Daniel Cirincione and Michael LaMonica

CSC410 – Project 2 Paper

Link to Our GitHub: https://github.com/DCirincione/CSC410FinalProject

Quick Note. The commit history is not correct, we were originally working off Mike's repo, but I made a new one to create our Django backend

Project Topic and Problem

For this project we built an NFL stat predictor focused on fantasy football scoring. Our goal was to predict weekly fantasy points for the four main offensive skill positions: QBs, RBs, WRs, and TEs. Given historical game logs and upcoming week matchups, we wanted to create an AI model that could produce projected fantasy score for each player and rank the top options at each given position. We broke this problem down into a regression task. Given features describing a players recent performance, season production, and the strength of the opponent's defense against the position, predict the number of fantasy points that player will score in the upcoming week.

Data and Data Acquisition

We did not scrape or manually collect our data ourselves. Instead we relied on the Python library 'nflreadpy,' which is a wrapper around the public NFLverse datasets. Using this library we pulled two main kinds of data. The first was per player game stats, via 'nfl.load_player_stats(season=[season]).' This gives one row per player, per game, with columns such as player, team, position, week, and multiple fantasy scoring fields such as PPR, half-PPR, and standard. The second type of data was season schedule, via 'nfl.load_schedules(season=[season]).' We then filtered these raw tables down to a single season and a single position. We also had to deal with some schema variation. For example, some datasets use position while others use position_group or pos, and fantasy scoring columns can be named fantasy_points_ppr, fantasy_points_half_ppr, and fantasy_points, etc. To make the code robust, we wrote small helper functions that automatically pick the correct column from a set of options. All of the heavy lifting for downloading and caching the raw data is handled by nflreadpy. In our project, we focus on filtering, cleaning, and transforming the data into a training set for our machine learning model.

Model Choice and How it Works

We chose to use a Random Forest regression model, implemented via sklearn.ensemble.RandomForestRegressor. There were a few reasons why we made this choice. To start, it handles nonlinear relationships and interactions between features. Next, it is relatively robust to noisy features and doesn't require heavy feature scaling. And finally, it provides reasonable performance without requiring complex hyperparameter tuning. Conceptually a Random Forest regressor is a collection of many decision trees. Each tree is trained on a bootstrap sample of the training data (sampling rows with replacement). At each split it considers a random subset of features and chooses the split that best reduces prediction error, which is typically minimalized squared error for a regressor. Once the forest is trained, to make a prediction for a new player week, the model sends the feature vector down each tree, collets the predicted value from each tree, and averages them to get the final predicted fantasy score. The main parameters that define our model are: 'n_estimators=400' which is the number of trees in the forest. 'Max_depth=8" sets a cap on how deep each tree can grow, which helps to control overfitting. 'random_state=42' is used for reproducibility, and 'n_jobs=-1' uses all available CPU cores. The input features we fed into the forest included fantasy points in the players most recent game, their average of their last 3 games, average in their last 5 games, and their season average up to the previous week, and finally the week so the model can learn general patterns like early vs late season differences. For defense, we input the fantasy points the defense allowed to this position in its most recent game, their 3 and 5 game averages vs a position, and their season average vs a position. The target variable is 'target_fp' which is the fantasy points the player actually scored in week t. So each training row corresponds to player X, in week t, with features computer from weeks <= t-1, and the target is what happened in week t.

Data Preprocessing and Cleaning

We used a combination of Polars and Pandas for preprocessing. For filtering and sorting, we loaded full player stats for a season. We filtered rows to only keep the chosen position. We sorted by player_id, season, and week so that rolling windows over games are well defined. When choosing the fantasy scoring columns, we based it on a command line –scoring flag (ppr, half, standard) and mapped to the correct fantasy points column using the helpers mentioned earlier. When it came to building player level rolling features, for each player and each potential target week t (starting at week 2), we aggregated all their games up through week t-1 and computed last game fantasy points, 3 and 5 game rolling mean, and season to date average. We built defense vs position features using the same stats tables. We treated the opponent team as the defense and for each defense and week, summed the fantasy points allowed to that position. We then computed similar stats, getting the last game, 3 and 5 game averages, and season to date average, where each rows values are based only on games before the target week. To combine everything together and clean missing values, we joined the offensive features, defensive features, and the actual week t fantasy points into a single training table. Early in the season some players/defenses don't have enough games to build a 3 or 5 game rolling window, so these rows contain missing values. We dropped rows that had missing values in any critical feature or in the target. After all the feature engineering in Polars, we converted the final training table to a Pandas DataFrame and selected the feature columns and target column for scikit-learn. For predicting the upcoming week, we repeated a similar process but only using data up to the latest completed week, then used the season schedule to determine which defense each team will face in the upcoming week. We merged in the latest defensive rolling stats for those defenses and dropped cases where we still lack enough history.

Training Algorithm Library and Overview

Like we mentioned earlier, we used RandomForestRegressor from scikit-learn. The training pipeline looks like this:

1. Build the training dataset with build_position_training_dataset(season=season, position, and scoring), which returns:
   df: pandas DataFrame with feature columns and target_fp
   max_week: latest week available in seasons data
2. Define the feature list:
   ```
   ["fp_prev1", "fp_roll3", "fp_roll5", "fp_season_avg",
    "week",
    "def_fp_prev1", "def_fp_roll3", "def_fp_roll5", "def_fp_season_avg"]
   ```
3. Extract:
   ```
   X = df[feature_cols]
   Y = df["target_fp"]
   ```
4. Instantiate and fit the model:
   ```
   model = RandomForestRegressor(
      n_estimators=400,
      max_depth=8,
      random_state=42,
      n_jobs=-1,
   )
   model.fit(X, y)
   ```

Under the hood, the training algorithm does this for each tree. Draw a bootstrap sample of training row, then recursively grow a decision tree. At each node, consider a random subset of features. For each candidate feature and split value, compute the reduction in mean squared error if the data is split there. Choose the split that yields the best error reduction. Stop when reaching max_depth or too few samples at a node. After building all trees, the forest stores each trees structure and leaf predictions. The final model prediction is the average of all tree predictions. We also computed an in sample mean absolute error (MAE) on the training data by comparing model.predict(X) to the true y. This is mainly a sanity check that the model is at least learning some structure in the data; a proper evaluation would ideally include separate validation or test set.

Challenges and Trial and Error

We ran into several challenges while building this project. Our first issue was with our data schema variation. As mentioned above, the different versions of NFLverse data use slightly different column names, and our initial code assumed fixed column names and broke on certain seasons. We solved this by writing the general helper functions that search through a list of possible column names and raise clear errors if none are found. Our second issue came when building realistic features. Our initial versions focused only on season averages or a small number of recent games and ignored defense strength. These models tended to overvalue high volume stars regardless of their matchup. Adding the defense vs position rolling features improved realism of the predictions, especially for players with middle of the pack talent but favorable matchups. Our third big challenge was avoiding data leakage. It's easy to accidentally use information from the same week you're trying to predict, and sometimes even future weeks when computing rolling stats. We had to be careful to shift and roll such that for each target week t, the player features only use weeks <= t-1. Defense features for a week only use games played before that week. Getting this timeline right required rethinking the order of group-by and rolling operations, as well as double checking the logic itself. The next big challenge we faced was handling missing history. Early in the season some players or defenses don't have enough games for their rolling stats, which creates missing values. We experimented with different approaches, like filling with season averages, last available value, or dropping these rows. For simplicity and clarity, we ended up dropping rows that didn't have enough history in any of the required columns. Our last large challenge we faced was more so trial and error when choosing our hyperparameters. We experimented with different values of n_estimators and max_depth. Very shallow trees underfit, while very deep trees overfit heavily to outlier performances. We settled on 400 trees and a depth of 8 as a reasonable tradeoff between model complexity and stability.

Evaluation, Impact, and Future Directions

Overall, our system worked as a proof of concept. For a chosen season, position, and scoring system, it successfully trains a model and produces a ranked list of projected fantasy points for the upcoming week. The players at the top of these rankings generally match their predictions, establishing starters with strong performance and favorable matchups, as well as the in sample MAE indicating that the model is capturing meaningful signal rather than just predicting a constant average.

At the same time, there are clear limitations. We mainly evaluated the model on in sample performance instead of using proper train/validation/test split, so it's true out of sample accuracy is unknown outside of limited testing with our completed model. The current version also trains on a single season at a time and only uses fantasy points plus defense vs position features, ignoring potentially important information like target share, snap counts, betting lines, injuries, depth chart shifts and weather. A natural next step would be to build a more rigorous evaluation pipeline using the important information talked about above, pool multiple seasons into a larger dataset, and compare Random Forests against other models such as gradient boosting, XGBoost, or neural networks.

Machine learning in this space has several potential benefits. Fantasy players could use projections as a decision support tool for setting lineups or making waiver and trade decisions. Content creators and analysts could plug a model like ours into web tools and dashboards, and in academic context, this kind of project is a compelling way to teach data pipelines, feature engineering, and model evaluation using a familiar domain. However, there are also risks, especially involving gambling and over reliance on automated predictions. Similar models can easily be repurposed for sports betting or player props, and users may treat outputs as truth while ignoring uncertainty, late injury news, or other important context. To mitigate these concerns, future work should focus on the models uncertainty, frame it primarily as an educations and entertainment tool, encourage users to combine predictions with their own judgment and up to date info, and, if deployed publicly, include responsible use messaging rather than over promising accuracy.