ML Final Report

Mr. O'clock insanely practices

December 22, 2024

1 Introduction

This final project is aim to use machine learning to predict HTMLB outcome. We have explore through many algorithms and data preprocess, and result in 0.58989 accuracy in predicting same season outcome(stage1) and 0.55473 in predicting a whole new season(stage2). We are going to break it down what have we done with every model and preprocess and what are the results.

2 Data Preprocess

Dealing with NAN

There are many NAN in all kinds of data. To deal with them, we can either fill in 0, fill in mean of other non-NAN rows, or fill in mean of the data of same team.

Team Attribute and Pitcher Attribute

We try to add 60 boolean columns: is_home_team_{team name} and is_away_team_{team name} which indicate that if home team or away team equal to team name. Same method can be used on pitcher name.

However we try it on random forest an observed that these attribute may have low importance. We guess that perhaps this kind of data is too scatter to make it important

Alternatively, we can labelize string columns, this may work in algorithm based on decision tree.

Drop std and skew

Since we think std and skew has little relationship with winning or lose. We try to drop all of those column.

The result is that there are no difference in accuracy. However, training speed is faster using this data.

3 Models

KNN

PLA

Linear Regression

Logistic Regression

Logistic regression is a machine learning technique that predicts the probability of an outcome based on the weighted contribution of predictors. The sigmoid function ensures the probabilities are valid, and the coefficients are adjusted during training to maximize how well the model explains the observed data.

This is one of the very first models we have tried on, and the best one among all trained logitic regression models used the parameters as follows:

```
model = LogisticRegression(solver='liblinear', max_iter=1000, random_state=42)
```

where the logistic regression model is included in the package *sklearn.linear_model*.

We split the provided training data into 80% for training and 20% for validation. The above model could reach a training accuracy of 63.42% and validation accuracy 54.52%. The public score we got on kaggle is 56.9% for stage 1, and 57.64% for stage 2. For private, we got an accuracy of 57.1% for stage 1 and 53.85% for stage 2. In stage 2, a larger gap emerges between the public and private scores, which we attribute to using the same training technique for both stages. We did not account for the influence of time, an especially critical factor in stage 2, which should have been incorporated into our model. Overall, it's an efficient model but assumes a linear relationship which might not always be the case especially in complex problems like this.

SVM

Choosing the best model in SVM

In this section, we implement SVM model to predict the MLB result. SVM a.k.a. support vector machine can select multiple parameters to decide the data transformation method and hyperplane to seperate the data. To choose the best model, we use grid search by following parameters

```
param_grid = {
'C': [0.1, 1, 10, 100],
'gamma': ['scale', 'auto', 0.1, 1],
'kernel': ['rbf', 'linear', 'poly']
}
```

After the bruteforce searching, the result turns out that (kernel : linear, C=1.0, gamma : scale) and (kernel : rbf, C=1.0, gamma : scale) have the best performance which results in training data accuracy around 0.58 and validation data accuracy about 0.56 with 5-folds validations. After submit the test results to the kaggle in stage 1 and stage 2, the first parameters combination turns out to be in stage 1 : pulic 0.56584, private 0.57142, and in stage 2 : public 0.50830, private 0.50980, and the second combinations results in stage 1 : public 0.57101, private 0.57110, stage 2 : public 0.57558, private 0.54656.

Linear Kernel SVM

The performance of the linear kernel SVM is relatively stable when comparing public and private test results. However, when the trained model is applied to stage 2 data, the accuracy drops significantly from approximately 0.57 to around 0.51. This suggests that the validation method used for the linear kernel may need improvement to better handle stage 2 predictions. We propose that since the RBF kernel achieves a stable 0.57 accuracy in stage 2 and the linear kernel is inherently a simpler model, overfitting is unlikely to be the primary issue. Instead, the linear model may be too simplistic to accurately predict stage 2 data, which spans the entire year, particularly with an unsuitable validation approach. Despite the instability in performance, the linear kernel's training speed is highly efficient, especially when the regularization parameter C is set below 100.

RBF Kernel SVM

The RBF kernel SVM demonstrates strong and stable accuracy, with only a slight drop in private test results during stage 2. With proper regularization, it effectively models and separates the data while minimizing overfitting, making it a reliable choice for this task. However, compared to the linear kernel, the RBF kernel requires more training time and is more prone to overfitting in certain scenarios. Additionally, the RBF kernel projects the data into an infinite-dimensional space, which reduces the model's interpretability. As a result, while it often achieves excellent prediction performance, it provides limited insight into the underlying patterns learned during training.

SVM with bagging and blending

Since both parameters combinations have the similar prediction accuracy in stage 1 public tests, we come up with an idea to combine those parameters combinations by bagging and blending with SVM. When it comes to bagging, we use bootstrapping to generate different model with kernel=linear and

kernel=rbf by selecting different values in the maximum number of features (abbreviated as max_feature which can increase the diverity of models), the maximum number of samples (maximum_number of train samples abbreviated as max_samples) and the number of estimator (the number of different hypothesis generated from bagging).

This time we utilize optuna package to help us find the best parameters by following commands (Since optuna can tune the parameters in more precision ways, we also take C and gamma into considerations to expect better outcomes).

```
C = trial.suggest_loguniform('C', 0.1, 100.0)
gamma = trial.suggest_categorical('gamma', ['scale', 'auto'])
n_estimators = trial.suggest_int('n_estimators', 10, 50)
max_samples = trial.suggest_float('max_samples', 0.5, 1.0)
max_features = trial.suggest_float('max_features', 0.3, 1.0)
```

After optimization with Optuna, the submission result on stage 1 is public 0.57198, private 0.57887 and on stage 2 is public 0.54900, private 0.53349 . Compared to the results before applying bagging, the linear regression model achieved its best performance at 0.57142, while the RBF kernel achieved its best performance at 0.57110, both considering stage 1, stage 2, private, and public datasets. The application of bagging and blending outperformed the individual linear regression and RBF kernel models, demonstrating the advantages of combining multiple hypotheses. However, the prediction performance in stage 2 remains suboptimal.

We suggest that this issue may stem from mispredictions caused by the linear kernel SVM, which exhibited significantly poor performance in stage 2. Overall, the motivation to combine the two models due to their similar performance in stage 1 is validated by the improved results obtained through bagging and blending, as expected.

That said, the process of bagging and blending is computationally intensive, as it requires generating multiple hypotheses using the linear kernel SVM and RBF kernel SVM models. Additionally, parameter selection demands a considerable amount of time. Nonetheless, while the computational cost increases significantly for large datasets, the added diversity from bagging and blending can lead to more precise predictions.

Package references

```
sklearn, optuna
```

Random Forest(Best)

Random Forest is an ensemble learning method that builds multiple decision trees using bootstrap sampling and random feature selection, then combines their predictions to improve accuracy and reduce overfitting. It is robust, handles high-dimensional data well, and is effective for both classification and regression tasks.

This model could be said as the one that performed the best during the kaggle competition, and the best one among all trained random forest models used the parameters as follows:

```
model = RandomForestClassifier(n_estimators=12000, random_state=1,
max_depth =7,min_samples_leaf=1,class_weight="balanced")
```

where the random forest model is included in the package *sklearn.ensemble*.

Here's an interesting story: we chose this set of parameters because, during class, the instructor shared his experience at the KDD Cup, where he used 12,000 estimators and a random state of 1 for Random Forest, inspiring us to try it. This configuration became our best model, achieving the highest accuracy. The choice of setting *max_depth* to 7 resulted from trial and error, as we found depth 6 led to underfitting, while depth 8 caused overfitting.

We split the provided training data into 80% for training and 20% for validation. The above model could reach a training accuracy of 76.5% and validation accuracy 53.9%. The public score we got on kaggle is 59.26% for stage 1, and 58.39% for stage 2. For private, we got an accuracy of 58.989% for stage 1 and 52.45% for stage 2. Similar to our findings in the logistic regression analysis, we observed a lower private score in stage 2, likely indicating overfitting to noise, which may be due to our failure to consider the significance of the time attribute and using the exact same data for training in the two stages. In short, Random Forest is a powerful model that excels with complex data (as in this case)

by combining the outcomes of decision trees but is computationally intensive due to the time required to build and aggregate multiple trees.

XGboost

XGboost is a machine learning algorithm package that based on GBDT, gradient boosting deciesion tree. We utilize optuna to help us select the best parameters by following ranges :

```
param = {
    "verbosity": 0,
    "objective": "multi:softmax",
    "num_class": 2,
    "eval_metric": "mlogloss",
    "max_depth": trial.suggest_int("max_depth", 1, 10),
    "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.3),
    "n_estimators": trial.suggest_int("n_estimators", 30, 200),
    "subsample": trial.suggest_float("subsample", 0.3, 1.0),
    "colsample_bytree": trial.suggest_float("colsample_bytree", 0.5, 1.0),
    "gamma": trial.suggest_float("gamma", 0, 10),
    "lambda": trial.suggest_float("lambda", 0, 10),
}
```

The validation method implemented in this case is first sort the date of given training data and then take the first 80% of the data in each each season as training data set and the remaining ones as validation test data (which is designed for stage 1 prediction).

Submit the prediction results obtained by training XGBoost with the best parameters selected by Optuna. The accuracy achieved in stage 1 is as follows: private score: 0.57725 and public score: 0.58683. In stage 2, the accuracy is private score: 0.54166 and public score: 0.58554.

The accuracy of XGBoost among all the algorithms we used ranks second, which was an unexpected outcome. Compared to the CatBoost algorithm, we initially expected CatBoost to deliver more accurate results since it natively handles string data and is also based on GBDT (Gradient Boosting Decision Trees). However, with data preprocessing that converts string information into a sparse table, XGBoost was able to incorporate this information effectively and produce precise predictions.

The performance of XGBoost between the two stages was stable in the public test, achieving an accuracy of 0.58 in both stages. Nonetheless, there was some fluctuation in the private test, where accuracy dropped from 0.57725 in stage 1 to 0.54166 in stage 2. We suggest that adopting a different validation method could improve stage 2 results and lead to more stable performance.

Compared to SVM bagging and blending methods, XGBoost is undoubtedly an efficient algorithm. While its performance is similar to CatBoost, which also uses GBDT, it is slightly slower than simpler models like PLA, Linear Regression, and Logistic Regression. However, selecting optimal hyperparameters for XGBoost can take several hours with tools like Optuna. Despite this, XGBoost's support for multithreading and GPU optimization ensures excellent scalability, enabling efficient training even on large datasets.

Package references

```
optuna, xgboost
```

CATboost

Catboost is an algorithm based on Gradient Boosting Tree, which is similar to XGboost and LightGBM. However, Catboost can deal with string data type directly, which is a very good characteristic for this problem since we think string attribute like team name or pitcher name are important for training and cannot be dropped.

In all of the following experiment, i use mean of column to replace numerical column's NAN, and use 'NAN' to replace all NANs in string column.

In the initial tries, we use parameters as following

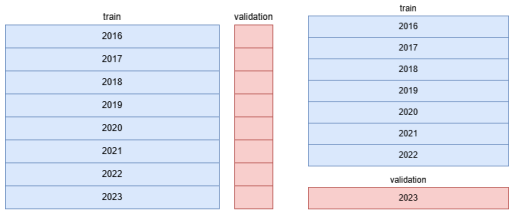
```
iterations = 500, depth = 5, learning_rate = 0.05, loss_function = 'Logloss'
```

This is our first algorithm that reaches 0.58 in public test. However, it is still not good enough to pass benchmark.

To find a better set of parameters, we use optuna to help us find it. And to mimic to stage1 / stage2 situation we will met, we use following different strategy to cut training and validation test for optuna.

stage1 Since we are predicting the last half season given each year's first half season, we use the last 0.2 of each season's data as validation data, and other as training data, and feed these to optuna to find the best parameters.

stage2 We are predicting a whole new year. Hence we take 2023's data as validation data, and other as training data, and feed these to optuna to find best parameters.



When using optuna, we search in following range

```
params = {
    "iterations": trial.suggest_int("iterations", 100, 1000),
    "depth": trial.suggest_int("depth", 4, 10),
    "learning_rate": trial.suggest_float("learning_rate", 1e-3, 0.3, log=True),
    "l2_leaf_reg": trial.suggest_float("l2_leaf_reg", 1, 10),
    "bagging_temperature": trial.suggest_float("bagging_temperature", 0, 1),
    "random_strength": trial.suggest_float("random_strength", 0.1, 10),
    "loss_function": "Logloss",
}
```

The best parameters lead to a 0.57985 accuracy in stage1. And 0.54493 in stage2. Which is not quite stable between two stages. Catboost is a very efficient model, which finish a single round of prediction in a minute. However, doing optuna will take hours to find best parameters (depends on rounds).

Package Reference

```
optuna, catboost
```

4 Conclusion