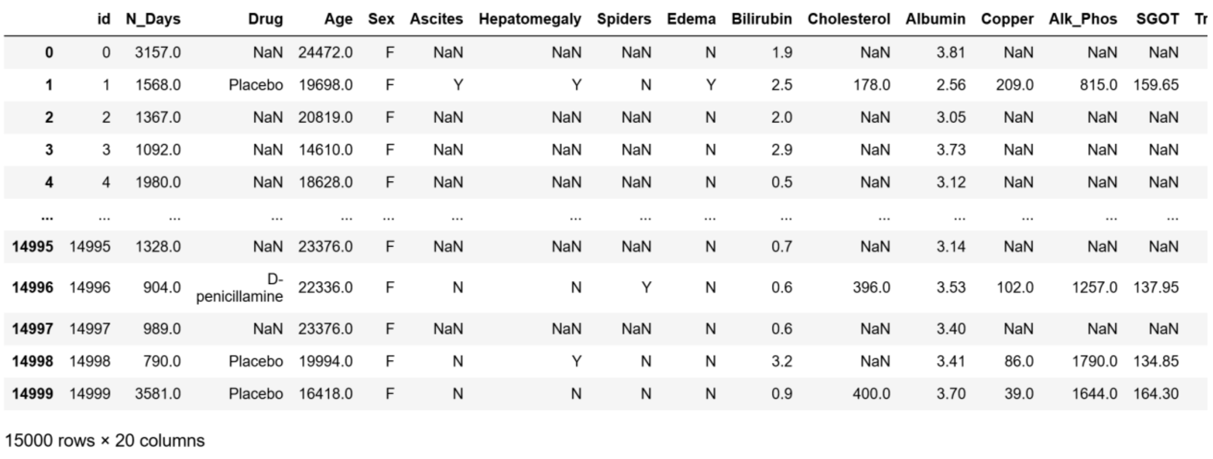
Kaggle Report

Tony Chae

Data Overview & Cleaning

The Overall data structure was “table type”. Which have 20 features of 15000 training data and 10,000 test data. Our goal is to classify by three classes, so we need at least 33% of accuracy.



	id	N_Days	Drug	Age	Sex	Ascites	Hepatomegaly	Spiders	Edema	Bilirubin	Cholesterol	Albumin	Copper	Alk_Phos	SGOT	Tr
0	0	3157.0	NaN	24472.0	F	NaN	NaN	NaN	N	1.9	NaN	3.81	NaN	NaN	NaN	
1	1	1568.0	Placebo	19698.0	F	Y	Y	N	Y	2.5	178.0	2.56	209.0	815.0	159.65	
2	2	1367.0	NaN	20819.0	F	NaN	NaN	NaN	N	2.0	NaN	3.05	NaN	NaN	NaN	
3	3	1092.0	NaN	14610.0	F	NaN	NaN	NaN	N	2.9	NaN	3.73	NaN	NaN	NaN	
4	4	1980.0	NaN	18628.0	F	NaN	NaN	NaN	N	0.5	NaN	3.12	NaN	NaN	NaN	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
14995	14995	1328.0	NaN	23376.0	F	NaN	NaN	NaN	N	0.7	NaN	3.14	NaN	NaN	NaN	
14996	14996	904.0	D-penicillamine	22336.0	F	N	N	Y	N	0.6	396.0	3.53	102.0	1257.0	137.95	
14997	14997	989.0	NaN	23376.0	F	NaN	NaN	NaN	N	0.6	NaN	3.40	NaN	NaN	NaN	
14998	14998	790.0	Placebo	19994.0	F	N	Y	N	N	3.2	NaN	3.41	86.0	1790.0	134.85	
14999	14999	3581.0	Placebo	16418.0	F	N	N	N	N	0.9	400.0	3.70	39.0	1644.0	164.30	

15000 rows × 20 columns

Figure 1 – Raw train data

First, I did look up the distribution and types of feature data to classify into 1) categorical 2) continuous 3) integer. (below)

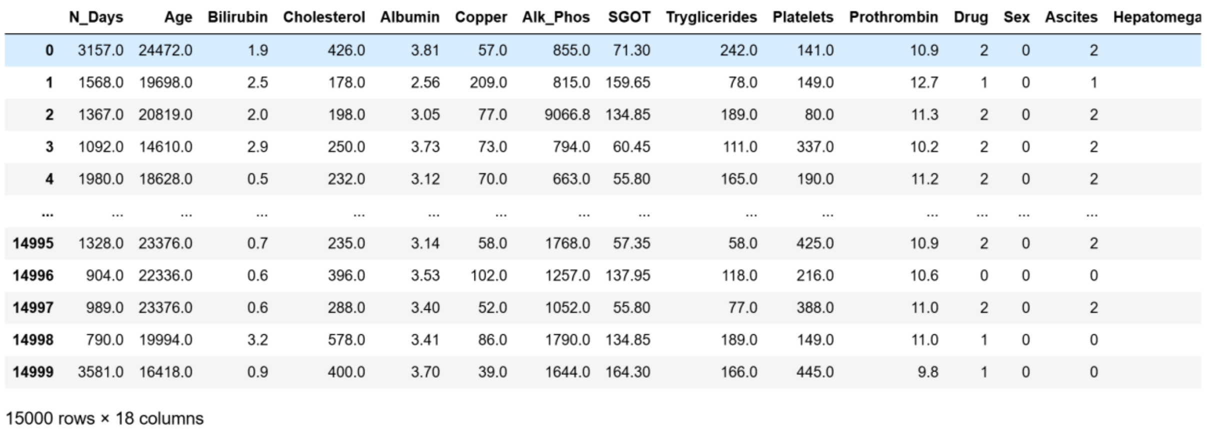
```
categorical_cols = ['Drug', 'Sex', 'Ascites', 'Hepatomegaly',
'Spiders', 'Edema','Stage']


continous_cols = ['N_Days', 'Age', 'Bilirubin', 'Cholesterol',
'Albumin', 'Copper', 'Alk_Phos',
'SGOT','Tryglicerides','Platelets','Prothrombin']
```

Second, the interesting part of data was had lot of “N/A” data. There was few approaches and combinations to find a best performance. I thought For general tree-type model, and artificial neural network type model’s input data should be different. The continuous data could be either normalized or not normalized, substitute N/A data into mean value or most frequent value or not change at all. For categorical data, it could be one-hot encoded or encoding into single integer. Below are the combinations of data cleaning and input data transformation I have tried for different models. Also, I have tried random imputer that I have manually made that filling the missing data into *random.choice* from the feature column data. Luckily it worked out well.

Combinations I have tried for data cleaning

Model	Simple Imputer	Iterative Imputer(Missing Forest)	Random imputer(choose random from column	One – hot encoding	Normalizing & Standard Scale	Leave N/A value
Random Forest	●	●	●	●	●	●
Other tree models	●	●	●	●	●	●
ANN & CNN	●	●	●	●	●	●

After Inputing(filling the missing data) the categorical and continuous data look like below and the features are 18 columns(except label).



	N_Days	Age	Bilirubin	Cholesterol	Albumin	Copper	Alk_Phos	SGOT	Tryglicerides	Platelets	Prothrombin	Drug	Sex	Ascites	Hepatomega
0	3157.0	24472.0	1.9	426.0	3.81	57.0	855.0	71.30	242.0	141.0	10.9	2	0	2	
1	1568.0	19698.0	2.5	178.0	2.56	209.0	815.0	159.65	78.0	149.0	12.7	1	0	1	
2	1367.0	20819.0	2.0	198.0	3.05	77.0	9066.8	134.85	189.0	80.0	11.3	2	0	2	
3	1092.0	14610.0	2.9	250.0	3.73	73.0	794.0	60.45	111.0	337.0	10.2	2	0	2	
4	1980.0	18628.0	0.5	232.0	3.12	70.0	663.0	55.80	165.0	190.0	11.2	2	0	2	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
14995	1328.0	23376.0	0.7	235.0	3.14	58.0	1768.0	57.35	58.0	425.0	10.9	2	0	2	
14996	904.0	22336.0	0.6	396.0	3.53	102.0	1257.0	137.95	118.0	216.0	10.6	0	0	0	
14997	989.0	23376.0	0.6	288.0	3.40	52.0	1052.0	55.80	77.0	388.0	11.0	2	0	2	
14998	790.0	19994.0	3.2	578.0	3.41	86.0	1790.0	134.85	189.0	149.0	11.0	1	0	0	
14999	3581.0	16418.0	0.9	400.0	3.70	39.0	1644.0	164.30	166.0	445.0	9.8	1	0	0	

15000 rows × 18 columns

Figure 2 - Data after Imputing

Feature Engineering

I have thought, is there any way we could transform the data and augment the feature and have additional data rather than substituting N/A values with Simple imputer or iterative imputer. Because, the model performance shows that adding random data in these kind of small data sets increased noise of data and make training loss and test loss worse. So, I have looked up medical research papers about feature explanations of Cirrhosis. The correlating information between features and existence of N/A are below.

Reference paper to understand features relationship:

- Sharma B, John S. Hepatic Cirrhosis. [Updated 2022 Oct 31]. In: StatPearls [Internet]. Treasure Island (FL): StatPearls Publishing; 2024 Jan-. Available from: https://www.ncbi.nlm.nih.gov/books/NBK482419/

- How to diagnose Cirrhosis: https://www.hepatitis.va.gov/cirrhosis/background/how-to-diagnose.asp

1. Ascites, Hepatomegaly, Spiders: These three characteristics are all clinical signs of cirrhosis. If one of them is N/A, the others are also likely to be N/A, as they are assessed during the same examination process.

2. Bilirubin and Edema: High bilirubin levels increase the likelihood of edema. If bilirubin is N/A, edema may also be N/A.

3. Age and other characteristics: Age is almost always known, so it's rare for it to be N/A. However, if Age is N/A, many other characteristics are likely to be N/A as well.

4. Drug and other characteristics: If drug treatment information is N/A, other clinical characteristics (Ascites, Hepatomegaly, Spiders, Edema) may also be N/A, possibly indicating the patient is in the early diagnostic stage or hasn't been followed up.

5. N_Days and other characteristics: If N_Days is N/A, all other characteristics are likely to be N/A, suggesting unclear follow-up duration. This is because more than 80% of N/A data occurs when drug or date information is N/A, making other characteristics likely to be N/A as well.

Feature engineering was performed based on these relationships. The following is the Python structure reflecting information above:

1. Grouping of clinical signs: Analyzes N/A patterns among Ascites, Hepatomegaly, and Spiders.

2. Bilirubin and Edema relationship: Identifies cases where both features are N/A simultaneously.

3. Age-related characteristics: Analyzes the N/A status of other features when Age is N/A.

4. Drug information and clinical characteristics: Identifies N/A relationships between drug information and clinical characteristics.

5. N_Days and other characteristics: Assesses overall data quality when N_Days is N/A.

6. Severity score of clinical signs: Calculates a severity score based on the clinical signs.

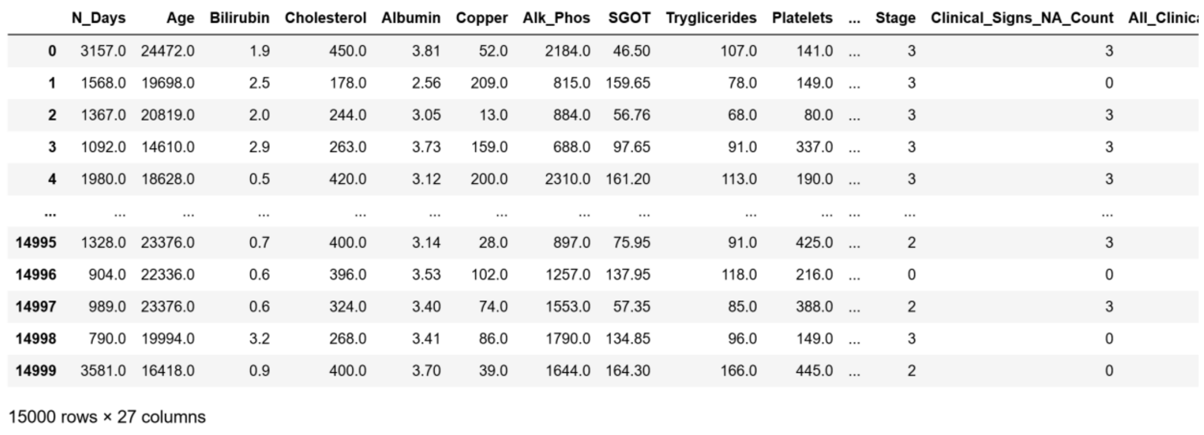
7. Bilirubin outlier detection: Detects outliers in bilirubin levels.

8. Duration of drug treatment: Calculates treatment duration for each drug.

9. Clinical sign differences by gender: Analyzes interactions between gender and clinical signs.

10. N/A pattern encoding: Encodes N/A patterns of key features as strings. (count)

Using this code to generate new features enables the model to better understand the patterns of N/A values and in result, it helped my model to improve cirrhosis prediction accuracy and minimize logloss.



	N_Days	Age	Bilirubin	Cholesterol	Albumin	Copper	Alk_Phos	SGOT	Tryglicerides	Platelets	...	Stage	Clinical_Signs_NA_Count	All_Clinic
0	3157.0	24472.0	1.9	450.0	3.81	52.0	2184.0	46.50	107.0	141.0	...	3	3	
1	1568.0	19698.0	2.5	178.0	2.56	209.0	815.0	159.65	78.0	149.0	...	3	0	
2	1367.0	20819.0	2.0	244.0	3.05	13.0	884.0	56.76	68.0	80.0	...	3	3	
3	1092.0	14610.0	2.9	263.0	3.73	159.0	688.0	97.65	91.0	337.0	...	3	3	
4	1980.0	18628.0	0.5	420.0	3.12	200.0	2310.0	161.20	113.0	190.0	...	3	3	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
14995	1328.0	23376.0	0.7	400.0	3.14	28.0	897.0	75.95	91.0	425.0	...	2	3	
14996	904.0	22336.0	0.6	396.0	3.53	102.0	1257.0	137.95	118.0	216.0	...	0	0	
14997	989.0	23376.0	0.6	324.0	3.40	74.0	1553.0	57.35	85.0	388.0	...	2	3	
14998	790.0	19994.0	3.2	268.0	3.41	86.0	1790.0	134.85	96.0	149.0	...	3	0	
14999	3581.0	16418.0	0.9	400.0	3.70	39.0	1644.0	164.30	166.0	445.0	...	2	0	

15000 rows × 27 columns

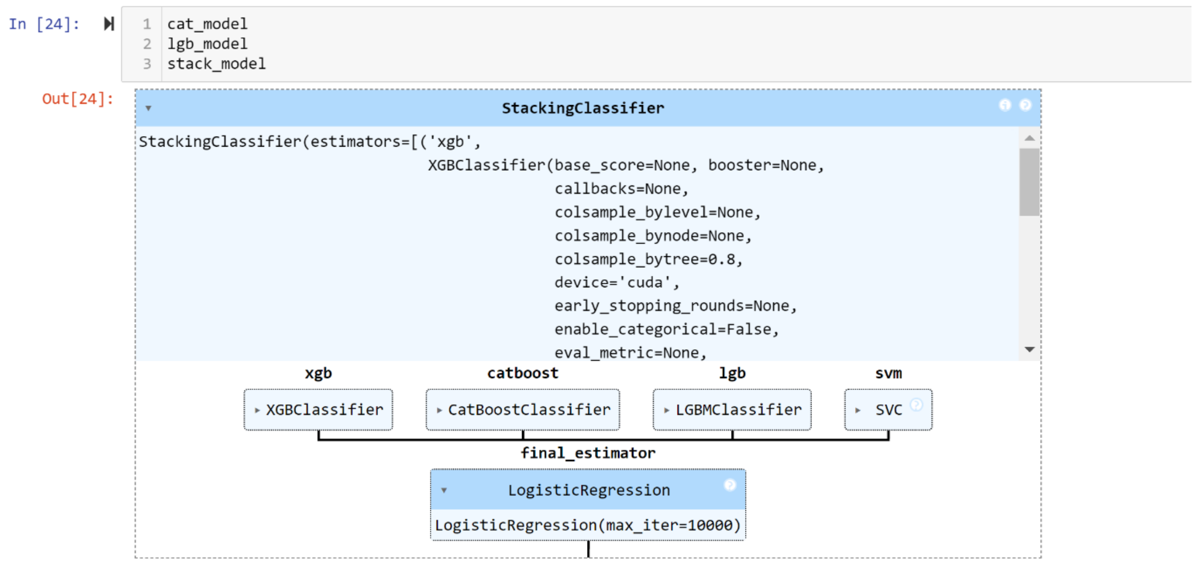
Figure 3- After Feature Engineering

As can see figure above, after feature adding, the feature increased into 27 columns. (augmented column)

Models

1. Using Multiple models:

For using multiple models, I have tried ensembling Random forest, XGBoost, light gradient descent(LGB), SVM model, and catboost using stacking method. I have used logistic regression to understand and to make a final output layer of probability. However it performed worse than I thought so I decided to use single XGBoost model that it performed best among those models.



```
In [24]:
cat_model
lgb_model
stack_model
```

```
Out[24]:
StackingClassifier
StackingClassifier(estimators=[('xgb',
                                XGBClassifier(base_score=None, booster=None,
                                              callbacks=None,
                                              colsample_bylevel=None,
                                              colsample_bynode=None,
                                              colsample_bytree=0.8,
                                              device='cuda',
                                              early_stopping_rounds=None,
                                              enable_categorical=False,
                                              eval_metric=None,
```

2. Singular model – XGBOOST only

a) Hyperparameter tuning: Finding the best hyper parameter was crucial find a global minimum of the data log loss, as well as have a best accuracy. I will talk about how did I tuned and find best parameters on next section.

b) I have used ensemble model to average the output of predicted probability to find a better unbiased result. It performed better than single XG-boost output. I have tried ensembling 3-100 model and 5-10 performed best on test log loss. So I have choosed 5 models to ensemble with different random_seed values.

```
# Number of models to train
import xgboost as xgb

# Number of models to train
num_models = 5
models = []

X_pseudo_simple_xgb=X_train
pseudo_train_label=train_label

#final_param={'max_depth': 16, 'min_child_weight': 14, 'subsample':
0.9237806795685649, 'colsample_bytree': 0.14309918521189277,
'learning_rate': 0.03883023984291649, 'n_estimators': 522}
# final_param={'max_depth': 15, 'min_child_weight': 10, 'subsample':
0.9974547490148263, 'colsample_bytree': 0.15312805153329764,
'learning_rate': 0.023214911136020238, 'n_estimators': 611}
final_param={'max_depth': 12, 'min_child_weight': 8, 'subsample':
0.9665601547562184, 'colsample_bytree': 0.12355113792160276,
'learning_rate': 0.0262390732061374, 'n_estimators': 687}
# Initialize an array to store the predictions
test_probs = np.zeros((X_test_random_xgb.shape[0], 3))  # Assuming 3
classes

for i in range(num_models):
    # Initialize the model with best parameters and changing random
seed
    model = xgb.XGBClassifier(
        objective='multi:softprob',
        num_class=3,
        tree_method='hist',
        device='cuda',
        eval_metric='mlogloss',
        **final_param,
        random_state=i
    )


    # Train the model
    model.fit(X_pseudo_simple_xgb, pseudo_train_label)

    # Store the model in the list
    models.append(model)

for model in models:
    # Get probabilities from each model and accumulate them

    test_probs += model.predict_proba(X_test_random_xgb)
    print(test_probs)

# Average the accumulated probabilities
test_probs /= num_models

output_df = pd.DataFrame(test_probs, columns=['Status_C',
'Status_CL', 'Status_D'])
output_df.insert(0, 'id', test_id)  # Assuming test_id contains the
IDs for your test data
output_df.to_csv('Tony_final_ensemble_ver5_with_augment.csv',
index=False)
```

3. Neural networks

I have tried Articitial Neural Network(ANN) and Convolutional Neural Network(CNN) to test our data set. Unfortunately, it did not performed well.

Below were log loss and accuracy based on SGD and Adam optimizers. (Adam was better)

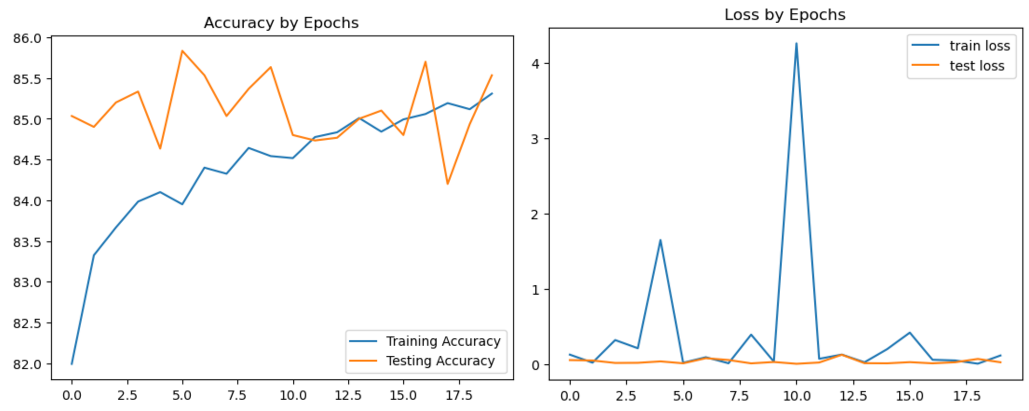
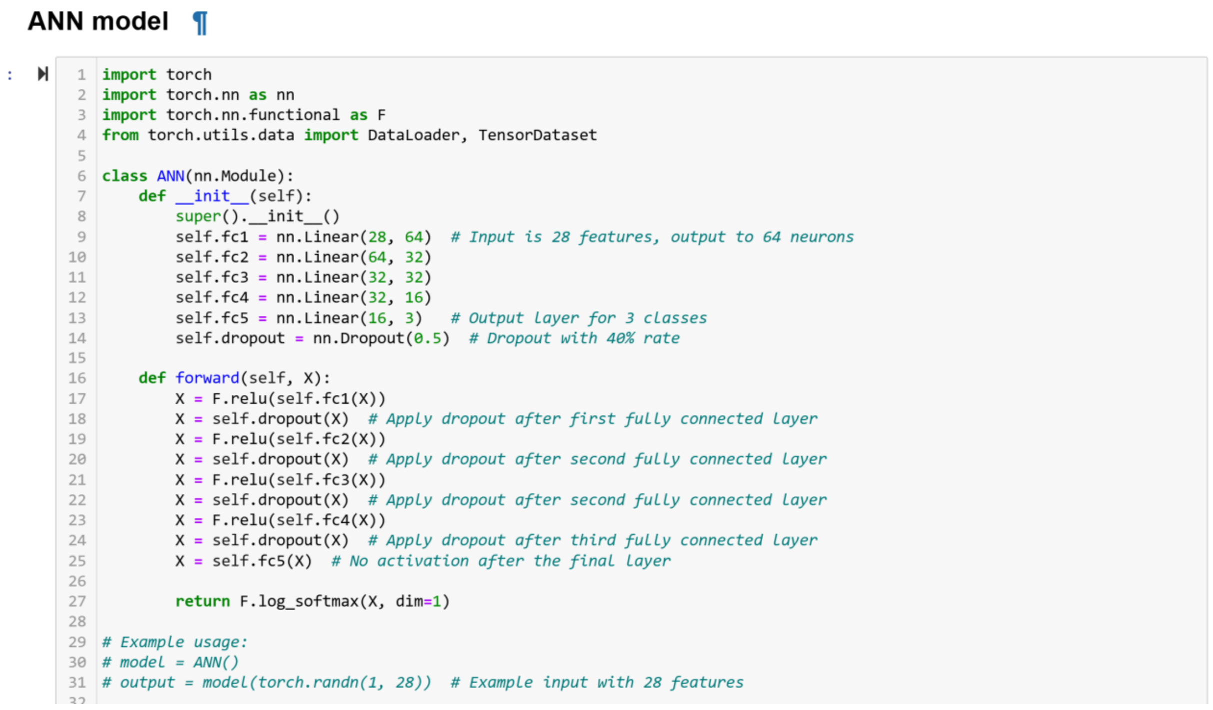


Figure 4 - Log loss of CNN

Figure 5 - Accuracy of CNN

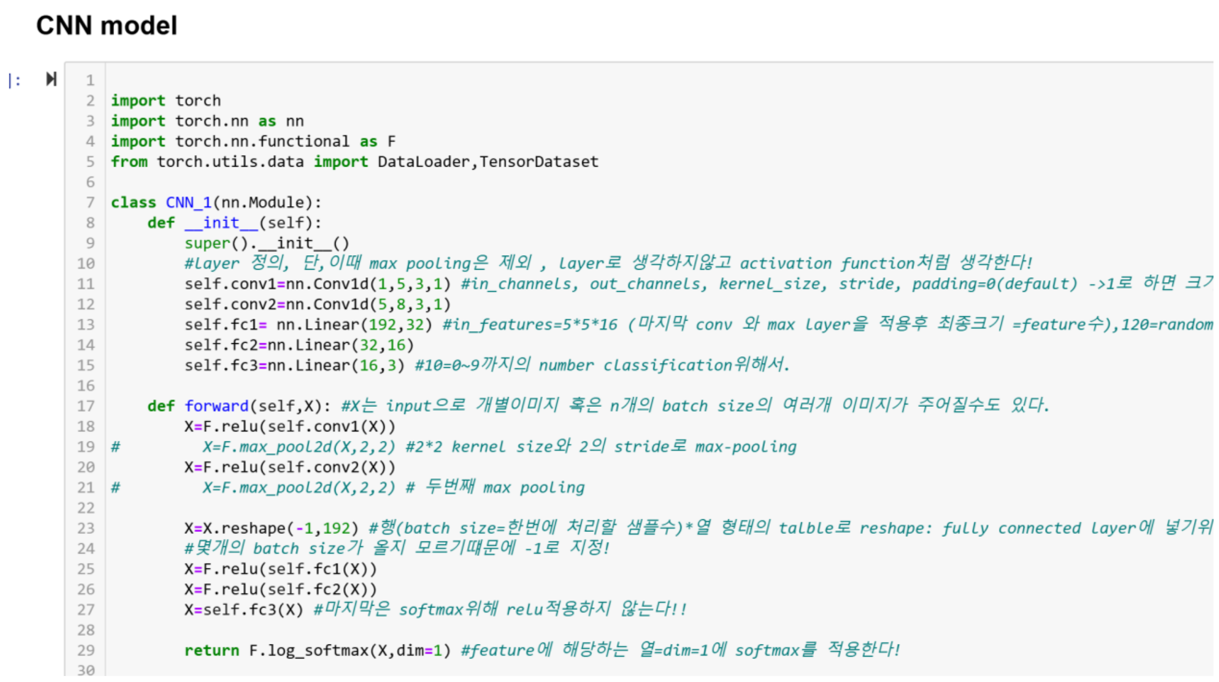
So I have decided to use XGBoost rather than neural networks.



ANN model ¶

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset

class ANN(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28, 64)  # Input is 28 features, output to 64 neurons
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 32)
        self.fc4 = nn.Linear(32, 16)
        self.fc5 = nn.Linear(16, 3)   # Output layer for 3 classes
        self.dropout = nn.Dropout(0.5)  # Dropout with 40% rate

    def forward(self, X):
        X = F.relu(self.fc1(X))
        X = self.dropout(X)  # Apply dropout after first fully connected layer
        X = F.relu(self.fc2(X))
        X = self.dropout(X)  # Apply dropout after second fully connected layer
        X = F.relu(self.fc3(X))
        X = self.dropout(X)  # Apply dropout after second fully connected layer
        X = F.relu(self.fc4(X))
        X = self.dropout(X)  # Apply dropout after third fully connected layer
        X = self.fc5(X)  # No activation after the final layer

        return F.log_softmax(X, dim=1)

# Example usage:
# model = ANN()
# output = model(torch.randn(1, 28))  # Example input with 28 features
```

Figure 6 - ANN model



CNN model

```python

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader,TensorDataset

class CNN_1(nn.Module):
    def __init__(self):
        super().__init__()
        #layer 정의, 단,이때 max pooling은 제외 , layer로 생각하지않고 activation function처럼 생각한다!
        self.conv1=nn.Conv1d(1,5,3,1) #in_channels, out_channels, kernel_size, stride, padding=0(default) ->1로 하면 크
        self.conv2=nn.Conv1d(5,8,3,1)
        self.fc1= nn.Linear(192,32) #in_features=5*5*16 (마지막 conv 와 max layer을 적용후 최종크기 =feature수),120=random
        self.fc2=nn.Linear(32,16)
        self.fc3=nn.Linear(16,3) #10=0~9까지의 number classification위해서.

    def forward(self,X): #X는 input으로 개별이미지 혹은 n개의 batch size의 여러개 이미지가 주어질수도 있다.
        X=F.relu(self.conv1(X))
#         X=F.max_pool2d(X,2,2) #2*2 kernel size와 2의 stride로 max-pooling
        X=F.relu(self.conv2(X))
#         X=F.max_pool2d(X,2,2) # 두번째 max pooling

        X=X.reshape(-1,192) #행(batch size=한번에 처리할 샘플수)*열 형태의 talble로 reshape: fully connected layer에 넣기위
        #몇개의 batch size가 올지 모르기때문에 -1로 지정!
        X=F.relu(self.fc1(X))
        X=F.relu(self.fc2(X))
        X=self.fc3(X) #마지막은 softmax위해 relu적용하지 않는다!!

        return F.log_softmax(X,dim=1) #feature에 해당하는 열=dim=1에 softmax를 적용한다!

```

Figure 7 - CNN model

Model Training and Optimizing

For Hyper parameter tuning I have used three methods

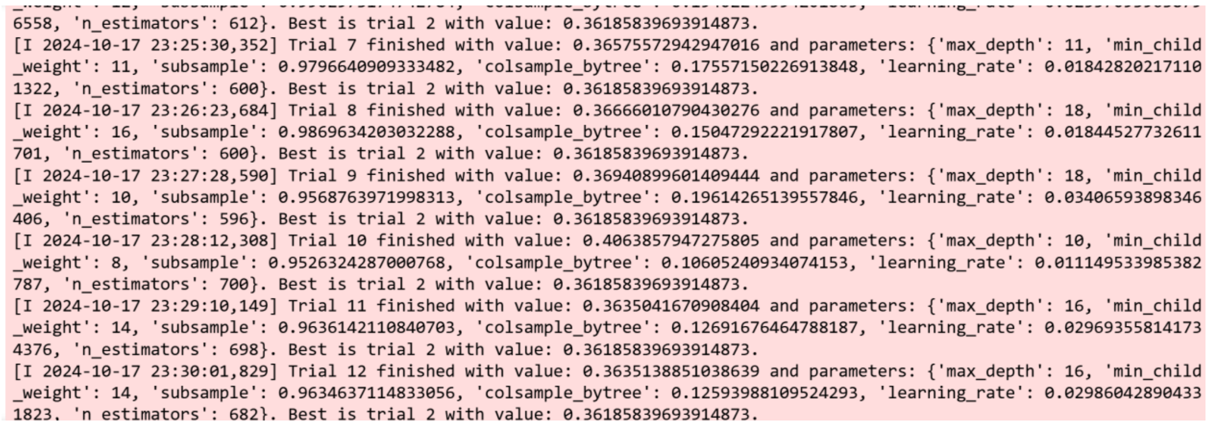
For random search and grid search I have tried Pararell and Serial search method. In a sequential grid search, the process involves testing one parameter at a time, optimizing it before moving to the next parameter. This approach is generally was faster, as it quickly narrows down the search to better parameter ranges. However, it has the disadvantage of not accounting for dynamic interactions between parameters, since it evaluates each one independently before. On the other hand, a parallel grid search explores multiple combinations of parameters simultaneously, allowing it to capture interactions between different parameters more effectively. But it was way slower than the sequential approach because it evaluates a wider range of combinations, especially in large parameter spaces.

a) Gridsearch + cross validation: Grid search takes so much time to train so I have changed to random search.

b) RandomSearch + cross validation: Since Randomsearch was not fine enough to critically minimize logloss, I have searched other method such as optuna library

c) Optuna library : Optuna uses a history record of trials to determine which hyperparameter values to try next. Using this data, it estimates a promising area and tries values in that area. Optuna then estimates an even more promising region based on the new result. It repeats this process using the history data of trials completed thus far. Specifically, it employs a Bayesian optimization algorithm called Tree-structured Parzen Estimator. So the I have used Optuna to find overall hyperparameter that performs best and did refine and narrow down the grid and search again. The hyperparameter I found **best was below** for my XGBOOST model.

```
final_param={'max_depth': 12, 'min_child_weight': 8, 'subsample':
0.9665601547562184, 'colsample_bytree': 0.12355113792160276,
'learning_rate': 0.0262390732061374, 'n_estimators': 687}
```



```
6558, 'n_estimators': 612}. Best is trial 2 with value: 0.36185839693914873.
[I 2024-10-17 23:25:30,352] Trial 7 finished with value: 0.36575572942947016 and parameters: {'max_depth': 11, 'min_child
_weight': 11, 'subsample': 0.9796640909333482, 'colsample_bytree': 0.17557150226913848, 'learning_rate': 0.01842820217110
1322, 'n_estimators': 600}. Best is trial 2 with value: 0.36185839693914873.
[I 2024-10-17 23:26:23,684] Trial 8 finished with value: 0.36666010790430276 and parameters: {'max_depth': 18, 'min_child
_weight': 16, 'subsample': 0.9869634203032288, 'colsample_bytree': 0.15047292221917807, 'learning_rate': 0.01844527732611
701, 'n_estimators': 600}. Best is trial 2 with value: 0.36185839693914873.
[I 2024-10-17 23:27:28,590] Trial 9 finished with value: 0.36940899601409444 and parameters: {'max_depth': 18, 'min_child
_weight': 10, 'subsample': 0.9568763971998313, 'colsample_bytree': 0.19614265139557846, 'learning_rate': 0.03406593898346
406, 'n_estimators': 596}. Best is trial 2 with value: 0.36185839693914873.
[I 2024-10-17 23:28:12,308] Trial 10 finished with value: 0.4063857947275805 and parameters: {'max_depth': 10, 'min_child
_weight': 8, 'subsample': 0.9526324287000768, 'colsample_bytree': 0.10605240934074153, 'learning_rate': 0.011149533985382
787, 'n_estimators': 700}. Best is trial 2 with value: 0.36185839693914873.
[I 2024-10-17 23:29:10,149] Trial 11 finished with value: 0.3635041670908404 and parameters: {'max_depth': 16, 'min_child
_weight': 14, 'subsample': 0.9636142110840703, 'colsample_bytree': 0.12691676464788187, 'learning_rate': 0.02969355814173
4376, 'n_estimators': 698}. Best is trial 2 with value: 0.36185839693914873.
[I 2024-10-17 23:30:01,829] Trial 12 finished with value: 0.3635138851038639 and parameters: {'max_depth': 16, 'min_child
_weight': 14, 'subsample': 0.9634637114833056, 'colsample_bytree': 0.12593988109524293, 'learning_rate': 0.02986042890433
1823, 'n_estimators': 682}. Best is trial 2 with value: 0.36185839693914873.
```

Figure 8 - Optuna Training

For Training I have used my DELL laptops own RTX 4070 **GPU** – cuda to boost my training speed.

Pseudo Label:

To make the test log loss minimized, I have tried using pseudo label. I have selected the best robust test accuracy model I had – 93% with stacked model and XGBOOST ensemble model. I used them to label the test dataset. After that I have used a 90% of threshold of probability to select the most "reliable"(`confidence Threshold)` and data label that I can believe(I have tried 85%~99%) but 90 % threshold was reasonable. So mostly 3000~6000 data could be additionally act as my new training data. So I have used augmented training data with pseudo label. Which it resulted **15000→22009** dataset. Below are code for making pseudo label dataframe. However the logloss was significantly smaller than our all of the models, there was a problem.

Our multi-model stacking model logloss: **3.759343**

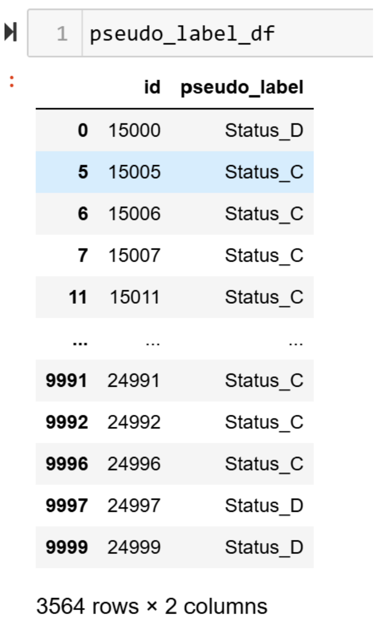
Best score for XGBoost ensemble model for non-pseudo label data with feature augmentation: **0.3657089375934389(best)**

XGBoost with pseudo label data: **-0.2860589699142558**

As you can see, pseudo label data seems like perform "significantly" well on test and train data, however, it is because we have already labeled train and the test data. But when I submitted to Kaggle the pseudo data label model, it actually performed worse than no-pseudo labeled training data version. I think the noise was to big compared with the actual test label.

```
import pandas as pd
import numpy as np
test_data=pd.read_csv("test.csv")
#make pseudo labels:

confident_ids=[]
predicted= pd.read_csv('Tony_final_ensemble_ver3.csv') #best log
loss currently 3.5


print(predicted.columns[1:])

confidence_threshold = 0.90 # this is the confidence Threshold
# confident_indices = np.max(predicted[predicted.columns[1:]],
axis=1) > confidence_threshold
confident_indices = predicted[predicted.columns[1:]].max(axis=1) >
confidence_threshold

confident_ids = predicted.loc[confident_indices, 'id'].tolist()

#create pseudo labels highest probability for each confident ID
pseudo_labels = predicted.loc[confident_indices,
predicted.columns[1:]].idxmax(axis=1)

# Combine the IDs

pseudo_label_df = pd.DataFrame({
    'id': predicted.loc[confident_indices, 'id'],
    'pseudo_label': pseudo_labels
})

#display or use the pseudo-label
print(pseudo_label_df)

confident_ids
print(len(confident_ids))
```

```
pseudo_label_df
```



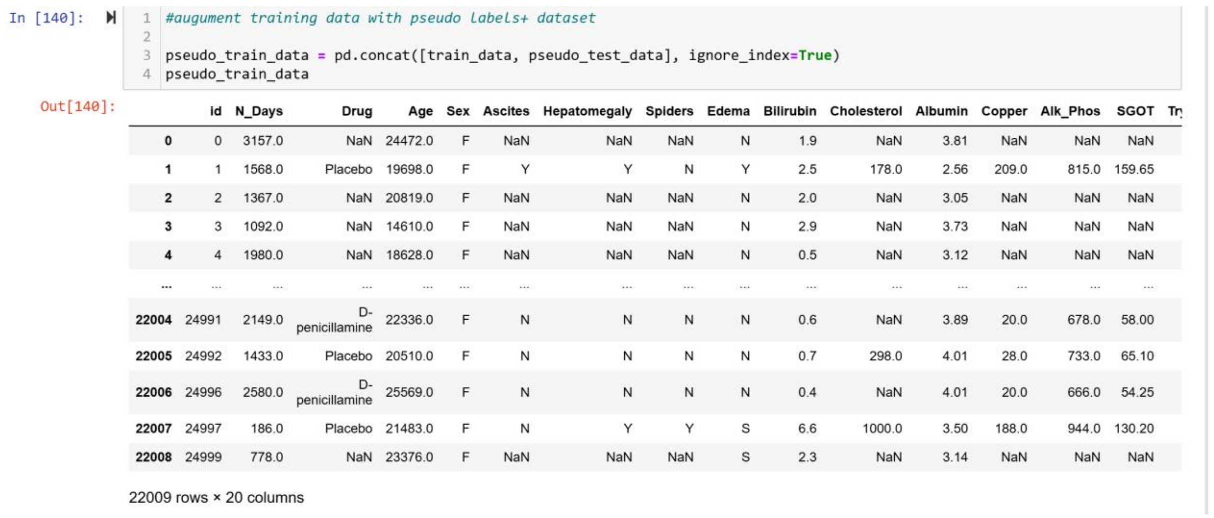
```
pseudo_label_df
```

:

	id	pseudo_label
0	15000	Status_D
5	15005	Status_C
6	15006	Status_C
7	15007	Status_C
11	15011	Status_C
...	...	...
9991	24991	Status_C
9992	24992	Status_C
9996	24996	Status_C
9997	24997	Status_D
9999	24999	Status_D

3564 rows × 2 columns

Figure 9 - Pseudo label with 95% confidence threshold



In [140]:

```
#augument training data with pseudo labels+ dataset

pseudo_train_data = pd.concat([train_data, pseudo_test_data], ignore_index=True)
pseudo_train_data
```

Out[140]:

	id	N_Days	Drug	Age	Sex	Ascites	Hepatomegaly	Spiders	Edema	Bilirubin	Cholesterol	Albumin	Copper	Alk_Phos	SGOT	Tr
0	0	3157.0	NaN	24472.0	F	NaN	NaN	NaN	N	1.9	NaN	3.81	NaN	NaN	NaN	
1	1	1568.0	Placebo	19698.0	F	Y	Y	N	Y	2.5	178.0	2.56	209.0	815.0	159.65	
2	2	1367.0	NaN	20819.0	F	NaN	NaN	NaN	N	2.0	NaN	3.05	NaN	NaN	NaN	
3	3	1092.0	NaN	14610.0	F	NaN	NaN	NaN	N	2.9	NaN	3.73	NaN	NaN	NaN	
4	4	1980.0	NaN	18628.0	F	NaN	NaN	NaN	N	0.5	NaN	3.12	NaN	NaN	NaN	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
22004	24991	2149.0	D-penicillamine	22336.0	F	N	N	N	N	0.6	NaN	3.89	20.0	678.0	58.00	
22005	24992	1433.0	Placebo	20510.0	F	N	N	N	N	0.7	298.0	4.01	28.0	733.0	65.10	
22006	24996	2580.0	D-penicillamine	25569.0	F	N	N	N	N	0.4	NaN	4.01	20.0	666.0	54.25	
22007	24997	186.0	Placebo	21483.0	F	N	Y	Y	S	6.6	1000.0	3.50	188.0	944.0	130.20	
22008	24999	778.0	NaN	23376.0	F	NaN	NaN	NaN	S	2.3	NaN	3.14	NaN	NaN	NaN	

22009 rows × 20 columns

Figure 10 - After adding Pseudo data

Tricks and Tips

I have tried using tricks of improving logloss. The use of the logarithm provides extreme punishments for being both confident and wrong. In the worst possible case, a prediction that something is true when it is actually false will add an infinite amount to your error score, it would be much better to keep our probabilities between 0.05–0.95(or 0.01~0.99)so that we are never very sure about our prediction. In this case, we won’t see the massive growth of an error function. However it turns out that didn't perform well on our small dataset of ours, but it was fun to try.

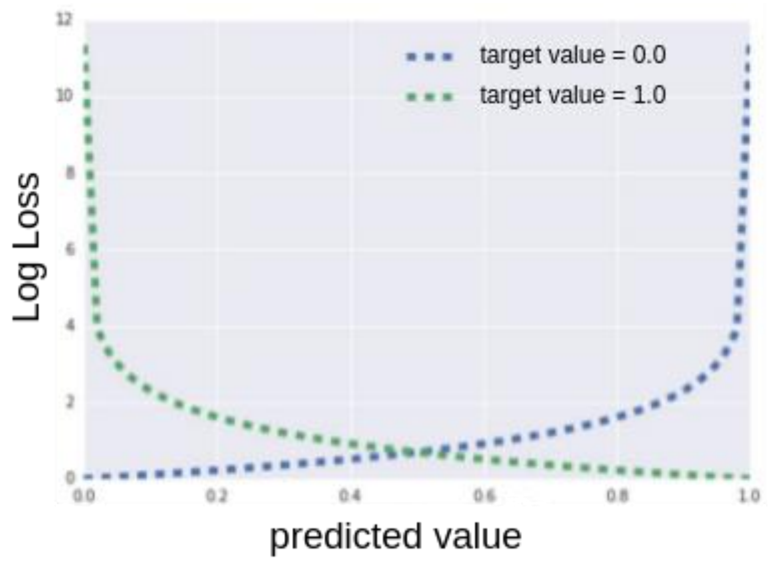


Figure 11 - Log loss clip trick