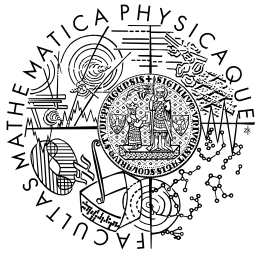


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Adam Filandr

Utilizing simulated annealing for molecular fingerprints optimization for virtual screening

Department of Software Engineering

Supervisor of the bachelor thesis:	RNDr. David Hoksza, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Utilizing simulated annealing for molecular fingerprints optimization for virtual screening

Author: Adam Filandr

Department: Department of Software Engineering

Supervisor: RNDr. David Hoksza, Ph.D., Department of Software Engineering

Abstract: Ligand based virtual screening can be realised with various molecular representations. Fragment-feature representation represents the molecules as a set of fragments, where each fragment receives a set of descriptors. First goal of this thesis is to find suitable similarity function for such representation. This representation can also be improved by assigning a weight for each descriptor, which gives it a priority in a given similarity function. The second goal of this thesis is to examine simulated annealing as an algorithm used to find the weights. We experimentally analysed the influence of various fragment types, descriptor types, similarity functions, correlated descriptors, fragment noise and parameters of simulated annealing. Because the experiments are computationally demanding, we also created a tool for large scale computations.

Keywords: virtual screening similarity function simulated annealing

I would like to thank my supervisor RNDr. David Hoksza, Ph.D. and Mgr. Petr Škoda for their advice and time. Special thanks goes to my parents and my brother for their continued support.

Contents

Introduction

Drug discovery is a process of finding a new small molecule which has desired effects on a biological system. *Virtual screening* is a method used for optimization of drug discovery. When pharmacologists want to develop a new drug, one of the problems they face is which molecule to pick for laboratory testing. There are milions of potential candidates so a method that is better than random selection would be greatly appreciated. Virtual screening is exactly that - a method that tries to select active molecules (molecules with desired effects) from a pool of active and inactive candidates *in silico*. After a researcher selects a small amount of molecules using virtual screening, they are tested physicaly in a laboratory (*high-throughput screening - HTS*), which consumes large amounts of time and resources.

Ideally all of the chosen molecules are active (pharmacologist can then pick the best one regarding side-efects, cost of production etc.). However with current state of virtual screening that is often not the case and resources are wasted by testing inactive molecules. Thus by improving this method, we could potentialy make drug discovery cheaper and side-effects less harmful.

There are two main categories of virtual screening *ligand-based* (LBVS) and *structure-based* (SBVS). Focus of our work is LBVS.

The core of LBVS is *similar property principle* [1] which states that similar compounds have similar effects with respect to given medical condition. This category requires to already know at least one active compound. We use the information stored in this set of known actives to determine the similarity between known active molecules and candidate molecules. In the end, the most similar molecules to the known active ones are selected.

There are many different ways to utilise the information stored in known actives and candidates, however our interest lies in the descriptor method. Descriptor is a representation of features and properties of molecule or molecular fragment in form of a value. Most commonly *vector of values* or *fingerprint* (vector of zeros and ones) is created from descriptors based on properties of given compound (e.g. weight or number of carbon atoms).

Again there are many different ways how to compare molecules using descriptors. In order to improve LBVS we introduce combination of two different approaches which were suggested by David Hoksza [2].

First approach - fragment-feature molecular representation: Traditionally we would create one vector of descriptors for the whole molecule - instead of that we split the molecule into fragments and assign descriptors to each one of them (the same set of descriptors for each fragment). Thus representation of the molecule is a set of vectors instead of a single vector.

Second approach - weighting the descriptors: To determine similarity between candidate molecules and known active ones, each part of descriptor vector is equally important in traditional methods. However, we assign a weight to each descriptor which gives it a priority in respect to given situation. The core of this work is to explore *simulated annealing* as an algorithm used to find the weights.

Our hypothesis is that some parts of molecules (fragments) and some molecu-

lar features (descriptors) are more important for the activity of molecule against given medical condition than others. By combining these two approaches we get a molecular representation which enables us to compare molecules based on this criteria. This is something the traditional molecular representations can not provide us. We call this approach weighted fragment-feature screening.

First goal of our work is to implement suitable *similarity function* for fragment-feature molecular representation. Second goal is to explore the capabilities of *simulated annealing* as a method for optimization of weights in weighted fragment-feature representation. To achieve both goals, the sub-goals of our work are:

- Create a tool which will enable us to easily calculate large numbers of screenings. This involves optimization of the whole process to make it run as fast as possible - usage of cluster, paralelisation of processes, better data representation.
- Use simulated annealing to optimize weights of descriptors.
- Find out which way of spliting molecule into fragments is most suitable for our method.
- Explore which descriptor type is the most benefitial for our method.
- Find suitable algorithm to compare molecules as sets of vectors.
- Analyze the influence of fragment noise, correlated descriptors and parameters of simulated annealing.

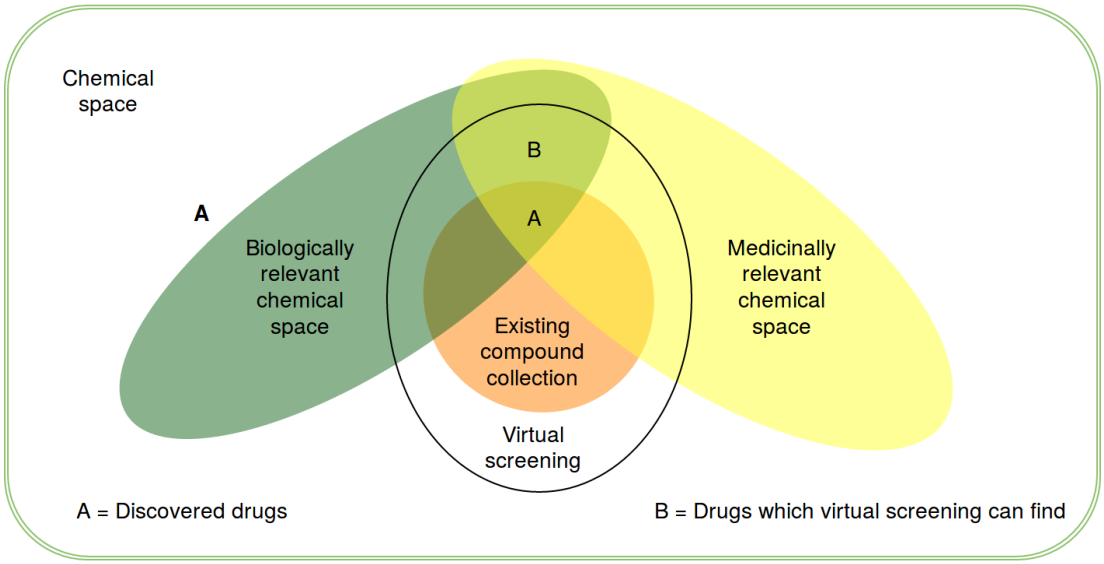
1. Introduction to virtual screening and related theory

Due to the nature of given problem and our solution, it is fitting to introduce reader into virtual screening as a whole and some related theory. In this chapter, we first explore the theory and categories of virtual screening, then we shift focus to molecular descriptors and fragments, take a quick look at one popular way used to compare the performance of various virtual screening methods and at last we explore the simulated annealing.

1.1 What is virtual screening and its categories

The problem of finding a molecule with certain desired effects can be ilustrated by the concept of *chemical space* 1.1 which is an imaginary space containing all possible chemical compounds and "can be viewed as being analogous to the cosmological universe in its vastness, with chemical compounds populating space instead of stars" [3]. To imagine the size of chemical space one estimate states that there are more than 10^{63} molecules with less than 30 atoms and molecular weight less than 500, which are stable at room temperature and stable against oxygen and water [4]. Thus virtual screening acts as a telescope of sorts, which enables us to look at different compounds in this chemical universe and predict their effects or even predict the existence of certain compounds. The laboratory testing of those compounds (*high-throughput screening* - HTS) would then be equivalent of flying to those places in a spaceship.

Figure 1.1: Diagram of chemical space - ilustration from [5]



Target is a short for *biological target* - any structure in organism (e.g. protein or nucleic acid) to which a molecule can bind, resulting in changes of behavior or function of this structure in given organism. Essentially goal of virtual screening is

to find new molecules which can bind to a given target and we call these molecules active in respect to given target. *Decoys* are known inactive molecules and *ligands* are known active molecules (in respect to given target). Set of molecules in which we are searching for new active compounds is called set of candidate molecules.

New active molecules which we found also need to fullfil other requirements in order to be useful. One is often interested in molecules which structurally differ from known active compounds - the motivations vary, but the most common ones are to find molecule stronger in activity, molecule with different side-effects or to generate new intelectual property that is different enough from their competitors. Other requirement for new active molecules is to be *druglike*. Druglikeness determines a set of characteristics necessary for compound to be safe and orally administered (absorption, permeability, metabolic stability, toxicity and others) [5]. All of those requirements can be adressed by virtual screening too, however in our work we focus only on identification of new active compounds.

Since 1970s virtual screening diversified into many different methods. The two main categories of approaches are *ligand-based* (LBVS) and *structure-based* (SBVS) virtual screening.

Underlying principle of LBVS is *similar property principle* [6] which states that similar molecules function in similar way. One can quickly find arguments against this principle - small changes in compounds resulting in great changes in biological activity or large structural changes resulting in little to no change in activity, but this principle is generally accepted and it has been statistically demonstrated [7]. However what exactly is a "similarity" between molecules is not defined - the whole meaning changes depending on how we describe molecules and how we compare molecules according to this description. For example, we could describe molecules by their molecular weights and similarity function could be just a difference of these weights. Other approach could describe molecules as set of substructures and similarity could be defined as the number of shared substructures divided by the total number of substructures. Thus LBVS methods are mainly distinguished by the method of describing a molecule and by the method of comparing a molecule using this description. Three main sub-categories of LBVS exist - *similarity searching*, *compound classification* and *compound filtering* [8].

Similarity searching includes namely our approach, volume/surface matching, substrucutre searching, pharmacophore searching and descriptor methods. Although they all differ in how to describe and compare molecules, the general approach stays the same. With at least one already known active compound (compound classification requires multiple known actives) the screening can begin: Create representation of known actives and candidate molecules, compare candidate molecules against known actives, evaluate the results, select small number of molecules for HTS. In our work we focus on the descriptor method.

The compound classification sub-category includes methods such as clustering/partitioning, mapping and various machine learning methods. However this sub-category is not utilised in our work.

Compound filtering is often used as pre-screening and profiling of large compound libraries. The goal is usually to filter out undesired molecules such as reactive and toxic ones. Filtering can be also used for estimating basic ADME (Absorption, Distribution, Metabolism, Excretion) - which are parameters re-

quired for orally avaliable medicine (Rule-Of-Five [9]). We also did not use this sub-category.

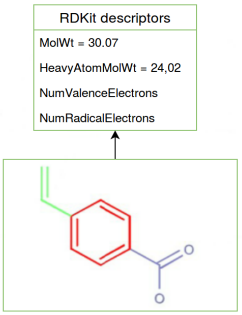
Even though we do not focus on SBVS in our work, quick overview can be useful. The SBVS is based on docking the candidate molecules into a target (e.g. protein). This process involves creation of target model from 3D structural information and selection of binding site. With the model prepared, each compound of library is docked into the target. The priority of candidate molecule is determined by the ability of binding to the target with high affinity, estimated by scoring a function for docking. In the end the molecules that bind the best are selected. As always there are multiple approaches on how to dock and score molecules. This category is still very computationaly demanding - LBVS is much less demanding and also receives greater attention. Interested reader can find more information in publication by Lionta [10].

1.2 Ligand based virtual screening with descriptors

The most inspiration was drawn from this approach. "The molecular descriptor is the final result of a logic and mathematical procedure which transforms chemical information encoded within a symbolic representation of a molecule into a useful number or the result of some standardized experiment." [1] In other words descriptors represent features and properties of a molecule in form of a value. They can be than formed into vector of values or *fingerprint* (vector of zeros and ones).

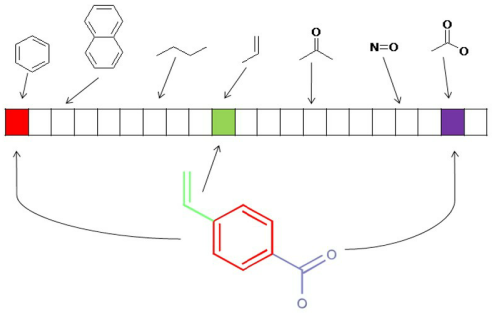
There is a great variety of descriptors and they are mainly categorised by their dimension. 1D descriptors describe molecular properties such as weight or number of carbon atoms. The presence, absence, topology or number of different substructures (*molecular fragments*) are 2D descriptors. 3D descriptors encode shape and functionality [11].

Figure 1.2: Ilustration of descriptors - from [12]



Descriptors which are formed into vector of values 1.2 are generated by software such as PaDEL [13], RDKit [14], JOELib, Chemistry Development Kit (CDK) [15] and Chemical Descriptors Library [16]. Each part of vector represents value of certain chosen 1D, 2D or 3D descriptor. We use PaDEL and RDKit descriptors in our work.

Figure 1.3: Ilustration of fingerprint, colorful bits are set to 1, the rest is set to 0 - from [12]



Fingerprints put simply are bitstrings, where each bit represents presence or abscene of certain molecular feature, substructure, range of values or other feature 1.3. Types of fingerprint descriptors include substructure-keys based (e.g. MACCS), topological torsions based [17] (e.g. Daylight) and circular based (e.g. ECFP [18]). Fingerprints can also be folded. Fingerprints however are not in the focus of our work.

As said earlier LBVS with descriptors is included in sub-category of LBVS called similarity searching. It also follows the general similarity searching approach 1.4:

1. Create representation for known actives and candidate molecules = assign descriptors to known actives and candidate molecules
2. Compare candidate molecules against known actives = use appropriate similarity/distance function between known actives and candidate molecules using representation from previous step.
3. evaluate the results = use appropriate scoring function
4. select small number of molecules for HTS

1.3 Molecular Fragments

Because we use molecular fragments in our approach we will take a quick look at the concept here. Molecular fragment is any form of continuous substructure of given molecule, however, we focus on fragments which follow certain rules - namely *linear fragments* and *circular fragments*.

Linear fragments can be imagined as paths through the molecule of certain lengths where length is the number of atoms in path. For example linear fragments of length 1 are all single atoms in a molecule, linear fragments of length 2 are all atom pairs in a molecule. In our work we use linear fragments of length 2, 3 and 4.

Circular fragments consist of substructures of certain radius around atoms where radius is measured in number of atom bonds 1.5. For example circular fragment of radius 0 are all single atoms in a molecule, circular fragments of radius 1 consist of central atom and all of its direct neighbours (other atoms connected by bond with the central atom of fragment). In our work we use circular fragments with radius of 1 and 2.

Figure 1.4: Diagram of LBVS with descriptors

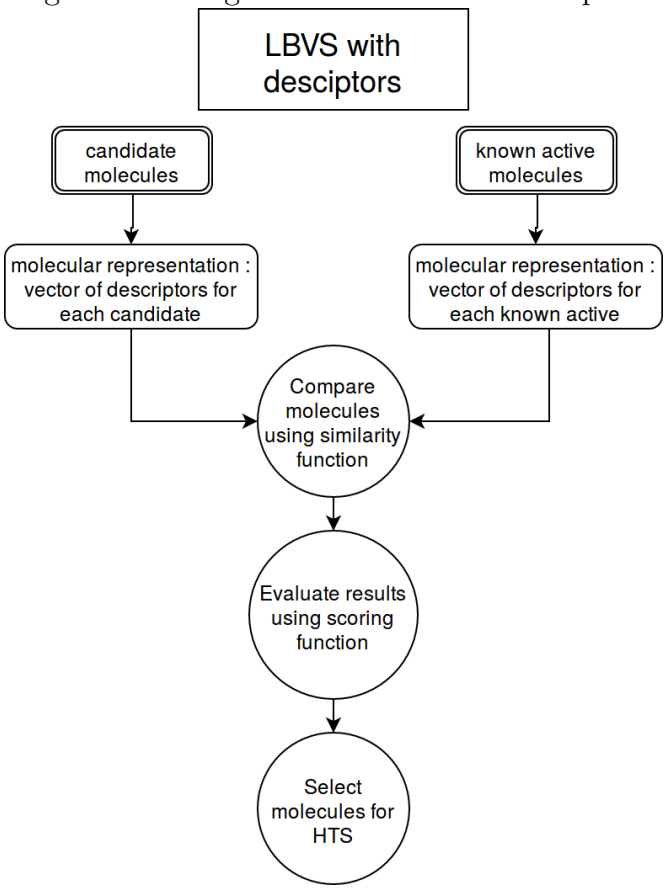
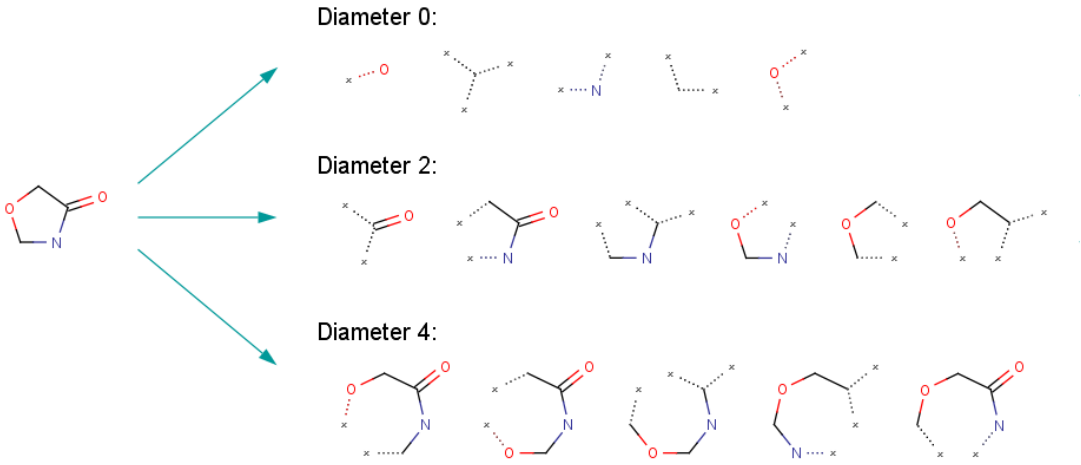


Figure 1.5: Ilustration of circular fragments - from [19]



One can also think about molecular fragment as a connected induced subgraph (following certain rules) of a connected graph (which represents the molecule).

1.4 RDKit and PaDEL descriptors

Both PaDEL [13] and RDKit [14] are open source software for calculating molecular descriptors which we use in our work.

PaDEL currently calculates 797 desciptors (663 1D and 2D, 134 3D descriptors) and was programmed using Java language. Advantages of PaDEL are that it provides both GUI and command line interface, can work on any platform that supports Java (Windows, Linux, MacOS), supports more than 90 different molecular file formats and the speed of computation is quite good (thanks to the usage of multi-thread computation). The disadvantage of PaDEL is that it does not calculate as many descriptors as other software such as DRAGON, Molconn-Z, MODEL and PreADMET Descriptor.

RDKit has core algorithms written in C++ and wrappers in Python, Java and C#. RDKit has more general purpose than PaDEL, because it can provide other functions besides just descriptor calculation (e.g. molecular fragment extraction). Advantages of RDKit include the support of many input and output formats and ability to work on all major platforms (Windows, Linux, MacOs). However RDKit calculates only 192 descriptors which is even less than PaDEL.

RDKit and PaDEL assign descriptors to any molecular structure, meaning we can assign descriptors either to complete molecules or to molecular fragments.

1.5 Scoring - ROC and AUC

With such a great number of different LBVS methods the question arises, how to compare performance of those methods. One popular way is to calculate area under the curve (AUC) of receiver operating characteristic (ROC) which gives us single number evaluating used method.

ROC curves are used to identify the diagnostic ability of a binary classifier system. The classifier system receives different positive or negative test examples and assignes them positivity or negativity using some algorithm. In the end we can determine how often the system is *true positive* (TP - predicts positivity in positive example), *true negative* (TN predicts negativity in negative example), *false positive* (FP predicts positivity in negative example) and *false negative* (FN predicts negativity in positive example). The curve is created by plotting the *true positive rate* against the *false positive rate*. True postitive rate (sometimes called sensitivity) is defined as $\frac{TP}{TP+FN}$ and the false positive rate (sometimes called specifity) is defined as $\frac{TN}{TN+FP}$

In other words, ROC curve tells us how well can the algorithm of our classifier separate active molecules from the inactive ones. To be able to do that, our method (classifier) needs to be ranking - molecules with higher rank have to be more likely to be active. In our approach we rank molecules by the similarity to an already known active molecule.

The algorithm to plot a ROC curve goes as follows:

- Rank each molecule and sort them in decreasing order
- Start at $s = (x, y)$ where $x = 0$ and $y = 0$
- For each molecule m (using the decreasing order)
  If m is active $x+ = 1$
  If m is inactive $y+ = 1$
  Plot coordinate s

On this graph the y-axis represents the true positive rate and the x-axis represents the false positive rate.

AUC (sometimes also AUROC) is used to convert the ROC curve into a number. By computing the area under the ROC curve we give the curve a score. The bigger the score is the better. Perfect AUC equals to 1 - all active molecules are ranked before any negative one. The worst AUC equals to 0 - all inactive molecules are ranked before any active one.

The AUC value is popular way to measure the performance of different virtual screening methods because the only requirement is to have a ranking classifier. One feature of AUC which reader could notice is that only the order of test examples (molecules in our case) is important, not the exact value of classifier. The disadvantage of this feature is that if for example half of all molecules were 10x more similar than the other half and contained all of the active molecules, we would not notice, even though this is probably useful information.

1.6 Simmulated Annealing

Simmulated annealing is a method used to find global optimum of a function. We use this method to find weights of the molecular descriptors. In order to explain how simulated annealing works, lets compare it to hillclimbing. The algorithm for hillclimbing (when searching for maxima) is following:

Lets define:
$f : R^n -> R$ function for which we want to find global maxima. $x, y \in R^n$ vectors. $N : R^n -> R^n$ neighbour function.

1. $x = some\_initial\_position$
2. $y = N(x)$
3. If $f(y) > f(x)$ then $x = y$
4. Go to step 2.

This algorithm can work nicely, if our function has no local maxima. However if there are local maxima, the chance is, we will get stuck in those (and our function has probably many of those). In order to avoid this problem one would like to somehow jump out of the local maxima but in the end stay in the global maxima.

Which is where simulated annealing comes in. Instead of always moving to better neighbour position, we also allow to sometimes move into worse neighbour position - we define the *acceptance criterion*. The crucial part is, that the acceptance criterion changes in time. This is called the *cooling schedule* and the variable that affects the acceptance criterion is called *temperature*. A cooling chedule is specified by the initial temperature, decrement function for lowering

the value of temperature and final value of temperature at which the annealing stops. [20] At first with high temperature we want to move into worse positions relatively frequently but as time goes on and temperature cools down we will behave more and more like hillclimbing.

The idea is that we jump out of local maxima at the begining and towards the end we climb the global maxima (or value close to global maxima). This concept is inspired by annealing in metalurgy, where controlled cooling yields metals with different crystal structures (and different desired properties). The algorithm for simulated annealing goes as follows:

Lets define:

$f : R^n -> R$ function for which we want to find global maxima. $x, y \in R^n$ vectors $N : R^n -> R^n$ neighbour function $P : (R, R, R) -><0, 1>$ acceptance criterion T temperature C decrement function

1. $x = some\_initial\_position$
2. $y = N(x)$
3. $x = y$ with the probability of $P(f(x), f(y), T))$
4. $T = C(T)$, go to step 2.

When implementing this algorithm we need to solve mainly three different issues which are specific for the kind of function for which we want to find global optima: how to implement neighbour positions function, acceptance criterion and decrement funtion. All of those questions are answered in chapter 3.1.

2. Our approach to similarity searching LBVS

With underlying concepts such as molecular fragments, descriptors and scoring explored in previous chapter, we can finally put together our approach. In this chapter, we focus on individual pieces of our method: the way of representing a molecule, why we think this approach can be succesful, the concept of weights and similarity functions.

2.1 Fragment-feature molecular representation

The core of every ligand based virtual screening method lies in the molecular representation. The traditional way is to describe a molecule as a whole using set of 1D and 2D descriptors in form of vector or a fingerprint while each descriptor has the same priority in comparison.

Our molecular representation differs significantly. Instead of describing features of a molecule as a whole, we describe features of fragments of given molecule. Same set of descriptors is used for every fragment of each molecule - this allows us to compare any pair of fragments. Duplicit fragments are removed from molecules, since they decrease the performance - each molecule would be full of basic fragments like strings of carbon atoms and unique fragments would be hidden.

Now each molecule is represented by a set of vectors (instead of single vector), where the size of the set depends on number of fragments extracted from molecule (which depends on size of given molecule) and dimension of vectors depends on number of descriptors used. This approach allows us to look at molecule from different perspective - we can for example determine that two molecules, which are overall different, are very similar in half of their fragments, even if those fragments are not identical. Perspective such as this can be crucial, if those fragments are responsible for activity of the molecule. This distinction is not possible with traditional representations.

This representation is called fragment-feature molecular representation and was proposed by Hoksza and Škoda in [2].

Creation of representation consist of two steps:

1. Split molecules into suitable fragments
2. Calculate descriptors for each fragment

First step:

One can split a molecule into fragments in many different ways, so how to decide which way is the most suitable? First we had to consider which technical solution is best for us - we decided to use RDKit to generate linear and circular fragments. Now with our focus narrowed to linear and circular fragments we chose to explore linear fragments of lengths two, three, four and circular fragments of radius one and two. Larger fragments for both types were computationally

too demanding. Which type of fragments is most suitable one is determined experimentally in section 5.3.1.

Second step:

PaDEL and RDKit were chosen to calculate descriptors. The reason being both PaDEL and RDKit are opensource and PaDEL calculates greater number of descriptors than RDKit. We compare how the number of descriptors influence performance of our method experimentally in section 5.3.1.

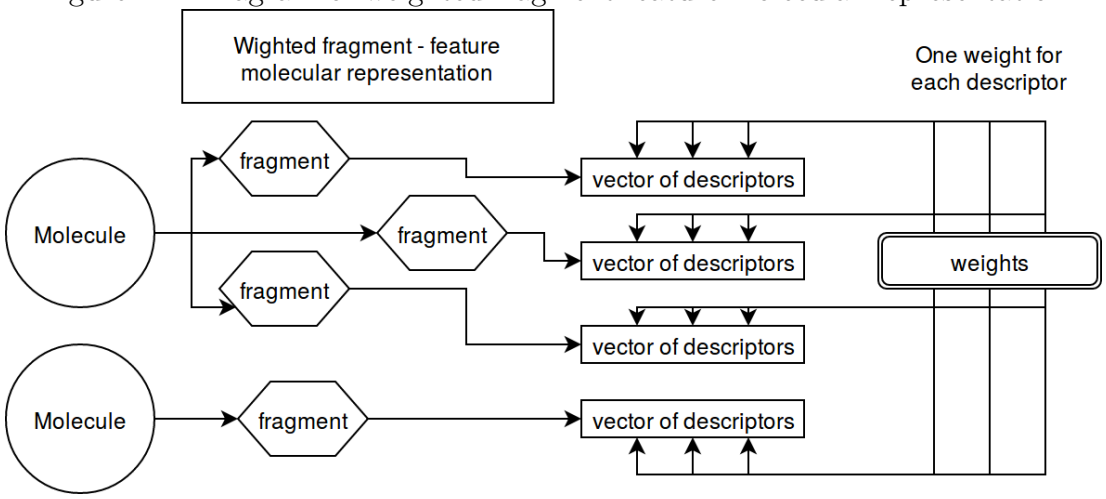
With this representation in hand we can then add weights to individual descriptors and compare two molecules using a similarity function on which we focus in section 2.5.

2.2 Weights

In traditional methods, each descriptor used in molecular representation has the same priority during comparison of molecules. In our approach each descriptor is given weight which influences its priority when comparing two fragments 2.1.

Weights essentialy allow us to specify which features are more important when deciding similarity to known active compounds than others (e.g. difference in the number of carbon atoms would affect similarity more than difference in the number of hydrogen atoms). We hypothesize that when the right features are prioritised above others (in a meaningful way) we can improve the precision of similarity search between known actives and candidate molecules.

Figure 2.1: Diagram of weighted fragment-feature molecular representation



It is important that those weights are not universal for all molecules, but are specific for given known active molecules and a target. Essentialy each pair of known actives and a target has its own set of weights, which helps to identify other active molecules in a candidate set. The reason being that for different known sets of active molecules and different targets the features which are important are different too.

2.3 Why use fragments

Lets compare the use of fragment-feature representation to other methods.

First lets discuss fingerprints (topological torsions, ECFP, MACCS). They are somewhat similar in nature thanks to the focus on fragments - although fingerprints focus mainly on presence or absence of fragments. However, our method can distinguishe how similar fragments are. We think that comparison of active and candidate molecules should focus more on presence or absence of crucial properties of fragments rather than on presence or absence of whole fragments.

Lets imagine this situation: we have three molecules - one known active and two candidates - and we take two pairs active+candidate. We also calculate that having an aromatic ring is crucial property for active molecule so we give this descriptor high weight. First pair does not share any fragments, but some fragments of candidate molecule have aromatic rings. Second pair share many fragments but not a single fragment of candidate molecule has aromatic ring. If we used fingerprints, we would declare the second pair much more similar than the first one. With our approach the first pair can be more similar than the second one (when the right similarity function is used). If in this situation having aromatic ring really was the reason for activity of a compound, our method would prove to be more useful.

Second, lets take a look at traditional descriptor methods (single vector of descriptors for molecule). We think that the ability to look at molecule in fragments is more useful than the global way of looking at a molecule. For example if we used one descriptor per molecule and determined that two molecules have similarity of 0.5 (on a scale from 0 to 1) using Euclidean distance, we would not know, if the two molecules are similar in mediocre way overall or the two molecules have parts which are completly identical and other parts which are completly different. With our method we can tell the difference (with the right similarity function) and even tell, if the similar side is important thanks to weighted descriptors. If only a part of molecule is important for a molecule to be active, our method would be more useful.

Another advantage of our method is the ability to change what similarity means with great range of similarity functions which can work with this representation.

However the disadvantage of our method is that it is more computationally demanding than the other methods.

2.4 Why use weights and how to find them

Lets suppose we would like to compare humans for various reasons. Each human is described by a set of descriptors - i.e. height, inteligence, strength, facial features, genetic code. Now we would like to know the chance that 2 random people are related. One option is to take in account all features and estimate, that people which are the most similar over all should be related. This is obviously not so great idea. We can clearly see, that we have a descriptor ideal for this comparison - the genetic code. Other features can throw our calculation off.

Now lets compare people to a basketball player and calculate the chance, that this person is also a basketball player. In this case it is not so clear, which descriptors are important, but facial features will probably be the least important descriptor.

For different comparisons different descriptors are useful which also applies to molecules. The difference is that when comparing molecules, it is almost impossible to determine, which descriptors are useful and which are not in advance. Althought almost certainly not every feature should be equally important. That is why each descriptor has a weight which describes how much important it is (0% as not relevant to 100% maximum importance) .

So now we know that we want to assign weights to descriptors. The question is how to find the best weights. We can imagine virtual screening being multidimensional continuous function which gives us a value from 0 to 1 (AUC value) for each set of coordinates (different weights). We would like to find a set of coordinates which gives us the global maxima of AUC in this multidimensional space. If we had only 3 descriptors, we can imagine that we search for highest peak in a cube which contains some sort of a landscape. If we had infinite computational power we could just calculate AUC for each set of weights and find the maximum value. However since we do not have such luxury, we need to find some other method. This is common optimisation problem and the variety of solutions is great - hillclimbing, beamsearch, simulated annealing, tabu search, harmony search, genetic algorithms, parallel tempering etc. We chose to use simulated annealing. The implementation of simulated annealing is described in section 3.1.

2.5 Similarity Functions

With our representation created we now face the problem of choosing the right similarity function for comparing molecules. Since we represent molecules as sets of fragments and we represent fragments as vectors, essentially we want to determine the similarity of two sets of vectors. Different similarity functions allow us to take different perspective and influence outcome of the method greatly - for example we can focus on average similarity of fragments (global perspectvie) or focus on the most similar fragments (narrow perspective) etc.

No matter which similarity function was used, the final similarity assigned to a candidate molecule is always calculated as $\max(s(x, a) : a \in A)$m where s is the similarity function, x is candidate molecule and A is set of known actives. Formulas in this section use following symbols: M_1 and M_2 are compared molecules, f_1 and f_2 are fragments from those molecules, $Descriptors$ are indexes which identifies parts of vector of descriptors, $weights$ is an array of weights, one weight for each descriptor. The $f_1(d)$ and $f_2(d)$ are values of d descriptor in their fragments, $|M_1|$ and $|M_2|$ is the number of fragments of molecule M_1 or M_2. We implemented 6 similarity functions:

2.5.1 Euclidean Similarity

Euclidean similarity is a popular way of computing similarity between two sets of vectors. We simply measure the euclidean distance between all fragments from the two compared molecules and average the sum to get a distance of molecules and then substract that from 1 to get similarity:

$$1 - \frac{\sum\limits_{f_1 \in M_1, f_2 \in M_2} \sqrt{\sum\limits_{d \in Descriptors} (f_1(d) - f_2(d))^2 * weights(d)}}{|M_1| \times |M_2| \times \sqrt{|Descriptors|}} \tag{2.1}$$

For every pair of fragments f_1 and f_2 from molecule M_1 and M_2 we compute the euclidean distance of the descriptors vectors. We then average the sum by $|M_1| \times |M_2|$ (number of pairs of fragments from both molecules). The division by square root of the number of descriptors will scale our value into [0,1] - since each descriptor can have maximal value of 1 (thanks to scaling of descriptor values which we did in the pre-processing phase) $|\sqrt{Descriptors}|$ is the maximal value which euclidean distance of two molecules can have. For purpose of our work we also added weights to the equations.

Time complexity of this method is $O|M_1| \times |M_2| \times |Descriptors|$ - which is the number of $(f_1(d) - f_2(d))^2$ calculated.

2.5.2 Manhattan Similarity

Manhattan similarity uses taxicab geometry in which distance of two points is defined as the sum of diferences of each dimension. The whole formula is:

$$1 - \frac{\sum\limits_{f_1 \in M_1, f_2 \in M_2} \sqrt{\sum\limits_{d \in Descriptors} |f_1(d) - f_2(d)| * weights(d)}}{|M_1| \times |M_2| \times |Descriptors|} \tag{2.2}$$

Thus the only difference between Euclidean and Manhattan similarity is the change of $(f_1(d) - f_2(d))^2$ to $|f_1(d) - f_2(d)|$ when calculating fragment distance and change of division by $\sqrt{|Descriptors|}$ to $|Descriptors|$. Time complexity stays the same as $O|M_1| \times |M_2| \times |Descriptors|$.

2.5.3 Simple Matching Similarity

This strategy utilises binned values of fragment vectors. Binning is done in the pre-processing phase. For each descriptor we find the minimal and maximal value in all active and candidate compounds. Minimal value goes to the first bin, maximal to the last bin and values in between are assigned accordingly depending on the number of bins. Similarity of two fragments is then computed as number of shared bins divided by number of all bins. Similarity of two molecules is then determined as mean of fragment pairs similarities. Again weights were added for the purpose of this work:

$$\frac{\sum\limits_{f_1 \in M_1, f_2 \in M_2} \sum\limits_{d \in Descriptors} \Big(Indicator(f_1(d), f_2(d)) * weights(d)\Big)}{|M_1| \times |M_2| \times |Descriptors|} \tag{2.3}$$

$$Indicator(f_1(d), f_2(d)) = \begin{cases} 1 & \text{if } f_1(d) = f_2(d) \\ 0 & \text{if } f_1(d) \neq f_2(d) \end{cases} \tag{2.4}$$

The disadvantage of this method is that we do not measure how different the values are, we just determine if they are or are not similar enough (the threshold is given by the number of bins). Time complexity is $O(|M_1| \times |M_2| \times |Descriptors|)$.

2.5.4 Nfrag Similarity

The disadvantage of Euclidean, Manhattan and Simple similarity is that two identical compounds do not have the similarity of 1 (unless they have only 1 fragment), so two identical molecules can be less similar than two different molecules. This can lead to major errors when used to compare candidate molecules and known active ones.

Nfrag similarity attempts to solve the problem stated above. It can be used with any distance function. Instead of calculating distance of two molecules as average of all fragment pairs distances we choose N of the most similar pairs. In fact the number N is different for every pair of molecules:

$$N = \left\lfloor \max(|M_1|, |M_2|) * \frac{(P}{100)} \right\rfloor \tag{2.5}$$

Formula for similarity:

$$1 - \frac{\sum_{i=1}^{N} SortedFragDist(i)}{N \times \sqrt{|Descriptors|}} \tag{2.6}$$

$$SortedFragDist = Sort\left(\bigcup_{f_1 \in M_1, f_2 \in M_2} Distance(f1, f2) \right) \tag{2.7}$$

Distance is function which computes distance of two fragments using all descriptors - we use simple, Euclidean and Manhattan distance.

Sort is any sorting function which sorts a set of numbers in descending order.

This way two identical molecules have the similarity value of 1, because for every fragment of one molecule exist the same fragment in the other. Also because we throw away duplicate fragments (in the pre-processing phase) the only other way of obtaining the similarity value of 1 would be by comparing two molecules which have only 1 type of fragment, however, in different numbers. For example the fragment is pair of carbon atoms and M_1 consist of two of those fragments and M_2 consist of three of those fragments - these molecules are not exactly the same but our method would give them similarity of 1. Which certainly is a disadvantage of this method, however, this situation does not occur very often.

Just note that for any other P than 100 this method will again have the same disadvantage as simple, Euclidean and Manhattan similarity.

Time complexity of this method is $O(|M_1| \times |M_2| \times |Descriptor| + |M_1| \times |M_2| \times \log(|M_1| \times |M_2|))$ where first half of the sum represents the time complexity of fragment pairs comparison while the second half represents sorting of the distance to obtain $SortedFragDist$.

3. Finding the weights

In this chapter we examine the simulated annealing, descriptor scaling and removal of fragment noise and corelated descriptors. We also take a look at training and testing process and in the end present our method as a whole.

3.1 Implementation of simulated annealing

Essentialy weights are used to maximize the performance of our LBVS approach, which is measured in AUC. In each iteration of simulated annealing we calculate AUC for given weights and then decide if we keep this new position or not (as explained in section 1.6). Thus each iteration calculates the whole virtual screening.

As such the function for which we want to find global maxima is the whole virtual screening $f : R^n -> R$, where f transforms vectors of weights (n is the number of descriptors) into AUC values. As explained in section 1.6, we need to solve the issue of neighbour function, acceptance criterion and decrement function.

For the neigbour function we define *dim* as number of weights in a vector of weights to change (number of dimensions to change - user can configure this number). Weights can not get below 0 or above $1/n$, where n is the number of weights. $rand(x, y)$ is a function which generates random integer in the interval $< x, y - 1 >$. *dist* is a constant implemented as $\frac{1}{n} * \frac{neigbour\_distance}{100}$, where *neigbour_distance* is a value in interval <0,100 >which user can configure. The neighbour function:

- For i in range(0,dim):
  $descriptor = rand(0, n)$
  $weights[descriptor] += (-1)^{rand(0,1)} * dist$

The acceptance criterion is implemented as

$$P(f(x), f(y), T) = \begin{cases} 1 & \text{if } f(x) =< f(y) \\ exp\left(\frac{f(y)-f(x)}{T}\right) & \text{if } f(x) > f(y) \end{cases} \tag{3.1}$$

The decrement function is implemented as $C = T * cooling$, where c is a constant which user can configure (usualy 0.999). User can also configure the starting temperature T. The temperature at which simulated annealing stops is $T = 1$.

Inspiration for these functions was drawn from [20].

We tested this implementation by limiting our method to only 15 descriptors and seetting *neigbour_distance* to 25 - thus weights could obtain 4 values. For this configuration there exists $4^{15} = 1073741824$ possible weights. We then searched for weights from random starting position using cca. 9200 iterations (T=10000, *cooling*=0.999).

Already 1073741824 combinations are too much for us to calculate, since one screeing takes about 0.3 seconds. However we were able to find the same weights from random positions almost all the time using simulated annealing (they occasionaly varied in value of one or two weights slighty). We take this as a confirmation that the algorithm works.

Charts of simulated annealing with various configurations can be found in section 5.3.4.

3.2 Descriptor scaling and reducing fragment noise

Simmulated annealing can be very computationally demanding therefore finding a way how to speed up similarity search is crucial for our approach. In order to improve performance we can use similarity function with better time complexity, reduce the number of descriptors or reduce the number of fragments. We decided to reduce the number of descriptors by erasing *correlated descriptors* and reduce the number of fragments by reducing *fragment noise*. Reducing fragment noise should also improve the accuracy of our method. Other pre-processing consist of *descriptor scaling* and creating better *data format*. When used for simple matching similarity method we also binn descriptors in pre-processing phase.

Descriptors scaling:

Descriptors need to lie in the same range in order to affect similarity in the same way (raw descriptors lie in various ranges). By scaling them to the range of 0 to 1, final similarity will range from 0 to 1 too. Scaling is done by finding out maximal and minimal value of given descriptor in all candidate and known active molecules and calculating scaled descriptor as: $scale(x) = \frac{x-min(d)}{max(d)-min(d)}$ where x is a value of descriptor d and $min(d)$ or $max(d)$ are minimal or maximal values of given descriptor from all fragments.

Reducing fragment noise:

Some fragments are much more frequent than others - for example a string of carbon atoms as a fragment lies in nearly 50% of molecules (when the molecule is split into linear fragments). We think that these fragments act as an information noise which can throw our method off. Thus we erase fragments which are present in certain percentage of molecules. We examine this feature experimentally in section 5.3.3.

3.3 Removing correlated descriptors

It seems like not every descriptor is useful for similarity searching LBVS. One can for sure erase descriptors which have constant value for all fragments (they do not contribute to any of our similarity functions). By erasing correlated descriptors with the right correlation treshold we should get rid of only those that are not so important. The reason for removing correlated descriptors is following: when using for example RDKit descriptors, fragments receive 192 different descriptors. To find weights for representation which uses 192 descriptors we would need to search in space with 192 dimensions. This number is too big for practical

implementation of simulated annealing. We examine experimentaly the influence of correlated descriptors in section 5.3.2.

At first we remove constant descriptors - they do not contribute to similarity search. After that we eliminate correlated descriptors is done usign the correlation matrix C, in which $C_{i,j} = corr(D_i, D_j)$, where $corr$ is a correlation function between two vectors. D_i and D_j are vectors containing i-th and j-th descriptor from each fragment. For example first RDKit descriptor is weight of molecule, so D_1 would containt weight of every fragment. Thus the size of each vector D is n, where n is the number of unique fragments of the set of molecules for which we want to remove correlated descriptors. Vectors D are ordered by fragments.

The size of C is $m * m$, where m is the number of descriptors. The matrix contains value for every descriptor pair. We used *numpy.corrcoef* asi $corr$.

With C we can combine correlated descriptors into clusters and pick only one descriptor representing each cluster. At first we create a cluster for each descriptor, then we merge clusters in which each descriptor pair is correlated (in absolute value) above a treshold, which the user defines. We are done after we can no longer merge any cluster pair.

We remove correlated descriptors only in the training phase. In test phase we, instead of removing correlated descriptors, remove those descriptors, which were not used in training phase. This way we ensure that weights match with descriptors in training and test phase (since in test data different descriptors could correlate).

3.4 Training and testing weights

In order to simulate different situations and evaluate method more accurately we define a split - a subset of known active and inactive molecules in respect to a target divided into train and test molecules. Thus we calculate unique weights for each split.

Now lets look what situation the split is simulating. Reader can notice that we have complete information about all molecules - we know exactly which are active and which are not. This is not how real world application of virtual screening looks like, however we need this information in order to evaluate how well our approach to virtual screening is performing. The situation which splits are simulating is that some researcher might know only about some subset of known active molecules and his chosen candidate molecules might consist of some other subset of active and inactive molecules. Our approach needs to work well with different subsets of molecules - by evaluating different splits we can be sure that we did not just have luck with certain subset.

The search for weights is performed using train data for each split. From train data 1/4 of the actives is used as known actives and 3/4 of actives are used as candidates among with the inactive molecules.

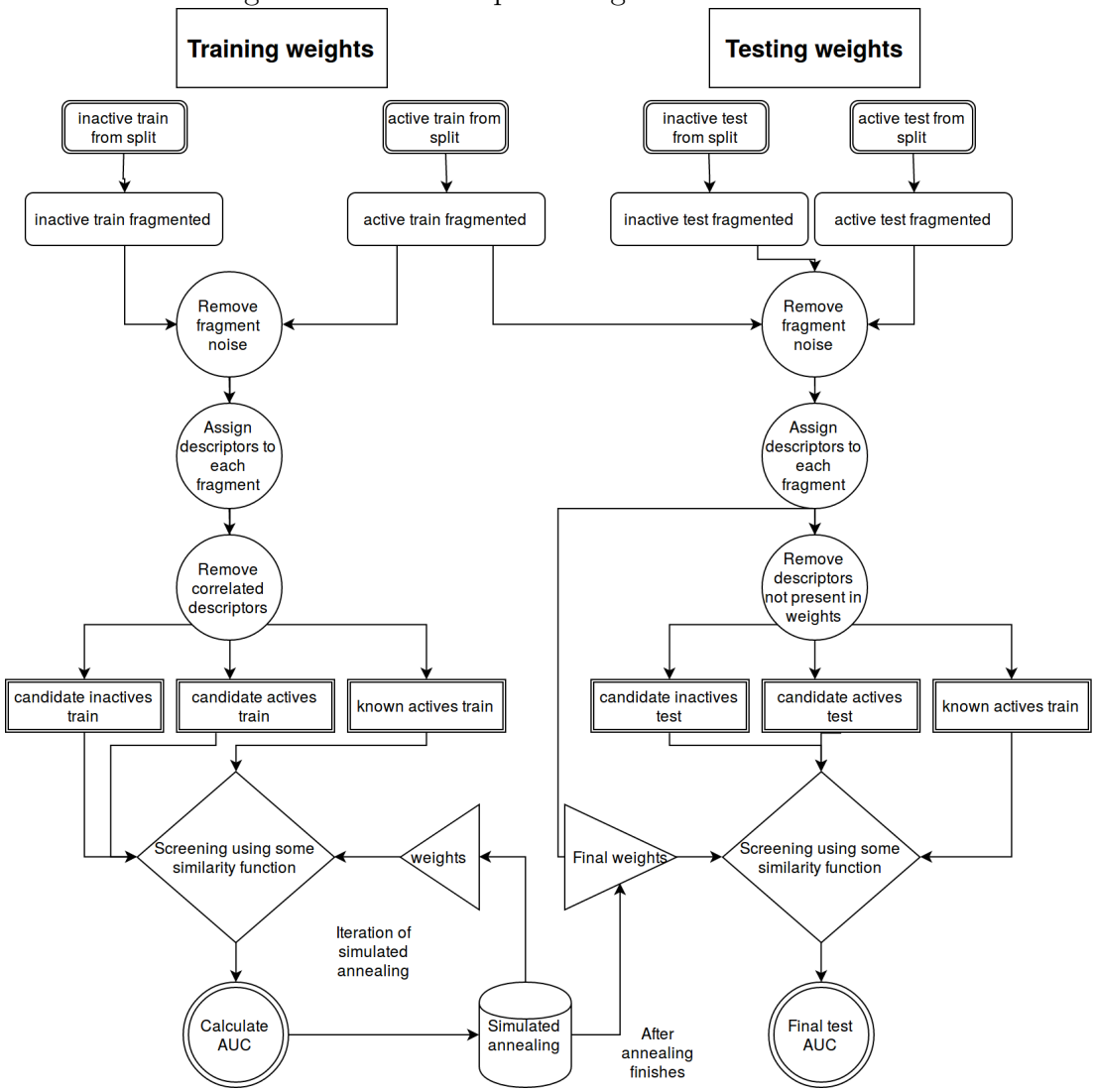
After we find the weights, we want to see if they work on some set of molecules other than the train data. This is after all our goal - to find weights for given set of known actives which will improve our method when used with different candidate molecules. For this purpose we use the test data. Active and inactive

test molecules act as candidates and active molecules from training phase act as known actives.

3.5 The whole method put together

Now we can take a look at our method as a whole in the diagram 3.1.

Figure 3.1: The complete diagram of our method



4. WeFrag

When designing our program WeFrag we focused mainly on the ability to easily configure various aspects of our approach to virtual screening (e.g. type of fragments or type of similarity function) and compute those configurations fast. The reason for this is that we needed to perform many experiments in order to find how much certain options influence the performance and find the best configuration.

To easily perform great number of different computations WeFrag offers client and server side of application for paralel computing on cluster. The user interface enables the user to quickly assign several thousands of different computations (using grid options) which will be then dynamicaly distributed and computed. Results are saved in client computer and can be aggregated into a .csv file using built-in function. WeFrag also provides user with various features regarding the maintenance of cluster - namely monitoring of running computations and the ability to update data and code on servers.

When distributing computations (tasks) on the cluster, we also monitor current state of every cluster computer (usage of memory and CPU) in order not to disturbe other users of cluster.

WeFrag also provides user with option of local computation, when cluster is not avaliable or user wants to run just a small number of computations. Local computations can be again assigned through menu, but there is also the option to use only command line arguments.

More information about the implementation of WeFrag can be found in section B. The whole program is included in attachement.

4.1 All options WeFrag provides

Options regarding input:

- Input through menu using cluster.
- Input through menu using local computer.
- Input through command line using local computer.

Capabilities regarding virtual screening and simulated annealing:

- To select targets from data, which the user provides.
- To select which splits to compute.
- To choose from linear fragments of length 2, 3 and 4 and from circular fragments of radius 1 and 2.
- To choose from RDKit or PaDEL descriptors.
- To set fragment noise reduction treshold percentage.
- To set descriptor correlation treshold percentage.
- To select similarity function from simple, Euclidean, Manhattan and Nfrag version of simple, Euclidean and Manhattan.
- To choose from none simulated annealing or simulated annealing with radnom starting position or simulated annealing with constant starting position.

- To set starting simulated annealing temperature.
- To set cooling rate.
- To set the distance of neighbours in simulated annealing.
- To choose how many dimensions to change in one step of simulated annealing.

Other features regarding mainly the monitoring of computation and updating cluster are:

- To get information about running computations and state of cluster. Also the ability to stop running and planed computations.
- To update code, data and preprocessed data on cluster.
- To set the number of computations per computer.
- To set the seed for random number generation.

Users are also provided with configuration file in which they can configure:

- Location of code and data on server.
- Number of splits.
- Seed used in randomizer.
- Names and adresses of cluster computer.

4.2 Optimization

Simulated annealing is quite computationally demanding since it requires to calculate the whole virtual screening in each iteration. Since we need at least several thousand iterations, even a few seconds to calculate a screening is too much. For example 10000 iterations (which is quite low number) and 3 seconds per screening would take about 8 hours to compute. On top of that we need to calculate weights for each split (and in our datasets we have 168 splits as explained in section 5.1). That is of course too much for any practical experiments.

At the beggining, when we used Python to implement screening and did not use any custom data format, one single screening took about 20-40 minutes and the whole computation was serial. At the end one screening takes from 0.1 to 0.5 seconds and screenings of splits are computed in paralel on any number of cluster computers. The times vary depending on types of fragments, descriptors and correlation treshold used. Without this improvement we could never execute our experiments.

Other than screenig the data preprocessing (creating fragments, calculating descriptors, removing correlated descriptors etc.) needs to be optimized too. One such preprocessing can take about 15 minutes. Thanks to reusing some preprocessed data we managed to improve this process too.

We have already discussed means of speeding up the computation on the data level - removing fragment noise and correlated descriptors. However great performance improvements were also made through technical solutions. In this section we discuss the most profound ones.

4.2.1 Caching

Caching of fragment comparisons made probably the greatest improvement in performance. This solution seems obvious however it is not so much from the view of a programmer who is not a chemist and does not know much about molecules and its fragments. The idea goes as follows: When comparing molecules we compare its fragments. These fragments are not unique for each molecule - some fragments occur in many molecules. So when we compare a pair of fragments we save the result somewhere and when we need to compare this same pair of fragments again (when comparing some other pair of molecules) we can just use the already computed value. There are two crucial factors which need to be fulfilled in order for caching to work.

First, there can not be too many different types of fragments or else we would run out of memory - we need to hold an array (or equivalent data type) with size: number of different fragments in all molecules squared. Second, we need to compare some fragments pairs often enough in order for caching to take effect - if every fragment pair was computed only once, caching would even hurt our method.

Both of those requirements are satisfied (which was a little bit surprising). There are typicaly only thousands or tens of thousands different fragment structures. And the same fragment pairs are compared often enough to speed up computation several fold.

4.2.2 C++ vs Python

At first the core of screening and simulated annealing was written in Python. When rewritten into C++ the same computation was suddenly computed much faster. One reason for the speed up is that along with the C++ code we implemented custom data format. With Python code we tried to use fragments in JSON format and descriptors in text format (direct output of RDKit), both represented in memory as dictionary. These two formats proved to be terribly slow thanks to the need to constantly search through the dictionaries. With C++ our data format is focused on pointers and is explained in A.2.

The speed up is also probably due to the fact that C++ can perform some optimization during compilation. We also use tightly packed arrays of pointer (in hopes of being cache friendly) in C++ while in Python we used dictionaries, which are much slower. Python is also dynamicaly typed which can generate many overhead machine instructions.

4.2.3 Better sorting algorithm

Another rather simple idea like caching. When comparing molecules using the Nfrag similarity function one need to sort the results of fragment pairs comparisments and pick the top N. However using the standard C++ sorting algorithm is an "over kill". We do not need to have the entire set of results sorted - we just need to know the top N. So by implementing this top N sort using linked list we achieved considerable speed up.

4.2.4 Memory optimization

In theory JSON output of fragmented molecules and output of RDKit or PaDEL is everything needed to perform screening and annealing. Data in this format could be as big as several hundred Mbits. This is unacceptable since we want to run multiple screenings on one computer. We also need to create table for caching which can be quite big when larger fragments are used like ecfp.2 (since more unique fragments are then present). So another priority was to reduce the size of data which we need to hold in memory. This was again done through our custom data format explained in A.2

4.2.5 Reusing the preprocessed data

Data input is handled in the preprocessing phase, where we convert the raw molecules in SDF format into our molecular representation, which we later use for computations. For each configuration the preprocessing needs to be done only once. First thing to understand when talking about creating our molecular representation is input data format. Input to our program is in form of .sdf files, which is data format used to describe molecules. SDF (structure-data file) is a chemical-data file format which holds structural information about molecule and wraps other data format - MDL Molfile which holds information about the atoms, bonds, connectivity and coordinates of a molecule. This data format was created by MDL Information Systems.

Files can be provided to WeFrag either through a *directory system* or as a *command line parameter*.

The advantage of using input through directory system is that all of the files created (either final or temporary) can be re-used when preprocessing certain other configurations - for example if the type of fragments stays the same, we do not separate fragments again. Or if we want to compute with different correlation treshold, we need to only recompute final preprocessing part. The whole preprocessing phase is explained in section B.2

This is possible thanks to the additional information the directory system provides. When using command line input we can not perform such optimizations since we do not have enough information about the data.

Thus directory system saves u a lot of time when testing great number of slightly different configurations since the preprocessing phase can be fairly time consuming.

4.2.6 Parallelization

We used two levels of parallelization. One level is implemented using Python threads. Threads were used only for enabling the client-server communication and for assignment of computations on a server or local computer.

The other level of parallelization is realized by paralelization of preprocessing and screening of splits. Parallelization of preprocessing is only partial - for example 2 splits of the same target can not be preprocessed at the same time, since creation of temporary files would collide.

All screening tasks can be computed in paralel, however some preprocessing tasks cannot be performed in paralel and they always need to be computed before screening tasks. This is why cluster is so useful - with cca. 38 computers in our laboratory we can in theory preprocess 38 different configurations in paralel - and mind that some configurations partialy use data from other configurations so this speeds up the preprocessing even more when more configurations are computed. After all preprocessing is finished we can in theory compute 304 splits in paralel (8 tasks per computer is most efficient). However when using some larger fragment types (such as ecfp.2) only about 1 or 2 splits per computer is possible thanks to massive memory demand.

5. Experiments

In this chapter, we take a look at our experiments. First we need to understand the data which we use for the experiments, after that we can discuss what we want to find out through the experiments. In the end we examine the results.

The naming for Euclidean similarity, Manhattan similarity and simple similarity is euclidean, manhattan and simple.

We use simple100, euclidean100 and mahnattan100 as names for Nfrag versions of simple, Euclidean and Manhattan functions with $P = 100$ (as explained in section 2.5.4).

Name ecfp.1 stands for circular fragments of radius 1 and tt.2, tt.3 and tt.4 stand for linear fragments of length 2, 3 and 4.

5.1 Data used

First datasets we considered is used in paper by Hoksza and Škoda [21]. However in those datasets we only had about 5 to 15 active molecules in train sets. Other datasets we looked into are from another paper by Hoksza and Škoda [2], however in those datasets there is not enough inactive molecules in train data.

As such we decided to create new 2 new dataset which has both more active and inactive molecules in train set. These datasets were proposed to be included in [21] so that they can be tested by other virtual screening methods.

Our data consist of 2 datasets each having the same 42 targets. Each target has 4 splits. Each split is divided into train and test data.

Dataset_20_1000_50_8000

- The **train data** consist of 20 active molecules and 1000 inactive molecules.
- The **test data** consist of 50 active molecules and 8000 inactive molecules.

Dataset_50_1000_50_8000

- The **train data** consist of 50 active molecules and 1000 inactive molecules.
- The **test data** consist of 50 active molecules and 8000 inactive molecules.

Training and testing is implemented as shown in the section 3.4 and the metric used to measure performance is AUC.

The AUC value of a target is calculated as average of AUC values of its splits. The AUC value for whole dataset is calculated as average of AUC values of all targets.

5.2 Focus of experiments

What exactly do we want to find out through experiments? We want to figure out if we are able to improve fragment-feature molecular representation when using various similarity functions with weights found through simulated annealing. We also want to examine how much do the parameters, which we can configure,

influence the performance. In the end, we want to compare our approach to traditional methods.

The parameters include:

1. Type of
   fragments chosen from *ecfp.1*, *tt.2*, *tt.3* and *tt.4*.
   Descriptors chosen from RDKit and PaDEL
   Similarity function chosen from simple, Euclidean, Manhattan and their Nfrag versions
2. Correlated descriptors removal treshold
3. Fragment noise reduction treshold
4. Simulated annealing factors:
   Starting temperature
   Cooling
   Distance of vectors in a iteration of annealing
   Number of dimensions which we change in the vector of weights in each iteration

Originally we wanted to calculate also with ecfp.2 fragments, however they proved to be too slow for the scope of experiments we needed.

Experiments were not performed in grid-like fashion, meaning we did not compute every combination of options. Instead after establishing how much do fragment types, descriptor types and similarity functions influence the method we chose the most promising combination. Using this combination we then explored the correlation treshold and examined the noise reduction treshold. Then we focused on simulated annealing factors. We also experimented with larger training set. Finaly we can compare our method against traditional LBVS methods.

Experiments in this chapter were computed using seed 1984. We also computed the same experiments with seed 5000 and 987654321. The differences were minimal (AUC+- 0.01).

5.3 Results of experiments

The tables follow color scheme for AUC values as shown in table 5.1. If a cell contains 2 values, the value on the right determines the color.

Table 5.1: The color scheme

0.5	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	1

The values in tables are average AUC values of all targets from all datasets using the average of all 4 splits of a target.

Initial value of weights before annealing is $\frac{1}{numberofdescriptors*2}$. Meaning all weights have the same value and thus have no effect.

5.3.1 Fragments, descriptors, similarity functions

In this section we focus on the effects of different fragment types, descriptors and similarity functions. Those are the parameters which influence our approach

the most. For the purpose of this experiment we disabled correlated descriptors removal and fragment noise elimination (so that they do not influence the outcome).

The setup of experiments:
Dataset: Dataset_20_1000_50_8000
Fragments: ecfp.1, tt.2, tt.3, tt.4
Descriptors: RDKit, PaDEL
Similarity function: simple100, euclidean100, mahnattan100, euclidean, manhattan and simple
Fragment noise treshold: 100 (essentialy is disabled)
Correlated descriptors treshold: 1 (essentialy is disabled)
Temperature: 1000
Cooling: 0.999
Distance of vectors: 25 (*neighbour_distance* value from 3.1)
Number of dimensions to change: 1 (*dim* value from 3.1)

We chose this temperature and cooling as it is the maximum which we could compute at this scale of experiment.

With this distance of vectors, weights can have 4 values. We limited weights to only 4 values, because in this set of experiments we commonly use cca. 126 RDKit descriptors (and even more PaDEL descriptors), which means we can have 4^{126} = combinations. In these conditions simulated annealing probably will not find the global maxima, however at least we are no going to get stuck in some small local maxima.

Train Data

First let us examine the results on training data:

Table 5.2: Train without weights / Train with weights - Fragment types, descriptors and similarity functions

RDKit	ecfp.1	tt.2	tt.3	tt.4
RDKit_euclidean100	0,84 / 0,94	0,75 / 0,83	0,81 / 0,89	0,83 / 0,92
RDKit_euclidean	0,64 / 0,98	0,69 / 0,95	0,65 / 0,97	0,68 / 0,97
RDKit_manhattan100	0,84 / 0,95	0,75 / 0,82	0,8 / 0,9	0,83 / 0,92
RDKit_manhattan	0,65 / 0,98	0,68 / 0,96	0,66 / 0,97	0,69 / 0,97
RDKit_simple100	0,63 / 0,97	0,56 / 0,94	0,64 / 0,96	0,65 / 0,96
RDKit_simple	0,72 / 0,98	0,69 / 0,96	0,72 / 0,97	0,75 / 0,97
PaDEL	ecfp.1	tt.2	tt.3	tt.4
PaDEL_euclidean100	0,84 / 0,86	0,73 / 0,74	0,8 / 0,81	0,81 / 0,83
PaDEL_euclidean	0,62 / 0,75	0,65 / 0,8	0,61 / 0,75	0,66 / 0,76
PaDEL_manhattan100	0,83 / 0,86	0,72 / 0,74	0,79 / 0,81	0,81 / 0,82
PaDEL_manhattan	0,61 / 0,73	0,63 / 0,78	0,61 / 0,73	0,66 / 0,75
PaDEL_simple100	0,57 / 0,83	0,57 / 0,79	0,55 / 0,77	0,61 / 0,78
PaDEL_simple	0,77 / 0,85	0,7 / 0,78	0,72 / 0,79	0,75 / 0,8

From table 5.2 and table 5.3 we can clearly see that we are able to find weights which improve the results on train data for each fragment type and similarity

Table 5.3: Difference in AUC with and without weights using train data- Fragment types, descriptors and similarity functions

RDKit	ecfp.1	tt.2	tt.3	tt.4
RDKit_euclidean100	0,10	0,08	0,09	0,09
RDKit_euclidean	0,33	0,27	0,32	0,29
RDKit_manhattan100	0,11	0,07	0,10	0,10
RDKit_manhattan	0,32	0,28	0,32	0,28
RDKit_simple100	0,34	0,38	0,31	0,31
RDKit_simple	0,26	0,27	0,25	0,21
PaDEL	ecfp.1	tt.2	tt.3	tt.4
PaDEL_euclidean100	0,02	0,01	0,02	0,01
PaDEL_euclidean	0,13	0,15	0,14	0,10
PaDEL_manhattan100	0,02	0,02	0,02	0,01
PaDEL_manhattan	0,12	0,15	0,13	0,09
PaDEL_simple100	0,26	0,22	0,22	0,17
PaDEL_simple	0,08	0,08	0,07	0,05

function when using both RDKit and PaDEl. **Thus simulated annealing is a succesful strategy for the search of weights on train data.** However for some similarity functions and fragment types we are able to find better weights than for others and PaDEL performs much worse than RDKit.

Influence of fragment types:

We can notice that type of fragments influence training a lot. The ecfp.1 fragments performed better for all similarity functions and both descriptors type than tt.2, tt.3 and tt.4 fragments. Also tt.4 performed better than tt.2 and tt.3 and came close to ecfp.1 fragments. The reason for this is probably because of the size and shape of fragments. The ecfp.1 fragments are much bigger than tt.2 and tt.3 while tt.4 are closer in size. The size of fragments is important because it influences the number of fragments created from a molecule. For example since tt.2 and tt.3 are essentialy paths of length two and three in a graph (molecule) the number of paths of length three is greater than the number of paths of length two (for most graphs - molecules). Also the bigger the fragments are, the greater is the variety of structuraly different fragments created. So by using bigger fragments we get greater number of unique fragments. However the ecfp fragments are also shaped differently than tt fragments which can influence the performance.

Influence of descriptor types:

Using RDKit we get a greater increase in AUC than when using PaDEL. This is probably due to the fact, that PaDEL has much more descriptors (when using correlation treshold equal to 1), which makes it harder to find suitable weights. Without weights RDKit also performs better. It is a little bit surprising that RDKit performs better than PaDEL without weights, since the number of descriptors which PaDEL calculates is approximately four times larger than what RDKit calculates. The reason for this could be that great number of of PaDEL descriptors are correlated, close to constant for all molecules or just not important.

Influence of similarity functions:

It is much harder to train using the euclidean100 and manhattan100 function. However euclidean and manhattan perform worse without weights. It seems like the simple100 can be trained better than the simple similarity function. After we examine the test data we estimate why do functions behave like this.

Test Data

Now lets take a look at the results of test data and piece all of the information together.

Table 5.4: Test without weights / Test with weights - Fragment types, descriptors and similarity functions

RDKit	ecfp.1	tt.2	tt.3	tt.4
RDKit_euclidean100	0,9 / 0,91	0,79 / 0,79	0,86 / 0,87	0,89 / 0,89
RDKit_euclidean	0,62 / 0,84	0,66 / 0,8	0,63 / 0,82	0,65 / 0,82
RDKit_manhattan100	0,89 / 0,9	0,78 / 0,79	0,85 / 0,87	0,88 / 0,89
RDKit_manhattan	0,64 / 0,85	0,67 / 0,81	0,64 / 0,82	0,67 / 0,83
RDKit_simple100	0,63 / 0,82	0,57 / 0,8	0,64 / 0,81	0,65 / 0,81
RDKit_simple	0,72 / 0,86	0,68 / 0,81	0,71 / 0,82	0,74 / 0,83
PaDEL	ecfp.1	tt.2	tt.3	tt.4
PaDEL_euclidean100	0,9 / 0,9	0,71 / 0,72	0,84 / 0,85	0,85 / 0,85
PaDEL_euclidean	0,61 / 0,69	0,64 / 0,74	0,6 / 0,69	0,65 / 0,71
PaDEL_manhattan100	0,89 / 0,9	0,71 / 0,71	0,84 / 0,84	0,85 / 0,85
PaDEL_manhattan	0,59 / 0,68	0,62 / 0,71	0,6 / 0,69	0,65 / 0,71
PaDEL_simple100	0,61 / 0,72	0,61 / 0,69	0,56 / 0,69	0,6 / 0,68
PaDEL_simple	0,77 / 0,81	0,69 / 0,73	0,72 / 0,76	0,75 / 0,77

Table 5.5: Difference in AUC with and without weights using test data - Fragment types, descriptors and similarity functions

RDKit	ecfp.1	tt.2	tt.3	tt.4
RDKit_euclidean100	0,01	0,01	0,01	0,00
RDKit_euclidean	0,22	0,14	0,19	0,18
RDKit_manhattan100	0,01	0,01	0,01	0,01
RDKit_manhattan	0,21	0,14	0,18	0,16
RDKit_simple100	0,19	0,23	0,18	0,17
RDKit_simple	0,13	0,13	0,11	0,09
PaDEL	ecfp.1	tt.2	tt.3	tt.4
PaDEL_euclidean100	0,00	0,00	0,00	0,00
PaDEL_euclidean	0,09	0,10	0,09	0,06
PaDEL_manhattan100	0,00	0,00	0,00	0,00
PaDEL_manhattan	0,09	0,10	0,09	0,05
PaDEL_simple100	0,12	0,08	0,13	0,09
PaDEL_simple	0,04	0,04	0,04	0,03

In tables 5.4 and 5.5 we see that weights which we found on the train data do improve the performance on test data. However it largely depends on the similarity function used - euclidean100 and manhattan100 seem to be improved only

a little bit. However other methods are improved greatly. **As such we can say that we are able to improve perofrmance of our method using weights found through simulated annealing on train data.** Interestingly the simple similarity works reasonably well too - apparently the information optained from bins is enough.

Influence of fragment types:

Different fragment types behave similarly like in train data. Ecfp.1 again performs better than tt.2, tt.3 and tt.4. This is still probably caused by the number and size of fragments.

Influence of descriptor types:

Again RDKit performs better than PaDEL both with and without weights. We think that the reason is the same as with the train data - since PaDEL has more descriptors, it is harder to train and many of them are probably correlated.

Influence of similarity functions:

This is the most interesting part. It seems like we are not able to improve euclidean100 and manhattan100 using weights on test data even with different fragment and descriptor types. However even though we are not able to improve those methods with weights, they still perform much better than other similarity functions.

So why is every method other than euclidean100 and manhattan100 improved by weights? First of all euclidean100 and manhattan100 are very similar methods, since Manhattan distance can be though of as just a approximation of Euclidean distance. Second we hypothesise that the Nfrag method puts much greater priority on the type of fragments which the pair of molecules containt than the standard method.

Standard Euclidean or Manhattan similarity computes similarity of pair of molecules as average of similarities of all their fragments. This way we essentialy blurr all information about what kind of fragments those molecules contained (thanks to the average over all fragments) - two molecules with mediocre similarity in all fragments are similiar just as two molecules with half of their fragments identical and half different completly. This way weights can influence the outcome more easily as descriptors are essentialy the only information which the similarity function can utilise.

Since Nfrag method computes similarity as an average of top N similar fragments, it definitely uses some information about what kind of fragments these molecules contain. Two molecules with mediocre similarity in all fragments are less similar than two molecules with half of their fragments identical and half different completly. It seems like the information which the method obtains this way is so strong that the weights can not overcome it easily. Other possibilty is that it is easy to overtrain the weights when using this similarity function.

In order to test this, we tried setting the correlation treshold to 0 using the RDKit descriptors (table 5.6). This way we end up with only 1 descriptor - the molecular weight (MolWt).

As we can see euclidean100 still works much better than euclidean. This supports the hypothesis that Nfrag similarity methods use a lot of iformation about what kind of fragments the molecules contain, since molecular weight can be almost used as an unique identifier of a fragment.

Table 5.6: RDKit descriptors, ecfp.1 fragments, euclidean100 similarity function when used with correlation 0 - which means only molecular weight is used as a descriptor

Corr	0
RDKit_euclidean100_ecfp.1	0,77
RDKit_euclidean_ecfp.1	0,65

We also tried generating random weights for seeds 1, 2, 3, 4 and 5 (table 5.7).

Table 5.7: RDKit descriptors, ecfp.1 fragments, euclidean100 similarity function with random weights

Seed	1	2	3	4	5
RDKit_euclidean100_ecfp.1	0.903	0.900	0.899	0.902	0.901
RDKit_euclidean_ecfp.1	0.611	0.596	0.601	0.620	0.607

The size of interval in which euclidean100 values lie is 0.004 and for euclidean the size is 0.024. Thus even random weights influence euclidean more than euclidean100. This is probably due to the fact, that there are multiple descriptors which can act as an identifier of a fragment (which euclidean100 can utilise) and the chance that some set of weights would minimize the importance of all those descriptors is very small.

However it is a little bit surprising that random weights do not have a greater influence. **This shows, that weights which we found using simulated annealing and which improve various similarity functions are not just a lucky find.**

Conclusion

- Some fragments are better than others for the implementatin of our approach - ecfp.1 seems to be the best performing.
- Greater number of descriptors makes it harder for simulated annealing to find weights. Greater number of descriptors also does not improve the method when used without wieghts. RDKit seems to be better than PaDEL.
- Some similarity function are better suited for our approach than others. Some similarity functions are improved by weights like euclidean, but some are not like euclidean100.
- Random weights do not influence the performance too much. Thus the weights which improve the preformance are with high probabilty not doing it by luck.
- We picked ecfp.1 as fragments used in future experiments. Similarity functions chosen are euclidean and euclidean100, since they performed the best. Also euclidean represents a function influenced by weights and euclidean100 represents function not influenced by weights. We want to examine if other parameters change this behaviour.

5.3.2 Correlated descriptors

Now we want examine how much does the correlation treshold influence the method, especially how does RDKit and PaDEL perform with various correlation tresholds.

The setup of experiments:
Dataset: Dataset_20_1000_50_8000
Fragments: ecfp.1
Descriptors: RDKit, PaDEL
Similarity function: euclidean100, euclidean
Fragment noise treshold: 100 (essentialy is disabled)
Correlated descriptors treshold: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Temperature: 1000
Cooling: 0.999
Distance of vectors: 25
Number of dimensions to change: 1

We use ecfp.1 fragments since they the had best results in previous experiments. Both RDKit and PaDEL are used since we want to see how the correlation treshold influences each descriptor type. We chose euclidean and euclidean100 as euclidean showed strong improvement with weights and euclidean100 did not react to them and it would be interresting to see if correlated descriptors change this behaviour.

Train data table is not included in these experiments since we already established that the training phase succesfully finds the weights.

The table shows correlation treshold =>average number of descriptors which are assigned to a fragment in first and forth row. The number of descriptors of a fragment is different for each split, since for each split we calculate correlated descriptors separately.

Table 5.8: Test without weights / Test with weights - Correlation treshold =>Number of descriptors

RDKit	0,1=>40,20	0,2=>46,90	0,3=>57,60	0,4=>65,50	0,5=>72,20	0,6=>83,40	0,7=>90,50	0,8=>100,50	0,9=>111,80	1=>125,80
RDKit_euclidean100	0,88 / 0,88	0,89 / 0,89	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9	0,9 / 0,91	0,9 / 0,9	0,9 / 0,91
RDKit_euclidean	0,57 / 0,81	0,6 / 0,82	0,64 / 0,84	0,64 / 0,83	0,64 / 0,84	0,65 / 0,84	0,65 / 0,84	0,64 / 0,84	0,62 / 0,83	0,62 / 0,84
PaDEL	0,1=>49,30	0,2=>62,30	0,3=>82,30	0,4=>104,90	0,5=>135,10	0,6=>169,90	0,7=>214,80	0,8=>283,20	0,9=>382,90	1=>739,10
PaDEL_euclidean100	0,89 / 0,9	0,89 / 0,89	0,9 / 0,9	0,9 / 0,91	0,9 / 0,91	0,9 / 0,91	0,9 / 0,91	0,9 / 0,91	0,9 / 0,91	0,9 / 0,9
PaDEL_euclidean	0,61 / 0,82	0,59 / 0,8	0,64 / 0,83	0,66 / 0,83	0,67 / 0,83	0,63 / 0,83	0,63 / 0,83	0,62 / 0,82	0,63 / 0,8	0,61 / 0,69

The results from table 5.8 are pretty surprising. It seems like the correlation treshold does not really affect the performance of RDKit descriptors at all - results on test data and train data are consistent up to the 0.2 correlation treshold. Even after that they are worse only by a little bit.

The only big difference is between PaDEL paired with euclidean method from correlation 1 to correlation 0.9. We removed a lot of descriptors and thus the simulated annealing can find suitable weights more easily and improve the performance. We can also notice below the correlation treshold of 0.8 the PaDEL descriptors perform very similarly to RDKit descriptors.

Lastly we can see that a lot of descriptors are correlated, since the average number of used descriptors drops very fast (especialy for PaDEL).

Conclusion

A lot of descriptors are correlated or non-important. When using PaDEL it is benefitial for our method to remove some correlated descriptors. We can also safely remove a lot of descriptors and speed our calculations up without having to wory about losing performance.

We can also safely use RDKit instead of PaDEL since PaDEL does not bring any benefits to our method even though it has much more descriptors.

5.3.3 Fragment noise reduction treshold

In this set of experiments we examine how does fragment noise influence the performance. We use only RDKit descriptors since we established that PaDEL does not bring any advantage in previous experiments.

The setup of experiments:
Dataset: Dataset_20_1000_50_8000
Fragments: ecfp.1
Descriptors: RDKit
Similarity function: euclidean100, euclidean
Fragment noise treshold: 1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Correlated descriptors treshold: 0.5
Temperature: 1000
Cooling: 0.999
Distance of vectors: 25
Number of dimensions to change: 1

Correlation treshold is set to 0.5 since it does not seem to harm perfromance when using RDKit descriptors.

Train data table is not included in these experiments since we already established that the training phase succesfuly find the weights.

The table shows fragment noise treshold =>number of unique fragments in a split averaged over all splits and targets (not by average number of fragments in a single molecule averaged over all splits and targets!) in the first line.

Table 5.9: Test without weights / Test with weights - Fragment noise treshold =>Number unique fragments in all molecules

RDKit	1=>645,5	5=>754,9	10=>786,5	20=>809,4	30=>819	40=>822,8	50=>826	60=>827,9	70=>828,4	80=>828,6	90=>828,9	100=>829,8
euclidean100	0,75 / 0,75	0,82 / 0,83	0,85 / 0,85	0,89 / 0,89	0,9 / 0,89	0,91 / 0,91	0,91 / 0,91	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9
euclidean	0,72 / 0,72	0,7 / 0,75	0,68 / 0,77	0,66 / 0,82	0,66 / 0,82	0,67 / 0,83	0,67 / 0,83	0,66 / 0,84	0,66 / 0,83	0,66 / 0,83	0,66 / 0,83	0,62 / 0,84

From the table 5.9 it is clearly visible that fragment noise reduction worsen the performance both for euclidean and euclidean100 when using weights. Without weights the euclidean method improves.

This improvement in euclidean method without weights could really be thanks to the reduction of harmful fragment noise, however it seems like the same fragments also contribute to the performance of euclidean with weights.

It is interesting that even at 1% treshold the number of fragments is still quite large. This means that a lot of molecules have fairly unique fragments.

Conclusion

The fragment noise reduction is not a siutable way of increasing precision of our approach as we loose too much useful information. Nor is it suitable way of improving performance, since we still end up with large quantities of fragments on average.

5.3.4 Temperature, cooling, distance and number of dimension to change

In this set of experiments we examine the parameters of simulated annealing from section 3.1.

The setup of experiments:
Dataset: Dataset_20_1000_50_8000
Fragments: ecfp.1
Descriptors: RDKit
Similarity function: euclidean100, euclidean
Fragment noise treshold: 60
Correlated descriptors treshold: 0.5
Temperature: 10 100 1000 10000 100000 1000000
Cooling: 0.99 0.999
Distance of vectors: 25
Number of dimensions to change: 1

Correlation treshold is set to 0.5 and fragment noise reduction treshold is set to 60 since it does not seem to harm performance when using RDKit descriptors.

Train data table is not included in these experiments since we already established that the training phase succesfuly find the weights.

Table 5.10: Test without weights / Test with weights - Temperature and cooling

Temperature	10	100	1000	10000	100000	1000000
euclidean100_0,99	0,9 / 0,91	0,9 / 0,91	0,9 / 0,91	0,9 / 0,91	0,9 / 0,91	0,9 / 0,91
euclidean100_0,999	0,9 / 0,91	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9
euclidean_0,99	0,66 / 0,79	0,66 / 0,81	0,66 / 0,81	0,66 / 0,81	0,66 / 0,81	0,66 / 0,81
euclidean_0,999	0,66 / 0,83	0,66 / 0,83	0,66 / 0,84	0,66 / 0,83	0,66 / 0,83	0,66 / 0,83

In table 5.10 different temperatures do not influence the euclidean100 similarity function. However different cooling seems to improve euclidean100 slightly. This could mean that weights are overtrained for euclidean100, since at lower temperatures and faster coolings overtraining should be reduced. However the change is too small to tell for sure. It seems like euclidean100 is just very resiliant to the influence of weights.

However euclidean method is influenced singificanlty by the speed of cooling. At 0.999, it performs much better. This means that euclidean similarity function performs better when given greater time to find weights.

We also tried different distances of vectors. The setup is the same as above, with the difference in:

Temperature: 1000000

Distance of vectors: 5, 10, 25, 50, 100

Since we are using correlation 0.5, that means we have 72 RDKit descriptors on average. With different values of distances, we are essentialy giving more positions for each weights to obtain. For example at 100 distance the weights can be either 1 or 0, thus we have 2^{72} combinations of weights on average. With distance value of 10, we have 10^{72} combinations of weights. Thus smaller distances should result in greater precision of weights but also make it harder to find the weights since there the number of possible weights gets larger.

Table 5.11: Test without weights / Test with weights - Distance of weights

Distance	5	10	25	50	100
euclidean100	0,9 / 0,91	0,9 / 0,91	0,9 / 0,9	0,9 / 0,9	0,9 / 0,9
euclidean	0,66 / 0,82	0,66 / 0,83	0,66 / 0,83	0,66 / 0,83	0,66 / 0,83

Table 5.11 shows interesting result. It seems like there is almost no difference in the effect of weights on test data between weights which can have only 2 positions and weights with 20 posible positions. First this could mean that more precise weights do not translate better from train data to test data. Second this could mean that when searching through so many possible combinations of weights, we are not able to utilise the precision of smaller distances and find just rough weights anyway.

To get a better idea, we created charts of training on a single split (target DRD2_Antagonist, split s_001) 5.1 5.2 for euclidean100 similarity function. The X axis shows the number of iterations and the Y axis shows the value of AUC. We can see that when training with distance 100 the graph gets much more jittery, however the annealing still works relatively well and can find some kind of maxima. This shows that the difference between training a split with distance 10 and 100 is significant, however when averaged over all splits and targets the result is similar.

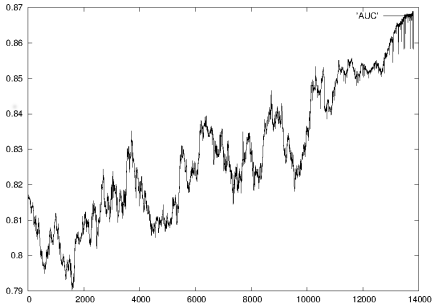


Figure 5.1: euclidean100, distance 10, simulated annealing for 1 split

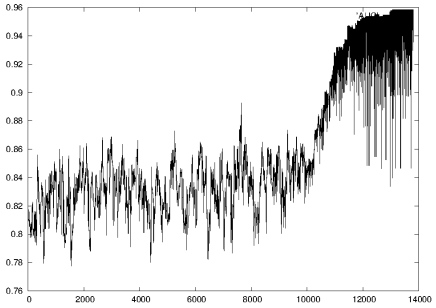


Figure 5.2: euclidean100, distance 100, simulated annealing for 1 split

Lastly we experimented with the number of dimensions changed in each iteration of annealing.

The setup is the same as above with the difference in:

Temperature: 100000

Distance of vectors: 10

Number of dimensions to change: 1, 2, 3, 5, 10, 15, 25, 50, all dimensions

Greater number of dimensions of vector of weights which we change in each iteration should make it harder for simulated annealing. The reason for this is that we essentialy jump from position to other position in the vector space acros multiple dimensions and loose the information which dimension improves the method and which does not.

With the correlation set to 0.5 we have on average cca. 70 descriptors for a fragment.

Table 5.12: Test without weights / Test with weights - Dimension

Dimensions	1	2	3	5	10	15	25	50	all
euclidean100	0,91 / 0,91	0,91 / 0,91	0,91 / 0,9	0,91 / 0,91	0,91 / 0,9	0,91 / 0,91	0,91 / 0,91	0,91 / 0,91	0,91 / 0,9
euclidean	0,66 / 0,83	0,66 / 0,84	0,66 / 0,84	0,66 / 0,84	0,66 / 0,84	0,66 / 0,84	0,66 / 0,83	0,66 / 0,83	0,66 / 0,83

From the table 5.12 we can see that again the effect of various numbers of dimensions which we change in each iteration is little.

In order to get a better picture we again created charts of training on a single split (target DRD2_Antagonist, split s_001) 5.3 5.3 for euclidean similarity function.

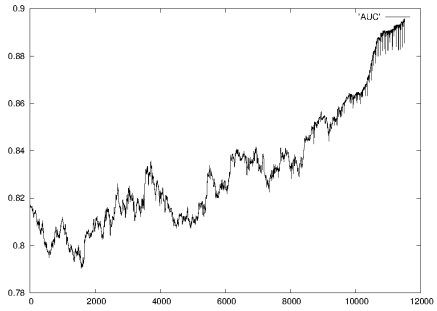


Figure 5.3: euclidean, dimension 1, simulated annealing for 1 split

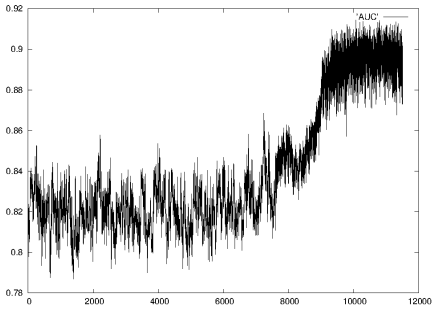


Figure 5.4: euclidean, all dimensions, simulated annealing for 1 split

Interrestingly these charts look very similar to the previous ones, where we manipulated with distances of vectors. In the end both distance and dimension variable influence how much are weights in each iteration different.

Conclusion

The speed of cooling influences performance singnificantly, while the starting temperature and distane of weights do not.

The effects of various numbers of dimensions to change and distances seems similar. For both variables it is probably better to use lower values.

We hypothesise that using even slower cooling and greater starting temperature we could see a greater difference in various distances of weights and numbers of dimensions to change.

To improve the performance of simulated annealing, the best strategy is to lower the cooling and increase the temperature, other variables set as low as possible.

5.3.5 Effects of larger train data

It seems like a lot of parameters do not influence the performance. The greatest differences are between different fragment types, similarity functions and coolings. Different descriptors, correlation tresholds, fragment noise tresholds, temperatures, distances and number of dimensions to change do not seem to really cause a change.

Now we want to see if larger train data can improve the effects of weights using ecfp.1 and similarity functions euclidean and eucldiean100.

The setup of experiments:
Dataset: Dataset_20_1000_50_8000, Dataset_50_1000_50_8000
Fragments: ecfp.1
Descriptors: RDKit
Similarity function: euclidean100, euclidean
Fragment noise treshold: 60
Correlated descriptors treshold: 0.5
Temperature: 1000
Cooling: 0.999
Distance of vectors: 25
Number of dimensions to change: 1

Dataset_20_1000_50_8000 is marked as $D20$.
Dataset_50_1000_50_8000 is marked as $D50$.

Greater train set will cause better overall AUC, however we are interested in the increase of perfromance when using weights.

Table 5.13: Test without weights / Test with weights - Different datasets

Dataset	D1	D2
euclidean100	0,9 / 0,9	0,94 / 0,94
euclidean	0,66 / 0,84	0,66 / 0,87

Table 5.14: Difference in AUC values with and without weights in test data - Different datasets

Datasets	D1	D2
euclidean100	0,000	0,002
euclidean	0,173	0,212

In table 5.13 and 5.14 we can notice that euclidean method sees better improvement when used on larger training set. However euclidean100 still is not improved by weights at all.

This shows again, that weights used as an improvement of our method depends greatly on the similarity function used. Some similarity functions do not respond well to trained weights. Table 5.15 and 5.16 show that we really did find weights on train data even for euclidean100 method (although worse weights than for euclidean method).

Table 5.15: Train without weights / Train with weights - Different datasets

	D1	D2
euclidean100	0,85 / 0,95	0,88 / 0,95
euclidean	0,67 / 0,97	0,68 / 0,96

Table 5.16: Difference in AUC values with and without weights in train data - Different datasets

Datasets	D1	D2
euclidean100	0,107	0,075
euclidean	0,302	0,279

Conclusion

Not even weights found on a dataset with larger train data influence the euclidean100 similarity function. However euclidean similarity function sees greater improvement. This shows us again, that some similarity functions respond better to weights than others.

Weights found through larger train set are better than weights found through smaller train set.

5.3.6 Comparison to traditional methods

After examining the effects of different parameters on the performance of our approach, we can now compare it to other traditional methods. For this comparison we picked the best performing combination.

The setup of computation:
Dataset: Dataset_20_1000_50_8000
Fragments: ecfp.1
Descriptors: RDKit
Similarity function: euclidean100
Fragment noise treshold: 50
Correlated descriptors treshold: 0.5
Temperature: 1000
Cooling: 0.999
Distance of vectors: 25
Number of dimensions to change: 1

The graph 5.5 contains AUC value on Y axis and targets of given dataset on X axis.

As we can see euclidean100 performs similarly as the traditional methods, however is outperformed by ECFP 2 2048. Full table of values for each method and target can be found in attachements 6.1.

Now lets take a look at the dataset Dataset_50_1000_50_8000 5.6. The setup remains the same, only the dataset used changed.

Again euclidean100 performs almost as well as the traditional methods, outperformed by ECFP 2 2048. Full table of values for each method and target can be found in attachements 6.2.

Figure 5.5: Our method using ecfp.1 fragments, euclidean100 similarity function and RDKit descriptors compared to traditional methods on Dataset_20_1000_50_8000

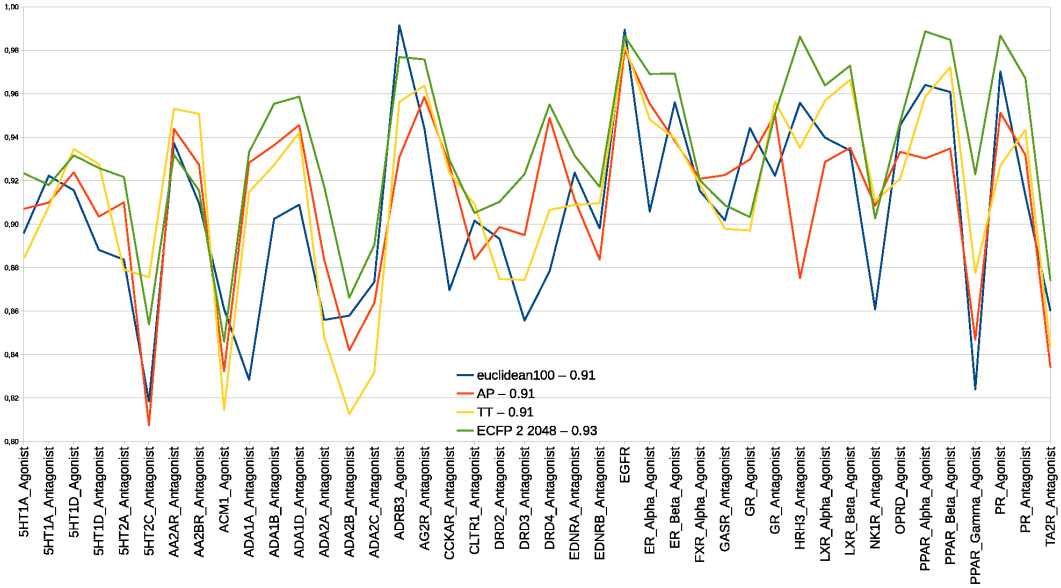
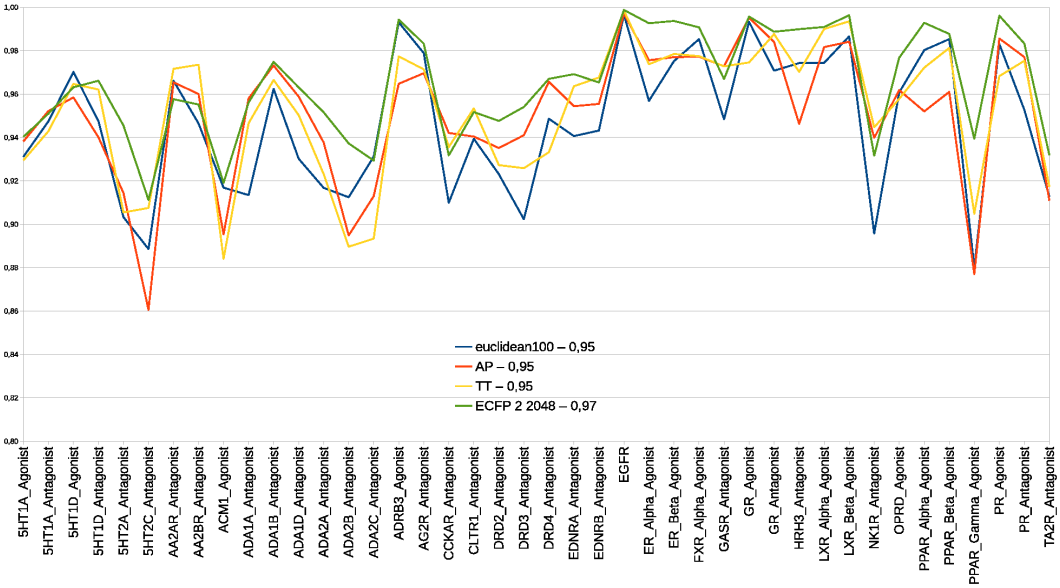


Figure 5.6: Our method using ecfp.1 fragments, euclidean100 similarity function and RDKit descriptors compared to traditional methods on Dataset_20_1000_50_8000



Conclusion

The euclidean100 performs in a comparable way to traditional methods on both datasets. This means that with the right similairy function the fragment-feature molecular representation is comparable to traditional methods. Unfortunately euclidean100 does not seem to be improved by weights - if euclidean100 improved in similar fashion as euclidean for example, we would outperform the traditional methods. We believe that weighted fragment-feature representation has the potential to outperform the traditional methods, one just need to design suitable similarty function.

6. Discussion

In our work we have implemented weighted fragment-feature molecular representation, designed multiple similarity functions for this representation, implemented simulated annealing and added the elimination of correlated descriptors and fragment noise. We also created a tool for large scale computation of experiments.

We have demonstrated that when given the right similarity function, weights found through simulated annealing improve the performance of weighted fragment-feature molecular representation. We have also found a similarity function which performs almost as well as other traditional methods. The influence of different parameters of our approach was examined through experiments and we determined which are benefitial and which are not. Thus we completed the two main goals of our work.

All of the subgoals of our work were also completed. We found that the best performing fragments are circular fragments of radius 1, best performing descriptors are from RDKit, euclidean100 is the best performing similarity function and we explored the correlation treshold and fragment noise.

As we were able to both succesfuly design well performing similarity function, demonstrate the potential of weights found through simulated annealing and complete all of the subgoals, we think of this work as a succes.

6.1 Future work

With the information obtained through this work, we see a potential in weighted fragment-feature representation paired with simulated annealing. As the similarity function seems to be the most influential factor, we think that a solution to outperform the traditional methods lies in finding a better one. Perhaps similarity function based on the Nfrag similarity, hovewer altered in such a way that weights improve it.

We also want to use our method on other datasets, such as datasets with very low and very high number of training molecules. Also it would be interesting to try extremely long training times on these datasets.

Bibliography

[1] Roberto Todeschini and Viviana Consonni. *Handbook of Molecular Descriptors*. Wiley-VCH, 2000.

[2] Škoda P. Hoksza D. Using bayesian modeling on molecular fragments features for virtual screening. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2016.

[3] C. Lipinski and A. Hopkins. Navigating chemical space for biology and medicine. *Nature*, page 855–861, 2004.

[4] McMartin C. Bohacek, R.S. and W.C. Guida. The art and practice of structure-based drug design: a molecular modeling perspective. *Medicinal Research Reviews*, page 3–50, 1996.

[5] Hugo Kubinyi Gerd Folkers Christoph Sotriffer, Raimund Mannhold. *Virtual Screening: Principles, Challenges, and Practical Guidelines*. Wiley-VCH, 2011.

[6] M.A. Johnson and G.M. Maggiora. *Concepts and Applications of Molecular Similarity*. John Wiley & Sons, Inc., New York, 1990.

[7] Cramer R.D. Ferguson A.M. Clark R.D. Patterson, D.E. and L.E. Weinberger. Neighborhood behavior: a useful concept for validation of molecular diversity descriptors. *Journal of Medicinal Chemistry*, page 3049–3059, 1996.

[8] F.L. Stahura and J. Bajorath. Virtual screening methods that complement hts. *Combinatorial Chemistry & High Throughput Screening*, 7:259–269, 2004.

[9] F.; Dominy B.W.; Feeney P.J. Lipinski, C.A.; Lombardo. Experimental and computational approaches to estimate solubility and permeability in drug discovery and development settings. *Adv. Drug Deliv. Rev.*, pages 3–25, 1997.

[10] Vassilatis DK Cournia Z Lionta E, Spyrou G. Structure-based virtual screening for drug discovery: principles, applications and recent advances. *Curr Top Med Chem*, pages 1923–38, 2014.

[11] V. Todeschini, R.; Consonni. *Molecular Descriptors for Chemoinformatics*. Wiley-VCH, 2009.

[12] Qiannan Hu Yi-Zeng Liang Dong-Sheng Cao, Qingsong Xu. Manual for chemopy. `https://www.researchgate.net`. Accessed: 2010-09-30.

[13] Yap CW. Padel-descriptor: An open source software to calculate molecular descriptors and fingerprints. *Journal of Computational Chemistry*, pages 1466–1474, 2011.

[14] Greg Landrum. Rdkit: Open-source cheminformatics. `http://www.rdkit.org`. Accessed: 2010-09-30.

[15] C.; Kuhn S.; Floris M.; Guha R.; Willighagen E. L. Steinbeck, C.; Hoppe. Recent developments of the chemistry development kit (cdk) - an open-source java library for chemo- and bioinformatics. *Current Pharmaceutical Design*, pages 2111 – 2120, 2006.

[16] D. E. J. Sykora, V. J.; Leahy. Chemical descriptors library (cdl): a generic, open source software library for chemical informatics. *Chem Inf Model*, page 1931, 2008.

[17] J. S. Dixon R. Nilakantan, N. Bauman and R. Venkataraghavan. Topological torsion: A new molecular descriptor for sar applications. comparison with 0ther descriptors. *J Chem Inf Comput Sci*, page 82–85, 1987.

[18] D. Rogers and M. Hahn. Extended-connectivity fingerprints. *J Chem Inf Comput Sci*, page 742–754, 2010.

[19] ChemAxon. Chemaxon docs. `https://docs.chemaxon.com`. Accessed: 2010-09-30.

[20] Jan K. Lenstra Emile Aarts. *Local Search in Combinatorial Optimization.* John Wiley & Sons, 1997.

[21] Škoda P. Hoksza D. Benchmarking platform for ligand-based virtual screening. *Bioinformatics and Biomedicine (BIBM)*, 2016.

List of Figures

List of Tables

List of Abbreviations

LBVS Ligand based virtual screening

euclidean100 Nfrag similarity function using Euclidean distance and P value of 100.

manhattan100 Nfrag similarity function using Manhattan distance and P value of 100.

simple100 Nfrag similarity function using simple distance and P value of 100.

euclidean Euclidean similarity.

manhattan Manhattan similarity.

simple Simple similarity.

ecfp.1 and ecfp.2 Circular fragments with radius of 1 and 2.

tt.2, tt.3, tt.4 Linear fragments of length 2, 3 and 4.

Attachments

Table 6.1: Complete table for comparions of euclidean100 to traditional methods for the Dataset_20_1000_50_8000

	euclidean100 – 0.91	AP – 0.91	TT – 0.91	ECFP 2 2048 – 0.93
5HT1A_Agonist	0,90	0,91	0,88	0,92
5HT1A_Antagonist	0,92	0,91	0,91	0,92
5HT1D_Agonist	0,92	0,92	0,93	0,93
5HT1D_Antagonist	0,89	0,90	0,93	0,93
5HT2A_Antagonist	0,88	0,91	0,88	0,92
5HT2C_Antagonist	0,82	0,81	0,88	0,85
AA2AR_Antagonist	0,94	0,94	0,95	0,93
AA2BR_Antagonist	0,91	0,93	0,95	0,92
ACM1_Agonist	0,86	0,83	0,81	0,85
ADA1A_Antagonist	0,83	0,93	0,91	0,93
ADA1B_Antagonist	0,90	0,94	0,93	0,96
ADA1D_Antagonist	0,91	0,95	0,94	0,96
ADA2A_Antagonist	0,86	0,88	0,85	0,92
ADA2B_Antagonist	0,86	0,84	0,81	0,87
ADA2C_Antagonist	0,87	0,86	0,83	0,89
ADRB3_Agonist	0,99	0,93	0,96	0,98
AG2R_Antagonist	0,94	0,96	0,96	0,98
CCKAR_Antagonist	0,87	0,93	0,92	0,93
CLTR1_Antagonist	0,90	0,88	0,91	0,91
DRD2_Antagonist	0,89	0,90	0,87	0,91
DRD3_Antagonist	0,86	0,89	0,87	0,92
DRD4_Antagonist	0,88	0,95	0,91	0,96
EDNRA_Antagonist	0,92	0,91	0,91	0,93
EDNRB_Antagonist	0,90	0,88	0,91	0,92
EGFR	0,99	0,98	0,98	0,99
ER_Alpha_Agonist	0,91	0,96	0,95	0,97
ER_Beta_Agonist	0,96	0,94	0,94	0,97
FXR_Alpha_Agonist	0,92	0,92	0,92	0,92
GASR_Antagonist	0,90	0,92	0,90	0,91
GR_Agonist	0,94	0,93	0,90	0,90
GR_Antagonist	0,92	0,95	0,96	0,95
HRH3_Antagonist	0,96	0,88	0,94	0,99
LXR_Alpha_Agonist	0,94	0,93	0,96	0,96
LXR_Beta_Agonist	0,93	0,94	0,97	0,97
NK1R_Antagonist	0,86	0,91	0,91	0,90
OPRD_Agonist	0,95	0,93	0,92	0,95
PPAR_Alpha_Agonist	0,96	0,93	0,96	0,99
PPAR_Beta_Agonist	0,96	0,93	0,97	0,98
PPAR_Gamma_Agonist	0,82	0,85	0,88	0,92
PR_Agonist	0,97	0,95	0,93	0,99
PR_Antagonist	0,91	0,93	0,94	0,97
TA2R_Antagonist	0,86	0,83	0,84	0,87

Table 6.2: Complete table for comparions of euclidean100 to traditional methods for the Dataset_50_1000_50_8000

	euclidean100 - 0,95	AP - 0,95	TT - 0,95	ECFP 2 2048 - 0,97
5HT1A_Agonist	0,93	0,94	0,93	0,94
5HT1A_Antagonist	0,95	0,95	0,94	0,95
5HT1D_Agonist	0,97	0,96	0,96	0,96
5HT1D_Antagonist	0,95	0,94	0,96	0,97
5HT2A_Antagonist	0,90	0,91	0,91	0,95
5HT2C_Antagonist	0,89	0,86	0,91	0,91
AA2AR_Antagonist	0,97	0,97	0,97	0,96
AA2BR_Antagonist	0,95	0,96	0,97	0,96
ACM1_Agonist	0,92	0,90	0,88	0,92
ADA1A_Antagonist	0,91	0,96	0,95	0,96
ADA1B_Antagonist	0,96	0,97	0,97	0,97
ADA1D_Antagonist	0,93	0,96	0,95	0,96
ADA2A_Antagonist	0,92	0,94	0,92	0,95
ADA2B_Antagonist	0,91	0,89	0,89	0,94
ADA2C_Antagonist	0,93	0,91	0,89	0,93
ADRB3_Agonist	0,99	0,96	0,98	0,99
AG2R_Antagonist	0,98	0,97	0,97	0,98
CCKAR_Antagonist	0,91	0,94	0,94	0,93
CLTR1_Antagonist	0,94	0,94	0,95	0,95
DRD2_Antagonist	0,92	0,94	0,93	0,95
DRD3_Antagonist	0,90	0,94	0,93	0,95
DRD4_Antagonist	0,95	0,97	0,93	0,97
EDNRA_Antagonist	0,94	0,95	0,96	0,97
EDNRB_Antagonist	0,94	0,96	0,97	0,97
EGFR	1,00	1,00	1,00	1,00
ER_Alpha_Agonist	0,96	0,98	0,97	0,99
ER_Beta_Agonist	0,98	0,98	0,98	0,99
FXR_Alpha_Agonist	0,99	0,98	0,98	0,99
GASR_Antagonist	0,95	0,97	0,97	0,97
GR_Agonist	0,99	1,00	0,97	1,00
GR_Antagonist	0,97	0,98	0,99	0,99
HRH3_Antagonist	0,97	0,95	0,97	0,99
LXR_Alpha_Agonist	0,97	0,98	0,99	0,99
LXR_Beta_Agonist	0,99	0,98	0,99	1,00
NK1R_Antagonist	0,90	0,94	0,94	0,93
OPRD_Agonist	0,96	0,96	0,96	0,98
PPAR_Alpha_Agonist	0,98	0,95	0,97	0,99
PPAR_Beta_Agonist	0,99	0,96	0,98	0,99
PPAR_Gamma_Agonist	0,88	0,88	0,90	0,94
PR_Agonist	0,98	0,99	0,97	1,00
PR_Antagonist	0,95	0,98	0,98	0,98
TA2R_Antagonist	0,91	0,91	0,92	0,93

A. File Formats and Directory System

A.1 Input data format and file system

Our program supports this format of input when using directory system:

- Data/datasetName/targetName/split/targetName_active_test.sdf
- Data/datasetName/targetName/split/targetName_decoys_test.sdf
- Data/datasetName/targetName/split/targetName_active_train.sdf
- Data/datasetName/targetName/split/targetName_decoys_train.sdf
- Data/datasetName/targetName/split/targetName_active_validation.sdf
- Data/datasetName/targetName/split/targetName_decoys_validation.sdf

"TargetName" is replaced by real target name in practice and split is replaced by s_001 or s_002 etc.

When using command line the only requirement is that the files must be SDF.

A.2 The output of phase_preprocess.py

Data is saved in a text file and the system is following:

- 1. line = number of descriptors
- 2. line = number of unique fragments of given molecules
- 3. line = number of active molecules train
- 4. line = number of active molecules validation
- 5. line = number of active molecules test
- 6. line = number of inactive molecules tran
- 7. line = number of inactive molecules validation
- 8. line = number of inactive molecules test
- The same number of lines as number of fragments then contain only values of descriptors for each fragment (without name - essentialy 2D array of descriptors, each row being a fragment).
- For each number of molecules in the same order (from lines 3. to 8.) is each molecule written as line of fragments, where each fragment is represented by index to a line, which contains the descriptors of given fragment (index to a row in array described previously).
- Last line is a line with names of descriptors.

A.3 Output format

When using command line, output of python main.py -type model is

- Line with names of descriptors
- Lines with values of weights

- Line with information about calculation: AUC of train data without weights, AUC with weights, AUC with random weights, number of descriptors used and number of unique fragments in molecules.

When using command line, output of python main.py -type test is

- Line with information about calculation: AUC of test data without weights, AUC with weights, AUC with random weights, number of descriptors used and number of unique fragments in molecules.

When using the menu, output is

- Line with names of descriptors
- Lines with values of weights
- Line with information about calculation: AUC of train data without weights, AUC with weights, AUC with random weights, number of descriptors used and number of unique fragments in molecules.
- Line with information about calculation: AUC of test data without weights, AUC with weights, AUC with random weights, number of descriptors used and number of unique fragments in molecules.

When using the menu output uses directory system: Results/datasetName/targetName/fragmentTy

Through menu option [3] these outputs can be aggregated into single .csv file.

B. Programmer Documentation

The project is implemented using mix of Python, C++ and Bash. Python is used to implement the main logic and functionality including client-server communication. C++ is used to handle the most computationaly demanding part - simulated annealing and screening. Bash is used for few simple scripts. C++ program is called *screening* and can be used independently on on the python code.

We choose Python because of RDKit, which is library used to extract fragments and assign descriptors (and has many other useful functions like AUC calculation), is implemented in Python. The original plan was to use Python on screening and annealing too, but in the end we switched to C++. Hence we never actualy use RDKit in our code, but Python was still a great choice, because the general logic of a program is written much faster in this langauge.

C++ was chosen for its performance and also because it does not require any virtual machine like Java or C# does. Thus our cluster does not need to have .Net, Mono or JVM installed.

Python code consists of a set of scripts:

- **main.py** - Handles communication with user, communication with servers, manages given tasks (their assignment to servers or local computer)
- *WeFrag_server.py* - Communicates with client, runs computations on server.
- **merge_sdf.py** - Small script which merges sdf files provided.
- **remove_duplicates.py** - Small script which removes duplicate molecules from .sdf file.
- **split_info.py** - Generates info about split from given .sdf files into a .json file.
- **extract_fragments.py** - Creates .json file containing information about molecules and its fragments. This script was provided by Petr Škoda.
- **padel_descriptors.py** - Assignes PaDEL descriptors to each fragment. Requires .json file of molecules and fragments (same file which extract_fragments.py outputs). This script was provided by Petr Škoda.
- **rdkit_descriptos.py** - Assignes RDKit descriptors to each fragment. Requires .json file of molecules and fragments (same file which extract_fragments.py outputs). This script was provided by Petr Škoda.
- **phase_preprocess.py** - Final preprocess of training or test data. For train data emoves correlated descriptors, fragment noise and scales the descriptors. For test data removes descriptors descriptors not used in weights, fragment noise and scales the descriptors.

Bash scripts consist of:

- **generate_data.sh** - Generates data in a format useful for screening from the input .sdf files. Executes the scripts used for preprocessing of data and checks for existing/non-existing files. Used with the directory system. Used when commands are provided through menu.
- **cmd_line_run.sh** Does the same as generate_data.sh but do not relie on directory system and on top of that starts the screening. Used when commands are provided through command line.

- **pinger.sh** - Can check if cluster computer is online, if someone is using it or to start WeFrag_server.py on it.
- **starter.sh** - Small script which is meant to inicialise supporting software before the start of server. We are using to start RDKit, it can be rewritten for other needs.

C++ program *screening* consist of:

- **screening.cpp** - Handles main logic, reading input and producing output.
- **Comparer.cpp** and **Comparer.h** - Parent class used in order to easily implement new similarity functions for comparing molecules.
- **Euclidean_NFrag.cpp** and **Euclidean_NFrag.h** - Child of Comparer, implements Euclidean similarity and Nfrag Euclidean similarity.
- **Manhattan_NFrag.cpp** and **Manhattan_NFrag.h** - Child of Comparer, implements Manhattan similarity and Nfrag Manhattan similarity.
- **Simple_NFrag.cpp** and **Simple_NFrag.h** - Child of Comparer, implements simple similarity and Nfrag simple similarity.
- **Node.h** and **val.h** - Only supporting struct.
- **Optimizer.cpp** and **Optimizer.h** - Parent class used in order to easily implement new optimizing functions for finding weights.
- **Simulated_Annealing.cpp** and **Simulated_Annealing.h** - Child of Optimizer, implements simulated annealing.

B.1 Client-Server Communication

This whole funcionality has been designed to work mainly on the computer laboratory of Faculty of Mathematics and Physics, Charles University. In theory it should work with any cluster of computers with shared file system and other requirements fulfilled from section C.2.

Users can configure which computers to use as a part of their cluster in cluster_config.txt.

When talking about our cluster computing system, we need to first understand what a *task* is. We define task as a screening of a split or preprocessing of certain configuration. For example when user wants to perform screening of 10 splits with certain configuratin, 11 tasks are created. First the preprocessing task is executed on the cluster and then the 10 screening tasks are performed in paralel.

The server application on cluster servers is started through SSH. After that the cluster is controlled through Python sockets and data can be sent to computers through SCP. In order not to disturbe user by repeatedly asking him for password when starting the cluster (since we need ssh command for every computer in cluster), we use sshpass. However there is a security risk involved in this solution, since the sshpass command can be found in command history containing uncyphered password and login. This could be solved by enabling ssh keys, however it is another level of dificulty for new user to start using cluster. If the user is running WeFrag on his secured computer, sshpass should not be a security issue.

When starting the cluster we scan computers and those which are dead or occupied by someone else (and use significant ammount of resources) are excluded.

This scaning can be periodically repeated in order to avoid blocking any computer. The whole system is also sturdy against sudden death of a computer (tasks computed on this computer will be redirected somewhere else). The user can also configure how many tasks to compute on a computer.

The core of WeFrag_server.py is the use of sockets, threading and bash subprocess, which the python language enables. The whole process looks like this:

- Task is chosen from queue of tasks
- Server with free computation capabilites is selected
- Bash command of task in form of string is generated on client
- Socket connection to the server is established
- Server creates new thread for this connection
- String with command is sent to the server
- Server receives the command and runs it as a subprocess
- Server collects the result of the subprocess
- Server sends the result back to the client
- Client closes the socket connection and handles the response

Failur in any of the steps above should only result in putting the task to the back of the queue.

B.2 Creating molecular representation - data preprocessing

:

Input molecules in SDF format need to be preprocessed in order to be useful for computations. This is done by a combination of scripts. Each script can be interchanged by some new one, if the format of input and output stays the same. This enables to implement new fragment or descriptors types.

The sequence of data processing scripts is executed by generate_data.sh when directory system is used or by cmd_line_run.sh.

The data processing phase:

1. If the directory system is used, aggregate molecules in SDF from all splits of given target into two files in SDF (decoys and ligands) and remove duplicates. If command line is used, only remove duplicates. Handled by merge_sdf.py. Expects .sdf input and .sdf output.
2. Generate information about split we are currently preprocessing - puts names of train, validate and test molecules into new file. Handled by split_info.py. Expects .sdf as input and .json as output.
3. Create new file containinig fragments of molecules from step 1. Handled by extract_fragments.py. Expects .sdf as input and .json as output.
4. Create new file containing descriptors of fragments from step 3. Handled by rdkit_descriptors.py or PaDEL-Descriptor.jar. Expects .json as input and outputs text file in special format.
5. Create final file for train data - remove correlated descriptors and fragment noise, scale descriptors values. Handled by phase_preprocess.py. Expects output from step 2, 3 and 4 as input. Outputs text file in special format.

6. Create final file for test data - remove descriptors not used in weights, remove fragment noise, scale descriptors values. Handled by phase_preprocess.py. Expects output from step 2, 3 and 4 as input. Outputs text file in special format.

The output formats are described in A.2.

B.3 Creating New Similarity Function

The creation of new similarity function is fairly easy. All logic connected to simulated annealing and comparison of molecules is implemented in C++ program *screening*. A programer just needs to implement new C++ class which inherits from the class Comparer and implement the *calc_weights* method. Then in the *screening.cpp* utilise polymorphism to assign this new class parent Comparer pointer which is used to call the calc_weights method.

The expected behaviour is that the method according to parameters provided compares two sets of molecules and the result of each comparison pushes into vector *best_compared*.

B.4 Writing Custom Optimization Algorithm

Again this is handled in the C++ program *screening*. A programer needs to implement new c++ class which inherits from the class Optimizer and implement the *calc_weights* method. Then in the *screening.cpp* utilise polymorphism to assign this new class parent Optimizer pointer which is used to call the calc_weights method.

It is expected that a optimization algorithm changes the weights provided on input accordingly to the rest of parameters provided.

C. User Documentation

C.1 System Requirements

Unix operating system is required among with Python 3.X and RDKit software pack in order to run WeFrag on client computer for local computation.

In order to use the cluster it is required to have Unix system supporting ssh and scp on each server among with Python 3.X and RDKit. Client computer requires Python 3.X, ssh, scp and sshpass. Cluster also needs to implement shared filesystem and all computers have to use the same login and password. Very important thing is to have strong internet connection! The system was tested on 100 Mbit Ethernet. From our experience weak Wi-fi connection is not enough!

If user wants to use the PaDEL descriptors, the machine needs to support JVM.

C.2 Setup

The C++ code of screening.exe program can be compiled using Clion or CMake version 3.6. or higher. The compiled screening.exe needs to be in Code_config directory.

When setting up local computation there are 2 requirements: The users need to enable RDKit in their environment and insert data accordingly to the directory system refdir _sys.

When setting up the cluster, users need to configure two files, insert data accordingly to filesystem and run a few commands in our program.

Cluster_configuration.txt - In this file users need to configure the directory on cluster in which they wish to store code and data, number of splits in their dataset and seed they wish to use. After this, user has to write the name and IP adress of each cluster computer he wish to use on a single line. Template of configure file can be found in attachment with the rest of project.

starter.sh - In this file user can write any commands which need to be run before the start of WeFrag_server.py. For example we needed to start RDKit on each computer. Template of starter.sh can be found in attachment with the rest of project.

After having those files configured and data in filesystem, user can run the command *python main.py -type menu* and select *setup*. This command sends code to the cluster.

Last step required - users need to send data to the cluster. They can either send all of raw .sdf data (which can be fairly big) using *[7] Update Data* command or they can preprocess the data locally (preprocessed data are much smaller) and send it to the server using *[9] Update Preprocessed*.

C.3 Input - commands and data

One option to input commands is through menu C.1. When using menu data input is realised through directory system. This type of input is designed for large scale computations (on cluster or local) - meaining tens of targets, hundreds of splits and various configurations.

Figure C.1: Screenshot of WeFrag menu

```
================
Welcome to WeFrag
================

Local run, use cluster or setup cluster (when used first) [local/cluster/setup]:
local
Write a number to choose one of the following

[1] Preprocess datasets
[2] Virtual screening
[3] Get results
[4] Cluster info
[5] Computations info
[6] Update Code_config
[7] Update Data
[8] Update Preprocessed_tmp
[9] Update Preprocessed
[10] Set number of tasks per computer
[11] Set the seed for random number generation
[12] Close application
```

Other option is to input commands through command line parameters. This type of input is much more limited - user can perform 2 actions. First action is to calculate weights (with certain configuration of parameters), second action is to test the weights (with certain configuration of parameters). User cannot use cluster with command line input. This type of input is designed for small scale computation on local computer - for example quick calculation of single split in various configurations.

Run program as *python main.py -type model ...(parameters)* or *python main.py -type phase_2 ...(parameters)*. Parameter *model* computes the weights from active_train, decoys_train, active_validation and inactive_validation (validation data optional) Paramter *test* tests the weights using active_train, active_validation, active_test, decoys_test (validation data optional)

User can choose from all the parameters which are avaliable in menu mode. *model* requires: -atr, -itr, -o, -tmp *test* requires: -atr, -ate, -ite, -o, -tmp, -w

All parameters possible:

- -atr - active train
- -itr - decoys train
- -ate - active test
- -ite - decoys test
- -ava - active validation
- -iva - decoys validation
- -w - weights
- -o - output

- -tmp - tmp directory
- -f - type of fragments (ecfp.1, ecfp.2, tt.2, tt.3, tt.4) - default ecfp.1
- -p - fragment noise treshold percentage [0,100] - default 50
- -d - descriptors used (RDKit, PaDEL) - default RDKit
- -sb - scaling or binning of descriptor values - default scaling
- -corr - correlation treshold [0,1] - default 0.5
- -m - similarity function used (euclidean, manhattan, simple or Nfrag methods euclidean[0-100], manhattan[0-100], simple[0-100])
- -a - annealing type (none, rand_start, const_start)
- -T - starting temperature - default 1000
- -dist - distance of weights in annealing [0,100] - default 25
- -parts - how many parts of weights vector to change in an iteration of annealing - default 1
- -c - cooling rate - default 0.999
- -seed - Seed used for random numbers generation (RANDOM seed by default)

C.4 Examples of Usage

When using the command line input

- python main.py -type phase _1 -atr DRD2 _Antagonist _actives _train.sdf -itr DRD2 _Antagonist _decoys _train.sdf -o weights -tmp tmp
- python main.py -type phase_2 -atr DRD2 _Antagonist _actives _train.sdf -ate DRD2 _Antagonist _actives _test.sdf -ite DRD2 _Antagonist _decoys _test.sdf -o vals -tmp tmp2 -w weights
- python main.py -type phase_1 -atr DRD2 _Antagonist _actives _train.sdf -itr DRD2 _Antagonist _decoys _train.sdf -o weights -tmp tmp -f tt.2 -p 50 -d RDKit -sb scaling -corr 0.3 -m manhattan100 -a none -T 0 -dist 0 -parts 0 -c 0
- python main.py -type phase_1 -atr DRD2 _Antagonist _actives _train.sdf -itr DRD2 _Antagonist _decoys _train.sdf -o weights -tmp tmp -f tt.3 -p 25 -d PaDEL -sb scaling -corr 0.5 -m euclidean -a none -seed RANDOM
- python main.py -type phase_1 -atr DRD2 _Antagonist _actives _train.sdf -itr DRD2 _Antagonist _decoys _train.sdf -o weights -tmp tmp -f ecfp.1 -p 25 -d RDKit -sb scaling -corr 0.5 -m euclidean100 -a local -T 100 -dist 50 -parts 1 -c 0.999
- python main.py -type phase_2 -atr DRD2 _Antagonist _actives _train.sdf -ate DRD2 _Antagonist _actives _test.sdf -ite DRD2 _Antagonist _decoys _test.sdf -o vals -tmp tmp2 -f ecfp.1 -p 50 -d RDKit -sb scaling -corr 0.5 -m euclidean100 -w weights

When using the menu (menu will guide you however) - example of the usage of grid options.

1. Run python main.py -type menu
2. Write local
3. Write 2

4. Write Dataset _20 _1000 _50 _8000
5. Write ecfp.1 tt.2 (both fragments will be computed)
6. Write RDKit
7. Write euclidean100 manhattan (both similarity functions will be computed)
8. Write 100 (100% fragment noise treshold)
9. Write 1 (1 correlation treshold)
10. Write const _(start with neutral weights)
11. Write 1000 (100 starting temperature)
12. Write 0.999 (0.999 cooling)
13. Write 25 (25% distance of weights - weights can have 4 values)
14. Write 1 (1 dimension to change per iteration)
15. Write 001 (split 001)

The results will be: compute ecfp.1, euclidean100 and the rest of parameters compute ecfp.1, euclidean and the rest of parameters compute tt.2, euclidean100 and the rest of parameters compute tt.2, euclidean and the rest of parameters

Beware that when local mode is selected, all computations are computed locally and if cluster mode is selected, all computations are computed on the cluster!