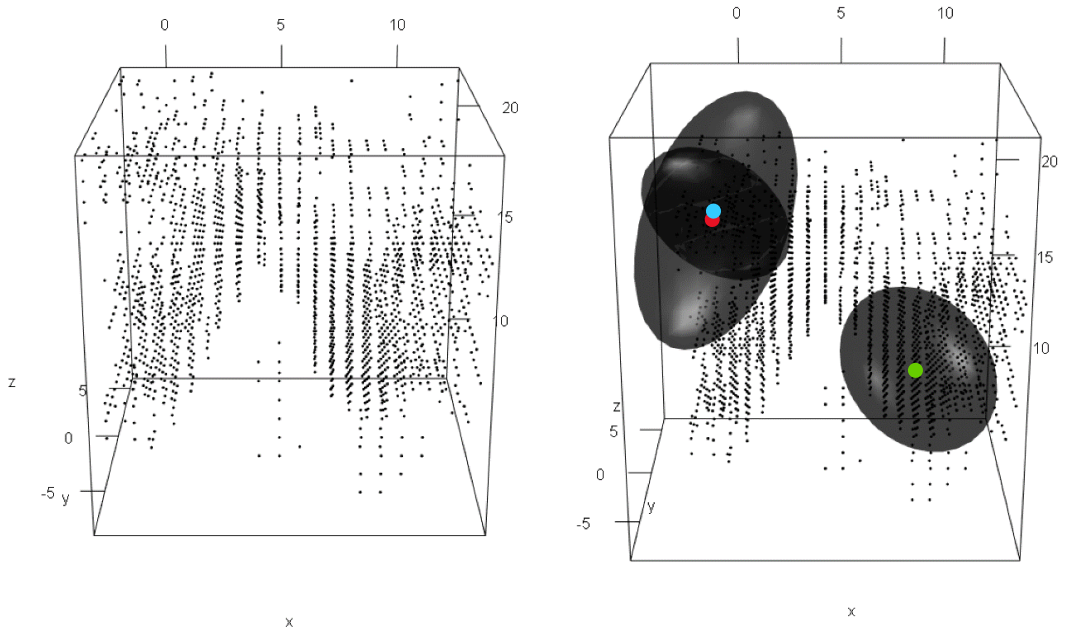


Abb. 9: Punktewolken erzeugt durch GRAIL (links) und beispielhaftes Clustering und Clusterzentren (farbige Punkte), berechnet durch GMM (rechts)

Filterung der Daten

Die gebildeten Cluster wurden aufgrund ihrer Dichte („feature density“ = aus wie vielen Punkte wurde der Cluster herausgerechnet im Verhältnis zu seinem Volumen) und buriedness („[Buriedness] Is a measure based on the proximity of the amino acid to the surface of the protein.“[24], d.h. wie weit von der Oberfläche des Proteins lagen die Cluster entfernt) gefiltert. Dazu wurde die sogenannte „Median Absolute Deviation“[25], kurz MAD, verwendet.

Datenverarbeitung

Dieser Schritt wurde eingesetzt um weitere Feature Origins herauszufiltern, damit nur mehr Punkte im Datensatz enthalten waren, deren Ursprungs-Cluster eine akzeptable Form hatten und die mit mindestens einem anderen Punkt verbunden waren.

Artefakt-Entfernung und Extraktion des größten „Connected Component“

Da die energetische Berechnung durch GRAIL in einem würfelförmigen Raum stattgefunden hat, das Innere einer Proteinbindungstasche jedoch meistens nicht würfelförmig ist, gab es fast immer Artefakte, also Feature Origins, die außerhalb des eigentlich interessanten Bereichs (z.B. in den Ecken des Würfels) lagen. Diese Artefakte könnten die Fingerprint- und Index-Berechnung stören, weshalb es notwendig war, vorher den größten connected component zu extrahieren, wodurch unverbundene Punkte herausgefiltert wurden.

Der Threshold

Die Grundlage der Graph-Generierung war der sog. „Threshold“ (engl. Schwelle), ein Abstand in Å (Angström = 10^{-10}m), der beliebig festgelegt werden kann. Befinden sich zwei Feature Origins unter einem gewählten Threshold, gelten sie als verbunden, das heißt, zwischen ihnen verläuft eine Kante. Über Trial-and-Error hat sich für den verwendeten Datensatz ein Wert von 3,5 bis 4 Å als ideal herauskristallisiert.
Bei einem Threshold <3,5 ergaben sich meistens sehr viele kleine Gruppen, wodurch bei Extraktion des größten connected component ein Großteil der Feature Origins verworfen worden wäre. Bei einem Threshold >4 waren zunehmend alle Input-Punkte im größten connected component enthalten, wodurch sich wahrscheinlich vermehrt Artefakte, die für die analysierte Bindungstasche von geringer Relevanz sind, im Graph befinden würden. Weiters führt ein zu hoher Threshold zu einer verstärkten Vernetzung der Feature Origins untereinander, wodurch Information über die Entfernung der Knotenpunkte zwischen einander verloren gehen würde.

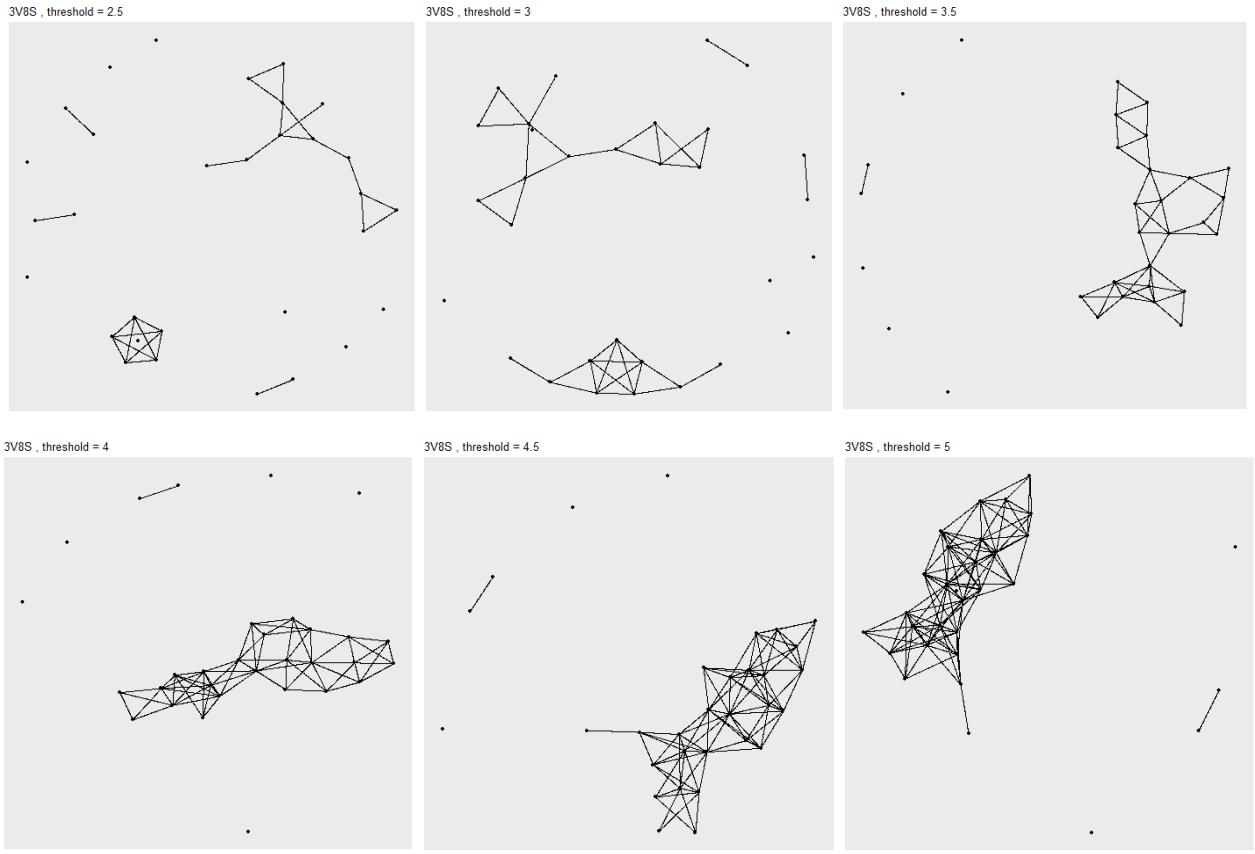


Abb. 10: Graphen generiert aus dem selben Protein (PDB-ID 3V8S = ROCK1, eine Serin-/Threoninkinase), mit Thresholds 2.5 – 3.0 - 3.5 – 4.0 – 4.5 – 5.0 (v.l.o.n.r.u). Es wurde ein Layout namens „with_kk“ verwendet, das versucht, Knotenpunkte in bestimmten geometrischen Figuren darzustellen und sie räumlich möglichst gleichmäßig voneinander zu trennen, weshalb die Abstände auf den Bildern nicht die tatsächliche räumliche Positionierung widerspiegeln und die gezeigten Gruppen nicht immer an der gleichen Position sind.

Filtern der Feature Origins nach ihren Ursprungs-Ellipsoiden

Eine weitere Quelle an Artefakten ist die Clustering-Methode. Einige der durch GMM fälschlicherweise zu Clustern zusammengelegten Punkte wurden bereits im Vorfeld entfernt (mithilfe der feature density und buriedness), allerdings wurde ein weiteres Kriterium angewendet, nämlich die Form der Ellipsoide, aus denen die Feature Origins herausgerechnet wurden: mit GMM erzeugte Ellipsoide sind nicht immer gleichmäßig geformt. Sie können in der Länge ihrer drei Hauptachsen variieren und deshalb kugel-, ei-, teller- oder nadelförmig sein (s. folgende Abb.).

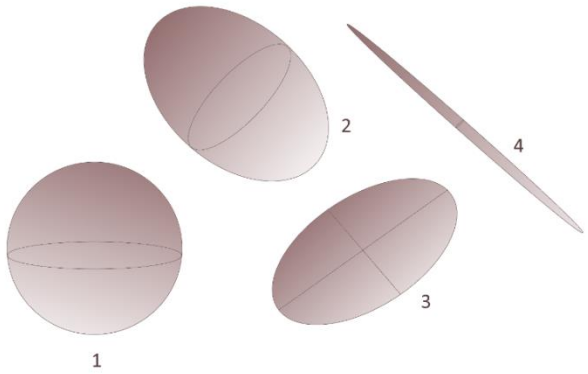


Abb. 11: Darstellung der möglichen Form-„Archetypen“:
1 – Kugel | 2 – Ellipsoid | 3 – Teller | 4 – Nadel

Es wurde die Annahme getroffen, dass teller- und nadelförmige Ellipsoide Artefakte darstellen, da sie im 3-dimensionalen Raum kaum Volumen besitzen und aus diesem Grund wahrscheinlich keine realen Interaktionseffekte beschreiben.
Um sie herauszufiltern wurde eine Maßzahl, genannt „Shape Index“ (S), verwendet, der mit drei Parametern die Form eines Ellipsoids als Skalar ausdrückt:

$$S = \frac{a^2}{b^2 * c^2} * \left(\frac{1}{3}\right)^2$$

Dabei sind a, b und c die Anteile der Längen der drei ein Ellipsoid beschreibenden Eigenvektoren/Hauptachsen an der Summe ihrer Längen, wobei gilt, dass a ≥ b ≥ c.

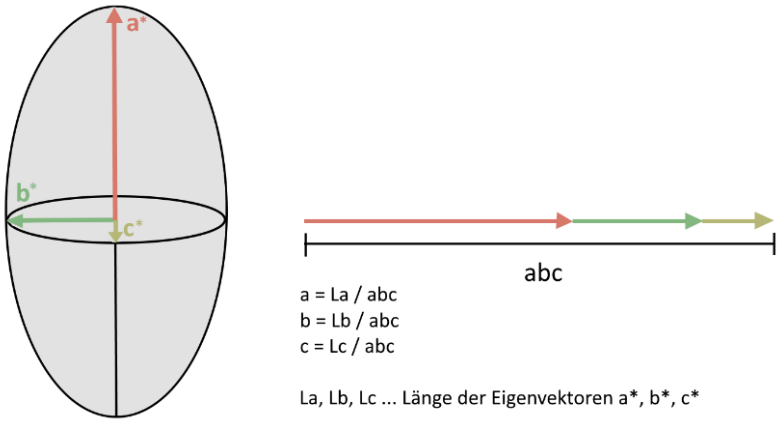


Abb. 12: Ein Ellipsoid mit eingezeichneten Eigenvektoren

Durch diese Regeln ergibt sich ein begrenzter Verteilungsraum für a, b und c (s. Abb.).

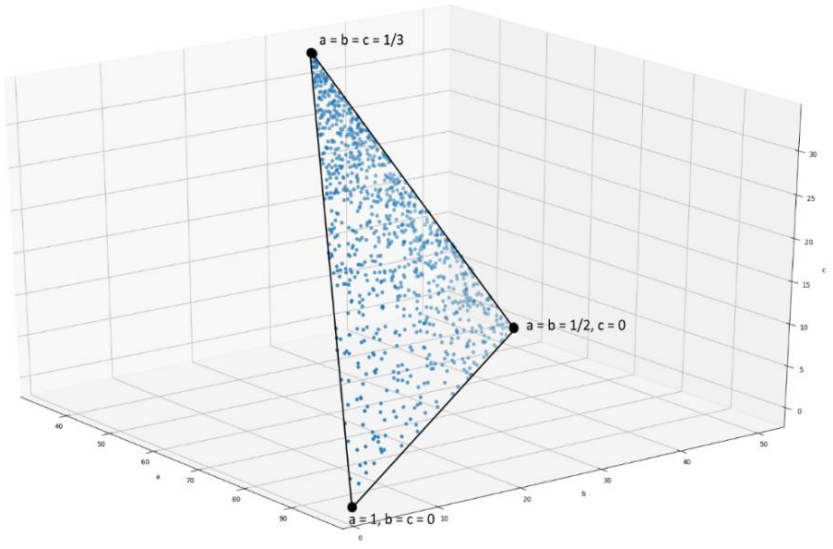


Abb. 13: Verteilung der möglichen Zusammensetzungen von a, b und c im dreidimensionalen Raum, errechnet mithilfe der Monte Carlo-Methode. Der Verteilungsraum begrenzt sich auf eine 2-dimensionale Fläche (ein Dreieck), wobei a maximal 100%, b maximal 50% und c maximal 33,33% (die jeweiligen Spitzen des Dreiecks) der Gesamtlänge aller drei Eigenvektoren ausmachen können.

Nachfolgend ein paar kurze Beispiele, um zu demonstrieren, wie die Formel funktioniert:

1.) S am Beispiel einer perfekten Kugel ($a = b = c = \frac{1}{3}$)

Da alle Hauptachsen gleich lang sind, reduziert sich die Formel bei Berechnung einer perfekten Kugel auf den folgenden Ausdruck:

$$\frac{\frac{1}{3} * \frac{1}{3}}{\frac{1}{3} * \frac{1}{3} * \frac{1}{3} * \frac{1}{3}} * \left(\frac{1}{3} * \frac{1}{3}\right) = 1$$

Der Faktor von $(1/3)^2$ normiert den Index, um zu erreichen, dass er in diesem besonderen Fall genau 1 ist.

2.) S am Beispiel eines leicht eiförmigen Ellipsoids

Nimmt man als Verteilung der Eigenvektoren $a = 0{,}45 \quad b = 0{,}30 \quad c = 0{,}25$ an, errechnet sich für den Shape-Index

$$\frac{0{,}45^2}{0{,}30^2 * 0{,}25^2} * \left(\frac{1}{3}\right)^2 = \frac{0{,}2025}{0{,}005625} * \frac{1}{9} = 4.$$

Da dieses Ellipsoid noch sehr nahe an der Form einer Kugel liegt, weicht der Wert des Index (im Vergleich zu Tellern oder Nadeln) nur wenig vom kleinstmöglichen Wert (1) ab.

3.) Einige weitere Beispiele

Bezeichnung	a	b	c	Shape Index
Kugel	1/3	1/3	1/3	1
Ellipsoid	0,45	0,3	0,25	4
Ellipsoid	0,6	0,2	0,2	25
Teller	0,55	0,42	0,03	211.71
Teller	0,72	0,27	0,01	7.901,23
Nadel	0,92	0,05	0,03	41.797,53
Nadel	0,98	0,01	0,01	~10^7

Abb. 14: Einige zufällige Beispiele für den Shape Index bei verschiedenen Verteilungen/Formen.

Man kann aus der Tabelle ablesen, dass der Shape-Index sich exponentiell vergrößert, wenn sich die analysierte Form von der einer Kugel entfernt.
Die Verwendung des Shape Index hatte zwei Begründungen:
Einerseits besteht durch ihn die Möglichkeit, Ellipsoide mit einem Index über einer bestimmten Perzentile (in dieser Arbeit wurde ein Cut-Off von 90% gewählt) – und damit Ellipsoide, deren Form zu stark von der einer Kugel abweichen – vor der Graph-Generierung auszuschließen.
Andererseits, kann aus ihm ungefähr die Form des analysierten Ellipsoids herausgelesen werden, wobei vor allem die Trennung zwischen Tellern und Nadeln die Hauptmotivation war (was möglicherweise für spätere Verwendungszwecke nützlich ist).

Erzeugung von Adjacency Matrix und Graph

Nachdem die unerwünschten Feature Origins herausgefiltert wurden, wurden die nahe aneinander gelegenen verknüpft, um den Graph zu formen.
Dazu wurden zunächst die euklidischen Distanzen aller Feature Origins zu allen anderen berechnet und in eine Distanz-Matrix eingetragen, welche die Distanzen aller Feature Origins zu allen anderen angab:

	`1`	`2`	`3`	`4`
	<dbl>	<dbl>	<dbl>	<dbl>
1	0	13.6	3.52	3.40
2	13.6	0	11.2	10.2
3	3.52	11.2	0	1.92
4	3.40	10.2	1.92	0

Abb. 15: Ausschnitt einer Distanz-Matrix. Rot markierte Distanzen führen bei einem Threshold von 3.5 zu einer Verbindung, die gelben gerade nicht mehr.

Auf Grundlage des Threshold-Wertes wurden dann alle Werte kleiner oder gleich dem Threshold zu 1 und alle größer t zu 0 . Danach wurde die Diagonale auf 0 gesetzt, da Knoten definitionsgemäß nicht mit sich selbst verbunden sein dürfen.

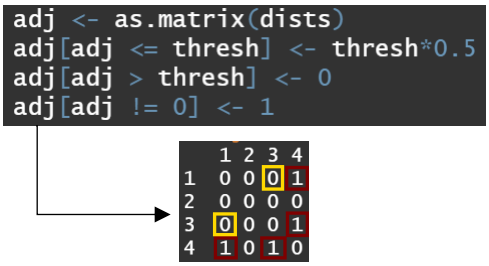


Abb. 16: Umwandlung aller Distanzen zu 0 oder 1, je nachdem, ob sie über oder unter dem Threshold lagen und die daraus resultierende „adjacency matrix“, an den gleichen Stellen markiert wie Abb. 15

Mithilfe der Funktion „graph_from_adjacency_matrix“ aus der igraph-library[26] lässt sich aus der Adjacency Matrix ein Graph erzeugen:

```
g <- graph_from_adjacency_matrix(adj, mode = 'undirected')
```

Extraktion des größten connected component

Dieser Graph hat, je nach Höhe der Threshold-Variable, immer noch eine bestimmte Anzahl an Artefakten, die jetzt durch Extraktion des größten connected component entfernt werden können.

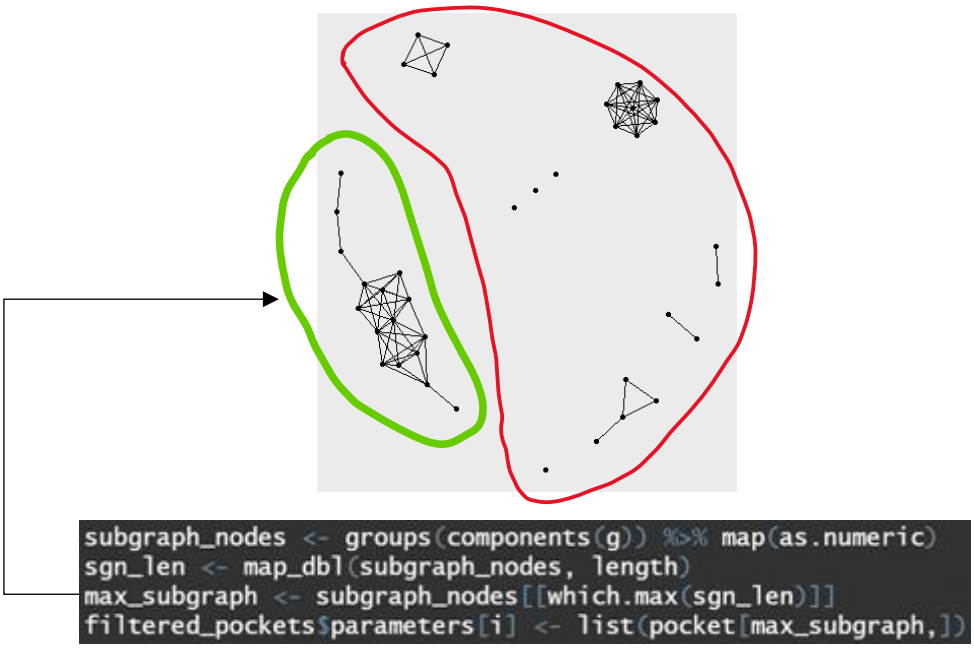


Abb. 17: Ein Graph vor Extraktion des größten Subgraphen, mit Artefakten rot und dem größten connected component grün eingekreist.

Die Variable „max_subgraph“ enthält alle Knotenpunkte des größten Teilgraphen und wird in eine Liste eingetragen, alle anderen Knoten können für die Berechnung der Fingerprints und Indices, welche im nächsten Teil durchgeführt wird, verworfen werden.
Dadurch ist die Filterung der Feature Origins beendet.

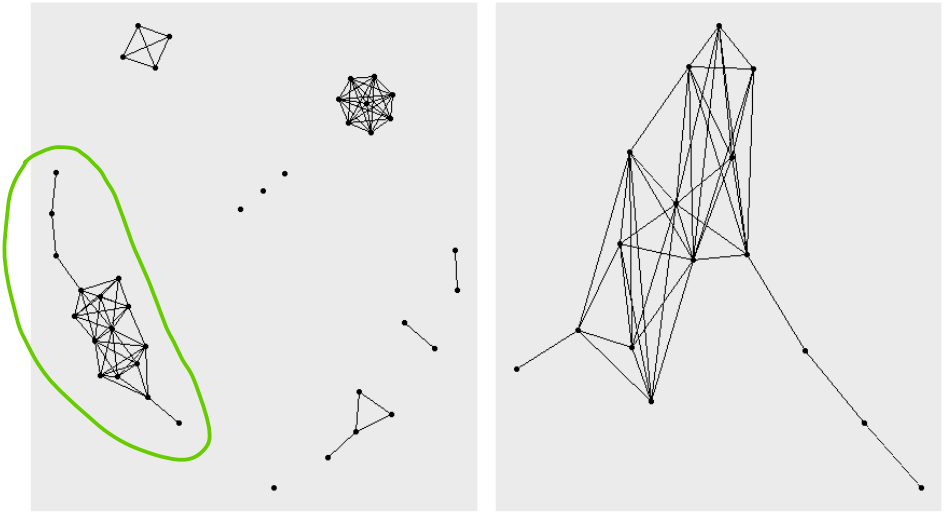


Abb. 18: Graph vor und nach Extraktion des größten Subgraphen

Erzeugung der Fingerprints

Der Algorithmus für die Berechnung der Graph-Indices und der Fingerprints funktioniert bis zur Generierung des Graph-Objekts exakt gleich, wie der zur Extraktion des größten Subgraphen, aber unter Ausschluss der Knotenpunkte, die nicht zu diesem Subgraphen gehört haben. Dadurch wurden die korrekten/bereinigten Graphen generiert.
Um nun einen Fingerprint aus diesen fertigen Graphen zu berechnen, waren drei weitere Schritte notwendig:
Zuerst mussten alle Pfade der Längen 2, 3 und 4 gefunden werden, die im Graph vorkommen.
Dann mussten diese Pfad-Muster in einen Bitvektor eingetragen werden.
Schließlich wurden die Graph-Indices berechnet.

Auffinden aller Pfade der Längen 2, 3 und 4

Dieser Vorgang läuft additiv in drei Teilschritten ab:
Zuerst werden alle Pfade mit Länge zwei (die Länge eines Pfads gibt immer an, wie viele Knotenpunkte er enthält) ausfindig gemacht. Dann werden mit ihnen die Pfade der Länge drei berechnet und mit diesen schlussendlich die Pfade der Länge vier.

1.) Pfade der Länge zwei

Diese Pfade lassen sich leicht erzeugen, indem man mit der igraph-Funktion „neighborhood“ eine Liste der Nachbarn jedes Knotenpunkts in einem Graph erzeugt:

```
> neighbor_list <- map(neighborhood(g), ~.x[-1])
> neighbor_list
[[1]]
+ 6/22 vertices, from 95e3856:
[1]  5 11 13 14 18 19

[[2]]
+ 8/22 vertices, from 95e3856:
[1]  3  5  7  9 12 16 17 20

[[3]]
```

Abb. 19: Eine Liste der Nachbarn aller Knotenpunkte (map(…, ~.x[-1]) entfernt den ersten Knoten jedes Listeneintrags, also den Knoten, dessen Nachbarschaft angezeigt wird)

Um aus dieser Liste die Pfade der Länge zwei zu generieren, muss man nur durch die Liste iterieren und den Index der Liste mit allen Einträgen kombinieren (mit der in R eingebauten paste-Funktion)

```
paths_2 <- list()
paths_3 <- list()
paths_4 <- list()

for(v in seq_along(neighbor_list)){
  paths_2 <- c(paths_2, strsplit(paste(v, neighbor_list[[v]]), " "))
}
paths_2 <- map(paths_2, as.integer)
```

```
> paths_2
[[1]]
[1] 1 3

[[2]]
[1] 1 6

[[3]]
```

Abb. 20: Generierung aller Pfade der Länge zwei

2.) Pfade der Länge drei

Um Pfade aus drei Knoten zu erhalten, wurde wiederum die gesamte Nachbarschaft des zweiten Knotenpunkts aller Pfade der Länge zwei mit allen ihren benachbarten Knotenpunkten kombiniert.

```
for(v in paths_2){
  v3_vec <- as.vector(neighbor_list[[v[2]]])
  p3 <- paste(v[1], v[2], v3_vec)
  paths_3 <- c(paths_3, strsplit(p3, " "))
}
paths_3 <- map(paths_3, as.numeric)
keep_3 <- map_lgl(paths_3, ~length(unique(.x))==3)
paths_3 <- paths_3[keep_3]
```

```
> head(paths_3)
[[1]]
[1] 1 3 2

[[2]]
[1] 1 3 4

[[3]]
```

Abb. 21: Generierung aller Pfade der Länge drei

Danach wurden alle Pfade ausgeschlossen, die einen Knotenpunkt mehr als einmal enthielten, also vom zweiten Knoten wieder zurück auf den ersten verwiesen haben (z.B. „1 3 1“, „2 7 2“, etc.). Dazu wurde die unique-Funktion auf alle Listeneinträge angewendet und nur solche Pfade behalten, die danach immer noch eine Länge von drei hatten.

3.) Pfade der Länge vier

Die Pfade mit Länge vier wurden exakt nach dem selben Schema wie die Pfade der Länge drei berechnet, weshalb dies hier nicht näher beschrieben wird.

Erzeugung des Bitvektors

Die Pfade lagen nun als Vektoren von Zahlen vor, die die Knotenpunkte des Graphen repräsentierten. Da jedoch für diese Arbeit keine Zahlenkombinationen interessant waren, sondern Musterkombinationen, mussten die Zahlen zuerst zu Feature Types zurückübersetzt werden:

```
chosen_paths_labelled <- map(chosen_paths, ~labeling_df$label[.x]) %>% unique
```

Mit „unique“ wurde sichergestellt, dass jede Musterkombination höchstens einmal vorkommt, da für einen Bitvektor nur wichtig ist, ob eine Kombination vorkommt und nicht, wie oft.

Alle möglichen Permutationen von Labels, die eine Kombination aus zwei, drei oder vier Feature-Origin-Namen darstellen, wurden generiert und einem „leeren“ (mit Nullen gefüllten) Bitvektor der selben Länge zugewiesen.

Zur Erklärung: Permutationen sind Kombinationen bei denen die Reihenfolge der kombinierten Objekte von Bedeutung ist, d.h. „1 2“ und „2 1“ sind zwei unterschiedliche Permutationen, aber dieselbe Kombination. Warum Musterpermutationen und nicht -kombinationen verwendet werden müssen, ist leicht zu zeigen:

Hätte man die Feature Types a, b und c, wären alle möglichen Dreier-Kombinationen:

aaa, aab, aac, abb, abc, acc, bbb, bbc, bcc, ccc.

Permutationen wie aca, bac, bcb, etc. würden nicht erzeugt werden, obwohl sie nötig sind, damit alle möglichen Pfade in einem Graph codiert werden können.

```
bv_labels <- unite(combination_table, nms)$nms
bitvector <- rep(0, nrow(combination_table)) #
names(bitvector) <- bv_labels
```

```
> combination_table
# A tibble: 4,096 x 4
   X1    X2    X3    X4
   <chr> <chr> <chr> <chr>
 1 hph   hph   hph   hph
 2 hph   hph   hph   ary
 3 hph   hph   hph   arp
 4 hph   hph   hph   pia
 5 hph   hph   hph   acc
 6 hph   hph   hph   don
 7 hph   hph   hph   pos
 8 hph   hph   hph   neg
 9 hph   hph   ary   hph
10 hph   hph   ary   ary
# ... with 4,086 more rows
```

Abb. 22: Bitvektor-Label- Permutationen

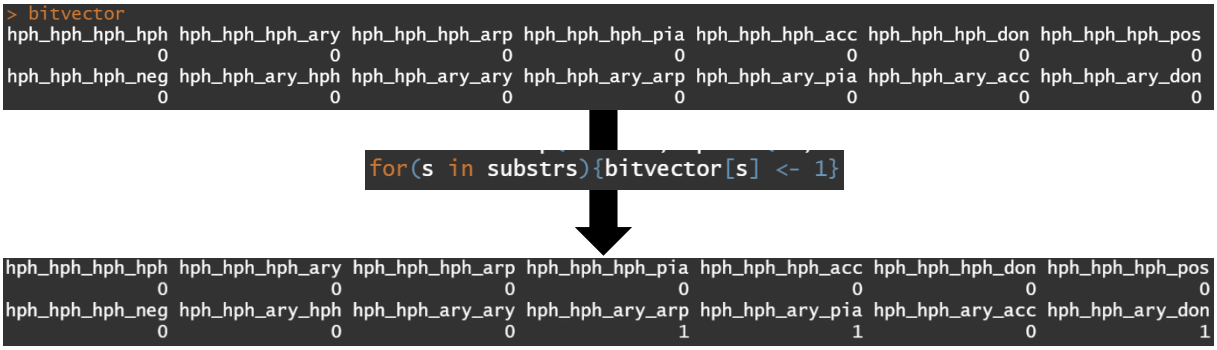


Abb. 23: Erzeugung und Füllung des Bitvektors. Wenn ein Muster mit dem selben Namen in den Musterkombinationen vorkommt (nachdem diese in das gleiche Format wie die Namen des Bitvektors transformiert wurden (gespeichert in der Variable „substrs“)), wird der Bitvektor an dieser Stelle auf 1 gesetzt

Ergebnisse, Interpretation, Diskussion

Durch die hier beschriebenen Methoden ist es gelungen, Proteinbindungstaschen in Form von Graphen und in weiterer Linie in Form von Zahlenketten und Indices zu abstrahieren. Dies liefert Potential für die Unterstützung des computergestützten Wirkstoffdesign.

Ursprüngliche Ziele

Zuerst war vorgesehen, die im Datensatz enthaltenen pk_D-, pk_{on}- und pk_{off}-Werte zu benützen, um ein Machine-Learning-Modell zu trainieren, dass aufgrund von Bitvektoren und Graph-Indices (Maßzahlen für die Graph-Architektur) Vorhersagen über Bindungskinetik treffen kann. Jedoch wurde die Reichweite des Projektes zu groß und so wurde in dieser Arbeit nur auf die Methodik der Graph- und Fingerprint-Generierung (inkl. Graph-Indices) eingegangen, wobei die 3D-Modelle der im genannten Datensatz aufgelisteten Proteine als Grundlage für die Graph-Generierung genommen wurden.

Was ist neu an P-3?

Feature-Platzierung

Die mit GRAIL berechneten Molecular Interaction Fields (MIFs) werden über Gaussian Mixture Modelling (GMM) statistisch auf einen Ursprungspunkt (oder „Feature Origin“) zurückgerechnet. Dadurch werden Unsicherheiten bezüglich Platzierung vermieden und Überlappungen vermieden.

Generierung der Fingerprints

Ein großer Teil von P-3 basiert darauf, aus „Feature Origins“ Graphen zu erzeugen. Diese Graphen ermöglichen es, Fingerprints durch Analyse der vorkommenden Muster zu generieren. Diese Muster werden in einen Bitvektor mit sämtlichen theoretisch möglichen Mustern gewünschter Länge (bis jetzt wurden nur Fingerprints mit Permutations-Mustern aus 2, 3 und 4 Feature-Typen erzeugt) eingetragen (als 1, alle nicht vorhandenen Muster werden mit einer 0 repräsentiert).

Warum Graphen?

Erhöhung der Methodenvielfalt zur Proteinanalyse

Graphbasierte Fingerprintgenerierung für Proteinbindungstaschen ist (zumindest auf die hier beschrieben Art) eine neue Herangehensweise an das Problem der Abstrahierung von Proteinen. Sie könnte bereits vorliegende oder noch in der Zukunft liegende Probleme lösen oder bei der Lösung helfen.

Optimierbarkeit

Graphen bieten einzigartige Möglichkeiten zur Optimierung, die bei Methoden mit Distanz-Feature Type-Fingerprints nicht gegeben sind, etwa der Einsatz von gewichteten Graphen oder Graph Neural Networks. Dadurch sind sie flexibler als Methoden, deren Fingerprints direkt aus dem Pharmakophor „abgelesen“ werden, bieten also mehr Hyperparameter (=Stellschrauben), an denen herumgedreht werden kann.

Graph Indices

Eine weitere, Graph-spezifische Eigenschaft sind die bereits erwähnten Graph Indices, die Potential für Machine Learning bieten.

Vergleich der erwähnten Ansätze

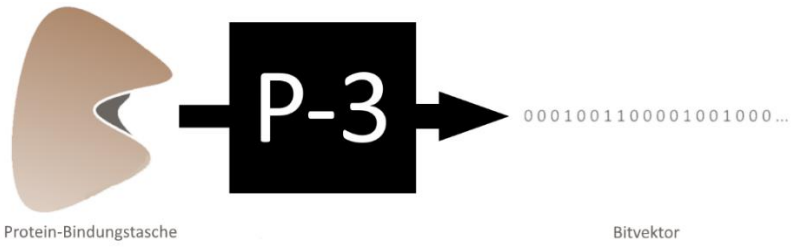


Abb. 24: Überblick über den Ablauf von P-3

In einem Satz erklärt ist das Ziel von P-3 die Erzeugung von Fingerprints als Funktion der Architektur von Proteinbindungstaschen um damit Proteine vergleichbar zu machen und es zu ermöglichen, Machine Learning-Methoden für deren Analyse anzuwenden.

Unterschiede zwischen KRIPO und P-3

1.) Die Feature-Platzierung wurden bei KRIPO auf Basis geometrischer Überlegungen und Berechnungen durchgeführt. Bei P-3 wurden die theoretischen physiko-chemischen Interaktionen zwischen Protein und Ligand berechnet und diese statistisch auf einzelne Punkte reduziert. P-3 ist so näher an der Realität und weniger anfälliger für unvorhersehbare Effekte, die aus der Komplexität reeller Begebenheiten erwachsen können.

2.) Die Erzeugung der Fingerprints war bei KRIPO eine Funktion von Proteinbindungstaschen und Liganden/Ligandenbruchstücken, bei P-3 war sie nur eine Funktion der Bindungstasche. Das bedeutet, dass Proteinbindungstaschen keinen einheitlichen Bitvektor besitzen, da dieser in Abhängigkeit des verwendeten Liganden entsteht.

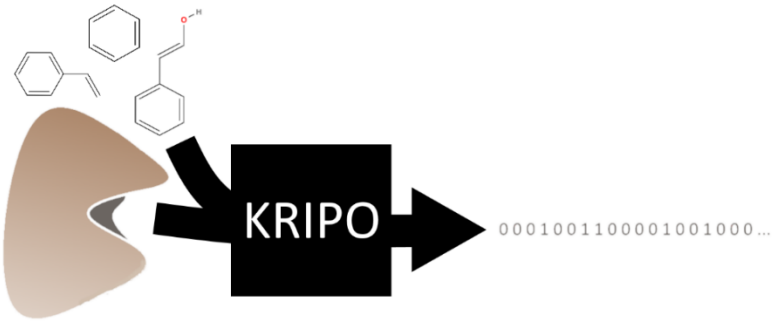


Abb. 25: Überblick über den Ablauf von KRIPO

3.) Die Fingerprint-Bits codieren etwas Anderes, nämlich Feature Type-Distanz-Kombinationen, während bei P-3 Pfad-Muster in Graphen codiert werden.

4.) Das Ziel von KRIPO ist konkreter, nämlich die Auffindung bioisosteren Ersatzes für gewünschte Moleküle, während die Fingerprints von P-3 allgemeiner sind.

Unterschiede zwischen FLAP und P-3

1.) FLAP funktioniert auf eine sehr ähnliche Weise wie GRAIL, weshalb bei P-3 ebenfalls Probleme mit Überlappungen existieren. Im Gegensatz zu FLAP wird P-3 aber um das Gaussian Mixture Modelling erweitert, welches das Überlappungsproblem löst, indem es Punktewolken zu distinkten Ellipsoiden umrechnet, deren Mittelpunkte berechnet werden können (im Gegensatz zu ungeclusterten Punktewolken).

2.) FLAP-Fingerprints codieren – wie die KRIPO-Fingerprints – für Feature Type-Distanz-Kombinationen

Methode	Feature-Platzierung	Fingerprint
KRIPO	Nach geometrischen Berechnungen	Entfernungen und Feature-Type-Kombinationen (2er-, 3er- und 4er-Kombinationen)
FLAP	Energetisch günstig	Entfernungen und Feature-Type-Kombinationen
P-3	Energetisch günstig, geclustert und auf Punkte reduziert	Theoretisch mögliche Muster in einem Graph

Abb. 26: Gegenüberstellung der bereits erwähnten Methoden

ANWENDUNGEN

1.) Es wäre möglich, über den Tanimoto- oder erweiterten Tanimoto-Koeffizienten (siehe Kapitel Indices) die Bitvektoren der Protein-Bindungstaschen (und darüber die Bindungstaschen selbst) zu vergleichen, um auf Ähnlichkeiten zu überprüfen. Dadurch könnte man z.B. versuchen, vorherzusagen, ob Liganden an bestimmten Targets binden werden, indem mit Bindungstaschen verglichen wird, die diesen Liganden binden bzw. nicht binden.

2.) Eine weitere Idee, die versucht wurde, aber im Rahmen dieser Diplomarbeit nicht vervollständigt werden konnte, ist, Molekül-Fingerprints, die das Vorliegen bestimmter funktioneller Gruppen codieren, und Protein-Fingerprints zu einem zweidimensionalen Fingerprint zu kombinieren, in dem jede Kombination aus Protein-Bit und Molekül-Bit codiert wird.
Für jedes Feld dieses neuen Fingerprints könnte man durchschnittliche pharmakokinetische Parameter berechnen, um herauszufinden, welche Musterkombinationen mit besonders starker oder schwacher Interaktion zwischen Proteinen und Liganden korrelieren. Dadurch wäre es möglich, vorherzusagen, wie man, je nach vorliegendem Target oder Anti-Target, ein Molekül adaptieren könnte, um z.B. die Stärke der Bindungsinteraktion (K_D) in eine gewünschte Richtung zu beeinflussen. Auf diese Weise könnte man, wie bei KRIPO, Bioisostere finden, toxikologische Vorhersagen oder Liganden-Screening machen.

3.) Man könnte über Deep Learning vorhersagen, welche kinetischen Parameter wahrscheinlich aus gegebenen Bitvektoren und Graph-Indices hervorgehen werden.

Vorteile der Methode

Unter der Voraussetzung, dass die Methode funktioniert wie erhofft, könnte man Proteine untereinander vergleichen oder Interaktionen von vielen Liganden mit einem Protein vorhersagen, weitaus schneller als herkömmliche Methoden wie Docking, die viel Zeit, Aufwand und Rechnerleistung benötigen. Es ließen sich z.B. auf Grundlage einer Liste von Proteinnamen Datenbanken explorativ nach Hits screenen, ohne jegliche menschliche Intervention.
Weiters könnten durch Kombination der Informationen von Liganden und Targets bessere Modelle zu Toxizität oder Bindungskinetik erstellt werden, als durch alleinige Verwendung von Liganden, da eventuelle Zusammenhänge zwischen Liganden- und Target-Fingerprints mehr Aufschluss über Interaktionen liefern könnten.

Probleme und Verbesserungsmöglichkeiten

Beim Workflow der Fingerprint- und Index-Erzeugung gibt es noch einige Probleme und ungelöste Fragen. Dazu zählen die Platzierung der Feature-Origins, der Linking-Threshold und die Exklusion der Feature Types.

1 – Origins

Es wurde davon ausgegangen, dass jedes Ellipsoid von genau einem Feature Origin ausgeht. Das kann möglicherweise fehlerhaft sein. Es könnten Muster-Kombinationen im Bitvektor verfälscht werden oder Graphen unverbunden bleiben, die eigentlich verbunden sein sollten da ein wichtiger Knotenpunkt fehlt.

2 – Threshold

Der Threshold wurde über Trial-and-Error in einem bestimmten Bereich festgelegt.
Er hat sehr großen Einfluss darauf, wie fertige Graphen aussehen. Deshalb ist es sehr wichtig, einen idealen Wert zu finden, entweder statisch (Festlegung für alle Proteine) oder dynamisch (z.B. Anpassung über Rückkopplung).
Der statische Lösungsansatz würde über Trial-and-Error funktionieren. Man könnte, unabhängig vom verwendeten Datensatz, verschiedene Threshold-Werte ausprobieren, für jeden Wert eigene Ergebnisse erzeugen und die
Der Nachteil dieser Herangehensweise ist, dass die Anzahl der möglichen (ungerundeten) Threshold-Werte unendlich groß wird und, wenn man seine Ergebnisse publizieren oder in einer Datenbank eintragen will, den oder die verwendeten Threshold-Wert/-Werte angeben muss, um

Ergebnisse nachvollziehbar und/oder reproduzierbar zu machen. Im Endeffekt würde dadurch eine Normierung erschwert.

Die individuelle Herangehensweise würde auf dem verwendeten Datensatz basieren.
Man könnte z.B. einen Threshold wählen, der einen gewissen Anteil der zunächst vorhandenen Feature Origins entfernt, bzw. andersrum einen zu exkludierenden Anteil auswählen, der ab einem bestimmten Threshold gegeben ist.
Dieser Wert wäre vermutlich besser zu argumentieren, weil die Auswirkungen leichter vorzustellen sind und er an eine konkrete Anforderung geknüpft wäre, anstatt zufällig gewählt zu werden.
Allerdings würde das das Problem der Parameter-Wahl nur in einen anderen Teil des Programms verschieben, zusätzliche Funktionen benötigen und mehr Berechnungszeit in Anspruch nehmen, da der einzige Weg, auf dem man herausfinden kann, wie viele Features bei einem gegebenen Threshold exkludiert würden ist, ihn über den gesamten Datensatz anzuwenden und dann Schritt für Schritt anzupassen.

3 – Exklusion von Feature Types
Eine weitere Schwachstelle ist der Shape-Index, der zwar bei erster Erprobung sinnvolle Ergebnisse erzielt hat, aber eventuell Schwachstellen aufweist bzw. unnötig kompliziert ist.
Allerdings wurden einige andere ähnliche Formeln ausprobiert und der jetzige Shape-Index hat sich als die sinnvollste Methode erwiesen, um Feature-Types zu exkludieren und gleichzeitig nach Form unterscheiden zu können.
Vor allem jedoch der zweite Punkt, nämlich die Unterscheidung der Formen, ist ein komplizierter Fall.
Erstens ist es möglich, dass diese Funktionalität unbrauchbar ist, entweder, weil sie unzuverlässig, oder die Unterscheidung gar nicht notwendig ist (diese Funktionalität wurde nur „auf Verdacht“ eingebaut). In diesem Fall wäre es besser, etablierte Maßzahlen wie die „Spherizität“[27] zu verwenden.
Zweitens wäre es, falls eine Unterscheidung doch notwendig oder sinnvoll ist, möglicherweise besser, die Exklusion der Feature-Types und die Beschreibung ihrer Formen mit zwei verschiedenen Indices zu beschreiben.
Etwa mithilfe eines stetigen Index zur Exklusion der Feature-Types (auch dafür wäre Spherizität wahrscheinlich geeignet) und dazu separat ein diskreter Index oder Regelsatz, der die Formen kategorisch einteilt (oder alternativ: Kategorisierung mithilfe logistischer Regression durch manuelles Labelling eines Formen-Datensatzes).

4 – Datenvorbereitung

Die Datenvorbereitungs-Pipeline war stark von manueller Arbeit geprägt und ist in Zukunft wahrscheinlich ein Punkt, der starke Optimierungsmöglichkeiten zulässt.

5 – Proteinkonstellationen

In dieser Arbeit wurden sämtliche Proteinbindungstaschen nur in einer gebundenen Konformation analysiert. In der Realität sind Proteine aber beweglich, wodurch sich auch die Positionierung von Feature Origins ändern kann. Deshalb wäre es denkbar von Vorteil, wenn man pro Protein mehrere Schlüsselkonstellationen hätte.

6 – Extraktion des größten connected component

Es wurden keine Sicherheitsmaßnahmen eingebaut, falls es bei der Extraktion zwei gleich große connected component gibt. In diesem Fall wird einfach zufällig durch die which.max-Funktion (s. Code, „Extraktion des größten connected component“) einer der beiden ausgewählt.
Man könnte über verschiedene Werte Priorisierung erzeugen, etwa nach Anzahl der Verbindungen in den connected component (also: wie „connected“ ist der connected component?).

7 – Weitere Graph-Indices und andere Maßzahlen

In dieser Arbeit wurde nur eine Auswahl von Graph-Indices vorgestellt, die nach Bedarf frei erweiterbar ist.
Außerdem korrelieren die Indices zwar mit Molekül-Eigenschaften, jedoch sind die in P-3 gebildeten Graphen anders vernetzt als Moleküle. Deshalb lassen sich vielleicht nicht alle, keine oder nur gänzlich andere Maßzahlen für die Vorhersage von Proteineigenschaften heranziehen.

8 – 3D-Modelle, Auswahl und Vorbereitung der Bindungstaschen

Da die Bindungstaschen aus 3D-Modellen extrahiert werden, ergeben sich ebenfalls zahlreiche Probleme.
Erstens sind 3D-Modelle von Natur aus fehlerbehaftet. Zum Beispiel ist die Wahl der Protein-Konformation nicht trivial und eine mögliche Fehlerquelle, da jeglicher Unterschied in der Position des Proteins das Graph-Muster beeinflussen kann und nicht klar ist, welche Konformation die „Richtige“ ist (oder ob alle Schlüsselkonformationen gleich wichtig sind). Auch hier wäre natürlich eine Trial-and-Error-Methode angebracht, jedoch werden dadurch Rechenzeit und Komplexität stark erhöht, vor allem in Kombination mit anderen Trial-and-Error-Methoden.
Weiters bedeutet die Vorbereitung der Bindungstaschen aufwändige händische Vorarbeit, was

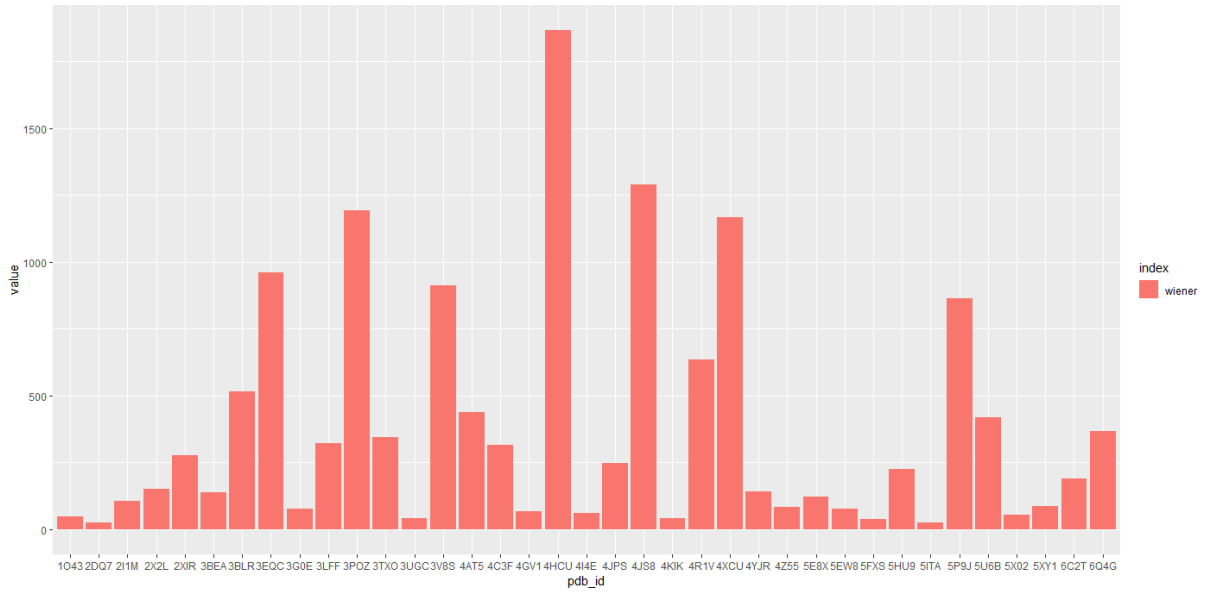
allerdings teilweise oder ganz durch Automatisierung umgangen werden könnte. Man könnte etwa versuchen, Bindungstaschen eines Proteins automatisch zu finden und dies mit der weiteren Verarbeitung (Standardisierung, Clustering, Filterung der Cluster) in eine Bearbeitungs-Pipeline zu vereinen. Allerdings ist natürlich nicht gesagt, wie gut oder ob dies funktionieren würde. Schlimmstenfalls müssten sämtliche Proteine händisch ausgewählt werden, wie es in dieser Arbeit der Fall war, was bei Analyse einiger Zielproteine kein unüberwindbares Problem darstellen, für breitere Analysen, wie etwa ein Vergleich sämtlicher Proteine der PDB, jedoch jeglichen Rahmen sprengen würde.

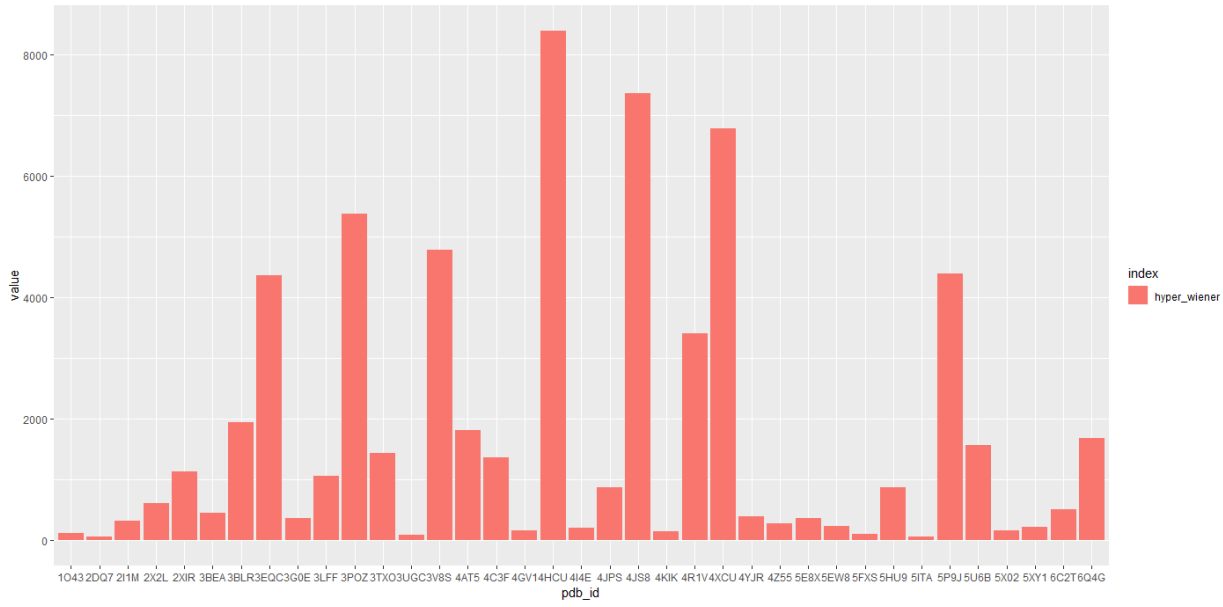
Resultate

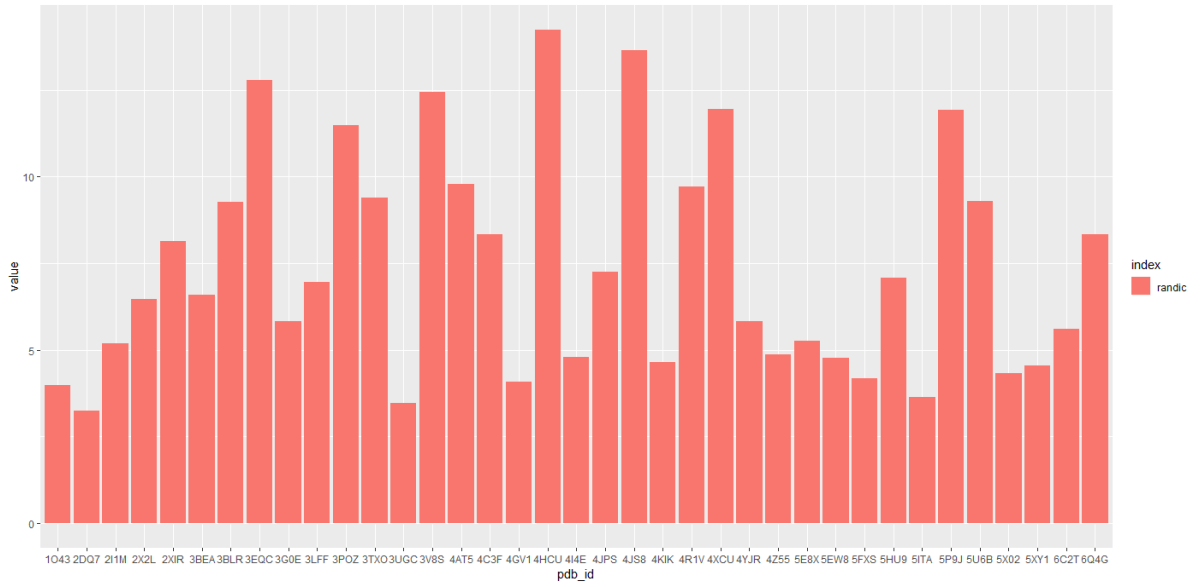
Verteilung der Graph-Indices

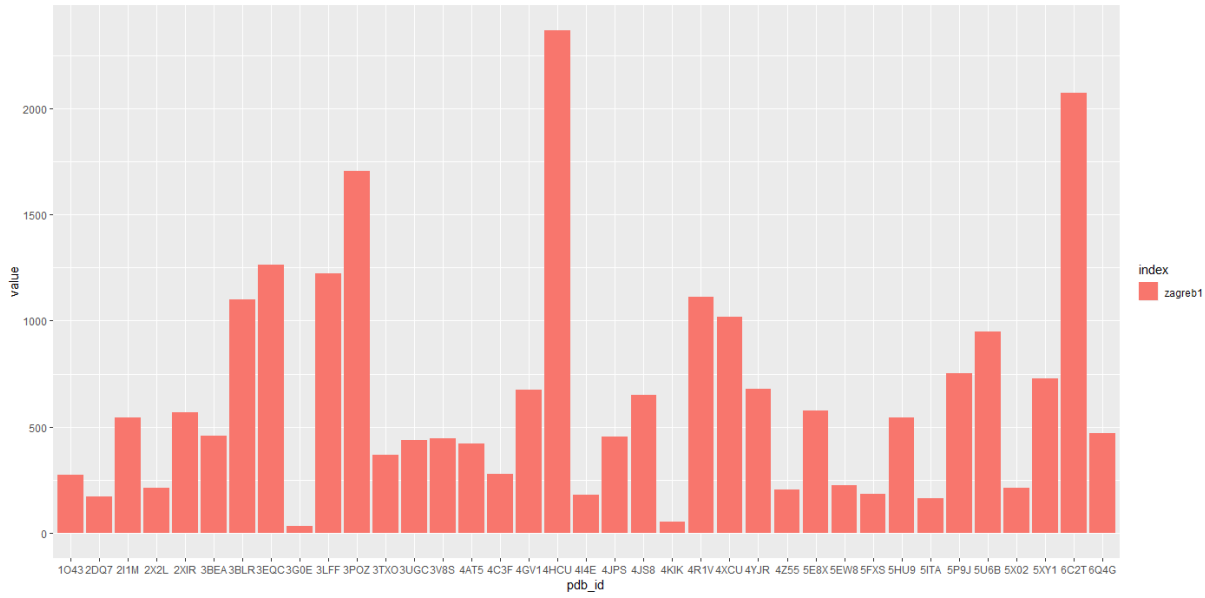
Nachfolgende Abbildungen (Abb. 27-36) zeigen, wie die Werte der verschiedenen Graph-Indices über die vorhandenen PDB-IDs variieren. Ob Graph-Indices mit Protein-Eigenschaften korrelieren wird allerdings erst in Zukunft zu sehen sein.

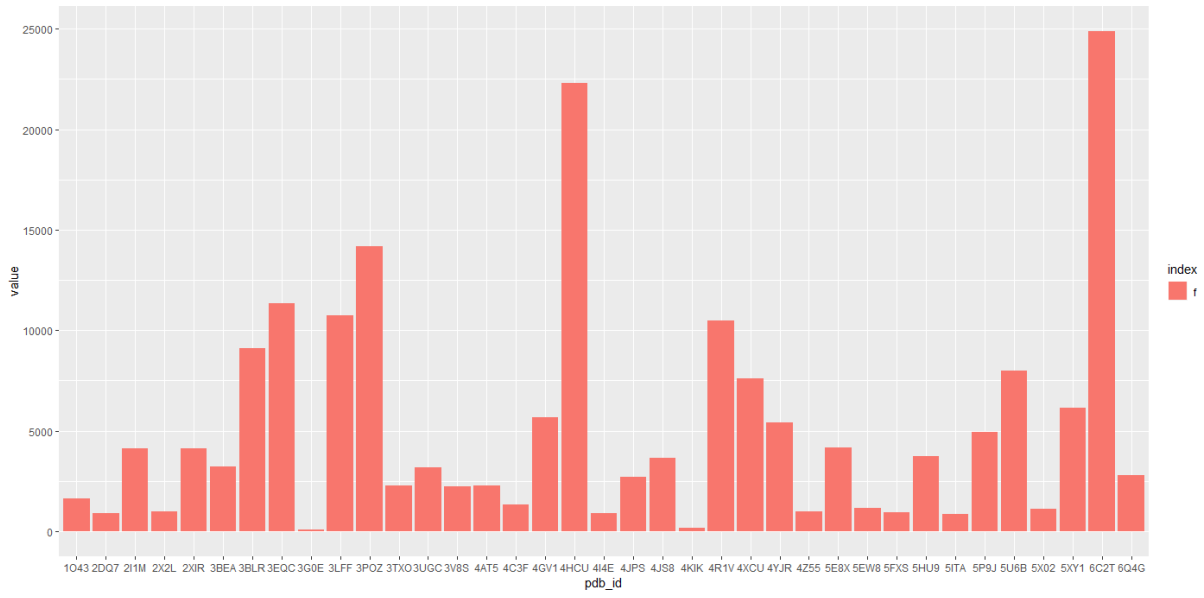
Reihenfolge:
Wiener-, Hyper-Wiener-, Randic-, erster Zagreb-, F-, Narumi-Katayama-, Szeged-, Padmakar-Ivan-, zweiter Zagreb- und Balaban-J-Index.
Anm.: Die Graphik für den Narumi-Katayama-Index verwendet eine logarithmische y-Achse, da sonst nur der größte Wert zu sehen ist.

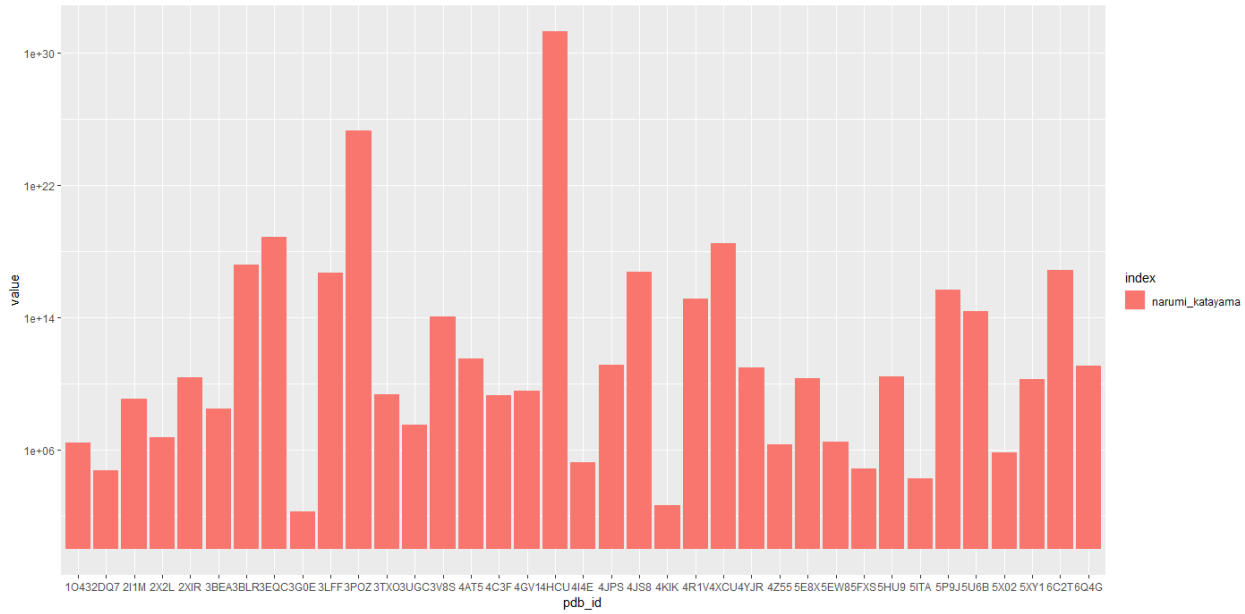


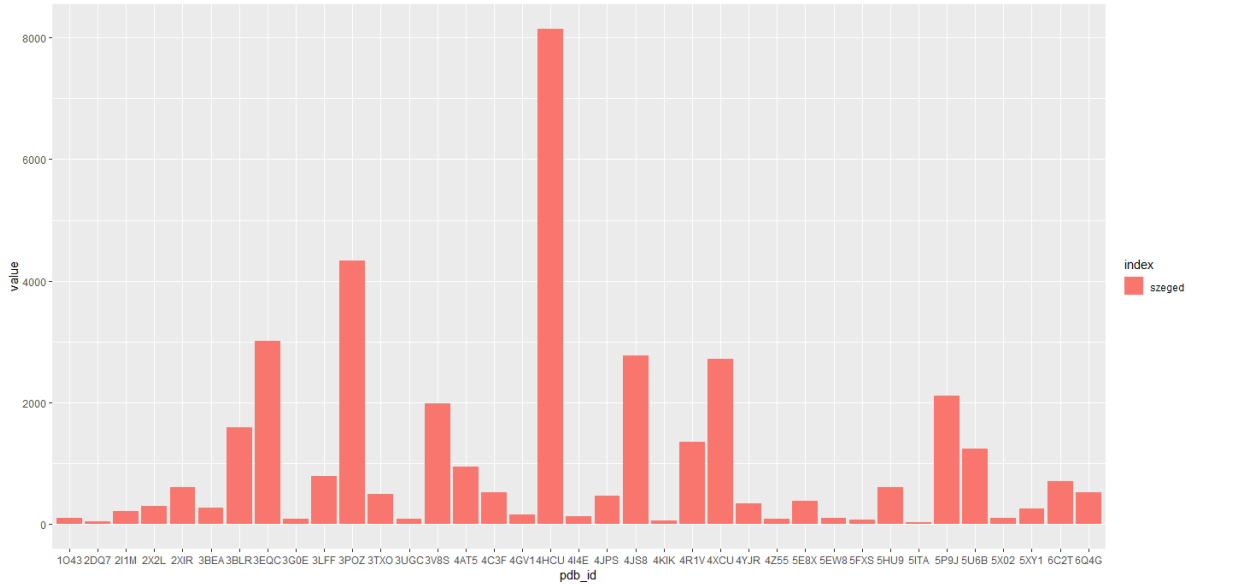


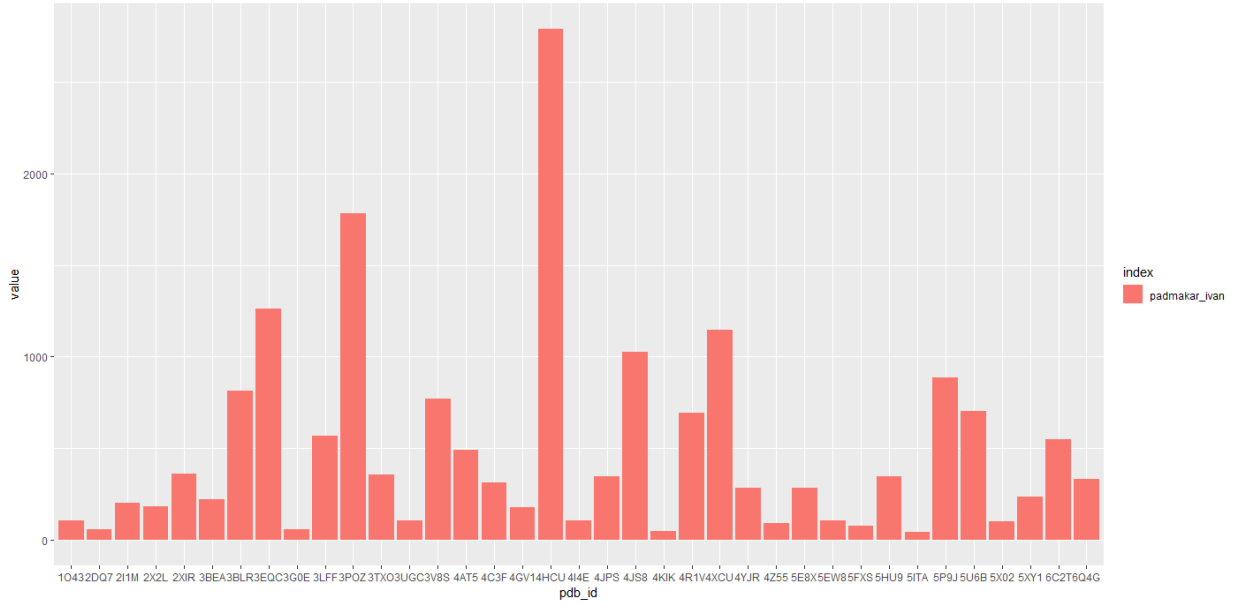

value
10
5
0
1O43 2DQ7 2I1M 2X2L 2XIR 3BEA 3BLR 3EQC 3G0E 3LFF 3POZ 3TXO 3UGC 3V8S 4AT5 4C3F 4GV1 4HCU 4I4E 4JPS 4JS8 4KIK 4R1V 4XCU 4YJR 4Z55 5E8X 5EW8 5FXS 5HU9 5ITA 5P9J 5U6B 5X02 5XY1 6C2T 6Q4G
pdb_id
index
randic

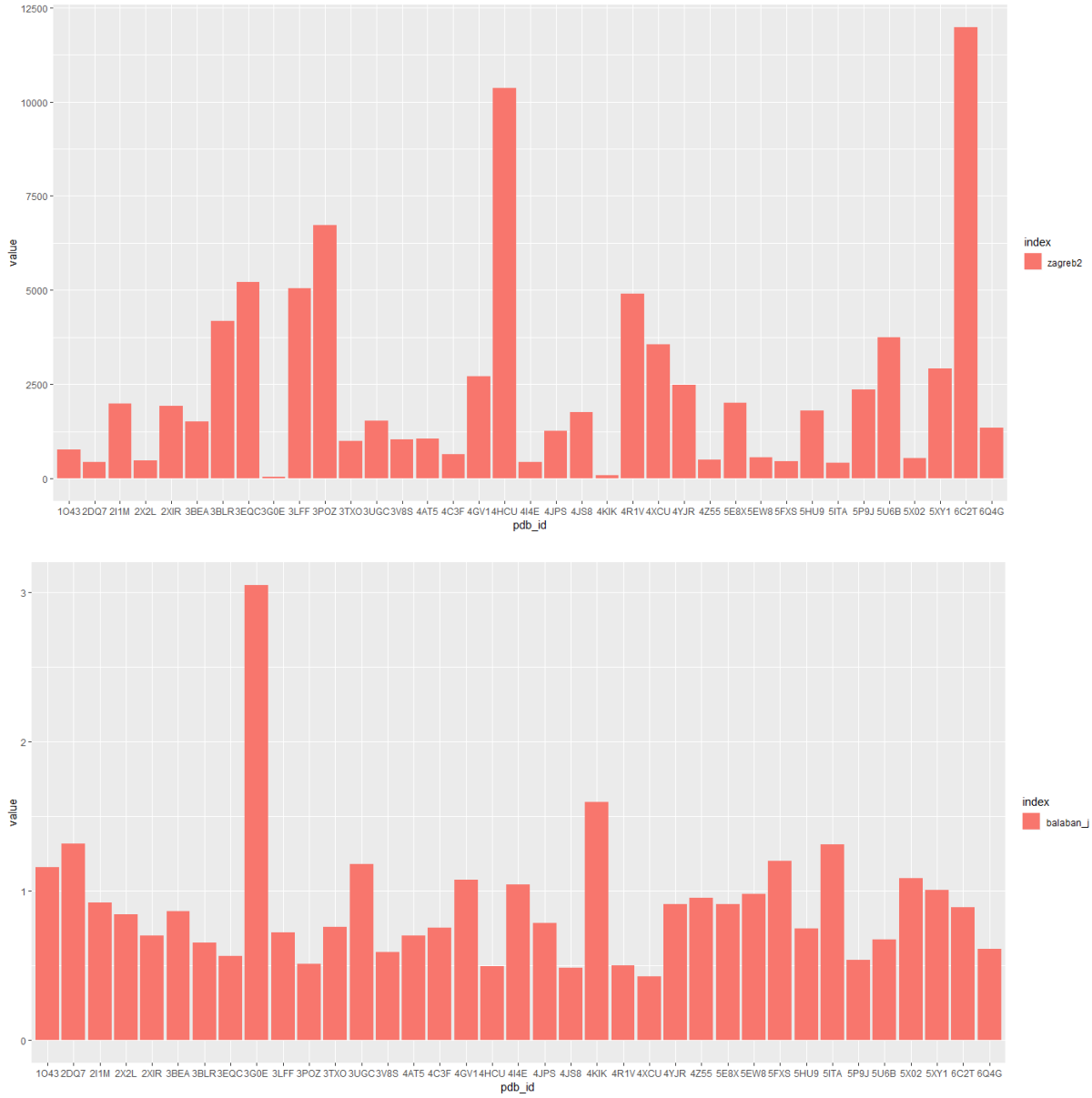

value
2000
1500
1000
500
0
1O43 2DQ7 2I1M 2X2L 2XIR 3BEA 3BLR 3EQC 3G0E 3LFF 3POZ 3TXO 3UGC 3V8S 4AT5 4C3F 4GV1 4HCU 4I4E 4JPS 4JS8 4KIK 4R1V 4XCU 4YJR 4Z55 5E8X 5EW8 5FXS 5HU9 5ITA 5P9J 5U6B 5X02 5XY1 6C2T 6Q4G
pdb_id
index
zagreb1


value
25000
20000
15000
10000
5000
0
1O43 2DQ7 2I1M 2X2L 2XIR 3BEA 3BLR 3EQC 3G0E 3LFF 3POZ 3TXO 3UGC 3V8S 4AT5 4C3F 4GV1 4HCU 4I4E 4JPS 4JS8 4KIK 4R1V 4XCU 4YJR 4Z55 5E8X 5EW8 5FXS 5HU9 5ITA 5P9J 5U6B 5X02 5XY1 6C2T 6Q4G
pdb_id
index
f


1e+30
1e+22
1e+14
1e+06
value
index
narumi_katayama
1O43 2DQ7 2I1M 2X2L 2XIR 3BEA 3BLR 3EQC 3G0E 3LFF 3POZ 3TXO 3UGC 3V8S 4AT5 4C3F 4GV1 4HCU 4I4E 4JPS 4JS8 4KIK 4R1V 4XCU 4YJR 4Z55 5E8X 5EW8 5FXS 5HU9 5ITA 5P9J 5U6B 5X02 5XY1 6C2T 6Q4G
pdb_id


8000
6000
4000
2000
0
value
index
szeged
1O43 2DQ7 2I1M 2X2L 2XIR 3BEA 3BLR 3EQC 3G0E 3LFF 3POZ 3TXO 3UGC 3V8S 4AT5 4C3F 4GV1 4HCU 4I4E 4JPS 4JS8 4KIK 4R1V 4XCU 4YJR 4Z55 5E8X 5EW8 5FXS 5HU9 5ITA 5P9J 5U6B 5X02 5XY1 6C2T 6Q4G
pdb_id


2000
1000
0
value
index
padmakar_ivan
1O43 2DQ7 2I1M 2X2L 2XIR 3BEA 3BLR 3EQC 3G0E 3LFF 3POZ 3TXO 3UGC 3V8S 4AT5 4C3F 4GV1 4HCU 4I4E 4JPS 4JS8 4KIK 4R1V 4XCU 4YJR 4Z55 5E8X 5EW8 5FXS 5HU9 5ITA 5P9J 5U6B 5X02 5XY1 6C2T 6Q4G
pdb_id

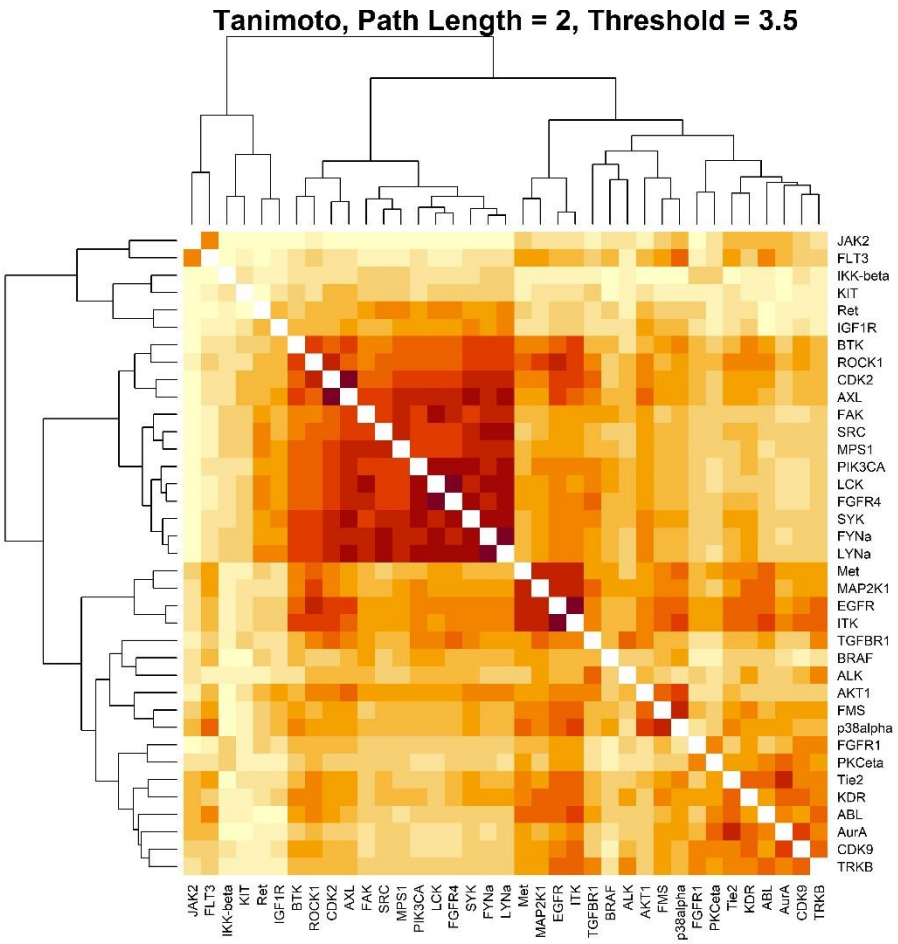


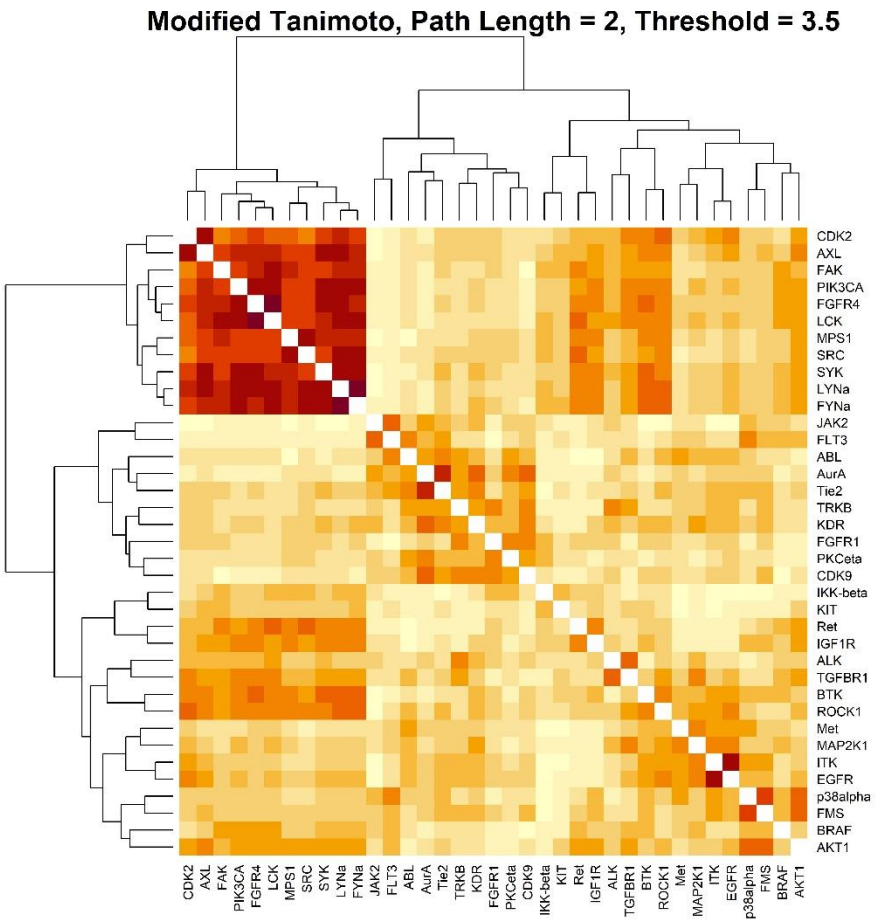
Clustering der verwendeten Kinasen

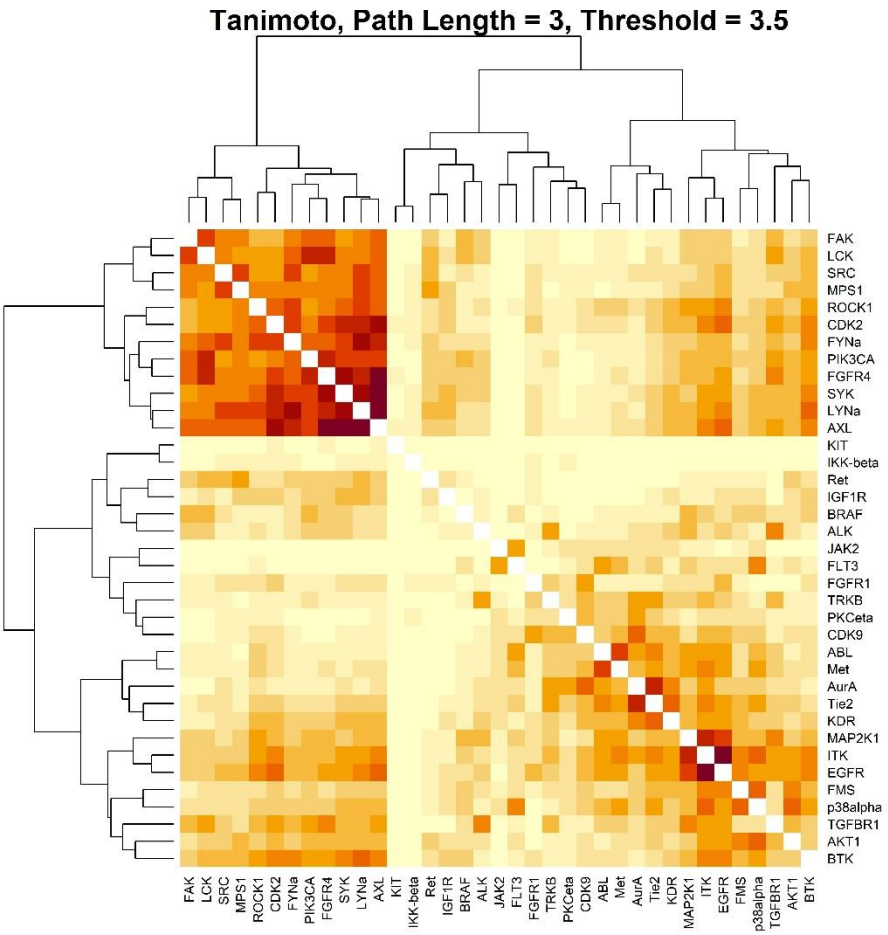
Der für die Graphen- und Fingerprint-Generierung zugrundliegende Datensatz besteht aus verschiedenen Kinasen. Es wurde deshalb versucht, diese anhand ihrer Tanimoto-Distanz zueinander zu clustern (gezeigt werden hier alle Kombination aus Pfadlängen von 2 bis 4 und Tanimoto- und modifiziertem Tanimoto-Index bei Threshold = 3.5) (Abb. 37-42).

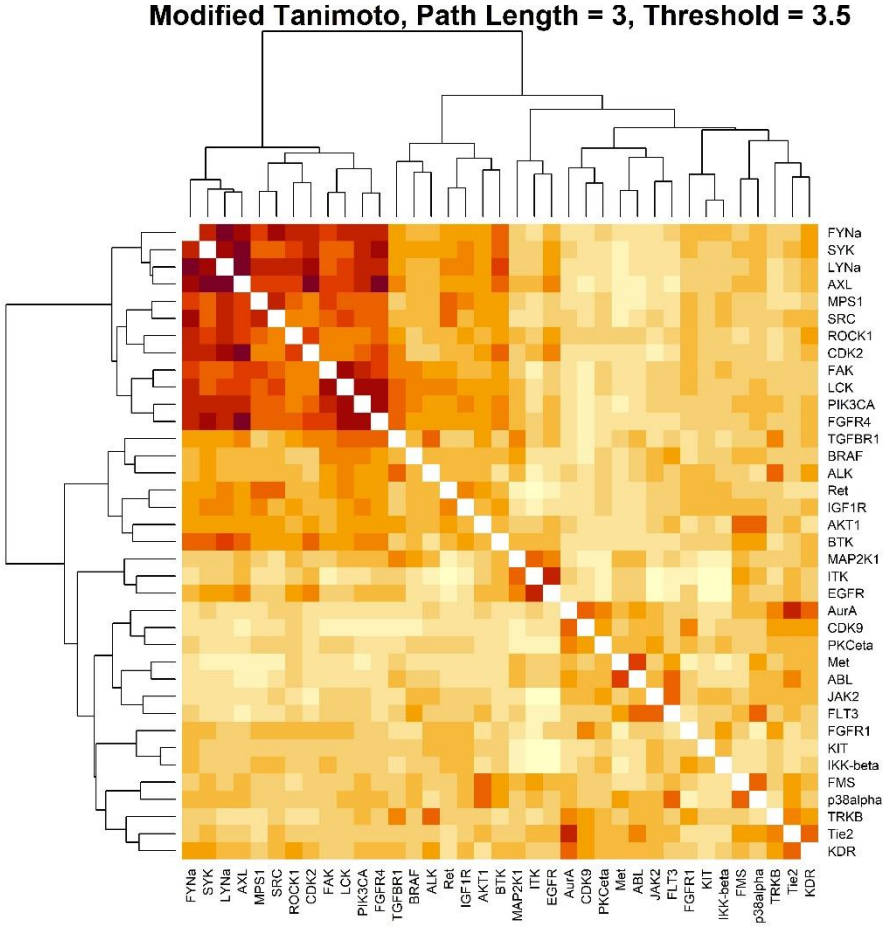
Man kann sehen, dass z.B. LYNa, FYNa, SRC und LCK, welche alle zur Src-Familie gehören, hohe Ähnlichkeit aufweisen (s. z.B. Abb. 38). Allerdings scheinen einige Kinasen aus anderen Familie wenig Ähnlichkeit untereinander aufzuweisen, z.B. CDK2 und CDK9 oder ITK und BTK.

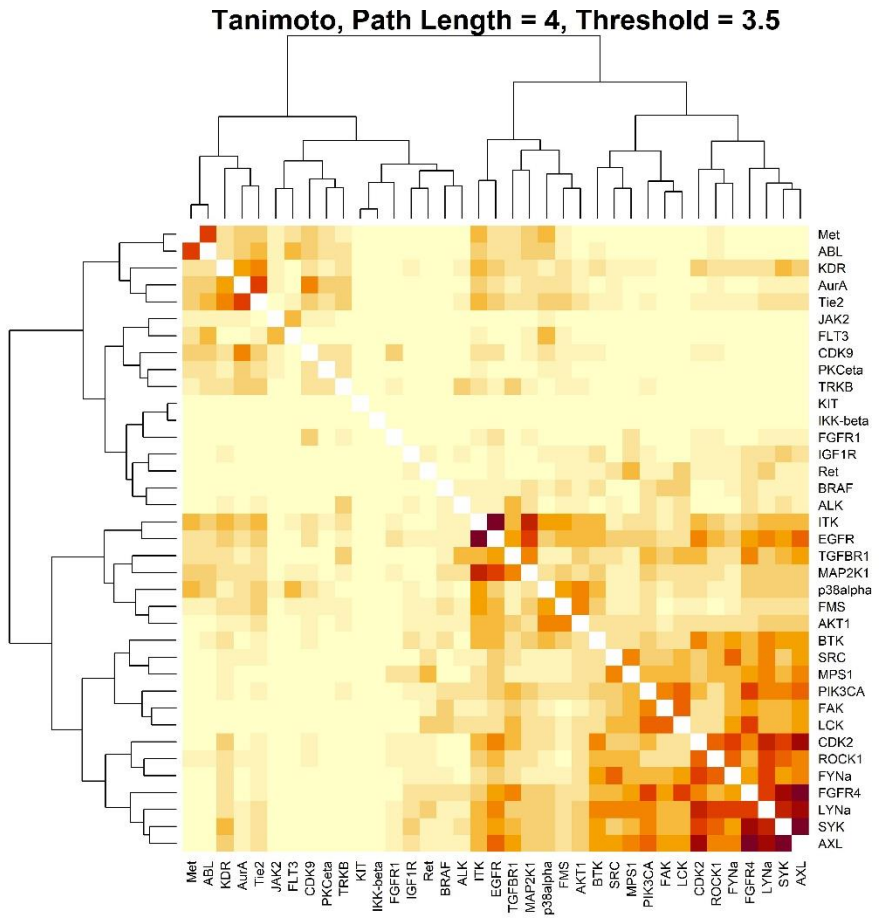
Um das Clustering sinnvoll bewerten zu können, müsste diese Methode allerdings auf bereits gelabelte/geclusterte Proteine angewendet und die Cluster verglichen werden, da es sonst schwer ist, qualitative Aussagen zu treffen.

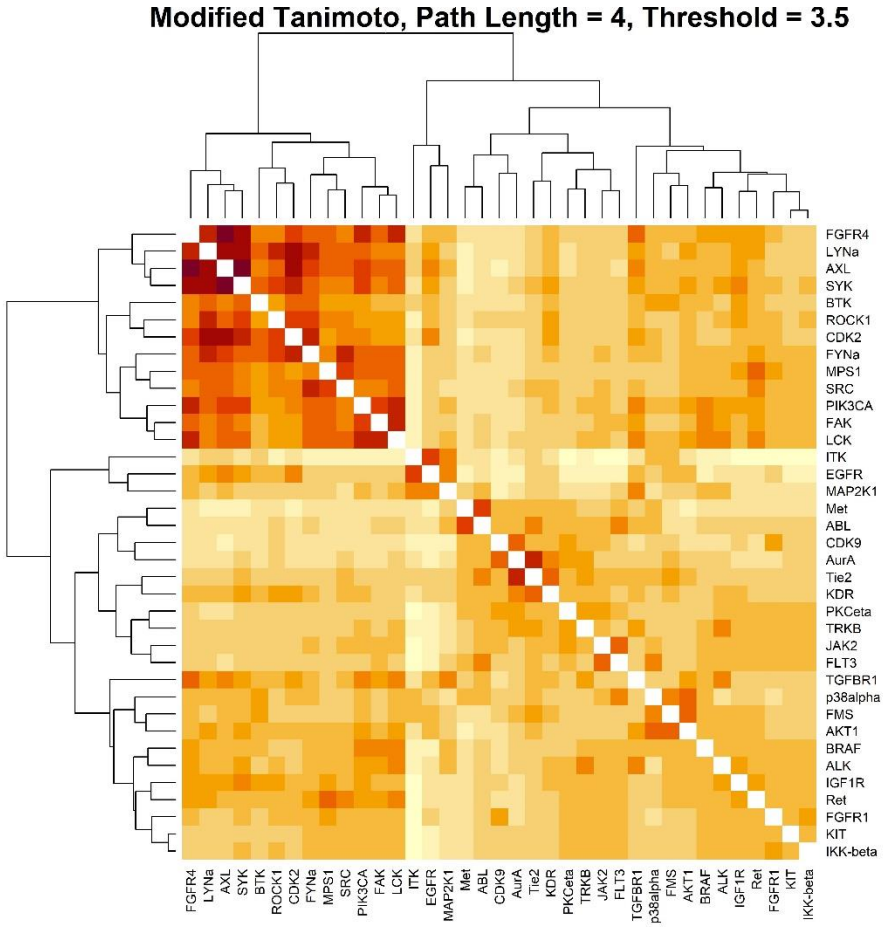

Tanimoto, Path Length = 2, Threshold = 3.5
JAK2
FLT3
IKK-beta
KIT
Ret
IGF1R
BTK
ROCK1
CDK2
AXL
FAK
SRC
MPS1
PIK3CA
LCK
FGFR4
SYK
FYNa
LYNa
Met
MAP2K1
EGFR
ITK
TGFBR1
BRAF
ALK
AKT1
FMS
p38alpha
FGFR1
PKCeta
Tie2
KDR
ABL
AurA
CDK9
TRKB


Modified Tanimoto, Path Length = 2, Threshold = 3.5
CDK2
AXL
FAK
PIK3CA
FGFR4
LCK
MPS1
SRC
SYK
LYNa
FYNa
JAK2
FLT3
ABL
AurA
Tie2
TRKB
KDR
FGFR1
PKCeta
CDK9
IKK-beta
KIT
Ret
IGF1R
ALK
TGFBR1
BTK
ROCK1
Met
MAP2K1
ITK
EGFR
p38alpha
FMS
BRAF
AKT1


Tanimoto, Path Length = 3, Threshold = 3.5
FAK
LCK
SRC
MPS1
ROCK1
CDK2
FYNa
PIK3CA
FGFR4
SYK
LYNa
AXL
KIT
IKK-beta
Ret
IGF1R
BRAF
ALK
JAK2
FLT3
FGFR1
TRKB
PKCeta
CDK9
ABL
Met
AurA
Tie2
KDR
MAP2K1
ITK
EGFR
FMS
p38alpha
TGFBR1
AKT1
BTK


Modified Tanimoto, Path Length = 3, Threshold = 3.5
FYNa
SYK
LYNa
AXL
MPS1
SRC
ROCK1
CDK2
FAK
LCK
PIK3CA
FGFR4
TGFBR1
BRAF
ALK
Ret
IGF1R
AKT1
BTK
MAP2K1
ITK
EGFR
AurA
CDK9
PKCeta
Met
ABL
JAK2
FLT3
FGFR1
KIT
IKK-beta
FMS
p38alpha
TRKB
Tie2
KDR


Tanimoto, Path Length = 4, Threshold = 3.5
Met
ABL
KDR
AurA
Tie2
JAK2
FLT3
CDK9
PKCeta
TRKB
KIT
IKK-beta
FGFR1
IGF1R
Ret
BRAF
ALK
ITK
EGFR
TGFBR1
MAP2K1
p38alpha
FMS
AKT1
BTK
SRC
MPS1
PIK3CA
FAK
LCK
CDK2
ROCK1
FYNa
FGFR4
LYNa
SYK
AXL


Modified Tanimoto, Path Length = 4, Threshold = 3.5
FGFR4
LYNa
AXL
SYK
BTK
ROCK1
CDK2
FYNa
MPS1
SRC
PIK3CA
FAK
LCK
ITK
EGFR
MAP2K1
Met
ABL
CDK9
AurA
Tie2
KDR
PKCeta
TRKB
JAK2
FLT3
TGFBR1
p38alpha
FMS
AKT1
BRAF
ALK
IGF1R
Ret
FGFR1
KIT
IKK-beta

Datensatz für weiteres Arbeiten

```
> kin_params_full
# A tibble: 2,780 x 9
   kinase pdb_id compound                    protein_fp    compound_fp indices      pKd  pKon pKoff
   <chr>  <chr>  <chr>                       <list>        <list>      <list>     <dbl> <dbl> <dbl>
 1 ABL    5HU9   Afatinib (BIBW2992)         <dbl [4,672]> <dbl [166]> <dbl [10]>  5.99 -5.57 0.420
 2 ABL    5HU9   "AG-1024 "                  <dbl [4,672]> <dbl [166]> <dbl [10]>  6.25 -5.63 0.650
 3 ABL    5HU9   AG-1478 (Tyrphostin AG-1478) <dbl [4,672]> <dbl [166]> <dbl [10]>  6.59 -5.89 0.710
 4 ABL    5HU9   AMG 900                     <dbl [4,672]> <dbl [166]> <dbl [10]>  6.01 -4.71 1.38
 5 ABL    5HU9   AMG458                      <dbl [4,672]> <dbl [166]> <dbl [10]>  6.69 -4.92 1.83
 6 ABL    5HU9   AT9283                      <dbl [4,672]> <dbl [166]> <dbl [10]>  7.88 -7.35 0.430
 7 ABL    5HU9   Aurora A Inhibitor I        <dbl [4,672]> <dbl [166]> <dbl [10]>  6.87 -5.22 0.760
 8 ABL    5HU9   Axitinib                    <dbl [4,672]> <dbl [166]> <dbl [10]>  8.02 -6.79 1.22
 9 ABL    5HU9   AZ628                       <dbl [4,672]> <dbl [166]> <dbl [10]>  6.90 -5.30 1.63
10 ABL    5HU9   AZD4547                     <dbl [4,672]> <dbl [166]> <dbl [10]>  6.72 -5.93 0.88
# ... with 2,770 more rows
```

Abb. 43: Ausschnitt aus dem für Machine Learning vorgesehenen Datensatz.

In weiteren Arbeiten kann versucht werden, die Protein-Fingerprints, Compound-Fingerprints (hier wurden für die Liganden Fingerprints verwendet, die das Vorliegen bestimmter funktioneller Gruppen codieren) und Graph Indices mit den pharmakologischen Parametern (pk_D, pk_{on}, pk_{off}) zu korrelieren, etwa, um Vorhersagen für andere Interaktionen treffen zu können.

Abschließende Worte

Durch diese Arbeit wurde bewiesen, dass es möglich ist, Proteinbindungstaschen in eine vergleichbare und leicht zu verarbeitende und zu speichernde Form zu abstrahieren.

Da dieses Feld (genauer gesagt, die Verwendung von Graphen in Kombination mit Proteinbindungstaschen) allerdings noch sehr neu ist, ist zurzeit nicht sicher, wie oder ob diese Form der Abstraktion genutzt werden kann und es sind noch sehr viele Möglichkeiten offen, die Methodik zu optimieren.

Sollte dies gelingen, wäre sie möglicherweise ein großer Beitrag für computergestütztes Wirkstoffdesign.

R-Skripte

Graph-Funktionen

```
shape_idx <- function(a, b, c) 2*a/(b*b+c*c)

# p ist die durchschnittliche Bit-Dichte im zugrundeliegenden Datensatz
tanimotos <- function(bv1, bv2, p = NA){
  if(!is.numeric(bv1) || !is.numeric(bv2)){
    stop("input needs to be numeric")
  }else if(!all(bv1 == 0 | bv1 == 1) | !all(bv2 == 0 | bv2 == 1)){
    stop("input needs to be two bitvectors")
  }else if(length(bv1) != length(bv2)){
    stop("bitvectors need to be the same length")
  }else{
    n <- length(bv1)
    x <- which(bv1 == 1)
    y <- which(bv2 == 1)

    A <- sum(bv1)
    B <- sum(bv2)
    C <- sum(x%in%y)

    if(is.na(p)) p <- (A+B)/(length(bv1)+length(bv2))

    t <- C/(A+B-C)
    t_inv <- (n-A-B+C)/(n-C)
    t_mod <- ((2-p)/3)*t + ((1+p)/3)*t_inv
    tanims <- c(t, t_inv, t_mod)
    names(tanims) <- c("Tanimoto", "inverse Tanimoto", "modified Tanimoto")
    return(tanims)
  }
}

wieners <- function(g){
  d <- distances(g)
  wiener <- sum(d)/2
  hyper_wiener <- sum(d + d^2)/2
  return(c(wiener, hyper_wiener))
}

randic_zagreb1_f_nk <- function(g){
  deg <- degree(g)
  randic <- sum(deg**-0.5)
  zagreb <- sum(deg**2)
  f <- sum(deg**3)
  nk <- prod(deg)
  return(c(randic, zagreb, f, nk))
}

szeged_padivan_zagreb2 <- function(g){
  szgd <- 0
  pdvn <- 0
  zgrb <- 0
  edgl <- get.edgelist(g)
  for(i in seq_along(edgl[,1])){
    row <- edgl[i,]
    node_one <- row[1]
```

```
    node_two <- row[2]
    path_one <- shortest.paths(g, node_one)
    path_two <- shortest.paths(g, node_two)
    degree_one <- degree(g, node_one)
    degree_two <- degree(g, node_two)
    number_one <- sum(path_one < path_two)
    number_two <- sum(path_one > path_two)
    szgd <- szgd + (number_one*number_two)
    pdvn <- pdvn + (number_one+number_two)
    zgrb <- zgrb + (degree_one*degree_two)
  }
  return(c(szgd, pdvn, zgrb))
}

balaban_j <- function(g){
  q <- length(E(g))
  n <- length(V(g))
  μ <- q - n
  part_one <- q/(μ+1)

  d <- distances(g)
  d_sums <- apply(d, 1, sum)
  part_two <- 0
  for(i in 1:(length(d_sums)-1)){
    part_two <- part_two + 1/sqrt(d_sums[i]*d_sums[i+1])
  }

  j <- part_one * part_two
  return(j)
}
```

EXTRAKTION DES GRÖßTEN CONNECTED COMPONENT

```
# Einlesen der notwendigen Daten
kin_df <- read_csv('robust_mega_df.csv', col_names = TRUE) %>%
  mutate(shape_index = shape_idx(frac_a, frac_b, frac_c)) %>%
  filter(!is.na(cluster_id))

quantiles <- kin_df %>% group_by(feature_type) %>% summarize(q90 =
quantile(shape_index, .5, na.rm = T)) %>% ungroup()
pocket_names <- unique(kin_df$PDBID)
no_of_pockets <- length(pocket_names)

filtered_pockets <- tibble(
  pocket_id = 1:no_of_pockets, pdb_id = pocket_names,
  linking_threshold = thresh,
  parameters = list(NA), adj_matrix = list(NA), indices = list(NA), bitvectors =
list(NA)
  )

pb <- progress_bar$new(format = "> Subgraph Extraction\n> [:bar]\n>
:current/:total\n>:percent", total = no_of_pockets)

for(i in 1:no_of_pockets){
  cat("\014") # löscht alle Einträge aus der Konsole
  pb$tick()

  pocket <- kin_df %>% filter(PDBID == pocket_names[i])
  for(j in 1:nrow(quantiles)){
    to_remove <- which(pocket$feature_type == quantiles$feature_type[j] &
pocket$shape_index > quantiles$q90[j])
    if(length(to_remove)){ # = wenn to_remove mehr als 0 Einträge hat
      pocket <- pocket[-to_remove,]
    }
  }

  xyz <- data.frame(
    x = pocket$center_x,
    y = pocket$center_y,
    z = pocket$center_z
    )

  n <- 1:nrow(pocket)
  cols <- as.character(n)

  # erzeugt einen Vektor mit durchnummerierten Clustern,
  # die später den Knoten zugeordnet werden
  clusters <- paste('Cluster', cols)

  # fügt n Spalten hinzu, die alle mit 0en gefüllt sind
  xyz[cols] <- 0

  dists <- xyz[cols]
  xyz <- xyz[c('x', 'y', 'z')]

  cluster_combo_mat <- combinations(n, 2)
  if(length(cluster_combo_mat) == 0){
    filtered_pockets$parameters[i] <- NA
    next
```

```
  }
  cluster_combos <- c()
  for(j in 1:nrow(cluster_combo_mat)){
    cluster_combos <- append(cluster_combos, paste(cluster_combo_mat[j,1],
cluster_combo_mat[j,2], sep = '_'))
  }

  dist_vec = NULL
  for(j in seq_along(cluster_combos)){
  # berechnet die euklidische Distanz der Cluster zueinander
    clusts <- strsplit(cluster_combos[j], '_')[[1]]
    xyz_clust_1 <- xyz[clusts[1], ]
    xyz_clust_2 <- xyz[clusts[2], ]
    dist_vec[j] <- TSdist::EuclideanDistance(as.numeric(xyz_clust_1),
as.numeric(xyz_clust_2))
  }

  names(dist_vec) <- cluster_combos
  for(j in seq_along(names(dist_vec))){
    # fügt die Distanzen an den richtigen Stellen im xyz-df ein
    positionen <- strsplit(cluster_combos[j], '_')[[1]]
    pos_1 <- positionen[1]
    pos_2 <- positionen[2]
    dists[pos_1, pos_2] <- dist_vec[j]
    dists[pos_2, pos_1] <- dist_vec[j]
  }

  # fügt die cluster und ihre Distanzen zu einem df zusammen
  xyz <- cbind(clusters, xyz, dists) %>% as_tibble()
  xyz$clusters <- as.character(xyz$clusters)
  adj <- as.matrix(dists) # Adjacency-Matrix
  adj[adj <= thresh] <- -1
  adj[adj > thresh] <- 0
  adj[adj == -1] <- 1
  diag(adj) <- 0 # Matrix-Diagonale = 0 (per Definition)
  g <- graph_from_adjacency_matrix(adj, mode = 'undirected')

  subgraph_nodes <- groups(components(g)) %>% map(as.numeric)
  sgn_len <- map_dbl(subgraph_nodes, length)
  max_subgraph <- subgraph_nodes[[which.max(sgn_len)]]

  # die Spalten adj_matrix & indices werden erst später mit Werten gefüllt
  filtered_pockets$parameters[i] <- list(pocket[max_subgraph,])
}
```

Berechnung der Fingerprints und Graph-Indices

(Nur dieses Skript muss ausgeführt werden. Es ruft die beiden anderen über die source-Funktion auf)

```
# Das Skript erzeugt einen Graph basierend auf im
# 3-dimensionalen Raum verteilten Cluster-Zentren
# (xyz-Koordinaten), deren Verknüpfungen auf Abstand
# im 3D-Raum (euklidische Distanz) basieren (verbunden = ja oder nein).
pkgs <- c("tidyverse", "TSdist", "igraph", "ggraph", "arrangements", "progress",
"readxl", "hashmap")
invisible(lapply(pkgs, library, character.only = TRUE)) # lädt alle Packages
setwd("path/to/files")
paths <- 2:4 # welche Pfadlängen kommen in den kombinierten Bitvektor
substr_len <- 3 # wie viele Label-Zeichen für Abkürzungen
thresh <- 3.5 # Entfernung < thresh = Verbindung

source("Diplom-Funktionen 3.R")
source("Subgraph-Extractor 4.R")

old_labs = c("H-H", "AR-AR", "AR-PI", "PI-AR", "HBA-HBD", "HBD-HBA", "PI-NI", "NI-
PI")
new_labs <- c("hphobic", "aryl", "arpi", "piar", "acceptor", "donor", "pos-ion",
"neg-ion") %>% tolower
h <- hashmap(keys = old_labs, values = new_labs)

for(i in 1:nrow(filtered_pockets)){
  if(is.logical(filtered_pockets$parameters[[i]])){next}
  filtered_pockets$parameters[[i]]$feature_type <-
h[[filtered_pockets$parameters[[i]]$feature_type]]
}

kin_pdb_ids <- read_excel("KinData - überarbeitet.xlsx") %>% dplyr::select(Kinase,
PDB_ID) %>% set_names(c("kinase", "pdb_id"))
without_pdb_id <- c("FGFR3", "IKK-Alpha")
# für manche Kinasen hat es keine sinnvolle pdb-id

kin_params <- read_delim("raw_kinetic_data_kinases.csv", delim = ";") %>%
filter(`FP:MACCS` != "NA")
kin_params$fingerprint <- list(NA)
mol_fp_len <- 166
mol_empty_fp <- rep(0, mol_fp_len)

for(i in 1:nrow(kin_params)){
  ones_loc <- kin_params$`FP:MACCS`[i] %>% strsplit(" ") %>% unlist %>% as.numeric
# 1er-Locations als numerischer Vektor
  fp <- mol_empty_fp
  fp[ones_loc] <- 1
  kin_params$fingerprint[[i]] <- fp
}

kin_params <- kin_params %>% filter(!kinase %in% without_pdb_id)
kin_params <- kin_params %>% dplyr::select(kinase, Compound, fingerprint, pKD,
pkon, pkoff) %>% dplyr::arrange(kinase, Compound)
names(kin_params) <- c("kinase", "compound", "compound_fp", "pKd", "pKon",
"pKoff")
kin_params <- left_join(kin_params, kin_pdb_ids, by = "kinase") %>%
dplyr::select(kinase, pdb_id, compound, compound_fp, pKd, pKon, pKoff)

pb <- progress_bar$new(format = "> Fingerprint-/Index-Calculation\n> [:bar]\n>
:current/:total\n>:percent", total = no_of_pockets)
```

```
for(i in 1:no_of_pockets){
  cat("\014")
  pb$tick()

  # nächster Loop, wenn keine Verbindung existiert
  if(is.logical(filtered_pockets$parameters[[i]])) next


  xyz <- as.data.frame(filtered_pockets$parameters[[i]][c("center_x", "center_y",
"center_z")])
  names(xyz) <- letters[24:26]

  # erzeugt Spalten, in die später die Distanzen zu den
  # anderen Punkten eingetragen werden
  cols <- paste(1:nrow(filtered_pockets$parameters[[i]]))
  # erzeugt einen Vektor mit durchnummerierten Clustern,
  # die später den Knoten zugeordnet werden
  clusters <- filtered_pockets$parameters[[i]]$cluster_id
  labeling_df <- data.frame(label = filtered_pockets$parameters[[i]]$feature_type)
  labeling_df$label <- as.character(labeling_df$label)
  xyz[cols] <- 0
  dists <- xyz[,4:length(colnames(xyz))]
  xyz <- xyz[, 1:3]
  rownames(labeling_df) <- clusters

  xyz_len <- 1:length(xyz$x)
  cluster_combos <- combinations(xyz_len, 2) %>% as.data.frame %>%
unite(throwaway_name) %>% unlist %>% set_names(NULL)

  dist_vec = NULL
  for(j in seq_along(cluster_combos)){
  # berechnet die euklidische Distanz der Cluster zueinander
    clusts <- strsplit(cluster_combos[j], "_")[[1]]
    xyz_clust_1 <- xyz[clusts[1],]
    xyz_clust_2 <- xyz[clusts[2],]
    dist_vec[j] <- EuclideanDistance(as.numeric(xyz_clust_1),
as.numeric(xyz_clust_2))
  }
  names(dist_vec) <- cluster_combos

  for(j in seq_along(names(dist_vec))){
  # fügt die Distanzen an den richtigen Stellen im xyz-df ein
    positions <- strsplit(cluster_combos[j], "_")[[1]]
    pos_1 <- positions[1]
    pos_2 <- positions[2]
    dists[pos_1, pos_2] <- dist_vec[j]
    dists[pos_2, pos_1] <- dist_vec[j]
  }

  # fügt die cluster und ihre Distanzen zu einem df zusammen
  xyz <- cbind(clusters, xyz, dists) %>% as_tibble
  # ...daraus entsteht die Adjacency-Matrix
  adj <- as.matrix(xyz[5:length(colnames(xyz))])
  colnames(adj) <- NULL
  adj[adj <= thresh] <- thresh*0.5
  adj[adj > thresh] <- 0
  adj[adj != 0] <- 1
  diag(adj) <- 0
  filtered_pockets$adj_matrix[i] <- list(adj)
```

```
  g <- graph_from_adjacency_matrix(adj, mode = "undirected")

  neighbor_list <- map(neighborhood(g), ~.x[-1])
  paths_2 <- list()
  paths_3 <- list()
  paths_4 <- list()

  for(v in seq_along(neighbor_list)){
    paths_2 <- c(paths_2, strsplit(paste(v,neighbor_list[[v]]),"  "))
  }
  paths_2 <- map(paths_2, as.integer)

  for(V in paths_2){
    v3_vec <- as.vector(neighbor_list[[V[2]]])
    p3 <- paste(V[1], V[2], v3_vec)
    paths_3 <- c(paths_3, strsplit(p3, "  "))
  }
  paths_3 <- map(paths_3, as.numeric)
  keep_3 <- map_lgl(paths_3, ~length(unique(.x))==3)
  paths_3 <- paths_3[keep_3]

  for(V in paths_3){
    v4_vec <- as.vector(neighbor_list[[V[3]]])
    p4 <- paste(V[1], V[2], V[3], v4_vec)
    paths_4 <- c(paths_4, strsplit(p4, "  "))
  }
  paths_4 <- map(paths_4, as.numeric)
  keep_4 <- map_lgl(paths_4, ~length(unique(.x))==4)
  paths_4 <- paths_4[keep_4]

  chosen_paths <- c(paths_2, paths_3, paths_4)
  # Nodes => Feature Types
  chosen_paths_labelled <- map(chosen_paths, ~labeling_df$label[.x]) %>% unique

  bitvector_list <- list()

  for(p in paths){
    # chop = chosen paths
    chop_filter <- map_lgl(chosen_paths_labelled, ~length(.x) == p)
    # alle Kombinationen mit Pfadlänge x
    chosen_paths_filtered <- chosen_paths_labelled[chop_filter]

    mat <- permutations(length(new_labs), p, replace = TRUE)
    df <- data.frame(mat)
    df <- map_dfc(df, ~new_labs[.x])
    combination_table <- map_dfc(df, ~substring(.x, 1, substr_len))

    # alle möglichen Bitvektor-Kombinationen als Strings
    bv_labels <- unite(combination_table, nms)$nms
    bitvector <- rep(0, nrow(combination_table)) # leerer Bitvektor
    names(bitvector) <- bv_labels

    substrs <- map(chosen_paths_filtered, ~substring(.x, 1, substr_len))
    substrs <- map(substrs, ~paste(.x, collapse = "_"))
    for(s in substrs) bitvector[s] <- 1

    bitvector_list <- c(bitvector_list, list(bitvector))
  }
  filtered_pockets$bitvectors[[i]] <- bitvector_list
```

```
  idc <- c(wiener(g), randic_nk_f(g), szeged(g))
  names(idc) <- c("Wiener", "Hyper-Wiener", "Randic", "F", "Narumi-Katayama",
"Szeged")
  filtered_pockets$indices[[i]] <- idc
}

combined_bvs <- tibble(pdb_id = filtered_pockets$pdb_id, protein_fp =
map(filtered_pockets$bitvectors, unlist))
kin_params_full <-
  left_join(kin_params, combined_bvs, by = "pdb_id") %>%
  left_join(., filtered_pockets, by = "pdb_id") %>%
  dplyr::select(kinase, pdb_id, compound, protein_fp, compound_fp, indices, pKd,
pKon, pKoff) %>%
  filter(map_dbl(protein_fp, length) > 3) # herausfiltern der leeren Fingerprints
```

QUELLEN

[1] "Daylight Theory: SMILES," accessed July 16, 2020, https://www.daylight.com/dayhtml/doc/theory/theory.smiles.html.

[2] "InChI Trust - Developing the InChI Chemical Structure Standard," InChI Trust, accessed July 16, 2020, http://www.inchi-trust.org/.

[3] "Extended-Connectivity Fingerprints | Journal of Chemical Information and Modeling," accessed July 16, 2020, https://pubs.acs.org/doi/10.1021/ci100050t#.

[4] Subhash C. Basak, "Use of Graph Invariants in Quantitative Structure-Activity Relationship Studies," *Croatica Chemica Acta* 89 (December 1, 2016), https://doi.org/10.5562/cca3029.

[5] David J. Wood et al., "Pharmacophore Fingerprint-Based Approach to Binding Site Subpocket Similarity and Its Application to Bioisostere Replacement," *Journal of Chemical Information and Modeling* 52, no. 8 (August 27, 2012): 2031–43, https://doi.org/10.1021/ci3000776.

[6] "Permutation," in *Wikipedia*, November 25, 2019, https://de.wikipedia.org/w/index.php?title=Permutation&oldid=194357260.

[7] Francesca Perruccio et al., "FLAP: 4-Point Pharmacophore Fingerprints from GRID," in *Molecular Interaction Fields* (John Wiley & Sons, Ltd, 2006), 83–102, https://doi.org/10.1002/3527607676.ch4.

[8] Anna Artese et al., "Molecular Interaction Fields in Drug Discovery: Recent Advances and Future Perspectives," *Wiley Interdisciplinary Reviews: Computational Molecular Science.* 3 (November 1, 2013), https://doi.org/10.1002/wcms.1150.

[9] Gui-dong Yu, Li-fang Ren, and Xing-xing Li, "Wiener Index, Hyper-Wiener Index, Harary Index and Hamiltonicity Properties of Graphs," *Applied Mathematics-A Journal of Chinese Universities* 34, no. 2 (June 1, 2019): 162–72, https://doi.org/10.1007/s11766-019-3565-9.

[10] Ivan Gutman, Boris Furtula, and Vladimir Katanić, "Randić Index and Information," *AKCE International Journal of Graphs and Combinatorics* 15, no. 3 (December 1, 2018): 307–12, https://doi.org/10.1016/j.akcej.2017.09.006.

[11] Ramin Kazemi, "The First Zagreb and Forgotten Topological Indices of D-Ary Trees," *Hacettepe Journal of Mathematics and Statistics* 2 (January 1, 2017), https://doi.org/10.15672/HJMS.20174622758.

[12] Ivan Gutman, "Degree-Based Topological Indices," *Croatica Chemica Acta* 86 (December 1, 2013): 351–61, https://doi.org/10.5562/cca2294.

[13] Ali Iranmanesh, Yaser Alizadeh, and Bahman Taherkhani, "Computing the Szeged and PI Indices of VC5C7[p,q] and HC5C7[p,q] Nanotubes," *International Journal of Molecular Sciences* 9, no. 2 (February 5, 2008): 131–44.

[14] Iranmanesh, Alizadeh, and Taherkhani.

[15] K. Das, Kinkar Das, and Ivan Gutman, "Some Properties of the Second Zagreb Index," *MATCH - Communications in Mathematical and in Computer Chemistry* 52 (September 1, 2004).

[16] Aleksandar Ilic and Milovan Ilic, "On Some Algorithms for Computing Topological Indices of Chemical Graphs," n.d., 10.

[17] Wood et al., "Pharmacophore Fingerprint-Based Approach to Binding Site Subpocket Similarity and Its Application to Bioisostere Replacement."

[18] Wood et al.

[19] Victoria Georgi et al., “Binding Kinetics Survey of the Drugged Kinome,” *Journal of the American Chemical Society* 140, no. 46 (November 21, 2018): 15774–82, https://doi.org/10.1021/jacs.8b08048.

[20] RCSB Protein Data Bank, “RCSB PDB: Homepage,” accessed December 26, 2019, https://www.rcsb.org/.

[21] Doris A. Schuetz et al., “GRAIL: GRids of PhArmacophore Interaction FieLds,” *Journal of Chemical Theory and Computation* 14, no. 9 (September 11, 2018): 4958–70, https://doi.org/10.1021/acs.jctc.8b00495.

[22] “Download R-3.6.1 for Windows. The R-Project for Statistical Computing.,” accessed December 26, 2019, https://cran.r-project.org/bin/windows/base/old/3.6.1/.

[23] “UniProt,” accessed December 26, 2019, https://www.uniprot.org/.

[24] Rodrigo Dorantes-Gilardi, *Rodogi/Buriedness*, Python, 2018, https://github.com/rodogi/buriedness.

[25] “Median Absolute Deviation,” in *Wikipedia*, July 9, 2019, https://en.wikipedia.org/w/index.php?title=Median_absolute_deviation&oldid=905525974.

[26] “Igraph R Package,” accessed December 26, 2019, https://igraph.org/r/.

[27] “Sphärizität (Geologie),” in *Wikipedia*, September 22, 2019, https://de.wikipedia.org/w/index.php?title=Sph%C3%A4rizit%C3%A4t_(Geologie)&oldid=192491092.