Final Report for Predict Student Performance From Game Play competition

Introduction

Predict Student performance from Game Play is a competition on kaggle, which allows participants access to current data that is related to game-based learning. “Most game-based learning platforms do not sufficiently make use of knowledge tracing to support individual students. Knowledge tracing methods have been developed and studied in the context of online learning environments and intelligent tutoring systems. But there has been less focus on knowledge tracing in educational games”. By data analysis, developers can have more information to improve the game-based learning platform.

In our simulation, we used the training data to shape the data and train our model to predict unknown results in the testing data. “For each <session_id>_<question #>, [we] are predicting the correct column, identifying whether [we] believe the user for this particular session will answer this question correctly, using only the previous information for the session”. The data types that is provided in the training data is shown below:

- **session_id** - the ID of the session the event took place in
- **index** - the index of the event for the session
- **elapsed_time** - how much time has passed (in milliseconds) between the start of the session and when the event was recorded
- **event_name** - the name of the event type
- **name** - the event name (e.g. identifies whether a notebook_click is is opening or closing the notebook)
- **level** - what level of the game the event occurred in (0 to 22)
- **page** - the page number of the event (only for notebook-related events)
- **room_coor_x** - the coordinates of the click in reference to the in-game room (only for click events)
- **room_coor_y** - the coordinates of the click in reference to the in-game room (only for click events)
- **screen_coor_x** - the coordinates of the click in reference to the player’s screen (only for click events)
- **screen_coor_y** - the coordinates of the click in reference to the player’s screen (only for click events)
- **hover_duration** - how long (in milliseconds) the hover happened for (only for hover events)
- **text** - the text the player sees during this event
- **fqid** - the fully qualified ID of the event
- **room_fqid** - the fully qualified ID of the room the event took place in

- **text_fqid** - the fully qualified ID of the
- **fullscreen** - whether the player is in fullscreen mode
- **hq** - whether the game is in high-quality
- **music** - whether the game music is on or off
- **level_group** - which group of levels - and group of questions - this row belongs to (0-4, 5-12, 13-22)

Use of Function

Data Engineering

In train_lables.csv file, the session_id column content consists of 2 parts: session number and question number. Therefore, we first split session number and question apart by code below:

```
labels['session'] = labels.session_id.apply(lambda x: int(x.split('_')[0]) )
labels['q'] = labels.session_id.apply(lambda x: int(x.split('_')[-1][1:]) )
```

Then we choose appropriate categorical features and numerical features:

```
cat_features = ['event_name', 'name','fqid', 'room_fqid', 'text_fqid']
num_features =
['elapsed_time','level','page','room_coor_x','room_coor_y',
'screen_coor_x', 'screen_coor_y', 'hover_duration']
```

Another **data_engineer()** function that we implemented is inspired from an open source code online, which you can see in the link in the code. The function is shown below:

```
def feature_engineer(dataset_df):
    dfs = []
    for c in cat_features:
        tmp = dataset_df.groupby(['session_id','level_group'])[c].agg('nunique')
        tmp.name = tmp.name + '_nunique'
        dfs.append(tmp)
    for c in num_features:
        tmp = dataset_df.groupby(['session_id','level_group'])[c].agg('mean')
        dfs.append(tmp)
    for c in num_features:
        tmp = dataset_df.groupby(['session_id','level_group'])[c].agg('std')
        tmp.name = tmp.name + '_std'
        dfs.append(tmp)
    dataset_df = pd.concat(dfs,axis=1)
    dataset_df = dataset_df.fillna(-1)
    dataset_df = dataset_df.reset_index()
    dataset_df = dataset_df.set_index('session_id')
    return dataset_df
```

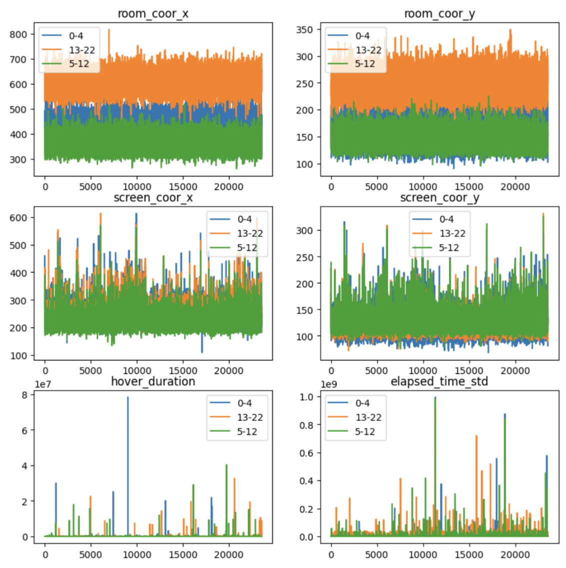
It first initializes an empty list called dfs. It then loops through the categorical features in the input dataset_df, and groups the dataframe by session_id and level_group. It then aggregates each category by the number of unique values in that category for each session and level group combination. It renames the resulting series with _nunique appended to its original name and appends it to the dfs list.

The function then loops through the numerical features in dataset_df. For each numerical feature, it again groups the dataframe by session_id and level_group and calculates the mean and standard deviation of the feature for each group. It renames the resulting standard deviation series with _std appended to its original name and appends both the mean and standard deviation series to the dfs list.

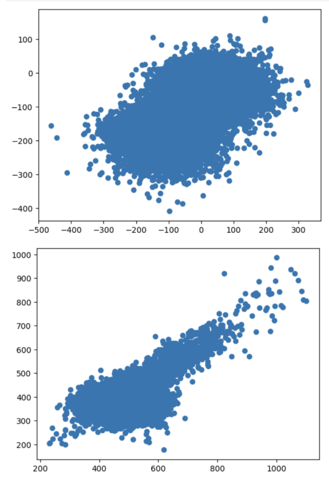
The dfs list is then concatenated along the columns axis (axis=1) to form a new dataframe, which is filled with -1 wherever there are missing values (NaN) and reset with a new index. Finally, the index is set to session_id and the resulting dataframe is returned.

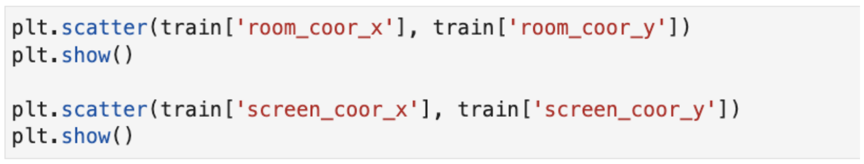
Overall, this function appears to be creating new features from the input dataset_df by aggregating the categorical and numerical features over different combinations of session_id and level_group. The resulting feature-engineered dataset can be used for training a machine learning model. We applied this function on training data.

Visualization of data



A 3x2 grid of subplots is created, with each subplot showing the standard deviation of one numerical feature across different 'level_group' values.
Each subplot represents the standard deviation of a specific feature, grouped by the 'level_group' column.
The x-axis represents the data points (in sequential order) within each level group.
The y-axis represents the standard deviation values for the respective features.
The line plots for different level groups are displayed with different colors and legends.
The Second Part outside the loop create scatter plots for 'room_coor_x' vs 'room_coor_y' and 'screen_coor_x' vs 'screen_coor_y' respectively.



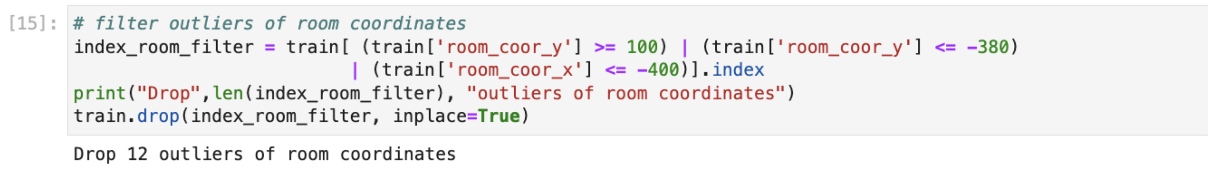


```
plt.scatter(train['room_coor_x'], train['room_coor_y'])
plt.show()

plt.scatter(train['screen_coor_x'], train['screen_coor_y'])
plt.show()
```

After get the data, we can have a basic idea how the data distributes and what are the **outliers**

Filter outliers



```
[15]: # filter outliers of room coordinates
      index_room_filter = train[ (train['room_coor_y'] >= 100) | (train['room_coor_y'] <= -380)
                                  | (train['room_coor_x'] <= -400)].index
      print("Drop",len(index_room_filter), "outliers of room coordinates")
      train.drop(index_room_filter, inplace=True)

      Drop 12 outliers of room coordinates
```

This code filters and removes the outliers in the 'train' DataFrame based on 'room_coor_x' and 'room_coor_y' values.

It first creates a boolean mask called 'index_room_filter' that identifies the rows in the DataFrame where either 'room_coor_y' is greater than or equal to 100, less than or equal to -380, or 'room_coor_x' is less than or equal to -400. These conditions are used to filter out the outliers in the room coordinates.

The code then prints the number of outliers found and removes these outliers from the 'train' DataFrame using the drop() function with the inplace=True parameter, which modifies the DataFrame directly by dropping the specified rows.

Train test Split

```
def split_dataset(dataset, test_ratio=0.20):
    USER_LIST = train.index.unique()
    split = int(len(USER_LIST) * (1 - 0.20))
    return dataset.loc[USER_LIST[:split]],
dataset.loc[USER_LIST[split:]]

train_x, valid_x = split_dataset(train)
```

This code defines a function called split_dataset that takes in a dataset and a test ratio, and returns a tuple containing the training set and validation set.

The function first defines a list called USER_LIST by extracting the unique values of the index column of the train dataset. It then calculates the index at which the split between the training and validation sets should occur. Specifically, it calculates the integer value of len(USER_LIST) * (1 - 0.20), where 0.20 is the test ratio, and len(USER_LIST) is the total number of unique users in the dataset. This index corresponds to the position of the last user in the training set.

The function then returns two dataframes: the first contains the data from the first user up to the user at the calculated split index, while the second contains the data from the user at the split index up to the last user in the dataset.

Finally, the last line of code calls the split_dataset function with the train dataset as input and stores the resulting training and validation sets in two separate variables called train_x and valid_x. The test_ratio parameter has a default value of 0.20, which means that if split_dataset is called without specifying a value for test_ratio, it will use 0.20 as the default test ratio.

XGB model

Threshold method

The threshold method is a technique used in binary classification problems to determine the optimal cutoff point for making decisions between two classes. It helps in adjusting the decision boundary based on predicted probabilities, improving the overall performance of the classification model.

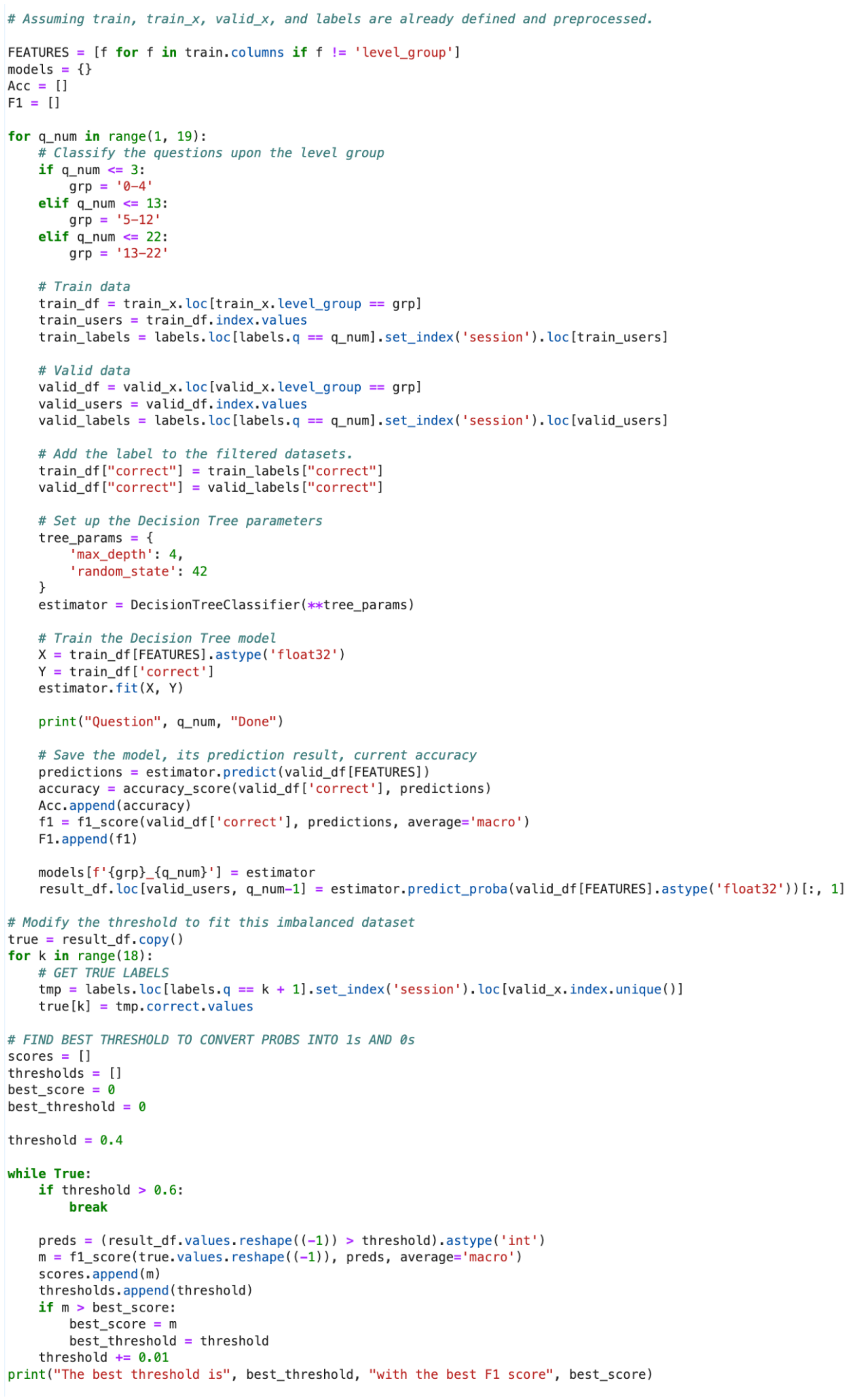
In the context of the our code, in the **TREE_TEST** and **XGB_TEST** parts, the threshold method can be employed to decide the probability threshold for classifying whether a student answered a question correctly or incorrectly based on the predicted probabilities from the trained XGBoost classifier.

The threshold method works by setting a cutoff value, usually ranging between 0 and 1, which is used to separate the predicted probabilities into two classes. For instance, if the threshold is set to 0.5, any predicted probability above 0.5 would be classified as a correct answer (1), while probabilities below 0.5 would be classified as an incorrect answer (0).

To determine the best threshold, multiple cutoff values can be tested, and the one that yields the best performance metrics, such as accuracy, precision, recall, or F1-score, can be chosen. In this specific code, the accuracy and F1-score are used as performance metrics.

Once the best threshold is identified, it can be applied to the final classification of new, unseen data. This helps in improving the model's performance by fine-tuning the decision boundary between the two classes.

Tree - Method covered in class



```python
# Assuming train, train_x, valid_x, and labels are already defined and preprocessed.

FEATURES = [f for f in train.columns if f != 'level_group']
models = {}
Acc = []
F1 = []

for q_num in range(1, 19):
    # Classify the questions upon the level group
    if q_num <= 3:
        grp = '0-4'
    elif q_num <= 13:
        grp = '5-12'
    elif q_num <= 22:
        grp = '13-22'

    # Train data
    train_df = train_x.loc[train_x.level_group == grp]
    train_users = train_df.index.values
    train_labels = labels.loc[labels.q == q_num].set_index('session').loc[train_users]

    # Valid data
    valid_df = valid_x.loc[valid_x.level_group == grp]
    valid_users = valid_df.index.values
    valid_labels = labels.loc[labels.q == q_num].set_index('session').loc[valid_users]

    # Add the label to the filtered datasets.
    train_df["correct"] = train_labels["correct"]
    valid_df["correct"] = valid_labels["correct"]

    # Set up the Decision Tree parameters
    tree_params = {
        'max_depth': 4,
        'random_state': 42
    }
    estimator = DecisionTreeClassifier(**tree_params)

    # Train the Decision Tree model
    X = train_df[FEATURES].astype('float32')
    Y = train_df['correct']
    estimator.fit(X, Y)

    print("Question", q_num, "Done")

    # Save the model, its prediction result, current accuracy
    predictions = estimator.predict(valid_df[FEATURES])
    accuracy = accuracy_score(valid_df['correct'], predictions)
    Acc.append(accuracy)
    f1 = f1_score(valid_df['correct'], predictions, average='macro')
    F1.append(f1)

    models[f'{grp}_{q_num}'] = estimator
    result_df.loc[valid_users, q_num-1] = estimator.predict_proba(valid_df[FEATURES].astype('float32'))[:, 1]

# Modify the threshold to fit this imbalanced dataset
true = result_df.copy()
for k in range(18):
    # GET TRUE LABELS
    tmp = labels.loc[labels.q == k + 1].set_index('session').loc[valid_x.index.unique()]
    true[k] = tmp.correct.values

# FIND BEST THRESHOLD TO CONVERT PROBS INTO 1s AND 0s
scores = []
thresholds = []
best_score = 0
best_threshold = 0

threshold = 0.4

while True:
    if threshold > 0.6:
        break

    preds = (result_df.values.reshape((-1)) > threshold).astype('int')
    m = f1_score(true.values.reshape((-1)), preds, average='macro')
    scores.append(m)
    thresholds.append(threshold)
    if m > best_score:
        best_score = m
        best_threshold = threshold
    threshold += 0.01
print("The best threshold is", best_threshold, "with the best F1 score", best_score)
```

A loop is used to train a separate Decision Tree model for each of the 18 questions (q_num from 1 to 18). The level_group of each question is determined based on its q_num, and training and validation DataFrames are created by filtering the train_x and valid_x DataFrames based on the level_group. Labels are assigned for both the training and validation of DataFrames.

A Decision Tree Classifier is then instantiated with a maximum depth of 4 and a random_state of 42, which ensures consistent results across different runs. The model is then trained using the features and the 'correct' column as the target variable. The trained model is saved in the 'models' dictionary with a key format of '{level_group}_{q_num}'.

Predictions are made on the validation dataset, and the accuracy and F1 scores are calculated and appended to the respective lists. The predicted probabilities are stored in the 'result_df' DataFrame.

After training all the models, the best threshold for converting predicted probabilities into binary predictions (1s and 0s) is determined by iterating through possible thresholds between 0.4 and 0.6 and choosing the one that results in the highest F1 score. The F1 score is used here because it balances precision and recall, which is especially useful for imbalanced datasets.
However, under this part, we got f1-macro like this:
The best threshold is 0.5900000000000002 with the best F1 score 0.7033760158724128

The result is not as good as in XGB - we are going to introduce it later.

XGB - Method we do research

XGB stands form extreme gradient boosting, it is a more extreme approach of a gradient boosting tree algorithm. It’s fundamental is based on a decision tree. The one interpretation we used in this project is classification.

The algorithm behind the eXtreme gradient boosting is a technique to combine weak learners to form strong learners. It is an excellent model to use when dealing with data in the train.csv. The whole idea is to minimize the training loss by iterative through a decision tree, then combine these tree through a weighted sum to make predictions.

The gradient boosting is work as below:

- Initialization: start with a initial model that predict a constant value, it is usually the average of majority class for classification.
- Then for each iteration:
  - Calculate the gradient and second derivative of the loss function with respect to the current model for each training instance.
  - Build a new decision tree predicts gradient and second derivative of the loss function instead of the target value in the initialization.
  - Determine a optimal weight for the new decision tree, a weight that minimize the loss function and combine with the previous decision tree.
  - Stop the algorithm after the performance of model stop improving.

The extreme gradient boosting have improved several part of the above regular gradient boosting algorithm, such that

- The XGB introduce L1 and L2 regularization term in the loss function, prevent overfitting by penalize complex model.
- It use column block to penalize the model, which is a compressed memory sufficient model. The task can be done across multiple CPU core.
- XGB can handle sparisity data and missing value efficiently, by learning optimal default direction for missing values during the tree construction.
- Allow early stoping.

In our model, we have choose many hyperparameters to allow the more efficient classification of the data,

```
xgb_params = {
   'objective' : 'binary:logistic',
   'eval_metric':'logloss',
   'learning_rate': 0.05,
   'max_depth': 4,
   'n_estimators': 1000,
   'early_stopping_rounds': 40,
   'tree_method':'hist',
   'subsample':0.8,
   'colsample_bytree': 0.4}
```

Objective: this is a binary task so that logistic is using.
Eval_metric: measures the performance of a classification model where the predicted output is a probability value between 0 and 1.
Learning_rate: control the step size the model update in each round, for our model is 0.05
Max_depth : the maxim depth of each decision tree, if number is too large or too small will lead to insufficient prediction.
N_estimators: How many round the tree will be boosted.

Early_stopping_rounds: after 40 round of boosting, the tree is allow to stop if there is no effective improvement.
Tree_method: creates histograms of the continuous features to enable faster tree construction and uses less memory
Subsample: controls the fraction of the training data to be used for each boosting round
Colsample_bytree: controls the fraction of the features to be used for each tree construction.

The initial result we get form XGB classifier is 0.499939616506168 as F1 macro for average of all 18 questions in prediction. Then we think it is might because of classifying imbalanced data, such that classify output to 0 or 1 by the mark of 0.5 is inefficient. Then we used a method to modify the threshold, that better evaluate the imbalancing data, that improve the performance. The final result we get is 0.7149008488782065 with threshold 0.5900000000000002, which improve the performance of the model by around 0.2 on the F1 macro score.

By using the XGB classifier model, it enable us to achieve high performance with training data, at the same time limit the memory usage, as one of the major advantage of XGB is it allow early stop and more memory sufficient.

Conclusion(result & lessons)

By comparing the performance of Decision Tree and XGboost model, we decide to use XGboost for prediction. The F1 score result of 0.7149008488782065 indicates that the model's performance is decent. This means our prediction is helpful if the game-based learning platform decides to use our data for reference to improve the platform performance. However, the F1 score of 0.71 is still not in the range between 0.8 to 0.9, where F1 score can be considered as really good. The result can be improved by using larger dataset, or the researchers can collect other factors of data which might help the classification, for instance, age might be an important variable, but it is not collected in the current dataset.

During the process of exploring reforming data, trying out different models, and learning and researching on models like XGboost that haven't been covered in class, our team has learned a lot about the XGboost model and worked diligently to get the best result out of the model we chose. Even though the project is not easy, our hard work and teamwork supported us to get the result we have now.

Credit

Our group worked together on every category of code below based on a frame that one of us set up, we also tried multiple models individually, but we are only keeping Tree model and XG boost model as the 2 main models for consideration:
Data Engineering and Data Visualization: Jady Huang
Tree model: Hang Liao
XGB: John Xie

We also did great work on collaboration on this report, see our outline below:

Intro//Jady Finished

- describe the project(purpose, dataset)
- Data type

Use of Function

- Feature engineering//Jady Finished
- Train test split//Jady Finished
- XGB model*John Xie Finished
- Threshold method //Hang Finished
- Filter outliers //Hang Finished

Visualization of data//Hang Finished

- What graph represent

Method covered in class: Tree //Hang finished
-Tree_Result(Worse than
The best threshold is 0.5900000000000002 with the best F1 score 0.7033760158724128

Methods that not covered in class: xgboost //John Xie Finsihed

1. Theory (maybe provide some graph)
2. Reason of using this(why this is better than regular tree model, compare result)

xgboost_Result //John Xie Finished

1. Explain parameters below
2. outcome(F1 macro)
3. Explain the F1 macro, how it explains data why is this significant
4. Advantage of xgboost

Conclusion//Jady Finished

1. How the result contribute the the purpose of the competition
2. What we learn from the project

Teamwork//Jady Finished
What does each group member do in this project