KTH Royal Institute of Technology

ID2223 - Scalable Machine Learning and Deep Learning

Clash Royale Machine Learning Project Report

Authors:
Stefano Tonini - stonini@kth.se
Jacopo Dallafior - jacopoda@kth.se

Contents

1 Introduction

This project aims to create a predictive system for Clash Royale, a strategy-based card game where players strategically build an eight-card deck from a pool of 181 unique cards. The selection of each card in the deck is crucial, as the synergy, counter-abilities, and overall balance of the deck significantly impact the likelihood of victory. The objective of this system is to estimate the win probability of a match given the decks of two opponents, providing insights into how card combinations influence outcomes.
By analyzing historical match data and leveraging advanced machine learning techniques, this system identifies patterns and interactions between different card combinations.

2 Methods

2.1 Historical Dataset

The dataset used for this project is sourced from Hugging Face, containing historical match data with approximately 2,500,000 previous matches. To utilize the data effectively, the dataset was encoded, requiring the definition of a card mapping dictionary. This dictionary maps each of the 181 cards to unique identifiers, enabling efficient data processing and feature extraction.

The primary features extracted from the dataset include:

- **Deck Composition:** The individual cards that form the eight-card decks for each player.
- **Match Outcome:** The result of the match, indicating the winning player.

These features are crucial for building a machine learning model to predict match outcomes. By analyzing this large dataset, patterns in card combinations, synergies, and counters are identified, forming the foundation for estimating win probabilities and optimizing deck strategies.

2.2 API

In addition to the historical dataset, the project leverages the official Clash Royale API to retrieve live player data. This API requires the player's tag as input and fetches data from the most recent 30 matches played by that individual. For each match, the API provides:

- **Deck Composition**
- **Match Outcome**

This integration enables real-time analysis and validation of the predictive system, ensuring that the model remains relevant and accurate by incorporating up-to-date gameplay data. The combination of historical and live data enhances the system's ability to identify evolving trends in card usage and strategy.

3 Data Representation and Preprocessing

3.1 Deck Representation Using One-Hot Encoding

To make the deck understandable for our chosen prediction model, we opted to use one-hot encoding for each card. This method represents each of the 181 available cards in the game as a binary vector. A deck is then encoded as a single vector of 181 elements, where all positions are set to 0 except for those corresponding to the cards present in the deck, which are set to 1.

The choice of one-hot encoding was motivated by its ability to effectively handle categorical data without introducing unintended ordinal relationships between the cards. By treating each card as an independent feature, this approach avoids any assumptions about card rankings or order, ensuring that

the model focuses on the presence or absence of specific cards rather than any perceived hierarchy. Additionally, this method maintains interpretability, as each position in the vector directly corresponds to a specific card.

3.2 Outcome Encoding and Dataframe Creation

To simplify the prediction task, a convention was established for encoding match outcomes:

- A value of **1** indicates that Player 1 won the match.
- A value of **0** indicates that Player 2 won the match.

Using this convention, the data was organized into a dataframe, which was stored on Hopsworks for efficient access and management. Each row in the dataframe represents a single match and includes:

- The encoded deck of Player 1.
- The encoded deck of Player 2.
- The match outcome (1 or 0).

3.3 Data Preprocessing

To ensure consistency and eliminate potential biases in the dataset, several preprocessing steps were applied:

- A dictionary containing all available cards in the game was created. Each card was mapped to its corresponding one-hot encoded vector, enabling seamless transformation of raw data into the binary format required for modeling.
- The dataset was structured such that each row corresponds to a single match, with columns representing player tags, the decks of both players, and the match outcome.
- Since the dataset inherently listed the winning player as Player 1, rows were randomized in pairs to prevent the model from learning a bias towards predicting the outcome in favor of the first player. This randomization step mitigated overfitting and ensured the model's generalizability.

These preprocessing steps allowed us to prepare a robust dataset suitable for training a predictive model capable of analyzing the impact of card combinations and match outcomes effectively.

4 Modeling and Evaluation

4.1 Models Tested

Two machine learning models were tested in this project to predict match outcomes based on the two input decks. The models included a neural network and an XGBoost classifier.

4.1.1 First Model: Neural Network

The first model was implemented using TensorFlow and consisted of a multi-layer neural network. The preprocessing and training steps were as follows:

- **Feature Standardization:** Input features were standardized using the `StandardScaler` from the `sklearn` library. This step ensured that the features had a mean of 0 and a standard deviation of 1, which is critical for effective training of neural networks.
- **Architecture:**

  - Input layer with 128 neurons and ReLU activation.
  - Hidden layer with 64 neurons and ReLU activation.
  - Output layer with 1 neuron and a tanh activation function, designed to produce predictions in the range [-1, 1].

- **Training:** The model was trained for 20 epochs with a batch size of 32, using the Adam optimizer and a mean squared error loss function. A validation split of 20% was used to monitor performance during training.

4.1.2 Second Model: XGBoost Classifier

The second model utilized the XGBoost classifier, a powerful gradient-boosting algorithm for classification tasks. Key features of this model include:

- **Configuration:** The classifier was initialized with `use_label_encoder=False` and evaluated using the log-loss metric (`eval_metric='logloss'`).
- **Training:** The model was trained using the same 80/20 train-validation split as the neural network. Hyperparameter tuning was conducted to optimize its performance.

4.2 Model Training and Evaluation

The dataset was split into training and validation sets using an 80/20 split to ensure robust evaluation of model performance. Hyperparameter tuning was conducted using grid search, where key parameters for each model were systematically tested to achieve optimal performance.

Additionally, the XGBoost classifier provided competitive results, demonstrating the effectiveness of gradient-boosting techniques in this domain.

4.3 Performance Metrics

Each model was evaluated based on the following metrics:

- **Accuracy:** The proportion of correctly predicted outcomes.
- **Precision and Recall:** To assess the model's ability to identify wins and losses accurately.
- **Validation Loss:** To monitor overfitting during training.

These metrics provided comprehensive insights into the strengths and weaknesses of each model, guiding the selection of the final predictive system.

5 Conclusion

The results from both models highlight their strengths and weaknesses, with XGBoost emerging as the preferred model for this task.

5.1 Model Performance

The XGBoost classifier demonstrated significantly better performance, achieving an overall accuracy of **84.3%** on the test set. The detailed classification report shows that it performs well in terms of both precision and recall for both classes (win/loss), with a macro F1-score of **83%**. This indicates that the model generalizes well and is capable of capturing the intricate relationships between card combinations in player decks.

In contrast, the neural network model showed a relatively low test accuracy of **56.62%**. This performance highlights potential challenges with the model architecture, hyperparameter tuning, or

insufficient representational power to capture the complex patterns in the dataset. Despite attempts to optimize the neural network's performance through preprocessing and training, it failed to outperform the simpler and more efficient XGBoost model.

5.2 Why XGBoost Was Chosen

XGBoost was chosen as the final model for the following reasons:

- **Higher Accuracy:** The XGBoost classifier consistently outperformed the neural network in predicting match outcomes.
- **Efficiency:** Training and tuning XGBoost was computationally efficient compared to the neural network, which required longer training times and careful hyperparameter adjustments.
- **Robustness:** XGBoost demonstrated robustness in handling the high-dimensional input data (one-hot encoded vectors) and provided stable predictions with minimal overfitting.

6 How to Run the Code

This section explains how to run the Clash Royale feature retrieval and inference code using the provided GitHub Actions workflow.

6.1 Automated Execution Using GitHub Actions

The provided GitHub Actions workflow automates the retrieval, processing, and inference steps.

Workflow Steps

1. **Triggering the Workflow:** The workflow can be triggered manually using the `workflow_dispatch` event or scheduled using the `cron` option in the YAML file.
2. **Setup:** The workflow installs the required Python dependencies and sets up Jupyter Notebook tools for execution.
3. **Feature Retrieval (Notebook 2):**
   The workflow runs `2_clash_royale_feature_pipeline.ipynb`, retrieving the latest data using the Clash Royale API and storing the processed features in Hopsworks.
4. **Inference (Notebook 4):**
   The workflow runs `4_clash_royale_batch_inference_copy.ipynb`, fetching the processed data, running the XGBoost model, and generating predictions.
5. **HTML Dashboard:** Notebook 4 is converted to an HTML file for visualization, creating a dashboard.
6. **Deployment to GitHub Pages:** The output HTML file is deployed to GitHub Pages for public access.

How to Run the Workflow

1. **Add Secrets to GitHub Repository:** Navigate to the repository settings and add the following secrets:
   - `HOPSWORKS_API_KEY`: API key for Hopsworks.
   - `CLASH_ROYALE_API_KEY`: API key for Clash Royale.
   - `GITHUB_TOKEN`: (Automatically available in GitHub Actions).

2. **Manually Trigger the Workflow:** Go to the "Actions" tab in your repository, select the `clash-royale-daily` workflow, and click "Run workflow."

3. **Optional Scheduling:** Uncomment the `schedule` section in the YAML file to run the workflow automatically. For example:

```
schedule:
  - cron: '11 6 * * *'
```

4. **Access the Dashboard:** Once the workflow completes, visit your GitHub Pages site to view the predictions:

```
https://<your-username>.github.io/<repository-name>/dashboard.html
```

Key Points

- **Notebook 2:** Retrieves and processes features using the Clash Royale API.
- **Notebook 4:** Performs inference using the trained XGBoost model and generates predictions.
- The entire process is automated via GitHub Actions, with the results deployed to GitHub Pages for easy access.

References

[1] Kaggle, *Clash Royale Games Dataset*, `https://www.kaggle.com/datasets/s1m0n38/clash-royale-games`.

[2] Clash Royale API, `https://developer.clashroyale.com/#/`.