Player tier systems analysis

Andrew Carr
Brigham Young University
Provo, Utah
andrewcarr06@gmail.com

ABSTRACT

League of Legends is an international video gaming phenomenon. The game boasts 27 million daily active users, and 67 million monthly users [1]. One aspect of the game that makes it so popular is the ranking system. This system allows players of similar skill level to be matched up against each other in the 5 vs 5 play style. In this work, I explore the relations of the over 140 different champions [2] across these ranks. Using modern machine learning methods we find that as player skill increases our ability to predict win percentage decreases. In other words, we were able to answer the question "As player skill increases, are we better able to use in game actions to predict win rate?"

ACME Reference Format:
Andrew Carr. 2018. Player tier system analysis . ACME, Provo, UT, USA, 10 pages.

1 INTRODUCTION

1.1 game overview

League of Legends (LoL) was released Oct 27, 2009 [3] and falls into the category of Multi-player Online Battle Arena (MOBA). This popular style involves two teams that work against each other to accomplish some goal. In the case of LoL, there are five players per team. One team is assigned the red side, and another the blue side.

1.2 game goal

The entire objective of the game is for one of the teams to eliminate towers (shown as dots shown in **fig 1**) along the various *lanes* and destroy the enemy base. They do this while defending their own towers and base.

There are many nuances of strategy that won't be explored here, but some background needs to be given into the mechanics of the game.

1.3 minions

Each team has a constant (unlimited) supply of non-playable *minions*. These minions, sometimes referred to as *creeps* follow fairly set paths down the various lanes and attack enemy units.

These minions give gold to a player if the player is the reduces the minion's health to zero. Gold is an important resource to buy power boosting items that can give you an edge when fighting with an opponent.

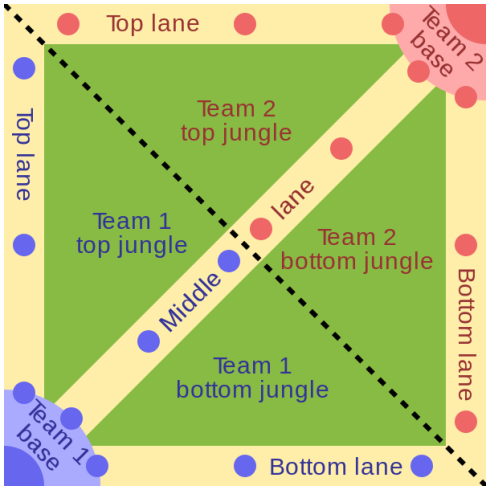

Figure 1: simplified map [4]

Figure 2: minions [6]

1.4 champions

Every player, before the match begins, selects a *champion* they will control during that game. These 140+ unique champions are the main piece of our analysis. They each have different abilities, strengths, and weaknesses.

Players choose whichever champion best fits their play style, team dynamic, and opposing team weaknesses.

Final Project, Winter 2018, ACME BYU
2018.

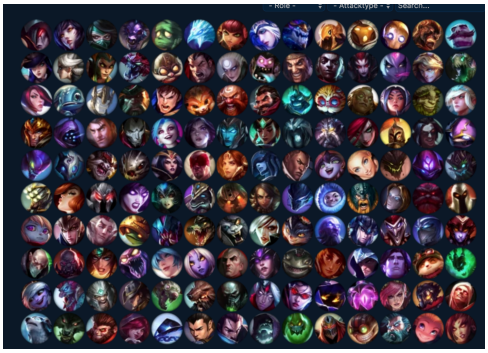

Figure 3: examples of champions [7]

Figure 4: Roles icons - top, mid, adc, sup, jungle

1.5 roles

Another very important piece of game play is the various *roles* that players fill during the game. These roles are: **Top, Mid, Attack Damage Carry (ADC), Support, and Jungle**. There can only be one of each role per team, and they all need to work together to give the best chance of winning.

1.6 player tier ranking

In the competitive play style of the game, there are a number of tiers players climb as they increase their skill. The tiers serve as a noisy approximation of player skill level. Placement games are played at the beginning of each new season, and then win percentage determines a player's rise or fall in the rankings. The rankings, for our purposes, can be listed as:

- Bronze
- Silver
- Gold
- Platinum
- Platinum+ (which includes Diamond, Master, Challenger)

2 DATA COLLECTION AND CLEANING

2.1 collection

The data was collected from http://champion.gg which is an aggregation website that pulls pseudo real time data from the League of Legends API regarding a number of in game actions. We scrapped this data, in accordance with the TOC and robots.txt, and compiled it into five separate data frames.

The data contains information on champions played at various skill levels, we can use this data to predict win percentage of a champion at certain tiers, and in certain roles.

As we can see in **fig 5** the win percentage across tier rankings is relatively consistent. An astute observer will notice that they are not all at an equal 50% win rate. This is due to the delay in data collection of http://champion.gg. Not all games are represented, and so the win rates are also a noisy approximation of the true tier win rate.

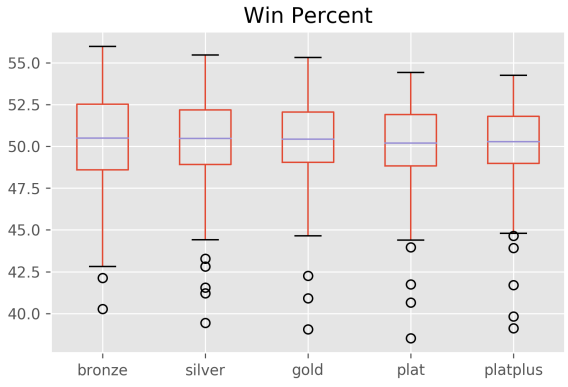

Figure 5: Exploration of win percentage over rank

Table 1: Description of data available

Column Name	Column Description
Champion	Name of the champion
Role	The role in which the champion is normally played (e.g., top). This means some champions have multiple rows in the dataset
Win Percent	% of games won
Play Percent	Measure of popularity, % of games played as this champion, out of all games played
Ban Rate	Players can ban certain champions before the game begins, this measures how often this champion is banned
Kills	Average number of enemies killed
Deaths	Average number of deaths
Assists	Average number of assists on enemy death
Largest Killing Spree	Largest consecutive number of kills
Damage Dealt	Average amount of damage dealt to enemies
Damage Taken	Average amount of total damage taken
Total Healing	Average amount of healing to self or team
Minions Killed	Average number of minions killed
Enemy Jungle CS	Average number of Jungle Monsters killed by team
Team Jungle CS	Average number of Jungle Monsters killed by enemy
Gold Earned	Average amount of gold earned in game

2.2 dataset

In **table 1**, the in game actions are described. These are the features I have to work with, and provide valuable knowledge of how players at different skill levels act in games.

2.3 feature engineering

In addition to the features of the data we have, I did some very light feature engineering and added **Role code** and **Rank** columns

which represent the numeric *role* played (e.g., 0 for ADC, 1 for Jungle) and which rank this particular data point was collected from (e.g., bronze, silver). These were useful in clustering and when data was viewed in aggregate.

See **Data Cleaning**, and **Feature Engineering** in code appendix.

3 METHODS

3.1 naive baseline

As a baseline we do a simple linear fit correlation between how many times a player dies and the win percentage. This is to strengthen our confidence that the in game actions we have do correspond to win percentage values. See **fig 6**. See `naive_baseline`.

3.2 regression

Predicting win percentage with in games actions is a straight forward regression problem. However, finding the optimal model and combination of features adds complexity that requires care to ensure things are analyzed properly. To start, first recall that regression can be thought of as function approximation

$$f(x) = y$$

where we use various methods to estimate our function f. Efficacy of the approximation can be gathered by using the r^2 value which indicates how correlated our predicted values would with the actual values based on the model we use.

I decided to compare three different regression methods.

3.2.1 ridge regression. I decided to use ridge regression as our non naive baseline because it is a commonly used method that handles ill-posed problems well. It allows for the tuning of a hyper parameter α which acts as a regularization constant to prevent overfitting.

3.2.2 gradient boosted trees. This ensemble method is the primary work horse in many data scientist's tool kits. It uses a collection of decision trees to iteratively fit itself to the data. It is a very strong method because it uses 1-D optimization during each step of the training process to decide how to split and expand.

3.2.3 Light GBM. This gradient boosting machine method, recently published by Microsoft [8]. It splits leaf wise and is designed from the ground up to support parallelization. It is shown, in some circumstances, to have higher accuracy [9] and almost always has faster training time. In our case, we care less about training time, and more about accuracy. As such, this is a great opportunity to put these claims to the test and potentially get the best results of any method we try.

We performed 3-fold cross validation over feature space, and parameter space for each of the methods below. This took 17 hours on my Macbook pro and resulted in the graph in **fig 7**. These results will be discussed in depth in the Analysis section.

3.3 clustering

In addition to my original question, I was intensely curious to see if we could also cluster the champions by their roles using the same in game action data we used to predict win percentage. As such, I used **KMeans** and **Agglomerative** clustering methods after reducing the dimension of the dataset with **PCA** for easy plotting.

3.3.1 PCA. Principle Component Analysis is the machine learning name for Singular Value Decomposition which allows you to calculate the directions of greatest variance. This means you can reduce the dimensions of a dataset while maintaining a lot of the underlying characteristics of the data.

3.3.2 KMeans. This is an extremely standard clustering method that allows you to arbitrarily choose k centers and then iteratively minimizes the distance of all points to these centers (moving the centers along the way). I choose the number of clusters as there are roles, just to see which method pulls out the most accurate clusters.

3.3.3 Agglomerative. A lesser known clustering method. It takes into account connectedness of potential clusters. While KMeans is a centroid clustering method, Agglomerative is actually a connectivity based method, which means it takes into account hierarchical relations in the data.

4 ANALYSIS

4.1 naive baseline

The results from this experiment, found in **fig 6**, are very positive. I performed simple linear regression and we can see that there is indeed a correlation between in game actions (deaths in this case) and win percentage. As such, I felt confident to continue the analysis.

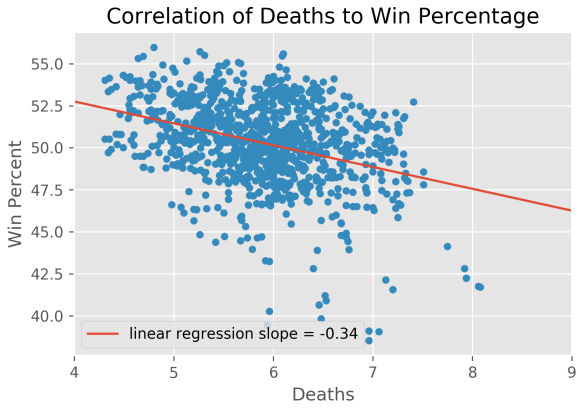

Figure 6: Results across all ranks

4.2 regression

The results in **fig 7** were quite surprising. See `grid_search`. In the x-axis, the player tier ranking is represented with r^2 values on the y-axis. What we see here is that as player skill increases, our ability to predict win percentage seems to consistently decrease.

This figure is the result of one of the thousands of experiments described in section 3.2. I chose this one in particular because it gave the best average r^2 value and was indicative of the largest majority of other results.

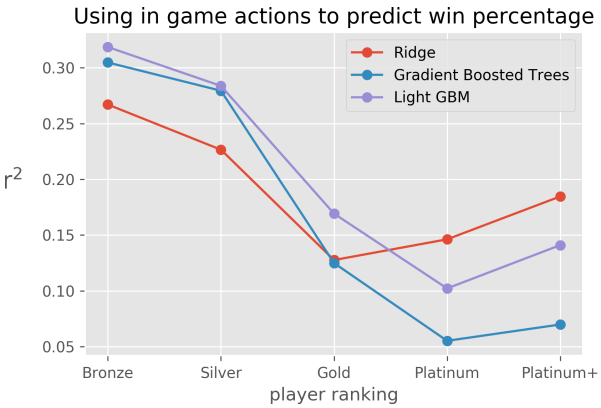

Figure 7: Cross validation results

This particular result used all available features and near default parameter values for the various regression methods.

Unsurprisingly, Light GBM lives up to the analysis on predicting bronze outcomes with the standard Gradient Boosted method a close second. However, what is truly surprising is that the baseline method, ridge regression, out performs these more complicated, and assumed better, methods when predicting platinum and platinum+ games.

When I first saw these results, I was intrigued. What about higher player skill makes these games harder to predict?

One possible explanation is that since almost 70% of players are bronze and silver they are more similar, being less skilled. However, this explanation doesn't seem to make much sense. As such, I tried exploring the data to see if there were drastically different means or standard deviations in the features. What I found was fascinating. It turns out that the standard deviation on the Kills column for gold, platinum and platinum+ ranks is almost double all the other ranks. This means that one of the most intuitively valuable features is far noisier at higher tier ranks. See `explore_variation`.

This variation causes all of our regressors to struggle, but the ridge regression's ability to handle ill posed problems shines in this case. The two ensemble methods struggle with this higher variability due to the complex nature of their loss functions.

Finally, from a non qualitative perspective, the margin of error in higher tiers might be much lower. This is a non measurable quantity that our noisy approximation of player skill does not capture. This means that a mistake in higher tier would be punished while the same action in bronze would not drastically affect the game.

4.3 clustering

See `reduce_and_cluster`

4.3.1 actual roles. Unsurprisingly we see that certain champions, when clustered by roles, often exhibit very similar in game actions. Which most roles being very similar with support roles being quite different. This clustering is unsurprising given common strategies employed by each role. Support champions typically cannot be played ("flexed") in other roles. However, Mid, Top, and ADC champions often can be swapped interchangeably between roles with little effort.

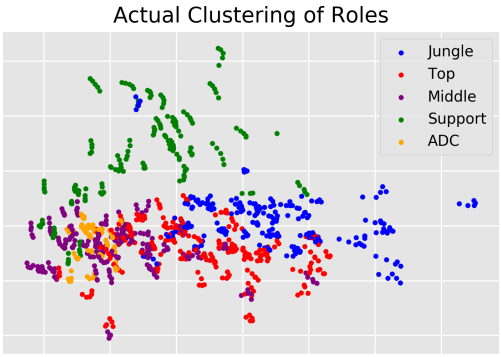

Figure 8: Actual clustering of champion roles

4.3.2 kmeans. The KMeans clustering does a very good job of separating Support and Jungle champions. However, it struggles greatly when trying to segregate Middle, Top, and ADC roles. This is because KMeans is inherently building k centroids in the data even if the data is not separable in this manner. You may notice that these clustering figures don't have legends. That is because the legend would simply read "cluster 1, cluster 2, etc." This is the nature of unsupervised methods, the shape and collection of colors is the most interesting.

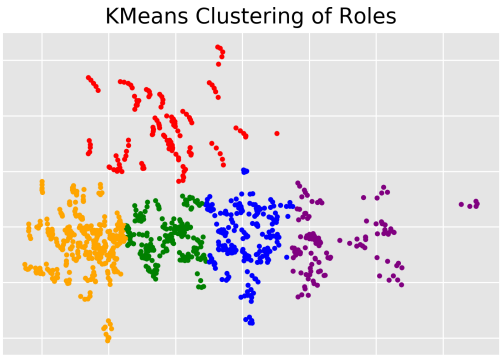

Figure 9: KMeans clustering of champion roles

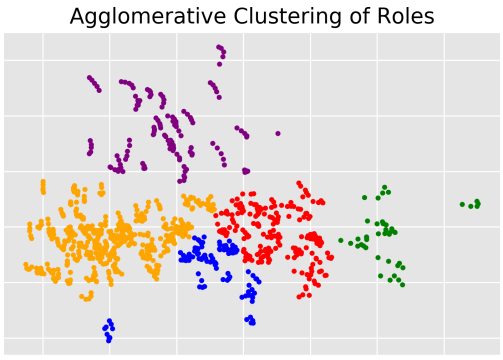

Figure 10: Aggregate clustering of champion roles, takes into account connectedness

4.3.3 agglomerative. The final clustering method performs quite well in separating Support and Jungle champions from the main group, and it is better able to identify Middle vs Top champions with a small group of ADC being picked out correctly. This is because it does take connected data into account better than centroid methods. However, it still is not a perfect clustering.

5 CONCLUSION

The game of League of Legends, created by Riot games in 2009 is an extremely popular choice of modern players. The simplicity of the goals and complex strategies it provides makes this came feel like chess with explosions. It is extremely satisfying to play. As such, there has grown to be quite a competitive scene in LoL. This competition separates players into different tiers based on their skill level.

The original question we wished to answer was "As player skill increases, are we better able to use in game actions to predict win rate?" It is very possible to predict the outcome of a game given in game actions. However, as player skill level increases, the variance of certain features increases as well. This means that regression methods perform worse at these higher tier levels.

Tangentially, I found that centroid clustering methods performed worse than hierarchical ones on determining which role a champion fills combined across all skill levels.

6 BIBLIOGRAPHY

References

[1] Purchese, Robert (January 28, 2014). "LOL: 27 million people play it every day!". Eurogamer. Gamer Network.

[2] http://leagueoflegends.wikia.com/wiki/List_of_champions

[3] "League of Legends - GameSpot.com". GameSpot.

[4] https://en.wikipedia.org/wiki/League_of_Legends#/media/File:Map_of_MOBA.svg

[5] https://en.wikipedia.org/wiki/League_of_Legends

[6] http://leagueoflegends.wikia.com/wiki/Minion?file=Minions.png

[7] screenshot from League of Legends game play client

[8] https://github.com/Microsoft/LightGBM

[9] https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/

7 CODE APPENDIX

7.1 Imports

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split, cross_validate
from sklearn.linear_model import Ridge
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from scipy.stats import linregress

from lightgbm import LGBMRegressor
from itertools import combinations
from collections import OrderedDict

from tqdm import tqdm
import pickle

plt.style.use("ggplot")
```

7.2 Data Cleaning

```
# I saved the data by hand into static html files
platplus = pd.read_html("platplus.html")[0]
plat = pd.read_html("plat.html")[0]
gold = pd.read_html("gold.html")[0]
silver = pd.read_html("silver.html")[0]
bronze = pd.read_html("bronze.html")[0]

# and combined them into a single list of data frames
data_frames = [bronze, silver, gold, plat, platplus]

# I cleaned the data
for frame in data_frames:
    for column in ['Win Percent', 'Play Percent', 'Ban Rate']:
        frame[column] = frame[column].str.replace("\%", "").astype("float")
```

7.3 Feature Engineering

```
# original full list of features
feature_list = ['Kills', 'Deaths', 'Assists', 'Largest Killing Spree', 'Damage Dealt', 'Damage Taken', '↩
    Total Healing', 'Minions Killed', 'Enemy Jungle CS', 'Team Jungle CS', 'Gold Earned']

for frame in data_frames:
    frame['Role code'] = frame['Role'].astype("category").cat.codes

# and added rank to each row, for when it was combined for clustering
bronze['rank'] = 'bronze'
silver['rank'] = 'silver'
gold['rank'] = 'gold'
plat['rank'] = 'plat'
```

```
platplus['rank'] = 'platplus'

# this displays the categorical code used in the data frames
df("Role")['Role code'].first()

# which I then saved for easy access when plotting things
role_dict = {
    0: "ADC",
    1: "Jungle",
    2: "Middle",
    3: "Support",
    4: "Top"
}
```

7.4 Regression

```
def naive_baseline(all_data):
    """calculate and plot basic regression over deaths/win percentage for all data
    parameters: all_data (pd.DataFrame) - combined data over all tier levels
    returns: no return value
    """
    slope, intercept, r, _, _ = linregress(all_data['Deaths'], all_data['Win Percent'])

    x = np.linspace(4,9,100)
    fig, ax = plt.subplots()
    all_data.plot(kind='scatter', x='Deaths', y='Win Percent',ax=ax)
    ax.plot(x, slope*x+intercept, label=r"linear regression r$^2$ = "+str(round(r,2)))
    plt.legend()
    plt.title("Correlation of Deaths to Win Percentage")
    plt.xlim((4,9))
    plt.savefig("win_death.png", dpi=300)
    plt.show()


"""In this method, I do not include the parameter search since it did not yield anything valuable and would ↩
    just needlessly increase the length of the code"""

def grid_search(feature_list):
    """search over feature space, saving and plotting results
    parameters: feature_list (list) - list of all features in feature space
    """
    reference = {}
    it = 0
    for k in range(1, len(feature_list)+1):
            # try all different sorts of features
            for features in combinations(feature_list, k):
                # save data for referencing which features were used when looking at saved images
                    reference[it] = list(features)

                    # make an array of meta data for plotting later
                    meta_data = np.zeros((5, 3))

                 # iterations to make sure we get valid results
                    for iteration in tqdm(range(10)):
                        for i, frame in enumerate(data_frames):
                            group_name = lookup[i]
```

```
                        # ridge regression
                    rid_reg = Ridge()

                        # gradient boosted trees
                        grad_reg = GradientBoostingRegressor()

                        # microsoft's new lightgbm
                    lgbm_reg = LGBMRegressor()

                        j = 0
                        for regressor in [rid_reg, grad_reg, lgbm_reg]:
                         # cross validation
                            results = cross_validate(regressor, frame[list(features)], frame['Win ↩
                                Percent'])

                            # store meta data
                            meta_data[i][j] += results['test_score'].mean()
                            j += 1

                meta_data /= 10
                print(list(features))

        # plotting
                plt.plot(meta_data[:,0], marker='o', label="Ridge")
                plt.plot(meta_data[:,1], marker='o', label="Gradient Boosted Trees")
                plt.plot(meta_data[:,2], marker='o', label="Light GBM")

        # formatting
                plt.title("Using in game actions to predict win percentage")
                plt.xticks([0,1,2,3,4], ['Bronze', 'Silver', 'Gold', 'Platinum', 'Platinum+'])
                plt.ylabel(r"r$^2$", fontsize=16, rotation=0, labelpad=20)
                plt.xlabel("player ranking")
            plt.legend()
                plt.savefig("win_percentage{}.png".format(it), dpi=300)
                plt.show()
                it += 1
    # save reference
    pickle.dump(reference,open("reference.p", 'wb'))

def explore_variation(data_frames):
    """explore and plot variation graphs
    parameters: data_frames (list) - list of pd.DataFrame objects
    """
    # used after the regression results to figure out what was going on
    for feature in ['mean', 'std', 'min', 'max']:
        for frame, name in zip(data_frames, ['Bronze', 'Silver', 'Gold', 'Platinum', 'Platinum+']):
                to_plot = frame.describe().drop(
                    ['Rank','Win Percent','Role Position', 'Position Change', 'Role code']
                    , axis=1).loc[feature]
                to_plot /= to_plot.sum()
                plt.plot(to_plot.values, label=name)
        plt.legend()
        plt.show()
```

7.5 Clustering

```
def reduce_and_cluster(Cluster, all_data, plot_title, out_filename, actual=False):
    """reduce the data, cluster, and plot the clusters
    parameters: Cluster (sklearn clustering object) - clustering method
                all_data (pd.DataFrame) - combined data across tiers
                plot_title (string) - title for the plot
                out_filename (string) - location of file for saving
                actual (bool) - boolean to determine clustering logic
    """

    # reduce the dimension
    pca = PCA(n_components=2).fit_transform(all_data[feature_list])

    # make new data structure
    df = np.zeros((1024,3))
    df[:,0] = pca[:,0]
    df[:,1] = pca[:,1]
    df[:,2] = all_data['Role code']

    # if you want to plot the actual data
    if actual:
        # get color based on role code
        colored = [colors[int(k)] for k in df[:,2]]
        for i in range(1024):
            plt.scatter(pca[i,0], pca[i,1],  color = colored[i], label=role_dict[df[i,2]], s = 8.5)

        # make a custom legend
        handles, labels = plt.gca().get_legend_handles_labels()
        by_label = OrderedDict(zip(labels, handles))
        plt.legend(by_label.values(), by_label.keys())
    else:
        # otherwise we want to cluster the data
        clusterer = Cluster(n_clusters=5).fit(pca)
        centers = clusterer.cluster_centers_
        c_preds = clusterer.predict(pca)

        # and plot the clusters
        colors = ['orange','blue','purple','green', 'red']
        colored = [colors[int(k)] for k in c_preds]
        plt.scatter(pca[:,0], pca[:,1],  color = colored, s=8.5)

    plt.title(plot_title)

    # some formatting
    plt.tick_params(
         axis='x',          # changes apply to the x-axis
            which='both',      # both major and minor ticks are affected
            bottom='off',      # ticks along the bottom edge are off
            top='off',         # ticks along the top edge are off
            \labelbottom='off') # labels along the bottom edge are off

    plt.tick_params(
        axis='y',          # changes apply to the x-axis
            which='both',      # both major and minor ticks are affected
            left='off',      # ticks along the bottom edge are off
            right='off',         # ticks along the top edge are off
```

```
            labelleft='off') # labels along the bottom edge are off

    plt.savefig(out_filename, dpi=300)
    plt.show()
```