Mentos Wrap-up Report

- Github
  - research : https://github.com/bcaitech1/p4-dkt-mentos (https://github.com/bcaitech1/p4-dkt-mentos)
  - serving : https://github.com/cheat-tos (https://github.com/cheat-tos)
- **회고 이전 부분은 팀 공통 부분이므로 생략하셔도 됩니다!**

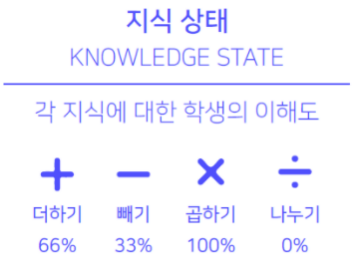
목차

1. 개요

1.1. Competition 소개

1.1.1. 개요

DKT란? **Deep Knowledge Tracing**의 약자로 사용자의 Interaction 데이터를 이용하여 사용자의 "**지식 상태**"를 추적하는 딥러닝 방법론입니다.

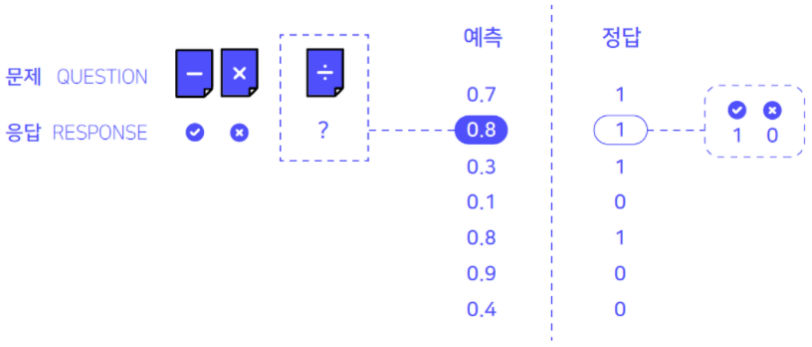


이렇게 사용자의 지식상태를 추적하여 아직 풀지 않은 미래의 문제에 대해서 맞을지 틀릴지 예측하고, 맞춤화된 교육을 제공하기 위해 아주 중요한 역할을 맡게 되어 **교육 AI의 추천**이라고도 불립니다.

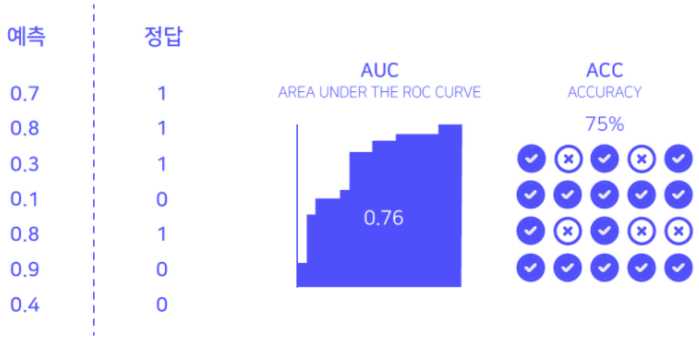
대회에서는 지식상태보다는 **주어진 문제를 맞췄는지 틀렸는지에 집중한다!**

1.1.2. 평가방법

DKT는 주어진 **마지막 문제**를 맞았는지 틀렸는지로 분류하는 **이진 분류 문제**.



평가 Metric은 **AUROC**(Area Under the ROC curve)



1.1.3. Data Format

	userID	assessmentItemID	testId	answerCode	Timestamp	KnowledgeTag
0	0	A060001001	A060000001	1	2020-03-24 00:17:11	7224
1	0	A060001002	A060000001	1	2020-03-24 00:17:14	7225
2	0	A060001003	A060000001	1	2020-03-24 00:17:22	7225
3	0	A060001004	A060000001	1	2020-03-24 00:17:29	7225
4	0	A060001005	A060000001	1	2020-03-24 00:17:36	7225
...	...	...	...	...	...	...
2266581	7441	A030071005	A030000071	0	2020-06-05 06:50:21	438
2266582	7441	A040165001	A040000165	1	2020-08-21 01:06:39	8836
2266583	7441	A040165002	A040000165	1	2020-08-21 01:06:50	8836
2266584	7441	A040165003	A040000165	1	2020-08-21 01:07:36	8836
2266585	7441	A040165004	A040000165	1	2020-08-21 01:08:49	8836

- `userID` : 사용자의 고유번호
- `assessmentItemID` : 문항의 고유번호
- `testId` : 시험지의 고유번호
- `answerCode` : `1` 과 `0` 으로 정답여부를 표기. 예측해야하는 문항의 경우 `-1` 로 표기
- `Timestamp` : 답안 제출 시각
- `KnowledgeTag` : 문항별 Tag

1.2. 목표

1.2.1. Research

- 문제 정의 및 가설을 통한 검증 으로 모델의 성능 향상
- EDA 를 통한 데이터 이해 및 분석
- 모델에 더 많은 정보를 제공하기 위한 `Feature Engineering`
- 창의적인 실험 을 통한 성능 향상

1.2.2. Serving

- ML모델의 End-to-End Pipeline **Lifecycle 경험**
- `Docker` 를 이용한 **Containerization 구현**
- `BentoML` 을 이용한 **Serving pipeline 구축**
- `Airflow` 를 이용한 **Workflow management**

1.2.3. Team work

- `Git` 을 이용한 효과적인 Task Management
- `Notion` 을 이용한 체계적인 Documentation

1.3. 결과

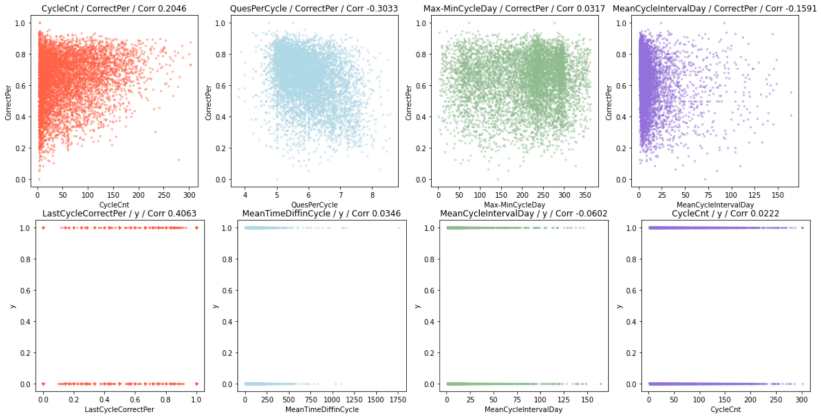
LB Score

Data set	ACC	AUC	Ranking
Public	`0.7581`	`0.8276`	8^{st} / 15
Private	`0.7500`	`0.8330`	8^{st} / 15

2. Research

2.1. EDA

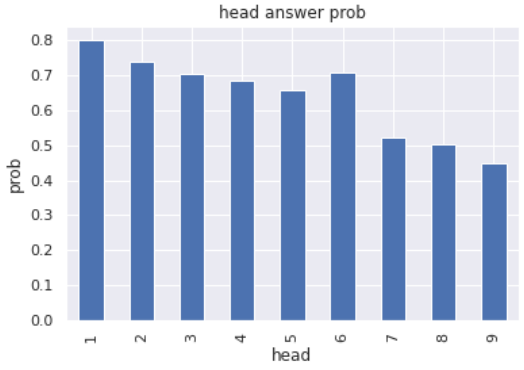
2.1.1. 생성된 feature와 정답률, 마지막 문항의 정답 간의 상관 관계를 파악



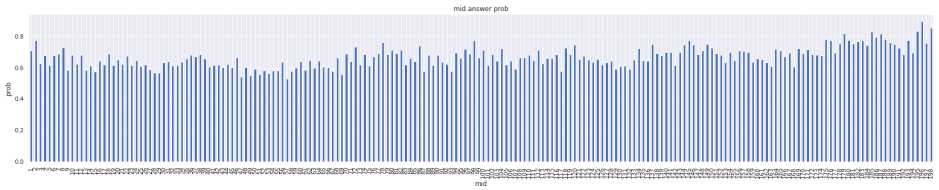
- feature와 정답률 간의 상관관계(위)와 마지막 문항의 정답 간의 상관관계(아래)의 양상은 달랐으며 마지막 문항이 0, 1로만 이루어져 있어 상관관계가 feature의 유의성 지표가 될 수 없다고 판단

2.1.2. assessmentItemID 분해

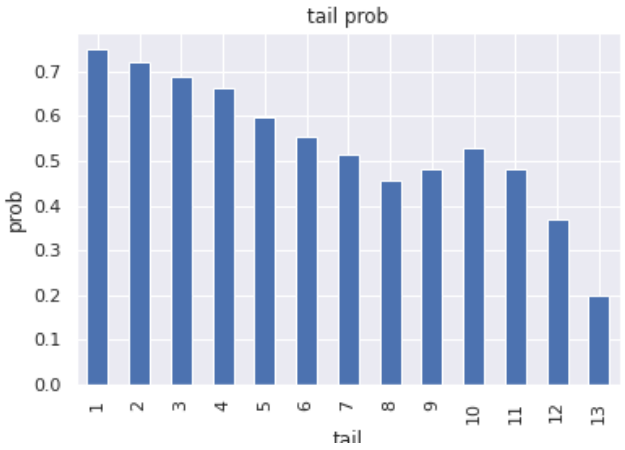
A060001001 형식의 assessmentItemID를 head(A060), mid(001), tail(001)로 나누고 EDA를 진행하였다. head를 Main category라 가정하고 mid를 Chapter, tail을 문항번호라 가정하였다. 따라서 head와 mid가 합쳐져 있는 testID는 하나의 시험지라고 보았다.



head에 대한 추측으로는 학년에서 부터 과목, 난이도 등 여러 가지 추측이 있었지만 개인적으로는 과목이라고 생각했다. 학년으로 보기에는 1년이라는 기간동안 같은 head 내에서 문제풀이가 이루어지지 않았고, 6번 항목이 아니었다면 난이도라고 봤을지도 모르겠다.



mid의 총 개수는 198개이다. 많은 분류만큼 피처로 사용하기 좋을거라는 생각이 들었다. 실제로도 head + mid, mid + tail 피처가 성능향상을 주었다.

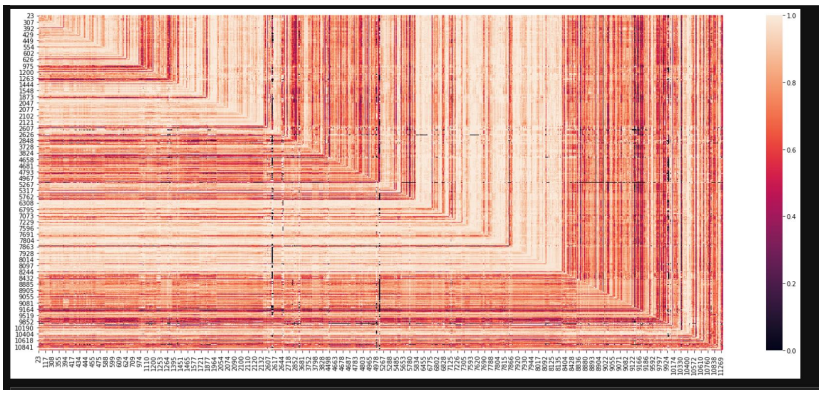


tail은 시험지 별 문항 번호로 가정을 하였는데, 정답률도 뒷 문제로 갈수록 정답률이 낮아지는 일반적인 모양새로 보였다. 최대 갯수가 13개이었기 때문에 이후에 group by를 통한 augmentation 이후 max sequence length를 정하는데 참고하였다.

2.1.3. KnowledgeTag간의 선후행 관계 파악

문제A를 맞췄을 때, 문제B를 맞추는 유저 비율을 구하고, 이를 히트맵으로 나타내어 상관관계를 파악하고자 하였다.

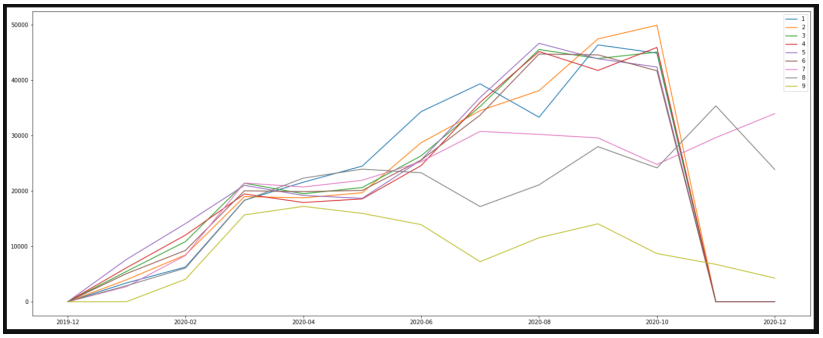
그 결과 아래와 같았다.



어떠한 관계가 있어 보이는 것 같긴하지만, 실제로 적용하기엔 구체적인 방법이 떠오르지 않아 추가로 진행하지 못하였다.

2.1.4. 9가지 대분류에 대한 학년/과목 파악

각 대분류의 월별 Interaction 수를 그래프로 나타내면 아래와 같다.



동일한 UserID를 대상으로 약 2학기가 시작될 시기에 Interaction이 증가하는 것으로 보아, 선행학습을 하는것으로 예상된다.

따라서 9가지의 대분류는 과목보다는 학년일 가능성이 높다고 판단된다.

2.2. Preprocessing

2.2.1. Categorical feature

- sklearn.preprocessing의 LabelEncoder를 통한 처리와 Unknown 클래스 추가

2.2.2. Numeric feature

- 마찬가지로 sklearn.preprocessing의 StandardScaler 사용. 건모님 실험에서 MinMax scaler 보다 더 좋은 결과가 나와서 사용하였다.

2.3. Feature Engineering

2.3.1. ML model 학습을 위한 feature 생성

- userID, testId, KnowledgeTag, assessmentItemID 등을 기준으로 정답률에 대한 통계량을 feature로 추가
- Cycle로 묶어 정답률에 대한 통계량을 feature로 추가
  (cycle : 한 사용자가 푼 하나의 test, 한 사용자가 같은 test를 두 번 풀어도 두 test는 다른 cycle)
- 이전 timestamp를 이용하여 한 문제를 푼 시간 feature 추가
- 한 사용자가 같은 test를 몇 번 풀었는지 나타내는 feature 추가
- 위 feature를 포함하여 총 33개의 feature 생성

2.3.2. DL model 학습을 위한 feature 생성

- `assessmentItemID` 는 A 제외하고 앞의 6자리는 시험번호, 6자리 시험번호 중 앞의 3자리는 0x0 형태로, x는 0~9의 대분류를 나타내고, 그 뒤의 마지막 3자리는 문항번호이다

- `assessmentItemID` 의 총 9자리를 3자리씩 자르고, 그들을 적절히 조합하는 방식으로 feature를 생성
  - `head` : 대분류 9개(1~9)
  - `mid` : 중분류 198개(1~198)
  - `tail` : 문항번호인 마지막 3자리 13개(1~13)
  - `paperID` : 6자리 시험번호 (`head + mid`)
  - `head_tail` : feature를 추가하기 전에도 의미에 의심이 갔고, 성능저하로 제외
  - `mid_tail` : mid의 다양성에서 정보를 얻은 것 같다

- `Timestamp` feature를 활용해서 문제를 푸는데 걸리는 시간인 elasped time을 계산해서 numeric feature로 사용

- 이외에도 `userID` 별, `head` , `mid` , `tail` 별 정답률 feature를 만들어 학습했으나 성능에 큰 향상이 없어 정답률 feature는 제외

- `time diff` timestamp 기준으로 전, 후의 차이를 구하여 feature를 추가하였다. 첫 문제에 대한 시간은 두번째 문제의 시간으로 대체하였다. (backfill) 이후에 qcut, cut을 사용하여 범주화 하였고, 최대 시간 제한을 600초로 만들고 1초 단위로 cut을 해주었을 때 제일 좋은 성능이 나왔다.

2.4. Augmentation

2.4.1. Group by Augmentation

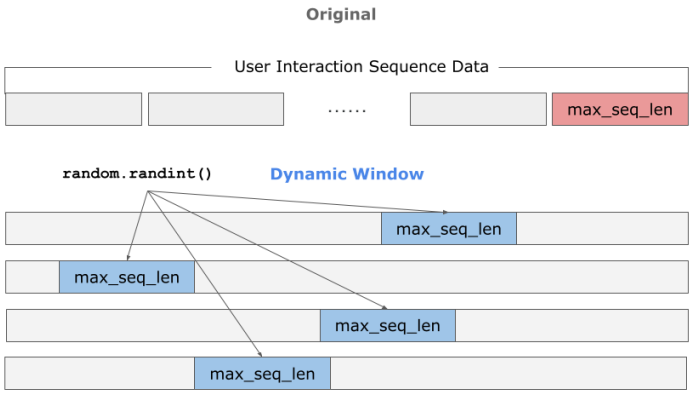
- `userID` 와 대분류, 중분류로 group by하여 사용자가 푼 모든 문제지에 대한 데이터를 사용
- 기존 6,698개의 세트에서 365,164로 증강
- 문제지 별 문항의 최대 개수는 13이었기 때문에 모델에 넣어줄 최대 sequence 수를 13으로 하였습니다

2.4.2. Augmentation

	userID	assessmentItemID	testId	answerCode	Timestamp	KnowledgeTag	newID
1	0	A060001001	A060000001	1	2020-03-24 00:17:11	7224	0
2	0	A060001002	A060000001	1	2020-03-24 00:17:14	7225	0
3	0	A060001003	A060000001	1	2020-03-24 00:17:22	7225	0
4	0	A060001004	A060000001	1	2020-03-24 00:17:29	7225	0
5	0	A060001005	A060000001	1	2020-03-24 00:17:36	7225	0
6	0	A060001007	A060000001	1	2020-03-24 00:17:47	7225	0
7	0	A060003001	A060000003	0	2020-03-26 05:52:03	7226	0
8	0	A060003002	A060000003	1	2020-03-26 05:52:10	7226	0
9	0	A060003003	A060000003	1	2020-03-26 05:53:14	7226	0
10	0	A060003004	A060000003	1	2020-03-26 05:53:29	7226	0
11	0	A060003005	A060000003	1	2020-03-26 05:53:48	7226	0
12	0	A060003006	A060000003	1	2020-03-26 05:53:55	7226	0
13	0	A060003007	A060000003	1	2020-03-26 05:54:11	7226	0
14	0	A060005001	A060000005	1	2020-03-31 05:02:52	7227	0
15	0	A060005002	A060000005	1	2020-03-31 05:03:04	7228	0
16	0	A060005003	A060000005	1	2020-03-31 05:03:11	7228	0
17	0	A060005004	A060000005	1	2020-03-31 05:03:26	7228	0
18	0	A060005005	A060000005	1	2020-03-31 05:03:36	7228	0
19	0	A060005006	A060000005	0	2020-03-31 05:05:14	7227	0
20	0	A060005007	A060000005	1	2020-03-31 05:05:48	7228	0
21	0	A060007001	A060000007	1	2020-04-02 04:53:37	7229	1
22	0	A060007002	A060000007	1	2020-04-02 04:53:46	7229	1

- `userID` 내에서 `testId` 를 3개씩(임의의 수) 묶으면 `seq_len` 이 약 20개로, `max_seq_len` 과 비슷해지게 쪼갤 수 있다
- `userID` 로 groupby해서 만들어지는 한 user의 seuqence 길이는 최소 9, 최대 1860, 평균 338인데 `max_seq_len` 을 20(임의의 수)로 자르면 나머지 sequence를 사용하지 못하는 문제점을 해결하고자 함
- 기존 6,698개의 세트에서 116486 세트로 증강

2.4.3. Dynamic Window



- 기존 베이스라인 코드에서는 마지막 Sequence만 사용하여 데이터를 생성
  - GroupBY는 UserID 기준으로 할 경우 약 6900개 정도의 학습 데이터가 생성
  - Bert 같은 언어 모델링 모델을 학습하기에는 터무니 없이 작은 데이터 셋
  - 데이터 증강 기법을 사용하여 학습 데이터를 증가 할 필요 요구
- 동적으로 마지막 Sequence를 사용하지 않고 랜덤하게 Sequence를 사용하도록 구현
  - Validation에서 AUC가 소폭 상승했음.

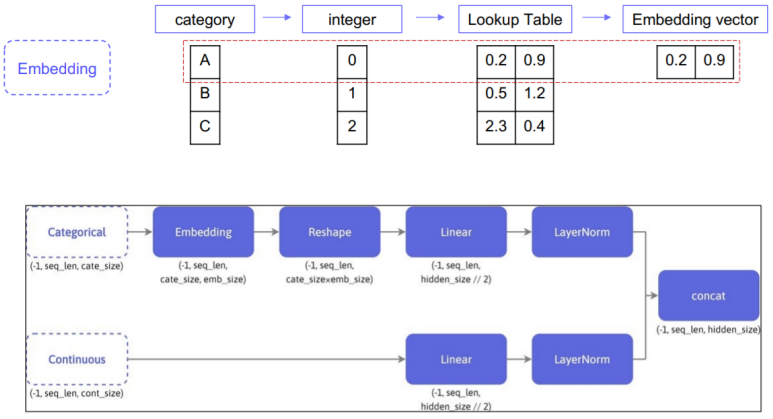
2.5. Model

2.5.1. LGBM

- Feature와 정답율 간의 상관 관계를 확인하고 RFE로 feature selection
- LightGBM과 Pycaret 라이브러리를 활용하여 k fold까지 수행
- userID, testID head, Cycle을 기준으로 데이터를 aggregate하여 분석해보았다.
  y는 가장 마지막 행의 answer
  - userID 기준 LB score : 0.7269
  - testID head 기준 LB score : 0.7290
  - Cycle 기준 LB score : 0.7313
- 데이터를 aggregate하지 않고 row data에서 feature을 추가하여 분석해보았다.
  각 행의 y는 가장 마지막 행의 answer code로 하거나 다음 문제에 대한 answer로 하였다.
  - 다음 행의 answer을 y로 했을 때 LB score : 0.7685
  - 마지막 행의 answer을 y로 했을 때 LB score : 0.7842
  - 마지막 행의 answer을 y로 하고 pycaret을 사용했을 때 LB score : 0.7893
- Aggregation data를 사용했을 때, 성능이 낮은 이유는 aggregation으로 인한 데이터 손실, 그에 따른 정보량의 감소라고 생각한다.

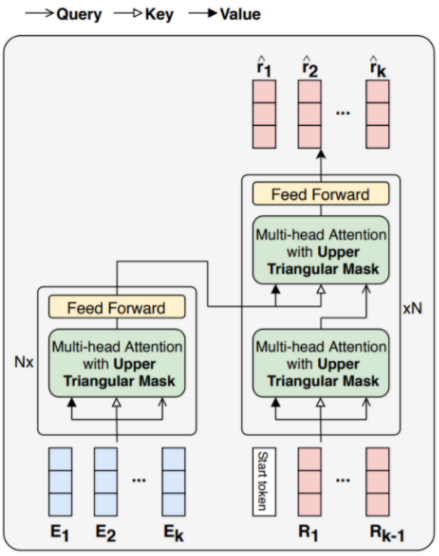
2.5.2. BERT

- Hugging face의 BERT Model 사용, 왜 BERT를 사용하였나?
  기본 베이스라인 기준으로 AUC가 가장 낮게 나와서 최대한 성능향상을 시킨 이후에 더 좋다는 모델로 바꾼다면 점수가 더 오를거라 기대했다. 하지만, 이후에 DKT 분야에서 더 좋은 결과를 얻었다는 SAINT 모델을 실험하였지만 기존 BERT를 넘을 수 없었고, 추가로 SAINT는 decoder를 사용하였기 때문에 시간도 오래걸려서 계속 BERT를 사용하였다.
- Categorical feature를 embedding 과정을 거친 후 concat하여 encoder의 inputs_embeds로 주고 numeric feature가 있다면 inputs_embeds 바로 전에 concat 해서 사용하였다.



- LayerNorm과 BatchNorm으로 Normalization 적용
  각 categorical feature를 embedding 한 뒤 concat 하고서 LayerNorm 적용, numeric feature는 BatchNorm 우선 적용한 뒤 LayerNorm 적용. 이후에 두 결과를 concat 후 LayerNorm적용!
- Relu + Drop out으로 Regularization 적용
  BERT model에서 나온 첫 번째 값에 적용.

2.5.3. SAINT



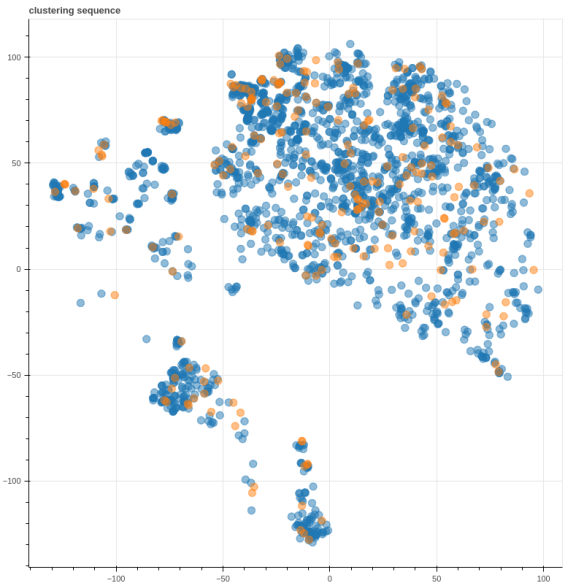
- Exericse와 Response sequence를 나눠서 각각 Encoder와 Decoder에 넣어준다
  - Exercise는 정답여부 제외한 나머지 feature
  - Response는 정답여부에 대한 feature
  - 이번 task에서는 Response를 정답여부 feature 뿐 아니라, Exercise의 모든 feature를 합친 것을 Response로 사용하였다
- 깊은 self-attentive 계산으로 exercises와 responses 사이에 다양한 features 뽑을 수 있게, 즉 더 복잡한(complex) 관계를 계산할 수 있게 해준다
- `Categorical feature` 는 embedding해서 concat 한 후 Linear projection으로 dimension을 맞춰주고 LayerNorm 적용
- `Continuous feature` 는 BatchNorm 적용한 후 Linear projection으로 원하는 dimension을 맞춰주고 LayerNorm 적용
- 이후에 두 결과를 concat 한 후 Linear projection으로 Transformer의 input과 dim을 맞춰준 후 LayerNorm을 거치고 Transformer에 넣어준다
- Encoder와 Decoder는 같은 과정이고, Decoder에는 정답 여부에 대한 feature가 추가되서 cateogrical feature 개수가 하나 늘어난 것이 차이점

2.6. Ensemble

2.6.1 Loss Sequence Ensemble

- 기존 모델에서 마지막에서 K개의 문제를 잘 맞추는 모델로 변형하여 학습
- Inference 단계에서 풀어야(마지막) 하는 문제를 Shift 하면서 확률 값 수행
- (문제-N번째 순서) + (문제-(N-1)번째 순서) + ... + (문제-(N-K)번째 순서)의 확률들의 평균값으로 추론
- Validation에서 AUC가 0.80에서 0.83으로 성능 향상

2.6.2 Input Data Clustering



- 의미있는 Feature 벡터들을 Concat하여 L2유사도로 이웃 K개의 벡터로 추론하는 방법
  - Test 벡터와 대분류 내에서 유사한 K개의 Train 벡터들의 정답 결과의 평균값을 사용
  - Validation 데이터에서 ACC가 0.76으로 준수했으나 LB ACC는 0.46으로 낮은 성능이 나옴.
- KNN알고리즘과 유사한 방식이라 결과가 비슷할 줄 알았지만 KNN LB ACC는 0.68의 성능이 나옴.

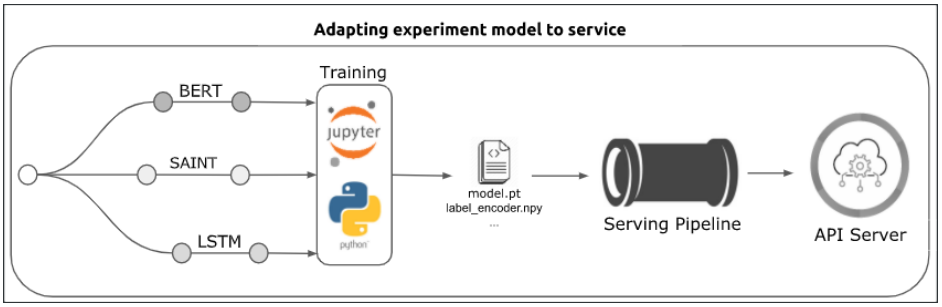
2.6.3 Soft Voting

- 마지막 submission으로 제출
- 서로 다른 구조의 모델들을 앙상블하면 서로 강점이 있는 부분이 달라, 앙상블 시 성능이 오르고 더 robust한 결과를 낼 것이라고 기대
- BERT (`0.8272`), SAINT (`0.8148`), LGBM (`0.7893`), LSTM-Attention (`0.8119`)의 softmax 값들을 weighted sum하는 방식으로 ensemble 진행
  - 비율은 8 : 1 : 0.5 : 0.5
- 최종 Public LB `0.8276` , Private LB `0.8330` (Metirc : AUROC)

3. Serving

다양한 Tool과 Infra를 활용하여 Data를 추출하고, Versioning하고, 새로운 모델을 만들어 컨테이너화 하고, 무중단 배포까지 주기적으로 수행하는 **Fully Automated Serving**을 시도했다.

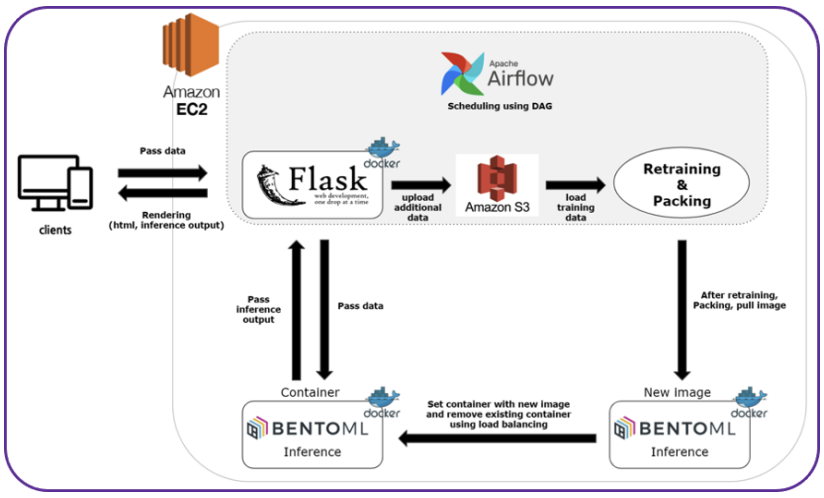
아래는 서비스 제공까지의 전체적인 흐름이다.



실험팀에서 실험한 내용을 토대로 각 모델에 맞는 python 코드가 주어지면, 이것을 토대로 model과 embedding을 만들어내고 serving 파이프라인에 태운다. 파이프라인을 통과하면 API Server가 생성될 것이다.

3.1. Architecture

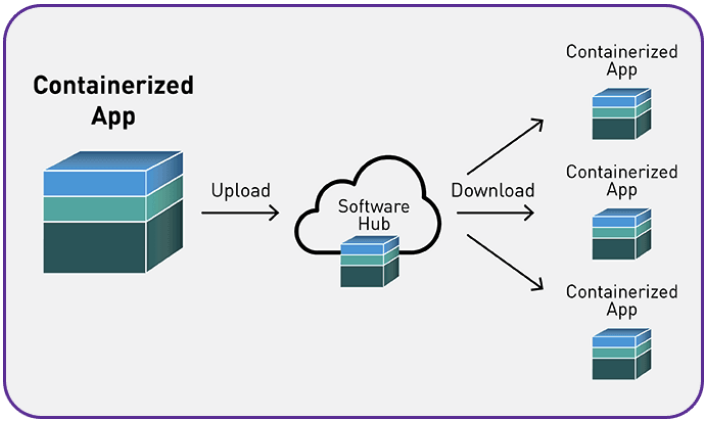
Serving 측의 파이프라인 아키텍쳐는 다음과 같다(최종 수정되었으나, 아래 구조도에는 반영되지 못함)



- 서비스 모델이 실시간으로 제공될 서버를 **Inference Server**로 정의하고, 네이버 클라우드 플랫폼 서버를 사용한다.
- 클라이언트가 편리하게 접근하여 사용하기 위해 서비스 플랫폼은 웹으로 정하고, 이를 구현하기 위해 템플릿을 띄워 Interaction을 받도록 하기 위해 **클라이언트 서버로 `Flask` 를 사용**한다.
- `Flask` 를 통해 생성된 사용자의 Interaction Data를 `S3` Storage에 주기적으로 Update한다.
- Update된 데이터를 load하여 `Airflow` 가 재학습 task를 수행하고, model weight 파일이 나오면 Inference 코드와 합쳐 `BentoML` 로 Packing한다.
- Packing한 서비스를 Docker Image로 만들어 Docker Hub에 업로드한다. 이제 어디에서나 해당 이미지를 내려받아 컨테이너를 생성할 수 있다.
- 이렇게 생성된 Image를 토대로 재학습된 버전의 컨테이너를 기존의 Container와 교체한다. 이 때, Flask Container와 Inference Container의 로드밸런싱 및 무중단 배포를 수행하기 위해 Docker Swarm의 Rolling Update 기능을 사용한다.

이러한 Cycle을 주기적으로 수행하여 Fully Automated Serving을 구현하였다.

3.2. Docker

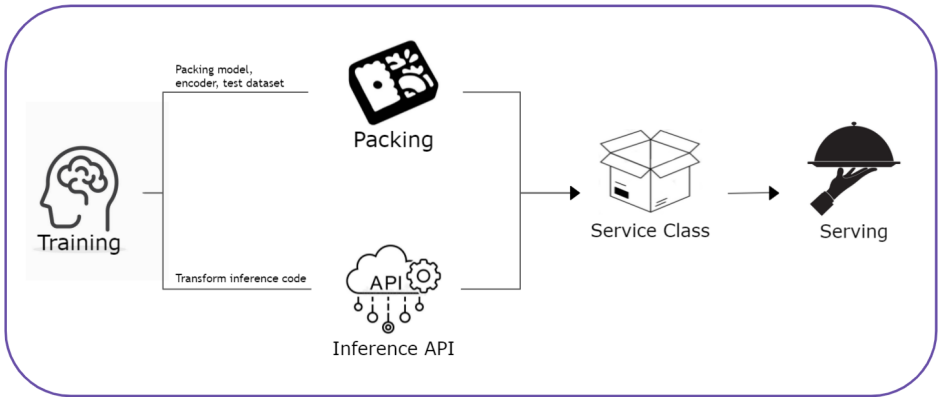


- Docker로 Client, Model Server에 대하여 격리된 컨테이너를 구성함으로써 환경 간섭을 최소화하고, 어느 플랫폼에서나 동작할 수 있도록 할 수 있었다.

- 추가 데이터로 재학습된 모델은 주기적으로 build & push하여 Docker Hub에서 버젼을 관리하였다. 따라서 매 주기마다 새로 학습된 모델을 토대로 한 이미지가 기록되었다.

- Request 급증 시 Image를 토대로 컨테이너를 새로 찍어낼 수 있었다. Docker Swarm의 Health Check와 로드밸런싱 기능을 사용했을 뿐만 아니라, 필요한 시기에 자유자재로 Scale-in/out이 가능해지게 되었다.

3.3. BentoML



- BentoML은 Python 라이브러리 형태로 CLI나 Python Script로 동작할 수 있다. 따라서 Airflow Scheduler로 task에 등록하여 사용하였다.

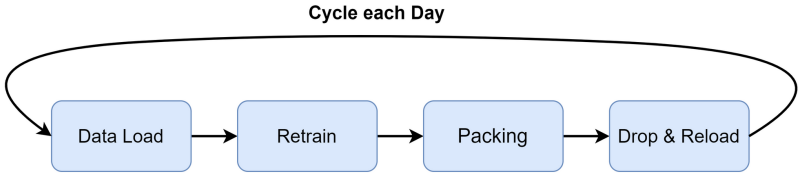
- BentoML은 inference시에 사용되는 모델이나 encoder 등을 디렉토리에 넣어두고 artifact로 한 번에 묶어서 보관할 수 있다. 또한, 디렉토리 내에 packing 후 save 과정을 거치면 코드 내에서 사용된 라이브러리를 추론하여 requirements.txt를 만들어주고 dockerfile까지 자동으로 생성해준다.

- 또, 클라우드 인프라 및 컨테이너 서비스와의 연동성도 좋아 현존하는 메이저 플랫폼 대부분을 지원한다.

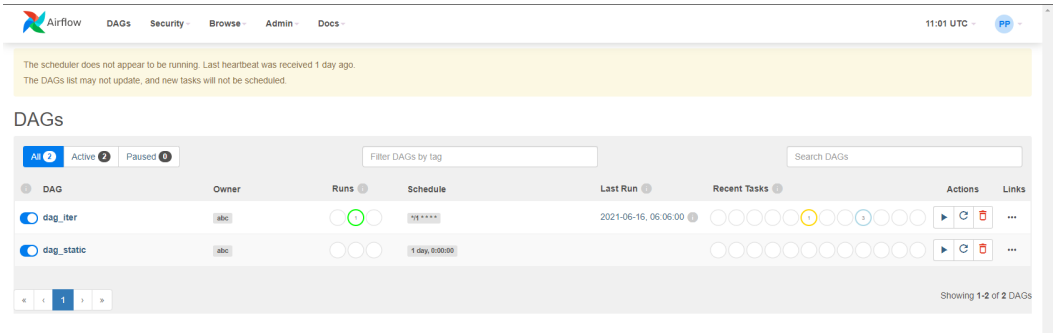
- BentoML에서는 inference api를 제공하지만, json 형태로 data를 전해주어야하므로, 이를 클라이언트 interaction으로 데이터를 받기 위해서 Flask 서버를 Client로 두어 데이터를 추출하고 request로 전송했다. 이에 대한 Inference Response를 Flask로 받아 다시 렌더링 하였다.

3.4. Airflow

효과적인 Workflow Scheduling 및 Monitoring을 수행하기 위해 **Airflow** 라이브러리를 이용하였다.



Airflow는 `DAG` 를 이용하여 Workflow를 원하는 시간대/빈도로 수행할 수 있는데, `DAG` 구성시에 `Bash_Operator` , `Python_Operator` 외에도 다양한 Operator를 지원하여 확장성, 이식성이 뛰어나다.

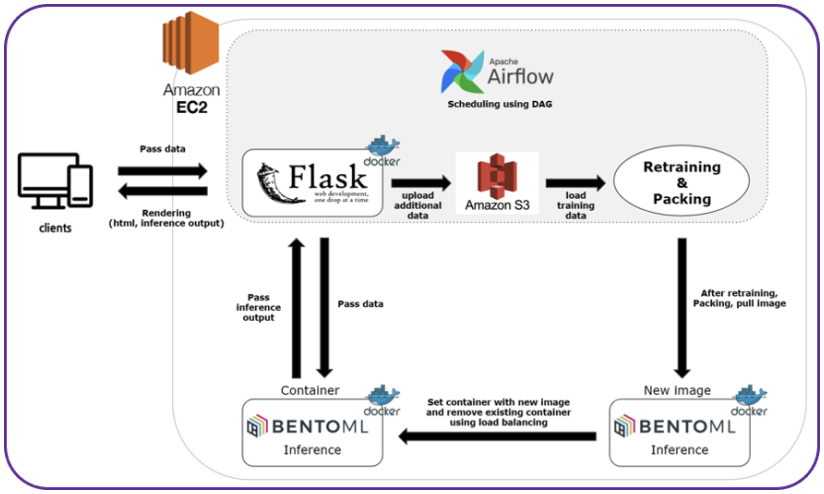


또한 자체 GUI 웹서버가 존재하므로, DAG의 수행상태와 결과에 대한 기록을 시각적으로 확인할 수 있어 직관적인 모니터링이 가능했다.

분기처리로 Lifecycle 중 상황에 맞추어 유연한 예외처리가 가능하다는 장점도 존재한다.

4. 회고

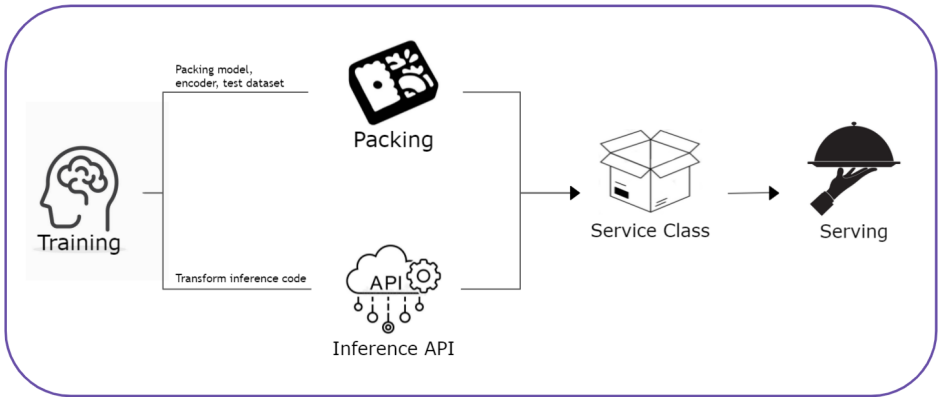
4.1. 개인회고



이번에 서빙의 파이프라인은 크게

- **BentoML로 inference api 생성**
- **Flask로 랜더링**
- **재학습한 모델로 도커 이미지 생성 및 inference container 교체**
- **교체 시 지속적인 서비스 제공을 위한 로드 밸런싱**
- **파이프라인 스케줄링을 위한 Airflow**

로 구성된 것 같다.



처음에는 bentoML로 api와 클라이언트 연결을 모두 해결하고 싶었지만, bentoML의 기능을 알아보니 HTML 랜더링 기능은 없어서 **Flask로 랜더링**을 진행하게 되었다.

bentoML에서는 inference api 구성을 위해 **기존 실험에 사용된 코드를 변형**하고 **model이나 encoder 같은 파일들을 packing** 해주었다.

이렇게 bentoML, Flask를 다루고 어느정도 서빙에 대한 이해를 가지게 된 것은 나에게 **큰 발전**이었다.

그럴만도 한게 처음 서빙에 대한 이야기를 나눌 때, 성익님께서 어느정도 계획한 파이프라인을 설명해주셨는데 api, docker, image 등 거의 모든 단어와 개념을 몰랐었다.

하지만 구글링과 감사한 팀원분들의 설명으로 어느 정도 감을 잡고 프로젝트를 진행할 수 있었다.

BentoML을 모두 구성한 뒤에 EC2 서버를 빌려 안에서 학습을 하고 이를 이미지로 말아 inference는 하나의 컨테이너로 구성하기로 하였다.

이는 성익님께서 담당해주셨는데, 서버 내의 환경은 로컬과 많이 달랐던 것 같다. 리소스가 부족한 서버는 작은 딥러닝 모델도 학습하기가 버거웠고 결국 부스트캠프에서 제공받은 서버에서 학습한 뒤 이미지로 말기로 하였다.

또한, Airflow로 스케줄링을 진행하였는데, 서버 내에서 Airflow와 학습을 동시에 진행하지 못하는 문제도 있으셨다고 한다.

이를 해결하기 위해서는 Kubernetes의 크론잡으로 일정 시간대에 반복적으로 학습을 시키게 할 수 있다는데, 이래서 Kubernetes를 사용하는구나 느끼셨다고 한다.

당장은 cs 공부가 시급하지만 나중에는 Kubernetes, kubeflow를 공부해야 겠다.

그 외에도 여러 특강, 오피스 아워, 이정우 멘토님의 멘토링을 통해 여러 전문가분들의 경험을 간접적으로 느낄 수 있었다. 아직 딥러닝 입문자라 크게 와닿지 않는 말씀들도 있었지만, 무엇 하나 값지지 않은 것이 없었다. 필기해뒀으니 두고 두고 봐야겠다.

그리고 깃을 통한 협업은 이번이 처음이었는데, 유지님께서 깃의 동작 원리를 친절하게 설명해주셨다. 원래는 pull이나 merge의 개념도 몰랐고 branch는 아무데서나 만들어서 사용했는데 원리를 알고나니 확실히 conflict가 안나기 시작했다.

이번 프로젝트를 통해 어느 때보다 많이 경험하고 많이 배웠으며 낯선 것을 배울 때 가장 많이 배우는구나 싶었다.

어쨌든, 길다면 긴 6개월 간의 부스트 캠프가 끝이 났다. 낯선 지식들과 프로젝트 들로 쉽지 않은 교육이었지만, 그만큼 배운 것도 얻은 것도 많았다.

이번에 졸업도 하게 되어 취업 전선에 뛰어들었는데 이 교육을 계기로 자신을 다듬고 열심히 준비해야겠다.

항상 좋은 말씀 해주시고 좋은 기회 주신 멘토님 감사합니다! :)