CS6024: Mini-Project Report

Faster ML Techniques for template-based protein structure prediction

Dhanush Kalva and Karan Bardhan

IIT Madras cs20b034@smail.iitm.ac.in, cs20b036@smail.iitm.ac.in

1 Introduction

1.1 Background

Determining the 3D structure of proteins is an important field of study in biochemistry, molecular biology and biophysics. The 3D structure of a protein determines its interactions with other molecules and, thus, its role in various biological processes. Some prime uses of knowing the structure of proteins are in drug design to enhance the creation of drugs targeting specific regions of proteins, understanding enzyme catalysis and how it can be modified to change catalytic activity and in protein engineering to improve properties of proteins and create new functions.

The current methods used to determine 3D structures of proteins include X-ray crystallography, NMR spectroscopy, and cryo-electron microscopy. However, determining the 3D structure of a protein experimentally is a difficult and time-consuming process. As a result, protein structure prediction methods are rapidly becoming more and more accurate, cost-effective and efficient.

One of the theoretical ways to predict the 3D structure of a protein is template-based (homology) modelling. In this method, we try to map the given query protein to various template (target) proteins whose structures are already known. The similarity check is compared by sequence alignment of the amino acid sequence, and high similarity indicates better chances of common protein structures. Template-based methods such as these are practical because they allow researchers to study a protein's structure and function even when experimental modelling methods are not feasible.

Rapid advances in this field have led to the rise of machine-learning and neural network methods to improve protein structures further. This not only lets less common proteins be aligned, but also reduces human bias (creeps in via substitution score matrix during sequence alignment) and provides more realistic structures. With increasingly complex models and innovative ML methods being discovered, and competitions such as CASP (Critical Assessment of Structure Prediction) growing, protein structure prediction (both template-based and non-template-based) is becoming easier and more accessible.

1.2 Motivation for Project

Improving the speed and accuracy of protein structure prediction using template-based modelling from its amino acid sequence requires suitable homology-detecting algorithms and better sequence alignment techniques once a template is detected. Current template-based methods use the same sequence alignment generated while searching for homologs, and do not realign the query and homolog proteins once a match is found. This causes a less optimal structure generated, since alignments in the current methods focus more on finding homologs.

This problem is easily addressed by realigning the two protein sequences once a homolog is found. One of the ways to do this is by using ML, such as KNN and FLANN techniques used in this paper [1]. However, these methods are by nature extremely slow, and finding faster ML techniques for realignment while keeping accuracy at a similar level is the need of the hour, which is what we aim to do.

2 Literature Survey

2.1 Datasets

The datasets used in the paper were taken from the Structural Classification of Proteins (SCOP) database. The SCOP database consists of redundant sequences as the classification uses manually curated functions. The data is classified by class, folds, superfamily, family, and domain. So, to combat redundant sequences, the paper used the SCOP40 database. This subset of data is helpful for the ML model to avoid overfitting, and it also cuts down on computation time and costs. The paper treats proteins within the same superfamily to be structurally similar.

2.2 Alignment Calculation and parameter optimisation

The paper computes the sequence alignment using the Smith-Waterman algorithm. The algorithm needs a substitution score for each residue pair. To calculate this score, the paper used an ML algorithm, KNN (which was then improved to FLANN). The training model that we feed into the model is created by using the similarities of proteins within the same group. For such proteins, there is a high frequency of a few amino acids at each position in the protein. Using this information, they create Position Specific Score Matrices (PSSMs) for each protein. PSSM for a protein is a 20 x window size matrix containing probabilities of each amino acid being at that position, considering a window around that position in other proteins.

The paper makes the training and testing datasets by the following process.
For a **query protein Q** with size n and a **target protein T** with size m, the training set will have dimensions [(n*m) samples , (2*w*20) features].
For a label corresponding to Q_x and T_y is 0 if $Q_x \neq T_y$ and 1 if $Q_x = T_y$.
The values of the vector $V_{x,y}$ is the probability of similarity in the given window which is calculated using the PSSM matrices of Q and T.
For testing, the paper takes a query Q and a template T', and builds a similar vector. We use an ML method to calculate the probabilities of Q_x and T'_y to be the same.
For each step of alignment generation, we go to the block with the highest probability of being the same.
This gives us the best sequence realignment for the given query and template protein within the superfamily.

Once the approach to alignment is straightforward, the next step is to find the suitable parameters for the number of neighbours and gap penalties (both open and extend). Using the standard gap penalties does not work since the substitution scores generated by this method were much lower in magnitude. For this, the paper uses the validation dataset to choose from a manual range of options, and finally settled on 1000 neighbours, -0.1 penalty for gap-open and -0.0001 penalty for gap-extend.

2.3 Pros and Cons

Pros:

- The paper has high accuracy in 3D modelling the protein given a query and matching templates, almost on par with purely structural alignment techniques.
- It allows the combination of the best homology detection methods and structural alignment techniques by identifying the possibility of re-aligning the query sequence with the template sequence before 3D modelling. This modularity allows for improvements in both steps separately without worrying about the effect of the overall picture.

  Cons:

- The time taken by KNN and FLANN increased exponentially as the dataset grew in size. This caused a reduction in training dataset size to 1/10th of its original, which is a huge decrease. Even after that reduction, building up the initial model takes a high runtime (5 hours+).
- The paper requires query and template proteins to be given, and is poor at detecting homologs itself.

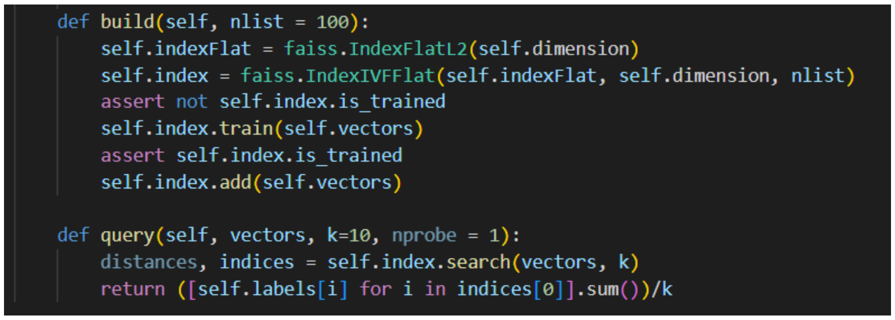
3 Materials and Methods

3.1 FAISS [2]

The idea behind Faiss is to separate the data into clusters and store the cluster centroids in the inverted file (or index). The distance to the centroids is calculated efficiently using the L2 distance. Faiss is designed to handle high-dimensional vector spaces efficiently. It employs techniques such as product quantization and residual quantization to reduce memory usage and accelerate search in high-dimensional spaces; this is a great feature as we are dealing with a significantly large dataset. Apart from that since we split the datasets into multiple clusters, we can select only a few clusters to calculate the nearest neighbours to the given data point. We can be confident that we will accurately calculate the nearest neighbours by choosing only the nearest centroids to the data point. This reduces computation time greatly. Another factor for time reduction is that we can answer multiple queries simultaneously.
FAISS also has many features that aren't being used in this project. The reason is that we wanted to compare the performance of FAISS to FLANN keeping conditions as similar as possible, neglecting performance enhancers such as GPU usage (FAISS), index building (FLANN), Dataset size variance, etc. Some of the impressive features that faiss has that we would love to include in future works are:

- Faiss leverages the power of GPUs to accelerate search operations. It provides GPU-enabled index structures, allowing for fast similarity search on large datasets.
- Faiss integrates with popular deep learning frameworks such as PyTorch and TensorFlow, making it easier to incorporate similarity search capabilities into your deep learning pipelines.

Below is the code that we used to implement FAISS:



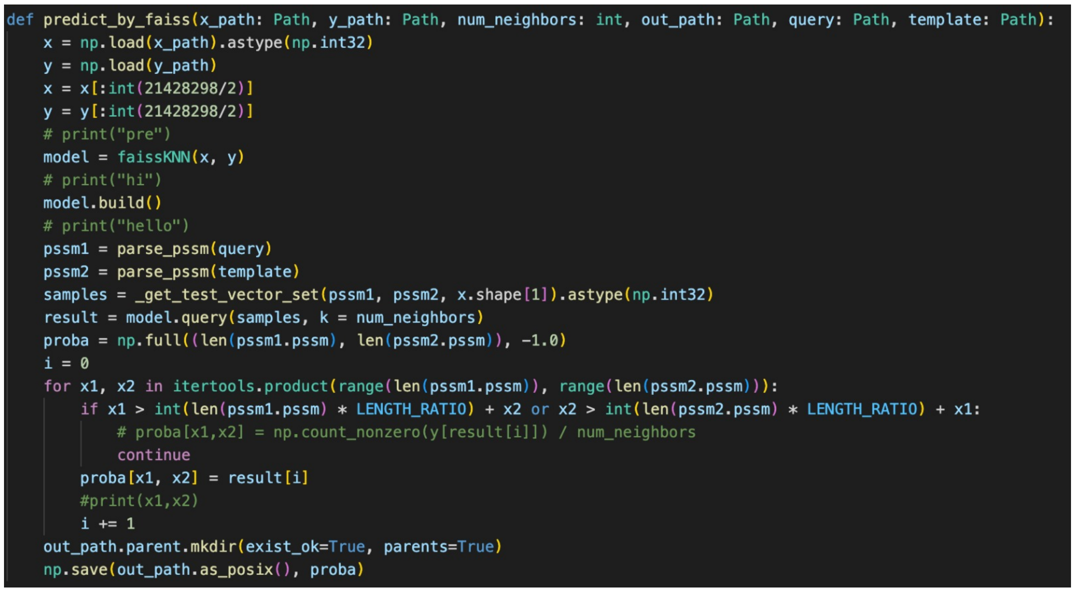
```
def build(self, nlist = 100):
    self.indexFlat = faiss.IndexFlatL2(self.dimension)
    self.index = faiss.IndexIVFFlat(self.indexFlat, self.dimension, nlist)
    assert not self.index.is_trained
    self.index.train(self.vectors)
    assert self.index.is_trained
    self.index.add(self.vectors)

def query(self, vectors, k=10, nprobe = 1):
    distances, indices = self.index.search(vectors, k)
    return ([self.labels[i] for i in indices[0]].sum())/k
```

Fig. 1. The implementation of FAISS

Let's look at the implementation step-by-step:

– IndexFlat: Provides the base indexing for the data, this is basically an optimised data structure used to find the nearest neighbours in efficient time.
– IndexIVFFlat: This is known as the quantiser. This helps divide the index formed earlier into clusters. nlist here denotes the number of clusters to be formed.
– The rest of the code is self-explanatory. nprobe denotes the number of clusters to be searched. In the final model, we used nlist = 100 and nprobe = 1

And predicting the alignment is done by the following code:



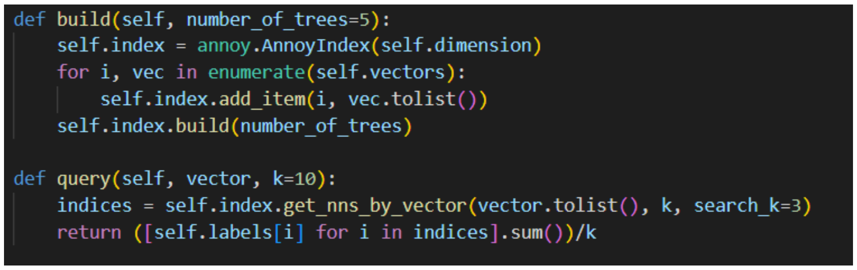
```
def predict_by_faiss(x_path: Path, y_path: Path, num_neighbors: int, out_path: Path, query: Path, template: Path):
    x = np.load(x_path).astype(np.int32)
    y = np.load(y_path)
    x = x[:int(21428298/2)]
    y = y[:int(21428298/2)]
    # print("pre")
    model = faissKNN(x, y)
    # print("hi")
    model.build()
    # print("hello")
    pssm1 = parse_pssm(query)
    pssm2 = parse_pssm(template)
    samples = _get_test_vector_set(pssm1, pssm2, x.shape[1]).astype(np.int32)
    result = model.query(samples, k = num_neighbors)
    proba = np.full((len(pssm1.pssm), len(pssm2.pssm)), -1.0)
    i = 0
    for x1, x2 in itertools.product(range(len(pssm1.pssm)), range(len(pssm2.pssm))):
        if x1 > int(len(pssm1.pssm) * LENGTH_RATIO) + x2 or x2 > int(len(pssm2.pssm) * LENGTH_RATIO) + x1:
            # proba[x1,x2] = np.count_nonzero(y[result[i]]) / num_neighbors
            continue
        proba[x1, x2] = result[i]
        #print(x1,x2)
        i += 1
    out_path.parent.mkdir(exist_ok=True, parents=True)
    np.save(out_path.as_posix(), proba)
```

Fig. 2. Prediction with FAISS model

3.2 ANNOY [3]

Annoy focuses on providing fast approximate nearest neighbour search. It uses a combination of random projection trees and indexing techniques to build a tree-based data structure that enables an efficient search for nearest neighbours. In Annoy, in order to construct the index, we create a forest (aka many trees). Each tree is constructed in the following way, we pick two points at random and split the space into two by their hyperplane, we keep splitting into the subspaces recursively until the points associated with a node are small enough. In order to search the constructed index, the forest is traversed to obtain a set of candidate points from which the closest to the query point is returned. We mainly have two parameters that govern the performance of ANNOY-search i.e., the number of trees and search_k. Both values are proportional to accuracy as well as time. Performance is discussed in the results section.

Below is the code that we used to implement ANNOY:



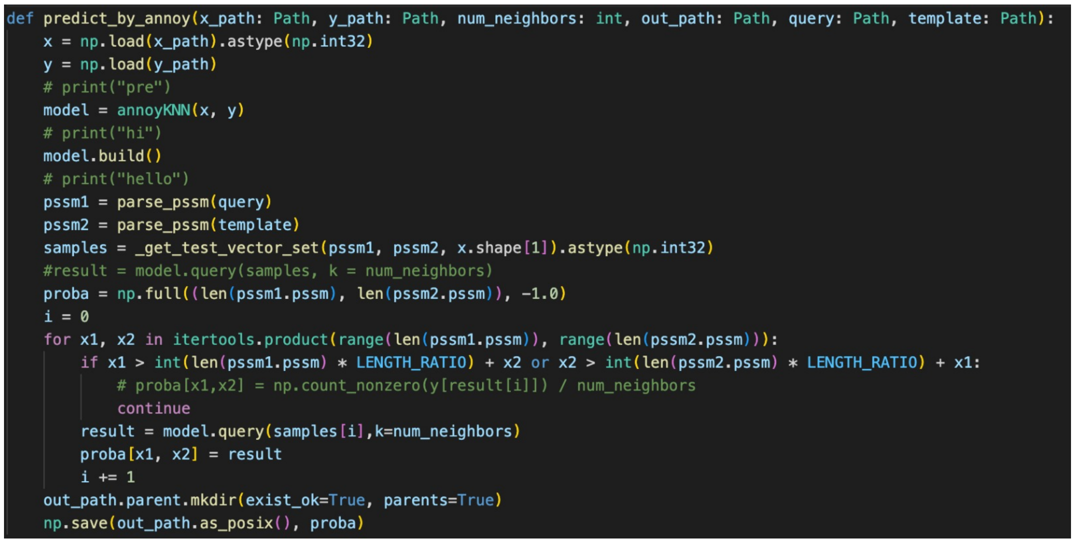
```python
def build(self, number_of_trees=5):
    self.index = annoy.AnnoyIndex(self.dimension)
    for i, vec in enumerate(self.vectors):
        self.index.add_item(i, vec.tolist())
    self.index.build(number_of_trees)

def query(self, vector, k=10):
    indices = self.index.get_nns_by_vector(vector.tolist(), k, search_k=3)
    return ([self.labels[i] for i in indices].sum())/k
```

Fig. 3. The implementation of ANNOY

Let's look at the implementation step-by-step:

- AnnoyIndex: The index will be built using the method discussed above.
- Adding the dataset to index: We now add our data points one by one into the model
- We then build random trees as discussed above. We build 5(number_of_trees) trees in this case.
- We then search in 3 (search_k) of the 5 trees built and report the k nearest neighbours.

And predicting the alignment is done by the following code:



```python
def predict_by_annoy(x_path: Path, y_path: Path, num_neighbors: int, out_path: Path, query: Path, template: Path):
    x = np.load(x_path).astype(np.int32)
    y = np.load(y_path)
    # print("pre")
    model = annoyKNN(x, y)
    # print("hi")
    model.build()
    # print("hello")
    pssm1 = parse_pssm(query)
    pssm2 = parse_pssm(template)
    samples = _get_test_vector_set(pssm1, pssm2, x.shape[1]).astype(np.int32)
    #result = model.query(samples, k = num_neighbors)
    proba = np.full((len(pssm1.pssm), len(pssm2.pssm)), -1.0)
    i = 0
    for x1, x2 in itertools.product(range(len(pssm1.pssm)), range(len(pssm2.pssm))):
        if x1 > int(len(pssm1.pssm) * LENGTH_RATIO) + x2 or x2 > int(len(pssm2.pssm) * LENGTH_RATIO) + x1:
            # proba[x1,x2] = np.count_nonzero(y[result[i]]) / num_neighbors
            continue
        result = model.query(samples[i],k=num_neighbors)
        proba[x1, x2] = result
        i += 1
    out_path.parent.mkdir(exist_ok=True, parents=True)
    np.save(out_path.as_posix(), proba)
```

Fig. 4. Prediction with ANNOY model

4 Results

We ran the small datasets on all three models and compared their runtimes, we also divided the medium-sized dataset into 4 parts and 2 parts and ran each algorithm on the sub-dataset. This is done because the laptop we used to run the tool did not have sufficient computational capacity to run the Nearest Neighbor algorithms on such a large dataset. In the table below we see the runtimes required for each algorithm on the specified datasets.

dataset size	FLANN	ANNOY	FAISS
small	21.3 sec	30.2 sec	**13.2 sec**
medium/4	3 min 58.9 sec	-	**2 min 9.4 sec**
medium/2	-	-	**11 min 38.6 sec**

The missing values here indicate that the algorithm couldn't complete execution after around 2 hours after which the execution was terminated. It is useful to note that FLANN provides an advantage that we can store the index after completion of one iteration of execution however it can be seen that for smaller datasets, FAISS has a competitive runtime in comparison to FLANN's pre-built index. An example is shown below.

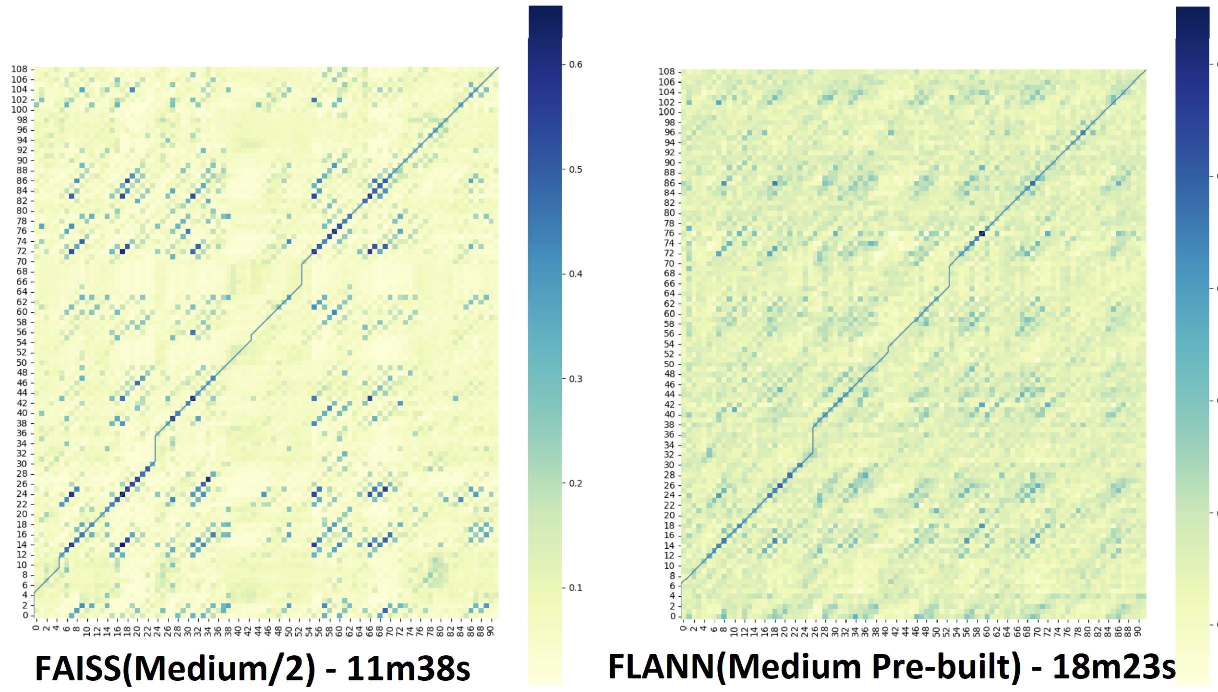


Fig. 5. X and Y axes represent the actual sequence and aligned sequence respectively. We see here that FAISS, including the index-building step, performs better than the pre-built index of FLANN. It can be argued that the size of the dataset causes this however we can approximately estimate that FAISS would take around 35 mins for the medium dataset, which is a factor of 2. This factor reduces as the size of the dataset decreases

Another comparison of FAISS vs pre-built FLANN is the small dataset where as mentioned previously FAISS takes 13.2 seconds, whereas FLANN takes 11.9 seconds on the pre-built index.

Also as mentioned previously, we also compare ANNOY runtimes using different values for the number of trees and search_k. Below is the table that compares the runtimes for ANNOY on the small dataset.

Number of trees	search_k	runtime
5	1	30.2 seconds
5	2	30.8 seconds
10	1	47.7 seconds
20	1	1 min 11 seconds
20	4	1 min 9.4 seconds

The time difference is not the only metric used to compare our models to the existing FLANN, we also measured the accuracy obtained through our models. The downstream analysis in the tool was difficult to conduct due to missing modules, so we measured accuracy by comparing the alignment obtained from our models to the alignment obtained by using FLANN.

dataset size	ANNOY	FAISS
small	**92.59%**	88.88%
medium/4	-	**90.74%**
medium	-	**94.44%**

The metric shown as a percentage is the number of same alignments of amino acids in the sequences generated by FLANN (used in the paper) and the model in question.
The missing values here indicate that the algorithm wasn't able to complete execution after around 2 hours after which the execution was terminated. We see that in the one case where ANNOY does terminate, it is more accurate than FAISS, this can be attributed to the similarity between FLANN and ANNOY.

5 Conclusions

We draw the following observations from the experiments that we conducted:

– FAISS is typically faster than FLANN and also uses lesser space to build the index. Hence it is an ideal model to use for conducting new experiments on less sophisticated computers. If we wish to conduct the same experiment multiple times, we could use a pre-built index from FLANN
– ANNOY is typically slower than FLANN and FAISS however it produces more similar alignments to FLANN than FAISS does.
– The accuracy of alignments increases as the size of the dataset increases, this implies that larger datasets are less sensitive to changes in the model, hence it is better to choose faster models which in this case is FAISS if we want to conduct experiments on larger datasets.

5.1 Future Work Possible

– Include GPU optimisations in FAISS to decrease the runtime further
– Implement CNN-esque algorithms to increase the accuracy of the predicted models

References

1. Shuichiro Makigaki and Takashi Ishida. Sequence alignment using machine learning for accurate template-based protein structure prediction. Bioinformatics, 36(1):104–111, 06 2019.
2. Faiss module. `https://faiss.ai/`.
3. Annoy module. `https://github.com/spotify/annoy`.