Aidan Liu & Arabel Rachman

Final Project

COE 379L

April - May 2025

ML Models on Teamfight Tactics Strategies

1. Introduction and project statement:

For this project we are using the data from Riot games regarding Teamfight Tactics (TFT). For a brief introduction for this game, this game is a more complex version of autochess, where players are in a lobby against 7 other players (8 counting the player). In each set of the game, there are unique champions (which can resemble pawns or bishops in normal chess) and traits each champion has. Each set contains a unique set of champions and traits. Each set is swapped out for a new set every 6 months or so as to not make the game stale. For the data that we are using we will be pulling from set 13. For a deeper understanding of how the game works, there are team compositions, item builds, and augments that also add more complexity into the game. For example, Caitlyn is a 5 cost Sniper trait and Enforcer trait. She is mostly attack damage carry so we must equip her with attack items. These items are tailored for specific champions so AD (attack damage) carries cannot use tank items effectively or AP (ability power) carries either. For the data we are pulling we are not including item win rates but it affects the total win rate for multiple compositions. Compositions are composed of different champions that share unique traits. For example, if we have 6 Enforcer traits with caitlyn, all 6 of those Enforcers will benefit from their activated trait granting bonus stats to all Enforcers. Each trait

will have unique benefits as well. For a team composition to work well there needs to be activated traits and also strong units. The summary of the goals of this project are: create a model that can show how strong a certain composition can be.

1.1 Problem definition

The problem that we are solving for is going to be the final lobby placement (1st - 8th) of the player given the player's finished board at the end of the game. The finished board data will consist of the list of champions that are on the field, each unit's star level, the active traits and the tiers, player level, and remaining gold. The model will use the data provided to statistically treat the final compositional in terms of strength and rate it. This model will not be using other player data in the same game or previous boards of the player in the same game to predict the final placement. This problem can be applicable to real life scenarios regarding the game as player's can watch another player and ask if they replicate the end game board, what placement they should expect to get. Building this model will allow us to measure the strength of a board without accounting for the popularity of the board, enabling data-driven tier lists and balancing insights for future balance changes.

1.2 Research Objectives

There are 2 objectives that this model is trained to do. Firstly, this model will quantify each composition's overall performance, this means that given a certain comp, what place will it achieve on average given the star level, cost, traits of the resulting board. For example, a 2 star 5

cost champion, a very high cap addition to a team, will on average boost the placement of the player significantly.

We will be training our model on thousands of Set 13 match records to accurately estimate the placement as a variable from 1-8. Secondly, we will aim to promote the interpretability of data. We will generate plots using Python's matplotlib/seaborn library for placement, level, and gold left. Using these plots will reveal which units, traits, or augments drive the current state of the meta, helping Riot balance the game and understand why certain comps perform better than others.

1.3 Scope of Contribution

Currently there's 3 main websites that scrap the data from tft games and transform them into results, namely MetaTFT, Tactics.tools, and TFTAcademy. These websites rank comps based on playtesting or raw win-rate leaderboards that mix together skill, matching making luck, and item variance. Our thesis is that these websites focus more towards reinforcement learning rather than pure win rate statistics. For example, if the website published an S tier comp, players will see these rankings and will try the comp out, inflating the win rate and playrate of the comp. By doing so, these websites make reinforcement learning the main priority for their models to work. Our contribution is a middle ground, a reproducible pipeline that uses data from Riot's public API. Our contribution model will engineer unit/trait vectors and benchmark multiple neural and gradient boosting models.

1.4 Project Statement

For this project, after cleaning up data from separate game accounts totaling to over 5000+ games, 40000+ units, and 35000+ traits, we will evaluate five key machine learning models and their results: Test RMSE, Random Forest RMSE, XGBoost RMSE, CatBoost Test RMSE, and Transformer Test RMSE.

First will be a cleaned up CSV file that is mostly accurate containing the TFT data (units, win rates, team comps, traits, etc.). This CSV will be read from our python script which has been sculpted to fit the CSV format that we had to download from Riot API. This included cleaning up Champion names, Augment names, and Trait names for over 100+ champions and traits.

Secondly we will have a trained model that can predict the placement of a composition. This model will be our best model that has the highest statistical probability of accuracy. We will test and hone this model to be the most accurate out of everything we have tried.

Thirdly, we will include our Jupyter Notebook folder, this will include our previous trials, mistakes, and successes. This will include many different models ranging from a simple Dense ANN to TabTransformers to Gradient Trees. This will also include our trials on ETL, EDA, model training, and overall evaluation.

Fourthly, we will be delivering a concise PDF report that summarizes the Project requirement sections: Introduction and project statement, Data sources and technologies used, Methods employed, Results, References.

2. Data Sources and Technologies Used:

As Teamfight Tactics is a game with constant updates, where each new set introduces completely different units and traits, we had to use the most up-to-date data for the specific set we are analyzing (Set 13). Since we could not find any thorough datasets for this set, we opted to get data directly stored by Riot Games using the RiotAPI. The RiotAPI allows us to have detailed and robust match data to train our models on. We created the dataset by making calls to the API, creating a dataframe out of the match data, and exporting it as a .CSV file. For each match, the data consists of the placement, the player level, the gold left available, the match length, the activated traits, and the player's units along with their levels. A sample of this data can be found within the .CSV file attached to the Github repository.

After creating a developer account on the Riot Portal, we can generate an API key that lasts for 24 hours. After this we exported this key to an environment variable called RIOT_API_KEY (export RIOT_KEY="") and used shell scripts to reference the key and get the player's PUUID (curl -H "X-Riot-Token: $RIOT_KEY" \"https://americas.api.riotgames.com/riot/account/v1/accounts/by-riot-id/{player name}/{player tag}"). Doing this will generate the player's unique PUUID ({"puuid":"Cta8nRQ-RIoCcQeTa1voONydYBCSF5DdqM5CbXEzhSJKRgFa09UwJxN1VsmfOurztHnfIfVC_q9SDw","gameName":"Eris3360","tagLine":"NA1"}%). This PUUID is important for resolving the player's identity as all TFT match endpoints are keyed through the PUUID which acts like a global unique player ID, and is used instead of the player's summoner name. This is because PUUID's are immutable and are created when the Riot account is created and cannot be changed unlike summoner name changes, thus we only need to resolve this once per player.

Next our python script data2.py takes the PUUID of the player and gets the 1000 most recent games that the player has played. Afterward the final match details for all 1000 games are dumped into a single CSV file which includes, champions, level, traits, augments, etc. We read in the CSV file using pandas and clean the data up for further use. This script that we made that makes the CSV file uses a handful of different technologies, including “requests” for HTTPS calls, “tenacity” for retries, “tqdm” for some UI visuals on the progress bats, “python-dotenv” to load the API key from the .env.

3. High-Level Methods & Technologies:

In this section, I’ll go over a concise, dense walkthrough of our process, methods, and technologies that we have employed in 3 distinct sections. First, the preparation of the dataset and feature engineering. Secondly, the neural network models that are employed. Thirdly, the ensemble and tree based models.

3.1 Data Preparation and Cleaning

The data is first processed by reading eight separate CSVs (match_data.csv through match_data8.csv) and placing them into individual dataframes. These are all stacked into one dataframe called “matches” with 3840 rows and 8 columns of data. Next, we proceed with basic cleaning and inspection of the data. We drop non-predictive columns such as “match-id”, “game_time”, and “augments”. Then we check for duplicates and missing values where we found 2 duplicates and no missing values. Next we use the ".describe()” function to get summary stats on placements, levels, and gold_left. We then use graphs from seaborn to analyze the data. We plotted histograms for “placement”, “level”, and “gold_left” to see the distributions and then

we explored the relationships between them. We engineer the data using one hot encoding: using the “MultiLabelBinarizer” to expand each trait (ex: Bruisers, Enforcers) into its binary column. We do this for each unit (ex: draven, darius). These new binary columns will then widen our dataframe with hundreds of new binary features. We then do a sparsity check to estimate how sparse the columns of one hot feature are by computing the fraction of zeroes and we discovered that almost 4.2% of entries are zero which means that most features are present in the matches. We also had to make a map for the vocabularies of each unit for their name, trait, and team trait. Additionally, the max amount of units were capped out at 14 units per match and each match produced a tensor of shape (14, 575). These combined unit-level: ID, star, level, gold_left, unit-trait flags, team-trait: activation flags, tiers, counts, and match-level aggregates: avg star, 3 star count, avg trait tier, etc.

3.2 Neural Network Models

In this section, we will try to test some neural network models. We first trained our data on a 70/30 split on the one hot features that we previously prepped. We also had to convert the X_train and X-test to float Numpy arrays in order to prepare the data for 1D CNN. Also we had to shift placements from 1st place to 8th place to 0th place to 7th place for easier data management and one hot encoded those. We then trained the 1D CNN (one hot feature) for 40 epochs on a size of 32 with a 20% split. The result of that was a high training accuracy of around 61% but the testing accuracy remained low at around 24%. This told us that there was a possibility of overfitting happening. We also trained the 1D CNN (Tensor features) which were trained for 30 epochs and yielded 49% accuracy for training and 26% accuracy for testing. We then reshaped the data splits with 15% for testing, 15% for validation, and the remaining for

training. This meant we had about 2770 samples for training, 490 for validation, and 570 for testing. Next, we tried CNN regression which was the same tensor-CNN but MSE + RMSE metric. This yielded the test RMSE to about 1.9374. After that we tried transformer regression for 60 epochs which was the same regression architecture but applied to X_dense. This yielded an improved Test RMSE to about 1.578.

3.3 Ensemble and Tree-Based Models

For this last part we used ensemble and tree based models which basically will leverage multiple decision trees to enhance accuracy and robustness. Starting off with the AdaBoost baseline model which had a test accuracy of 27%. Next we used Random Forest with 200 trees and a depth of 20. This had a test RMSE of about 1.5173. For XGBoost the test RMSE was about 1.4765 and after grid search tuning with best params the test RMSE was reduced to 1.4557. For LightGBM the test RMSE was about 1.5064 and after grid search tuning with best params the test RMSE was reduced to about 1.50. For CatBoost the test RMSE was about 1.4635 and after grid search tuning with a small grid the test RMSE was increased to 1.5496 but with best params (extended grid) the test RMSE was reduced to about 1.4428 which was the best overall.

4. Results:

Overall results:

Model	**Task**	**Metric**	**Score**
1D CNN (tensor-CNN regression)	Regression	RMSE	1.9374
Transformer	Regression	RMSE	1.5756

(tensor-CNN regression)			
Random Forest	Regression	RMSE	1.5173
LightGBM	Regression	RMSE	1.5064
XGBoost	Regression	RMSE	1.4765
CatBoost (fixed)	Regression	RMSE	1.4635
CatBoost (tuned)	Regression	RMSE	1.4428

For the data, the RMSE is the Root Mean Squared Error and in this context tells us the predicted placement. For example a 1.44 RMSE means that the model's predicted placement (1-8) is only 1.44 placements off from the true result.

4.1 Results/takeaways for each model

For 1D CNNs, the model slides convolutional filters to capture local patterns such as how a 3 star carry paired with bruisers impacts placement. This model was generally weak from its heavy overfitting on sparse inputs which yielded a very high test error of 1.94. For Transformer Encoders, this model weighs relationships between pairs of units more flexibly than having fixed convolutional filters. For example the synergies between a 2 star sniper and a 2 star sorcerer. This model had a better generalization which was reflected by a RMSE of 1.58 but was still outperformed by the tree ensembles. For random forest, this model builds many decision trees on match features, with each tree having a vote on the overall placement. This was strong in interpreting important features in a match such as having a 3 star carry and was also robust to irrelevant features. The random forest had a RMSE of 1.52. For LightGBM and XGBoost, these models learned from gradient boosted trees that could correct its predecessor's errors with the

LightGBM being optimized for speed, and the XGBoost optimized for accuracy. This did better at a score of 1.51 for LightGBM and 1.48 for XGBoost. Finally we had CatBoost, which had gradient boosting with symmetric trees. This did particularly well for handling categorical patterns which were important for TFT which is all about traits and synergies. The strength of this model lies in its robustness and highly tuned features and had the best test RMSE of 1.44 after grid search tuning.

4.2 Application

Using the grid tuned CatBoost model was our best bet for having a reliable tool in predicting our placements in game. There are many applications for this model in our games such as having an AI assistant or coach which can take in live feed of your current board with the current IDs, star levels, traits, gold, and use the CatBoost model to forecast your future placement. This can come especially handy when you want fast in-game results such as if you needed to make a pivot to another team comp mid game. This model can predict your improved placement when you do so.

5. References:

- "Best TFT Meta Comps for Set 14 (Patch 14.3)." Tftacademy.com, 2023, tftacademy.com/tierlist/comps. Accessed 2 May 2025.
- "What Is Teamfight Tactics?" Teamfighttactics.Leagueoflegends.Com, teamfighttactics.leagueoflegends.com/en-us/news/game-updates/what-is-teamfight-tactics/. Accessed 2 May 2025.
- "TFT Meta Stats - Best Team Comps & Builds for Set 13 Patch 14.3." Tactics.tools, 2025, tactics.tools/team-compositions. Accessed 2 May 2025.