Exploring Fan Sentiment in the Digital Realm

Dylan Jorling

May 24, 2023

I. Introduction

In recent years, social media platforms have become a significant source of user-generated content that reflects people's opinions, sentiments, and experiences. Analyzing and understanding the sentiment expressed in these social media posts can provide valuable insights into public opinions, consumer behavior, and market trends. One popular social media platform for studying sentiment analysis is Reddit, a vast online community where users share and discuss various topics.

Social media sentiment analysis involves extracting and analyzing the sentiment expressed in textual data, such as posts, comments, and reviews, to determine overall sentiment polarity (positive, negative, or neutral) and gain a deeper understanding of people's attitudes and opinions. Sentiment analysis has gained considerable attention in the field of natural language processing (NLP) due to its wide range of applications in market research, brand management, public opinion analysis, and customer feedback analysis.

Reddit is a social news aggregation, web content rating, and discussion platform where users can submit posts, comment on them, and engage in conversations. Reddit is structured as a group of subreddits, where each subreddit corresponds to a specific topic. One of the most popular group of subreddits is professional sports, with r/NBA, r/NFL and r/soccer each having several million subscribers. A more granular group of subreddits involves specific team subreddits where fans can interact with each other and discuss everything related to their favorite team. r/Lakers is a subreddit with nearly 500,000 subscribers and is dedicated to the Los Angeles Lakers, a popular National Basketball Association (NBA) professional basketball team.

The purpose of this project is to conduct sentiment analysis over the entirety of the 2022-2023 NBA season using data scraped from r/Lakers. Specifically, the project focuses on an entity-level sentiment analysis related to every player on the team as well as the team owner, general manager and head coach.

Traditional sentiment analysis approaches treat entire text observations as a single unit and assign sentiment scores at the document level. Entity-level sentiment analysis aims to go beyond this by identifying and analyzing sentiment towards specific entities, leading to more complex insights. This paper leverages NLP techniques, including entity recognition, unsupervised sentiment analysis, and data visualization, to gain insights into the sentiment expressed towards each member of the Lakers organization throughout the season. Finally, player sentiment is measured against on-court team and player performance to uncover player-specific relationships.

II. Data Overview

The dataset for this project spans the entirety of the 2022-2023 NBA season from mid-October 2022 to mid-April 2023. To collect the necessary data from Reddit, the Upshift API was used instead of the popular Reddit API which limits scraping to only the most recent 1,000 posts in a specific subreddit.

The final dataset is comprised of full text data for approximately 11,000 posts and 147,000 comments made in r/Lakers throughout the season. The collected data includes essential metadata consisting of post/comment ID, timestamp, author username, upvotes received, and parent post/comment IDs.

III. Text Processing and Methodology

Basic Cleaning and Entity Recognition

Standard text cleaning techniques are applied to the collected data including converting all text to lowercase, removing special characters, eliminating GIFs, and removing hyperlinks. Emojis are deemed important indicators of sentiment and therefore left in the text.

The subsequent task focused on identifying specific players mentioned in each post and comment. Initially, Named Entity Recognition (NER) was employed using a pre-trained model from the SpaCy python library. However, this approach resulted in significant inaccuracies and inconsistencies and as a result, an alternative approach was explored that leveraged both domain knowledge and trial-and-error with regular expression (regex) patterns.

By combining understanding of the entities of interest and iterative testing with regex patterns, a customized approach was developed to identify each specific entity. This approach resulted in initially capturing desired entity references for 76,000 text-mention pairings.

Improving the Base Sentiment Model

Sentiment Analysis was conducted utilizing the Vader Sentiment library in Python, which employs a lexicon-based approach to determine the sentiment of given text. This pre-trained model is specifically trained on social media data and is thus an excellent candidate for use on reddit-sourced data.

Upon testing the sentiment ratings on a subset of several hundred posts and comments, it became evident that the sentiment analysis accuracy was suboptimal. While training a model on a large volume of similar data would provide the most accurate results, such an approach necessitates labeled data in significant quantities and is therefore not realistic. Consequently, incremental adjustments to the lexicon and other easier techniques were identified to enhance performance, resulting in four specific avenues: nickname adjustments, emoji-lexicon adjustments, basketball-related lexicon adjustments, and part-of-speech resolving.

Domain knowledge was leveraged to identify common nicknames associated with each entity that typically implied positive or negative sentiment. These nicknames were extracted and utilized to automatically label posts or comments with respect to the specific entity as either positive or negative. It is important to note that for the numerous posts and comments that mention multiple entities, sentiment is determined with respect to each entity.

Observing the Vader Sentiment Emoji Dictionary revealed that the lexicon ratings for certain emojis were outdated and inaccurate in the context of the data. Additionally, several commonly used emojis were not present in the lexicon at all. To address these issues, missing emojis were added to the lexicon and the sentiment assignments of several other emojis were revised. For example, the sentiment rating for the "fire" emoji initially contained a very negative polarity score, and was thus adjusted to a positive polarity score. In total, 106 emojis were either added to the lexicon or had its sentiment adjusted.

It was determined that incorporating sentiment analysis specific to basketball-related terms could further enhance the model's performance. This involved identifying commonly used basketball-related terms and assigning sentiment polarity scores to them. For instance, a positive sentiment value was assigned to the term "beast" and a negative sentiment value to the term "turnover." Overall, a total 185 basketball-related words were adjusted for.

In specific cases where the sentiment polarity of certain words was context-dependent, SpaCy's part-of-speech (POS) identification was utilized. SpaCy POS attempts to accurately identify the part of speech of each word used in text. One prominent challenge encountered involved the word "like" which has a positive polarity score by default. Upon detailed analysis it was apparent that "like" was frequently used as a preposition, where a neutral sentiment is more appropriate. To address this, text was modified by deleting instances of "like" when used as a preposition, and leaving "like" with a positive polarity score when used as a verb.

These incremental adjustments to the lexicon and the incorporation of domain-specific knowledge aimed to enhance the accuracy and relevance of the sentiment analysis performed on the desired entities in r/Lakers.

Co-reference Resolving, Further Entity Extraction, and Sentence Tokenization

To extract player mentions more comprehensively, a process called co-reference resolving was implemented. Co-reference resolving involves identifying all expressions within the text that refer to the same entity. The utilization of co-reference resolving in this project served two purposes: 1) To extract more intra-text mentions and 2) To extract indirect mentions. Extraction of intra-text mentions involves identifying additional references in text where an entity is already identified and replacing references with the entity name. Applying co-reference resolution to these cases would enhance the viability of using sentence-specific sentiment analysis instead of entire-text sentiment analysis. Extraction of indirect mentions applies to comments that only indirectly mention an entity and made in reply to a parent post/comment with a direct entity mention. Using coreference resolution in these cases resulted in entity identification that otherwise would have been missed.

Co-reference resolution was implemented using the SpaCy-experimental co-reference model, which was trained specifically to identify co-references. The SpaCy NER model was configured to identify actual before the co-reference model identified entity co-references and replaced them with entity names. Due to the aforementioned inconsistencies in SpaCy NER's identification of player entities, custom entities were added to the NER model for all players to ensure their consistent recognition. Since each player was referred to by various names in the data, each mention was replaced with a single name for each entity, substantially reducing the number of custom entities added to the NER model. Furthermore, an entity filter was created to ensure that SpaCy co-reference replace only co-references related to desired player entities, thus drastically speeding up the process.

Following a top-down approach, co-reference resolving was initially applied to all posts containing entity mentions, leading to the extraction of further intra-text entities. Top-level comments, or comments directly replying to a post, were the next set of texts to be resolved. In such cases, the parent-post text was combined with the comment text and the entire text was co-referenced together. This was especially useful in the case where an entity was directly named in a post, but only indirectly named in the comment. The process was repeated for second-level comments by combining the text of the parent first-level comment in a similar fashion and then iterated down five levels of comments. Taking advantage of the hierarchical structure of reddit in this way proved valuable in extracting entity mentions that were missed previously, resulting in an additional 8,000 comment-entity pairings.

The final technique employed to enhance entity-level sentiment accuracy involved implementing sentence tokenization, which involves breaking down text into a list of individual sentences. The rationale behind incorporating sentence tokenization in the sentiment analysis process is to isolate sentences that mention specific entities while disregarding sentences that do not refer to the entity of interest. Intra-text co-reference resolving prior to sentence-tokenization enables all mentions of an entity to be captured and is a much more robust approach than simply utilizing direct mentions.

The evaluation process provides insights into the performance of each method discussed above and contributes to the understanding of entity-specific sentiment within NBA-related discussions on Reddit.

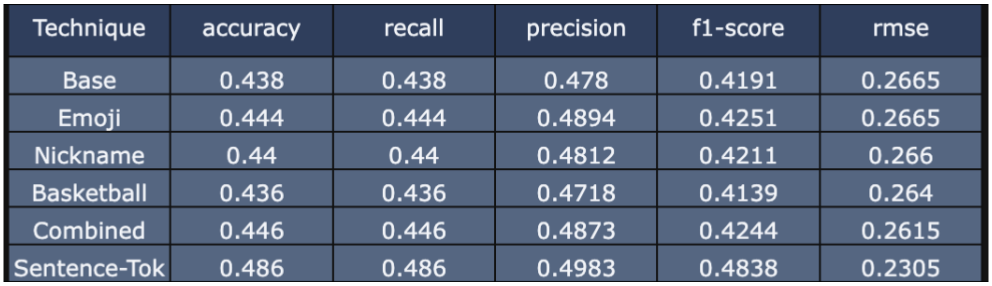
IV. Analysis

Comparing Sentiment Techniques

Sentiment scores were measured for 6 different sentiment techniques:

1) Base: Vader Sentiment lexicon with no adjustments
2) Emoji-Adjusted: Vader Sentiment lexicon with emoji lexicon adjustments
3) Nickname-Adjusted: Vader Sentiment lexicon with no adjustments; positive or negative nicknames automatically assign sentiment
4) Basketball-Lexicon Adjusted: Vader Sentiment lexicon with only basketball-related adjustments
5) Combined: Vader Sentiment with emoji, nickname and basketball adjustments
6) Sentence-Tokenized: Uses combined adjustments and assigns sentiment only for sentences where a specific entity is mentioned

To evaluate the effectiveness of each method, a random sample of 500 text-entity pairings was drawn and manually labeled. The results are shown in Figure 1:



Technique	accuracy	recall	precision	f1-score	rmse
Base	0.438	0.438	0.478	0.4191	0.2665
Emoji	0.444	0.444	0.4894	0.4251	0.2665
Nickname	0.44	0.44	0.4812	0.4211	0.266
Basketball	0.436	0.436	0.4718	0.4139	0.264
Combined	0.446	0.446	0.4873	0.4244	0.2615
Sentence-Tok	0.486	0.486	0.4983	0.4838	0.2305

Figure 1: Comparing Sentiment Performance

Each technique yields marginal improvement in either raw accuracy score, or RMSE, which penalizes two level misses (e.g., positive instead of negative) more than one-level misses (e.g. positive instead of neutral). Combining the first four techniques results in the greatest RMSE reduction and accuracy increase, although the improvement over the base model is marginal.

Utilizing sentence-tokenization resulted in a much more substantial increases in accuracy and f1-score along with a larger decrease in RMSE. While isolating entities using sentence tokenization is extremely useful in determining entity-level sentiment, sentence-tokenization cannot capture intra-sentence sentiment, which would require more advanced techniques. Exploring such techniques in addition to refining the other four used would likely result in even more improvement over the base model.

General and On-Court Performance Analysis

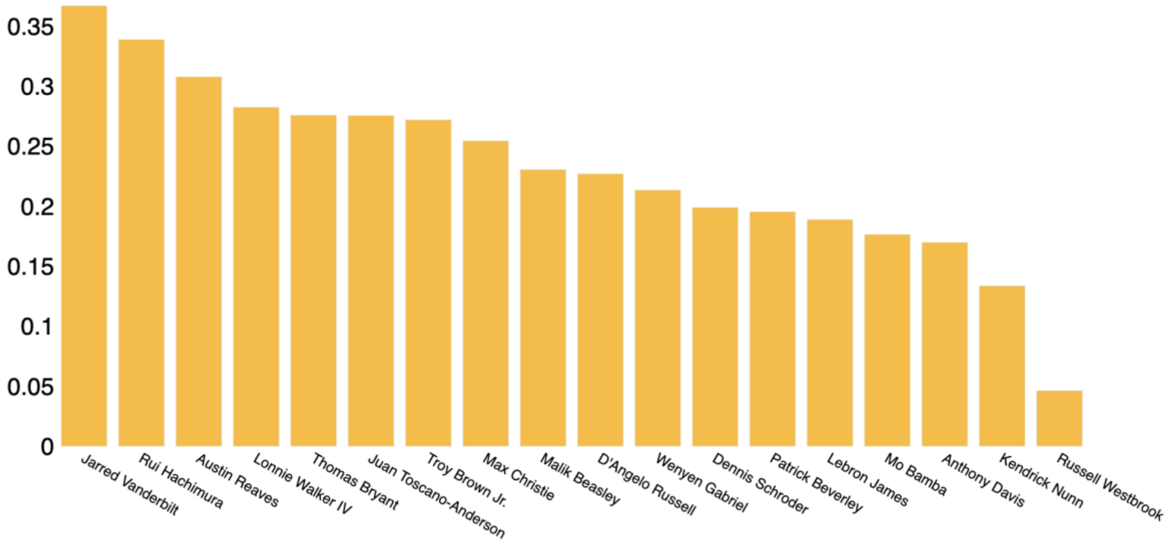
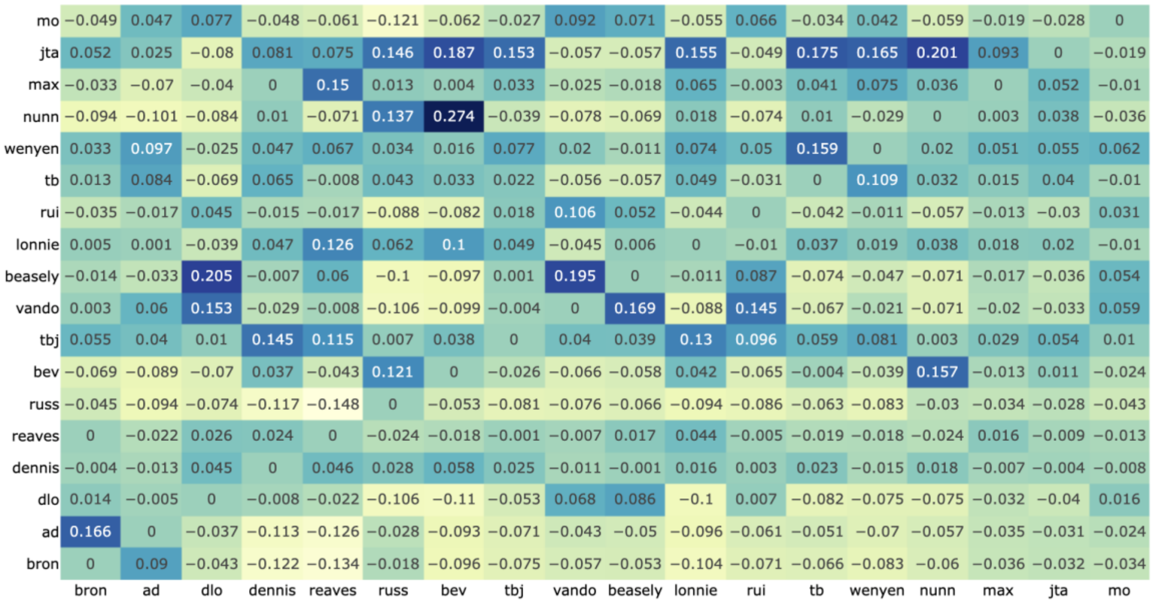


Figure 2: Overall Player Sentiment Scores

Figure 2 displays overall sentiment scores for each player who played significant minutes during the season. Sentiment scores were calculated by subtracting the proportion of negative posts/comments from the proportion of positive posts/comments for each individual player. Additionally, posts were given 2.5x weight compared to comments to account for their higher status. It is unsurprising that the two highest sentiment scores belong to players who were acquired mid-season, given the Lakers' below-average excellent finish. While star players Lebron James and Anthony Davis accounted for approximately 40% of total player mentions, they had two of the lowest sentiment scores on the entire team. Fans were clearly not impressed with the team’s mediocrity for a majority of the season and seemingly put a majority of the blame on its best players.

Young, new players on the other hand had generally higher sentiment ratings than their more established counterparts. This could potentially be linked to excitement over a new, unknown player or enthusiasm about a young player’s potential going forward. Russell Westbrook, acquired the previous season, had a low sentiment score, likely due to the large drop-off in team performance coinciding with his arrival.



	bron	ad	dlo	dennis	reaves	russ	bev	tbj	vando	beasely	lonnie	rui	tb	wenyen	nunn	max	jta	mo
mo	−0.049	0.047	0.077	−0.048	−0.061	−0.121	−0.062	−0.027	0.092	0.071	−0.055	0.066	−0.034	0.042	−0.059	−0.019	−0.028	0
jta	0.052	0.025	−0.08	0.081	0.075	0.146	0.187	0.153	−0.057	−0.057	0.155	−0.049	0.175	0.165	0.201	0.093	0	−0.019
max	−0.033	−0.07	−0.04	0	0.15	0.013	0.004	0.033	−0.025	−0.018	0.065	−0.003	0.041	0.075	0.036	0	0.052	−0.01
nunn	−0.094	−0.101	−0.084	0.01	−0.071	0.137	0.274	−0.039	−0.078	−0.069	0.018	−0.074	0.01	−0.029	0	0.003	0.038	−0.036
wenyen	0.033	0.097	−0.025	0.047	0.067	0.034	0.016	0.077	0.02	−0.011	0.074	0.05	0.159	0	0.02	0.051	0.055	0.062
tb	0.013	0.084	−0.069	0.065	−0.008	0.043	0.033	0.022	−0.056	−0.057	0.049	−0.031	0	0.109	0.032	0.015	0.04	−0.01
rui	−0.035	−0.017	0.045	−0.015	−0.017	−0.088	−0.082	0.018	0.106	0.052	−0.044	0	−0.042	−0.011	−0.057	−0.013	−0.03	0.031
lonnie	0.005	0.001	−0.039	0.047	0.126	0.062	0.1	0.049	−0.045	0.006	0	−0.01	0.037	0.019	0.038	0.018	0.02	−0.01
beasely	−0.014	−0.033	0.205	−0.007	0.06	−0.1	−0.097	0.001	0.195	0	−0.011	0.087	−0.074	−0.047	−0.071	−0.017	−0.036	0.054
vando	0.003	0.06	0.153	−0.029	−0.008	−0.106	−0.099	−0.004	0	0.169	−0.088	0.145	−0.067	−0.021	−0.071	−0.02	−0.033	0.059
tbj	0.055	0.04	0.01	0.145	0.115	0.007	0.038	0	0.04	0.039	0.13	0.096	0.059	0.081	0.003	0.029	0.054	0.01
bev	−0.069	−0.089	−0.07	0.037	−0.043	0.121	0	−0.026	−0.066	−0.058	0.042	−0.065	−0.004	−0.039	0.157	−0.013	0.011	−0.024
russ	−0.045	−0.094	−0.074	−0.117	−0.148	0	−0.053	−0.081	−0.076	−0.066	−0.094	−0.086	−0.063	−0.083	−0.03	−0.034	−0.028	−0.043
reaves	0	−0.022	0.026	0.024	0	−0.024	−0.018	−0.001	−0.007	0.017	0.044	−0.005	−0.019	−0.018	−0.024	0.016	−0.009	−0.013
dennis	−0.004	−0.013	0.045	0	0.046	0.028	0.058	0.025	−0.011	−0.001	0.016	0.003	0.023	−0.015	0.018	−0.007	−0.004	−0.008
dlo	0.014	−0.005	0	−0.008	−0.022	−0.106	−0.11	−0.053	0.068	0.086	−0.1	0.007	−0.082	−0.075	−0.075	−0.032	−0.04	0.016
ad	0.166	0	−0.037	−0.113	−0.126	−0.028	−0.093	−0.071	−0.043	−0.05	−0.096	−0.061	−0.051	−0.07	−0.057	−0.035	−0.031	−0.024
bron	0	0.09	−0.043	−0.122	−0.134	−0.018	−0.096	−0.075	−0.057	−0.053	−0.104	−0.071	−0.066	−0.083	−0.06	−0.036	−0.032	−0.034

Figure 3: Relative Player Associations

Figure 3 displays a heatmap of relative player associations. The plot highlights which players are most often mentioned with each other, relative to how often they are mentioned with all players. For example, the column for titled “bron”, displays the relative percent mentions of Lebron James with every other player. On average, Lebron James is mentioned in 26.1% of posts/comments where another player is mentioned. This 26.1% was subtracted from the initial proportions to adjust for Lebron being an extremely popular player mentioned in a high percentage of posts. The remaining values are thus the proportion of posts/comments above or below 26.1% that Lebron James is mentioned in, for each player.

The plot reveals that star players LeBron James and Anthony Davis are often mentioned together and that player acquired at the same time are also often grouped together. Austin Reaves, considered a "star role player," is much more often mentioned with team role players than team stars. Reaves has one of the highest sentiment scores on the team, and with that context, his low correlations with the star players and their relatively low sentiment scores make sense. Further exploration of these less obvious insights, including breaking down positive and negative mentions, could provide interesting insights into how fans group certain players.

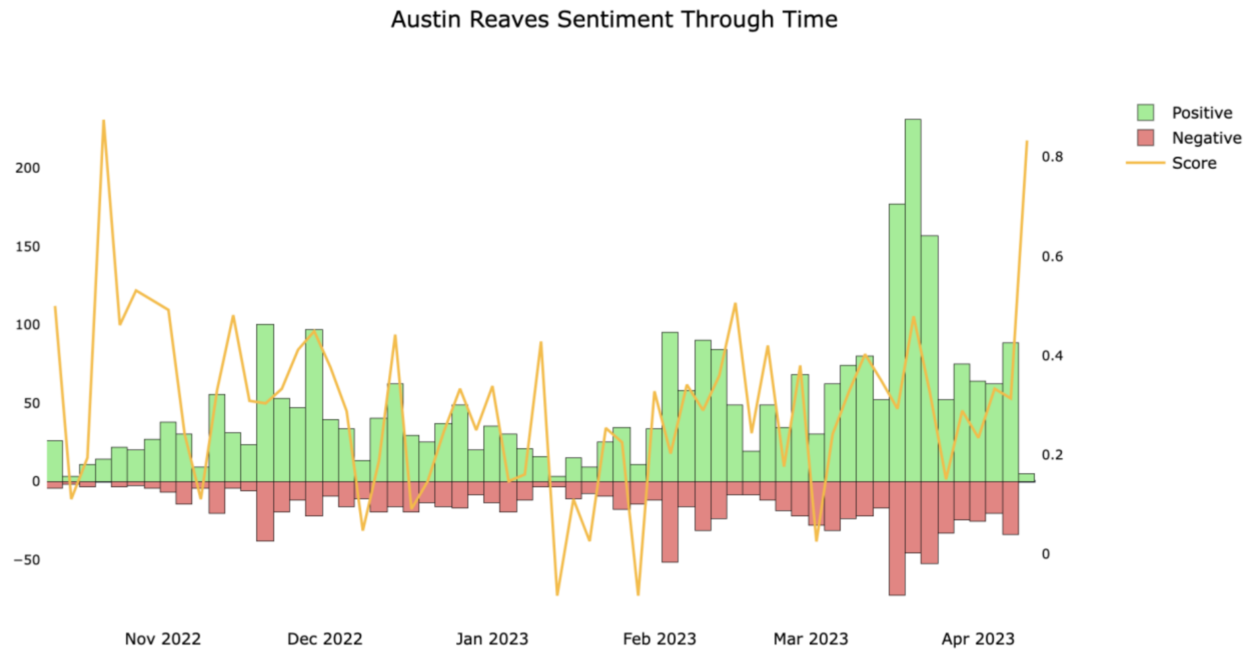


Figure 4: Austin Reaves Sentiment Through Time

Analyzing stats vs player sentiment is another way to learn more about how fans think and whose sentiment is most and least sensitive to both individual and team-based performance. Figure 5 plots the correlation between rolling 20-day player sentiment and rolling 20-day Net Rating. Net Rating is a stat that quantifies a team's point differential per 100 possessions while a certain player is on the floor. For example, if Anthony Davis played 1000 total possessions and the Lakers outscored opponents by 95 points in those possessions, his Net Rating would be +9.5. Net Rating is an easily quantifiable stat that does an adequate job determining how well a player is playing.

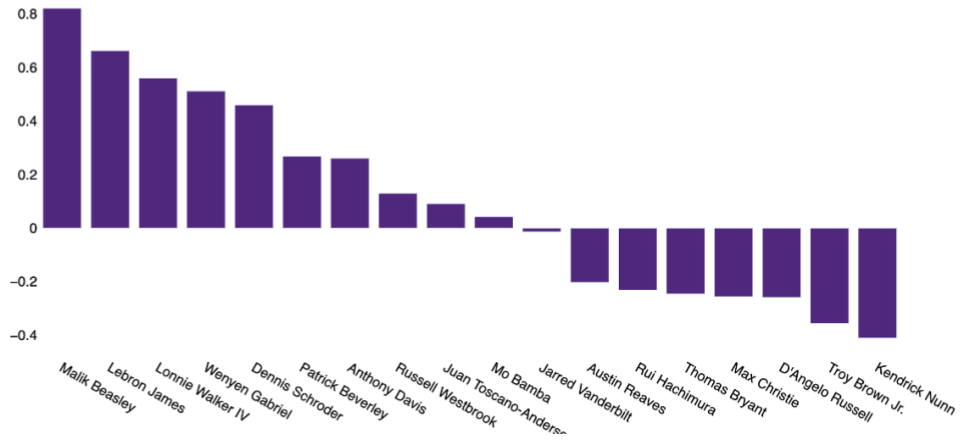


Figure 5: Player Sentiment-Net Rating Correlations

Malik Beasley, a solid yet unspectacular role player, has the highest sentiment-Net Rating correlation on the team. This could potentially be attributed to his high-variance play style as a high-volume, somewhat inconsistent three-point shooter. Additionally, his status as a new player acquired midseason likely contributes to this notion. It is not surprising to see fan sentiment toward star player LeBron James's being highly correlated with his on-court performance, but it is notable that the other team star, Anthony Davis, is more middle-of-the-pack in terms of sentiment correlation. On the other hand, players with negative correlation are mainly young players with future potential and generally high overall sentiment scores. Figure 6 below highlights the stark differences between Malik Beasley and Troy Brown Jr. in terms of sentiment-net rating correlation.

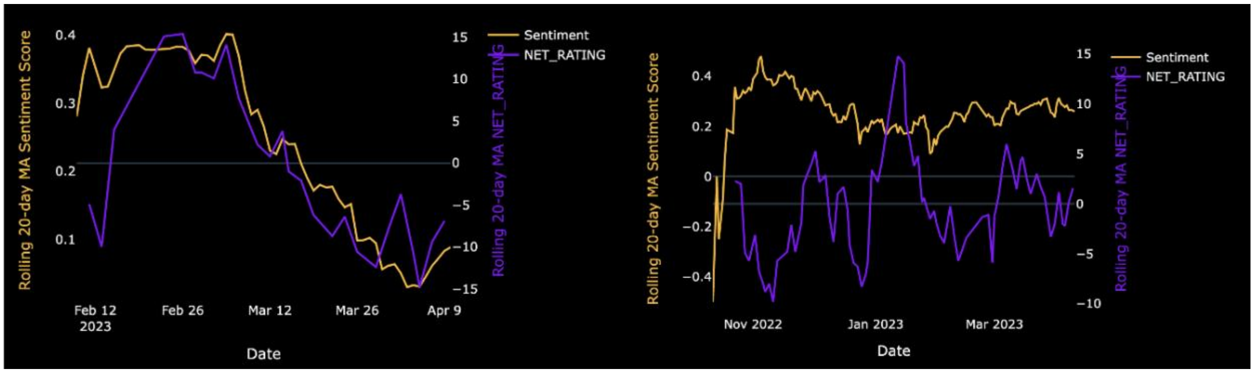


Figure 6: Malik Beasley vs. Troy Brown Jr. Sentiment-Net-Rating

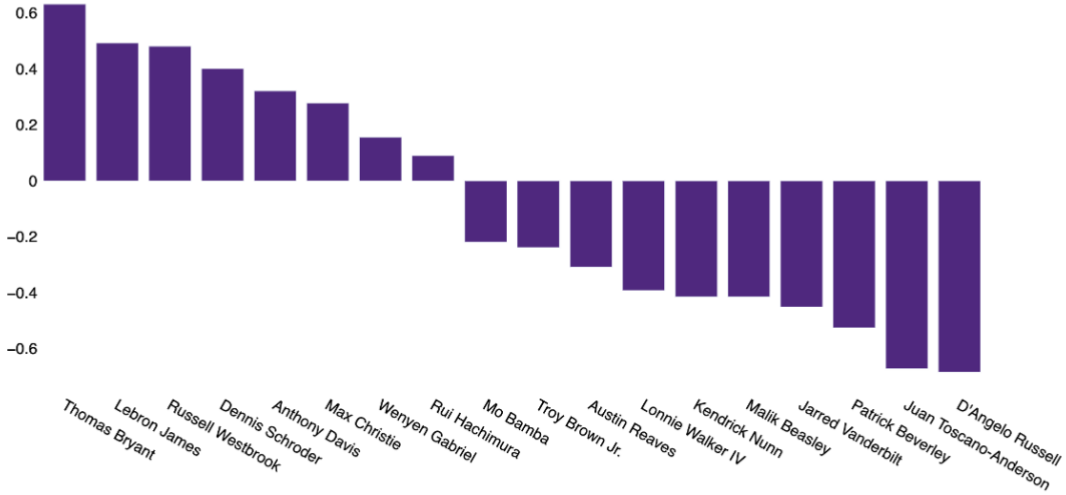


Figure 7: Player Sentiment-Team Wpct Correlations

Moving beyond individual statistics, Figure 7 measures each players' sentiment correlation with team winning percentage. At first glance, the plot appears to be evenly dispersed, with a slight negative skew. It is interesting to note that more players' sentiment is negatively correlated with team winning percentage than positively correlated. Sentiments towards the team's top players are amongst the most positively correlated with team winning percentage. Taken together, this indicates that sentiment towards star players is highly dependent on team performance, while sentiment towards role players is much more related to individual performance. Figure 8 displays two ends of this spectrum: Lebron James and D’Angelo Russell.

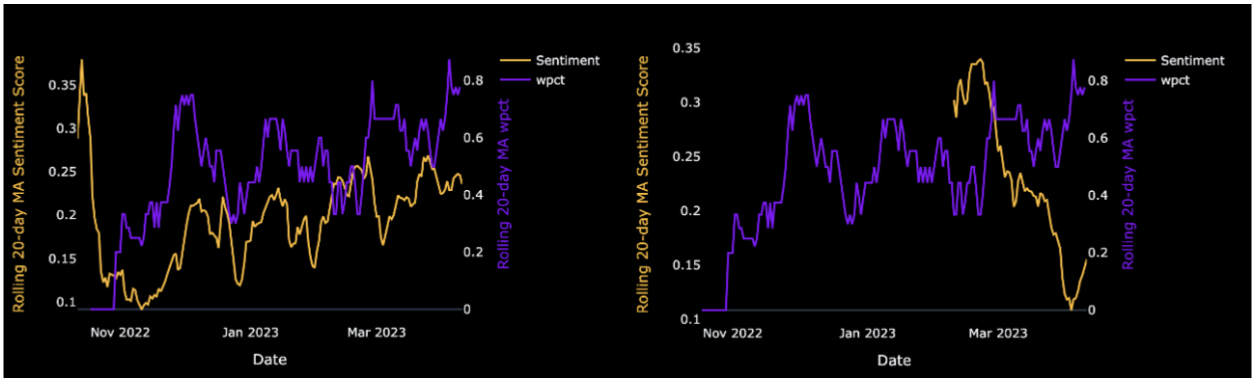


Figure 8: Lebron vs. D’Angelo Russell Sentiment-Team-Wpct

Entity-Level sentiment analysis provides far deeper and less obvious insights compared to traditional sentiment analysis with the examples above being just the tip of the iceberg. This type of analysis can provide valuable information regarding fan behavior and has potential as a key marketing tool.

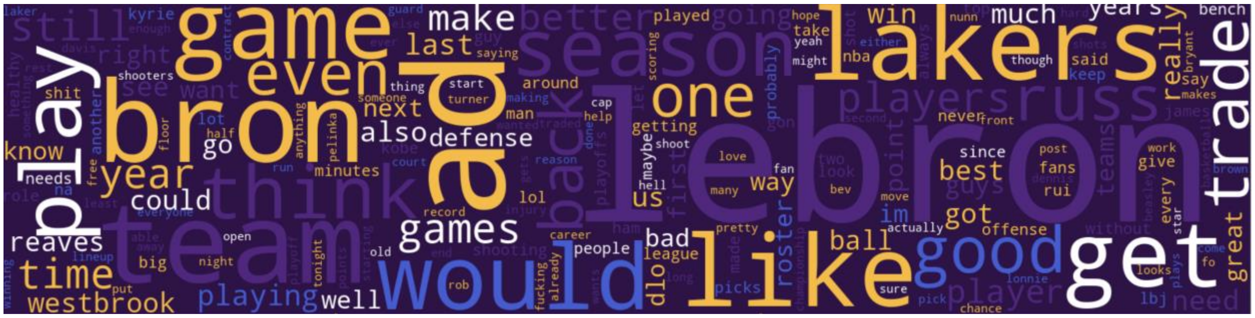
V. Future Research and Conclusion

This sentiment analysis project successfully achieved its objectives of testing and enhancing entity-level sentiment analysis techniques, examining the correlation between entity-level sentiment and basketball statistics, and providing valuable insights into fan behavior. Future research directions may include exploring more mathematically-based lexicon adjustments to further improve sentiment analysis accuracy and developing methods for intra-sentence sentiment detection, allowing for a deeper understanding of sentiment nuances.

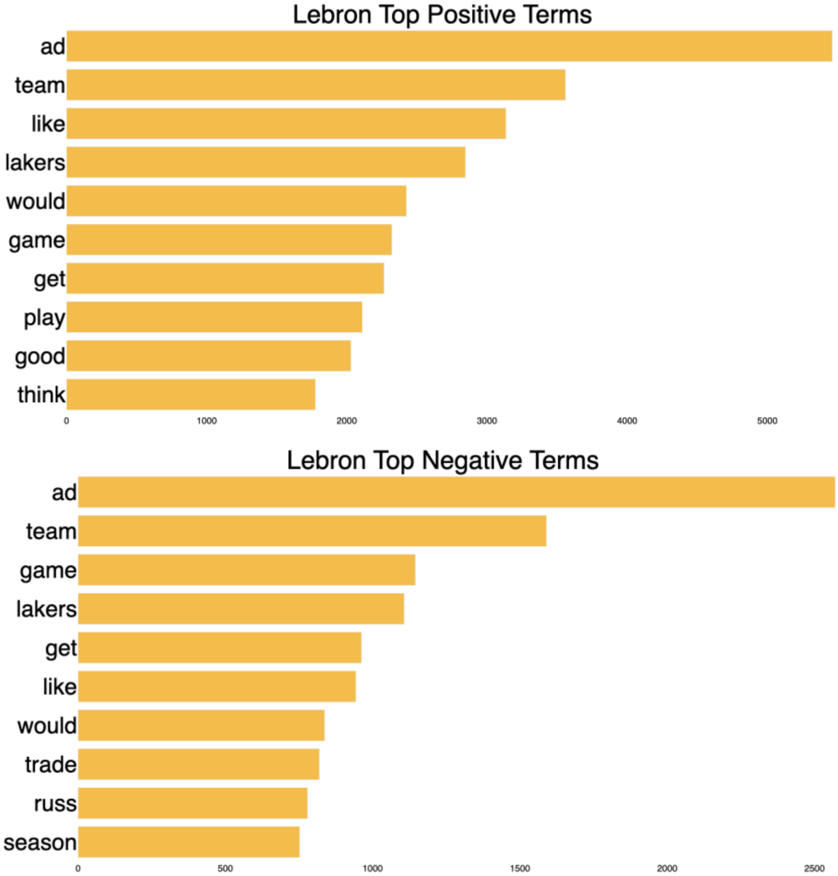
The findings of this project emphasize the power of player-level sentiment analysis in uncovering meaningful patterns in fan behavior, which can be instrumental in shaping effective marketing strategies. Moreover, entity-level sentiment analysis has broad applications beyond the realm of basketball, such as analyzing sentiment towards specific stocks on social media or gauging public sentiment towards potential Presidential candidates. The versatility of this analysis approach makes it a valuable tool in diverse domains and opens up countless opportunities for its application.

In conclusion, this sentiment analysis project has demonstrated the significance of entity-level sentiment analysis techniques and their potential impact on understanding fan sentiment and informing marketing decisions. By continuously refining and expanding these techniques, researchers and practitioners can unlock deeper insights into consumer behavior and sentiment dynamics across various domains, facilitating the development of targeted strategies and improving decision-making processes.

VI. Appendix Plots

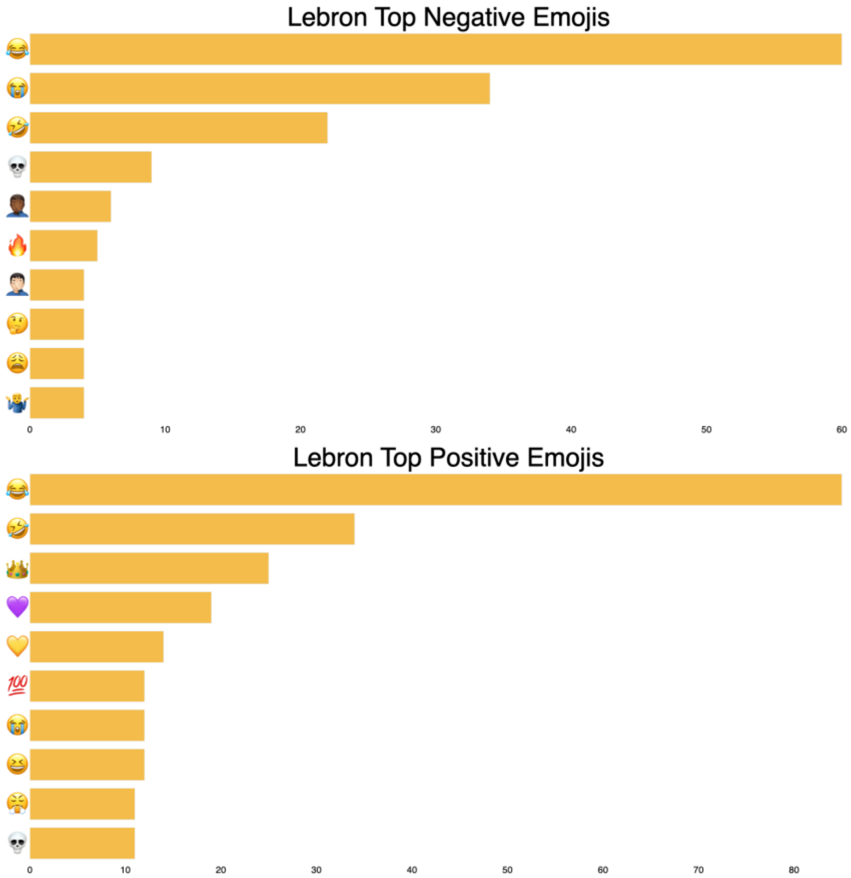


Appendix i: Lebron James WordCloud

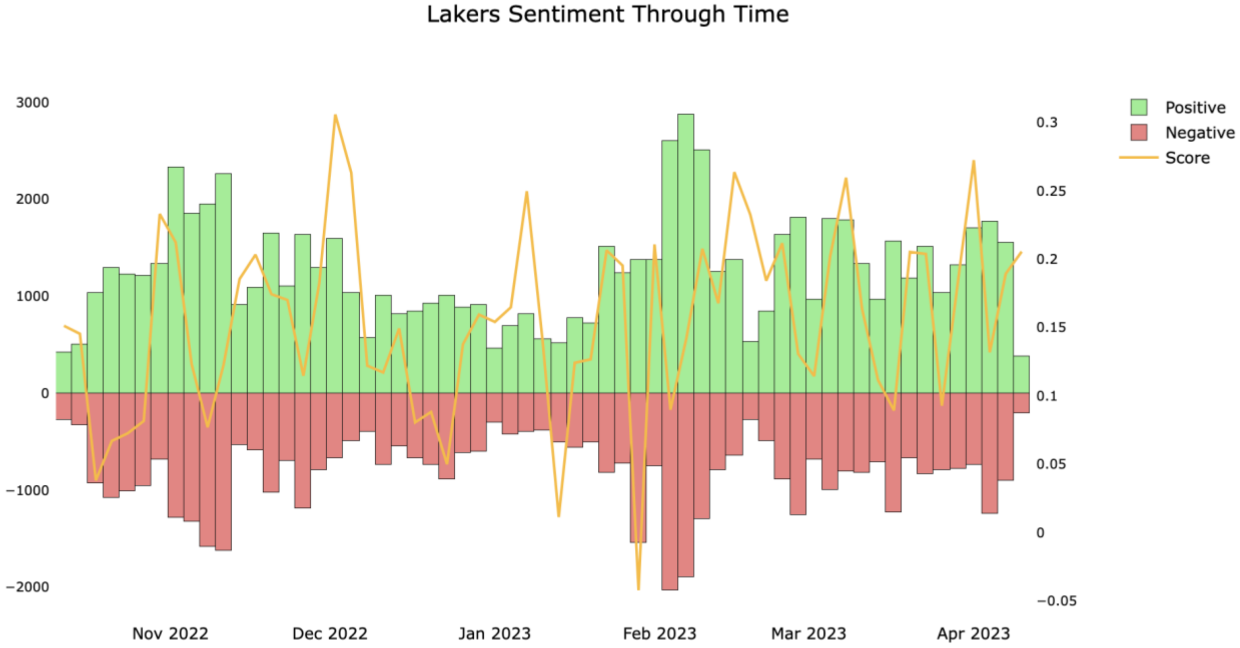

Lebron Top Positive Terms

Lebron Top Negative Terms

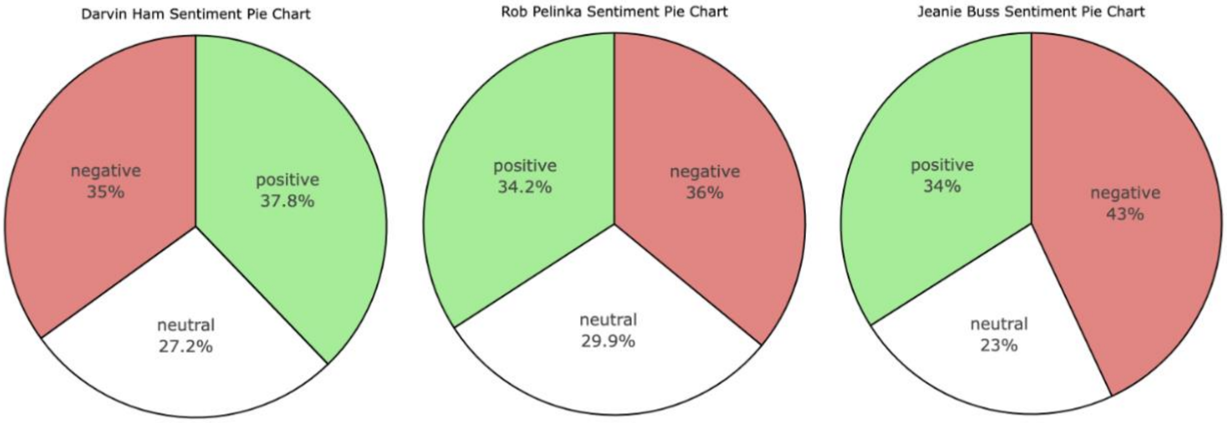
Appendix ii: Lebron James Top Positive/Negative Terms



Appendix iii: Lebron Top Positive/Negative Emojis



Appendix iv: r/Lakers Overall Sentiment Through Season



Appendix v: Lakers Coach, GM, Owner Sentiment Pie Charts

`

CODE

1) nbascrape.py: scrape nba stats

```
import pandas as pd
import json
import datetime as dt
import requests
import re
from time import sleep
from nba_api.stats.static import players
from nba_api.stats.endpoints import commonplayerinfo, teamgamelog, boxscoretraditionalv2, boxscoreadvancedv2,
teaminfocommon
from bs4 import BeautifulSoup


path = ("nba_data/")


### create list of players who played on lakers 2022-2023 season
player_list = [ 'Lebron James', 'Anthony Davis', 'D\'Angelo Russell', 'Dennis Schroder',
                'Austin Reaves', 'Russell Westbrook', 'Patrick Beverley', 'Troy Brown Jr.',
                'Jarred Vanderbilt', 'Malik Beasley', 'Lonnie Walker IV','Rui Hachimura',
                'Thomas Bryant', 'Wenyen Gabriel', 'Kendrick Nunn', 'Max Christie',
                'Juan Toscano-Anderson', 'Matt Ryan', 'Mo Bamba', 'Damian Jones',
                'Sterling Brown', 'Cole Swider', 'Scotty Pippen Jr.', 'Davon Reed'
              ]

other_team_ids = {'LAL':{'id':'1610612747', 'game_ids':[]},
                  'UTA':{'id':'1610612762', 'game_ids':[]},
                  'LAC':{'id':'1610612746', 'game_ids':[]},
                  'MIN':{'id':'1610612750', 'game_ids':[]},
                  'ORL':{'id':'1610612753', 'game_ids':[]},
                  'DEN':{'id':'1610612743', 'game_ids':[]},
                  'CHI':{'id':'1610612741', 'game_ids':[]},
                  'WAS':{'id':'1610612764', 'game_ids':[]}
                 }


def get_player_id_dict(player_list=player_list):
    """
    Given list of player names, fetches player ids and returns dict w player name and id.
    """
    player_dict = {}
    for p in player_list:
        try:
            p_info = players.find_players_by_full_name(p)
            player_dict[p] = p_info[0]['id']
        except:
            print("Could not fetch id for {}".format(p))
    return player_dict


def get_common_info(player_dict):
    """
    Given player dict with player name and id, returns df of common info for each player.

    Function also returns dictionary of player who were not loaded correctly.
    """
    common_info_dfs = []
    unsuccessful = {}
    for player, pid in player_dict.items():
        try:
            player_info = commonplayerinfo.CommonPlayerInfo(player_id=pid, timeout=2)
            common_info_dfs.append(player_info.get_data_frames()[0])
            print("{} successfully downloaded".format(player))
        except:
            print("{} failed to download!".format(player))
            unsuccessful[player] = pid

    return common_info_dfs, unsuccessful


def save_common_info(common_info_dfs):
    """
    Takes common_info_dfs list, concatenates them, and saves entire df to csv.
    """
    common_player_info = pd.concat(common_info_dfs)
    common_player_info = common_player_info[
    [
    'PLAYER_ID', 'FIRST_NAME', 'LAST_NAME', 'BIRTHDATE', 'SCHOOL',
    'HEIGHT', 'WEIGHT', 'SEASON_EXP', 'JERSEY', 'POSITION', 'DRAFT_YEAR',
    'DRAFT_ROUND', 'DRAFT_NUMBER'
    ]]

    common_player_info = common_player_info.reset_index().drop(columns='index')
    common_player_info.to_csv('common_player_info.csv')


def get_basic_game_logs(team_id, season=2022, season_type='Regular Season'):
    """
    Get basic game logs by team_id, beginning season year and season type.

    Parameters:
        team_id: nba.com team id; str
        season: year of beginning of season; int
        searon_type: one of: 'Regular Season', 'Playoffs', 'All-Star', 'All Star', 'Preseason'
    """
    if season_type not in ['Regular Season', 'Playoffs', 'All-Star', 'All Star', 'Preseason']:
        print("Invalid season type")
    else:
        tgl = teamgamelog.TeamGameLog(team_id=team_id,
                                      season_type_all_star=season_type,
                                      season=season)
        columns = tgl.get_dict()['resultSets'][0]['headers']
        tgl = pd.DataFrame(tgl.get_dict()['resultSets'][0]['rowSet'], columns=columns)

    return tgl


def get_team_box_scores(team_id, season=2022, traditional=True):
    """
    Given team_id and season start year, returns all team box scores for year.
```

```
Parameters:
    team_id: nba.com 10-digit team id
    season: starting year for desired season
    traditional: if true, returns traditional box scores, if false returns advanced; Default to True
"""
## get game ids ##
try:
    tgl = get_basic_game_logs(team_id=team_id, season=season)
    game_ids = tgl['Game_ID'].astype(str)
    print("game ids loaded")
except ValueError:
    print("Could not load game ids, please try again.")
sleep(3)

# for traditional box score stats
if traditional:
    # get cols and create empty df
    try:
        test = boxscoretraditionalv2.BoxScoreTraditionalV2(game_id=game_ids[0], timeout=3)
        columns = test.get_dict()['resultSets'][1]['headers']
        team_box_df = pd.DataFrame(columns=columns)
        print("Team box df created")

        # iterate through each game
        index = 0
        gid_no_load = []
        for gid in game_ids:
            try:
                sleep(1)
                game = boxscoretraditionalv2.BoxScoreTraditionalV2(game_id=gid, timeout=2)
                teams = game.get_dict()['resultSets'][1]['rowSet']

                # only upload stats for desired team, not opponent
                for team in teams:
                    if team[1] == team_id:
                        team_box_df.loc[index] = team
                        index += 1
```

```
                            print("{} successfully inserted for {}".format(team[2], gid))
                        else:
                            continue
                    print("Game {} complete".format(gid))
                except:
                    print("Could not load game_id: {}".format(gid))
                    gid_no_load.append(gid)
                sleep(3)
            return team_box_df, gid_no_load
        except:
            print("Could not create team box df")
            return game_ids
    else:
        # for advanced box score stats
        try:
            # get cols and create empty df
            test = boxscoreadvancedv2.BoxScoreAdvancedV2(game_id=game_ids[0], timeout=3)
            columns = test.get_dict()['resultSets'][1]['headers']
            team_box_df = pd.DataFrame(columns=columns)
            print("Team box df created")

            # iterate through each game
            index = 0
            gid_no_load = []
            for gid in game_ids:
                try:
                    sleep(1)
                    game = boxscoreadvancedv2.BoxScoreAdvancedV2(game_id=gid, timeout=2)
                    teams = game.get_dict()['resultSets'][1]['rowSet']

                    # only upload stats for desired team, not opponent
                    for team in teams:
                        if team[1] == team_id:
                            team_box_df.loc[index] = team
                            index += 1
                            print("{} successfully inserted for {}".format(team[2], gid))
                        else:
                            continue
                    print("Game {} complete".format(gid))
                except:
                    print("Could not load game_id: {}".format(gid))
                    gid_no_load.append(gid)
                sleep(3)
            return team_box_df, gid_no_load
        except:
            print("Could not create team box df")
            return game_ids



def get_player_box_scores(team_id, season=2022, traditional=True):
    """
    Given team_id and season start year, returns all player box score stats for year.

    Parameters:
        team_id: nba.com 10-digit team id
        season: starting year for desired season
        traditional: if true, returns traditional box scores, if false returns advanced; Default to True
    """
    ## get game ids ##
    try:
        tgl = get_basic_game_logs(team_id=team_id, season=season)
        game_ids = tgl['Game_ID'].astype(str)
        print("game ids loaded")
    except ValueError:
        print("Could not load game ids, please try again.")
    sleep(3)
    if traditional:
        try:
            # get cols and create empty df
            test = boxscoretraditionalv2.BoxScoreTraditionalV2(game_id=game_ids[0], timeout=2)
            columns = test.get_dict()['resultSets'][0]['headers']
            player_box_df = pd.DataFrame(columns=columns)
            print("Player box df created")
            index = 0
            gid_no_load = []
            for gid in game_ids:
                try:
                    game = boxscoretraditionalv2.BoxScoreTraditionalV2(game_id=gid, timeout=2)
                    players = game.get_dict()['resultSets'][0]['rowSet']
                    for player in players:
                        if str(player[1]) == team_id:
                            player_box_df.loc[index] = player
                            index += 1
                            print("{} successfully inserted for {}".format(player[5], gid))
                        else:
                            continue
                    print("Game {} complete".format(gid))
                except:
                    print("Could not load game_id: {}".format(gid))
                    gid_no_load.append(gid)
                sleep(3)
            return player_box_df, gid_no_load
        except:
            print("Could not create player box df")
            return game_ids
    else:
        try:
            # get cols and create empty df
            test = boxscoreadvancedv2.BoxScoreAdvancedV2(game_id=game_ids[0], timeout=2)
            columns = test.get_dict()['resultSets'][0]['headers']
            player_box_advanced_df = pd.DataFrame(columns=columns)
            print("Player box df created")
            index = 0
            gid_no_load = []
            for gid in game_ids:
                try:
                    game = boxscoreadvancedv2.BoxScoreAdvancedV2(game_id=gid, timeout=2)
                    players = game.get_dict()['resultSets'][0]['rowSet']
                    for player in players:
                        if str(player[1]) == team_id:
                            player_box_advanced_df.loc[index] = player
                            index += 1
                            print("{} successfully inserted for {}".format(player[5], gid))
                        else:
                            continue
                    print("Game {} complete".format(gid))
                except:
                    print("Could not load game_id: {}".format(gid))
                    gid_no_load.append(gid)
                sleep(3)
            return player_box_advanced_df, gid_no_load
        except:
            print("Could not create player box df")
            return game_ids

def get_common_team_info(teams):
    test = teaminfocommon.TeamInfoCommon(team_id=teams[0], season_nullable=2022).get_json()
    cols = json.loads(test)['resultSets'][0]['headers']

    tic_df = pd.DataFrame(columns=cols)
    index = 0
    for team in teams:
        js = teaminfocommon.TeamInfoCommon(team_id=team, season_nullable=2022).get_json()
        tic = json.loads(js)['resultSets'][0]
        tic_df.loc[index] = tic['rowSet'][0]

        index += 1



    return tic_df



def get_day_by_day_seedings(team, ew='w', start='2022-10-18', end='2023-04-09'):
    dt_start = dt.datetime.strptime(start, "%Y-%m-%d")
    dt_end = dt.datetime.strptime(end, "%Y-%m-%d")

    if ew == 'w':
        ew = 'Western Conference'
    else:
        ew = 'Eastern Conference'

    columns = [ew, 'W', 'L', 'W/L%', 'GB', 'PW', 'PL', 'PS/G', 'PA/G', 'RANK', 'DATE']

    current = dt_start
    seeding_on_date = pd.DataFrame(columns=columns)
    index = 0
    while current <= dt_end:
        url = "https://www.basketball-
reference.com/friv/standings.fcgi?month={}&day={}&year={}&lg_id=NBA".format(current.month, current.day,
current.year)
        x = requests.get(url)

        if str(x) == '<Response [429]>':
            print("Rate Limit Exceeded")
            return standings_on_date
            break



        soup = BeautifulSoup(x.content, "html.parser")
        tables = soup.find_all("table")

        if ew == 'Western Conference':
            standings = pd.read_html(str(tables[1]))[0]
        else:
            standings = pd.read_html(str(tables[0]))[0]

        standings[ew] = standings[ew].str.strip("*")



        team_row = standings[standings[ew] == team]
        rank = team_row.index[0] + 1
        date = dt.datetime.strftime(current, "%Y-%m-%d")

        insert = list(team_row.values[0])
        insert.append(rank)
        insert.append(date)

        seeding_on_date.loc[index] = insert

        current = current + dt.timedelta(days=1)
        index += 1

        print(str(date) + " complete")

        sleep(3)

    return seeding_on_date


def concat_all_team_data(abb, season, path=path):
    """
    Takes team info from tgl, traditional box and advanced box and concats into one df.
    """

    df = pd.read_csv(path + "tgl_{}.csv".format(abb), index_col=0)
    df2 = pd.read_csv(path + "team_box_traditional_{}.csv".format(abb), index_col=0)
    df3 = pd.read_csv(path + "team_box_advanced_{}.csv".format(abb), index_col=0)

    df4 = pd.concat([
    df[['GAME_DATE', 'MATCHUP', 'WL', 'W', 'L', 'W_PCT']],
    df2,
    df3.iloc[:, 6:]
    ],
    axis=1)

    df4['GAME_DATE'] = pd.to_datetime(df4['GAME_DATE'])

    df4 = df4.sort_values('GAME_DATE').reset_index().drop(columns=['index'])

    return df4
```

```python
def concat_player_data(path):
    """
    Concatenate individual box score stats for any non-lakers stats \
    for players who played on the lakers at some point during the season
    """
    teams = ls.load_json_file('teams.json', path=path)

    pbtt = pd.read_csv(path + "player_box_traditional_total.csv", index_col=0)
    pbtt['GAME_ID'] = '00' + pbtt['GAME_ID'].astype(str)
    pbat = pd.read_csv(path + "player_box_advanced_total.csv", index_col=0)
    pbat['GAME_ID'] = '00' + pbat['GAME_ID'].astype(str)

    df_full = pd.merge(pbtt,
                       pbat,
                       how='inner',
                       on=['GAME_ID', 'TEAM_ID', 'PLAYER_ID',
                           'TEAM_ABBREVIATION', 'TEAM_CITY', 'PLAYER_NAME'])

    true_min = []
    for m in df_full['MIN_x']:
        if type(m) == float:
            true_min.append(m)
        else:
            reg = re.findall(r"^(\d+):(\d+)", m)
            if len(reg) > 0:
                mins = float(reg[0][0])
                secs = float(reg[0][1])
                true_min.append(mins + secs / 60)
            else:
                reg = re.findall(r"^(\d+)\.(\d+)", m)
                mins = float(reg[0][0])
                secs = float(reg[0][1])
                true_min.append(mins + secs)
    df_full['MIN'] = true_min

    drop_cols = ['NICKNAME_x', 'START_POSITION_x', 'COMMENT_x', 'MIN_x',
                 'NICKNAME_y', 'START_POSITION_y', 'COMMENT_y', 'MIN_y']
```

```
df_full = df_full.drop(columns=drop_cols)

files = ["tgl_{}.csv".format(x) for x in list(teams.keys())[1:]]
tgl_dfs = [pd.read_csv(path + file, index_col=0) for file in files]
tgls = pd.concat(tgl_dfs)
tgls = tgls[['Game_ID', 'Team_ID', 'GAME_DATE', 'MATCHUP', 'WL']]
tgls['Game_ID'] = '00' + tgls['Game_ID'].astype(str)

df_fuller = pd.merge(
    df_full,
    tgls,
    how='left',
    left_on=['GAME_ID', 'TEAM_ID'],
    right_on=['Game_ID', 'Team_ID'])

return df_fuller
```

2) redditrequests.py: scrape reddit data

```
import pandas as pd # require pandas 1.5.3
import numpy as np
import decouple
import requests
import warnings
import os
import re
import datetime as dt
from datetime import datetime
from time import sleep
warnings.filterwarnings("ignore")



########### configure requests ###########
config = decouple.AutoConfig(' ')
key = config('APIKEY')
pub = config('PUBLICKEY')
user = config('USERNAME')
pw = config('PW')

auth = requests.auth.HTTPBasicAuth(pub, key)
data = {
   'grant_type': 'password',
   'username': user,
   'password': pw
}

headers = {'User-Agent': 'MYAPI/0.0.1'}
res = requests.post('https://www.reddit.com/api/v1/access_token',
                auth=auth, data=data, headers=headers)

TOKEN = res.json()['access_token']
headers = {**headers, **{'Authorization': f"bearer {TOKEN}"}}
```

```
###################################################################
def get_posts(subreddit, headers=headers, params= {'limit':100}):
    """
    Get up to 100 reddit posts and returns a df
    """
    res = requests.get("https://oauth.reddit.com/r/{}/new".format(subreddit),
                headers=headers, params=params)
    df = pd.DataFrame()

    for post in res.json()['data']['children']:
        # append relevant data to dataframe
        df = df.append({
            'subreddit': post['data']['subreddit'],
            'title': post['data']['title'],
            'selftext': post['data']['selftext'],
            'upvote_ratio': post['data']['upvote_ratio'],
            'ups': post['data']['ups'],
            'downs': post['data']['downs'],
            'score': post['data']['score'],
            'created_utc': datetime.fromtimestamp(post['data']['created_utc']).strftime('%Y-%m-%dT%H:%M:%SZ'),
            'id': post['data']['id'],
        }, ignore_index=True)

    return df


def get_more_posts(start, end, subreddit='lakers', limit=50):
    """ Get reddit posts back to the passed start_date.

        Parameters

        -start/end must be in YYYY-MM-DD string format

    """
    start = datetime.strptime(start, "%Y-%m-%d")
    start = int(start.timestamp())
    end = int(end.timestamp())
    end = datetime.strptime(end, "%Y-%m-%d")

    api_query = 'https://api.pushshift.io/reddit/submission/search/' \
                + '?subreddit={}&limit={}&after={}&before={}'.format(subreddit, limit, start, end)

    try:
        r = requests.get(api_query)
        json= r.json()
        df = pd.DataFrame(json['data'])

        df = df[['utc_datetime_str', 'id', 'title', 'author', 'selftext', 'upvote_ratio']]
        print("Successfully pulled data")
        return df
    except:
        print("Upload failed")




def get_comments(subreddit, tid, headers=headers):
    """ Given a subreddit and thread id, return all top_level comments in a df.
    """
    params = {'limit': 100}
    res = requests.get("https://oauth.reddit.com/r/{}/comments/{}".format(subreddit, tid),
                        headers=headers,
                        params=params)
    df = pd.DataFrame()
    comments = res.json()[1]['data']['children'] # note this gives top level comments only
    for i in range(len(comments)):
        comment = comments[i]['data']

        df = df.append({
            'id': comment['id'],
            'author': comment['author'],
            'pid': comment['parent_id'][3:],
            'body': comment['body'],
            'upvotes': comment['ups'],
            'downvotes': comment['downs']
        }, ignore_index=True)


    return df


def get_all_replies(tid, cid, headers=headers):
    """ Given a thread id and comment id, return all replies in a dataframe.
    """
    params = {'limit': 100}
    res =
requests.get('http://oauth.reddit.com/api/morechildren?link_id=t3_{}&children={}&api_type=json'.format(tid,cid),
                       headers=headers, params=params)
    all_comments = res.json()['json']['data']['things']

    df = pd.DataFrame()
    for comment in all_comments:
        df = df.append({
            'id': comment['data']['id'],
            'author': comment['data']['author'],
            'pid': comment['data']['parent_id'][3:],
            'body': comment['data']['body'],
            'upvotes': comment['data']['ups'],
            'downvotes': comment['data']['downs']
        }, ignore_index=True)

    return df


def get_all_comments(subreddit, tid, date, headers=headers):
    """ Given a subreddit and thread, return all comments and replies in a dataframe.
    """
    params = {'limit': 100}
    df_comments = get_comments(subreddit, tid)
    for id in df_comments.id:
        df_replies = get_all_replies(tid, id)
        if df_replies.shape[0] > 1:
            df_comments = pd.concat([df_comments, df_replies.iloc[1:]], axis=0)

    df_comments['tid'] = tid
    second_column = df_comments.pop('tid')
    df_comments.insert(0, 'tid', second_column)

    df_comments['datetime'] = date
    first_column = df_comments.pop('datetime')
    df_comments.insert(0, 'datetime', first_column)
    df_comments['datetime'] = pd.to_datetime(df_comments['datetime'])



    return df_comments.reset_index().drop(columns=['index'])

def try_daily_post_upload(start, end, subreddit='lakers', folder='data/daily_posts/', base ='r_lakers_', limit=50):
    """
    The PushShift API is somewhat unreliable, so this function attempts to pull data
    one day at a time to ensure nothing is missed. The function saves daily csvs to specified folder.

    The function also returns a list of dates that were not successfully pulled so that they can be re-tried.

    Parameters

        -start and end specify total range of data you want pulled
        -start/end must be in YYYY-MM-DD string format
        -folder is the root of data folder
        -base is base filename for saved csvs
        -limit is the max posts returnd



    """
    start_dt = datetime.strptime(start, '%Y-%m-%d')
    end_dt = datetime.strptime(end, '%Y-%m-%d')
    diff = (end_dt - start_dt).days
```

```python
    dates = [end_dt - dt.timedelta(days=x) for x in range(diff)]
    dates_str = [date.strftime('%Y-%m-%d') for date in dates]
    bad_dates = []
    for date in dates:
        start = date # note this is a different start than original since we pull for each day
        end = datetime.strptime(start, '%Y-%m-%d') + dt.timedelta(days=1) # convert to dt to add day
        end = datetime.strptime(end, '%Y-%m-%d') # convert back to string so works in function
        df = get_more_posts(start, end, subreddit=subreddit, limit=limit)
        try:
            start_string = start.replace('-','_') # change format to save
            df.to_csv(folder + base + start_string)
            print("Successfully uploaded data for {}".format(start_string))
        except:
            print("Failed to Upload Data for {}".format(start_string))
            bad_dates.append(start_string.replace('_', '-'))

    return bad_dates



def check_for_max_posts(limit = 50):
    """
    The PushShift API is rather unreliable and fails often and therefore different \
    limits are tried to pull data. Sometimes days that successfully loaded hit the \
    limit that was used. This function checks the df lengths to return dates that \
    hit the limit. You can then run the get_more_posts function with a higher limit \
    on the returned list.

    """
    directory = '/Users/dylanjorling/NBASA_reddit/data/daily_posts'
    files = os.listdir(directory)
    files = [x for x in files if len(re.findall(r"r_lakers", x)) == 1]
    max_post_files = []
    for file in files:
        try:
            x = pd.read_csv('data/daily_posts/' + file)
            if x.shape[0] == limit:
                max_post_files.append(file)
        except:
            continue
    max_post_files_date = [file[9:] for file in max_post_files]
    max_post_files_date = [file.replace('_', '-') for file in max_post_files_date]

    return max_post_files_date

def concat_daily_files(folder_daily='/Users/dylanjorling/NBASA_reddit/data/daily_posts/',
                       base='r_lakers_', folder_full='data/', name='full_posts.csv'):
    """
    This function takes in folder with daily csvs, concatenates them together and saves entire csv

    Parameters
        -folder_daily: folder containing daily posts
        -base: base file name for all csv files
        -folder_full: file location
        -name: desired name of full csv file

    Requires: pandas, os, re
    """

    # filter out unwanted files
    files = os.listdir(folder_daily)
    files = [x for x in files if len(re.findall(r"r_lakers", x)) == 1]

    # create list of dfs
    df_list = []
    for file in files:
        df_list.append(pd.read_csv('data/daily_posts/' + file, index_col=0))

    # concatenate df
    df_full = pd.concat(df_list, ignore_index=True)

    # clean df
    df_full.reset_index()
    df_full=df_full.rename(columns = {'utc_datetime_str':'datetime'})
    df_full['datetime'] = pd.to_datetime(df_full['datetime'])
    #df_full = df_full[df_full['datetime'] >= '2022-10-13'] # optional
    df_full = df_full.sort_values('datetime')
    df_full.set_index('datetime', inplace=True)
    df_full.reset_index(inplace=True)

    # save to folder
    df_full.to_csv(folder_full + base + name)



def save_monthly_comment_history(concat_daily_files_df,
                                 subreddit='lakers',
                                 folder='data/'):
    """
    Given a df of posts returned by concat_daily_files, gets all comments for each post \
    using the get_all_comments function, concatenates all comments for each month, and \
    saves dataframe to csv for each month. Also returns thread ids that did not load

    Parameters
        -concat_daily_files_df: dataframe in format that the concat_daily_files function saves to csv
        -subreddit: subreddit where posts were mad
        -folder:
    """
    df = concat_daily_files_df
    df['month'] = df['datetime'].dt.month_name()

    # iterate through unique months
    bad_ids = []
    for month in set(df['month']):
        # create monthly dfs
        df_int = df[df['month'] == month]

        # define empty df to store comments
        columns = ['datetime', 'tid', 'id', 'author', 'pid', 'body', 'upvotes', 'downvotes']
        dfc_month = pd.DataFrame(columns=columns)

        # iterate through each row and append to month
        for i, row in df_int.iterrows():
            try:
                dfc = rr.get_all_comments(tid=row.id,
                                          date=row.datetime,
                                          subreddit=subreddit)
                dfc_month = pd.concat([dfc_month, dfc])
                print("tid: {} date: {} successfully loaded".format(row.id, row.datetime))
            except:
                print("tid: {} date: {} failed to load".format(row.id, row.datetime))
                bad_ids.append(row.id)


        dfc_month.to_csv(name + 'monthly_comment_hist/{}_comments.csv'.format(month))
    return bad_ids
```

3) text_cleaning.py: all text cleaning related functions

```
import pandas as pd
import numpy as np
import datetime as dt
import math
import re
import itertools
import emoji
import spacy
import json
from spacy.pipeline import EntityRuler
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

eng_stopwords = stopwords.words('english')
nlp = spacy.load("en_core_web_sm")

def clean_gifs_hyperlinks_emojis(text):
    clean_text = str(text)
    clean_text = re.sub(r"!gif\(giphy.*\)", "", clean_text) # remove gifs
    clean_text = re.sub(r"\(https[^\s]*\)", " ", clean_text) # remove embedded hyperlinks
    clean_text = re.sub(r"https[^\s]*", " ", clean_text) # remove remaining hyperlinks
    clean_text = re.sub(r"amp;", " ", clean_text)

    return clean_text

def basic_clean(text):
    clean_text = str(text)
    clean_text = re.sub(r"’s|'s", " is", clean_text)
    clean_text = re.sub(r"’re|'re", " are", clean_text)

    # other
    clean_text = re.sub(r"i've", "i have", clean_text)
    clean_text = re.sub(r"i'd", "i would", clean_text)
    clean_text = re.sub(r"i'm", "i am", clean_text)
    clean_text = re.sub(r"i'll", "i will", clean_text)
    clean_text = re.sub(r"\bill\b", "i will", clean_text)

    # leave punctuation
    clean_text = re.sub(r"["’\[\]#,]", "", clean_text)
    clean_text = re.sub(r"[/\-_+&\\]", " ", clean_text)

    return clean_text

def clean_post_body(text):
    clean_text = str(text)
    clean_text = re.sub(r"\bnan\b", "", clean_text)
    clean_text = re.sub(r"\bremoved\b", "", clean_text)
    clean_text = re.sub(r"\bdeleted\b", "", clean_text)

    return clean_text

def clean_spaces(text):
    clean_text = str(text)
    clean_text = re.sub(r'\s{2,}', ' ', clean_text)
    clean_text = clean_text.lstrip()
    clean_text = clean_text.rstrip()

    return clean_text

def clean_all_text(text):
    clean_text = str(text)
    clean_text = clean_text.lower()
    clean_text = clean_gifs_hyperlinks_emojis(clean_text)
    clean_text = basic_clean(clean_text)
    clean_text = clean_post_body(clean_text)
    clean_text = clean_spaces(clean_text)
    clean_text = clean_gifs_hyperlinks_emojis(clean_text) # for good measure

    return clean_text

def clean_random(text):
    clean_text = re.sub(r"\bgt;", "", text)
    clean_text = re.sub(r"", "", clean_text)

    return clean_text

def roll_back_ll(text):
    clean_text = re.sub(r"\bwi\swill\b", "will", text)
    clean_text = re.sub(r"\bsti\swill\b", "still", clean_text)
    clean_text = re.sub(r"\bki\swill\b", "kill", clean_text)
    clean_text = re.sub(r"\bski\swill\b", "skill", clean_text)
    clean_text = re.sub(r"\bwon\snot\b", "will not", clean_text)
    clean_text = re.sub(r"\bisn\snot\b", "is not", clean_text)
    clean_text = re.sub(r"\bwasn\snot\b", "was not", clean_text)

    return clean_text

def clean_punctuation(text):
    clean_text = re.sub(r"[\.\?!\"]", "", text)

    return clean_text
```

```
def rm_stopwords(word_list):
    no_stop = [x for x in word_list if x not in eng_stopwords]

    return no_stop

def well_to_good(text):

    clean_text = re.sub(r"(\bplay\w*\s)(well\b)", r" \1" + "good", text)
    clean_text = re.sub(r"(\bplay\w*\s\w*\s)(well\b)", r" \1" + "good", clean_text)
    clean_text = re.sub(r"(\bshot\s)(well\b)", r" \1" + "good", clean_text)
    clean_text = re.sub(r"(\bshot\s\w*\s)(well\b)", r" \1" + "good", clean_text)
    clean_text = re.sub(r"(\bshoot\w*\s)(well\b)", r" \1" + "good", clean_text)
    clean_text = re.sub(r"(\bshoot\w*\s\w*\s)(well\b)", r" \1" + "good", clean_text)
    clean_text = re.sub(r"(\bdefen\w*\s)(well\b)", r" \1" + "good", clean_text)
    clean_text = re.sub(r"(\bdefen\w*\s\w*\s)(well\b)", r" \1" + "good", clean_text)

    return clean_text

def clean_bench_trade_start(player_df):
    full_name = player_df.player_ref[0]
    clean = []
    for text in player_df['resolved']:
        clean_text = re.sub(
            r"\bbench\s{}\b".format(full_name),
            "mid {}".format(full_name),
            text
        )
        clean_text = re.sub(
            r"\btrade\s{}\b".format(full_name),
            "mid {}".format(full_name),
            clean_text
        )
        clean_text = re.sub(
            r"\bstart\s{}\b".format(full_name),
            "solid {}".format(full_name),
            clean_text
        )
        clean.append(clean_text)

    player_df['resolved_final'] = clean

    return player_df

def replace_dual_meaning(text, token, replace, pos_list, reverse=False, nlp=nlp):
    doc = nlp(text)

    if reverse:
        new_toks = [tok for tok in doc if ((str(tok) == token) & (tok.pos_ in pos_list)) | (str(tok) != token)]
        clean_text = " ".join([token.text for token in new_toks])

    else:
        new_toks = [tok for tok in doc if ((str(tok) == token) & (tok.pos_ not in pos_list)) | (str(tok) != token)]
        clean_text = " ".join([token.text for token in new_toks])

#    print("done!")

    return clean_text

def find_text(pattern, df_column):
    """
    Iterates through post/comment df and prints out text matches.
    Use this to spot-check certain patterns.
    """
    find_text = []

    for i, t in enumerate(df_column):
        if re.search(pattern, t) != None:
            print(i)
            find_text.append(t)
    print("Length:" + str(len(find_text)))
    for i, t in enumerate(find_text):
        print(i)
        print(t)
        print()
    return find_text


def check_pos(token, check_df, pos_list, reverse=True):
    x = find_text(r"\b{}\b".format(token), check_df)
    for i, text in enumerate(x):
        doc = nlp(text)
        for tok in doc:
            if str(tok) == token:
                print(f'Index: {i} Text: {tok.text} Part-of-speech: {tok.pos_}')

    for i, text in enumerate(x):
        doc = nlp(text)

    for tok in doc:
        if (reverse) & (str(tok) == token) & (tok.pos_ in pos_list):
            print(f'Index: {i} Text: {tok.text} Part-of-speech: {tok.pos_}')
            print(text)
            print()

        elif (reverse==False) & (str(tok) == token) & (tok.pos_ not in pos_list):
            print(f'Index: {i} Text: {tok.text} Part-of-speech: {tok.pos_}')
        else:
            continue


def pos_clean(df_body):
    pos_clean = [
        {'token':'low',
         'replace':'',
         'pos_list':['ADV'],
         'reverse':False
        },
        {'token':'like',
         'replace':'',
         'pos_list':['VERB'],
         'reverse':True
        },
        {'token':'fire',
         'replace':'',
         'pos_list':['NOUN'],
         'reverse':False
        },
        {'token':'hell',
         'replace':'',
         'pos_list':['INTJ'],
         'reverse':False
        },
        {'token':'hell',
         'replace':'he will',
         'pos_list':['PROPN'],
         'reverse':False
        },
        {'token':'limit',
         'replace':'',
         'pos_list':['NOUN'],
         'reverse':False
        },

    ]

    for t in pos_clean:
        df_body = df_body.apply(replace_dual_meaning,
                                token=t['token'],
                                replace=t['replace'],
                                pos_list=t['PROPN'],
                                reverse=t['reverse']
                               )
    return df_body

def get_emoji_dict(posts, comments):
    emoji_dict = {}
    text_dfs = [posts, comments]
    for text_df in text_dfs:
        strings = text_df.body
        for string in strings:
            matches = emoji.emoji_list(string)
            text_emojis = []
            for i, _ in enumerate(matches):
                text_emojis.append(matches[i]['emoji'])
            unique_text_emojis = list(set(text_emojis))
            for emj in unique_text_emojis:
                if emj not in emoji_dict.keys():
                    emoji_dict[emj] = 1
                else:
                    emoji_dict[emj] += 1
    return emoji_dict

def extract_emojis(text):
    matches = emoji.emoji_list(text)
    if len(matches) == 0:
        return np.nan
    else:
        text_emojis = []
        for i, _ in enumerate(matches):
            text_emojis.append(matches[i]['emoji'])
        unique_text_emojis = list(set(text_emojis))

        return " ".join([str(emj) for emj in unique_text_emojis])

def remove_emojis(text):
    matches = emoji.emoji_list(text)
    for i, _ in enumerate(matches):
        emj = matches[i]['emoji']
        text = re.sub(emj, "", text)
    clean_text = text

    return clean_text
```

```
def get_emoji_col(df):
    """
    adds concatenated emoji col to post/comment df
    """
    df['emojis'] = df.body.apply(extract_emojis)
    return df

def sub_emojis(text, subbed_emoji, sub_emoji):
    clean_text = re.sub(subbed_emoji, sub_emoji, text)

    return clean_text




def create_player_dict(patterns, trade=0, patterns_trade=None, trade_date=None):
    """
    Use regex patterns to extract posts/comments with each unique player reference.
    Returns a dictionary with each unique reference and the post or comment ids the \n
    reference is contained in.

    Parameters
    __________
    patterns: list of patterns to search through
    trade: parameter to specify extraction of certain patterns only before/aftter a trade date
    patterns_trade: patterns to be searched for only before/after trade date if specified
    trade_date: cutoff date, should be in '%Y-%m-%d' form
    """
    if trade not in [0, 1, 2]:
        raise ValueError("Trade must be 0 (not traded), 1 (traded for) or 2 (traded away)")
    if (trade in [1, 2]) & (patterns_trade == None):
        raise ValueError("You have indicated you want before/after trade data. Please specify patterns")
    if (trade in [1, 2]) & (trade_date == None):
        raise ValueError("You have indicated you want before/after trade data. Enter trade_date in '%Y-%m-%d'")
    if trade_date != None:
        check_date = re.findall(r"\b\d{4}-\d{2}-\d{2}", trade_date)
        if len(check_date) != 1:
            raise ValueError("trade_date in wrong form, should be in '%Y-%m-%d'")



    player_dict = {}
    posts = pd.read_csv("data/clean_posts.csv", index_col=0, parse_dates=['datetime'])
    comments = pd.read_csv("data/clean_comments.csv", index_col=0, parse_dates=['datetime'])
    text_dfs = [posts, comments]

    for text_df in text_dfs:
        text_df.body = text_df.body.apply(str)
        for _, row in text_df.iterrows():
            for x in patterns:
                y = re.findall(x, row.body)
                for i in y:
                    if i not in player_dict.keys():
                        player_dict[i] = []
                        player_dict[i].append(row.id)
                    elif row.id not in player_dict[i]:
                        player_dict[i].append(row.id)
                    else: continue
    if trade == 0:
        for k, v in player_dict.items():
            print(k, len(v))
        return player_dict

    else:
        if trade == 1: # if 1, then we only want patterns after a trade (indicates traded for)
            posts_trade =  posts[posts['datetime'] >= trade_date]
            comments_trade = comments[comments['datetime'] >= trade_date]

        else:
            posts_trade =  posts[posts['datetime'] < trade_date]
            comments_trade = comments[comments['datetime'] < trade_date]

        text_dfs_trade = [posts_trade, comments_trade]
```

```
        for text_df in text_dfs_trade:
            text_df.body = text_df.body.apply(str)
            for _, row in text_df.iterrows():
                for x in patterns_trade:
                    y = re.findall(x, row.body)
                    for i in y:
                        if i not in player_dict.keys():
                            player_dict[i] = []
                            player_dict[i].append(row.id)
                        elif row.id not in player_dict[i]:
                            player_dict[i].append(row.id) # assure no duplicate ids for any name
                        else:
                            continue
        for k, v in player_dict.items():
            print(k, len(v))

        return player_dict

def create_player_post_df(player_dict, name_id):
    """
    Returns df for given player dict containing post details for every post player is mentioned
    """
    # load clean posts
    posts = pd.read_csv("data/posts_clean_extended_final_pos_coref.csv", index_col=0, parse_dates=['datetime'])
    posts.body = posts.body.apply(str)
    posts.body_pr = posts.body_pr.apply(str)
    posts.body_coref = posts.body_coref.apply(str)

    columns = list(posts.columns)
    columns.append('unique_ref')
    columns.append('player_ref')
    df_post = pd.DataFrame(columns=columns)
    df_post.loc[0] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    index1 = 0
    for k, v in player_dict.items():
        for i in player_dict[k]:
            if i in list(df_post.id):
                # if this player_ref is already in the ref column for this post,
                row1 = df_post[df_post['id'] == i]
                idx1 = row1.index[0]

                if k in row1.unique_ref[idx1].split("/"):
                    pass # dont double count if uses same specifc ref more than once
                else:
                    row1 = row1.loc[idx1] # change row from df to series
                    row1.unique_ref = row1.unique_ref + "/" + k
                    df_post.loc[idx1] = list(row1)


            elif i in list(posts.id):
                idx2 = posts[posts.id== i].index[0]
                row = list(posts[posts.id== i].loc[idx2])
                row.append(k)
                row.append(name_id)
                df_post.loc[index1] = row
                index1 += 1

            else:
                pass # the dict contains both posts and comments; if not a comment, pass

    return df_post


def create_player_comment_df(player_dict, name_id):
    """
    Returns df for given player dict containing comment details for every comment player is mentioned
    Parameters:
        player_dict: dict created by create_player_dict function
        name_id: single name id for player
    """
    # load clean comments
    comments = pd.read_csv("data/comments_clean_extended_final_pos.csv", index_col=0, parse_dates=['datetime'])
    comments.body = comments.body.apply(str)
    columns = list(comments.columns)
    columns.append('unique_ref')
    columns.append('player_ref')
    df_comments = pd.DataFrame(columns=columns)
    df_comments.loc[0] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    index1 = 0
    for k, v in player_dict.items():
        for i in player_dict[k]:
            if i in list(df_comments.id):
                # if this player_ref is already in the ref column for this post,
                row1 = df_comments[df_comments['id'] == i]
                idx1 = row1.index[0]
                if k in row1.unique_ref[idx1].split("/"):
                    pass # dont double count if uses same specifc ref more than once
                else:
                    row1 = row1.loc[idx1]
                    row1.unique_ref = row1.unique_ref + "/" + k
                    df_comments.loc[idx1] = list(row1)


            elif i in list(comments.id):
                idx2 = comments[comments.id== i].index[0]
                row = list(comments[comments.id== i].loc[idx2])
                row.append(k)
                row.append(name_id)
                df_comments.loc[index1] = row
                index1 += 1

            else:
                pass # the dict contains both posts and comments; if not a comment, pass

    return df_comments


def save_player_dfs(player_dict, name):
    with open('data/full_names.json', 'r') as f:
        full_names_json = json.load(f)
    full_names = dict(full_names_json)
    df_post = create_player_post_df(player_dict, full_names[name])
    df_comment = create_player_comment_df(player_dict, full_names[name])
    df_post.to_csv("data/{}_refs_posts.csv".format(name))
    df_comment.to_csv("data/{}_refs_comments.csv".format(name))


def add_post_entities():
    # load entities dict
    with open('data/entities.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    # load dfs
    posts = pd.read_csv("data/clean_posts.csv", index_col=0, parse_dates=['datetime'])
    posts.body = posts.body.apply(str)

    # reindex for easier slicing and add empty cols
    posts = posts.set_index('id')
    posts['entities'] = ""
    posts['num_entities'] = 0

    # load player post dfs
    df_post_list = [pd.read_csv("data/{}_refs_posts.csv".format(x), index_col=0, parse_dates=['datetime']) for x in
entities.keys()]


    for df in df_post_list:
        idxs = list(df.id)
        name = df.player_ref[0]

        # update entity cols
        posts.loc[idxs, "entities"] =  posts.loc[idxs, "entities"] + name + ", "
        posts.loc[idxs, "num_entities"] += 1

    return posts
```

```python
def get_cleaned_ents(text, rtrn='l'):
    if rtrn not in ['l', 'd', 'c']:
        raise ValueError("retrn variable must be in l (list), d(dict), or c(count)")

    # load entity dic and get references list
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
        entities = dict(entities)

    refs = [entities[x]['full_name'] for x in list(entities.keys())[:-1]]

    unique_refs = []
    unique_ref_w_count = {}
    for ref in refs:
        res = re.findall(r"\b{}\b".format(ref), text)
        if len(res) > 0:
            unique_refs.append(res[0])
            unique_ref_w_count[res[0]] = len(res)

    # convert list, dict to strings
    count = len(unique_refs)
    unique_refs_string = ",".join(x for x in unique_refs)
    unique_ref_w_count_string = json.dumps(unique_ref_w_count)

    if rtrn == 'l':
        return unique_refs_string
    elif rtrn == 'd':
        return unique_ref_w_count
    else:
        return count



def add_tid_ents(comments, posts):
    """
    Use posts to get comment tid ents and tid num ents
    """
    tid_entities_dict = {}
    tid_num_entities_dict = {}
    for _, row in posts.iterrows():
        tid_entities_dict[row.id] = 0
        tid_entities_dict[row.id] = (row.entities)
        tid_num_entities_dict[row.id] = 0
        tid_num_entities_dict[row.id] = (row.num_entities)

    tid_entities = []
    tid_num_entities = []
    for _, row in comments.iterrows():
        tid = row.tid
        tid_ents = tid_entities_dict[tid]
        tid_num_ents = tid_num_entities_dict[tid]
        tid_entities.append(tid_ents)
        tid_num_entities.append(tid_num_ents)


    comments['tid_entities'] = tid_entities
    comments['tid_num_entities'] = tid_num_entities


    return comments

def add_comment_entities():
    # load entities dict
    with open('data/entities.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    # load dfs
    comments = pd.read_csv("data/comments_with_tid_ents.csv", index_col=0, parse_dates=['datetime'])
    comments.body = comments.body.apply(str)

    # reindex for easier slicing and add empty cols
    comments = comments.set_index('id')
    comments['entities'] = ""
    comments['num_entities'] = 0
    # load player post dfs
    df_comment_list = [pd.read_csv("data/{}_refs_comments.csv".format(x), index_col=0, parse_dates=['datetime']) for
x in entities.keys()]


    for df in df_comment_list:
        idxs = list(df.id)
        name = df.player_ref[0]

        # update entity cols
        comments.loc[idxs, "entities"] =  comments.loc[idxs, "entities"] + name + ", "
        comments.loc[idxs, "num_entities"] += 1

    return comments


def add_parent1_comment_entities(df_comments):

    df_comments['parent1_entities'] = ""
    df_comments['parent1_num_entities'] = 0


    pid_entities_list = []
    pid_num_entities_list = []
    for _, row in df_comments.iterrows():
        if row.top_level == 0:
            row2 = df_comments.loc[row.pid]
            parent1_entities = row2.entities
            parent1_num_entities = row2.num_entities
            pid_entities_list.append(parent1_entities)
            pid_num_entities_list.append(parent1_num_entities)
        else:
            parent1_entities = row.tid_entities
            parent1_num_entities = row.tid_num_entities
            pid_entities_list.append(parent1_entities)
            pid_num_entities_list.append(parent1_num_entities)
    df_comments['parent1_entities'] = pid_entities_list
    df_comments['parent1_num_entities'] = pid_num_entities_list

    return df_comments

def add_parent2_comment_entities(comments):
    pid2_entities_list = []
    pid2_num_entities_list = []
    pid2 = []
    second_level = []

    for _, row in comments.iterrows():
        if row.top_level == 0:
            row2 = comments.loc[row.pid]
            if row2.top_level == 0:
                row3 = comments.loc[row2.pid]
                parent2_entities = row3.entities
                parent2_num_entities = row3.num_entities
                pid2_entities_list.append(parent2_entities)
                pid2_num_entities_list.append(parent2_num_entities)
                pid2.append(row2.pid)
                second_level.append(0)
            else:
                parent2_entities = np.nan
                parent2_num_entities = 0
                pid2_entities_list.append(parent1_entities)
                pid2_num_entities_list.append(parent1_num_entities)
                pid2.append(row2.tid)
                second_level.append(1)
        else:
            parent2_entities = np.nan
            parent2_num_entities = 0
            pid2_entities_list.append(parent1_entities)
            pid2_num_entities_list.append(parent1_num_entities)
            pid2.append(np.nan)
            second_level.append(0)
    comments['parent2_entities'] = pid2_entities_list
    comments['parent2_num_entities'] = pid2_num_entities_list
    comments['pid2'] = pid2
    comments['second_level'] = second_level

    return comments


def resolve_references(doc):
    # token.idx : token.text
    token_mention_mapper = {}
    output_string = ""

    if len(doc.ents) == 0:
        output_string =  doc.text

    else:
        str_ents = [str(x) for x in doc.ents]
        str_ents = list(set(str_ents))
        clusters = [
                v for k, v in doc.spans.items() if k.startswith("coref_cluster")
            ]

        for cluster in clusters:
            str_cluster = [str(x) for x in cluster]

            for ent in str_ents:
                if ent in str_cluster:
                    for mention_span in list(cluster)[1:]:
                        # Set first_mention as value for the first token in mention_span in the token_mention_mapper
                        token_mention_mapper[mention_span[0].idx] = ent + mention_span[0].whitespace_

                        for token in mention_span[1:]:
                            # Set empty string for all the other tokens in mention_span
                            token_mention_mapper[token.idx] = ""
        # Iterate through every token in the Doc
        for token in doc:
            # Check if token exists in token_mention_mapper
            if token.idx in token_mention_mapper:
                output_string += token_mention_mapper[token.idx]
            # Else add original token text
            else:
                output_string += token.text + token.whitespace_


    return output_string.split('& ')[1]


def resolve_references_simple(text, nlp):
    doc = nlp(text)
    # token.idx : token.text
    token_mention_mapper = {}
    output_string = ""
    if len(doc.ents) == 0:
        output_string =  doc.text
    else:
        str_ents = [str(x) for x in doc.ents]
        str_ents = list(set(str_ents))
        clusters = [
                v for k, v in doc.spans.items() if k.startswith("coref_cluster")
            ]
        for cluster in clusters:
            str_cluster = [str(x) for x in cluster]
            for ent in str_ents:
                if ent in str_cluster:
                    for mention_span in list(cluster)[1:]:
                        # Set first_mention as value for the first token in mention_span in the token_mention_mapper
                        token_mention_mapper[mention_span[0].idx] = ent + mention_span[0].whitespace_
                        for token in mention_span[1:]:
                            # Set empty string for all the other tokens in mention_span
                            token_mention_mapper[token.idx] = ""
        # Iterate through every token in the Doc
        for token in doc:
            # Check if token exists in token_mention_mapper
            if token.idx in token_mention_mapper:
                output_string += token_mention_mapper[token.idx]
            # Else add original token text
            else:
                output_string += token.text + token.whitespace_
    return output_string

def get_df_list(text_df):
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    text_df.entities = text_df.entities.apply(str)
    text_df_id_indexed = text_df.set_index('id')

    df_list = []
    for k in list(entities.keys())[:-1]:
        kidx = []
        for i, row in text_df.iterrows():
            ents = row.entities.split(',')
            if entities[k]['full_name'] in ents:
                kidx.append(row.id)
        df = text_df_id_indexed.loc[kidx]
        df_list.append(df)

    return df_list

def get_token_dict(text_col):
    token_dict = {}
    for t in text_col:
        tokenized = word_tokenize(t)
        no_stop = [x for x in tokenized if x not in eng_stopwords]
        for t2 in no_stop:
            if t2 in token_dict.keys():
                token_dict[t2] += 1
            else:
                token_dict[t2] = 1

    sorted_token_dict = sorted(token_dict.items(), key=lambda x:x[1], reverse=True)
    sorted_token_dict = [x for x in sorted_token_dict if len(re.findall(r"[a-z]+", x[0])) > 0]

    return sorted_token_dict
```

```
# load nlp and add custom entities
#nlp = spacy.load('en_core_web_sm')
#ruler = nlp.add_pipe("entity_ruler", after="ner")

#for name in full_names:
#    pattern= [{"label": "PERSON", "pattern": name}]
#    ruler.add_patterns(pattern)
```

4) lakers_sentiment_analysis.py: functions to conduct and tweak sensitivity analysis

```
import pandas as pd
import numpy as np
import json
import random
import re
import spacy
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
from spacy.pipeline import EntityRuler
from spacy.language import Language
```

```python
from nltk.tokenize import word_tokenize



########################## Initiation Functions ##########################

def initiate_lexicon():
    """
    Initiate sentiment analyzer with custom emoji lexicon
    """

    with open("data/reddit_hoop_lexicon.json", "r") as f:
        hoop_lexicon = json.load(f)
        hoop_lexicon_dict = dict(hoop_lexicon)

    with open("data/emoji_lexicon.json", "r") as f:
        emoji_lexicon = json.load(f)
        emoji_lexicon_dict = dict(emoji_lexicon)



    sia = SentimentIntensityAnalyzer()
    sia.lexicon.update(hoop_lexicon_dict)
    sia.lexicon.update(emoji_lexicon_dict)

    return sia


@Language.component("filter_entities")
def filter_entities(doc):
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    add_ents = [entities[x]['full_name'] for x in entities.keys()]
    add_ents = add_ents[:-1]

    blacklist = {'PERSON', 'ORG', 'DATE', 'GPE',
                 'ORDINAL', 'CARDINAL', 'QUANTITY', 'LOC',
                 'TIME', 'PERCENT', 'PRODUCT', 'MONEY',
                 'FAC', 'NORP', 'EVENT', 'WORK_OF_ART'}

    doc.ents = [ent for ent in doc.ents if ent.label_ not in blacklist]

    return doc

@Language.factory('my_ruler')
def create_entity_ruler(nlp, name):

    nlp = spacy.load("en_core_web_sm")

    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    add_ents = [entities[x]['full_name'] for x in entities.keys()]
    add_ents = add_ents[:-1]

    ruler = EntityRuler(nlp, overwrite_ents=True)

    product_label_id = nlp.vocab.strings["LAKER"]

    # define patterns for the names
    patterns = []
    for i, ent in enumerate(add_ents):
        ent_dict = {}
        ent_dict['label'] = product_label_id
        ent_dict['pattern'] = [{"LOWER": ent}]
        patterns.append(ent_dict)

    ruler.add_patterns(patterns)

    return ruler

def initiate_nlp():
    nlp = spacy.load("en_core_web_sm")
    assert "transformer" not in nlp.pipe_names
    nlp_coref = spacy.load("en_coreference_web_trf")
    nlp.add_pipe("transformer", source=nlp_coref)
    nlp.add_pipe("coref", source=nlp_coref)
    nlp.add_pipe("span_resolver", source=nlp_coref)
    nlp.add_pipe("span_cleaner", source=nlp_coref)
    nlp.add_pipe('filter_entities', after='ner')
    nlp.vocab.strings.add('LAKER')
    nlp.add_pipe('my_ruler', before='ner')

    return nlp

### initialize ###
#sia = initiate_lexicon()
nlp = spacy.load("en_core_web_sm")

######################### coreference cleaning #########################

def resolve_references(doc):
    # token.idx : token.text
    token_mention_mapper = {}
    output_string = ""

    if len(doc.ents) == 0:
        output_string =  doc.text

    else:
        str_ents = [str(x) for x in doc.ents]
        str_ents = list(set(str_ents))
        clusters = [
                v for k, v in doc.spans.items() if k.startswith("coref_cluster")
            ]

        for cluster in clusters:
            str_cluster = [str(x) for x in cluster]
```

```python
            for ent in str_ents:
                if ent in str_cluster:
                    for mention_span in list(cluster)[1:]:
                        # Set first_mention as value for the first token in mention_span in the token_mention_mapper
                        token_mention_mapper[mention_span[0].idx] = ent + mention_span[0].whitespace_

                        for token in mention_span[1:]:
                            # Set empty string for all the other tokens in mention_span
                            token_mention_mapper[token.idx] = ""

         # Iterate through every token in the Doc
        for token in doc:
            # Check if token exists in token_mention_mapper
            if token.idx in token_mention_mapper:
                output_string += token_mention_mapper[token.idx]
            # Else add original token text
            else:
                output_string += token.text + token.whitespace_

    return output_string.split('** ')[1]

def get_unique_refs(entities, entity_key, df_posts, df_comments):
    z = set(df_posts['unique_ref'])
    z = [x.split("/") for x in z]
    z = [x for y in z for x in y]
    z_posts = list(set(z))



    z = set(df_comments['unique_ref'])
    z = [x.split("/") for x in z]
    z = [x for y in z for x in y]
    z_comments = list(set(z))



    names = z_posts + z_comments
    names = list(set(names))
    entities[entity_key]['names'] = names

    return entities


######################### basic sentiment #########################
def format_output(output_dict, cutoff = 0.05):
    if (cutoff < 0.0) | (cutoff > 1.0):
        raise ValueError("Cutoff must be between 0.0 and 1.0")
    polarity = "neutral"

    if(output_dict['compound']>= cutoff):
        polarity = "positive"

    elif(output_dict['compound']<= -cutoff):
        polarity = "negative"

    return polarity

def predict_sentiment(text, sia, cutoff = 0.05):
    output_dict =  sia.polarity_scores(text)

    return format_output(output_dict, cutoff=cutoff)

def predict_sentiment_percent(text, sia):
    output_dict =  sia.polarity_scores(text)

    return output_dict

def get_overall_sentiment(sentiments):
    total = len(sentiments)
    pos = len(sentiments[sentiments == "positive"])
    neg = len(sentiments[sentiments == "negative"])
    prop_pos = pos / total
    prop_neg = neg / total

    return prop_pos, prop_neg
```

```
def print_emoji_sent():
    with open('data/emoji.json', 'r') as f:
        emoji_json = json.load(f)
    emoji_dict = dict(emoji_json)

    ### Update Emoji Sentiments ###
    sorted_emoji_dict = sorted(emoji_dict.items(), key=lambda x:x[1], reverse=True)
    ### check emoji sentiment and adjust as needed ###
    for emoji_tuple in sorted_emoji_dict:
        sent = sia.polarity_scores(str(emoji_tuple[0]))
        print(emoji_tuple[0] + ": " + str(sent['compound']))

######################## nickname sentiment ########################
def get_sentiment_nm_tags(df, ent_key):

    # load entities
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    if df.player_ref[0] != entities[ent_key]['full_name']:
        raise ValueError("Dataframe and ent_key do not match!")

    negative_nicknames = entities[ent_key]['bad_names']
    positive_nicknames = entities[ent_key]['good_names']

    # update sent for pos/neg nicknames
    name_sent = []
    for i, row in df.iterrows():
        sent = 0
        for name in negative_nicknames:
            if name in row.unique_ref.split("/"):
                sent -= 1
        for name in positive_nicknames:
            if name in row.unique_ref.split("/"):
                sent += 1
        name_sent.append(sent)

    df['name_sent'] = name_sent

    return df



def get_sentiment_with_nicknames(df, ent_key, cutoff=0.05):

    # load entities
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    if df.player_ref[0] != entities[ent_key]['full_name']:
        raise ValueError("Dataframe and ent_key do not match!")

    # get sent w/o nicknames
    df['sentiment'] = df['body'].apply(predict_sentiment, cutoff=cutoff)

    negative_nicknames = entities[ent_key]['bad_names']
    positive_nicknames = entities[ent_key]['good_names']

    # update sent for pos/neg nicknames
    for i, row in df.iterrows():
        sent = 0
        for name in negative_nicknames:
            if name in row.unique_ref.split("/"):
                sent -= 1
        for name in positive_nicknames:
            if name in row.unique_ref.split("/"):
                sent += 1
        if sent < 0:
            df.loc[i, 'sentiment'] = 'negative'
        elif sent > 0:
            df.loc[i, 'sentiment'] = 'positive'
        else:
            continue
    return df

def sub_nicknames(df, ent_key):
    # load entities
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    if df.player_ref[0] != entities[ent_key]['full_name']:
        raise ValueError("Dataframe and ent_key do not match!")


    # substitute nicknames for name
    ent_name = df.player_ref[0]
    print(ent_name)

    body_list = []
    for i, row in df.iterrows():
        body = row.body
        names = row.unique_ref.split("/")
        sorted_names = sorted(names, key=len, reverse=True)
        for name in sorted_names:
            body = re.sub(r"\b{}\b".format(name), ent_name, body)
        body_list.append(body)

    body_list_pos = []
    for i, row in df.iterrows():
        body_pos = row.body_pos
        names = row.unique_ref.split("/")
        sorted_names = sorted(names, key=len, reverse=True)
        for name in sorted_names:
            body_pos = re.sub(r"\b{}\b".format(name), ent_name, body_pos)
        body_list_pos.append(body_pos)
    # any nickname w 2 spaces goes first!

    df["body_pr"] = body_list
    df['body_pos_pr'] = body_list_pos

    return df


def sub_all_nicknames(df):
    """
    Takes full post or comment df and substitutes all player entity refs to player ent name
    """
    # load entities
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    df.entities = df.entities.apply(str)
    # substitute nicknames for name

    body_list = []
    for i, row in df.iterrows():
        body = row.body
        names = row.entities.split(", ")
        for k in entities.keys():
            ent_name = entities[k]['full_name']
            if ent_name in names:
                nm = entities[k]['names']
                nm_sorted = sorted(nm, key=len, reverse=True)

                for nm in nm_sorted:
                    body = re.sub(r"\b{}\b".format(nm), ent_name, body)
        body_list.append(body)



    # any nickname w 2 spaces goes first!
    df["body_pr"] = body_list

    return df

######################### sentiment methods #########################
def sentiment_basic(player_df, sia, player_df_sent_col='resolved_final', nicknames=True):
    # filter out un-needed cols
    columns = ['id', 'datetime', 'player_ref', 'name_sent', player_df_sent_col]
    df = player_df[columns]
    df['unique_id'] = df.id + "_" + df.player_ref

    # get sentiment
    sents = []
    for i, row in df.iterrows():
        sent = predict_sentiment(row[player_df_sent_col], sia=sia)
        sents.append(sent)

    df['sentiment'] = sents

    if nicknames:
        new_sent = []
        for i, row in df.iterrows():
            if row.name_sent < 0:
                new_sent.append('negative')
            elif row.name_sent > 0:
                new_sent.append('positive')
            else:
                new_sent.append(row.sentiment)
        df['sentiment'] = new_sent

    return df

def sentence_tokenized_sentiment_basic(text, player_ref, nlp, sia):
    sentences = re.split(r"[!.?]", text)
    sentences_player = []
    for sentence in sentences:
```

```python
        if player_ref in word_tokenize(str(sentence)):
            sentences_player.append(sentence)
    new_text = "".join(sentences_player)
    sentiment = predict_sentiment(new_text, sia=sia)

    return sentiment



def sentence_tokenized_sentiment_basic_df(player_df, nlp, sia, nicknames=True):
    player_ref = player_df.iloc[0].player_ref

    # filter out un-needed cols
    columns = ['id', 'datetime', 'player_ref', 'name_sent', 'resolved_final']
    df = player_df[columns]
    df['unique_id'] = df.id + "_" + df.player_ref

    sentiments = []
    for i, row in df.iterrows():
        sentiment = sentence_tokenized_sentiment_basic(
            text=row.resolved_final,
            player_ref=player_ref,
            nlp=nlp,
            sia=sia
        )
        sentiments.append(sentiment)

    df['sentiment'] = sentiments

    if nicknames:
        new_sent = []
        for i, row in df.iterrows():
            if row.name_sent < 0:
                new_sent.append('negative')
            elif row.name_sent > 0:
                new_sent.append('positive')
            else:
                new_sent.append(row.sentiment)
        df['sentiment'] = new_sent

    return df

def get_all_sentiment(posts, comments, sia):
    columns = ['id', 'datetime', 'body_pr']
    posts = posts[columns]
    comments = comments[columns]
    posts['pc'] = "p"
    comments['pc'] = "c"

    # combine posts and comments
    combined = pd.concat([posts, comments])
    combined = combined.sort_values('datetime').reset_index().drop(columns=['index'])

    # get overall sentiment for each post
    sentiments = []
    for i, row in combined.iterrows():
        sentiment = predict_sentiment(row.body_pr, sia=sia)
        sentiments.append(sentiment)
    combined['sentiment'] = sentiments

    return combined


######################### format df list into full sentiment df #########################
def save_df_list_sent(df_list, file_pattern):
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    path = "data/sent_scores/"

    for i, k in enumerate(list(entities.keys())[:-1]):
        name = "{}".format(k) + "_sent_" + file_pattern
        file = path + name + ".csv"
        df_list[i].to_csv(file)
```

```
def concat_player_sents(ent_key, file_pattern):
    path = "data/sent_scores/"

    # load data
    post_file = path + ent_key + "_sent_posts_" + file_pattern + ".csv"
    post_sent = pd.read_csv(post_file, index_col=0, parse_dates=['datetime'])
    comment_file = path + ent_key + "_sent_comments_" + file_pattern + ".csv"
    comment_sent = pd.read_csv(comment_file, index_col=0, parse_dates=['datetime'])

    # create p/c identifier and concat
    post_sent['pc'] = "p"
    comment_sent['pc'] = "c"

    combined = pd.concat([post_sent, comment_sent])
    combined = combined.sort_values('datetime').reset_index().drop(columns=['index'])

    return combined

def combine_all_sents(file_pattern):
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    file = "data/sent_scores/{}_sent_combined_" + file_pattern
    df_list_combined = [pd.read_csv(file.format(k), index_col=0, parse_dates=['datetime']) for k in list(entities.keys())[:-1]]

    combined_combined = pd.concat(df_list_combined)
    combined_combined = combined_combined.sort_values('datetime').reset_index().drop(columns=['index'])

    return combined_combined

def save_df_list_sent(df_list, file_pattern):
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)
    path = "data/sent_scores/"

    index = 0
    for k in list(entities.keys())[:-1]:
        name = "{}".format(k) +"_sent_" + file_pattern
        file = path+name
        df_list[index].to_csv(file)
        index += 1

def get_entity_prop(df_player):
    df = df_player[['id', 'player_ref', 'ents_exp']]
    player_ref = df.player_ref[0]
    df['unique_id'] = df.id + "_" + df.player_ref

    ent_props = []
    for i, row in df.iterrows():
        text = row.ents_exp
        names = re.findall(r"'(\b\w+\b)':", text)
        counts = re.findall(r":\s(\d{1,2})", text)
        counts = [int(x) for x in counts]
        idx = names.index(player_ref)
        count = counts[idx]
        total = np.sum(counts)
        prop = count / total
        ent_props.append(prop)

    df_ent_prop = pd.Series(index=df.unique_id, data=ent_props)

    return df_ent_prop

def get_sentence_prop(df_player):
    df = df_player[['id', 'player_ref', 'ents_exp', 'resolved_final']]
    player_ref = df.player_ref[0]
    df['unique_id'] = df.id + "_" + df.player_ref
    df.resolved_final = df.resolved_final.apply(str)
    sentence_props = []
    for i, row in df.iterrows():
        sentences = re.split(r"[!.?]", row.resolved_final)
        total = len(sentences)
        count = 0
        for sentence in sentences:
            if player_ref in word_tokenize(str(sentence)):
                count += 1
        prop = count / total
        sentence_props.append(prop)

    df_sentence_prop = pd.Series(data=sentence_props, index=df.unique_id)

    return df_sentence_prop


######################## sampling and testing ########################

def get_single_ent_post_sample():

    posts = pd.read_csv("data/posts_clean_extended_final.csv", index_col=0, parse_dates=['datetime'])
    posts.body = posts.body.apply(str)

    random.seed(18)
    single_ent_posts = posts[posts['num_entities'] == 1]
    single_ent_index = single_ent_posts.index
    sample_idx = random.sample(set(single_ent_index), 200)

    # create sample_posts
    sample_posts = posts.loc[sample].reset_index()

    sent_dict = {
    0: 'neutral',
    1: 'negative',
    2: 'positive'
    }
    # manually labeled sentiments
    sample_sentiments = [
        0, 0, 0, 0, 2, 2, 2, 0, 0, 0,
        0, 0, 1, 0, 1, 0, 0, 2, 2, 0,
        2, 2, 1, 0, 0, 0, 1, 0, 1, 0,
        0, 2, 1, 0, 2, 2, 0, 2, 0, 0,
        1, 0, 2, 1, 0, 2, 0, 1, 0, 1,
        2, 0, 2, 0, 0, 0, 2, 2, 2, 0,
        0, 2, 1, 1, 2, 0, 0, 0, 2, 0,
        0, 1, 0, 1, 1, 1, 2, 1, 1, 0,
        2, 0, 1, 0, 0, 2, 2, 2, 2, 1,
        0, 2, 2, 1, 0, 0, 2, 0, 2, 0,
        0, 0, 2, 2, 2, 0, 2, 0, 1, 1,
        0, 1, 1, 0, 0, 2, 0, 2, 0, 0,
        0, 2, 1, 2, 1, 2, 0, 0, 2, 2,
        0, 1, 2, 0, 1, 0, 0, 2, 0, 2,
        2, 2, 0, 0, 2, 2, 2, 2, 1, 0,
        2, 0, 0, 2, 2, 0, 0, 2, 1, 0,
        0, 1, 2, 2, 2, 0, 2, 2, 2, 1,
        0, 1, 1, 1, 1, 0, 0, 0, 2, 0,
        0, 1, 2, 1, 0, 0, 0, 1, 0, 0,
        2, 2, 0, 2, 1, 2, 1, 2, 2, 1
    ]

    sample_sentiment_words = [sent_dict[x] for x in sample_sentiments]
    sample_posts['sentiment_labels'] = sample_sentiment_words

    return sample_posts


def get_single_ent_comment_sample():
    comments = pd.read_csv("data/comments_clean_extended_final.csv", index_col=0, parse_dates=['datetime'])
    comments.body = comments.body.apply(str)



    random.seed(18)
    single_ent_tl_comments = comments[(comments['num_entities'] == 1) & (comments['top_level'] == 1) &
(comments['tid_num_entities'] == 0)]
```

```
single_ent_index = list(single_ent_tl_comments.index)
sample_idx = random.sample(set(single_ent_index), 300)

# create sample commments
sample_comments = comments.loc[sample_idx].reset_index()

sent_dict = {
    0: 'neutral',
    1: 'negative',
    2: 'positive'
}
# manually labeled sentiments
sample_sentiments = [
    2, 0, 0, 2, 1, 0, 2, 2, 2, 2,
    1, 1, 0, 0, 0, 1, 1, 2, 2, 0,
    2, 1, 2, 1, 0, 2, 2, 0, 0, 0,
    1, 2, 2, 1, 2, 1, 1, 0, 0, 0,
    0, 1, 1, 1, 0, 1, 1, 0, 1, 1,
    1, 0, 0, 0, 0, 0, 0, 2, 1, 2,
    1, 1, 0, 2, 0, 2, 0, 2, 0, 2,
    0, 1, 0, 0, 0, 2, 0, 2, 0, 2,
    0, 1, 0, 2, 0, 0, 0, 1, 1, 2,
    2, 1, 0, 0, 0, 1, 0, 2, 2, 0,
    1, 1, 2, 1, 1, 1, 2, 2, 2, 0,
    0, 1, 2, 1, 0, 0, 1, 2, 1, 1, # 120
    0, 0, 1, 2, 0, 0, 1, 2, 2, 1, # 130
    1, 1, 2, 0, 0, 0, 1, 2, 1, 0,
    0, 1, 0, 0, 1, 0, 1, 0, 1, 0, # 150
    0, 1, 0, 2, 2, 1, 2, 0, 0, 0,
    0, 1, 1, 0, 1, 0, 1, 0, 2, 0,
    1, 2, 1, 1, 1, 1, 1, 1, 1, 0, #180
    0, 0, 0, 2, 1, 1, 0, 1, 1, 2, #190
    1, 1, 2, 1, 2, 2, 1, 1, 2, 0, #200
    0, 1, 2, 0, 1, 2, 1, 2, 0, 0, #210
    0, 2, 0, 0, 1, 0, 0, 0, 1, 0, #220
    0, 1, 0, 0, 2, 0, 0, 2, 2, 2, #230
    1, 0, 2, 1, 1, 2, 2, 2, 0, 0, #240
```

```
        2, 1, 0, 1, 0, 2, 2, 0, 1, 0,
        0, 0, 0, 0, 0, 2, 2, 2, 0, 1, #260
        2, 2, 1, 1, 1, 1, 2, 1, 1, 2,
        1, 0, 0, 1, 0, 0, 2, 1, 1, 1,
        1, 0, 1, 1, 0, 0, 0, 2, 1, 0,
        1, 1, 0, 2, 0, 0, 0, 1, 0, 0
    ]

    sample_sentiment_words = [sent_dict[x] for x in sample_sentiments]
    sample_comments['sentiment_labels'] = sample_sentiment_words

    return sample_comments


def final_sample():
    all_sents = [pd.read_csv(path+n, index_col=0, parse_dates=['datetime']) for n in names]
    random.seed(18)
    df = all_sents[0]
    df = df.set_index('unique_id')
    sampleidx = random.sample(list(df.index), 500)

    # create sample_posts
    sample = df.loc[sampleidx].reset_index()

    sample_sentiments=[
        2, 2, 0, 2, 0, 0, 1, 2, 0, 2, #10
        1, 0, 1, 0, 1, 1, 0, 2, 1, 2, #20
        0, 0, 1, 1, 0, 0, 0, 0, 1, 0, #30
        0, 0, 0, 2, 0, 2, 0, 1, 2, 1, #40
        1, 2, 1, 1, 2, 2, 0, 0, 1, 0, #50
        2, 1, 1, 2, 2, 0, 2, 0, 1, 2, #60
        0, 0, 0, 1, 0, 0, 0, 0, 0, 2, #70
        1, 2, 2, 0, 2, 0, 2, 0, 0, 1, #80
        0, 2, 0, 1, 1, 0, 1, 0, 1, 0, #90
        1, 0, 2, 2, 2, 1, 0, 1, 0, 2, #100
        0, 2, 0, 1, 0, 0, 0, 0, 0, 0, #110
        2, 2, 0, 1, 0, 1, 0, 2, 0, 0, #120
```

```
1, 0, 1, 0, 1, 1, 2, 0, 1, 2, #130
0, 0, 2, 1, 0, 1, 2, 2, 1, 0, #140
1, 2, 2, 0, 0, 0, 1, 0, 1, 0, #150
2, 2, 0, 2, 0, 0, 1, 0, 0, 1, #160
2, 2, 0, 1, 2, 2, 2, 1, 0, 0, #170
0, 2, 1, 2, 0, 1, 0, 1, 0, 2, #180
2, 0, 1, 1, 0, 0, 0, 0, 0, 0, #190
0, 1, 1, 0, 2, 2, 0, 0, 1, 1, #200
0, 2, 0, 0, 0, 0, 1, 1, 0, 2, #210
0, 0, 0, 0, 0, 0, 2, 0, 0, 1, #220
2, 0, 2, 0, 1, 2, 2, 0, 0, 0, #230
1, 2, 0, 0, 0, 2, 0, 2, 2, 2, #240
2, 1, 0, 0, 2, 1, 2, 1, 1, 1, #250
1, 2, 2, 0, 1, 0, 0, 1, 0, 1, #260
0, 0, 2, 2, 0, 1, 0, 0, 2, 1, #270
2, 1, 1, 2, 1, 2, 1, 2, 0, 0, #280
0, 0, 1, 0, 1, 0, 0, 2, 2, 0, #290
1, 0, 0, 2, 1, 2, 1, 1, 1, 0, #300
2, 2, 0, 2, 2, 2, 0, 2, 2, 2, #310
0, 0, 1, 1, 0, 0, 2, 1, 0, 2, #320
0, 0, 2, 0, 1, 0, 0, 0, 1, 1, #330
2, 2, 0, 0, 1, 1, 0, 1, 2, 0, #340
0, 1, 0, 0, 1, 1, 2, 2, 2, 2, #350
0, 1, 1, 0, 0, 0, 2, 1, 1, 2, #360
0, 2, 2, 0, 1, 1, 0, 1, 1, 1, #370
1, 0, 2, 1, 1, 1, 0, 0, 1, 2, #380
2, 1, 0, 1, 1, 0, 0, 0, 2, 2, #390
1, 1, 1, 2, 2, 1, 2, 2, 2, 2, #400
2, 0, 2, 0, 2, 0, 2, 0, 1, 0, #410
0, 0, 1, 2, 2, 0, 0, 1, 0, 1, #420
2, 1, 1, 1, 2, 1, 0, 2, 1, 0, #430
1, 2, 2, 2, 1, 1, 1, 0, 0, 2, #440
0, 0, 0, 1, 1, 2, 2, 2, 0, 0, #450
0, 0, 0, 2, 0, 2, 2, 0, 0, 0, #460
2, 1, 1, 2, 1, 1, 2, 1, 0, 2, #470
2, 2, 0, 2, 1, 0, 2, 1, 0, 2, #480
0, 2, 2, 1, 2, 0, 0, 0, 2, 0, #490
```

```
        0, 0, 1, 1, 2, 1, 0, 1, 2, 2, #500

    ]
    sent_dict = {
            0: 'neutral',
            1: 'negative',
            2: 'positive'
        }
    sample_labs = [sent_dict[x] for x in sample_sentiments]
    sample = sample.drop(columns=['sentiment'])
    sample['sentiment_labels'] = sample_labs

    return sample

def get_post_sent_score_dict(df_list_posts, file_name):
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
        entities = dict(entities)

    index = 0
    for k in entities.keys():
        print(k)
        df_list_posts[index] = get_sentiment_with_nicknames(df_list_posts[index], k)
        print(df_list_posts[index].loc[0].player_ref)
        index += 1

    df_list_posts_scored = {}
    for df in df_list_posts:
        prop_pos, prop_neg = get_overall_sentiment(df['sentiment'])
        score = prop_pos - prop_neg
        df_list_posts_scored[df.player_ref[0]] = score

    # save dict
    with open("data/sent_dicts/sa_posts_{}.json".format(file_name), "w") as f:
        json_dict = json.dumps(df_list_posts_scored)
        f.write(json_dict)
    df_list_posts_w_sent = df_list_posts

    return df_list_posts_w_sent, df_list_posts_scored


def get_comment_sent_score_dict(df_list_comments, file_name): # this doesnt use names?
    with open('data/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
        entities = dict(entities)

    index = 0
    for k in entities.keys():
        print(k)
        df_list_comments[index] = get_sentiment_with_nicknames(df_list_comments[index], k)
        print(df_list_comments[index].loc[0].player_ref)
        index += 1



    df_list_comment_scored = {}
    for df in df_list_comments:
        prop_pos, prop_neg = get_overall_sentiment(df['sentiment'])
        score = prop_pos - prop_neg
        df_list_comment_scored[df.player_ref[0]] = score

    # save dict
    with open("data/sent_dicts/sa_comments_nicknames{}.json".format(file_name), "w") as f:
        json_dict = json.dumps(df_list_comment_scored)
        f.write(json_dict)

    df_list_comments_w_sent = df_list_comments

    return df_list_comments_w_sent, df_list_comment_scored

def sent_rmse(true, preds):
    sent_map = {
        'negative':-0.5,
        'neutral':0.0,
        'positive':0.5
    }

    index = range(true.shape[0])
    mses = []
    for i in index:
        mse = sent_map[true[i]] - sent_map[preds[i]]
        mses.append(mse ** 2)

    rmse = np.mean(mse)

    return mses

def compare_sentiments(sample, cols):
    reports = []
    for col in cols:
        report = classification_report(sample_ind.sentiment_labels,
                                       sample_ind[col],
                                       digits=3,
                                       output_dict=True)
        reports.append(report)
    rmses = []
    for col in cols:
        rmse = np.mean(lsa.sent_rmse(sample_ind.sentiment_labels,
                             sample_ind[col]))
        rmses.append(rmse)

    res = {}
    for i, col in enumerate(cols):
        res[col] = reports[i]
        res[col]['rmse'] = rmses[i]

    return res



#comments2.body = comments2.body.apply(tc.sub_emojis, subbed_emoji="🤦", sub_emoji="🤦")
```

```
#comments2.body = comments2.body.apply(tc.sub_emojis, subbed_emoji="♡", sub_emoji="💚")
#comments2.body = comments2.body.apply(tc.sub_emojis, subbed_emoji="❄", sub_emoji="⍰")
#comments2.body = comments2.body.apply(tc.sub_emojis, subbed_emoji="🗑", sub_emoji="🚮")
```

5) Sentiment Statistics

```
import pandas as pd
import numpy as np
import datetime as dt
import emoji
import re
import nltk
import json
from nltk import word_tokenize
nltk.download('stopwords')
from nltk.corpus import stopwords
eng_stopwords = stopwords.words('english')
### load data ###

def get_sentiment(sent_df, post_multiplier=2.5):
    """
    This function simply takes in sent df and returns average sentiment
    Parameters:
        post_multiplier: how much more should we weight post vs comment? Default = 2.5
```

```
"""
# get post sentiment score
try:
    post_sent = sent_df[sent_df['pc'] == 'p']
    post_length = post_sent.shape[0]
    try:
        post_prop_pos = post_sent.sentiment.value_counts().loc['positive'] / post_length
    except:
        post_prop_pos = 0
    try:
        post_prop_neg = post_sent.sentiment.value_counts().loc['negative'] / post_length
    except:
        post_prop_neg = 0
    post_score = post_prop_pos - post_prop_neg

except:
    post_score = 0

try:
    comment_sent =  sent_df[sent_df['pc'] == 'c']
    comment_length = comment_sent.shape[0]
    try:
        comment_prop_pos = comment_sent.sentiment.value_counts().loc['positive'] / comment_length
    except:
        comment_prop_pos = 0
    try:
        comment_prop_neg = comment_sent.sentiment.value_counts().loc['negative'] / comment_length
    except:
        comment_prop_neg = 0
    comment_score = comment_prop_pos - comment_prop_neg

except:
    comment_score = 0

if (post_score == 0) & (comment_score == 0):
    weighted = 0
else:
```

```python
        # get weighted score
        num = (post_score * post_length * post_multiplier) + (comment_score * comment_length)
        denom = (post_length * post_multiplier) + comment_length
        weighted = num / denom

    return weighted

def get_sent_other(player_ref, date):
    sent = pd.read_csv("assets/all_sents/stok.csv", index_col=0, parse_dates=['datetime'])
    sent = sent[sent['player_ref'] == player_ref]
    sent = sent[sent['datetime'] <= date]

    weighted = get_sentiment(sent)

    return weighted

def get_trending_sentiment(player_ref, date, n=20, n_comp=None):
    if type(n) != int:
        raise TypeError("Invalid Value Type. Must be int")

    sent = pd.read_csv("assets/all_sents/stok.csv", index_col=0, parse_dates=['datetime'])
    sent = sent[sent['player_ref'] == player_ref]
    sent = sent[sent['datetime'] <= date]

    date_dt = dt.datetime.strptime(date, "%Y-%m-%d")
    date_dt_first = date_dt - dt.timedelta(days=n)
    date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
    trend = sent[sent['datetime'] > date_first]

    sent_score_last_n = get_sentiment(trend)

    comp = sent[sent['datetime'] <= date_first]
    if n_comp != None:
        date_dt2 = dt.datetime.strptime(date_first, "%Y-%m-%d")
        date_dt_first2 = date_dt2 - dt.timedelta(days=n_comp)
        date_first2 = dt.datetime.strftime(date_dt_first2, "%Y-%m-%d")
        comp = comp[comp['datetime'] > date_first2]
    sent_score_comp = get_sentiment(comp)
    diff = (sent_score_last_n - sent_score_comp) / sent_score_comp

    return sent_score_last_n, sent_score_comp, diff

def get_trending_sentiment2(sent, date, n=20, n_comp=None):
    if type(n) != int:
        raise TypeError("Invalid Value Type. Must be int")

    date_dt = dt.datetime.strptime(date, "%Y-%m-%d")
    date_dt_first = date_dt - dt.timedelta(days=n)
    date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
    trend = sent[sent['datetime'] > date_first]

    sent_score_last_n = get_sentiment(trend)

    comp = sent[sent['datetime'] <= date_first]
    if n_comp != None:
        date_dt2 = dt.datetime.strptime(date_first, "%Y-%m-%d")
        date_dt_first2 = date_dt2 - dt.timedelta(days=n_comp)
        date_first2 = dt.datetime.strftime(date_dt_first2, "%Y-%m-%d")
        comp = comp[comp['datetime'] > date_first2]
    sent_score_comp = get_sentiment(comp)
    diff = (sent_score_last_n - sent_score_comp) / sent_score_comp

    return sent_score_last_n, sent_score_comp, diff



def get_rolling_sentiment_score_df(player_ref, date, n=20):
    sent = pd.read_csv("assets/all_sents/stok.csv", index_col=0, parse_dates=['datetime'])
    sent = sent[sent['player_ref'] == player_ref]
    sent = sent[sent['datetime'] <= date]

    cols = ['weighted']
    idx = sorted(list(set(sent['datetime'])))
    dfs = pd.DataFrame(columns=cols, index=idx)
    weighted=[]
    for i in idx:
        start = i - dt.timedelta(days=20)
        s = sent[(sent['datetime'] > start) & (sent['datetime'] <= i) ]
        w = get_sentiment(s)
        weighted.append(w)
    dfs['weighted'] = weighted

    return dfs


### functions related to word counts ###
def get_top_players(player_ref):
    df = pd.read_csv("assets/rm_hm_df.csv", index_col=0)
    col_name = player_ref + "_refs"
    df = df[col_name]
    t = df.sort_values(ascending=False)

    return t


def get_emoji_dict(player_ref, kind="all"):
    if kind not in ['all', 'positive', 'negative']:
        raise ValueError("Invalid kind, must be one of 'all', 'positive', or 'negative' ")
    # allow for use in TeamDate class as well as PlayerDate
    if player_ref=='team':
        df = pd.read_csv("assets/all_sents/total_sentiment.csv", index_col=0, parse_dates=['datetime'])
        df['body'] = df.body_pr
        df['body'] = df.body.apply(str)

    else:
        psent = pd.read_csv("assets/all_sents/base.csv", index_col=0, parse_dates=['datetime'])
        df = psent[psent['player_ref'] == player_ref]

    if kind == 'positive':
        df = df[df['sentiment'] == 'positive']
    elif kind == 'negative':
        df = df[df['sentiment'] == 'negative']
    else:
        df = df

    emoji_dict = {}
    for t in df.body:
        matches = emoji.emoji_list(t)
        text_emojis = []
        for i, _ in enumerate(matches):
            text_emojis.append(matches[i]['emoji'])
        unique_text_emojis = list(set(text_emojis))
        for emj in unique_text_emojis:
            if emj not in emoji_dict.keys():
                emoji_dict[emj] = 1
            else:
                emoji_dict[emj] += 1
    x = emoji_dict
    sorted_emoji_dict = sorted(x.items(), key=lambda x:x[1], reverse=True)

    return sorted_emoji_dict


def get_token_dict(player_ref, kind="all"):
    if kind not in ['all', 'positive', 'negative']:
        raise ValueError("Invalid kind, must be one of 'all', 'positive', or 'negative' ")

    if player_ref=='team':
        df = pd.read_csv("assets/all_sents/total_sentiment.csv", index_col=0, parse_dates=['datetime'])
        df['body'] = df.body_pr
        df['body'] = df.body.apply(str)
    else:
        psent = pd.read_csv("assets/all_sents/base.csv", index_col=0, parse_dates=['datetime'])
        df = psent[psent['player_ref'] == player_ref]

    if kind == 'positive':
        df = df[df['sentiment'] == 'positive']
    elif kind == 'negative':
        df = df[df['sentiment'] == 'negative']
    else:
        df = df


    token_dict = {}
    for t in df.body:
        tokenized = word_tokenize(t)
        no_stop = [x for x in tokenized if x not in eng_stopwords]
        for t2 in no_stop:
            if t2 in token_dict.keys():
                token_dict[t2] += 1
            else:
                token_dict[t2] = 1

    sorted_token_dict = sorted(token_dict.items(), key=lambda x:x[1], reverse=True)
    sorted_token_dict = [x for x in sorted_token_dict if len(re.findall(r"[a-z]+", x[0])) > 0]

    return sorted_token_dict



def get_ytd_player_sent_ranks(date='2023-04-10'):
    with open('assets/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    sent = pd.read_csv("assets/all_sents/stok.csv", index_col=0, parse_dates=['datetime'])
    sent = sent[sent['datetime'] <= date]

    scores = {}
    for k in list(entities.keys())[3:21]:
        fn = entities[k]['full_name']
        df = sent[sent['player_ref'] == fn]
        name =  entities[k]['init_name']
        score = get_sentiment(df)
        scores[name] = score

    scores_sorted = sorted(scores.items(), key=lambda x:x[1], reverse=True)

    names = [scores_sorted[i][0] for i, z in enumerate(scores_sorted)]
    scores = [scores_sorted[i][1] for i, z in enumerate(scores_sorted)]

    return names, scores
```

6) Lakers team/player classes, combining attributes, stats, sentiment stats etc.

```
import pandas as pd
import numpy as np
import json
import re
import datetime as dt
import plotly.graph_objects as go
import pandas as pd
import numpy as np
import datetime as dt
import random
import json
import re
from plotly.subplots import make_subplots
from wordcloud import WordCloud
from plotly.io import templates

# my imports
import sentiment_stats as ss

### load data ###
path = "assets/"
# player dict with names and ids
f = open(path + 'players.json')
player_dict = json.load(f)
# transaction dict with transaction info for players
f = open(path + 'transactions.json')
transactions_dict = json.load(f)
# teams dict contains lakers and teams laker players played for in 2022-23 w id and all game ids
f = open(path + 'teams.json')
team_dict = json.load(f)

def load_json_file(file, path=path):
    f = open(path + file)
    return json.load(f)


def find_nearest_date(df, column, date):
    """
    Given a df with a string date column, returns string date of nearest date that does not go past that date.
    Note: date must be inh '%Y-%m-%d' format
    Parameters:
        df: dataframe to reference
        column: df column to reference
        date: string format date
    """

    dt_date = dt.datetime.strptime(date, "%Y-%m-%d") # define date in dt format

    string_dates = list(df[column])
    dates = list(pd.to_datetime(df[column]))
    diffs = [dt_date - x for x in dates]
    idx = diffs.index(min([d for d in diffs if d.days >=0]))
    date = string_dates[idx]

    return date

def gmas(stat, stat_df):
    """
    Advanced stats are calculated using number of possessions so simple mean will not \
    calculate proper value. This function takes possessions into account to return the \
    correct average advanced stat
    """
    product = stat_df[stat] * stat_df['POSS']
    mean_stat = product.sum() / stat_df['POSS'].sum()
    return mean_stat


def get_cum_stats_any(df, x=None, n=None, laker_only=False, team=False):
        """
        Returns player's cumulative stats throughout season. /
        Can specify last x games/or n days and also whether to include all stats or laker
only
        Parameters:
            x_games: if set, gets cum stats last x games; default=None
            n_days: if set, gets cum stats last n days; default=None
        """
        if (x != None) & (n != None):
            raise ValueError("Can only specify either x games or n days, not both!")

        # define stats
        if laker_only:
            df = df[df['TEAM_ABBREVIATION'] == 'LAL']


        # drop games with 0 minutes played:
        df = df[~df['MIN'].isna()].reset_index().drop(columns=("index"))

        last_date = df['GAME_DATE'].iloc[-1]

        # filter last x games if x specified
        if x != None:
            if type(x) != int:
                raise TypeError("Invalid Value Type. Must be int")
            idx_first = df.shape[0] - x
            if idx_first < 0:
                idx_first = 0
            df = df.iloc[idx_first:]

        # filter last n days if n specified
        elif n != None:
            if type(n) != int:
                raise TypeError("Invalid Value Type. Must be int")
            date_dt = dt.datetime.strptime(last_date, "%Y-%m-%d")
            date_dt_first = date_dt - dt.timedelta(days=n)
            date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
            df = df[df['GAME_DATE'] > date_first]


        # drop columns that are not cumulative
        drop_columns = ['GAME_ID', 'TEAM_ID', 'TEAM_ABBREVIATION', 'TEAM_CITY',
                        'PLAYER_ID', 'PLAYER_NAME', 'FG_PCT', 'FG3_PCT', 'FT_PCT',
                        'GAME_DATE', 'MATCHUP', 'WL', 'TM_TOV_PCT']

        df = df.drop(columns=drop_columns)



        # filter our advanced columns that use special mean function
        adv_cols = list(df.columns[16:36])
        remove = ['AST_TOV', 'NET_RATING', 'E_NET_RATING', 'POSS']
        adv_cols = [x for x in adv_cols if x not in remove]


        # calculate mean traditional stats
        totals = df.sum(axis=0)
        totals['GAMES'] = df.shape[0]
        totals['FG_PCT'] = totals.FGM / totals.FGA
        totals['FG3_PCT'] = totals.FG3M / totals.FG3A
        totals['FT_PCT'] = totals.FTM / totals.FTA

        # calculate mean advanced stats
        for col in adv_cols:
            totals[col] = gmas(col, df)


        # calc remaining adv stats
        totals.NET_RATING = totals.OFF_RATING - totals.DEF_RATING
        totals.E_NET_RATING = totals.E_OFF_RATING - totals.E_DEF_RATING
        totals.AST_TOV = totals.AST / totals.TO


        # format data and return
        for i in totals.index:
            if "PCT" in i:
                totals.loc[i] = totals.loc[i] * 100
            elif i == "PIE":
                totals.loc[i] = totals.loc[i] * 100
        return totals.round(1)
```

```
def get_per_game_stats_any(df, x=None, n=None, laker_only=False, team=False):
    cum_stats = get_cum_stats_any(df=df, x=x, n=n, laker_only=laker_only)

    divide_col_index = list(cum_stats.index[:16])
    divide_col_index.append("POSS")
    divide_col_index.append("MIN")

    for idx in divide_col_index:
        name = idx + "_PG"
        cum_stats[name] = cum_stats[idx] / cum_stats.GAMES
        cum_stats.drop(index=idx, inplace=True)

    per_game = cum_stats

    return per_game.round(1)

def get_per_m_minutes_stats_any(df, m=36, x=None, n=None, laker_only=False,
team=False):
    cum_stats = get_cum_stats_any(df=df, x=x, n=n)

    divide_col_index = list(cum_stats.index[:16])
    divide_col_index.append("POSS")
    divide_col_index.append("MIN")
    total_min = cum_stats.MIN

    for idx in divide_col_index:
        name = idx + "_P{}M".format(m)
        cum_stats[name] = cum_stats[idx] / total_min * m
        cum_stats.drop(index=idx, inplace=True)

    per_minutes = cum_stats

    return per_minutes.round(1)

def get_per_p_possessions_stats_any(df, p=100, x=None, n=None, laker_only=False,
team=False):
    cum_stats = get_cum_stats_any(df=df, x=x, n=n)

    divide_col_index = list(cum_stats.index[:16])
    divide_col_index.append("POSS")
    divide_col_index.append("MIN")
    total_poss = cum_stats.POSS

    for idx in divide_col_index:
        name = idx + "_P{}P".format(p)
        cum_stats[name] = cum_stats[idx] / total_poss * p
        cum_stats.drop(index=idx, inplace=True)

    per_possessions = cum_stats

    return per_possessions.round(1)

def get_cum_stats_any_team(df, x=None, n=None):
    if (x != None) & (n != None):
            raise ValueError("Can only specify either x games or n days, not both!")
    # get last date
    last_date = df['GAME_DATE'].iloc[-1]

    # filter last x games if x specified
    if x != None:
        if type(x) != int:
            raise TypeError("Invalid Value Type. Must be int")
        idx_first = df.shape[0] - x
        if idx_first < 0:
            idx_first = 0
        df = df.iloc[idx_first:]

    # filter last n days if n specified
    elif n != None:
        if type(n) != int:
            raise TypeError("Invalid Value Type. Must be int")
        date_dt = dt.datetime.strptime(last_date, "%Y-%m-%d")
        date_dt_first = date_dt - dt.timedelta(days=n)
        date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
        df = df[df['GAME_DATE'] > date_first]

    # drop columns that are not cumulative
    drop_columns = ['GAME_DATE', 'MATCHUP', 'WL', 'W', 'L', 'W_PCT',
                   'GAME_ID', 'TEAM_NAME', 'TEAM_ID', 'TEAM_ABBREVIATION',
                    'TEAM_CITY', 'FG_PCT', 'FG3_PCT', 'FT_PCT',
                   ]
    df = df.drop(columns=drop_columns)

    # filter our advanced columns that use special mean function
    adv_cols = list(df.columns[17:])
    remove = ['AST_TOV', 'NET_RATING', 'E_NET_RATING', 'POSS']
    adv_cols = [x for x in adv_cols if x not in remove]


    # calculate mean traditional stats
    totals = df.sum(axis=0)
    totals['GAMES'] = df.shape[0]
    totals['FG_PCT'] = totals.FGM / totals.FGA
    totals['FG3_PCT'] = totals.FG3M / totals.FG3A
    totals['FT_PCT'] = totals.FTM / totals.FTA

    # calculate mean advanced stats
    for col in adv_cols:
        totals[col] = gmas(col, df)

    # calc remaining adv stats
    totals.NET_RATING = totals.OFF_RATING - totals.DEF_RATING
    totals.E_NET_RATING = totals.E_OFF_RATING - totals.E_DEF_RATING
    totals.AST_TOV = totals.AST / totals.TO

    for i in totals.index:
        if "PCT" in i:
            totals.loc[i] = totals.loc[i] * 100
        elif i == "PIE":
            totals.loc[i] = totals.loc[i] * 100


    return totals.round(1)

def get_per_game_stats_any_team(df, x=None, n=None):
    cum_stats = get_cum_stats_any_team(df=df, x=x, n=n)

    divide_col_index = list(cum_stats.index[:17])
    divide_col_index.append("POSS")

    for idx in divide_col_index:
        name = idx + "_PG"
        cum_stats[name] = cum_stats[idx] / cum_stats.GAMES
        cum_stats.drop(index=idx, inplace=True)

    per_game = cum_stats

    return per_game.round(1)

def get_per_p_possessions_stats_any_team(df, p=100, x=None, n=None):
    cum_stats = get_cum_stats_any_team(df=df, x=x, n=n)

    divide_col_index = list(cum_stats.index[:17])
    divide_col_index.append("POSS")
    total_poss = cum_stats.POSS

    for idx in divide_col_index:
        name = idx + "_P{}P".format(p)
        cum_stats[name] = cum_stats[idx] / total_poss * p
        cum_stats.drop(index=idx, inplace=True)

    per_possessions = cum_stats

    return per_possessions.round(1)


def random_color_func(word=None, font_size=None, position=None, orientation=None,
font_path=None, random_state=None):
    """
    Used in generate_wordcloud method.
    """
    colors = ['#552583', '#FDB927', 'white', '#405ED7', "#FDB927"]
    return random.choice(colors)




################################## Team Date Class
############################################

class TeamDate():
    """
    Create team date object. Note this is only valid for the 2022-23 season and only
teams \
    where Laker players played on during the season.
    """
    # set data path
    path = "assets/"

    def __init__(self, abb, date="2023-04-10"):
        self._set_abb(abb)
        self._set_date(date)
        self._set_stats(abb, date)
        self._set_common_info()
        self._set_full_name()
        self._set_current_seed()
        self.id = load_json_file("teams.json", path=path)[abb]['id']
        self.game_ids = load_json_file("teams.json", path=path)[abb]['game_ids']
        self._set_entities()
        self._set_token_dict()
        self._set_sentiment_df()

    def _set_abb(self, abb):
        if abb in load_json_file("teams.json", path=path).keys():
            self.abb = abb
        else:
            raise ValueError("Invalid Team Abbreviation, try again")

    def _set_date(self, date):
        match = re.findall(r"\d{4}-\d{2}-\d{2}", date)
        if len(match) == 1:
            if date < "2022-10-18":
                self.date = "beginning"
            else:
                self.date = date
        else:
            raise ValueError("Invalid Date Format; must be '%Y-%m-%d'")

    def _set_stats(self, abb, date):
        if self.date == "beginning":
            self.stats = "Date is prior to season started and therefore no stats exist"

        else:
            stats = pd.read_csv(path + "full_team_data_{}.csv".format(abb),
                                index_col=0,
                                parse_dates=['GAME_DATE'])
            stats['GAME_ID'] = '00' + stats['GAME_ID'].astype(str)
            self.stats = stats[stats['GAME_DATE'] <= self.date]
            self.stats['MIN'] = [int(x[:3]) for x in self.stats['MIN']]

    def _set_common_info(self):
        tic_df = pd.read_csv(path + "common_team_info.csv", index_col=0)
        ci = tic_df[tic_df['TEAM_ABBREVIATION'] == self.abb]

        self.common_info = ci

    def _set_full_name(self):
        ci = self.common_info
        full_name = ci['TEAM_CITY'] + " " + ci['TEAM_NAME']
        self.full_name = full_name[0]

    def _set_current_seed(self):
        seeding = pd.read_csv(path + "seedings_data_{}.csv".format(self.abb),
index_col=0)
        ci = self.common_info
        conf = ci['TEAM_CONFERENCE'].values[0]
        if self.date < '2022-10-18':
            self.current_seed = "No current seed! Season hasn't started!"
        elif self.date > '2023-04-09':
            team_seeding = seeding[seeding['DATE'] == '2023-04-09']
            seed = team_seeding.RANK.values[0]
            self.current_seed = conf + " Seed " + str(seed)
        else:
            team_seeding = seeding[seeding['DATE'] == self.date]
            seed = team_seeding.RANK.values[0]
            self.current_seed = conf + " Seed " + str(seed)

    def _set_entities(self):
        with open(path + 'entities_with_nicknames.json', 'r') as f:
            entities = json.load(f)
        entities = dict(entities)
        self.entities = entities

    def _set_token_dict(self):
        with open('assets/token_dicts/post_plus_comment_body_tokenize.json', 'r') as f:
            token_dict = json.load(f)
        self.token_dict = dict(token_dict)

    def _set_sentiment_df(self):
        tsent = pd.read_csv("assets/all_sents/total_sentiment.csv", index_col=0,
parse_dates=['datetime'])
        tsent = tsent[tsent['datetime'] <= self.date]
        self.sentiment_df = tsent

    def _set_fo_sent_df(self):
        fosent = pd.read_csv("assets/all_sents/stok.csv", index_col=0,
parse_dates=['datetime'])
        fosent = fosent[(fosent['player_date'] == 'jeanie_buss') | (fosent['player_date'] ==
'rob_pelinka') | (fosent['player_date'] == 'darvin_ham')]
        self.fo_sent = fosent


############################################### GET TEAM
STATS########################################################
    def get_record(self, x=None, n=None):
        """
        Returns current record or record over last x games or record over last n days.
        """
        # make sure both x and n are not specified
        if (x != None) & (n != None):
            raise ValueError("Can only specify either x games or n days, not both!")

        stats=self.stats

        if x != None:
            if type(x) != int:
```

```
        raise TypeError("Invalid number, must be int")

    idx_first = stats.shape[0] - x
    if idx_first < 0:
        idx_first = 0
    stats = stats.iloc[idx_first:]

    wins = 0
    losses = 0
    for i, row in stats.iterrows():
        if row.WL == 'W':
            wins += 1
        else:
            losses += 1
    return "{}-{}".format(wins, losses)

elif n != None:
    if type(n) != int:
        raise TypeError("Invalid number, must be int")

    date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
    date_dt_first = date_dt - dt.timedelta(days=n)
    date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
    stats = stats[stats['GAME_DATE'] > date_first]
    print(stats.shape[0])

    if len(stats) == 0:
        raise ValueError("No Data available between selected dates")

    wins = 0
    losses = 0
    for i, row in stats.iterrows():
        if row.WL == 'W':
            wins += 1
        else:
            losses += 1
    return "{}-{}".format(wins, losses)

else:
    if self.date == "beginning":
        return "0-0"
    if self.date >= "2023-04-09":
        return "43-39"
    date = find_nearest_date(self.stats, 'GAME_DATE', self.date)
    row = self.stats[self.stats['GAME_DATE'] == date]
    return "{}-{}".format(row.W.values[0], row.L.values[0])
    def current_streak(self):
        if self.date == "beginning":
            return "W0"
        else:
            stats = self.stats

            stats = stats.sort_values('GAME_DATE', ascending=False)
            row1 = stats.iloc[0]
            win_or_loss = row1.WL
            streak = 0

            for _, row in stats.iterrows():
                if (win_or_loss == 'W') & (row.WL == 'W'):
                    streak += 1
                elif (win_or_loss == 'L') & (row.WL == 'L'):
                    streak += 1
                else:
                    break
            return win_or_loss + str(streak)

    def get_cum_stats(self, x=None, n=None):
        """
        Returns team's cumulative stats throughout season. Can specify last x games/or n
days
        Parameters:
            x_games: if set, gets cum stats last x games; default=None
            n_days: if set, gets cum stats last n days; default=None
        """
        # make sure both x and n are not specified
        if (x != None) & (n != None):
            raise ValueError("Can only specify either x games or n days, not both!")

        # define stats
        stats = self.stats

        # filter last x games if x specified
        if x != None:
            if type(x) != int:
                raise TypeError("Invalid Value Type. Must be int")
            idx_first = stats.shape[0] - x
            if idx_first < 0:
                idx_first = 0
            stats = stats.iloc[idx_first:]

        # filter last n days if n specified
```

```
elif n != None:
    if type(n) != int:
        raise TypeError("Invalid Value Type. Must be int")
    date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
    date_dt_first = date_dt - dt.timedelta(days=n)
    date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
    stats = stats[stats['GAME_DATE'] > date_first]

# drop columns that are not cumulative
drop_columns = ['GAME_DATE', 'MATCHUP', 'WL', 'W', 'L', 'W_PCT',
               'GAME_ID', 'TEAM_NAME', 'TEAM_ID', 'TEAM_ABBREVIATION',
                'TEAM_CITY', 'FG_PCT', 'FG3_PCT', 'FT_PCT',
               ]
stats = stats.drop(columns=drop_columns)

# filter our advanced columns that use special mean function
adv_cols = list(stats.columns[17:])
remove = ['AST_TOV', 'NET_RATING', 'E_NET_RATING', 'POSS']
adv_cols = [x for x in adv_cols if x not in remove]


# calculate mean traditional stats
totals = stats.sum(axis=0)
totals['GAMES'] = stats.shape[0]
totals['FG_PCT'] = totals.FGM / totals.FGA
totals['FG3_PCT'] = totals.FG3M / totals.FG3A
totals['FT_PCT'] = totals.FTM / totals.FTA

# calculate mean advanced stats
for col in adv_cols:
    totals[col] = gmas(col, stats)

# calc remaining adv stats
totals.NET_RATING = totals.OFF_RATING - totals.DEF_RATING
totals.E_NET_RATING = totals.E_OFF_RATING - totals.E_DEF_RATING
totals.AST_TOV = totals.AST / totals.TO

for i in totals.index:
    if "PCT" in i:
        totals.loc[i] = totals.loc[i] * 100
    elif i == "PIE":
        totals.loc[i] = totals.loc[i] * 100


return totals.round(1)
```

```
def get_per_game_stats(self, x=None, n=None):
    cum_stats = self.get_cum_stats(x=x, n=n)

    divide_col_index = list(cum_stats.index[:17])
    divide_col_index.append("POSS")

    for idx in divide_col_index:
        name = idx + "_PG"
        cum_stats[name] = cum_stats[idx] / cum_stats.GAMES
        cum_stats.drop(index=idx, inplace=True)

    per_game = cum_stats

    return per_game.round(1)


def get_per_m_minutes_stats(self, m=240, x=None, n=None):
    cum_stats = self.get_cum_stats(x=x, n=n)

    divide_col_index = list(cum_stats.index[:17])
    divide_col_index.append("POSS")
    total_min = cum_stats.MIN

    for idx in divide_col_index:
        name = idx + "_P{}M".format(m)
        cum_stats[name] = cum_stats[idx] / total_min * m
        cum_stats.drop(index=idx, inplace=True)

    per_minutes = cum_stats

    return per_minutes.round(1)

def get_per_p_possessions_stats(self, p=100, x=None, n=None):
    cum_stats = self.get_cum_stats(x=x, n=n)

    divide_col_index = list(cum_stats.index[:17])
    divide_col_index.append("POSS")
    total_poss = cum_stats.POSS

    for idx in divide_col_index:
        name = idx + "_P{}P".format(p)
        cum_stats[name] = cum_stats[idx] / total_poss * p
        cum_stats.drop(index=idx, inplace=True)

    per_possessions = cum_stats
```

```
        return per_possessions.round(1)

    def get_trending_pg(self, x=10, x_comp=10, n=None, n_comp=None, full_szn=True):
        ### get latest stats ###
        stats = self.stats
        per_game = self.get_per_game_stats(x=x, n=n)


        if x != None:
            # get full comp group
            idx_first = stats.shape[0] - x
            if idx_first < 0:
                idx_first = 0
            comp = stats.iloc[:idx_first]

            if full_szn:
                x_comp=None
            comp_pg = get_per_game_stats_any_team(comp, x=x_comp)

        else:
            date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
            date_dt_first = date_dt - dt.timedelta(days=n)
            date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
            comp = stats[stats['GAME_DATE'] <= date_first]

            if full_szn:
                n_comp=None

            comp_pg = get_per_game_stats_any_team(comp, n=n_comp)

        ### compare groups and return groups and comp ###
        diff = (per_game.values - comp_pg.values) / comp_pg.values * 100
        diff = pd.DataFrame(diff, index=comp_pg.index, columns=['PERCENT_DIFF'])
        diff['RAW_DIFF'] = (per_game.values - comp_pg.values)

        return diff, per_game, comp_pg

    def get_trending_p100(self, x=10, x_comp=10, n=None, n_comp=None,
full_szn=True):
        ### get latest stats ###
        stats = self.stats
        per100 = self.get_per_p_possessions_stats(x=x, n=n, p=100)

        if x != None:
            # get full comp group
            idx_first = stats.shape[0] - x
            if idx_first < 0:
                idx_first = 0
            comp = stats.iloc[:idx_first]

            if full_szn:
                x_comp=None
            comp_p100 = get_per_p_possessions_stats_any_team(comp, x=x_comp,
p=100)

        else:
            date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
            date_dt_first = date_dt - dt.timedelta(days=n)
            date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
            comp = stats[stats['GAME_DATE'] <= date_first]

            if full_szn:
                n_comp=None
            comp_p100 = get_per_p_possessions_stats_any_team(comp, n=n_comp,
p=100)


        diff = (per100.values - comp_p100.values) / comp_p100.values * 100
        diff = pd.DataFrame(diff, index=comp_p100.index, columns=['PERCENT_DIFF'])
        diff['RAW_DIFF'] = (per100.values - comp_p100.values)

        return diff, per100, comp_p100

    def get_rolling_per_game_df(self, n=20):
        stats = self.stats
        cols = get_per_game_stats_any_team(stats).index
        dfpg = pd.DataFrame(columns=cols)

        for i, row in stats.iterrows():
            gd_current = row.GAME_DATE
            start_date = gd_current - dt.timedelta(days=n)
            s = stats[(stats['GAME_DATE'] > start_date) & (stats['GAME_DATE'] <=
gd_current)]
            data = list(get_per_game_stats_any_team(s))
            dfpg.loc[i] = data

        dfpg.index=stats.index
        dfpg['GAME_DATE'] = stats.GAME_DATE

        return dfpg

    def get_rolling_per_100_df(self, n=20):
        stats = self.stats
        cols = get_per_p_possessions_stats_any_team(stats).index
        dfp100 = pd.DataFrame(columns=cols)

        for i, row in stats.iterrows():
            gd_current = row.GAME_DATE
            start_date = gd_current - dt.timedelta(days=n)
            s = stats[(stats['GAME_DATE'] > start_date) & (stats['GAME_DATE'] <=
gd_current)]
            data = list(get_per_p_possessions_stats_any_team(s))
            dfp100.loc[i] = data

        dfp100.index=stats.index
        dfp100['GAME_DATE'] = stats.GAME_DATE
        return dfp100

################################## GET SENTIMENT STATS
#####################################
    def get_sentiment(self, post_multiplier=2.5):
        """
        This function simply takes in sent df and returns average sentiment
        Parameters:
            post_multiplier: how much more should we weight post vs comment? Default =
2.5
        """
        # get post sentiment score
        try:
            post_sent = self.sentiment_df[self.sentiment_df['pc'] == 'p']
            post_length = post_sent.shape[0]
            try:
                post_prop_pos = post_sent.sentiment.value_counts().loc['positive'] /
post_length
            except:
                post_prop_pos = 0
            try:
                post_prop_neg = post_sent.sentiment.value_counts().loc['negative'] /
post_length
            except:
                post_prop_neg = 0
            post_score = post_prop_pos - post_prop_neg
        except:
            post_score = 0

        try:
            comment_sent =  self.sentiment_df[self.sentiment_df['pc'] == 'c']
            comment_length = comment_sent.shape[0]
            try:
                comment_prop_pos = comment_sent.sentiment.value_counts().loc['positive'] /
comment_length
            except:
                comment_prop_pos = 0
            try:
                comment_prop_neg = comment_sent.sentiment.value_counts().loc['negative']
/ comment_length
            except:
                comment_prop_neg = 0
            comment_score = comment_prop_pos - comment_prop_neg
        except:
            comment_score = 0

        if (post_score == 0) & (comment_score == 0):
            weighted = 0
        else:
            # get weighted score
            num = (post_score * post_length * post_multiplier) + (comment_score *
comment_length)
            denom = (post_length * post_multiplier) + comment_length
            weighted = num / denom

        return weighted

    def get_trending_sentiment(self, n=20, n_comp=None):
        if type(n) != int:
            raise TypeError("Invalid Value Type. Must be int")

        sent = self.sentiment_df
        date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
        date_dt_first = date_dt - dt.timedelta(days=n)
        date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
        trend = sent[sent['datetime'] > date_first]

        sent_score_last_n = ss.get_sentiment(trend)

        comp = sent[sent['datetime'] <= date_first]
        if n_comp != None:
            date_dt2 = dt.datetime.strptime(date_first, "%Y-%m-%d")
            date_dt_first2 = date_dt2 - dt.timedelta(days=n_comp)
            date_first2 = dt.datetime.strftime(date_dt_first2, "%Y-%m-%d")
            comp = comp[comp['datetime'] > date_first2]
        sent_score_comp = ss.get_sentiment(comp)
        diff = (sent_score_last_n - sent_score_comp) / sent_score_comp
        return sent_score_last_n, sent_score_comp, diff

    def get_mentions_per_day(self):
        df = self.sentiment_df
        df = df[['datetime', 'sentiment']]
        idx = pd.period_range(min(df.datetime), max(df.datetime)).astype('datetime64[ns]')
        mentions = df.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0)

        return mentions.sentiment.mean()

    def get_trending_mentions_per_day(self, n=14, n_comp=None):
        if type(n) != int:
            raise TypeError("Invalid Value Type. Must be int")

        df = self.sentiment_df
        df = df[['datetime', 'sentiment']]
        idx = pd.period_range(min(df.datetime), max(df.datetime)).astype('datetime64[ns]')
        mentions = df.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0)

        date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
        date_dt_first = date_dt - dt.timedelta(days=n)
        date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
        trend = mentions[mentions.index > date_first]
        trend = trend.sentiment.mean()

        comp = mentions[mentions.index <= date_first]
        if n_comp != None:
            date_dt2 = dt.datetime.strptime(date_first, "%Y-%m-%d")
            date_dt_first2 = date_dt2 - dt.timedelta(days=n_comp)
            date_first2 = dt.datetime.strftime(date_dt_first2, "%Y-%m-%d")
            comp = comp[comp.index > date_first2]

        comp = comp.sentiment.mean()

        diff = (trend - comp) / comp

        return trend, comp, diff

    def get_rolling_sentiment_score_df(self, n=20):
        sent = self.sentiment_df
        cols = ['weighted']
        idx = sorted(list(set(sent['datetime'])))
        dfs = pd.DataFrame(columns=cols, index=idx)
        weighted=[]
        for i in idx:
            start = i - dt.timedelta(days=20)
            s = sent[(sent['datetime'] > start) & (sent['datetime'] <= i) ]
            w = ss.get_sentiment(s)
            weighted.append(w)
        dfs['weighted'] = weighted

        return dfs

    def get_rolling_mentions_df(self, n=20):
        df = self.sentiment_df
        df = df[['datetime', 'sentiment']]
        idx = pd.period_range(min(df.datetime), max(df.datetime)).astype('datetime64[ns]')
        df = df.set_index('datetime')
        mentions = df.groupby('datetime').count().reindex(idx, fill_value=0)
        cols = ['mentions']
        dfm = pd.DataFrame(columns=cols, index=idx)

        ms = []
        for i in idx:
            start = i - dt.timedelta(days=n)
            mm = mentions[(mentions.index > start) &(mentions.index <= i)]
            m = mm.sentiment.mean()
            ms.append(m)
        dfm['mentions'] = ms

        return dfm

    def get_rolling_team_wpct_df(self, n=20):
        team_stats = self.stats
        idx = pd.period_range(min(team_stats.GAME_DATE),
max(team_stats.GAME_DATE)).astype('datetime64[ns]')
        cols = ['wpct']
        dft = pd.DataFrame(columns=cols, index=idx)

        ws = []
        for i in idx:
            start = i - dt.timedelta(days=n)
            ww = team_stats[(team_stats.GAME_DATE > start) &
(team_stats.GAME_DATE <= i)].reset_index().drop(columns=['index'])
            wins = ww.iloc[-1].W - ww.W[0]
            losses = ww.iloc[-1].L - ww.L[0]
            wpct = wins / (wins + losses)
            ws.append(wpct)
        dft['wpct'] = ws
        return dft



################################################ PLOTS AND
VISUALIZATIONS ################################################
    def plot_sentiment_through_time(self):
        post_sent = self.sentiment_df[self.sentiment_df['pc'] == 'p']
        comment_sent =  self.sentiment_df[self.sentiment_df['pc'] == 'c']

        ## multiply poss by post_mult and combine post and comment sents w dates
        posts = pd.concat([post_sent, post_sent,
post_sent]).reset_index().drop(columns=['index'])
        posts = posts[['id', 'datetime', 'sentiment']]
        comments = comment_sent[['id', 'datetime', 'sentiment']]
        combined = pd.concat([posts,
comments]).set_index('datetime').sort_values('datetime')
        series = pd.Series(combined['sentiment'], index=combined.index)

        # get scores
        co_pos = series[series == 'positive']
        co_neg = series[series == 'negative']
        co_neu = series[series == 'neutral']

        # get index to fillnas
        idx = pd.period_range(min(series.index),
max(series.index)).astype('datetime64[ns]')

        # resample in 3D blocks
        pos_resamp = co_pos.groupby('datetime').count().reindex(idx,
fill_value=0).resample('3D').sum()
        neg_resamp = co_neg.groupby('datetime').count().reindex(idx,
fill_value=0).resample('3D').sum()
        neu_resamp = co_neu.groupby('datetime').count().reindex(idx,
fill_value=0).resample('3D').sum()

        combined_resamp = pd.concat([pos_resamp, neg_resamp, neu_resamp], axis=1)
        combined_resamp.columns = ['positive', 'negative', 'neutral']

        combined_resamp['pos_ratio'] = combined_resamp.positive /
(combined_resamp.positive + combined_resamp.negative + combined_resamp.neutral)
        combined_resamp['neg_ratio'] = combined_resamp.negative /
(combined_resamp.positive + combined_resamp.negative + combined_resamp.neutral)
        combined_resamp['score'] = combined_resamp.pos_ratio -
combined_resamp.neg_ratio
```

```
# Create the figure
fig = make_subplots(specs=[[{"secondary_y": True}]])

# Add the positive bars
fig.add_trace(
   go.Bar(
      x=pos_resamp.index,
      y=pos_resamp,
      name='Positive',
      marker_color='lightgreen',
   ),
   secondary_y=False,
)

# Add the negative bars
fig.add_trace(
   go.Bar(
      x=neg_resamp.index,
      y=-neg_resamp,
      name='Negative',
      marker_color='lightcoral',
   ),
   secondary_y=False,
)

fig.add_trace(
   go.Scatter(
      x=combined_resamp.index,
      y=combined_resamp['score'],
      name='Score',
      marker_color='#FDB927',
   ),
   secondary_y=True,
)

custom_template = templates["plotly_dark"]
custom_template.layout["xaxis"]["showgrid"] = False
custom_template.layout["yaxis"]["showgrid"] = False

# Configure the layout
fig.update_layout(
   title=dict(
      text='Sentiment Through Time'
   ),
            xaxis_title='Date',
            yaxis_title='Number of Impressions',
            barmode='overlay',
            bargap=0,
            barnorm=None,
            height=600,
            margin=dict(l=50, r=50, b=100, t=100, pad=4),
            template=custom_template,
            plot_bgcolor='#552583',
            paper_bgcolor='#552583',
            yaxis2=dict(showgrid=False, zeroline=False)

        )

        return fig

    def generate_wordcloud(self):
        token_dict = dict(self.token_dict)
        wordcloud = WordCloud(width=1600,
                              height=400,
                              background_color='#311149',
                              color_func=random_color_func,
                              min_font_size=10).generate_from_frequencies(token_dict)
        #plt.imshow(wordcloud, interpolation='bilinear')
        #plt.axis('off')
        return wordcloud.to_image()


    def basic_pie_chart(self):
        df = self.sentiment_df
        sentiment_col = df.sentiment
        labels = list(sentiment_col.value_counts().index)
        values = list(sentiment_col.value_counts())
        colors = {'positive':'lightgreen',
                  'negative':'lightcoral',
                  'neutral':'white'}
        colors = [colors[x] for x in labels]

        fig = go.Figure(data=[go.Pie(labels=labels, values=values, showlegend=False)])

        fig.update_traces(hoverinfo='label+percent',
                          textinfo='percent+label',
                          textposition='inside',
                          textfont_size=20,
                          marker=dict(colors=colors,
                                      line=dict(color='#000000',width=2)))
        fig.update_layout(
            margin=dict(l=0, r=0, t=0, b=0),
        )

        return fig

    def plot_top_ten(self, kind='all'):
        if kind not in ['all', 'positive', 'negative']:
            raise ValueError("Invalid kind, must be in 'all', 'positive', 'negative'")

        if kind == 'all':
            tok = sorted(self.token_dict.items(), key=lambda x:x[1], reverse=True)
        elif kind == 'positive':
            with open("assets/token_dicts/pos.json", "r") as f:
                token_dict = json.load(f)
            tok = sorted(token_dict.items(), key=lambda x:x[1], reverse=True)
        else:
            with open("assets/token_dicts/neg.json", "r") as f:
                token_dict = json.load(f)
            tok = sorted(token_dict.items(), key=lambda x:x[1], reverse=True)

        tok = tok[:50]

        blacklist = [
            'lakers', 'lakers_team', 'lebron_james', 'anthony_davis',
            'russell_westbrook'
        ]
        toks = []
        counts = []
        for i, _ in enumerate(tok):
            if tok[i][0] not in blacklist:
                toks.append(tok[i][0])
                counts.append(tok[i][1])

        toks = toks[:10]
        counts = counts[:10]
        toks.reverse()
        counts.reverse()

        fig = go.Figure(
            go.Bar(
                x=counts,
                y=toks,
                orientation='h',
                marker_color='#FDB927'
            )
        )

        fig.update_layout(
            title={
                'text': "Top 10 Frequency Terms",
                'x':0.2,
                'y':0.93
            },
            xaxis=dict(
                title="frequencies",
                showgrid=False,
                showline=False,
            ),
            yaxis=dict(
                tickmode='linear',
                showgrid=False,
                showline=False,
                automargin=True,

            ),
            plot_bgcolor='#552583',
            paper_bgcolor='#552583',
            margin=dict(l=20, r=20, t=50, b=20),
            font=dict(
                color='white',
                family='Montserrat, Helvetica, Arial, sans-serif',
            ),
        )
        return fig

    def plot_top_ten_emoji(self, kind='all'):
        if kind not in ['all', 'positive', 'negatives']:
            raise ValueError("Invalid kind, must be in 'all', 'positive', 'negative'")

        if kind == 'all':
            tok = sorted(self.token_dict.items(), key=lambda x:x[1], reverse=True)
        elif kind == 'positive':
            with open("assets/token_dicts/pos.json", "r") as f:
                token_dict = json.load(f)
            tok = sorted(token_dict.items(), key=lambda x:x[1], reverse=True)
        else:
            with open("assets/token_dicts/neg.json", "r") as f:
                token_dict = json.load(f)
            tok = sorted(token_dict.items(), key=lambda x:x[1], reverse=True)

        tok = tok[:50]
```

```
emj = ss.get_emoji_dict(player_ref='team', kind=kind)
emj = emj[:50]

emjs = []
counts = []
for i, _ in enumerate(emj):
    emjs.append(emj[i][0])
    counts.append(emj[i][1])

emjs = emjs[:10]
counts = counts[:10]
emjs.reverse()
counts.reverse()

fig = go.Figure(
        go.Bar(
            x=counts,
            y=emjs,
            orientation='h',
            marker_color='#FDB927'
        )
    )

fig.update_layout(
    title={
        'text': "Top 10 Frequency Emojis",
        'x':0.2,
        'y':0.93
    },
    xaxis=dict(
        title="frequencies",
        showgrid=False,
        showline=False,
    ),
    yaxis=dict(
        tickmode='linear',
        showgrid=False,
        showline=False,
        automargin=True,
        tickfont=dict(size=20)

    ),
    plot_bgcolor='#552583',
    paper_bgcolor='#552583',
            margin=dict(l=20, r=20, t=50, b=20),
            font=dict(
                color='white',
                family='Montserrat, Helvetica, Arial, sans-serif',
            ),
        )
        return fig

    def sent_vs_stat(self, stat_col, kind='g'):
        if kind not in ['g', 'p', 't']:
            raise ValueError("Invalid stat type")

        adj_cols=[
            'MIN', 'FGM', 'FGA', 'FG3M', 'FG3A', 'FTM', 'FTA', 'OREB', 'DREB', 'REB',
            'AST', 'STL', 'BLK', 'TO', 'PF', 'PTS', 'PLUS_MINUS', 'POS_PG'
        ]

        ## Load all needed stats ##
        sent = self.get_rolling_sentiment_score_df()

        if kind == 'g':
            stats = self.get_rolling_per_game_df()
            suffix = '_PG'
        elif kind == 'p':
            stats = self.get_rolling_per_100_df()
            suffix = '_P100P'
        else:
            stats = self.get_rolling_team_wpct_df()
            stat_col = 'wpct'
            stats['GAME_DATE'] = stats.index

        if stat_col in adj_cols:
            stat_col = stat_col + suffix


        # Create the figure

        fig = make_subplots(specs=[[{"secondary_y": True}]])

        fig.add_trace(
            go.Scatter(
                y=sent.weighted,
                x=sent.index,
                name='Sentiment',
                line=dict(color='#FDB927')
            ),
            secondary_y=False,
        )

        fig.add_trace(
            go.Scatter(
                x=stats.GAME_DATE,
                y=stats[stat_col],
                name='{}'.format(stat_col),
                line=dict(color='#840DFA')
            ),
            secondary_y=True,
        )

        custom_template = templates["plotly_dark"]
        custom_template.layout["xaxis"]["showgrid"] = False
        custom_template.layout["yaxis"]["showgrid"] = False

        fig.update_yaxes(title_text="Rolling 20-day MA Sentiment Score",
title_font=dict(color='#FDB927'), secondary_y=False)
        fig.update_yaxes(title_text="Rolling 20-day MA {}".format(stat_col),
title_font=dict(color='#840DFA'), secondary_y=True)

        fig.update_layout(
                title='Rolling 20-day Sentiment vs {}'.format(stat_col),
                xaxis_title='Date',
                template=custom_template,
                plot_bgcolor='black',
                paper_bgcolor='black',
            )

        return fig

    def sent_vs_stat_corr(self, stat_col, kind='g'):
        if kind not in ['g', 'p', 't']:
            raise ValueError("Invalid stat type")

        adj_cols=[
                'MIN', 'FGM', 'FGA', 'FG3M', 'FG3A', 'FTM', 'FTA', 'OREB', 'DREB', 'REB',
                'AST', 'STL', 'BLK', 'TO', 'PF', 'PTS', 'PLUS_MINUS', 'POS_PG'
            ]

        sent = self.get_rolling_sentiment_score_df()

        if kind == 'g':
            stats = self.get_rolling_per_game_df()
            stats['GAME_DATE'] = stats.GAME_DATE.dt.date
            suffix = '_PG'
        elif kind == 'p':
            stats = self.get_rolling_per_100_df()
            stats['GAME_DATE'] = stats.GAME_DATE.date
            suffix = '_P100P'
        else:
            stats = self.get_rolling_team_wpct_df()
            stat_col = 'wpct'
            stats.loc['2022-10-18'] = 0
            stats.loc['2022-10-19'] = 0

        if stat_col in adj_cols:
            stat_col = stat_col + suffix

        x = stats

        if kind != 't':
            xmin = stats.GAME_DATE.min()
            xmax = stats.GAME_DATE.max()
            y = sent.loc[xmin:xmax]
            date_range = pd.date_range(start=x.GAME_DATE.min(),
end=x.GAME_DATE.max(), freq='D')
            x = x.set_index('GAME_DATE')
            new_index = pd.Index(date_range, name='Date')
            x = x.reindex(new_index).ffill()

            corr = y['weighted'].corr(x[stat_col])
        else:
            xmin = stats.index.min()
            xmax = stats.index.max()
            y = sent.loc[xmin:xmax]
            corr = y['weighted'].corr(x[stat_col])
        return corr


############################## Player Date Class
############################################

class PlayerDate():
    # set data path
    path = "assets/"
    def __init__(self, name, date="2023-04-10"):
        self._set_name(name)
        self._set_date(date)
        self._set_pid()
        self._set_transactions()
        self._set_acquired()
        self._set_moved()
        self._set_waived()
        self._set_signed()
        self._set_traded_away()
        self._set_traded_for()
        self._set_other_teams()
        self._set_common_info()
        self._set_stats()
        self._set_laker_stats()
        self._set_non_laker_stats()
        self._set_ent_key()
        self._set_entities()
        self._set_token_dict()
        self._set_sentiment_df()
        self._set_laker_sentiment_df()


    def _set_name(self, name):
        if name in load_json_file("players.json", path=path).keys():
            self.name = name
        else:
            raise ValueError("Invalid player name, try again")

    def _set_date(self, date):
        match = re.findall(r"\d{4}-\d{2}-\d{2}", date)
        if len(match) == 1:
            if date < "2022-10-18":
                self.date = "beginning"
            else:
                self.date = date
        else:
            raise ValueError("Invalid Date Format; must be '%Y-%m-%d'")

    def _set_pid(self):
        players = load_json_file("players.json", path=path)
        self.pid = players[self.name]

    def _set_transactions(self):
        transactions = load_json_file("transactions.json", path=path)
        if self.name not in transactions.keys():
            self.transactions = None
        else:
            self.transactions = transactions[self.name]

    def _set_acquired(self):
        if self.transactions == None:
            self.acquired = False
        elif self.transactions['date_acquired'] == None:
            self.acquired = False
        else:
            self.acquired = self.transactions['date_acquired']

    def _set_moved(self):
        if self.transactions == None:
            self.moved = False
        elif self.transactions['date_moved'] == None:
            self.moved = False
        else:
            self.moved = self.transactions['date_moved']
            self.date = self.moved

    def _set_waived(self):
        if self.transactions == None:
            self.waived= False
        elif self.transactions['waived'] == False:
            self.waived = False
        else:
            self.waived = "Waived on {}".format(self.moved)

    def _set_signed(self):
        if self.transactions == None:
            self.signed= False
        elif self.transactions['signed'] == False:
            self.signed = False
        else:
            self.signed = "Signed on {}".format(self.acquired)

    def _set_traded_away(self):
        if self.transactions == None:
            self.traded_away = False
        elif self.transactions['traded_away'] == False:
            self.traded_away = False
        else:
            self.traded_away = "Traded on {}".format(self.moved)

    def _set_traded_for(self):
        if self.transactions == None:
            self.traded_for= False
        elif self.transactions['traded_for'] == False:
            self.traded_for = False
        else:
            self.traded_for = "Traded for on {}".format(self.acquired)

    def _set_other_teams(self):
        if self.transactions == None:
            self.other_teams = False
        else:
            self.other_teams = self.transactions['other_teams']

    def _set_common_info(self):
        pic_df = pd.read_csv(path + "common_player_info.csv", index_col=0)
        ci = pic_df[pic_df['PERSON_ID'] == self.pid]
        self.common_info = ci

    def _set_stats(self):
        if self.date == "beginning":
            self.stats = "Date is prior to season started and therefore no stats exist"
        else:
            stats = pd.read_csv(path + "player_total_total.csv",
                                index_col=0,
                                parse_dates=['GAME_DATE'])
            stats = stats.sort_values('GAME_DATE')
            stats = stats[stats['PLAYER_ID'] ==
self.pid].reset_index().drop(columns=('index'))
            stats = stats.drop(columns=['Game_ID', 'Team_ID', 'PACE_PER40'])
            stats = stats[stats['GAME_DATE'] <= self.date]
            self.stats = stats

    def _set_laker_stats(self):
        if self.transactions == None:
            self.laker_stats = self.stats
        else:
            stats = self.stats
            laker_stats = stats[stats['TEAM_ABBREVIATION'] == 'LAL']
            self.laker_stats = laker_stats.reset_index().drop(columns=('index'))

    def _set_non_laker_stats(self):
        if self.transactions == None:
            self.non_laker_stats = None

        else:
            stats = self.stats
            non_laker_stats = stats[stats['TEAM_ABBREVIATION'] != 'LAL']
            self.non_laker_stats = non_laker_stats

    def _set_ent_key(self):
        with open(path + "name_conv.json", "r") as f:
            name_conv = json.load(f)
        name_conv = dict(name_conv)

        self.ent_key = name_conv[self.name]

    def _set_entities(self):
        with open(path + 'entities_with_nicknames.json', 'r') as f:
            entities = json.load(f)
        entities = dict(entities)

        self.entities = entities[self.ent_key]

    def _set_token_dict(self):
        with open('assets/token_dicts/{}_token_dicts.json'.format(self.ent_key), 'r') as f:
            token_dict = json.load(f)
        self.token_dict = dict(token_dict)

    def _set_sentiment_df(self):
        psent = pd.read_csv("assets/all_sents/stok.csv", index_col=0,
parse_dates=['datetime'])
        psent_filtered = psent[psent['player_ref'] == self.entities['full_name']]
        psent_filtered = psent_filtered[psent_filtered['datetime'] <= self.date]
        self.sentiment_df = psent_filtered.reset_index().drop(columns=['index'])

    def _set_laker_sentiment_df(self):
        psent_filtered = self.sentiment_df

        if (not self.acquired) & (not self.moved):
            self.laker_sentiment_df = psent_filtered

        elif not self.acquired:
            psent_filtered = psent_filtered[psent_filtered['datetime'] <= self.moved]
            self.laker_sentiment_df = psent_filtered.reset_index().drop(columns=['index'])

        else:
            psent_filtered = psent_filtered[psent_filtered['datetime'] >= self.acquired]
            self.laker_sentiment_df = psent_filtered.reset_index().drop(columns=['index'])

    ################################################## GET PLAYER STATS
##################################################
    def get_cum_stats(self, x=None, n=None, laker_only=False):
        """
        Returns player's cumulative stats throughout season. /
        Can specify last x games/or n days and also whether to include all stats or laker
only
        Parameters:
```

```
    x_games: if set, gets cum stats last x games; default=None
    n_days: if set, gets cum stats last n days; default=None
"""
if (x != None) & (n != None):
    raise ValueError("Can only specify either x games or n days, not both!")

# define stats
if laker_only:
    stats = self.laker_stats
else:
    stats = self.stats

# drop games with 0 minutes played:
stats = stats[~stats['MIN'].isna()].reset_index().drop(columns=("index"))

# filter last x games if x specified
if x != None:
    if type(x) != int:
        raise TypeError("Invalid Value Type. Must be int")
    idx_first = stats.shape[0] - x
    if idx_first < 0:
        idx_first = 0
    stats = stats.iloc[idx_first:]

# filter last n days if n specified
elif n != None:
    if type(n) != int:
        raise TypeError("Invalid Value Type. Must be int")
    date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
    date_dt_first = date_dt - dt.timedelta(days=n)
    date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
    stats = stats[stats['GAME_DATE'] > date_first]


# drop columns that are not cumulative
drop_columns = ['GAME_ID', 'TEAM_ID', 'TEAM_ABBREVIATION', 'TEAM_CITY',
                'PLAYER_ID', 'PLAYER_NAME', 'FG_PCT', 'FG3_PCT', 'FT_PCT',
                'GAME_DATE', 'MATCHUP', 'WL', 'TM_TOV_PCT']

stats = stats.drop(columns=drop_columns)

# filter our advanced columns that use special mean function
adv_cols = list(stats.columns[16:36])
remove = ['AST_TOV', 'NET_RATING', 'E_NET_RATING', 'POSS']
adv_cols = [x for x in adv_cols if x not in remove]
```

```
        # calculate mean traditional stats
        totals = stats.sum(axis=0)
        totals['GAMES'] = stats.shape[0]
        totals['FG_PCT'] = totals.FGM / totals.FGA
        totals['FG3_PCT'] = totals.FG3M / totals.FG3A
        totals['FT_PCT'] = totals.FTM / totals.FTA

        # calculate mean advanced stats
        for col in adv_cols:
            totals[col] = gmas(col, stats)

        # calc remaining adv stats
        totals.NET_RATING = totals.OFF_RATING - totals.DEF_RATING
        totals.E_NET_RATING = totals.E_OFF_RATING - totals.E_DEF_RATING
        totals.AST_TOV = totals.AST / totals.TO

        # format data and return
        for i in totals.index:
            if "PCT" in i:
                totals.loc[i] = totals.loc[i] * 100
            elif i == "PIE":
                totals.loc[i] = totals.loc[i] * 100
        return totals.round(1)

    def get_per_game_stats(self, x=None, n=None, laker_only=False):
        cum_stats = self.get_cum_stats(x=x, n=n, laker_only=laker_only)

        divide_col_index = list(cum_stats.index[:16])
        divide_col_index.append("POSS")
        divide_col_index.append("MIN")

        for idx in divide_col_index:
            name = idx + "_PG"
            cum_stats[name] = cum_stats[idx] / cum_stats.GAMES
            cum_stats.drop(index=idx, inplace=True)

        per_game = cum_stats

        return per_game.round(1)

    def get_per_m_minutes_stats(self, m=36, x=None, n=None, laker_only=False):
        cum_stats = self.get_cum_stats(x=x, n=n)
        divide_col_index = list(cum_stats.index[:16])
        divide_col_index.append("POSS")
        divide_col_index.append("MIN")
        total_min = cum_stats.MIN
        for idx in divide_col_index:
            name = idx + "_P{}M".format(m)
            cum_stats[name] = cum_stats[idx] / total_min * m
            cum_stats.drop(index=idx, inplace=True)

        per_minutes = cum_stats

        return per_minutes.round(1)

    def get_per_p_possessions_stats(self, p=100, x=None, n=None, laker_only=False):
        cum_stats = self.get_cum_stats(x=x, n=n)

        divide_col_index = list(cum_stats.index[:16])
        divide_col_index.append("POSS")
        divide_col_index.append("MIN")
        total_poss = cum_stats.POSS

        for idx in divide_col_index:
            name = idx + "_P{}P".format(p)
            cum_stats[name] = cum_stats[idx] / total_poss * p
            cum_stats.drop(index=idx, inplace=True)

        per_possessions = cum_stats

        return per_possessions.round(1)

    def get_trending_pg(self, x=10, x_comp=10, n=None, n_comp=None,
laker_only=False, full_szn=True):
        ### get latest stats ###
        per_game = self.get_per_game_stats(x=x, n=n, laker_only=laker_only)

        if laker_only:
            stats = self.laker_stats
        else:
            stats = self.stats

        ### get compare group stats for x games ###
        if x != None:
            # get full comp group
            idx_first = stats.shape[0] - x
            if idx_first < 0:
                idx_first = 0
            comp = stats.iloc[:idx_first]

            if full_szn:
                x_comp=None

            comp_pg = get_per_game_stats_any(comp, x=x_comp)

        ### get compare group stats for n days ###
        else:
            # get full comp group
            date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
            date_dt_first = date_dt - dt.timedelta(days=n)
            date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
            comp = stats[stats['GAME_DATE'] <= date_first]

            if full_szn:
                n_comp=None

            comp_pg = get_per_game_stats_any(comp, n=n_comp)


        ### compare groups and return groups and comp ###
        diff = (per_game.values - comp_pg.values) / comp_pg.values * 100
        diff = pd.DataFrame(diff, index=comp_pg.index, columns=['PERCENT_DIFF'])
        diff['RAW_DIFF'] = (per_game.values - comp_pg.values)

        return diff, per_game, comp_pg

    def get_trending_p36(self, x=10, x_comp=10, n=None, n_comp=None,
laker_only=False, full_szn=True):
        ### get latest stats ###
        per36 = self.get_per_m_minutes_stats(x=x, n=n, laker_only=laker_only, m=36)

        if laker_only:
            stats = self.laker_stats
        else:
            stats = self.stats

        ### get compare group stats for x games ###
        if x != None:
            # get full comp group
            idx_first = stats.shape[0] - x
            if idx_first < 0:
                idx_first = 0
            comp = stats.iloc[:idx_first]

            if full_szn:
                x_comp=None
            comp_p36 = get_per_m_minutes_stats_any(comp, x=x_comp, m=36)

        ### get compare group stats for n days ###
        else:
            # get full comp group
            date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
            date_dt_first = date_dt - dt.timedelta(days=n)
            date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
            comp = stats[stats['GAME_DATE'] <= date_first]

            if full_szn:
                n_comp=None

            comp_p36 = get_per_m_minutes_stats_any(comp, n=n_comp, m=36)

        ### compare groups and return comp ###
        diff = (per36.values - comp_p36.values) / comp_p36.values * 100
        diff = pd.DataFrame(diff, index=comp_p36.index, columns=['PERCENT_DIFF'])
        diff['RAW_DIFF'] = (per36.values - comp_p36.values)

        return diff, per36, comp_p36

    def get_trending_p100(self, x=10, x_comp=10, n=None, n_comp=None,
laker_only=False, full_szn=True):
        ### get latest stats ###
        per100 = self.get_per_p_possessions_stats(x=x, n=n, laker_only=laker_only,
p=100)

        if laker_only:
            stats = self.laker_stats
        else:
            stats = self.stats

        ### get compare group stats for x games ###
        if x != None:
            # get full comp group
            idx_first = stats.shape[0] - x
            if idx_first < 0:
                idx_first = 0
            comp = stats.iloc[:idx_first]

            if full_szn:
                x_comp=None

            comp_p100 = get_per_p_possessions_stats_any(comp, x=x_comp, p=100)
        ### get compare group stats for n days ###
        else:
            # get full comp group
            date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
            date_dt_first = date_dt - dt.timedelta(days=n)
            date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
            comp = stats[stats['GAME_DATE'] <= date_first]

            if full_szn:
                n_comp=None

            comp_p100 = get_per_p_possessions_stats_any(comp, n=n_comp, p=100)

        ### compare groups and return comp ###
        diff = (per100.values - comp_p100.values) / comp_p100.values * 100
        diff = pd.DataFrame(diff, index=comp_p100.index, columns=['PERCENT_DIFF'])
        diff['RAW_DIFF'] = (per100.values - comp_p100.values)

        return diff, per100, comp_p100

    def get_trend_lal_non_lal_pg(self, x=10, xcomp=10, n=None, n_comp=None,
full_szn_lal=True, full_szn_other=True):
        """
        Compare laker vs non-laker stats per game. Can compare last x games or n days
of each.
        """
        # get lalppg
        if full_szn_lal:
            pglal = self.get_per_game_stats(laker_only=True)

        else:
            pglal = self.get_per_game_stats(x=x, n=n, laker_only=True)

        # get comp group
        nonlal = self.non_laker_stats

        if full_szn_other:
            comp = get_per_game_stats_any(nonlal)
        else:
            comp = get_per_game_stats_any(nonlal, x=xcomp, n=n_comp)

        ### compare groups and return groups and comp ###
        diff = (pglal.values - comp.values) / comp.values * 100
        diff = pd.DataFrame(diff, index=comp.index, columns=['PERCENT_DIFF'])
        diff['RAW_DIFF'] = (pglal.values - comp.values)
        return diff, pglal, comp

    def get_trend_lal_non_lal_p36(self, x=10, xcomp=10, n=None, n_comp=None,
full_szn_lal=True, full_szn_other=True):
        """
        Compare laker vs non-laker stats per 36 min. Can compare last x games or n days
of each.
        """
        # get lalp36
        if full_szn_lal:
            p36lal = self.get_per_m_minutes_stats(laker_only=True, m=36)
        else:
            p36lal = self.get_per_m_minutes_stats(x=x, n=n, laker_only=True, m=36)

        # get comp group
        nonlal = self.non_laker_stats

        if full_szn_other:
            comp = get_per_m_minutes_stats_any(nonlal, m=36)
        else:
            comp = get_per_m_minutes_stats_any(nonlal, x=x_comp, n=n_comp, m=36)


        ### compare groups and return groups and comp ###
        diff = (p36lal.values - comp.values) / comp.values * 100
        diff = pd.DataFrame(diff, index=comp.index, columns=['PERCENT_DIFF'])
        diff['RAW_DIFF'] = (p36lal.values - comp.values)

        return diff, p36lal, comp

    def get_trend_lal_non_lal_p100(self, x=10, xcomp=10, n=None, n_comp=None,
full_szn_lal=True, full_szn_other=True):
        """
        Compare laker vs non-laker stats per 100 poss. Can compare last x games or n
days of each.
        """
        # get lalp100
        if full_szn_lal:
            p100lal = self.get_per_p_possessions_stats(laker_only=True, p=100)
        else:
            p100lal = self.get_per_p_possessions_stats(x=x, n=n, laker_only=True, p=100)

        # get comp group
        nonlal = self.non_laker_stats

        if full_szn_other:
            comp = get_per_p_possessions_stats_any(nonlal, p=100)
        else:
            comp = get_per_p_possessions_stats_any(nonlal, x=x_comp, n=n_comp,
p=100)

        ### compare groups and return groups and comp ###
        diff = (p100lal.values - comp.values) / comp.values * 100
        diff = pd.DataFrame(diff, index=comp.index, columns=['PERCENT_DIFF'])
        diff['RAW_DIFF'] = (p100lal.values - comp.values)

        return diff, p100lal, comp

    def get_rolling_per_game_df(self, n=20):
        stats = self.laker_stats
        cols = get_per_game_stats_any(stats).index
        dfpg = pd.DataFrame(columns=cols)

        for i, row in stats.iterrows():
            gd_current = row.GAME_DATE
            start_date = gd_current - dt.timedelta(days=n)
            s = stats[(stats['GAME_DATE'] > start_date) & (stats['GAME_DATE'] <=
gd_current)]
            try:
                data = list(get_per_game_stats_any(s))
                dfpg.loc[i] = data
            except:
                if i == 0:
                    dfpg.loc[0] = 0
                else:
                    dfpg.loc[i] = dfpg.loc[i-1]

        dfpg.index=stats.index
        dfpg['GAME_DATE'] = stats.GAME_DATE

        return dfpg

    def get_rolling_per_36_df(self, n=20):
        stats = self.stats
        cols = get_per_m_minutes_stats_any(stats).index
        dfp36 = pd.DataFrame(columns=cols)

        for i, row in stats.iterrows():
            gd_current = row.GAME_DATE
            start_date = gd_current - dt.timedelta(days=n)
            s = stats[(stats['GAME_DATE'] > start_date) & (stats['GAME_DATE'] <=
gd_current)]
```

```
            try:
                data = list(get_per_m_minutes_stats_any(s))
                dfp36.loc[i] = data
            except:
                if i == 0:
                    dfp36.loc[0] = 0
                else:
                    dfp36.loc[i] = dfp36.loc[i-1]
        dfp36.index=stats.index
        dfp36['GAME_DATE'] = stats.GAME_DATE

        return dfp36

    def get_rolling_per_100_df(self, n=20):
        stats = self.laker_stats
        cols = get_per_p_possessions_stats_any(stats).index
        dfp100 = pd.DataFrame(columns=cols)

        for i, row in stats.iterrows():
            gd_current = row.GAME_DATE
            start_date = gd_current - dt.timedelta(days=n)
            s = stats[(stats['GAME_DATE'] > start_date) & (stats['GAME_DATE'] <=
gd_current)]
            try:
                data = list(get_per_p_possessions_stats_any(s))
                dfp100.loc[i] = data
            except:
                if i == 0:
                    dfp100.loc[0] = 0
                else:
                    dfp100.loc[i] = dfp100.loc[i-1]
        dfp100.index=stats.index
        dfp100['GAME_DATE'] = stats.GAME_DATE
        return dfp100

################################################ GET SENTIMENT STATS
################################################

    def get_sentiment(self, post_multiplier=2.5):
        """
        This function simply takes in sent df and returns average sentiment
        Parameters:
            post_multiplier: how much more should we weight post vs comment? Default =
2.5
        """
        # get post sentiment score
        try:
            post_sent = self.laker_sentiment_df[self.laker_sentiment_df['pc'] == 'p']
            post_length = post_sent.shape[0]
            try:
                post_prop_pos = post_sent.sentiment.value_counts().loc['positive'] /
post_length
            except:
                post_prop_pos = 0
            try:
                post_prop_neg = post_sent.sentiment.value_counts().loc['negative'] /
post_length
            except:
                post_prop_neg = 0
            post_score = post_prop_pos - post_prop_neg
        except:
            post_score = 0

        try:
            comment_sent =  self.laker_sentiment_df[self.laker_sentiment_df['pc'] == 'c']
            comment_length = comment_sent.shape[0]
            try:
                comment_prop_pos = comment_sent.sentiment.value_counts().loc['positive'] /
comment_length
            except:
                comment_prop_pos = 0
            try:
                comment_prop_neg = comment_sent.sentiment.value_counts().loc['negative']
/ comment_length
            except:
                comment_prop_neg = 0
            comment_score = comment_prop_pos - comment_prop_neg
        except:
            comment_score = 0

        if (post_score == 0) & (comment_score == 0):
            weighted = 0
        else:
            # get weighted score
            num = (post_score * post_length * post_multiplier) + (comment_score *
comment_length)
            denom = (post_length * post_multiplier) + comment_length
            weighted = num / denom

        return weighted

    def get_trending_sentiment(self, n=20, n_comp=None):
        if type(n) != int:
            raise TypeError("Invalid Value Type. Must be int")

        sent = self.laker_sentiment_df
        date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
        date_dt_first = date_dt - dt.timedelta(days=n)
        date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
        trend = sent[sent['datetime'] > date_first]

        sent_score_last_n = ss.get_sentiment(trend)

        comp = sent[sent['datetime'] <= date_first]
        if n_comp != None:
            date_dt2 = dt.datetime.strptime(date_first, "%Y-%m-%d")
            date_dt_first2 = date_dt2 - dt.timedelta(days=n_comp)
            date_first2 = dt.datetime.strftime(date_dt_first2, "%Y-%m-%d")
            comp = comp[comp['datetime'] > date_first2]
        sent_score_comp = ss.get_sentiment(comp)
        diff = (sent_score_last_n - sent_score_comp) / sent_score_comp

        return sent_score_last_n, sent_score_comp, diff

    def get_mentions_per_day(self):
        df = self.laker_sentiment_df
        df = df[['datetime', 'sentiment']]
        idx = pd.period_range(min(df.datetime), max(df.datetime)).astype('datetime64[ns]')
        mentions = df.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0)

        return mentions.sentiment.mean()

    def get_trending_mentions_per_day(self, n=14, n_comp=None):
        if type(n) != int:
            raise TypeError("Invalid Value Type. Must be int")

        df = self.laker_sentiment_df
        df = df[['datetime', 'sentiment']]
        idx = pd.period_range(min(df.datetime), max(df.datetime)).astype('datetime64[ns]')
        mentions = df.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0)

        date_dt = dt.datetime.strptime(self.date, "%Y-%m-%d")
        date_dt_first = date_dt - dt.timedelta(days=n)
        date_first = dt.datetime.strftime(date_dt_first, "%Y-%m-%d")
        trend = mentions[mentions.index > date_first]
        trend = trend.sentiment.mean()
        comp = mentions[mentions.index <= date_first]
        if n_comp != None:
            date_dt2 = dt.datetime.strptime(date_first, "%Y-%m-%d")
            date_dt_first2 = date_dt2 - dt.timedelta(days=n_comp)
            date_first2 = dt.datetime.strftime(date_dt_first2, "%Y-%m-%d")
            comp = comp[comp.index > date_first2]

        comp = comp.sentiment.mean()

        diff = (trend - comp) / comp

        return trend, comp, diff

    def get_rolling_sentiment_score_df(self, n=20):
        sent = self.laker_sentiment_df
        cols = ['weighted']
        idx = sorted(list(set(sent.datetime)))
        dfs = pd.DataFrame(columns=cols, index=idx)

        weighted=[]
        for i in idx:
            start = i - dt.timedelta(days=n)
            s = sent[(sent['datetime'] > start) & (sent['datetime'] <= i) ]
            w = ss.get_sentiment(s)
            weighted.append(w)
        dfs['weighted'] = weighted

        return dfs

    def get_rolling_mentions_df(self, n=20):
        df = self.laker_sentiment_df
        df = df[['datetime', 'sentiment']]
        idx = pd.period_range(min(df.datetime), max(df.datetime)).astype('datetime64[ns]')
        mentions = df.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0)
        cols = ['mentions']
        dfm = pd.DataFrame(columns=cols, index=idx)

        ms = []
        for i in idx:
            start = i - dt.timedelta(days=n)
            mm = mentions[(mentions.index > start) &(mentions.index <= i)]
            m = mm.sentiment.mean()
            ms.append(m)
        dfm['mentions'] = ms
        return dfm

    def get_rolling_team_wpct_df(self, n=20):
        team = TeamDate("LAL", date=self.date)
        team_stats = team.stats
        idx = pd.period_range(min(team_stats.GAME_DATE),
max(team_stats.GAME_DATE)).astype('datetime64[ns]')
        cols = ['wpct']
        dft = pd.DataFrame(columns=cols, index=idx)

        ws = []
        for i in idx:
            start = i - dt.timedelta(days=n)
            ww = team_stats[(team_stats.GAME_DATE > start) &
(team_stats.GAME_DATE <= i)].reset_index().drop(columns=['index'])
            wins = ww.iloc[-1].W - ww.W[0]
            losses = ww.iloc[-1].L - ww.L[0]
            if wins + losses == 0:
                wpct = 0
            else:
                wpct = wins / (wins + losses)
            ws.append(wpct)
        dft['wpct'] = ws

        return dft

################################################## GET Word-Assoc
##################################################

    def get_top_players(self):
        player_ref = self.entities['full_name']
        return ss.get_top_players(player_ref)

    def get_emoji_dict(self, kind='all'):
        player_ref = self.entities['full_name']
        return ss.get_emoji_dict(player_ref, kind=kind)

    def get_token_dict(self, kind='all'):
        player_ref = self.entities['full_name']
        return ss.get_token_dict(player_ref, kind=kind)

################################################## PLOTS AND
VISUALIZATIONS ##################################################
    def plot_sentiment_through_time(self):
        post_sent = self.laker_sentiment_df[self.laker_sentiment_df['pc'] == 'p']
        comment_sent =  self.laker_sentiment_df[self.laker_sentiment_df['pc'] == 'c']

        ## multiply poss by post_mult and combine post and comment sents w dates
        posts = pd.concat([post_sent, post_sent,
post_sent]).reset_index().drop(columns=['index', 'player_ref'])
        posts = posts[['id', 'datetime', 'sentiment']]
        comments = comment_sent[['id', 'datetime', 'sentiment']]
        combined = pd.concat([posts,
comments]).set_index('datetime').sort_values('datetime')
        series = pd.Series(combined['sentiment'], index=combined.index)

        # get scores
        co_pos = series[series == 'positive']
        co_neg = series[series == 'negative']
        co_neu = series[series == 'neutral']

        # get index to fillnas
        idx = pd.period_range(min(series.index),
max(series.index)).astype('datetime64[ns]')

        # resample in 3D blocks
        pos_resamp =
co_pos.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0).resample('3D').sum()
        neg_resamp =
co_neg.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0).resample('3D').sum()
        neu_resamp =
co_neu.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0).resample('3D').sum()

        combined_resamp = pd.concat([pos_resamp, neg_resamp, neu_resamp], axis=1)
        combined_resamp.columns = ['positive', 'negative', 'neutral']

        combined_resamp['pos_ratio'] = combined_resamp.positive /
(combined_resamp.positive + combined_resamp.negative + combined_resamp.neutral)
        combined_resamp['neg_ratio'] = combined_resamp.negative /
(combined_resamp.positive + combined_resamp.negative + combined_resamp.neutral)
        combined_resamp['score'] = combined_resamp.pos_ratio -
combined_resamp.neg_ratio


        # Create the figure
        fig = make_subplots(specs=[[{"secondary_y": True}]])

        # Add the positive bars
```

```
fig.add_trace(
    go.Bar(
        x=pos_resamp.index,
        y=pos_resamp,
        name='Positive',
        marker_color='lightgreen',
    ),
    secondary_y=False,
)

# Add the negative bars
fig.add_trace(
    go.Bar(
        x=neg_resamp.index,
        y=-neg_resamp,
        name='Negative',
        marker_color='lightcoral',
    ),
    secondary_y=False,
)

fig.add_trace(
    go.Scatter(
        x=combined_resamp.index,
        y=combined_resamp['score'],
        name='Score',
        marker_color='#FDB927',
    ),
    secondary_y=True,
)

custom_template = templates["plotly_dark"]
custom_template.layout["xaxis"]["showgrid"] = False
custom_template.layout["yaxis"]["showgrid"] = False

# Configure the layout
fig.update_layout(
    title=dict(
        text='Sentiment Through Time'
    ),
    xaxis_title='Date',
    yaxis_title='Number of Impressions',
    barmode='overlay',
    bargap=0,
    barnorm=None,
    height=600,
        margin=dict(l=50, r=50, b=100, t=100, pad=4),
        template=custom_template,
        plot_bgcolor='#552583',
        paper_bgcolor='#552583',
        yaxis2=dict(showgrid=False, zeroline=False)

    )

    return fig



def generate_wordcloud(self):
    token_dict = self.token_dict
    wordcloud = WordCloud(width=1600,
                          height=400,
                          background_color='#311149',
                          color_func=random_color_func,
                          min_font_size=10).generate_from_frequencies(token_dict)
    #plt.imshow(wordcloud, interpolation='bilinear')
    #plt.axis('off')
    return wordcloud.to_image()


def basic_pie_chart(self):
    df = self.laker_sentiment_df
    sentiment_col = df.sentiment
    labels = list(sentiment_col.value_counts().index)
    values = list(sentiment_col.value_counts())
    colors = {'positive':'lightgreen',
              'negative':'lightcoral',
              'neutral':'white'}
    colors = [colors[x] for x in labels]

    fig = go.Figure(data=[go.Pie(labels=labels, values=values, showlegend=False)])

    fig.update_traces(hoverinfo='label+percent',
                      textinfo='percent+label',
                      textposition='inside',
                      textfont_size=20,
                      marker=dict(colors=colors,
                                  line=dict(color='#000000',width=2)))
    fig.update_layout(
        margin=dict(l=0, r=0, t=0, b=0),
    )
```

```
#         fig.update_layout(
#             title={
#                 'text' : "Sentiment Pie Chart".format(self.name),
#                 'x':0.5,
#                 'xanchor': 'center'
#                 },
#             font_family="Courier New",
#             title_font_size=24
 #        )

        return fig

    def plot_top_ten(self, kind='all'):
        player_ref = self.entities['full_name']
        names = self.entities['names']
        tok = ss.get_token_dict(player_ref, kind=kind)
        tok = tok[:50]

        toks = []
        counts = []
        for i, _ in enumerate(tok):
            if tok[i][0] not in names:
                toks.append(tok[i][0])
                counts.append(tok[i][1])

        toks = toks[:10]
        counts = counts[:10]
        toks.reverse()
        counts.reverse()

        fig = go.Figure(
            go.Bar(
                x=counts,
                y=toks,
                orientation='h',
                marker_color='#FDB927'
            )
        )

        fig.update_layout(
            title={
                'text': "Top 10 Frequency Terms",
                'x':0.2,
                'y':0.93
            },
            xaxis=dict(
                title="frequencies",
                showgrid=False,
                showline=False,
            ),
            yaxis=dict(
                tickmode='linear',
                showgrid=False,
                showline=False,
                automargin=True,

            ),
            plot_bgcolor='#552583',
            paper_bgcolor='#552583',
            margin=dict(l=20, r=20, t=50, b=20),
            font=dict(
                color='white',
                family='Montserrat, Helvetica, Arial, sans-serif',
            ),
        )
        return fig

    def plot_top_ten_emoji(self, kind='all'):
        player_ref = self.entities['full_name']
        emj = ss.get_emoji_dict(player_ref, kind=kind)
        emj = emj[:50]

        emjs = []
        counts = []
        for i, _ in enumerate(emj):
            emjs.append(emj[i][0])
            counts.append(emj[i][1])

        emjs = emjs[:10]
        counts = counts[:10]
        emjs.reverse()
        counts.reverse()

        fig = go.Figure(
                go.Bar(
                    x=counts,
                    y=emjs,
                    orientation='h',
                    marker_color='#FDB927'
                )
            )
        fig.update_layout(
            title={
                'text': "Top 10 Frequency Emojis",
                'x':0.2,
                'y':0.93
            },
            xaxis=dict(
                title="frequencies",
                showgrid=False,
                showline=False,
            ),
            yaxis=dict(
                tickmode='linear',
                showgrid=False,
                showline=False,
                automargin=True,
                tickfont=dict(size=20)

            ),
            plot_bgcolor='#552583',
            paper_bgcolor='#552583',
            margin=dict(l=20, r=20, t=50, b=20),
            font=dict(
                color='white',
                family='Montserrat, Helvetica, Arial, sans-serif',
            ),
        )
        return fig

    def sent_vs_stat_corr(self, stat_col, kind='g'):
        if kind not in ['g', 'm', 'p', 't']:
            raise ValueError("Invalid stat type")

        adj_cols=[
            'FGM', 'FGA', 'FG3M', 'FG3A', 'FTM', 'FTA', 'OREB', 'DREB', 'REB',
            'AST', 'STL', 'BLK', 'TO', 'PF', 'PTS', 'PLUS_MINUS',
        ]

        sent = self.get_rolling_sentiment_score_df()

        if kind == 'g':
            stats = self.get_rolling_per_game_df()
            suffix = '_PG'
        elif kind == 'm':
            stats = self.get_rolling_per_36_df()
            suffix = '_P36M'
            elif kind == 'p':
                stats = self.get_rolling_per_100_df()
                suffix = '_P100P'
            else:
                stats = self.get_rolling_team_wpct_df()
                stat_col = 'wpct'
                stats.loc['2022-10-18'] = 0
                stats.loc['2022-10-19'] = 0

            if stat_col in adj_cols:
                stat_col = stat_col + suffix

            x = stats

            if kind != 't':
                xmin = stats.GAME_DATE.min()
                xmax = stats.GAME_DATE.max()
                y = sent.loc[xmin:xmax]
                date_range = pd.date_range(start=x.GAME_DATE.min(),
    end=x.GAME_DATE.max(), freq='D')
                x = x.set_index('GAME_DATE')
                new_index = pd.Index(date_range, name='Date')
                x = x.reindex(new_index).ffill()

                corr = y['weighted'].corr(x[stat_col])
            else:
                xmin = stats.index.min()
                xmax = stats.index.max()
                y = sent.loc[xmin:xmax]
                corr = y['weighted'].corr(x[stat_col])
            return corr




        def sent_vs_stat(self, stat_col, kind='g'):
            if kind not in ['g', 'm', 'p', 't']:
                raise ValueError("Invalid stat type")

            adj_cols=[
                'FGM', 'FGA', 'FG3M', 'FG3A', 'FTM', 'FTA', 'OREB', 'DREB', 'REB',
                'AST', 'STL', 'BLK', 'TO', 'PF', 'PTS', 'PLUS_MINUS',
            ]

            ## Load all needed stats ##
            mentions = self.get_rolling_mentions_df()
```

```
team = self.get_rolling_team_wpct_df()
sent = self.get_rolling_sentiment_score_df()

if kind == 'g':
   stats = self.get_rolling_per_game_df()
   suffix = '_PG'
elif kind == 'm':
   stats = self.get_rolling_per_36_df()
   suffix = '_P36M'
elif kind == 'p':
   stats = self.get_rolling_per_100_df()
   suffix = '_P100P'
else:
   stats = self.get_rolling_team_wpct_df()
   stat_col = 'wpct'
   stats['GAME_DATE'] = stats.index

if stat_col in adj_cols:
   stat_col = stat_col + suffix


# Create the figure

fig = make_subplots(specs=[[{"secondary_y": True}]])

fig.add_trace(
   go.Scatter(
      y=sent.weighted,
      x=sent.index,
      name='Sentiment',
      line=dict(color='#FDB927')
   ),
   secondary_y=False,
)

fig.add_trace(
   go.Scatter(
      x=stats.GAME_DATE,
      y=stats[stat_col],
      name='{}'.format(stat_col),
      line=dict(color='#840DFA')
   ),
   secondary_y=True,
)

custom_template = templates["plotly_dark"]
```

```
        custom_template.layout["xaxis"]["showgrid"] = False
        custom_template.layout["yaxis"]["showgrid"] = False

        fig.update_yaxes(title_text="Rolling 20-day MA Sentiment Score",
title_font=dict(color='#FDB927'), secondary_y=False)
        fig.update_yaxes(title_text="Rolling 20-day MA {}".format(stat_col),
title_font=dict(color='#840DFA'), secondary_y=True)

        fig.update_layout(
                title='Rolling 20-day Sentiment vs {}'.format(stat_col),
                xaxis_title='Date',
                template=custom_template,
                plot_bgcolor='black',
                paper_bgcolor='black',
            )

        return fig

##############################################################################
######################
def get_rolling_team_wpct_df(n=20, date='2023-04-10'):
    team = TeamDate("LAL", date=date)
    team_stats = team.stats
    idx = pd.period_range(min(team_stats.GAME_DATE),
max(team_stats.GAME_DATE)).astype('datetime64[ns]')
    cols = ['wpct']
    dft = pd.DataFrame(columns=cols, index=idx)

    ws = []
    for i in idx:
        start = i - dt.timedelta(days=n)
        ww = team_stats[(team_stats.GAME_DATE > start) & (team_stats.GAME_DATE
<= i)].reset_index().drop(columns=['index'])
        wins = ww.iloc[-1].W - ww.W[0]
        losses = ww.iloc[-1].L - ww.L[0]
        if wins + losses == 0:
            wpct = 0
        else:
            wpct = wins / (wins + losses)
        ws.append(wpct)
    dft['wpct'] = ws

    return dft

def sent_vs_wpct_corr(player_ref, date):
    sent = ss.get_rolling_sentiment_score_df(player_ref=player_ref, date=date)
    wpct = get_rolling_team_wpct_df(date=date)
    wpct.loc['2022-10-18'] = 0
    wpct.loc['2022-10-19'] = 0

    xmin = wpct.index.min()
    xmax = wpct.index.max()
    y = sent.loc[xmin:xmax]
    corr = y['weighted'].corr(wpct['wpct'])

    return corr

def sent_vs_wpct(player_ref, date):
    sent = ss.get_rolling_sentiment_score_df(player_ref=player_ref, date=date)
    wpct = get_rolling_team_wpct_df(date=date)
    wpct.loc['2022-10-18'] = 0
    wpct.loc['2022-10-19'] = 0

    # Create the figure
    fig = make_subplots(specs=[[{"secondary_y": True}]])

    fig.add_trace(
        go.Scatter(
            y=sent.weighted,
            x=sent.index,
            name='Sentiment',
            line=dict(color='#FDB927')
        ),
        secondary_y=False,
    )

    fig.add_trace(
        go.Scatter(
            x=wpct.index,
            y=wpct['wpct'],
            name='{}'.format('wpct'),
            line=dict(color='#840DFA')
        ),
        secondary_y=True,
    )

    custom_template = templates["plotly_dark"]
    custom_template.layout["xaxis"]["showgrid"] = False
    custom_template.layout["yaxis"]["showgrid"] = False

    fig.update_yaxes(title_text="Rolling 20-day MA Sentiment Score",
title_font=dict(color='#FDB927'), secondary_y=False)
```

```
    fig.update_yaxes(title_text="Rolling 20-day MA {}".format('wpct'),
title_font=dict(color='#840DFA'), secondary_y=True)

    fig.update_layout(
            title='Rolling 20-day Sentiment vs {}'.format('wpct'),
            xaxis_title='Date',
            template=custom_template,
            plot_bgcolor='black',
            paper_bgcolor='black',
        )

    return fig
```

7) Supplementary Visualization functions

```
import plotly.graph_objects as go
import pandas as pd
import numpy as np
import datetime as dt
import random
import json
import re
import matplotlib.pyplot as plt
from plotly.subplots import make_subplots
from wordcloud import WordCloud
from plotly.io import templates


# my imports
import sentiment_stats as ss


### basic pie chart ###
def basic_pie_chart(player_ref, date):
    sent = pd.read_csv("assets/all_sents/stok.csv", index_col=0, parse_dates=['datetime'])
    sent = sent[sent['player_ref'] == player_ref]
    df = sent[sent['datetime'] <= date]
    sentiment_col = df.sentiment
    labels = list(sentiment_col.value_counts().index)
    values = list(sentiment_col.value_counts())
    colors = {'positive':'lightgreen',
              'negative':'lightcoral',
              'neutral':'white'}
    colors = [colors[x] for x in labels]

    fig = go.Figure(data=[go.Pie(labels=labels, values=values, showlegend=False)])

    fig.update_traces(hoverinfo='label+percent',
                      textinfo='percent+label',
                      textposition='inside',
                      textfont_size=20,
                      marker=dict(colors=colors,
                                line=dict(color='#000000',width=2)))
    fig.update_layout(
        margin=dict(l=0, r=0, t=0, b=0),
    )
    return fig

### trending sentiment analysis ###
def get_last_n_day_sentiment(sent_df, date='2023-04-14', n=14):
    pass


def plot_sentiment_through_time(player_ref, date):
    sent = pd.read_csv("assets/all_sents/stok.csv", index_col=0, parse_dates=['datetime'])
    sent = sent[sent['player_ref'] == player_ref]
    sent = sent[sent['datetime'] <= date]
    post_sent = sent[sent['pc'] == 'p']
    comment_sent =  sent[sent['pc'] == 'c']

    ## multiply poss by post_mult and combine post and comment sents w dates
    posts = pd.concat([post_sent, post_sent, post_sent]).reset_index().drop(columns=['index', 'player_ref'])
    posts = posts[['id', 'datetime', 'sentiment']]
    comments = comment_sent[['id', 'datetime', 'sentiment']]
    combined = pd.concat([posts, comments]).set_index('datetime').sort_values('datetime')
    series = pd.Series(combined['sentiment'], index=combined.index)

    # get scores
    co_pos = series[series == 'positive']
    co_neg = series[series == 'negative']
    co_neu = series[series == 'neutral']

    # get index to fillnas
    idx = pd.period_range(min(series.index), max(series.index)).astype('datetime64[ns]')

    # resample in 3D blocks
    pos_resamp = co_pos.groupby('datetime').count().resample('1D').sum().reindex(idx,
fill_value=0).resample('3D').sum()
```

```
    neg_resamp = co_neg.groupby('datetime').count().resample('1D').sum().reindex(idx, fill_value=0).resample('3D').sum()
    neu_resamp = co_neu.groupby('datetime').count().resample('1D').sum().reindex(idx, fill_value=0).resample('3D').sum()

    combined_resamp = pd.concat([pos_resamp, neg_resamp, neu_resamp], axis=1)
    combined_resamp.columns = ['positive', 'negative', 'neutral']

    combined_resamp['pos_ratio'] = combined_resamp.positive / (combined_resamp.positive + combined_resamp.negative + combined_resamp.neutral)
    combined_resamp['neg_ratio'] = combined_resamp.negative / (combined_resamp.positive + combined_resamp.negative + combined_resamp.neutral)
    combined_resamp['score'] = combined_resamp.pos_ratio - combined_resamp.neg_ratio


    # Create the figure
    fig = make_subplots(specs=[[{"secondary_y": True}]])

    # Add the positive bars
    fig.add_trace(
        go.Bar(
            x=pos_resamp.index,
            y=pos_resamp,
            name='Positive',
            marker_color='lightgreen',
        ),
        secondary_y=False,
    )

    # Add the negative bars
    fig.add_trace(
        go.Bar(
            x=neg_resamp.index,
            y=-neg_resamp,
            name='Negative',
            marker_color='lightcoral',
        ),
        secondary_y=False,
    )

    fig.add_trace(
        go.Scatter(
            x=combined_resamp.index,
            y=combined_resamp['score'],
            name='Score',
            marker_color='#FDB927',
        ),
        secondary_y=True,
    )

    custom_template = templates["plotly_dark"]
    custom_template.layout["xaxis"]["showgrid"] = False
    custom_template.layout["yaxis"]["showgrid"] = False

    # Configure the layout
    fig.update_layout(
        title=dict(
            text='Sentiment Through Time'
        ),
        xaxis_title='Date',
        yaxis_title='Number of Impressions',
        barmode='overlay',
        bargap=0,
        barnorm=None,
        height=600,
        margin=dict(l=50, r=50, b=100, t=100, pad=4),
        template=custom_template,
        plot_bgcolor='#552583',
        paper_bgcolor='#552583',
        yaxis2=dict(showgrid=False, zeroline=False)

    )

    return fig
```

```
### wordcloud
def random_color_func(word=None, font_size=None, position=None, orientation=None, font_path=None,
random_state=None):
    colors = ['#840DFA', '#FDB927', 'white', '#405ED7', "#FDB927"]
    return random.choice(colors)

def generate_wordcloud(token_dict):
    token_dict = dict(token_dict)
    wordcloud = WordCloud(width=1600,
                          height=400,
                          background_color='#311149',
                          color_func=random_color_func,
                          min_font_size=10).generate_from_frequencies(token_dict)
    #plt.imshow(wordcloud, interpolation='bilinear')
    #plt.axis('off')
    return wordcloud.to_image()


### Top ten words and emojis ###
def plot_top_ten(ent_key, kind='all'):
    with open('assets/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    player_ref = entities[ent_key]['full_name']
    names = entities[ent_key]['names']
    tok = ss.get_token_dict(player_ref, kind=kind)
    tok = tok[:50]

    toks = []
    counts = []
    for i, _ in enumerate(tok):
        if tok[i][0] not in names:
            toks.append(tok[i][0])
            counts.append(tok[i][1])
```

```
toks = toks[:10]
counts = counts[:10]
toks.reverse()
counts.reverse()

fig = go.Figure(
    go.Bar(
        x=counts,
        y=toks,
        orientation='h',
        marker_color='#FDB927'
    )
)

fig.update_layout(
    title={
        'text': "Top 10 Frequency Terms",
        'x':0.2,
        'y':0.93
    },
    xaxis=dict(
        title="frequencies",
        showgrid=False,
        showline=False,
    ),
    yaxis=dict(
        tickmode='linear',
        showgrid=False,
        showline=False,
        automargin=True,

    ),
    plot_bgcolor='#552583',
    paper_bgcolor='#552583',
    margin=dict(l=20, r=20, t=50, b=20),
    font=dict(
        color='white',
            family='Montserrat, Helvetica, Arial, sans-serif',
        ),
    )
    return fig


def plot_top_ten_emoji(ent_key, kind='all'):
    with open('assets/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    player_ref = entities[ent_key]['full_name']
    emj = ss.get_emoji_dict(player_ref, kind=kind)
    emj = emj[:50]

    emjs = []
    counts = []
    for i, _ in enumerate(emj):
        emjs.append(emj[i][0])
        counts.append(emj[i][1])

    emjs = emjs[:10]
    counts = counts[:10]
    emjs.reverse()
    counts.reverse()

    fig = go.Figure(
            go.Bar(
                x=counts,
                y=emjs,
                orientation='h',
                marker_color='#FDB927'
            )
        )

    fig.update_layout(
        title={
            'text': "Top 10 Frequency Emojis",
            'x':0.2,
            'y':0.93
        },
        xaxis=dict(
            title="frequencies",
            showgrid=False,
            showline=False,
        ),
        yaxis=dict(
            tickmode='linear',
            showgrid=False,
            showline=False,
            automargin=True,
            tickfont=dict(size=20)

        ),
        plot_bgcolor='#552583',
        paper_bgcolor='#552583',
        margin=dict(l=20, r=20, t=50, b=20),
        font=dict(
            color='white',
            family='Montserrat, Helvetica, Arial, sans-serif',
        ),
    )
    return fig


def plot_ytd_player_sent_ranks(date='2023-04-10'):
    names, scores = ss.get_ytd_player_sent_ranks(date=date)
    fig = go.Figure(
            go.Bar(
                x=names,
                y=scores,
                marker_color='#FDB927'
            )
        )
    fig.update_layout(
        title={
            'text': "Player YTD Sentiment",
            'x':0.2,
            'y':0.99
        },
        xaxis=dict(
            showgrid=False,
            showline=False,
        ),
        yaxis=dict(
            #tickmode='linear',
            showgrid=False,
            showline=False,
            automargin=True,
            tickfont=dict(size=20)

        ),
        plot_bgcolor='#552583',
        paper_bgcolor='#552583',
        margin=dict(l=30, r=30, t=50, b=20),
        font=dict(
            color='white',
            family='Montserrat, Helvetica, Arial, sans-serif',
        ),
    )
    return fig

def generate_player_hm():
    with open('assets/entities_with_nicknames.json', 'r') as f:
        entities = json.load(f)
    entities = dict(entities)

    df = pd.read_csv("assets/rm_hm_df.csv", index_col=0)
    df=np.round(df, 3)
    ents = list(entities.keys())[3:21]
```

```
names = [entities[x]['init_name'] for x in ents]
df.columns = ents
df.index = ents

heatmap = go.Heatmap(z=df.values,
        x=df.columns,
        y=df.index,
        texttemplate="%{z}",
        colorscale='YlGnBu',
        showscale=False,
        zmax=0.1)

# Create a Figure object and add the heatmap trace
fig = go.Figure(data=heatmap)

fig.update_layout(
    margin=dict(l=20, r=20, t=20, b=10),
    plot_bgcolor='#552583',
    paper_bgcolor='#552583',

)

fig.update_xaxes(tickfont=dict(color='white'))
fig.update_yaxes(tickfont=dict(color='white'))

return fig
```