Pusan National University
Computer Science and Engineering
Technical Report 2022-09

신약 개발 후보물질 추천을 위한 머신러닝 기법 설계

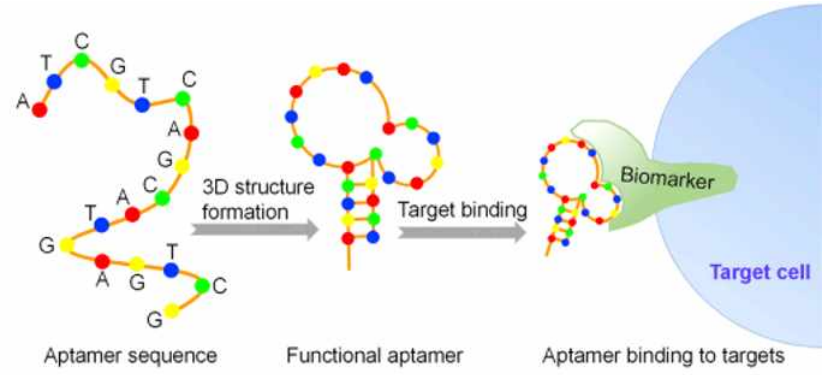
저자 1 201824633 김유진
저자 2 201724512 유경민
저자 3 201724617 허수민

지도교수 송길태

목 차

1. 서론

1.1 연구 배경

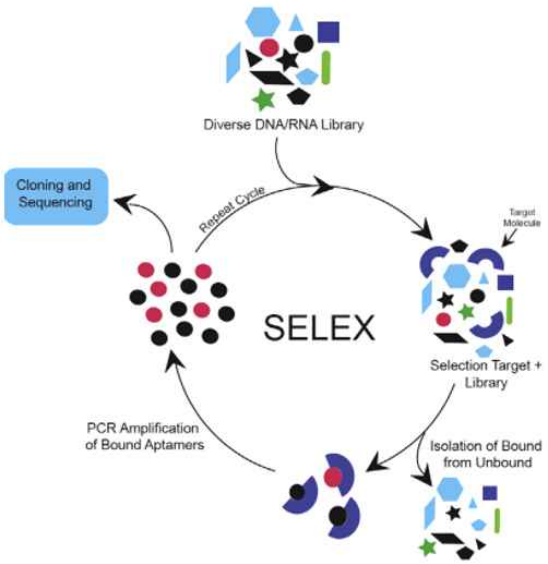


[그림 1] Aptamer의 형태와 실제 생물학적 결합 과정[1]

Aptamer(이하 압타머)는 RNA/DNA와 유사한 구조를 가지는 핵산 물질로, 표적 단백질에 다양한 형태로 결합하기에 암 치료제 분야에서 사용되고 있다. 압타머는 항체와 비교될 수 있을 정도의 높은 결합 친화도를 가지기 때문에 항체의 대체 물질로서 각광받고 있다. [그림 1]과 같이 일련의 염기서열로 이루어진 압타머 시퀀스가 있으면, 이를 구조적으로 변형하여 온전한 압타머를 생성하고, 이후 표적 단백질과 결합하여 특정 성분을 전달하게 된다.

기존에 생물학적으로 유전 물질을 합성하고 분리하여 선택적으로 결합 여부를 파악하는 방식을 넘어서 현대에는 컴퓨터를 활용해서 결합 여부를 파악하는 시뮬레이션 방식도 진행되고 있다. 본 과제 역시, 인공신경망에 기반한 딥러닝 모델을 활용해서 효과적인 결합 여부 예측 모델을 제작하였다.

1.2 기존 문제점



[그림 2] SELEX 기법의 전체 프로세스[2]

다양한 후보 압타머들 중에 가장 결합 친화도가 높은 것을 선택하기 위해서 제일 많이 사용되는 방법은 SELEX이다. SELEX는 유용한 유전 물질을 선택하는 과정을 [그림 2]와 같은 단계를 거쳐 반복 시뮬레이션을 통해 얻어내는 방식이다. 그러나 이 방법은 실제로 유전 물질을 합성 & 분리하기에 많은 노력과 시간이 필요하다. 따라서, 이를 단축시키기 위한 컴퓨터 시뮬레이션 시스템을 이용한 사례가 등장하였다. 가령, 다층 퍼셉트론(MLP)을 활용한 방식[3]이나, 합성곱 신경망(CNN)을 이용한 모델[4], 혹은 트리 기반 랜덤 포레스트(RF) 모델을 사용해서 예측을 수행하는 경우[5]가 있다.

본 과제에서는 그중에서도 MLP, CNN 등의 인공신경망을 적용한 예측 모델을 생성하되, 기존에 개발된 모델과 성능을 비교하여 모델 최적화를 수행할 예정이다. 또한, 모델 최적화를 수행하는 과정에서 더 나은 성능의 예측 모델을 만들 수 있도록 적절한 파생변수를 생성하고 샘플링을 수행하려 한다.

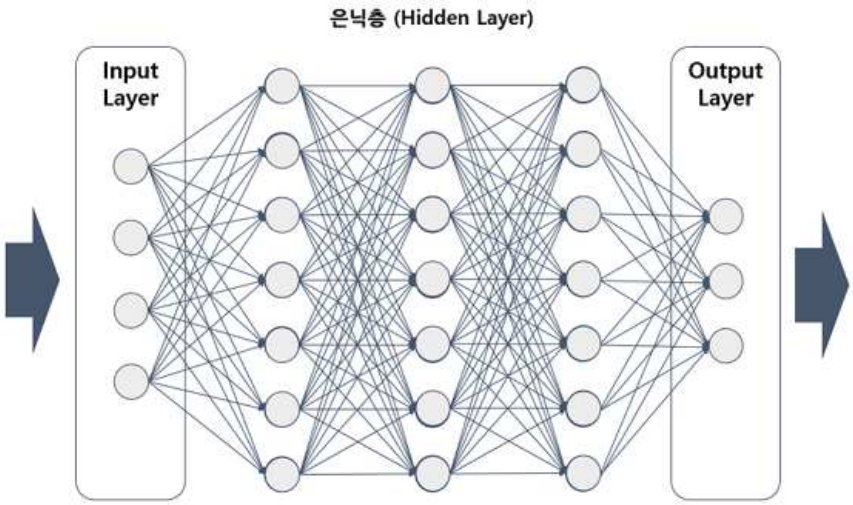
1.3 연구 목표

본 연구에서는 인공신경망을 이용하여 표적 단백질에 대해 결합 친화도가 높은 압타머를 생성하는 예측 모델을 구현할 계획이다. 이때, 예측 모델을 제작 및 학습을 수행하며 기존 사례 모델들과 성능을 비교할 것이다.

- 일반적인 형태의 결합 데이터에서도 효과적인 판별을 수행하도록 API 데이터의 전처리 수행
    - 결합 여부에 영향을 끼칠 수 있는 생물적인 정보 수집
    - 생물학적 정보를 데이터로 변환하여 결합 예측에 사용
- DeepBind, AptaNet 등의 기존 모델과 비교하여 성능 분석 및 모델 개선
    - 기존 모델을 토대로 구조 변형
    - 기존에 생성된 모델 데이터를 활용한 검증 및 성능 비교
- 인공신경망을 활용하여 주어진 단백질과 염기서열이 있을 때 결합 여부를 판별하는 모델 제작
    - MLP 기반 모델을 활용한 학습 모델 생성 및 비교 검증

2. 연구 배경

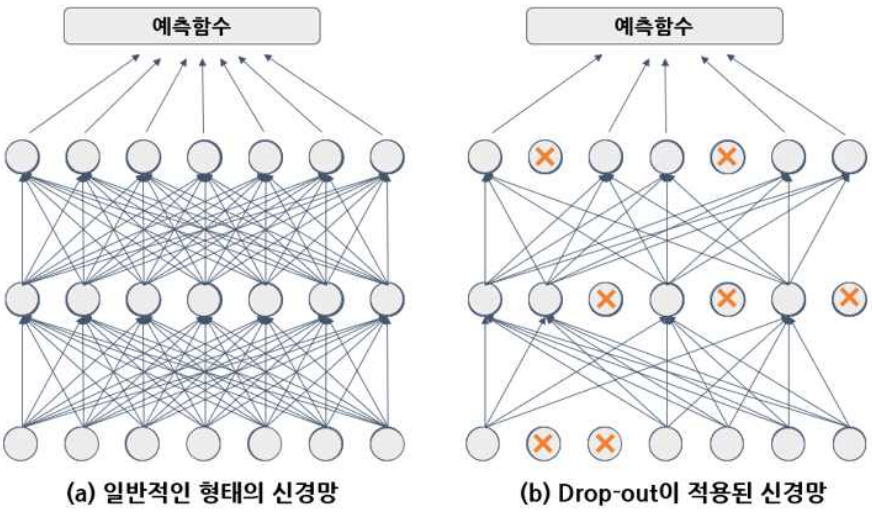
2.1 Multi Layer Perceptron & Drop-out



[그림 3] 다층 퍼셉트론의 개형

MLP는 [그림 3]의 형태처럼 단일 퍼셉트론들이 2층 이상 중첩된 fully connected된 구조로, 속성값이 들어가는 Input Layer와 예측 결과를 출력하는 Output Layer, 그리고 실질적인 연산 과정이 이루어지는 Hidden Layer가 존재한다. 각 노드간의 연결 정보에는 개별적인 가중치들이 존재하며, hidden unit의 수는 보통 실험적으로 조절해가며 적정 개수를 파악하게 된다.

은닉층이 단일 형태로 된 3층 구조 MLP를 활용했을 때 Hidden layer의 unit 수가 매우 충분하다면, 어떠한 함수의 형태건 학습을 통해 임의의 함수로 근사시킬 수 있음은 자명하다.[6] 그러나, 단순히 hidden unit의 수만 증가하면 중간 unit의 수가 너무 많아지고 학습 시간도 과도하게 소요된다. 따라서 본 과제에선 일정 개수의 노드로 이루어진 층을 여러 겹 쌓아 모델 학습 및 예측을 진행하려 한다.



[그림 4] 일반적인 MLP와 Drop-out이 적용된 MLP의 개형

[그림 4(a)]와 같이 MLP를 활용하는 모델은 Tree 모델처럼 Train data에만 과도하게

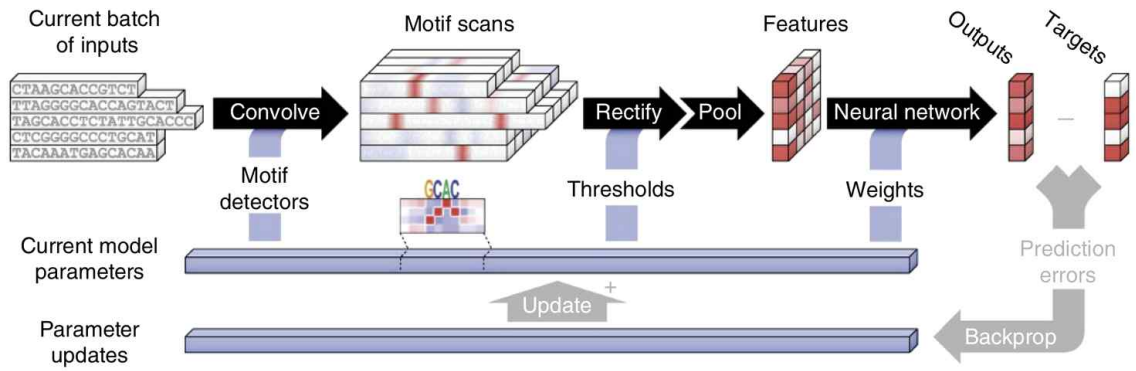
적합되어 Test data를 예측하기 힘들어지는 Overfitting 현상이 발생한다. 따라서 결정 트리에서 가지치기(Pruning)를 수행하듯, 신경망에서도 확률적으로 각 노드를 학습하지 않는 방식을 사용하게 된다. 이러한 과정을 Drop-out이라 한다.

Drop-out를 각 레이어 중간중간에 적용하면, [그림 4(b)]처럼 무작위로 노드가 끊어지고, 해당 노드는 학습 과정에서 가중치가 변화하지 않는다. 따라서 특정 가중치에만 치우치지 않고 학습할 수 있으며, 그로 인해 일반화되는 장점이 생긴다. 보통 20% ~ 50% 정도로 뉴런을 끊으며, 너무 적은 비율을 사용하면 과대 적합이, 너무 많은 비율을 사용하면 과소 적합이 발생하므로 적절히 비율을 조절해야 한다.

2.2 DeepBind & Aptanet

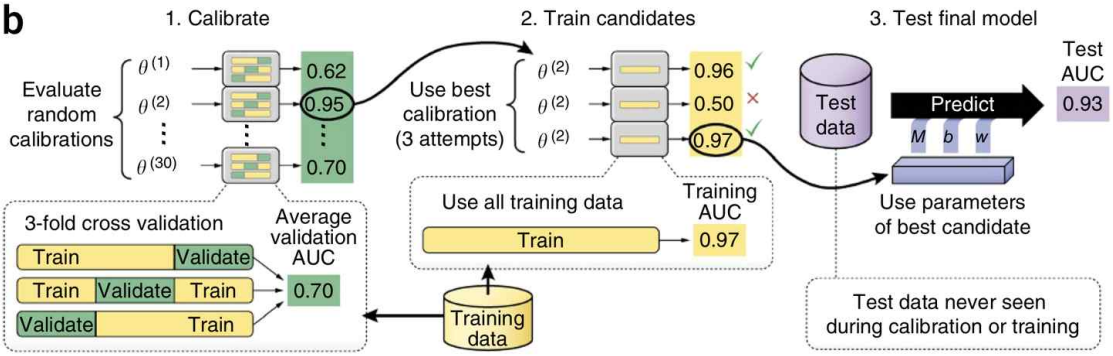
2.2.1 DeepBind

DeepBind란, Deep CNN에 기반을 두고 새로운 패턴을 추출하는 모델이다. 길이, 형태가 다른 모든 데이터에 적용할 수 있고 GPU를 이용해 수만 개의 시퀀스들을 학습할 수 있다. 또한, 레이블이 잘못 지정된 데이터나 노이즈가 포함된 데이터에도 적용하여 원활한 학습을 수행할 수 있기에, 향상된 성능의 신약 개발 후보물질을 추천 모델을 생성하기 위해 본 구조를 파악하려 한다.



[그림 5] DeepBind 모델 구조[4]

DeepBind 모델의 구조는 [그림 5]와 같은 형태로 구성되어 있다. 다양한 형태로 들어온 데이터를 one-hot encoding 방식으로 인코딩해 convolve를 수행하고, ReLU 레이어와 길이에 따라 Average-pooling 또는 Max-pooling을 적용해 신경망을 구축하는 방식을 이용한다. Hidden 레이어의 유무에 따라 weight와 bias를 달리해 학습한 뒤, back-propagation을 통해 변수를 업데이트한다. 위의 단계를 지속적으로 반복해 학습을 진행한다.



[그림 6] DeepBind 모델의 평가[4]

이후 모델 성능 측정 과정은 [그림 6]과 같은 과정을 거친다. 앞선 과정을 통해 구축된 모델을 기반으로 API 결합 여부 예측 정도를 테스트한다. 다양한 방식 중, 3-fold-cross validation을 이용해 새로운 후보를 생성해 모델을 이용해 점수를 매겨, 이 중 가장 높은 후보를 추천해주는 방식으로 진행한다.

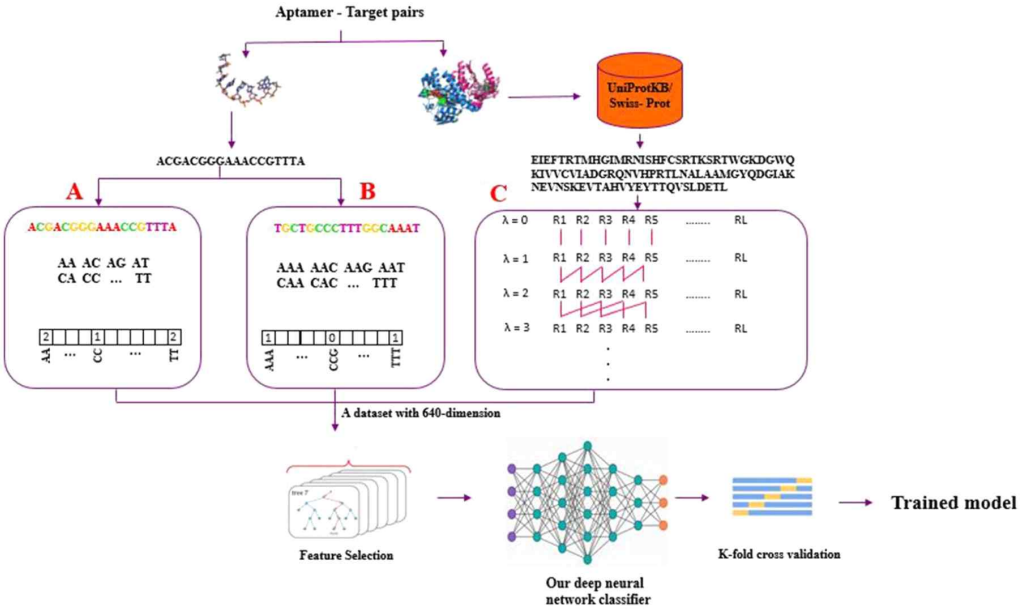
```
model = [
        motif_scan(num_motifs=16, motif_len=24,
        weight_decay=loguniform(1e-10, 1e-3),
        init_scale=loguniform(1e-7, 1e-3),),
        bias(),
        rectify(),
        maxpool(),
        full(num_units=32, weight_decay=loguniform(1e-10, 1e-3),
        init_scale=loguniform(1e-5, 1e-2),
        rectify(),
        dropout(expected_value=choice([0.5, 0.75, 1.0]),
        full(num_units=32, weight_decay=loguniform(1e-10, 1e-3),
        init_scale=loguniform(1e-5, 1e-2),
        bias(init_bias=-4.0),
]
```

[코드 1] DeepBind 모델의 pipeline

대략적인 DeepBind 모델의 pipeline은 [코드 1]과 같은 형태로 구축되어 있다. 앞선 [그림 5]의 구조를 실제 코드로 구현한 형태라 보면 된다. 본 연구에선 기존의 DeepBind 모델 구조를 이용, 딥러닝 모델을 구축 및 조정하여 기존 대비 높은 성능의 신약후보 물질을 추천해주는 모델을 만들어나가려 한다.

2.2.2 AptaNet

AptaNet이란, MLP를 기반으로 Aptamer-protein 데이터를 각 Sequece의 확률로 전처리 후 MLP를 통해 단백질 결합 여부를 예측하는 모델이다. 전체적인 모델 구조와 학습 과정은 [그림 7]에 나와 있다.



[그림 7] AptaNet 모델의 구조 및 학습 과정[3]

해당 모델의 경우, 불균형 형태의 API 데이터를 기반으로 학습을 진행하였다. 이러한 문제를 해결하기 위해서 데이터의 수를 조정하는 샘플링 기법을 수행하게 된다. 샘플링 기법에는 언더샘플링(Under-Sampling)과 오버샘플링(Over-Sampling)이 존재한다. 언더샘플링은 다수 클래스의 데이터를 비율에 맞춰 일부만 쓰는 방식이며, 오버샘플링은 반대로 소수 클래스의 수를 증배하는 방식이다. 그중, 본 모델에선 언더샘플링을 기반으로 모델 학습이 진행된다.

Neighborhood Cleaning algorithm:

1. Split data T into the class of interest C and the rest of data O.
2. Identify noisy data A_1 in O with edited nearest neighbor.
3. For each class C_i in O:

   If ($x \in C_i$ in 3-nearset neighbors of misclassified $y \in C$) and ($|C_i| > 0.5 \cdot |C|$)

   Then $A_2 = \{x\} \cup A_2$.
4. Reduced data $S = T - (A_1 \cup A_2)$.

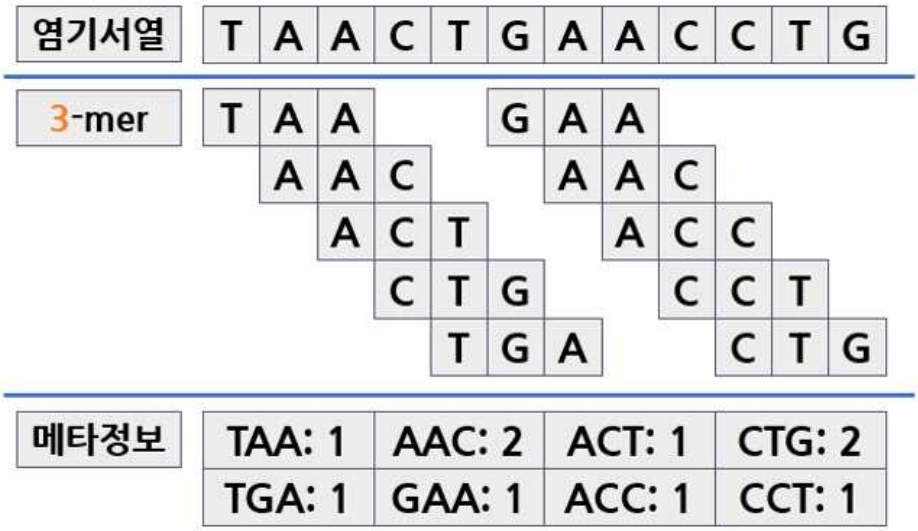
[그림 8] Neighborhood Cleaning 알고리즘의 동작 과정 의사코드(Pseudo-code)[3]

AptaNet에선, 언더샘플링의 대표적인 방식 중 하나로 NCL(Neighbourhood Cleaning Rule)을 이용하고 있다. NCL은 CNN(Condensed Nearest Neighbours)과 ENN(Edited Nearest Neighbours)을 결합하여 만든 알고리즘이다. 전체적인 동작은 [그림 8]에 나와 있다. CNN은 지도학습의 대표 모델 중 하나인 k-NN(k-Nearest Neighbor)에서 k=1인 1-NN 모형으로 분리되지 않는 데이터만을 남겨두는 알고리즘

이다. 그리고 ENN은 다수 클래스를 기준으로, 근접된 k개 데이터가 전체 혹은 일정 비율 이상의 다수 클래스인 경우가 아니면 샘플을 삭제하는 방식에 해당한다. 참고로 k-NN에서 마지막 N은 Neighbor, CNN과 ENN의 마지막 N은 Neighbours, 그리고 NCL에서 앞글자 N은 Neighbourhood에 해당한다.

이후 API 데이터에서 전처리를 거쳐 데이터 정제 및 적절한 특징을 도출하고, 최종적으로 MLP(Multi-Layer Perceptron) 모델을 구성하여 단백질 결합 여부를 분류하게 된다. 해당 과정에서 K-fold cross validation을 적용하여 모델 성능 평가를 수행하게 된다.

2.3 k-mer 지표



[그림 9] 주어진 염기서열에서 3-mer를 적용했을 때 얻을 수 있는 정보

주어진 문자열에서 등장할 수 있는 길이 k의 문자열 집합을 의미한다. 유전학에선 DNA 염기서열을 일부분 읽어가며 얻을 수 있는 모든 부분 염기서열을 일컫는다. 가령 [그림 9]와 같은 염기서열이 주어졌다고 하자. 이때 k=3일때 k-mer를 적용하게 된다면 인접한 3개의 문자를 기준으로 정보를 추출하게 된다. 이렇게 얻어낸 정보를 토대로 분포 계산, 연결 관계 파악 등에 활용할 수 있다. 본 과제에선 One-hot 인코딩 방식과 k-mer 방식을 결합하여 인코딩을 수행하고 유의미한 파생변수를 생성하였다.

2.4 Amino Acid Composition & Pseudo Amino Acid Composition

단백질의 구조와 염기서열의 정보에 따라서 결합 여부를 판별할 수 있도록 염기서열의 인코딩뿐만 아니라 단백질 구성 형태를 인코딩하는 기법이 필요하다. 이때, 아미노산의 빈도나 결합 구조 정보를 표현하는 기법이 존재하는데, 대표적으로 AAC와 PseAAC가 있다.

AAC는 단백질 시퀀스에서 특정 아미노산이 얼마나 출현하는지를 표현한다. 즉, 단

순히 전체 개수 대비 특정 아미노산 개수의 형태로 표현한다. 그러나, 이는 주어진 아미노산의 결합 구조 정보를 무시한다는 단점이 존재한다. 따라서 결합정보를 고려하여 단백질 유형을 표현하는 PseAAC라는 기법이 2001년 Kuo-Chen Chou에 의해 등장했다.[7]

본 과제에선 특정 단백질과 염기서열의 결합정보가 있을 시, 단백질 정보를 AAC와 PseAAC를 활용하여 인코딩을 수행하고, 결합 예측을 위해 해당 정보를 사용해 모델을 제작하게 되었다.

2.5 연구 개발환경

2.5.1 TensorFlow & Keras

[그림 10] TensorFlow[8]와 Keras[9] 라이브러리 로고

딥러닝 학습 모델을 제작하기 위한 수단으로 가장 범용적인 라이브러리인 텐서플로우(TensowFlow)와 케라스(Keras)를 파이썬을 기반으로 하여 활용할 예정이다. 텐서플로우는 구글이 개발한 오픈소스 머신러닝 프레임워크이며, 케라스는 텐서플로우를 기반으로 동작하는 머신러닝 라이브러리이다. 통상적으로 빠른 프로토타입 모델을 생성할 때는 케라스에 내장된 모델만으로 학습 모델을 구성하며, 정교한 모델 제작을 할 때는 텐서플로우를 사용하게 된다. 따라서 케라스를 활용해 모델 틀을 구성하고, 세부 조정은 텐서플로우를 활용해 레이어를 적층하였다.

2.5.2 Jupyter Notebook & Google Colab

[그림 11] Jupyter Notebook 로고[10]

파이썬 코드를 Interactive하게 실행하고 가동할 수 있는 환경 중, 오픈소스 기반 웹 플랫폼인 주피터 노트북(Jupyter Notebook)이 존재한다. 특히나 데이터 분석 및 머신러닝, 딥러닝 환경에서 자주 활용되며, 데이터를 정제하고 모델 구축 및 학습, 최적화 시 효과적으로 동작 과정을 확인할 수 있다는 점에서 주로 사용된다.

본 연구에선 기본적인 코드 및 모델 학습을 주피터 노트북에서 작성하고 동작하기

로 했다. 다만, 부득이하게 모델 학습을 위해 고성능의 컴퓨팅 자원이 필요한 경우 Google colab이라는 별도의 플랫폼을 추가로 사용하였다.

[그림 12] Google colab 로고[11]

Google colab, 혹은 Google colaboratoy 환경은, 데이터 분석 및 머신러닝 모델 제작을 위해 제공하는 무료 클라우드 서비스 기반의 실행 환경으로, Jupyter Notebook과 유사한 인터페이스를 지원한다는 특징이 있다. 특히나, 내부적으로 고성능의 GPU 및 TPU를 지원하며, 리소스 한도 내에서 고성능의 컴퓨팅 자원을 활용할 수 있다는 점에서 모델 구축 및 학습 시 자주 사용하게 된다.

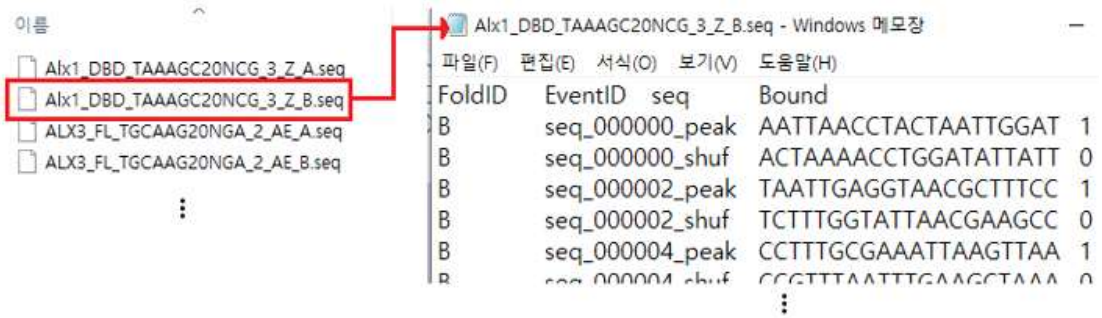
따라서 본 환경에선 고성능의 컴퓨팅 자원이 필요한 경우, Jupyter Notebook과 함께 Google colab을 이용해서 모델을 학습하는 데 활용하였다.

3. 연구 내용

3.1 데이터 전처리

3.1.1 초기 데이터 선정 및 기본 코드 작성

API 결합 데이터를 확보하기 위해 기존 선례 연구에서 얻어낸 4가지 대표 API 정보(Alx1_DBD, ALX3_FL, AR_DBD, AR_FL)를 토대로 데이터 병합 및 초기 전처리 과정을 진행하였다.



[그림 13] raw data의 형태(좌측)와 내부 기록 형태(우측)

API raw data의 경우 [그림 13]의 좌측 이미지처럼 .seq의 파일로 저장되어 있으며, 내부 데이터 형태는 [그림 13]의 우측과 같다. tab으로 각 값이 분리되어 있다. 세부적으로는 FoldID(A 그룹 or B 그룹), EventID(데이터 정보), seq(염기서열), Bound(API 결합 유무)에 해당하는 각 데이터가 나와 있다. 이때, FoldID가 그룹 A인 경우는 전부 Bound가 1로만 이뤄져 있으며, 그룹 B의 경우 1과 0이 고르게 나와 있었다. 따라서 전처리 과정에선 API의 종류가 같으면 그룹 A, B와 상관없이 하나로 병합하여 데이터

를 전처리하였다.

데이터 전처리를 위해 먼저 [그림 13]의 raw data를 분리하여 DataFrame 생성 및 csv파일 형태로 정보를 저장하는 추출 코드를 작성하였다. 주어진 .seq 파일을 전처리하여 하나의 csv 파일로 제작하는 함수(merge_data)를 제작하였고, 그 이후엔 bound 예측을 위해 필요한 속성들을 추가하는 방식으로 전처리를 진행하였다.

```
def get_base_list(rslt_lst, seq):
    n_pos = seq.find('N')

    if n_pos >=0:  # N(Any)이 있는 경우
        for base in ['A', 'C', 'G', 'T']:  # 재귀적으로 추가
            get_base_list(rslt_lst, seq[:n_pos] + base + seq[n_pos+1:])
    else:  # N이 없는 경우 정보를 list에 저장
        rslt_lst.append(seq)
```

[코드 2] 염기서열의 N 정보를 처리하기 위한 get_base_list 함수

주어진 데이터에서 염기서열을 분리하는 과정에서 'A', 'C', 'G', 'T' 외에도 'N'이라는 별도의 항목이 존재하였다. 관련 정보를 탐색한 결과, 결측치(측정 X) 내지는 와일드카드(A,C,G,T 모두 가능)라는 의미로 사용할 수 있는 문자임을 알아냈다. 따라서, N이라는 서열이 존재하는 경우 재귀적으로 A, C, G, T의 모든 문자열을 넣어서 API 정보를 복제(get_base_list, [코드 2] 참고)하는 방식으로 코드를 구성했다.

```
from itertools import product
def kmer(seq, k):
    base = ['A', 'C', 'G', 'T']

    keys = list(map(''.join, product(base, repeat=k)))
    dicts = {}
    for key in keys: dicts[key] =0

    for i in range(len(seq)-k+1):
        now = seq[i:i+k]
        dicts[now] +=1

    for key in keys: dicts[key] /= (len(seq)-k+1)
    return dicts
```

[코드 3] k-mer 정보 생성을 위한 kmer 함수

기본적으로 얻어낸 염기서열 정보에다가 k-mer 지표를 적용하기 위해 [코드 3]의 kmer 함수를 제작했고, 데이터마다 함수를 적용해 정보를 도출해내었다. seq는 A, C, G, T로 이뤄진 염기서열이며, k는 1 이상의 정숫값이 들어간다. 최종 결과는 [그림 14]처럼 절대적인 개수가 아닌 상대적인 비율(전체 합 1)의 형태로 도출된다. 이번 과제에서는 k의 값이 1~4에 해당하는 정보를 얻어내었다.

	A	B	C	D	E	F	G	H	I	J	K
1	protein	seq	bound	A	C	G	T	AA	AC	AG	AT
2	Alx1_DBD	GCAGATAATCTAATTACCCC	1	0.35	0.3	0.1	0.25	0.105263	0.052632	0.052632	0.157895
3	Alx1_DBD	CTCAGTCCTCGTCTCGATGG	1	0.1	0.35	0.25	0.3	0	0	0.052632	0.052632
4	Alx1_DBD	TCATAATCTAATTACGCTCG	1	0.3	0.25	0.1	0.35	0.105263	0.052632	0	0.157895
5	Alx1_DBD	GACTTCCTCAATCTAATTAG	1	0.3	0.25	0.1	0.35	0.105263	0.052632	0.052632	0.105263
6	Alx1_DBD	GCAGTTAATCTAATTAACCG	1	0.35	0.2	0.15	0.3	0.157895	0.052632	0.052632	0.105263
7	Alx1_DBD	TCCAGGCCTGATTTAATTAA	1	0.3	0.2	0.15	0.35	0.105263	0	0.052632	0.105263

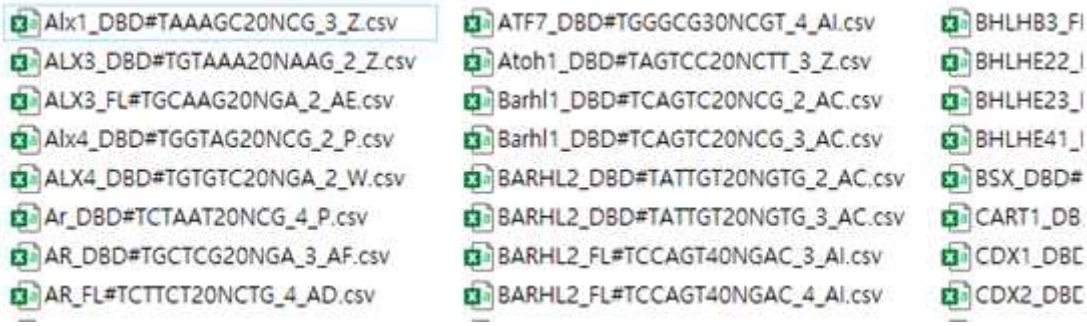
[그림 14] k-mer(k=1, …, 4) 지표를 적용하여 얻어낸 결과 샘플

단백질 정보의 경우 PseAAC 정보를 활용하려 했다. 따라서 R언어에서 코드를 별도로 작성하여 Rcpi 라이브러리를 이용해 단백질 정보를 추출하였다. 초기 데이터 추출 코드에선 단순히 4종류의 API 정보만 존재했기에, 직접 코드 내에 해당 단백질의 이름을 넣어서 protein_data.csv라는 정보를 얻어냈다. 단백질 정보는 UniProt[12]의 DB 정보를 기반으로 얻어내었다. 이후, 추가 데이터 확보로 인해서 코드의 구조를 변경하게 되었고, 최종 단백질 정보 추출 코드는 [코드 4]와 같이 작성하게 됐다.

```
# 라이브러리 불러오기
library("Rcpi")

file = "csv파일 경로 작성"
labelList = read.csv(file, header=TRUE)

rslt_df = c()
(i 1:nrow(labelList)){
        nowe = labelList[i, 1]
        nowl = labelList[i, 2]

        # 단백질 구조 추출
        seqs = getProt(nowe, from = 'uniprot', type = 'aaseq')[[1]][[1]]
        # 단백질 meta 정보 추출
        rslt = extractProtPAAC(seqs)

        trslt = t(rslt)  # 행/열 뒤집기
        rownames(trslt) = nowl  # 행 이름 지정

        rslt_df = rbind(rslt_df, trslt)
}

directory_path = '저장할 폴더 위치'
setwd(directory_path)
write.csv(rslt_df, file='protein_data.csv', quote=FALSE)
```

[코드 4] PseAAC 정보를 도출하기 위한 Rcpi 활용 R 코드

3.1.2 추가 데이터 확보 및 샘플링 코드 제작



[그림 15] 활용할 데이터셋을 분리한 결과 일부분

4종류의 API 정보를 토대로 기본적인 전처리 코드를 작성하고 추가 데이터셋을 탐색하던 도중, DeepBind 모델에서 활용했던 원본 데이터셋을 추가로 얻어낼 수 있었다. ([그림 15] 참고) 추가로 받은 데이터셋을 앞선 과정을 거쳐서 하나의 단일 데이터로 만들 생각이었다. 그러나, 데이터의 양이 방대해서 기존처럼 하나의 데이터로 합치는 방식으로 진행하지 못한다는 문제점에 직면했다. 따라서, 방식을 바꿔 각 단백질 정보마다 별도의 DataFrame 정보를 형성해서 저장하고, 학습 및 예측용 dataset을 생성하는 경우에만 별도의 데이터 생성 함수를 제작하여 데이터 샘플링 및 병합하는 방식을 쓰겠다고 결정했다. 또한, 이를 위해서 [코드 4]와 같이 단백질 정보 추출 코드의 구조도 추가로 변경하여 방대한 API 정보에서도 데이터셋을 생성할 수 있도록 했다.

entry	Q8C8B0	O95076	O95076	O35137	Q9H161	O00327	Q96QS3
label	Alx1_DBD	ALX3_DBD	ALX3_FL	Alx4_DBD	ALX4_DBD	ARNTL_DBD	Arx_DBD

[그림 16] 단백질 명칭과 UniProt의 인덱스간의 매핑 정보 일부

이때, 단백질 코드 정보를 저장해놓은 csv 파일(protein_label_data.csv)의 경우, 다음의 [그림 16]과 같은 형태에서 행/열이 전치된 형태로 정보가 기재되어있다. 본 보고서에서는 편의상 행/열을 전치하여 표현하였다. 해당 인덱스 정보는 별도로 얻어낼 수 없어서 직접 찾아가며 수기로 목록을 써넣었다.

```
In [14]: bound_df = merge_data()
         print()
         #print(bound_df.head())
         #bound_df.to_csv('raw_data.csv', index=False)
         BarhI1_DBD#TCAGTC2ONCG_3_AC clear!
         BARHL2_DBD#TATTGT2ONGTG_2_AC progressing...
         BARHL2_DBD#TATTGT2ONGTG_2_AC clear!
         BARHL2_DBD#TATTGT2ONGTG_3_AC progressing...
         BARHL2_DBD#TATTGT2ONGTG_3_AC clear!
         BARHL2_FL#TCCAGT4ONGAC_3_AI progressing...
                 => Removed: TCATCTAAGGACGATTAGTTTTAACNCNCTAGNNNNGGGA
                 => Removed: CTCCGAGAGCATAGTTTATATAGATGNNTTNCNNNNGGAC
                 => Removed: CATGAAAATGCGAAAGGAAGAAGCGANNGAGGNNNNTTGG
                 => Removed: CTCCTTTTGCAGGGCACACTCAATNANGAGACNGNNCNTN
                 => Removed: AAACGCTTTAACATACGAATGTTANANCAAGTNCNATNAN
                 => Removed: TGTAAATGAATTAACGAGAACACTNANTGCATNANTTNGN
                 => Removed: TTAANCATAATTAATAGAGNAGNCNNNCNAGTNTNNANAN
```

[그림 17] N이 과도하게 존재하는 염기서열 정보의 일부

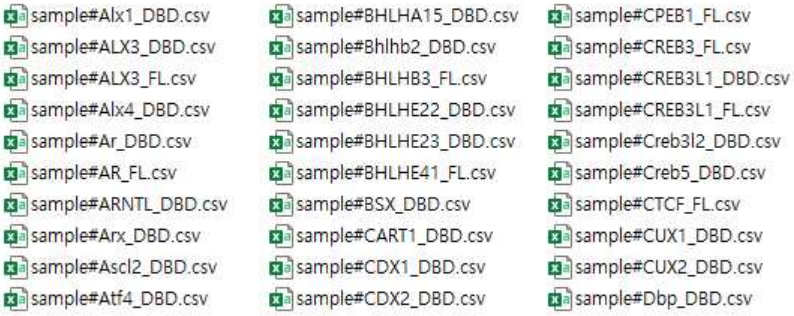
그런데, 앞선 방식으로 500종류 가량의 API 정보 내 염기서열을 처리하면서, 기존 4종류 API의 전처리 과정에서 보이지 않던 재귀함수의 메모리 초과 현상이 발생하였다. 원인을 찾아본 결과, [그림 17]과 같이 일부 염기서열에서 N 정보가 6개 이상 내지는 11개 이상 출현하는 때도 있었다. 해당 염기서열을 복제하게 될 때 4^11개, 즉 약 사백 십만 개 이상의 불필요한 데이터를 생성해야 한다는 문제점이 존재했다. 따라서, 염기서열의 중첩 복제를 허용하는 경우는 최대 N이 5개까지 존재하는 서열(약 1000개까지 복제 허용)에 관해서만 적용했고, 그 이상 N이 존재하는 염기서열은 사용하지 않았다.

이후 기존에 4종류의 API를 전처리하는 과정과 동일하게 '염기서열에서 k-mer 지표 적용 & 단백질 결합정보 추출 -> 추출된 API 정보에 결합'하는 과정을 진행하였다. 이렇게 생성한 원본 데이터의 양이 약 200GB이상 생성되었기에, 선택적으로 데이터를 추출해서 데이터셋을 생성하는 프로그램이 추가로 필요하다 생각했다.

```
import os
import numpy as np
import pandas as pd

# 각 dataframe에서 몇 퍼센트 가량 데이터 추출할지 결정
percentage =0.2
# 최소 샘플링 데이터 개수. 만일 전체 데이터가 이것보다 작으면 모든 데이터를 추출
min_sample =1000
# 최대 샘플링 데이터 개수. 최소/최대 반대로 값 작성 시 논리오류 발생할 수 있음
max_sample =10000

nowdir = os.getcwd()
datadir = nowdir +'/data/metacsv’

protein_data = pd.read_csv(nowdir +'/data/protein_data.csv')

file_list = os.listdir(datadir)

for file in file_list:
    now_protein =file.split('#')[1]
    file_name =file
    print(now_protein +' 진행 중…  ', end='')

    datafile = datadir +'/'+ file_name
    nowdf = pd.read_csv(datafile)

    nowsize =len(nowdf)
    sample_size = min(nowsize, max(min_sample,
        min(int(nowsize*percentage), max_sample)))

    one_protein = protein_data[protein_data.seq  == now_protein]

    nowdf_rand = nowdf.sample(n=sample_size)

    nowdf_cut = nowdf_rand.drop(['seq'], axis=1)
    nowdf_cut.protein = now_protein

    one_full_data = pd.merge(nowdf_cut, one_protein,
        left_on='protein', right_on='seq').drop(['protein'], axis=1)
    # 실제 파일 저장.
    one_full_data.to_csv(nowdir +'/data/finalcsv/sample#'+
        now_protein +'.csv', index =False)
    print('>>완료!<<', end='  [>  ')
    print('추출된 사이즈: '+str(sample_size))
```

[코드 5] 랜덤 샘플링 및 데이터 병합용 코드

추가로 학습을 위한 데이터셋을 생성하기 위해 원하는 비율에 맞춰서 데이터를 샘플링하고 병합하는 [코드 5]의 프로그램을 제작하였다. 원본 데이터들에서 원하는 비율 & 개수범위만큼의 데이터를 랜덤 추출하여 학습용으로 사용할 수 있도록 뽑아내는 동작을 수행한다. 최종적으로 얻어낸 데이터의 저장 형태와 내부 구조는 각각 [그림 18]과 [그림 19]에 해당한다.



[그림 18] 최종 샘플링된 데이터의 저장 형태 일부

bound	A	C	G	T	AA	...	Xc2.lambc	Xc2.lambc	Xc2.lambda.30
1	0.5	0.1	0.05	0.35	0.315789	...	0.022982	0.02506	0.023447
1	0.25	0.3	0.15	0.3	0.052632	...	0.022982	0.02506	0.023447
0	0.5	0.1	0.05	0.35	0.157895	...	0.022982	0.02506	0.023447
1	0.55	0.1	0	0.35	0.210526	...	0.022982	0.02506	0.023447
1	0.4	0.2	0.1	0.3	0.105263	...	0.022982	0.02506	0.023447
1	0.3	0.15	0.2	0.35	0.105263	...	0.022982	0.02506	0.023447
1	0.35	0.1	0.1	0.45	0.105263	...	0.022982	0.02506	0.023447
1	0.45	0.15	0.15	0.25	0.263158	...	0.022982	0.02506	0.023447
0	0.35	0.2	0.15	0.3	0.105263	...	0.022982	0.02506	0.023447
1	0.4	0.2	0.05	0.35	0.157895	...	0.022982	0.02506	0.023447
1	0.3	0.2	0.05	0.45	0.052632	...	0.022982	0.02506	0.023447
1	0.35	0.2	0.2	0.25	0.157895	...	0.022982	0.02506	0.023447
1	0.4	0.25	0.05	0.3	0.105263	...	0.022982	0.02506	0.023447
0	0.5	0.1	0.1	0.3	0.263158	...	0.022982	0.02506	0.023447
1	0.35	0.15	0.15	0.35	0.105263	...	0.022982	0.02506	0.023447

[그림 19] 최종 샘플링된 데이터의 내부 구조

3.2 모델 구축

3.2.1 평가 지표 선정

먼저, 모델 성능 평가를 위해 성능 지표를 결정하였다. 이때, 데이터의 불균형 속에서도 효과적으로 인공지능 모델의 성능을 평가할 수 있는 F1-score를 기반으로 모델 성능을 평가하게 되었다. 주요 성능 지표를 설명하기에 앞서, [표 1]의 이진 분류 기준의 오차 행렬을 기준으로 간단히 성능 지표를 설명하려 한다.

		Actual	
		Positive	**Negative**
Predicted	**Positive**	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

[표 1] 오차 행렬(Confusion Matrix)[13]

[표 1]은 이진 분류 상황에서 생성되는 오차 행렬이다. 실제값과 예측값에 따라서 맞거나 틀린 여부를 판별하기 위해 쓰는 구조이다. 이때, 단순히 개수로 평가하지 않

고, 이를 비율로 계산한 몇 가지 성능 지표가 존재하는데, 대표적으로 정밀도, 재현율, 그리고 이 둘을 결합한 F1-score가 존재한다. 정확도(Precision)는 예측이 Positive인 것 대비 실제도 Positive인 정도를, 재현율(Recall)은 실제가 Positive인 것 대비 예측도 Positive인 정도를 나타낸다. 그리고 F1-score는 정밀도와 재현율의 조화평균이다. 앞선 내용을 수식으로 전개하면 [그림 20]과 같다.

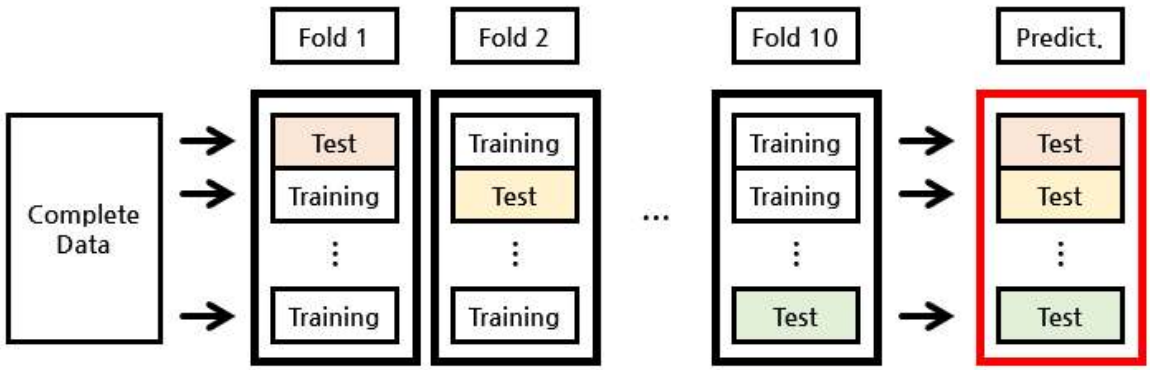
$$(Precision) = \frac{TP}{TP + FP} \qquad (Recall) = \frac{TP}{TP + FN}$$

$$(F1\text{-}score) = 2 \times \frac{1}{\frac{1}{Precision} + \frac{1}{Recall}} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

[그림 20] 주요 성능 지표 계산 방식

성능 지표로 정밀도와 재현율만 선택해서 측정할 수 있겠으나, 정밀도는 희박한 가능성을 판별하기 힘들다는 단점이, 재현율은 가능성이 매우 드문 경우도 불필요하게 예측한다는 단점이 존재한다. 따라서 이러한 Trade-off 관계를 보완할 수 있고, 데이터의 불균형 상태에서도 성능 평가가 용이한 F1-score를 기반으로 모델을 평가했다.

3.2.2 기존 모델의 성능 비교



[그림 21] K-Fold Cross Validation 과정

실험의 정확성을 위해서 향후 생성할 모델과 기존의 모델에 동일한 데이터셋으로 10-Fold 교차 검증을 실시했다. [그림 21]과 같이 총 10개의 Fold로 나눠서 매번 성능을 평가했으며, 이때 각 경우마다 F1-score를 계산했고, 10개 결과의 평균으로 최종 모델 성능을 측정했다. 이를 [코드 6]과 같이 구현한 다음 주요 모델 먼저 평가했다.

```
def testModel(x,y,num,epochs=100,batch_size=100):
    kfold = KFold(n_splits=10,shuffle=True,random_state=7)
    n_iter =0
    cv_f1_score = []

    for train_idx, test_idx in kfold.split(x):
        c = createModel(x,num)
        X_train, X_test = x[train_idx], x[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        if num ==2 or num ==3 or num ==5:
            c.fit(X_train,y_train)
        else:
            c.fit(X_train,y_train,epochs=epochs)

        fold_pred  = c.predict(X_test)
        n_iter +=1
        c_f1_score = np.round(f1_score(y_test,fold_pred>0.5),4)

        print(c_f1_score)
        print('₩n{} 교차검증 정확도 : {}, 학습 데이터 크기 : {},
                 검증 데이터 크기 : {}'.format(n_iter, f1_score,
                 X_train.shape[0], X_test.shape[0]))
        cv_f1_score.append(c_f1_score)

    print('\n average f1_score : ',np.mean(cv_f1_score))
    return np.mean(cv_f1_score)
```

[코드 6] 모델 테스트용 코드

3.2.3 모델 구축 과정

앞선 선례에 나온 모델인 DeepBind와 AptaNet에서 기본 형태가 되는 MLP 구조의 형태를 변형하고 기본 API 데이터의 설명 변수를 적절히 선정하는 방식을 혼합하여 모델 성능을 향상시키는 방식으로 실험을 진행했다.

그중에도 구조적으로 변형이 용이한 AptaNet 모델을 기준으로 하이퍼 파라미터 튜닝을 수행한 다음 Layer의 구조를 변형하였다. 변형을 통해 약 20가지의 후보 모델을 선정했고, 이들을 토대로 학습을 시켜 성능을 측정하였다. 최종적으로, 원 모델인 AptaNet보다 더 나은 성능을 보이는 두 모델을 우선 채택하게 됐다. 이후 추가적인 파라미터 튜닝과 구조 변경을 토대로 모델의 성능을 향상시켰다.

3.2.4 하이퍼 파라미터 튜닝 1 (batch_size, epochs)

```
def create_model():
    model = Sequential()
    opt=RMSprop(lr=0.00014, rho=0.9, epsilon=None, decay=0.0)
    model.add(Dense(128, input_dim=x_resampled.shape[1], activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
    return model

seed = 6
np.random.seed(seed)
model = KerasClassifier(build_fn = create_model, verbose =1)

batch_size = [50, 200, 100]
epochs = [50, 100,200]
param_grid = dict(batch_size=batch_size, epochs=epochs)

grid = GridSearchCV(estimator = model, param_grid = param_grid,
        cv = KFold(random_state=seed,shuffle=True), verbose =1)
grid_results = grid.fit(x_resampled, y_resampled)
print("Best: {0}, using {1}".format(
        grid_results.best_score_, grid_results.best_params_))

means = grid_results.cv_results_['mean_test_score']
stds = grid_results.cv_results_['std_test_score']
params = grid_results.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('{0} ({1}) with: {2}'.format(mean, stdev, param))
```

[코드 7] batch_size, epochs 기준 Grid Search

먼저, 과적합 현상을 방지하기 위해서 최적의 batch_size와 epochs를 찾는 작업을 진행했다. [코드 7]에 해당 과정이 나와 있다. batch_size는 50, 100, 200의 범위 내에서, epochs 역시 50, 100, 200의 범위 내에서 후보군을 선정했으며, Grid Search를 이용해 최적의 결과를 내는 조합을 찾아냈다. 가장 best의 경우는 [그림 22]의 결과처럼 batch_size와 epochs 모두 100의 값을 지니는 경우였으며, 이때 성능은 소수점 세 자리까지 표현 시, 0.900의 값이 나왔다.

```
Best: 0.8998814105987549, using {'batch_size': 100, 'epochs': 100}
0.8619400858879089 (0.04143294163598879) with: {'batch_size': 50, 'epochs': 50}
0.8926722049713135 (0.01974376316175716) with: {'batch_size': 50, 'epochs': 100}
0.8926671862602233 (0.023526674849552395) with: {'batch_size': 50, 'epochs': 200}
0.8448996663093566 (0.007430710366369083) with: {'batch_size': 200, 'epochs': 50}
0.863494050502777 (0.030913171415396783) with: {'batch_size': 200, 'epochs': 100}
0.8998814105987549 (0.014584248215795424) with: {'batch_size': 100, 'epochs': 100}
0.8384358286857605 (0.029215181191311116) with: {'batch_size': 100, 'epochs': 50}
0.84601229429245 (0.034495465520618294) with: {'batch_size': 200, 'epochs': 200}
0.8786211371421814 (0.04362401317217036) with: {'batch_size': 100, 'epochs': 200}
```

[그림 22] Grid Search 결과 (batch_size, epochs 기준)

3.2.5 하이퍼 파라미터 튜닝 2 (Learning Rate, rho)

```
def create_model(rho,lr):
    model = Sequential()
    opt=RMSprop(lr=lr, rho=rho, epsilon=None, decay=0.0)
    model.add(Dense(128, input_dim=x_resampled.shape[1], activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(128, activation='relu'))
    tf.keras.layers.AlphaDropout(0.3, noise_shape=None, seed=None)
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
    return model

seed =6
np.random.seed(seed)
model = KerasClassifier(build_fn = create_model,
                epochs =5, batch_size =20, verbose =0)

rho = [0.9, 0.8, 0.7]
lr = [0.0001, 0.001, 0.00001]
param_grid = dict(lr=lr, rho=rho)
grid = GridSearchCV(estimator = model, param_grid = param_grid,
                cv = KFold(random_state=seed,shuffle=True), verbose =1)
grid_results = grid.fit(x_resampled, y_resampled)
```

[코드 8] Learning Rate, rho 기준 Grid Search

이번엔 최적 해를 빠르고 정확하게 찾기 위해서 Learning Rate와 Rho 값을 Grid Search를 이용해서 최적 조합을 확인했다. [코드 8]과 같은 과정을 거쳐 모델 제작 및 학습, 평가를 진행했다. Learning Rate는 0.001, 0.0001, 0.00001로 설정했다. 또한, Rho의 값은 0.7, 0.8, 0.9의 값을 토대로 학습 및 평가를 수행했다. best case는 [그림 23]의 결과처럼 lr은 0.0001, rho는 0.9의 값을 지닐 때이며, 이때 성능은 0.838의 값이 나왔다.

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
Best: 0.8376875996589661, using {'lr': 0.0001, 'rho': 0.9}
0.7038209438323975 (0.015536282900006072) with: {'lr': 0.001, 'rho': 0.9}
0.68790682554245 (0.015500385441618405) with: {'lr': 0.0001, 'rho': 0.8}
0.6882755398750305 (0.013634766766607386) with: {'lr': 0.0001, 'rho': 0.7}
0.8376875996589661 (0.019345160104122565) with: {'lr': 0.0001, 'rho': 0.9}
0.8232771158218384 (0.02733706506326075) with: {'lr': 0.001, 'rho': 0.8}
0.8278290510177613 (0.010495489895340363) with: {'lr': 0.001, 'rho': 0.7}
0.6776666164398193 (0.010234468325862856) with: {'lr': 1e-05, 'rho': 0.9}
0.6776666164398193 (0.010234468325862856) with: {'lr': 1e-05, 'rho': 0.8}
0.6776666164398193 (0.010234468325862856) with: {'lr': 1e-05, 'rho': 0.7}
```

[그림 23] Grid Search 결과 (Learning Rate, rho 기준)

3.2.6 모델 변형

앞선 네 가지 하이퍼파라미터를 기반으로, 기존 모델의 Dense 구조를 수정해가며 더 나은 성능의 예측 모델을 만들어나갔다. 최종적으로 [코드 9]의 모델 1과 [코드 10]의 모델 2가 고성능 모델의 후보로 선정되었다.

최종 모델 1은 기본적인 Dense 레이어를 적절히 중첩하여 예측을 수행하도록 제작했다. 최종 모델 2는 Dense 레이어에다 Drop-out 규제항을 추가하여 더욱 일반화된 모델로 나아갈 수 있도록 구성하였다.

```
opt=RMSprop(lr=0.0001, rho=0.9, epsilon=None, decay=0.0)
model = Sequential()
model.add(Dense(1024, input_dim=x.shape[1], activation='relu'))
model.add(Dense(512, input_dim=x.shape[1], activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
return model
```

[코드 9] 최종 모델 1

```
opt=RMSprop(lr=0.0001, rho=0.9, epsilon=None, decay=0.0)
model = Sequential()

model.add(Dense(512, input_dim=x.shape[1], activation='relu'))
model.add(tensorflow.keras.layers.AlphaDropout(0.5, noise_shape=None, seed=None))
model.add(Dense(256, activation='relu'))
model.add(tensorflow.keras.layers.AlphaDropout(0.5, noise_shape=None, seed=None))
model.add(Dense(128, activation='relu'))
model.add(tensorflow.keras.layers.AlphaDropout(0.5, noise_shape=None, seed=None))
model.add(Dense(128, activation='relu'))
model.add(tensorflow.keras.layers.AlphaDropout(0.5, noise_shape=None, seed=None))
model.add(Dense(128, activation='relu'))
model.add(tensorflow.keras.layers.AlphaDropout(0.5, noise_shape=None, seed=None))
model.add(Dense(128, activation='relu'))
model.add(tensorflow.keras.layers.AlphaDropout(0.5, noise_shape=None, seed=None))
model.add(Dense(128, activation='relu'))
model.add(tensorflow.keras.layers.AlphaDropout(0.5, noise_shape=None, seed=None))
model.add(Dense(32, activation='relu'))
model.add(tensorflow.keras.layers.AlphaDropout(0.5, noise_shape=None, seed=None))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
return model
```

[코드 10] 최종 모델 2

최종적으로 데이터 샘플링을 거쳐서 불균형 문제를 해소한 데이터를 기반으로 로 구축한 모델의 성능을 측정했고, 기존 모델과 비교하였다.

4. 연구 결과 분석 및 평가

	GradientBoost	AptaNet	AdaBoost	XGBoost
F1-Score (CV)	0.817	0.831	0.867	0.898

[표 2] 기존 모델의 성능 비교

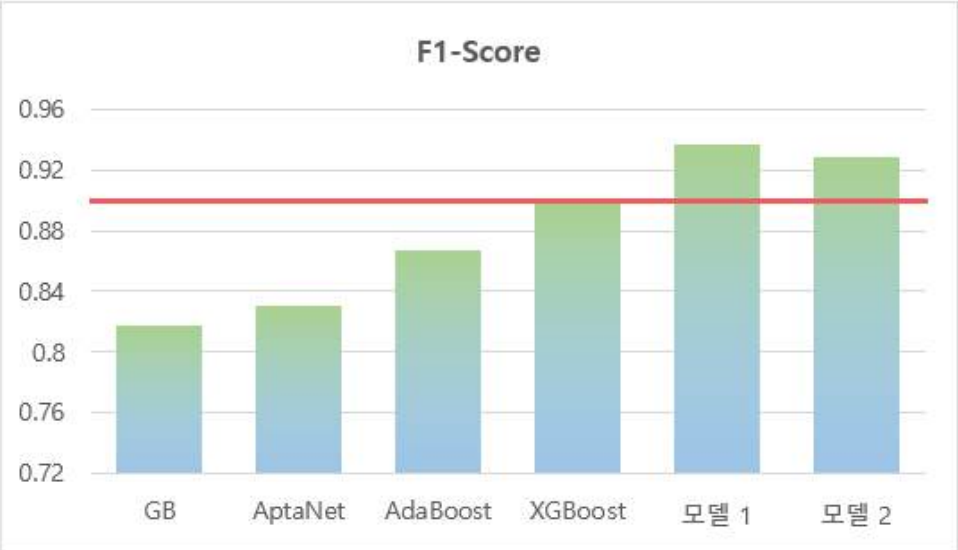
먼저 데이터 전처리 및 샘플링 이전의 모델 성능 결과를 [표 2]에 나타내었다. 기존에 존재하던 raw data를 기준으로 모델의 성능을 비교하였을 때, F1-Score를 기준으로 전체 0.8~0.9 이내의 성능을 보이는 것을 확인할 수 있다. 그 중 XGBoost에서 0.898이라는 가장 좋은 성능을 보임을 확인하였다.

	AdaBoost	XGBoost	Gradient Boost	AptaNet	모델 1	모델 2
F1-Score	0.901	0.929	0.931	0.931	**0.937**	**0.929**

[표 3] 최종 샘플링 이후 모델별 성능 비교

이후 데이터 전처리 및 샘플링을 수행하여 나온 모델별 성능은 [표 3]과 같다. 앞선 [표 2]의 단순 데이터로 학습 및 평가한 모델 성능과 비교하면, k-mer 지표 및

PseAAC 정보 등의 변수를 추가하고 불균형을 고려한 샘플링을 수행하여 모델 학습 시 예측 성능이 확연히 올라간 것을 확인할 수 있다. 모든 모델이 F1-Score 기준 0.900 이상의 값을 보이며, Gradient Boost와 AptaNet이 0.931이라는 높은 수치의 성능을 얻었음을 확인할 수 있다.



[그림 24] 전체 모델 성능 비교

최종 후보로 선정된 모델에도 같은 조건 하에 평가한 결과, 모델 1은 0.937, 모델 2는 0.929의 값을 보인다. 다른 기존 모델에 비해서 모델 1이 가장 좋은 성능을 보임을 확인했다. 기존 모델 성능과 직접 제작한 모델의 성능을 비교한 [그림 24]를 보면 그 차이를 확연하게 알 수 있다. 모델 1과 2 모두 0.90 이상의 F1-Score를 얻었으며, 가장 성능이 좋았던 XGBoost 모델과 비교 시, 절대치로 모델 1은 약 3%, 모델 2는 약 4% 가량 F1-Score를 향상시켰다. 기존에 목표하던 성능 개선을 이뤄냈다.

다만, Drop-out을 적용한 모델 2는 오히려 성능이 낮은 것을 확인할 수 있다. 이는, 현재 최종 모델에서 Drop-out 비율을 50%라는 파라미터로 일괄적으로 맞췄는데, 규제가 과하게 일어났기에 학습률이 다소 낮았을 것으로 생각한다. 따라서 모델 2를 기반으로 추가로 성능 개선을 수행한다고 할 시, Drop-out의 파라미터를 조정하는 방향으로 나아가면 좋은 성능의 모델을 얻어 낼 것이라 본다.

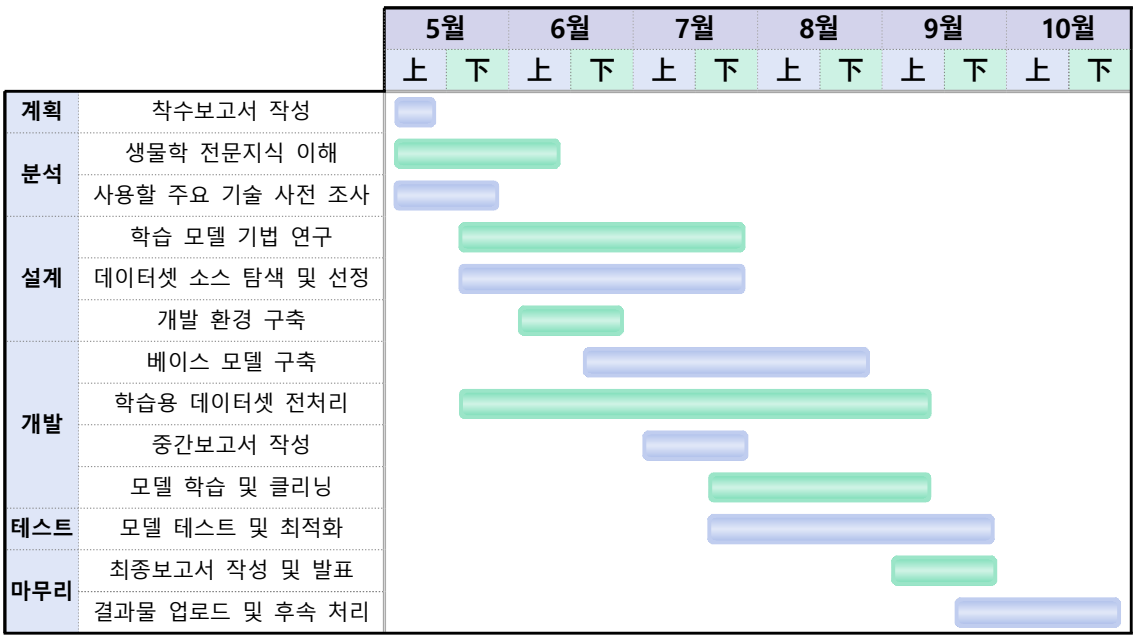
5. 결론 및 향후 연구 방향

데이터 전처리 및 샘플링 과정과 비교하였을 때, 기존 대비 최종 F1-Score 기준 전체 모델이 0.900 이상의 값을 지닌다는 점을 토대로, 전체 모델들의 성능 향상이 이루어진 것을 파악할 수 있었다. 향후 신약 개발 후보물질 추천 모델을 개선 시, 단순 파라미터 조정 이외에도 파생변수 생성과 데이터 샘플링에 집중하면 더 나은 성능의 모델을 얻을 수 있다고 본다.

모델 구축의 경우, 기본 MLP 구조의 파라미터를 조정해나가는 형태로 기존 모델 대비 더 나은 성능의 모델을 얻어낼 수 있었다. 다만, Drop-out을 적용한 모델이 상대적으로 성능이 낮게 나왔다는 결과를 얻었다. 이는 Drop-out 자체의 문제보단 Drop-out 파라미터값의 세밀한 조정이 필요하기 때문이라고 본다. 향후 후속 연구에선 레이어의 구조적인 측면 이외에도 Drop-out의 비율 정도를 고려하여 일반화된 모델을 얻어낸다면 현재 예측 모델보다 우수한 성능을 보일 수 있다고 생각한다.

더 나아가서 현재의 MLP를 이용한 단백질 결합 예측과 유전 알고리즘을 결합하면 일반화된 형태의 모델을 얻기 용이할 것이라 본다. 유전 알고리즘이란 생물학적 진화에 바탕을 둔 통계적 탐색 알고리즘 집합[14]이다. 간단하게 말해서, 어떠한 문제에 대한 해답을 경험적으로 구하는 '메타 휴리스틱'의 한 종류이다. 대체로 답이 정해져 있지 않고, 최적해 역시 알려지지 않은 문제를 해결하는데 주로 쓰인다. 이러한 유전 알고리즘을 본 모델과 결합하여 유전적인 요소도 고려한다면 정교한 모델을 구성할 수 있을 것이라 본다.

6. 개발 일정 및 역할 분담



[표 4] 전체 수행 일정에 관한 Gantt chart

학번	성명	구성원별 역할
201824633	김유진	- 학습용 데이터셋 탐색 - 딥러닝 기반 DeepBind 모델 연구 및 학습 모델 구축 - 학습 모델 파라미터 튜닝 및 최적화 - 보고서 및 발표 자료 작성
201724512	유경민	- 학습용 데이터셋 탐색 - 딥러닝 기반 AptaNet 모델 연구 및 학습 모델 구축 - 학습 모델 파라미터 튜닝 및 최적화 - 보고서 및 발표 자료 작성
201724617	허수민	- 학습용 데이터셋 탐색 및 전처리 - 모델 평가 지표 선정 - 프로그램 개발환경 선정 및 구축 - 보고서 및 발표 자료 작성

[표 5] 조원별 수행 역할

7. 참고 문헌

[1] Shuanghui Yang, Huan Li, Ling Xu, Zhenhan Deng, Wei Han, Yanting Liu, Wenqi Jiang, Youli Zu, Oligonucleotide Aptamer-Mediated Precision Therapy of Hematological Malignancies, Molecular Therapy - Nucleic Acids, Volume 13, Pages 164-175, 2018.

[2] Yi Xi Wu, Young Jik Kwon, Aptamers: The “evolution” of SELEX, Methods, Volume 106, Pages 21-28, 2016.

[3] Emami, N., Ferdousi, R. AptaNet as a deep learning approach for aptamer–protein interaction prediction. Sci Rep 11, 6074, 2021.

[4] Alipanahi, B., Delong, A., Weirauch, M. et al. Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning. Nat Biotechnol 33, 831–838, 2015.

[5] Lee G, Jang GH, Kang HY, Song G. Predicting aptamer sequences that interact with target proteins using an aptamer-protein interaction classifier and a Monte Carlo tree search approach. PLoS One. 2021.

[6] K. Hornik, M. Stincombe, and H. White, "Multilayer feedforward networks are universal approximators," Neural Networks, Vol.2, pp.359-366, 1989.

[7] Ding, Y. S., Zhang, T. L. & Chou, K. C. Prediction of protein structure classes with pseudo amino acid composition and fuzzy support vector machine network. Protein Pept. Lett. 14, 811–815, 2007.

[8] https://www.tensorflow.org 메인 홈페이지 로고

[9] https://keras.io 메인 홈페이지 로고

[10] https://jupyter.org 메인 홈페이지 로고

[11] https://colab.research.google.com 메인 홈페이지 로고

[12] UniProt 메인 페이지, https://www.uniprot.org/ - 해당 사이트에서 단백질 정보를 검색할 수 있다.

[13] Batista, Gustavo EAPA, Ronaldo C. Prati, and Maria Carolina Monard. "A study of the behavior of several methods for balancing machine learning training data." ACM Sigkdd Explorations Newsletter, 6(1) pp. 20-29, 2004.

[14] 장병탁. 유전 알고리즘 이론 및 응용. 전자공학회지. v.22 no.11. 1290-1300. 1995