FDS Challenge Report

Lorenzo Zanda Paolo Marchetti Davide Vittucci
Sapienza Università di Roma

1. Feature Engineering

Before designing the final feature set, we also constructed a custom *Pokédex*-like table derived directly from the dataset. We manually aggregated information for each Pokémon species, such as typing and base stats, by inspecting the dataset and extracting all relevant attributes. We designed a total of **47 features** that allowed us to cover different aspects of a **pokemon battle**. They fall into several groups:

- **Turn Dynamics**: Offensive rate (how frequently a player chooses damaging moves), speed advantage rate and switching behavior (voluntary vs. forced switches).
- **Damage Dynamics**: Total HP loss for each player across the battle.
- **Status Effects**: Frozen turns for both teams, paralysis timing and a weighted status score reflecting the cumulative impact of status conditions.
- **Type Matchups**: Counts of super-effective and not very-effective hits and a team-level type advantage score.
- **Team Statistics**: Mean base stats (HP, Atk, Def, SpA, SpD, Spe), critical hit rate and P2 team coverage (fraction of revealed Pokémon).
- **Endgame state**: Remaining team size, mean HP of surviving Pokémon, fainted Pokémon difference and status-affected survivors.

2. Previous Approaches

To improve model performance, several modifications were made to both **feature engineering strategies** and **modeling approach**.

2.1. Feature Changes

Originally, the feature set consisted of a combination of **static** and **dynamic features** that tracked the evolution of the battle starting from the beginning of the match. Later we changed our perspective to analyze the battle from its final state rather than its progression. As a result, features such as KO counters and status-condition indicators were removed because they overlapped with the newly introduced end-state features.

2.2. Model Changes

In the initial phase, the work focused on evaluating single models. We started with **Logistic Regression**, which already produced good results, for later experimenting with **RandomForest**. After this, we moved onto a fixed stacking ensemble composed of manually defined base classifiers. The function always returned the same ensemble, with no option to train or evaluate individual models separately.

3. Final strategy

We refactored the pipeline to allow for a more flexible model generation, moving away from fixed structures. The system can now instantiate either standalone classifiers or a full **stacking ensemble** depending on the selected configuration.
The final model used is a **stacking classifier** composed of four base estimators: **Logistic Regression**, **RandomForest**, **LightGBM** and **SVC**. The ensemble uses cross-validated predictions from the base models, which are then passed to a **Logistic Regression classifier** serving as the meta-learner. Hyperparameter optimization is handled by the custom `get_tuned_params` function.
For each supported model, a dedicated hyperparameter search space has been defined, and a `RandomizedSearchCV` is executed with a fixed number of iterations and cross-validation folds. The procedure identifies the best-performing configuration for each estimator, and the stacking model is reconstructed using the tuned parameters when tuning is enabled.

4. Results

The table 1 shows the final accuracy results.

Model	Mean CV Accuracy
Logistic Regression	0.8492
LightGBM	0.8434
SVC	0.8456
Random Forest	0.8357
StackingClassifier	**0.8505**

Table 1. Model Performance Comparison