FACULTY OF BIOSCIENCE ENGINEERING

HYPERDIMENSIONAL COMPUTING FOR PROTEIN LANGUAGE MODELING

...

Michael Fatjanov
Student ID: ...

Supervisor(s): Prof. Dr. Bernard De Baets

A dissertation submitted to Ghent University in partial fulfilment of the requirements for the degree of master in Bioinformatics.
Academic year: 2022-2023

GHENT UNIVERSITY

Gent, FILL IN THE DATE

The promotor,

Prof. Dr. Bernard De Baets

The author,

Michael Fatjanov

ACKNOWLEDGEMENTS

Thank you, all of you!

CONTENTS

SAMENVATTING

nederlandse samenvatting

SUMMARY

insert english summary here...

1. INTRODUCTION

1.1 outline

1.2 A historical perspective on bioinformatics

Many decades ago, around the 1950s, we did not know much about the molecules that carry our genetic information and how this would be translated into higher levels of biology. All that was known about deoxyribonucleic acid (DNA) was that it carries nucleotides in equimolar proportions so that there is as much guanine as cytodine and as much adenine as thymine. A major breakthrough came with Watson and Crick's discovery of the structure of DNA in 1953 [1]. Despite that, it took some more decades before the genetic code was deciphered and how this information is further transferred. Researchers came to know that DNA is essentially built up of a linear sequence of the 4 aforementioned nucleic acids. This sequence encodes information that undergoes transcription into ribonucleic acid (RNA) that in turn gets translated into proteins. The encoding of proteins is done in groups of three, known as codons. All of this was later stated as the *central dogma of molecular biology*, also by Crick in 1958 [2]. This was hugely important for later research since it gives us more insight into how the genetic code is translated and transferred.

In the same decade, major leaps were made in the research of protein structure and sequences. The first three-dimensional protein structures were determined via X-ray crystallography [3], which is still mostly the preferred method to this day. On top of that, the arrangement of the primary structure of a protein has been resolved after the first sequencing of a polypeptide. Sanger determined by sequencing insulin in 1953 [4] that a protein is built up of a sequence of amino acids, all connected by a peptide bond into a polypeptide. This established the idea that proteins are biological macromolecules that carry lots of information [5], which started a boom of research on more efficient methods for obtaining protein sequences. The

most popular method of that time was the Edman degradation method [6]. A major issue with this method was that only a theoretical maximum of 50 to 60 sequential amino acids could be sequenced. Larger proteins had to be cleaved into fragments that were small enough to be sequenced. Tracing back the input sequence from this data was a cumbersome process and thus published Dayhoff the first computational program applied to biological data, COMPROTEIN [7] in 1962. This program was essentially a *de novo* sequence assembler for Edman degradation data. Furthermore, sequencing amino acids was also increasingly made automated later in the 1960s. These innovations assisted the creation of the first published protein sequence database [8] in 1965.

Further in the 1960s, researchers discovered the evolutionary value of having protein sequence data of different species. The problem to be solved back then was the quantification of the similarity between sequences. Pairwise alignment algorithms such as the algorithm of Needleman & Wunsch [9] for global alignments and that of Smith & Waterman [10] for local alignments from the 1970s solved this issue and alignment is still considered to be a key bioinformatics task to this day. Together with this, mathematical frameworks for amino acid substitutions in the context of evolution such as PAMs and BLOSUMs also contributed to bioinformatics. These pairwise alignment algorithms are sufficient for comparing two sequences but unfeasible for searching databases for homologous sequences, hence faster algorithms like FASTA [11], BLAST [12] were developed in the 1980s and 1990s. These methods were and are still important for discovering functional, structural and evolutionary information in biological sequences since sequences that are in some way similar, have a high chance to have the same biological function. This also means that such sequences might be derived from a common ancestor and it became commonplace that sequence patterns may lead to structural and functional relevance. A natural extension of pairwise alignments is multiple sequence alignment (MSA) [13], which is to align multiple related sequences. The most popular and still widely used tools today include Clustal [14] and MUSCLE [15]. This reveals much more information than pairwise alignment can. It allows for the identification of conserved sequence patterns and critical amino acid residues with much more statistical significance which is of great value for constructing phylogenetic profiles of gene families [16]. This all showed the importance of computational biology and established bioinformatics as a beneficial field of science.

Development of DNA-based applications took some more time since the genetic code and how it translates to amino acids was not deciphered yet until 1968 [17]. Early DNA and RNA sequencing methods were first demonstrated in the 1970s [18, 19] and like with protein sequencing methods, these methods only became faster, more efficient and more scalable. The 1990s saw the appearance of whole genome sequencing and internet-accessible databases that we still use to this day, such as Genbank, Genomes and PubMed and so computational biology had to follow the ever-increasing amount of data that it had to process. With the advent of second-generation sequencing in the 2000s came even more Big Data issues, further challenging bioinformaticians by allowing us to sequence millions of DNA and RNA molecules in a single run. The sheer size and complexity of biological data can make it difficult to store and manage it, as for instance implementing error identification, security, quality standards and easy data retrievability. On top of that, the analysis of such large-scale data is not straightforward and traditional software tools won't be sufficient anymore. And lastly, the need for high-performance computing resources to handle all of these computational demands will only rise.

Today we see that research in the field of biology is becoming more and more computationally driven and that this trend will not slow down any time soon. As for this thesis, we will focus on this matter, more specifically on the level of protein research which is discussed in the next section.

1.3 Protein biology

1.3.1 Protein structure levels

Proteins are an essential part of molecular biology and are responsible for almost all cellular functions. They are part of the important biological macromolecules that make up life, hence a lot of effort has gone towards trying to understand the functions of protein and disruptions in its mechanisms that lead to many kinds of diseases. Proteins are composed of a linear chain of amino acids (AA) with a length ranging from 50 to tens of thousands of AAs, all connected by peptide bonds into a polypeptide. This is also referred to as the *primary structure* of a protein as mentioned earlier [5]. A sequence of amino acids is mostly determined by the genetic code without considering post-translational and post-transcriptional modifications etc. In the genetic

code of all living organisms, there are 20 different kinds of amino acids coded in that make up the 'language' of proteins. Each amino acid has the same backbone but differs by the chemical properties of its side chain, also known as the R-group. This sequence of amino acids does not occur as a mere linear chain of peptides and consists of much more intricacies, however. The polypeptide can form locally folded structures due to chemical interactions within the backbone (the polypeptide chain without the R-group), referred to as the *secondary structure* of a protein. α-helices and β-sheets are the most well-known and common examples of these structures. The overall three-dimensional structure that forms out of these structures is referred to as the *tertiary structure*. These are formed due to interactions between the R-groups and are much harder to classify due to the quasi-infinite amount of different combinations that can occur in an amino acid sequence. And lastly, a protein can also be made up of multiple polypeptide chains, referred to as its subunits. These subunits together form the *quaternary structure* of a protein.

The sought-after biochemical and cellular functions of a protein emerge from a combination of all of these structures. While a protein's 3D structure and function are dynamic and dependent on its surroundings such as the cellular state and other proteins and molecules, it is still defined by its underlying sequence. This means that a lot of the 3D-structural and functional information of a protein should be retrievable from its amino acid sequence [20]. Understanding how a protein's sequence translates to its structure and function, otherwise known as the 'protein folding problem', is the central problem of protein biology and is crucial for understanding disease mechanisms and designing proteins and drugs for therapeutic and bioengineering applications. Therefore, a lot of effort has gone into computational methods for structure and function predictions from protein sequences but this sequence-structure-function relationship continues to challenge bioinformaticians (insert visual aid). Further in this chapter, we will discuss traditional and state-of-art bioinformatics methods to obtain more knowledge in this field.

1.3.2 Methods

As mentioned earlier, the Edman degradation method was mostly used before to determine the primary structure of a protein. The current state-of-the-art methods for the identification of protein sequences are *de novo*

sequencing algorithms applied to tandem mass spectrometry data [21] and allow simultaneous sequencing of thousands of proteins per given sample. Plenty of valuable biological and evolutionary information is already retrievable from just the primary structure *via* traditional tools such as BLAST and MSA as discussed earlier and could provide more information for the prediction of secondary and tertiary structures. To cope with the number of recorded protein sequences rising exponentially, far more compute-efficient methods based on multiple sequence alignments had to be developed like PSI-BLAST [22], HHblits [23] and MMseqs [24]. However, these methods might not be able to keep up with the ever-increasing number of protein sequences stored in databases.

On the other end, the most common way to determine the 3D structure of a protein, so higher levels than the primary structure, has remained to be X-ray crystallography for more than half a century [3], with cyro-electron microscopy now catching up rapidly [25]. However, these kinds of laboratory approaches for structure determination of proteins are complex, expensive and in some cases not possible for the protein in question whilst sequence determination is relatively much easier to perform. Because of that, the structures and functions for a large fraction of the approximately 20000 known human proteins remain unknown. The number of verified three-dimensional structures in protein databases consequently has not kept up with the explosive growth in sequence information, further increasing the demand for computational structure/function prediction models.

A lot of methods have been developed to tackle this problem. Until recently, these methods were mainly based on statistical sequence models or physics-based structural simulations. *Ab initio* physics-based approaches such as ROSETTA [26] solve this problem by searching the protein's conformational space using atom energy functions and minimizing the total free energy of the system. ROSETTA has shown to be effective at predicting unknown structures and has been widely used for varying applications, but also assumes simplified energy models, is extremely computationally intensive and has limited accuracies [27]. Statistical sequence modeling of a set of related proteins, on the other hand, has proven to be very useful for discovering evolutionary constraints, homology searches and predicting residue-residue contacts. Improvement of these models has mainly been data-driven; exploiting databases to build large deep learning systems which culminated in the recent success of AlphaFold2 [28] at the Critical Assessment of protein Structure Prediction (CASP) 14. However, all of these

models are supervised methods that require labels. Labeled protein structure data essentially means retrieving the 3D coordinates of every atom in a protein which is very labor-intensive and time-consuming. Also, such kind of models would likely perform poorly when working with completely unrelated proteins since the model won't be trained on this kind of data.

One underlying theme of these computational methods for protein structure prediction is being able to translate the protein 'language' into numerical representations which computers can learn from. This is now known as the field of protein language modeling which is more thoroughly discussed in section 1.4.

1.4 State-of-the-art protein language modeling

It is intuitive to represent a protein as a sequence of letters with each letter corresponding to an amino acid. As with natural languages, we can find common elements between naturally evolved proteins. Noticeable patterns reoccurring in multiple (related) protein sequences are highly likely to be biologically relevant. These motifs and domains are essential to many biological processes and can easily be represented as words, phrases or sentences of amino acids in a language model perspective. This is why researchers are taking inspiration from the recent successes of natural language processing (NLP) and applying this to a biological context. NLP is a branch of artificial intelligence (AI) concerning itself with creating the ability for computers to learn and understand human languages by using statistical, machine learning and in recent years deep learning models. Common tasks in NLP include part-of-speech tagging (grouping words based on their function in a sentence), named entity recognition (recognizing specific entities in a sentence such as locations, persons, dates etc.) and natural language generation (letter/word prediction).

As with protein modeling, applying labels to millions of natural language containing web pages, articles, journals etc. is a labor-intensive procedure and thus state-of-the-art NLP models use a form of *self-supervised learning*, a form of unsupervised learning in which the context of the text is learned to fill in missing words, predict the next word in a sentence etc. during the training (insert visual aid). Well-known NLP methods of this kind include

bi-directional long-short term recurrent neural networks (biLSTMs) such as ELMo [29] and more recently transformers such as Google's BERT [30] and OpenAI's GPT-3 [31]. Despite the simplicity of these tasks, it is found to develop interesting capabilities as the scale of the model increases together with very little training on a specific task, now mostly referred to as *few-shot learning*.

These deep-learning methods also show promise in the field of protein biology with the most notable latest projects being TAPE [32], ProtTrans [33] and Meta AI's ESM-2 [34], one of the most recent and largest protein language models to ever have been developed at the time of writing. ESM-2 shows much success in capturing evolutionary information and structure prediction tasks. It consists of transformer protein language models with up to 15 billion parameters trained on 65 million unique sequences. Like natural language models, it has been observed

1.5 Hyperdimensional computing

Hyperdimensional computing (HDC) is a relatively new paradigm of computing in which data is represented and manipulated by high-dimensional (or hyperdimensional) vectors in the range of tens of thousands bit. This includes metrics for similarity measurements and manipultation by operations. This framework, developed by Kanerva [35], is inspired by the workings of the human brain and its ability to adapt, learn fast and easily understand semantic relations. The human brain consists of about 100 billion neurons (nerve cells) and 1000 trillion synapses that connect these neurons. Each neuron is connected to up to 10000 other neurons, creating massive circuits. This is likely fundamental to the workings of the human brain and what separates our brains from modern von Neumann computer architectures which operate on 8 to 64-bit vectors. This becomes clear when we compare the relative simplicity for a human to learn a language compared to computers. Computers use a large and complicated set of arithmetic operations in the form of deep learning networks which require terabytes of data and thousands of Watts of computing power to come close to mastering a language whilst a human can recognize other languages relatively easily when they don't even speak it. Likewise languages, we can very easily memorize and compare other intrinsically complex and contextual concepts such as images. A computer would have a hard time finding similarities

between a set of images and faces because this requires very complex machine learning models. The human brain can do this all with a very large efficiency by consuming only roughly 20 W of energy.

Achieving these kinds of flexible brain-like models based on high dimensionality is not entirely new and is being explored since the 1990s. Some of these earlier models include Holographic Reduced Representations [36], Spatter Code [37] etc. A hyperdimensional vector (HDV) can represent anything from a scalar number to any kind of concept. This vector is initially made up of totally random elements, but with a simple set of operations which will be explained later, we can use other vectors to combine some concepts into new similar or dissimilar concepts. For example, to show the essence of HDC and how it tries to simulate the brain, we can compare the concept of a *table* to the concept of a *brocolli*. We would not immediately conclude that they are in any way similar but as humans, we can trace back *table* to *plate* which has some similarities with *food* from which we can easily extract the concept of *brocolli* as in equation 1.1. These kinds of operations are not very obvious for a classical computer but creating these semantic pathways and recognizing links between distant objects are rather easy for humans.

$$\begin{aligned} &\text{table} \neq \text{brocolli} \\ &\text{table} \approx \text{plate} \approx \text{food} \approx \text{brocolli} \end{aligned} \tag{1.1}$$

The elements in an HDV can be made up of binary bits (values from the set 0, 1) like in classical computing but also of bipolar (values from the set -1, 1) or real numbers. The choice of the nature of the elements has also implications on the nature of the different operations and possibly the results.

An initial HDV is made up fully randomly. This *holistic* or *holographic* representation of a concept smeared out over a vector consisting of thousands of bits gives rise to interesting properties such as its robustness. These kinds of systems are very tolerant to noise and failure of bits since we introduce a lot of redundancy in the vector just by stochastics. This is very unlike classical computing where every bit counts and one failure in a bit can lead to immediate data corruption. Besides its robustness, it also has the potential to show much faster and more efficient calculations than traditional computer systems since it allows for more efficient data storage by encoding multiple objects into a vector.

1.5.1 Operations on hyperdimensional vectors

The interesting properties of HDC are based on only four basic operations we can perform on HDVs. We will discuss these for bipolar and binary vectors.

1.5.1.1 Similarity measurement

For many kinds of problems, it will be necessary to quantify the similarity between two HDVs. The method depends on the nature of the vectors. For binary vectors, the *Hamming distance* defined as in equation 1.2 is widely used.

$$Ham(A,B) = \frac{1}{d}\sum_{i=1}^{d} 1_{A_{(i)} \neq B_{(i)}} \tag{1.2}$$

The *cosine distance* as defined in equation 1.3 is most commonly used for bipolar vectors.

$$cos(A,B) = \frac{A \cdot B}{||A|| * ||B||} \tag{1.3}$$

The results of both of these measurements are summarized in table 1.1.

Table 1.1: Overview of similarity measurements in HDC depending on the nature of the HDVs

Measurement	Dissimilar	Orthogonal	Similar
Hamming distance	1	0.5	0
Cosine similarity	-1	0	1

It is important to note that two random HDVs will be quasi-orthogonal to each other just by stochastics. Also notice that the first quantifies a distance and the latter a similarity.

1.5.1.2 Addition

Also referred to as *bundling* or *aggregation*, the element-wise addition as in equation 1.4 of n input vectors $\{X_1 + X_2 + \cdots + X_n\}$ creates a vector X that is similar to the input vectors.

$$X = X_1 + X_2 + \cdots + X_n \tag{1.4}$$

For bipolar vectors, this entails a straightforward element-wise addition. The resulting vector is restricted to a bipolar nature too depending on the sign of each element, thus containing only -1, 1 but allowing 0 for elements that are in disagreement as shown in the following 6-dimensional example.

$$\begin{array}{rcccccc} X_1 = & +1 & -1 & +1 & +1 & -1 & -1 \\ X_2 = & +1 & +1 & +1 & -1 & -1 & -1 \\ X_3 = & -1 & -1 & +1 & +1 & -1 & +1 \\ X_4 = & -1 & -1 & -1 & +1 & -1 & +1 \\ \hline X_1 + X_2 + X_3 + X_4 = & 0 & -1 & +1 & +1 & -1 & 0 \end{array}$$

For binary vectors, the vectors are element-wise bundled based on the majority element. This is no problem if an odd number of input vectors are considered but ambiguity rises when bundling an even set of vectors. This can be solved by setting the element in question randomly. [38] Another possibility is to add another random vector however this may seem to add more unnecessary noise, especially when bundling a low number of vectors. We can also reverse this by an *inverse addition*. For bipolar vectors, this means just multiplying the vector of interest by -1. A binary vector can be flipped bit-wise.

Similar to an ordinary arithmetic summation, the bundling addition of hyperdimensional vectors is commutative so the result is not dependent on the order of addition.

$$X_1 + X_2 = X = X_2 + X_1 \tag{1.5}$$

1.5.1.3 Multiplication

Also referred to as *binding*, two vectors can be multiplied element-wise resulting in a vector maximally dissimilar to the input vectors. Vectors X and Y are bound together forming Z being orthogonal to X and Y as shown in equation 1.6.

$$Z = X * Y \tag{1.6}$$

This *binding* operation translates to a simple arithmetic element-wise multiplication for bipolar vectors. For binary vectors, this is represented by a *XOR*

bit-operation shown as follows.

$X =$	1	0	1	1	0	0
$Y =$	1	1	0	1	0	1
$X * Y =$	0	1	1	0	0	1

This operation can also be undone by multiplying with the same vector again. It is its own inverse so that

$$A * A = O \text{ where } O \text{ is a vector containing only 0s} \tag{1.7}$$

Likewise an ordinary multiplication, this operation is commutative and distributive over additions, meaning that transforming a bundle of concepts with binding is equivalent to binding every element before bundling.

$$A = Z * (X + Y) = XZ + YZ \tag{1.8}$$

1.5.1.4 Permutation

The permutation operation of an HDV, also known as *shifting*, is a simple reordering of the HDV. This can be random but a circular shift is widely employed [39] and makes the operation easily reversible. This results in a vector technically dissimilar from the input vector but still encoding its information. This will become important later when it will be used to encode sequential information such as tokens in a text. This operation will be denoted by Π.

$X =$	1	0	1	1	0	0
$\Pi(X) =$	0	1	0	1	1	0

1.5.2 Examples

There are many interesting possibilities given the relative simplicity of all these operations. We shall illustrate some applications and examples. From here, all implementations are written in the programming language Julia unless noted otherwise. All examples have been implemented with binary hyperdimensional vectors but with minimal changes, bipolar vectors could also be applied. Some key snippets of the code will be shown occasion-

ally but to preserve the self-contatinment of this project, all source code is provided in the appendicies.

1.5.2.1 Simple example

To get a feel for the operations, assume A, B, C, X, Y and Z to be random 10000-dimensional bipolar hypervectors and that $D = X * A + Y * B + Z * C$, let us then try to retrieve A from D by using the defined operations. We generate for A, B, C, X, Y and Z each a random 10000-D vector *via* function:

```
bitHDV(N::Int=10000) = bitrand(N)
```

To retrieve an approximation of A, D can be multiplied by X and the rest of the included vectors are then regarded as noise as done in equations 1.9. Because of the robustness of hyperdimensional vectors, a lot of information of A should still be contained within D.

$$\begin{aligned} A' &= X * D \\ &= X * (X * A + Y * B + Z * C) \\ &= \underbrace{X * X * A}_{A} + \underbrace{X * Y * B + X * Z * C}_{\text{noise}} \\ &\approx A \end{aligned} \tag{1.9}$$

Julia makes it possible to write incredibly efficient programs whilst still being a high-level and interpreter-based programming language including many packages suited for mathematical operations. By using Julia, the following functions are written to respectively add, multiply and compare hyperdimensional binary vectors. These are used in all further applications in this project.

```
function bitadd(vectors::BitVector ...)
    v = reduce(.+, vectors)
    n = length(vectors) / 2
    x = [i > n ? 1 : i < n ? 0 : rand(0:1) for i in v]
    return convert(BitVector, x)
end
bitbind(vectors::BitVector ...) =  reduce(.⊻, vectors)
hamming(x::BitVector, y::BitVector) = sum(x .!= y)/length(x)
```

Lastly, the procedure of equation 1.9 is repeated 10000 times because of the stochastic nature of these vectors. The results are illustrated in figure 1.1.

```
v = Vector()
for i in 1:10000
    a, b, c, x, y, z = bitHDV(), bitHDV(), bitHDV(), bitHDV(),
 ↪  bitHDV(), bitHDV()
    d = bitadd(bitbind(a, x), bitbind(b, y), bitbind(c, z))
    ap = bitbind(x, d)
    score = hamming(ap, a)
    append!(v, score)
end
```

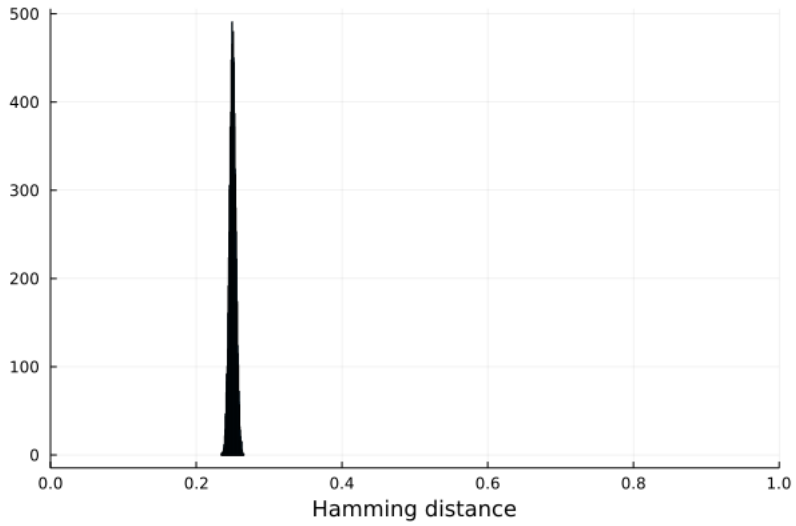

Figure 1.1: 10000 cases of random 10000-dimensional binary vectors are made and each time implemented following example 1.9. The resulting Hamming distances between A and A' are then plotted in a histogram.

We see that we can retrieve a lot of information with most of the Hamming distances centering around 0.25. Notice that two completely random HDVs would have a distance close to 1 just by stochastics. A Hamming distance of 0 would mean that we retrieved all bits of A correctly, which is impossible in this model due to the consideration of noise. This result is not comparable to state-of-the-art accuracies but the calculations are very efficient nonetheless as all of these calculations were done in less than 2 seconds. This same experiment was done with random bipolar 10000-dimensional vectors and it performs slightly slower as expected but retained the accuracy.

1.5.3 Examples of hyperdimensional computing on datasets

Now, the power of these simple operations will be demonstrated by applying them to a couple of datasets.

1.5.3.1 Zoo animal classification

As the first example, we will consider a simple dataset[40] containing 101 animals with 17 features such as their number of legs, their skin covering and other physical properties. In the end, we want to create a simple model that can classify these animals or other animals not present in the dataset. To tackle this problem, we first assign to each feature a random hyperdimensional vector. For each animal, all of its features can be bundled to obtain a final vector representing the animal. For example, it is known that a chicken lays eggs, is covered with feathers and has two legs so then these features can be bundled as in the following equation. C is a vector representing a chicken, E the ability to lay eggs, F the possession of feathers and T the possession two legs:

$$C = E + F + T \tag{1.10}$$

This is simple for all the binary features but the feature for the number of legs is variable. Although it is possible to assign completely random vectors to each number of legs, it would make a slightly more biologically realistic model if an animal with 2 legs would be more similar to one with 4 legs than to one with 8 legs. To address this, a range of numbers would have to be representable by hyperdimensional vectors, the range from 0 to 8 in this case. First, a random hyperdimensional vector representing the lower bound of the interval is generated. Next, a vector representing the next step in the interval is constructed by replacing a fraction of the vector with random bits. This last step is then repeated to obtain a vector of each number in a range.

To demonstrate the binding of HDVs, the types of features will be encoded into our feature set. For example, the 'milk' (M), and 'egg' (E) features give us information about the growth of the animal, so we will bind these features to another vector representing the growth feature (G) to obtain a more expanded model. This is also done for the skin protection features (S) and all the features considering the limbs (L). This also gives us the possibility to retrieve some features of the animals as in the procedure shown in the

previous example. Thus, equation 1.10 can be expanded into:

$$C = G * E + S * F + L * T \tag{1.11}$$

After all these procedures, there are 101 animals with each a 10000-dimensional vector as a feature. To effectively analyze this data, a principal component analysis (PCA) has been applied to essentially reduce the 10000 features into 2. This projection can then be easily shown in a 2D plot, resulting in figure 1.2. This shows 3 distinguishable clusters of animal classes that make sense from an evolutionary standpoint. The model could be easily improved on this part by including more features that should show more separation of these classes.

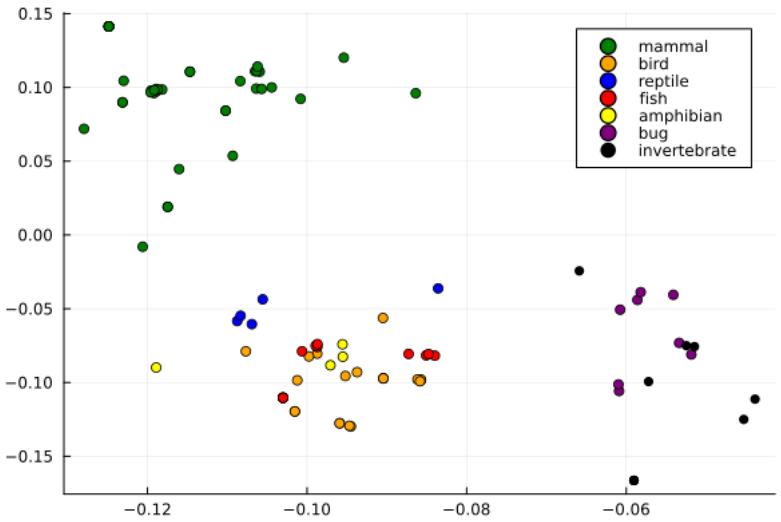

Figure 1.2: Scatter-plot of the two first principal components (PCs) of a 101×10000 matrix containing hyperdimensional vectors for every animal in the zoo dataset after a PCA procedure. These PCs account for roughly 48 % of the variance.

After an HDV has been made for every animal, all animals of the same class can be bundled to obtain an HDV representing the said class. So for example, if we have a hyperdimensional vector for a pigeon (P), chicken (C) and a kiwi (K), an HDV representing birds (B) can be made by doing:

$$B = P + C + K \tag{1.12}$$

To classify an animal, its HDV can be compared to the HDVs of every class and the most similar vector is then assumed to be its class. This has been tested for every animal in the dataset to observe if the animal's class type

is retrievable from its HDV. Depending on the run, 94 to 97 % of the animals' class type were retrievable. For further improvement, it would be possible to generate a set of animals not present in the dataset and test those in order to further understand how this model can be improved.

1.5.3.2 Protein classification

To illustrate an example more akin to this research topic, a model based on the principles of hyperdimensional computing will be built to classify a protein sequence dataset[41]. It contains 949 manually curated peptide sequences with their membranolytic anti-breast cancer activity level (very active, moderately active, experimentally inactive and virtually inactive). The virtually inactive sequences are omitted since they comprise 80 % of the dataset and thus tend to oversaturate the data. The model will be built with mostly the same procedure as for the animal classifier, but instead of animals, sequences have to be encoded into HDVs now. First, a random HDV is generated for every amino acid. Physicochemical properties, evolutionary constraints etc. could be introduced to make this model more realistic but that is not necessary for this demonstration. Next, a peptide sequence is to be considered as a bag of trimers. A vector representing a trimer is generated by bundling the three amino acids whilst retaining sequential information by shifting as in equation 1.13.

$$ABC = A + \Pi(B) + \Pi(\Pi(C)) \tag{1.13}$$

All retrievable trimers from a given sequence are then bundled together, forming a vector representative of the sequence. From here on, the same procedures as in the last example can be applied here too, so all HDVs of a class are bundled for further analysis. The PCA procedure did not generate interesting results because the two first principal components explained only 21 % of the variance. This means that it is not feasible to reduce the 10000 dimensions of the vectors to 2, likely because the information is too smeared out over the vectors. This occurrence is highly dependent on the training data. Nevertheless, this follows the philosophy of hyperdimensional computing in keeping holistic representations of concepts.

Next, a classifier was made correspondingly. Except that here, the dataset was split into a training set (comprising 80 % of the sequences) and a test set to better simulate how the model would perform with new data. With

100 runs, it could predict the class on average 85 % of the test sequences.

BIBLIOGRAPHY

[1] J. D. WATSON and F. H. C. CRICK. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, April 1953.

[2] FRANCIS CRICK. Central dogma of molecular biology. *Nature*, 227(5258):561–563, August 1970.

[3] J. C. Kendrew, G. Bodo, H. M. Dintzis, R. G. Parrish, H. Wyckoff, and D. C. Phillips. A three-dimensional model of the myoglobin molecule obtained by x-ray analysis. *Nature*, 181(4610):662–666, March 1958.

[4] F. Sanger and E. O. P. Thompson. The amino-acid sequence in the glycyl chain of insulin. 1. the identification of lower peptides from partial hydrolysates. *Biochemical Journal*, 53(3):353–366, February 1953.

[5] JOSEPH S. FRUTON. Early theories of protein structure. *Annals of the New York Academy of Sciences*, 325(1):1–20, 1979.

[6] P. Edman and G. Begg. A protein sequenator. *European Journal of Biochemistry*, 1(1):80–91, March 1967.

[7] J Wesley Leas. *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*. ACM, 1962.

[8] Robert T. Hersh, Richard V. Eck, and Margaret O. Dayhoff. Atlas of protein sequence and structure, 1966. *Systematic Zoology*, 16(3):262, September 1967.

[9] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[10] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[11] David J. Lipman and William R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[12] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[13] Michio Murata, Jane S Richardson, and Joel L Sussman. Simultaneous comparison of three protein sequences. *Proceedings of the National Academy of Sciences*, 82(10):3073–3077, 1985.

[14] Desmond G Higgins and Paul M Sharp. Clustal: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244, 1988.

[15] Robert C Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792–1797, 2004.

[16] Phylogeny in multiple sequence alignments. In *Multiple Biological Sequence Alignment: Scoring Functions, Algorithms and Applications*, pages 103–112. John Wiley & Sons, Inc., June 2016.

[17] Francis HC Crick. The origin of the genetic code. *Journal of molecular biology*, 38(3):367–379, 1968.

[18] F Sanger, G M Air, B G Barrell, N L Brown, A R Coulson, C A Fiddes, C A Hutchison, P M Slocombe, and M Smith. Nucleotide sequence of bacteriophage phi X174 DNA. *Nature*, 265(5596):687–695, February 1977.

[19] W. MIN JOU, G. HAEGEMAN, M. YSEBAERT, and W. FIERS. Nucleotide sequence of the gene coding for the bacteriophage MS2 coat protein. *Nature*, 237(5350):82–88, May 1972.

[20] O.B. Ptitsyn. How does protein synthesis give rise to the 3d-structure? *FEBS Letters*, 285(2):176–181, 1991.

[21] Simone König, Wolfgang M. J. Obermann, and Johannes A. Eble. The current state-of-the-art identification of unknown proteins using mass spectrometry exemplified on de novo sequencing of a venom protease from bothrops moojeni. *Molecules*, 27(15), 2022.

[22] S. Altschul. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, September 1997.

[23] Martin Steinegger, Markus Meier, Milot Mirdita, Harald Vöhringer, Stephan J. Haunsberger, and Johannes Söding. HH-suite3 for fast remote homology detection and deep protein annotation. *BMC Bioinformatics*, 20(1), September 2019.

[24] Martin Steinegger and Johannes Söding. MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. *Nature Biotechnology*, 35(11):1026–1028, October 2017.

[25] Ka Man Yip, Niels Fischer, Elham Paknia, Ashwin Chari, and Holger Stark. Atomic-resolution protein structure determination by cryo-EM. *Nature*, 587(7832):157–161, October 2020.

[26] Andrew Leaver-Fay, Michael Tyka, Steven M. Lewis, Oliver F. Lange, James Thompson, Ron Jacak, Kristian W. Kaufman, P. Douglas Renfrew, Colin A. Smith, Will Sheffler, Ian W. Davis, Seth Cooper, Adrien Treuille, Daniel J. Mandell, Florian Richter, Yih-En Andrew Ban, Sarel J. Fleishman, Jacob E. Corn, David E. Kim, Sergey Lyskov, Monica Berrondo, Stuart Mentzer, Zoran Popović, James J. Havranek, John Karanicolas, Rhiju Das, Jens Meiler, Tanja Kortemme, Jeffrey J. Gray, Brian Kuhlman, David Baker, and Philip Bradley. Rosetta3. In *Computer Methods, Part C*, pages 545–574. Elsevier, 2011.

[27] Tristan Bepler and Bonnie Berger. Learning the protein language: Evolution, structure, and function. *Cell Systems*, 12(6):654–669.e3, June 2021.

[28] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, July 2021.

[29] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[31] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[32] Roshan Rao, Nicholas Bhattacharya, Neil Thomas, Yan Duan, Xi Chen, John Canny, Pieter Abbeel, and Yun S. Song. Evaluating protein transfer learning with tape, 2019.

[33] Ahmed Elnaggar, Michael Heinzinger, Christian Dallago, Ghalia Rihawi, Yu Wang, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Martin Steinegger, Debsindhu Bhowmik, and Burkhard Rost. Prottrans: Towards cracking the language of life's code through self-supervised deep learning and high performance computing, 2020.

[34] Zeming Lin, Halil Akin, Roshan Rao, Brian Hie, Zhongkai Zhu, Wenting Lu, Nikita Smetanin, Robert Verkuil, Ori Kabeli, Yaniv Shmueli, Allan dos Santos Costa, Maryam Fazel-Zarandi, Tom Sercu, Salvatore Candido, and Alexander Rives. Evolutionary-scale prediction of atomic level protein structure with a language model. July 2022.

[35] Pentti Kanerva. Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors. *Cognitive Computation*, 1(2):139–159, jun 2009.

[36] T.A. Plate. Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6(3):623–641, 1995.

[37] Pentti Kanerva. The spatter code for encoding concepts at many levels. pages 226–229, 1994.

[38] Manuel Schmuck, Luca Benini, and Abbas Rahimi. Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory. *J. Emerg. Technol. Comput. Syst.*, 15(4), oct 2019.

[39] Lulu Ge and Keshab K. Parhi. Classification using hyperdimensional computing: A review. *IEEE Circuits and Systems Magazine*, 20(2):30–47, 2020.

[40] UCI Machine Learning. Zoo animal classification, 2016.

[41] Francesca Grisoni, Claudia S. Neuhaus, Miyabi Hishinuma, Gisela Gabernet, Jan A. Hiss, Masaaki Kotera, and Gisbert Schneider. 'de novo design of anticancer peptides by ensemble artificial neural networks. *Journal of Molecular Modeling*, 25(5):122, 2019.

APPENDIX A

DEFINE BASIC OPERATIONS OF HYPERDIMENSIONAL COMPUTING

All provided code is written in the programming language Julia unless noted otherwise.

A.1 Binary hypervectors

```
using Random
"""
Construct a binary vector. By default 10000 elements long.
"""
bitHDV(N::Int=10000) = bitrand(N)
"""
Bundles binary hyperdimensional vectors based on the
 ↪  element-wise majority rule.
"""
function bitadd(vectors::BitVector ...)
    v = reduce(.+, vectors)
    n = length(vectors) / 2
    x = [i > n ? 1 : i < n ? 0 : rand(0:1) for i in v]
    return convert(BitVector, x)
end
"""
Binds binary hyperdimensional vectors based on an element-wise
 ↪  XOR gate.
"""
bitbind(vectors::BitVector ...) =  reduce(.⊻, vectors)
"""
```

```
Permutes a binary hyperdimensional vector by an adjustable
 ↪  circular shift.
"""
bitperm(vector::BitVector, k::Int=1) = circshift(vector, k)
"""
Calculates the Hamming distance between two binary vectors.
"""
hamming(x::BitVector, y::BitVector) = sum(x .!= y)/length(x)
```

A.2 Bipolar hypervectors

```
using Random
using LinearAlgebra
"""
Construct a bipolar vector. By default 10000 elements long.
"""
hdv(N::Int=10000) = rand((-1,1), 1, N)
"""
Bundles bipolar hyperdimensional vectors.
"""
add(vectors::Vector{Int}...) = reduce(.+, vectors) .|> sign
"""
Binds bipolar hyperdimensional vectors.
"""
multiply(vectors::Vector{Int}...) = reduce(.*, vectors)
"""
Permutes a bipolar hyperdimensional vector by an adjustable
 ↪  circular shift.
"""
perm(vector::Vector, k::Int=1) = circshift(vector, (0, k))
"""
Calculates the cosine similarity between two bipolar vectors.
"""
cosine(x::Vector{Int}, y::Vector{Int}) = dot(x, y) / (norm(x) *
 ↪  norm(y))
```

APPENDIX B

EXAMPLES OF HYPERDIMENSIONAL COMPUTING ON DATASETS

B.1 Simple example

```
v = Vector()
for i in 1:10000
    a, b, c, x, y, z = bitHDV(), bitHDV(), bitHDV(), bitHDV(),
 ↪  bitHDV(), bitHDV()
    d = bitadd(bitbind(a, x), bitbind(b, y), bitbind(c, z))
    ap = bitbind(x, d)
    score = hamming(ap, a)
    append!(v, score)
end

using Plots
let
        p = histogram(v, xlims=(0,1), label="", xlabel="Hamming
 ↪  distance")
end
```

B.2 Zoo example

B.2.1 Data preprocessing

```
using DataFrames, CSV
data = CSV.read("zoodata/zoo.csv", DataFrame)
```

```
classls = data.class_type

mammal_idx = [i for i in 1:101 if data.class_type[i] == 1]
bird_idx = [i for i in 1:101 if data.class_type[i] == 2]
reptile_idx = [i for i in 1:101 if data.class_type[i] == 3]
fish_idx = [i for i in 1:101 if data.class_type[i] == 4]
amphibian_idx = [i for i in 1:101 if data.class_type[i] == 5]
bug_idx = [i for i in 1:101 if data.class_type[i] == 6]
invertebrate_idx = [i for i in 1:101 if data.class_type[i] ==
 ↪  7]

indices = [mammal_idx, bird_idx, reptile_idx, fish_idx,
 ↪  amphibian_idx, bug_idx, invertebrate_idx]

legs0 = [i == 0 ? 1 : 0 for i in data.legs]
legs2 = [i == 2 ? 1 : 0 for i in data.legs]
legs4 = [i == 4 ? 1 : 0 for i in data.legs]
legs5 = [i == 5 ? 1 : 0 for i in data.legs]
legs6 = [i == 6 ? 1 : 0 for i in data.legs]
legs8 = [i == 8 ? 1 : 0 for i in data.legs]

select!(data, Not(:legs))

data[!, :legs0] = legs0
data[!, :legs2] = legs2
data[!, :legs4] = legs4
data[!, :legs5] = legs5
data[!, :legs6] = legs6
data[!, :legs8] = legs8
```

B.2.2 Assigning HDVs to features

```
"""
Creates a list of binary hyperdimensional vectors representing
 ↪  an interval of numbers by constructing a random HDV
 ↪  representing the lower bound of the interval and replacing
 ↪  a fraction of the vector with random bits.
"""
```

```
function range_hdvs(steps)
        k = length(steps) - 1
        V = [bitHDV() for i in 1:k+1]
        for i in 2:k+1
                for j in 1:10000
                        V[i][j] = rand(0:1)
                end
        end
        return V
end

legs_steps = 0:1:8
legs_hdvs = range_hdvs(legs_steps)

feature_hdv=[bitHDV() for i in 1:15]
append!(feature_hdv, [legs_hdvs[1], legs_hdvs[3], legs_hdvs[5],
 ↪  legs_hdvs[6], legs_hdvs[7], legs_hdvs[9]])

growth_hdv = bitHDV()
skin_protection = bitHDV()
limbs = bitHDV()

for i in 1:2
    feature_hdv = replace!(feature_hdv, feature_hdv[i] =>
 ↪  bitbind(feature_hdv[i], skin_protection))
end

for i in 3:4
    feature_hdv = replace!(feature_hdv, feature_hdv[i] =>
 ↪  bitbind(feature_hdv[i], growth_hdv))
end

for i in 12:13;16:21
    feature_hdv = replace!(feature_hdv, feature_hdv[i] =>
 ↪  bitbind(feature_hdv[i], limbs))
end

hdv = BitVector[]
for i in 1:101
```

```
    v = BitVector[]
    for j in 2:16;18:23
        if data[!, j][i] == 1
            if j < 18
                push!(v, feature_hdv[j-1])
            end
            if j > 17
                push!(v, feature_hdv[j-2])
            end
        end
    end
    x = bitadd(v...)
    push!(hdv, x)
end

data[!, :species_hdv] = hdv

function grouper(group::Int)
    v = BitVector[]
    for i in 1:length(indices[group])
        push!(v, data.species_hdv[indices[group][i]])
    end
    x = bitadd(v...)
    return x
end

mammal_hdv = grouper(1)
bird_hdv = grouper(2)
reptile_hdv = grouper(3)
fish_hdv = grouper(4)
amphibian_hdv = grouper(5)
bug_hdv = grouper(6)
invertebrate_hdv = grouper(7)
```

B.2.3 Data analysis

```
list_group_hdvs = [mammal_hdv, bird_hdv, reptile_hdv, fish_hdv,
 ↪  amphibian_hdv, bug_hdv, invertebrate_hdv]
matrix_groups = permutedims(hcat(list_group_hdvs...))
```

data.species_hdv[94] refers to the 94th animal in the dataset, in this case that is vampire.

```
println("Similarity to mammal =
 ↪  ",string(hamming(data.species_hdv[94], mammal_hdv)))
println("Similarity to bird
 ↪  =",string(hamming(data.species_hdv[94], bird_hdv)))
println("Similarity to reptile
 ↪  =",string(hamming(data.species_hdv[94], reptile_hdv)))
println("Similarity to amphibian
 ↪  =",string(hamming(data.species_hdv[94], amphibian_hdv)))
println("Similarity to bug
 ↪  =",string(hamming(data.species_hdv[94], bug_hdv)))
println("Similarity to fish
 ↪  =",string(hamming(data.species_hdv[94], fish_hdv)))
println("Similarity to invertebrate
 ↪  =",string(hamming(data.species_hdv[94], invertebrate_hdv)))

axolotl_features = [0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0,
 ↪  0, 0, 0, 1, 0, 0, 0]
v = BitVector[]
    for j in 1:16
        if axolotl_features[j] == 1
            push!(v, feature_hdv[j])
        end
    end
    axolotl_hdv = bitadd(v...)


using MultivariateStats
M = fit(PCA, matrix_groups ; maxoutdim=2)
proj = projection(M)
```

```
using Plots
function plotter()
    colors = [:green, :orange, :blue, :red, :yellow, :purple,
 ↪  :black]
    labels = ["mammal", "bird", "reptile", "fish", "amphibian",
 ↪  "bug", "invertebrate"]
    fig = Plots.plot()

    for i in 1:7
        scatter!(fig, (proj[i, 1], proj[i, 2]),
 ↪  label=labels[i], mc = colors[i])
    end
    return fig
end
plotter()

matrix_species = permutedims(hcat(hdv...))
S = fit(PCA, matrix_species ; maxoutdim=2)
projS = projection(S)

using Plots
function plotter()
    colors = [:green, :orange, :blue, :red, :yellow, :purple,
 ↪  :black]
    labels = ["mammal", "bird", "reptile", "fish", "amphibian",
 ↪  "bug", "invertebrate"]
    fig = Plots.plot()

    for i in 1:7
        scatter!(fig, projS[indices[i], 1], projS[indices[i],
 ↪  2], mc = colors[i], label=labels[i])
    end
    return fig
end
plotter()
```

B.2.4 Animal classifier

```
function predict(animal)
    y = [hamming(mammal_hdv, animal), hamming(bird_hdv,
 ↪  animal), hamming(reptile_hdv, animal), hamming(fish_hdv,
 ↪  animal), hamming(amphibian_hdv, animal), hamming(bug_hdv,
 ↪  animal), hamming(invertebrate_hdv, animal)]
    return findmin(y)[2]
end
using StatsBase
pred = [predict(i) for i in data.species_hdv]
mean(classls .== pred)
```

B.3 Anticancer protein example

B.3.1 Data preprocessing

```
using DataFrames, CSV
data = CSV.read("ProtExdata/ACPs_Breast_cancer.csv", DataFrame)

unique(data.class)
class_num = [i == "very active" ? 1 : i == "mod. active" ? 2 :
 ↪  i == "inactive - exp" ? 3 : 4 for i in data.class]
data[!, :class_num] = class_num
```

B.3.2 Assigning HDVs to features

```
AA_list = ['A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I',
 ↪  'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W', 'Y', 'V', 'O', 'U',
 ↪  'B', 'J', 'Z', 'X']
AA_hdv = [bitHDV() for i in AA_list]
AA_dict = Dict(zip(AA_list, AA_hdv))

trimer_hdvs = Dict(aa1 * aa2 * aa3 =>
bitbind(AA_dict[aa1], bitperm(AA_dict[aa2]),
 ↪  bitperm(AA_dict[aa3], 2))
```

```
for aa1 in AA_list for aa2 in AA_list for aa3 in AA_list)

function embedder(sequence)
    l = [trimer_hdvs[sequence[i:i+2]] for i in
 ↪  1:length(sequence)-2]
    v = bitadd(hcat(l)...)
    return v
end
l = BitVector[]
for i in data.sequence
    push!(l, embedder(i))
end
data[!, :hdv] = l

active_hdv = bitadd(hcat([i for i in data[data.class_num .== 1,
 ↪  :hdv]])...)
modactive_hdv = bitadd(hcat([i for i in data[data.class_num .==
 ↪  2, :hdv]])...)
notactive_exp_hdv = bitadd(hcat([i for i in data[data.class_num
 ↪  .== 3, :hdv]])...)
notactive_virt_hdv = bitadd(hcat([i for i in
 ↪  data[data.class_num .== 4, :hdv]])...)
```

B.3.3 Data analysis

Print the similarity of the HDV of sequence "KWKLFKKILKFLHLAKKF" to every class HDV

```
println(hamming(data[data.sequence .== , :hdv]..., active_hdv))
println(hamming(data[data.sequence .== "KWKLFKKILKFLHLAKKF",
 ↪  :hdv]..., modactive_hdv))
println(hamming(data[data.sequence .== "KWKLFKKILKFLHLAKKF",
 ↪  :hdv]..., notactive_exp_hdv))
println(hamming(data[data.sequence .== "KWKLFKKILKFLHLAKKF",
 ↪  :hdv]..., notactive_virt_hdv))

using MultivariateStats
```

```
matrix_all = permutedims(hcat(data.hdv...))
S = fit(PCA, matrix_all; maxoutdim=2)
projS = projection(S)

indices = [[i for i in 1:nrow(data) if data.class_num[i] == j]
 ↪  for j in 1:4]

using Plots
function plotter()
    colors = [:green, :black, :blue, :red]
    labels = ["active", "mod. active", "inactive (exp)",
 ↪  "inactive (virt)"]
    fig = Plots.plot()

    for i in 1:4
        scatter!(fig, (projS[indices[i], 1], projS[indices[i],
 ↪  2]), label=labels[i], mc = colors[i])
    end
    return fig
end
plotter()
```

B.3.4 Classifier

```
n = nrow(data)

train = rand(n) .< 0.8
test = train = .! train

train_df = data[[i for i in 1:n if train[i] == 1], :]
test_df = data[[i for i in 1:n if test[i] == 1], :]

active_hdv_t = bitadd(hcat([i for i in
 ↪  train_df[train_df.class_num .== 1, :hdv]])...)
modactive_hdv_t = bitadd(hcat([i for i in
 ↪  train_df[train_df.class_num .== 2, :hdv]])...)
notactive_exp_hdv_t = bitadd(hcat([i for i in
 ↪  train_df[train_df.class_num .== 3, :hdv]])...)
```

```
notactive_virt_hdv_t = bitadd(hcat([i for i in
 ↪  train_df[train_df.class_num .== 4, :hdv]])...)

function predict(seq)
    y = [hamming(active_hdv_t, seq), hamming(modactive_hdv_t,
 ↪  seq), hamming(notactive_exp_hdv_t, seq),
 ↪  hamming(notactive_virt_hdv_t, seq)]
    return findmin(y)[2]
end

using StatsBase
pred = [predict(i) for i in test_df.hdv]
mean(test_df.class_num .== pred)
```