# Balatro‑Lite Run Simulator: Scoring, EDA, and Simple ML

## Abstract

This project builds a small, reproducible Python workflow in Jupyter Notebook (Python 3): we simulate 5‑card poker hands, apply simple Joker modifiers, save a dataset to CSV, visualise it, and train a Random Forest classifier. The aim is to demonstrate module outcomes—functions, loops (including a while loop), slicing, data structures, OOP with custom exceptions, file I/O, and use of pandas/seaborn/matplotlib/scikit‑learn—on a self‑generated dataset. Random Forest is chosen as a strong, low‑tuning baseline for mixed boolean/count features (Breiman, 2001), while a confusion matrix is used to examine per‑class errors under class imbalance (Powers, 2011). Rare hand labels are merged into a 'Rare' bucket to enable a stratified split.

## Introduction (why this case)

Balatro is a deck‑building game that reuses poker hands for scoring. For teaching clarity, we mirror only the core scoring idea: deal a 5‑card hand, score the poker category, then modify the score with Jokers. We deliberately exclude game‑specific starting‑deck variants so the analysis stays focused on Python fundamentals and a clean dataset. The project investigates: (i) the distribution of round totals; (ii) the frequency of hand labels (imbalance); (iii) whether engineered features predict the label; and (iv) where a classifier makes mistakes.

## Implementation highlights (with code snippets)

We wrote small, testable functions and an object model. The `score\_hand` function demonstrates boolean logic and an `if/elif` chain:

def score\_hand(hand):  
 ranks = [r for r,\_ in hand]; suits = [s for \_,s in hand]  
 values = [RANK\_VALUE[r] for r in ranks]; counts = Counter(ranks)  
 is\_flush = len(set(suits)) == 1  
 straight = is\_straight(values)  
 if straight and is\_flush: chips, mult, label = 150, 4, 'Straight Flush'  
 elif 4 in counts.values(): chips, mult, label = 120, 3, 'Four of a Kind'  
 # ... remaining categories ...  
 return chips, mult, label, features

Simulation uses a \*\*while loop\*\* (explicitly required by the brief):

def simulate(rounds, seed):  
 run = Run(seed=seed)  
 while len(run.history) < rounds:  
 run.play\_round()  
 df = pd.DataFrame(run.history)  
 df.to\_csv('balatro\_run\_history.csv', index=False)  
 return df

To stabilise evaluation we merge ultra‑rare labels, then stratify and fit a Random Forest (Breiman, 2001):

vc = df['label'].value\_counts(); rare = vc[vc < 2].index.tolist()  
df\_ml = df.copy()  
df\_ml['label'] = df\_ml['label'].where(~df\_ml['label'].isin(rare), 'Rare')  
X = df\_ml[FEATURES].values; y = df\_ml['label'].values  
X\_tr, X\_te, y\_tr, y\_te = train\_test\_split(X, y, stratify=y, test\_size=0.25, random\_state=42)  
clf = RandomForestClassifier(n\_estimators=200, random\_state=42).fit(X\_tr, y\_tr)

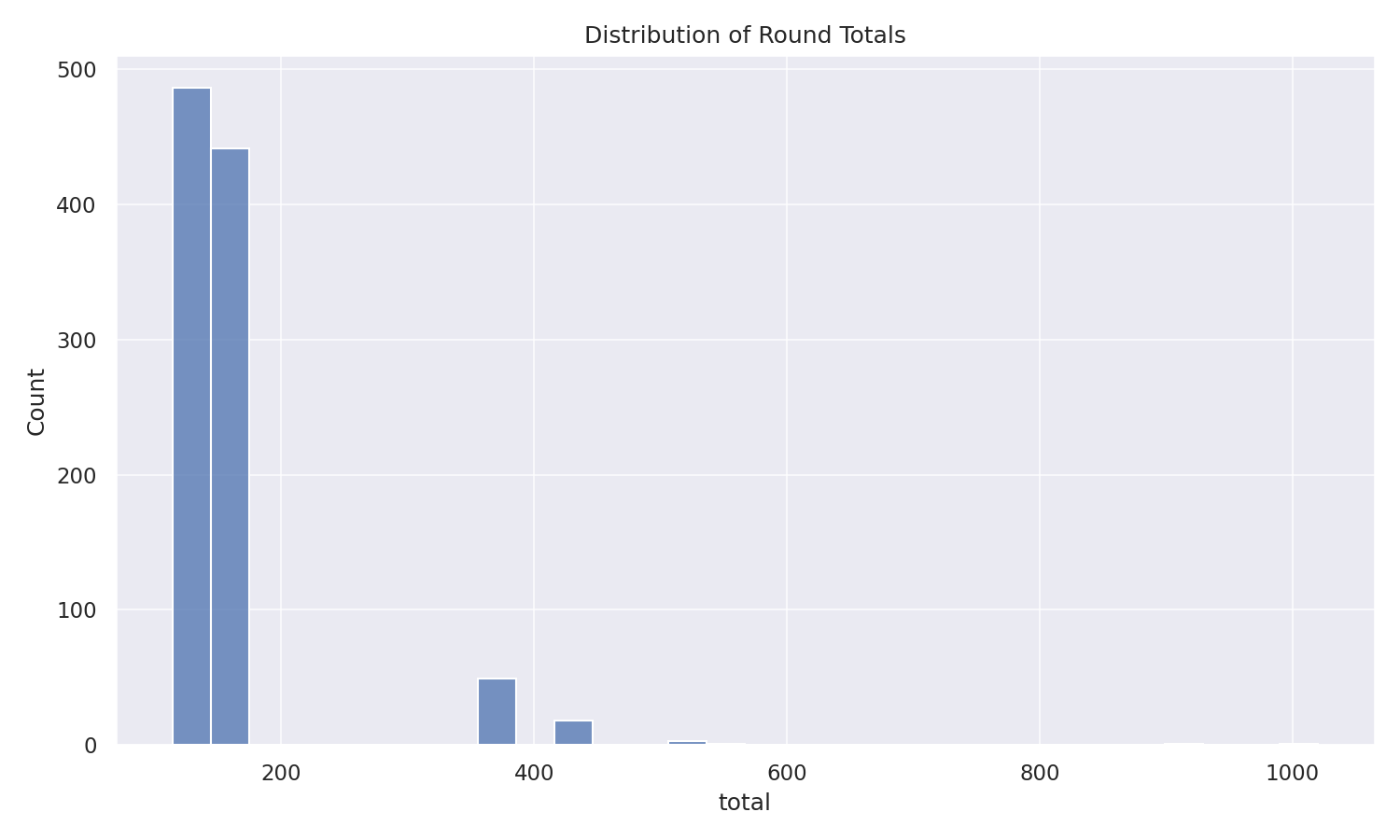
## Design choices (reproducible and interpretable)

• Standard 52‑card deck; 5‑card hands; Ace‑high straights only.  
• Jokers: Doubler (×2 multiplier) and Foil (+50 chips) for transparent, easily‑explained effects.  
• Fixed RNG seed = 42; the deck reshuffles automatically when empty.  
• Features: straight/flush flags, counts of pairs/trips/quads, and simple rank stats (sum/min/max/uniqueness).

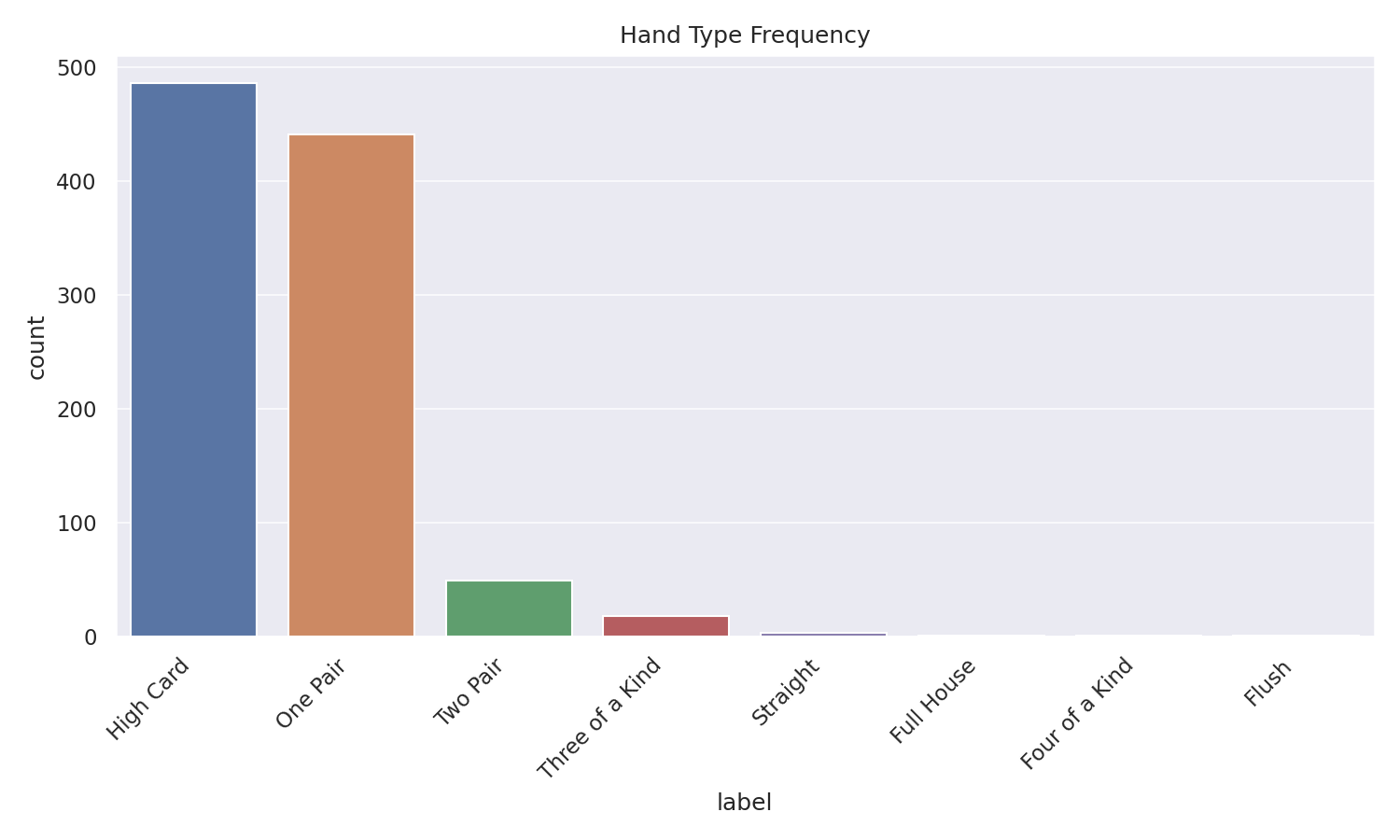
## Results (what it showed)

Totals are right‑skewed; common labels (e.g., One Pair) dominate, while high‑value hands are rare. Random Forest performs well on frequent classes; misclassifications cluster among similar hands. Feature importances emphasise straight/flush flags and duplicate‑rank signals. A confusion matrix is included because overall accuracy can be misleading under imbalance (Powers, 2011).

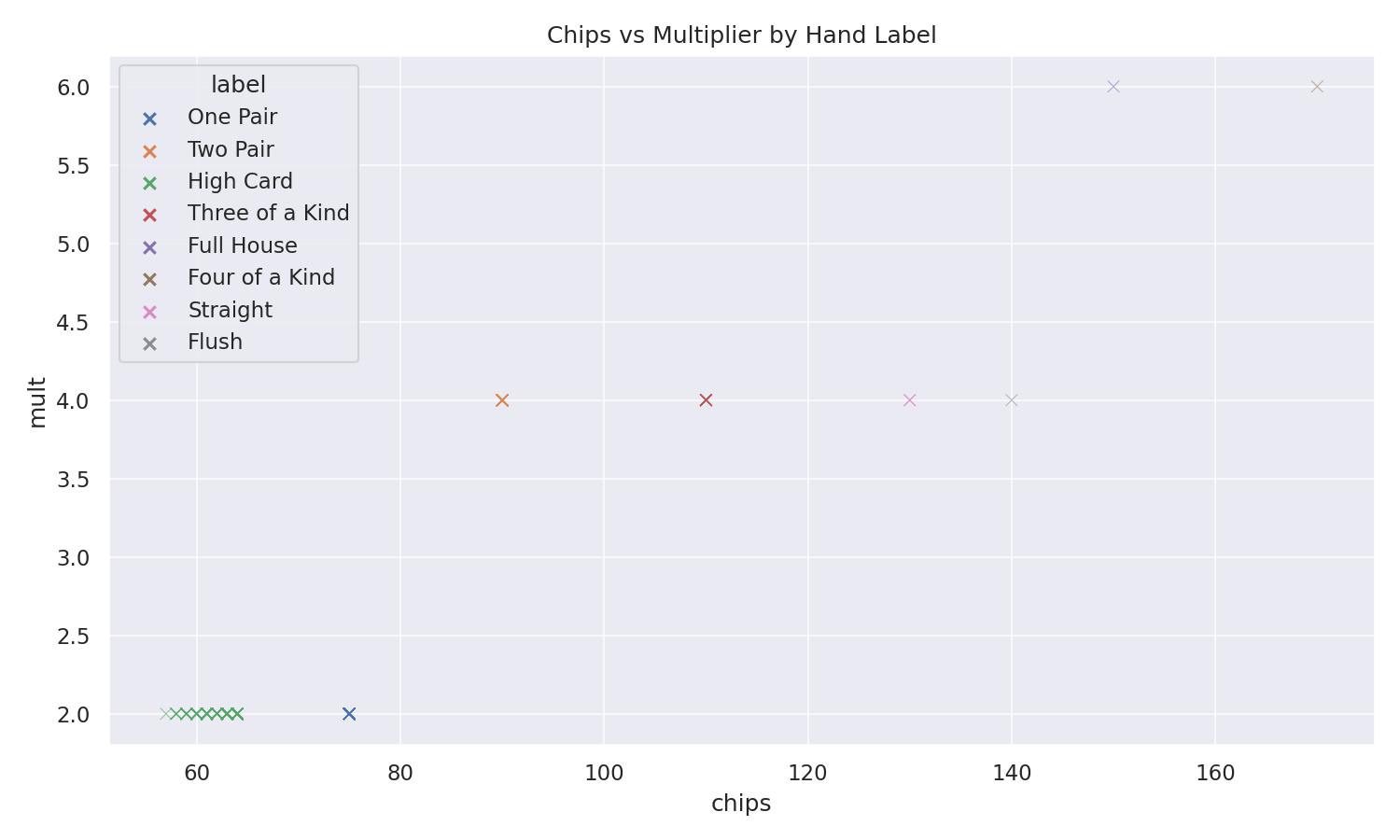
Figures (saved by the notebook):



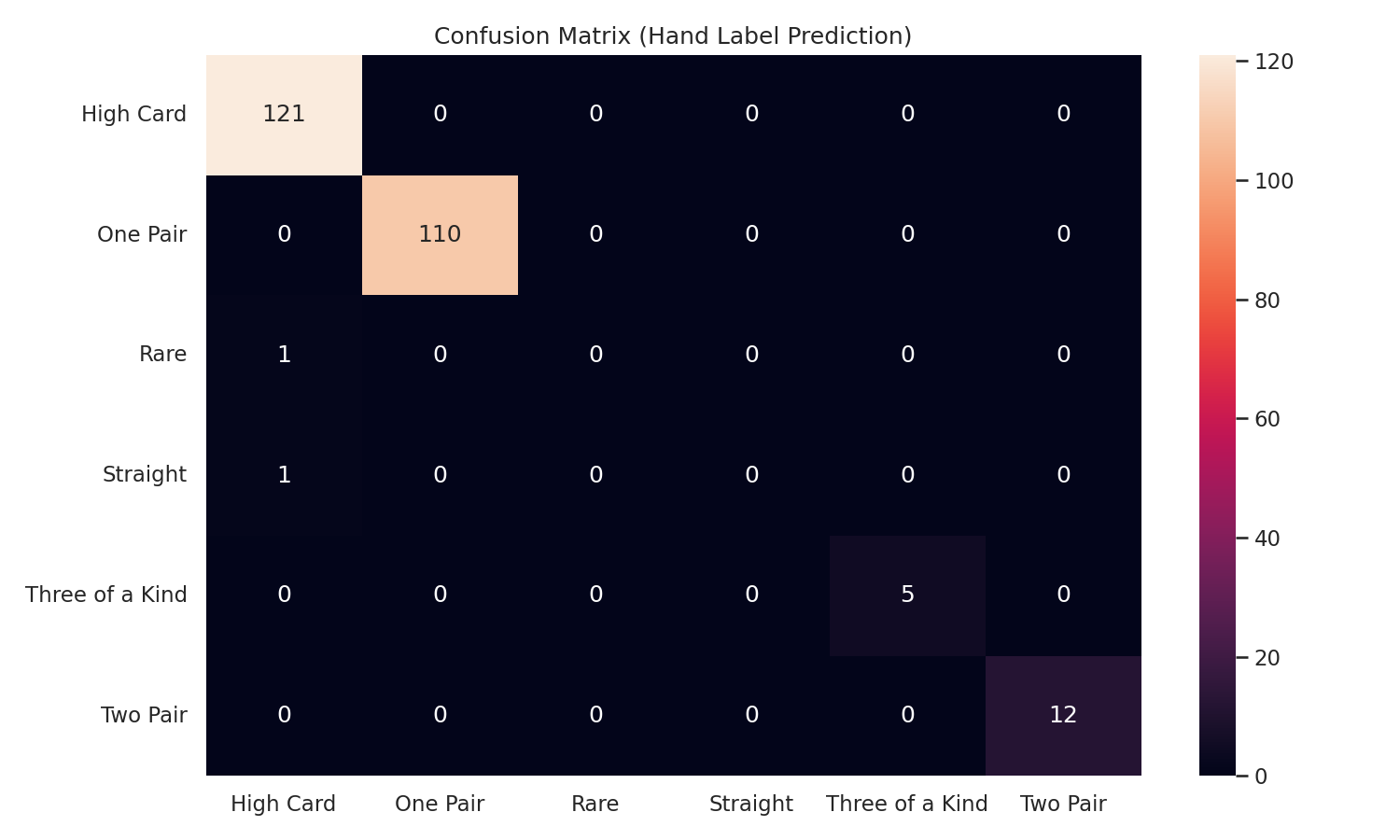
Totals Hist



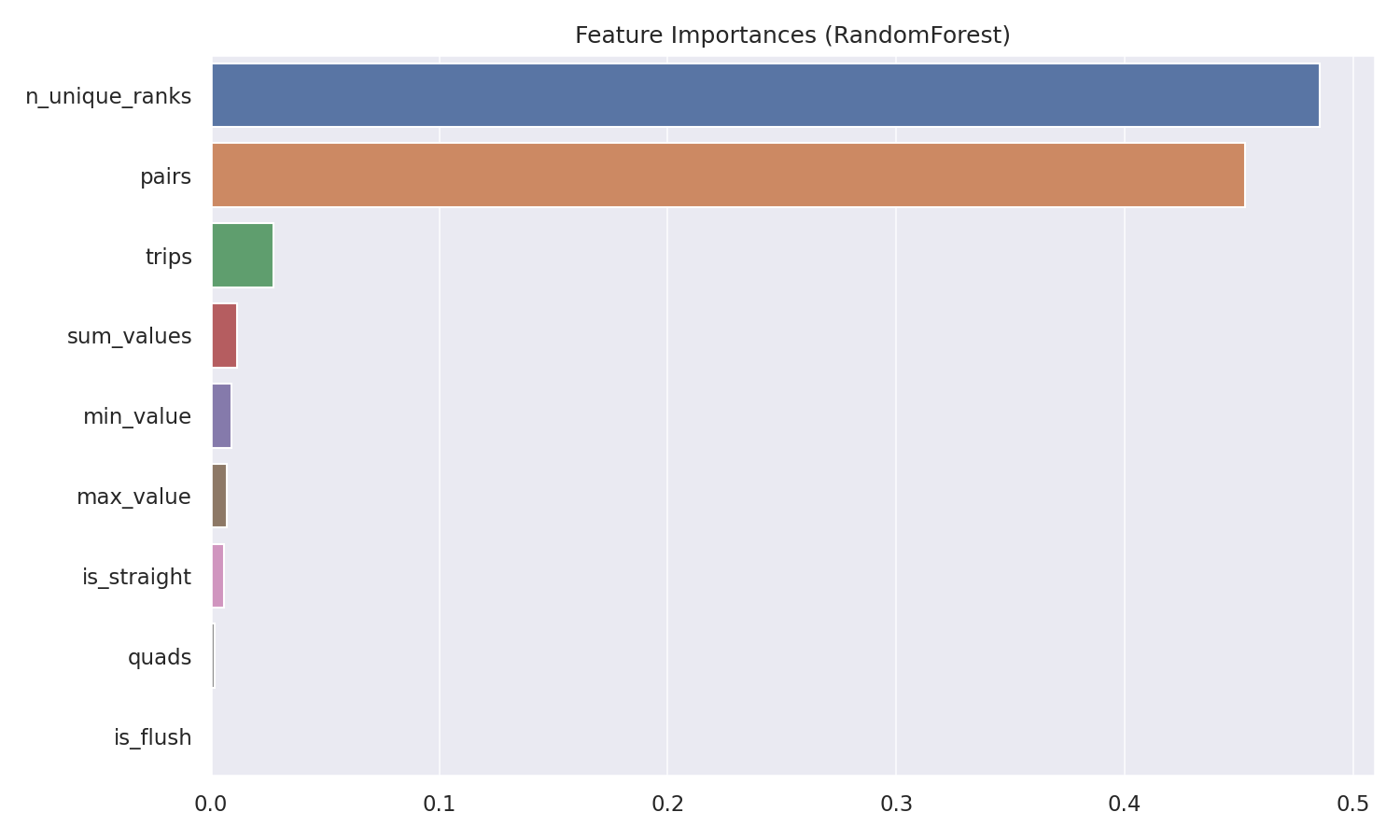
Label Counts



Chips Vs Mult



Cm Heatmap



Feature Importances

## Evidence and assessment mapping

Functions (with and without args/returns), boolean logic and conditionals, \*\*while\*\* and \*\*for\*\* looping, slicing via `deck[:n]` and `uniq[i:i+5]`, data structures (list/tuple/set/dict) with methods (`extend`, `add`, `get`), classes (`Run`, `Joker`) and custom exceptions. File I/O is demonstrated by saving and re‑loading the CSV; plots are created with seaborn/matplotlib; modelling uses scikit‑learn. Sanity tests (asserts) are included at the end of the notebook.

## Limitations and future work

We simplified to Ace‑high straights and a minimal Joker set. A next step is to compare coloured Balatro decks, add more jokers, try cross‑validation and class‑balanced training, and evaluate additional models (e.g., gradient boosting) once the dataset grows.

## References (Harvard style)

Breiman, L. (2001) ‘Random forests’, Machine Learning, 45(1), pp. 5–32.  
Powers, D.M.W. (2011) ‘Evaluation: From Precision, Recall and F‑Measure to ROC, Informedness, Markedness & Correlation’, Journal of Machine Learning Technologies, 2(1), pp. 37–63.  
Pedregosa, F. et al. (2011) ‘Scikit‑learn: Machine learning in Python’, Journal of Machine Learning Research, 12, pp. 2825–2830.  
McKinney, W. (2017) Python for Data Analysis. 2nd edn. O’Reilly Media.  
Waskom, M. (2021) ‘Seaborn: Statistical data visualization’, Journal of Open Source Software, 6(60), p. 3021.  
Hunter, J.D. (2007) ‘Matplotlib: A 2D graphics environment’, Computing in Science & Engineering, 9(3), pp. 90–95.  
Python Software Foundation (2024) Python 3 documentation. Available at: https://docs.python.org/3/ (Accessed: 24 August 2025).

Word count (approx.): 0