In [ ]:
```
heavy = 'https://boxrec.com/en/ratings?division=Heavyweight&offset=0&sex=M'
cruiser = 'https://boxrec.com/en/ratings?division=Cruiserweight&offset=0&sex=M'
lightheavy = 'https://boxrec.com/en/ratings?division=Light+Heavyweight&offset=0&sex=M'
supermiddle = 'https://boxrec.com/en/ratings?division=Super+Middleweight&offset=0&sex=M'
middle = 'https://boxrec.com/en/ratings?division=Middleweight&offset=0&sex=M'
superwelter = 'https://boxrec.com/en/ratings?division=Light+Middleweight&offset=0&sex=M'
welter = 'https://boxrec.com/en/ratings?division=Welterweight&offset=0&sex=M'
superlight = 'https://boxrec.com/en/ratings?division=Light+Welterweight&offset=0&sex=M'
light = 'https://boxrec.com/en/ratings?division=Lightweight&offset=0&sex=M'
superfeather = 'https://boxrec.com/en/ratings?division=Super_Featherweight&offset=0&sex=M'
feather = 'https://boxrec.com/en/ratings?division=Featherweight&offset=0&sex=M'
superbantam = 'https://boxrec.com/en/ratings?division=Super+Bantamweight&offset=0&sex=M'
bantam = 'https://boxrec.com/en/ratings?division=Bantamweight&offset=0&sex=M'
superfly = 'https://boxrec.com/en/ratings?division=Super+Flyweight&offset=0&sex=M'
fly = 'https://boxrec.com/en/ratings?division=Flyweight&offset=0&sex=M'
lightfly = 'https://boxrec.com/en/ratings?division=Light+Flyweight&offset=0&sex=M'
minimum = 'https://boxrec.com/en/ratings?division=Minimumweight&offset=0&sex=M'

url_list = []

url_list.extend([heavy, cruiser, lightheavy, supermiddle, middle, superwelter, welter, superlight, light,
                 superfeather, feather, superbantam, bantam, superfly, fly, lightfly, minimum])
```

In [ ]:
```
#List of the URLs containing the top 50 boxers by category - manually added the values and variables. created a list to add the url's to.
```

In [ ]:
```
import os
from urllib.request import urlopen, Request
from bs4 import BeautifulSoup
import re
import csv
import boxrec_urls as bru
import time

pages = set()

def get_links(page_url):
    print("running scraper...")
    global pages
    for url in bru.url_list:
        req = Request(url, headers = {'User-Agent': 'Mozilla/5.0'})
        html = urlopen(req)
        print(f'reading {url}')
        bs = BeautifulSoup(html.read(), 'html.parser')
        for link in bs.find_all('a', href=re.compile('^(/en/box-pro/)')):
            if 'href' in link.attrs:
                if link.attrs['href'] not in pages:
                    new_page = link.attrs['href']
                    print(new_page)
                    pages.add(new_page)
                    time.sleep(2)

os.system('cls')

get_links('')

print("scraping done.")

print("now writing to csv...")

with open('boxrec_top_50_urls.csv', mode='a+', newline ='') as boxrec_file:
    boxrec_writer = csv.writer(boxrec_file, delimiter=',')
    for data in pages:
        boxrec_writer.writerows([[data]])

print("created csv.")
```

In [ ]:
```
# Imports the necessary libraries and loops through the url_list to obtain the URL of the boxers in the top 50 of each division. There are 50 displayed on each top 50 rankings page.
# Once those are obtained, we save the URL endings to a csv file line by line.
```

In [ ]:
```
from bs4 import BeautifulSoup
import re
import csv
from scrapingbee import ScrapingBeeClient

# Creating lists to store data on boxers
# key_list has some hard coded values as these require alternative methods to scrape the corresponding data
key_list = ['name', 'division rating', 'wins', 'losses', 'draws', 'KO wins', 'KO losses']
dirty_value_list = []
clean_value_list = []
# Creating the CSV column headers and dict to store key-value pair
csv_headers = ['name', 'division rating', 'division', 'bouts', 'rounds', 'KOs', 'debut', 'age', 'stance', 'height', 'reach', 'residence', 'birth place', 'wins', 'losses', 'draws', 'KO wins', 'KO losses']
csv_dict = {}


def rotate_boxer_urls():
    """
    Function uses the scrapingbee library to evade bans on boxrec.com, API is stored on a txt file and gitignored.
    boxrec_urls.py was used to obtain the boxrec URLs of the top 50 boxers in each weight category (16), totalling 800.
    NOTE: Boxrec updates boxer's ratings actively and so it might be more accurate to timestamp the data.

    rotate_boxer_urls gets a BeautifulSoup response of the url before calling grab_boxer_data(response)
    The data obtained from the page is then cleaned via clean_lists() and clean_dict()
    Lastly the data which with a correctly corresponding KEY in the dictionairy is appended to a CSV file
    with write_to_csv()
    The rotate_boxer_url then continues to loop through each URL in the boxrec_top_50_urls until complete.
    """
    global key_list
    global dirty_value_list
    global clean_value_list

    # Obtaining API key
    with open('api_key.txt', 'r') as key_file:
        API_KEY = key_file.read()

    # Obtaining list of URLs to be scraped
    with open('boxrec_top_50_urls.csv', 'r') as top_50_urls_csv:
        reader = csv.reader(top_50_urls_csv)
        top_50_urls_list = []
        for row in reader:
            top_50_urls_list.append(', '.join(row))

    # Creating request to scrape each URL
    for url in top_50_urls_list:
        boxrec_url = 'https://boxrec.com' + url
        api_key = API_KEY
        PARAMS = {"render_js": 'False'}

        client = ScrapingBeeClient(api_key=api_key)

        # Catching errors and printing
        try:
            response = client.get(boxrec_url, params=PARAMS )
            response.raise_for_status()
        except Exception as err:
            print(f'Other error occurred: {err}')

        else:
            response.encoding = 'utf-8'
            # Scrape the page!
```

```
            bs = BeautifulSoup(response.text, 'lxml')

            grab_boxer_data(bs)

            clean_lists(dirty_value_list, clean_value_list)

            clean_dict(csv_dict)

            write_to_csv(csv_headers, csv_dict)

            # Reseting list values
            key_list = ['name', 'division rating', 'wins', 'losses', 'draws', 'KO wins', 'KO losses']
            dirty_value_list = []
            clean_value_list = []

            # Below check to see which url was scraped before moving to the next url
            print(boxrec_url)
            print('------------------------------------------------')


def grab_boxer_data(soup):
    """
    Gathers the required data from a boxer's boxrec profile page.
    Access variable describes whether the response was successful, if not closes the script.
    Due to the the profile page design, items have to be individually searched for in many cases.
    Name, rating, wins/losses/draws/ko wins/ ko losses, all require searching for individually.
    Reach is also individually obtained as it doesn't match the same structure as similar elements on the page
    The remaining values can be programmatically obtained for key-value pairs.
    """
    boxrec_tables = soup.find_all('td', {'class': 'rowLabel'})
    access = len(boxrec_tables)

    if access == 0:
        print('7.. 8.. 9.. 10! BOXREC KO!')
        print('Closing scripts...')
        exit()
    else:
        print(f'SCRAPING...')

    boxer_name = soup.find('h1').get_text()
    dirty_value_list.append(boxer_name)

    ratings = []
    for link in soup.find_all('a', href=re.compile('^(/en/ratings?)')):
        ratings.append(link.get_text())

    try:
        no_slash_n = [whitespace.replace('\n', '') for whitespace in ratings]
        no_space = [whitespace.replace(' ', '') for whitespace in no_slash_n]
        rating = no_space[1]
        division_rating = rating[1:3]
        dirty_value_list.append(division_rating)

    # Some boxers in the Top 50 can suddenly become "inactive" and hence have their rating removed
    # This is a catch for that
    except Exception:
        dirty_value_list.append('')

    wins = soup.find('td', {'class': 'bgW'}).get_text()
    dirty_value_list.append(wins)
    losses = soup.find('td', {'class': 'bgL'}).get_text()
    dirty_value_list.append(losses)
    draws = soup.find('td', {'class': 'bgD'}).get_text()
    dirty_value_list.append(draws)
    ko_wins = soup.find('th', {'class': 'textWon'}).get_text()
    dirty_value_list.append(ko_wins[:-4])
    ko_losses = soup.find('th', {'class': 'textLost'}).get_text()
    dirty_value_list.append(ko_losses[:-4])

    # Obtain key-value pairs for boxer data
    for table_value in boxrec_tables:
        first_td = table_value.find_all('b')
        for item in first_td:
            key = item.get_text()
            value = item.find_next().get_text()
            key_list.append(key)
            dirty_value_list.append(value)

            # Obtain data on boxers reach if it exists
            if item.get_text() == 'reach':
                value_reach = item.find_next().find_next().find_next()
                boxer_reach = value_reach.get_text()
                key_list.append(key)
                dirty_value_list.append(boxer_reach)


def clean_lists(dirty_list, clean_list):
    """
    Initial cleaning of the raw data from the scraping
    Target list matches the intended CSV headers
    Not all boxers will have the data on their page and so if the key doesn't exist, we create it,
    before inserting an empty '' value to the corresponding index in the values list.
    """
    global csv_headers
    global csv_dict

    print('CLEANING THE DATA...')
    # Cleans the data removing unwanted string values
    no_pct = [pct.replace('%', '') for pct in dirty_list]
    no_slash_n = [whitespace.replace('\n', '') for whitespace in no_pct]
    no_fwd_slash = [fwd_slash.replace('/', '') for fwd_slash in no_slash_n]
    no_space = [whitespace.replace(' ', '') for whitespace in no_fwd_slash]
    no_xa0 = [xa0.replace('\xa0', '') for xa0 in no_space]
    no_comma = [comma.replace(',', ' ') for comma in no_xa0]
    for value in no_comma:
        clean_list.append(value)

    target_list = ['name','division rating','division','bouts','rounds','KOs','debut','age','stance','height','reach','residence','birth place', 'wins', 'losses', 'draws', 'KO wins', 'KO losses']

    for key in target_list:
        if key in key_list:
            pass
        else:
            key_list.insert(target_list.index(key), key)
            clean_list.insert(target_list.index(key), '')
    # Creates a dictionary with the key list and cleaned values
    # This is all the data before selecting those that we want to send to the CSV
    boxer_dict = dict(zip(key_list, clean_value_list))

    # Takes only those values for the CSV as per csv_headers
    csv_values = [boxer_dict[header] for header in csv_headers]
    # Creates final dictionary ready for writing to the CSV
    csv_dict = dict(zip(csv_headers, csv_values))
    print('CREATED CSV DICTIONARY')


def clean_dict(csv_dict=csv_dict):
    """
    Cleans the values for respective keys.
    Perhaps need a specific fix instead of replacing all missing values with "NO DATA"
    This has caused an issue where columns filled with integers or dates get typed as strings
```

```
    """
    for item in csv_dict:
        if item == 'height':
            if csv_dict[item] != '':
                csv_dict[item] = csv_dict[item][-5:-2]
        elif item == 'reach':
            if csv_dict[item] != '':
                csv_dict[item] = csv_dict[item][-5:-2]
        elif item == 'name':
                split_name = re.findall('[A-Z][a-z]+', csv_dict[item])
                csv_dict[item] = ' '.join(split_name)
        elif item == 'residence':
                split_name = re.findall('[A-Z][a-z]+', csv_dict[item])
                csv_dict[item] = ' '.join(split_name)
        elif item == 'birth place':
                split_name = re.findall('[A-Z][a-z]+', csv_dict[item])
                csv_dict[item] = ' '.join(split_name)
        if csv_dict[item] == '':
            csv_dict[item] = 'None'

    print('DATA CLEANED, READY FOR CSV')


def write_to_csv(csv_header=csv_headers, csv_dict=csv_dict):
    """Simply appends the data to the boxrec_tables.csv file"""
    with open('boxrec_tables.csv', 'a', encoding="utf-8", newline='') as boxrec_csv:
        writer = csv.DictWriter(boxrec_csv, fieldnames=csv_headers)
        print('WRITING TO CSV...')
        writer.writerow(csv_dict)


# Run the script!
rotate_boxer_urls()
```

Functions have definitions - this cell obtains the first data set. NOTE - I am using the scrapingbee API as it provides a safe and fast way to scrape data without being rate blocked by websites. At the time of scraping (19/10/2022), boxrec does not ban bots from scraping its website as verified in the robots.txt file. This may change in the future and so the script should be used with caution. Further, when pushing the folder to the github repository, I am ignoring all CSV files which contain the data, so that the data is not released to the public for extra security.

In [ ]:
```
import requests
import re
from bs4 import BeautifulSoup
import pprint
import csv
from scrapingbee import ScrapingBeeClient
import pandas as pd


def rotate_boxer_urls():
    # Obtaining API key
    """
    Rotating urls in the url_list to be scraped via the scrapingbee API
    pushes the beautifulsoup response.text in lxml format to the check_can_scrape method
    """
    with open('api_key.txt', 'r') as key_file:
        API_KEY = key_file.read()

    # Obtaining list of URLs to be scraped
    with open('boxrec_top_50_urls.csv', 'r') as top_50_urls_csv:
        reader = csv.reader(top_50_urls_csv)
        top_50_urls_list = []
        for row in reader:
            top_50_urls_list.append(', '.join(row))

    # Creating request to scrape each URL
    for url in top_50_urls_list:
        boxrec_url = 'https://boxrec.com' + url
        api_key = API_KEY
        PARAMS = {"render_js": 'False'}

        client = ScrapingBeeClient(api_key=api_key)

        # Catching errors and printing
        try:
            response = client.get(boxrec_url, params=PARAMS )
            response.raise_for_status()
        except Exception as err:
            print(f'Other error occurred: {err}')
        else:
            response.encoding = 'utf-8'
            # Scrape the page!
            bs = BeautifulSoup(response.text, 'lxml')

            check_can_scrape(bs)

            # Below check to see which url was scraped before moving to the next url
            print(boxrec_url)
            print('-----------------------------------------------')

def launch_soup():
    """
    UNUSED FUNCTION WAS DESIGNED FOR TESTING ONLY
    """
    headers = {'User-Agent': 'Mozilla/5.0'}
    boxrec_url = 'https://boxrec.com/en/box-pro/15243'
    try:
        response = requests.get(boxrec_url, headers=headers )
        response.raise_for_status()
    except Exception as err:
        print(f'Other error occurred: {err}')
    else:
        response.encoding = 'utf-8'
        # Scrape the page!
        bs = BeautifulSoup(response.text, 'lxml')
        check_can_scrape(bs)

def check_can_scrape(soup):
    """
    takes the response and checks to see if we have access, if so pushes the soup to grab the data.
    """
    boxrec_tables = soup.find_all('td', {'class': 'rowLabel'})
    access = len(boxrec_tables)

    if access == 0:
        print('7.. 8.. 9.. 10! BOXREC KO!')
        print('Closing scripts...')
        exit()
    else:
        print(f'SCRAPING...')
        grab_table_data(soup)

def clean_name(name):
    """"
    same function used previously to clean the name
    """
    no_pct = name.replace('%', '')
    no_slash_n = no_pct.replace('\n', '')
    no_fwd_slash = no_slash_n.replace('/', '')
    no_space = no_fwd_slash.replace(' ', '')
    no_xa0 = no_space.replace('\xa0', '')
    no_comma = no_xa0.replace(',', ' ')
```

```
    split_name = re.findall('[A-Z][a-z]+', no_comma)
    cleaned_name = ' '.join(split_name)
    return cleaned_name

boxer_name_list = []
def grab_table_data(soup):
    """
    uses the soup to get the boxer name (which is a H1 tag)
    obtains a data on the date of a bout, the opponent url on that date,
    the opponents name, their win/loss/draw record, and the bout outcome.
    These are saved to respective lists.
    The data is then put together into a dictionary ready for further cleaning.
    """
    global boxer_name_list
    boxer_name = clean_name(soup.find('h1').get_text()) # GET BOXER NAME
    boxer_name_list.append(boxer_name)

    date_list = []
    opp_url_list = []
    opp_name_list = []
    opp_wins_list = []
    opp_losses_list = []
    opp_draws_list = []
    bout_result_list = []

    bout_hist_table = soup.find_all('table', {'class': 'dataTable'})
    for table_value in bout_hist_table:
        date_text = table_value.find_all('a', href=re.compile('^(/en/date?)'))
        for dates in date_text:
            date = dates.get_text() # GET ALL BOUT DATES
            date_list.append(date)

        a_tag_text = table_value.find_all('a', href=re.compile('^(/en/box-pro)'))
        for string in a_tag_text:
            grab_url = re.search('href="(.*)">', str(string))
            opp_url = grab_url.group(1) # GET OPPONENT BOXREC URL
            opp_url_list.append(opp_url)
            try:
                grab_name = re.search('>(.*)</a>', str(string))
                dirty_opp_name = grab_name.group(1)  # GET OPPONENT NAME
                opp_name = clean_name(dirty_opp_name)
                opp_name_list.append(opp_name)
            except AttributeError:
                dirty_opp_name = string.get_text() # GET OPPONENT NAME
                opp_name = clean_name(dirty_opp_name)
                opp_name_list.append(opp_name)

        text_won = table_value.find_all('span', {'class': 'textWon'})
        for text in text_won:
            opp_wins = text.get_text() # GET OPPONENT WINS
            opp_wins_list.append(opp_wins)

        text_lost = table_value.find_all('span', {'class': 'textLost'})
        for text in text_lost:
            opp_losses = text.get_text() # GET OPPONENT LOSSES
            opp_losses_list.append(opp_losses)

        text_draw = table_value.find_all('span', {'class': 'textDraw'})
        for text in text_draw:
            opp_draws = text.get_text() # GET OPPONENT DRAWS
            opp_draws_list.append(opp_draws)

        text_debut = table_value.select('span[style="font-weight:bold;color:grey;"]')
        for text in text_debut:
            debut_text = text.get_text()
            if debut_text == 'debut': # DEBUT BOOLEAN CHECK AND SET
                opp_debut = True
            else:
                opp_debut = False

        text_bout_result = table_value.select('div[class*="boutResult"]')
        for text in text_bout_result:
            bout_result = text.get_text() # GET BOUT RESULT
            bout_result_list.append(bout_result)

    number_of_bouts = list(range(len(date_text), 0, -1)) # GET NUMBER OF BOUTS FOUGHT BY BOXER BEING SCRAPED

    zipped_data = zip(number_of_bouts, date_list, opp_url_list, opp_name_list, opp_wins_list, opp_losses_list, opp_draws_list, bout_result_list)
    boxrec_dict(boxer_name, zipped_data)


bout_data_dict_list = []
def boxrec_dict(name, zipped_data):
    """
    creates a list of dicts of the data obtained through scraping
    before pushing through to create a merged dictionary
    """
    global bout_data_dict_list
    bout_data_dict = {
        name: ({
            bn: {'bout_date': bd, 'opp_url': ou, 'opp_name': on, 'opp_wins': ow, 'opp_losses': ol, 'opp_draws': od, 'bout_result': br}
            for bn, bd, ou, on, ow, ol, od, br in zipped_data
            })
    }

    bout_data_dict_list.append(bout_data_dict)
    #print(bout_data_dict_list)

    #print('writing to csv')
    #write_to_csv(bout_data_dict_list)
    create_merged_dict(bout_data_dict_list, final_dict)

final_dict = {}

def create_merged_dict(bout_list, main_dict):
    for boxer_dict in bout_list:
        main_dict = {**main_dict, **boxer_dict}

    create_dict_DF(main_dict)


def create_dict_DF(main_dict):
    """
    create a dataframe from the dictionary,
    this is done so that we can essentially have the csv file multi-indexed based on name and the bout number"""
    x = pd.DataFrame.from_dict({(i,j): main_dict[i][j]
                                for i in main_dict.keys()
                                for j in main_dict[i].keys()},
                           orient='index')

    x.to_csv('bout_data_two.csv', encoding='utf-8', mode='w', header=False, index=True,)


def write_to_csv(list_of_dict):
    """writing it all to a csv..."""
    with open('bout_data.csv','w', encoding="utf-8", newline='') as bout_data_csv:
        writer = csv.DictWriter(bout_data_csv, fieldnames=boxer_name_list)
        writer.writeheader()
        writer.writerows(list_of_dict)
```

```
# RUNNING THE SCRIPT
rotate_boxer_urls() # PROXY
# launch_soup()   # NO PROXY
```

Functions have descriptions, the script to obtain the second data set.

In [ ]:
```
import pandas as pd
import numpy as np

boxrec_data = pd.read_csv('boxrec_tables.csv')
df = pd.DataFrame(boxrec_data)
```

Importing the main libraries, pandas and numpy. Creating a variable to read the csv file containing the boxer data and creating a data frame called df with that csv file.

In [ ]:
```
df.replace(to_replace='None', value=np.nan, regex=True, inplace=True)

df.insert(3, 'div index', '')

div_list = ['heavy', 'cruiser', 'lightheavy', 'supermiddle', 'middle', 'superwelter', 'welter', 'superlight', 'light', 'feather', 'superbantam', 'bantam', 'superfly', 'fly', 'lightfly', 'minimum']

x = 0
for value in df['division']:
    df.at[x, 'div index'] = div_list.index(value)
    x += 1
```

The dataframe has strings named "None" which we are replacing with numpy.nan values. We insert a new column called `div index` and write a list of all the divisions in our csv file. From this we create a for loop to fill the `div index` column with the index value (an integer) of the division within the div_list list.

In [ ]:
```
ds = df.sort_values(by=['div index', 'division rating'], ascending=True)
# Can see age/height/reach need to be turned to float64s
cols_to_convert = ['age', 'height', 'reach']

for col in cols_to_convert:
    ds[col] = ds[col].astype(float)
```

We then sort by div index in ascending order, and assign several columns to convert data types. A for loop is used to turn the values in those columns to float datatypes.

In [ ]:
```
dropped_missing_height_ds = ds.dropna(subset=['height'])
#Dropping missing height values gives 695 rows
dropped_age_ds = dropped_missing_height_ds.dropna(subset=['age'])
#Dropping the missing age values from the new dataframe leaves 688 rows of data.
new_ds = dropped_age_ds
```

We begin looking at feature engineering. Here we decide to drop height values as these are going to be hard to impute based off the existing data. This leaves us with 695 rows of data. From that we decide to remove those remaining rows where the age value is missing. This new dataframe is saved as new_ds.

In [ ]:
```
def replace_nan_with_mean(dataframe, column, grouping_column):
    """replaces not a number/null values in a column with the mean value rounded to two decimal places within the column, grouped by the grouping column"""
    dataframe[column] = dataframe.groupby(grouping_column)[column].transform(lambda x: x.fillna(round(x.mean(), 2)))
```

A function designed to replace nan values with the mean of the column grouped by the grouping column. The idea is to use fill the missing values with the mean of the boxer's respective division.

In [ ]:
```
diff_reach = new_ds.reach - new_ds.height
new_ds.insert(12, 'diff reach', diff_reach)

replace_nan_with_mean(new_ds, 'diff reach', 'division')
```

```
C:\Users\smbal\AppData\Local\Temp\ipykernel_15596\3461136964.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  dataframe[column] = dataframe.groupby(grouping_column)[column].transform(lambda x: x.fillna(round(x.mean(), 2)))
```

Here we create a new variable diff_reach which records the difference between the existing reach and height values, before creating a new column named `diff reach`. Then we use the replace_nan_with_mean function with the new_ds dataframe and the respective column and grouping column to get the desired result

In [ ]:
```
new_ds.reach.isnull().sum()
```

Out[ ]:
```
279
```

Here we identify that there are 279 rows with null data in the reach column (by summing the count of the isnull boolean values within the column)

In [ ]:
```
new_list = new_ds.reach.isnull().index
reach_min = new_ds.reach.min()
# creating reach values for those which are missing my adding height to the new diff_reach values
for x in new_list:
    if new_ds.reach[x] >= reach_min:
        pass
    else:
        new_ds.reach[x] = new_ds.height[x] + new_ds['diff reach'][x]
```

```
C:\Users\smbal\AppData\Local\Temp\ipykernel_15596\2549277302.py:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  new_ds.reach[x] = new_ds.height[x] + new_ds['diff reach'][x]
```

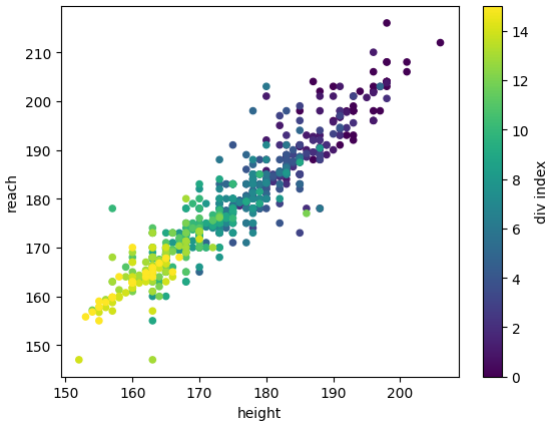
Here we save the index of those values to a list called new_list, and identify the lowest value in the reach column with new_ds.reach.min(). We then go through each value in the new_list, passing it into new_ds.reach[] and checking if it is greater than the reach_min variable. Null values would all be below the minumum, and so only on those values do we apply the formula to obtain new reach values.

In [ ]:
```
import matplotlib.pyplot as plt
```

importing that matplotlib.pyplot library as plt so that we can begin visualisations

In [ ]:
```
new_ds.plot.scatter(x='height', y='reach', c='div index', colormap='viridis')
```

Out[ ]:
```
<AxesSubplot: xlabel='height', ylabel='reach'>
```



Exploratory scatterplot, a clear and logical relationship between height and reach - the heavier the division (darker colour with div index = 0) the greater the boxer's height and reach. There are some obvious outliers in the data too.

In [ ]:
```
average_everything_numeric = new_ds.mean(numeric_only=True).round(2)
median_everything_numeric = new_ds.median(numeric_only=True).round(2)

average_by_division = new_ds.groupby(['div index']).mean(numeric_only=True).round(2)
median_by_division = new_ds.groupby(['div index']).median(numeric_only=True).round(2)
```

Obtaining some basic statistics of the existing data frame through the use of the mean() and median() methods, ensuring that only numeric values are passed through and that the output is rounded to two decimal places. The X_by_division variables obtain the same statistics except filtered for division.

In [ ]:
```
average_everything_numeric
```

Out[ ]:
```
division rating     24.66
bouts               23.48
rounds             129.72
KOs                 55.04
age                 29.92
height             174.78
reach              178.25
diff reach           3.48
wins                20.80
losses               2.14
draws                0.47
KO wins             12.82
KO losses            0.71
dtype: float64
```

In [ ]:
```
median_everything_numeric
```

Out[ ]:
```
division rating     24.00
bouts               22.00
rounds             113.00
KOs                 56.25
age                 29.50
height             175.00
reach              178.00
diff reach           3.26
wins                19.00
losses               1.00
draws                0.00
KO wins             12.00
KO losses            0.00
dtype: float64
```

In [ ]:
```
average_by_division
```

Out[ ]:

div index	division rating	bouts	rounds	KOs	age	height	reach	diff reach	wins	losses	draws	KO wins	KO losses
0	26.36	23.47	111.19	69.30	33.21	193.04	199.01	5.97	21.28	1.77	0.28	15.81	0.85
1	24.00	24.64	129.90	62.04	32.79	187.05	192.78	5.73	21.98	2.31	0.29	14.88	0.88
2	24.47	21.51	110.98	58.31	31.42	184.86	187.39	2.53	19.47	1.77	0.26	12.44	0.93
3	25.29	24.54	136.42	58.51	30.40	182.38	185.64	3.26	21.69	2.27	0.52	13.90	0.73
4	24.60	24.35	130.56	58.92	30.81	181.05	184.65	3.60	21.95	1.79	0.56	14.21	0.72
5	25.92	25.96	150.67	55.39	30.52	177.81	181.42	3.61	23.06	2.27	0.56	13.94	0.71
6	25.12	22.29	122.04	55.35	30.69	177.23	180.50	3.27	20.25	1.54	0.40	12.25	0.52
7	24.76	23.35	123.43	59.41	29.52	175.43	178.66	3.22	21.50	1.50	0.28	13.61	0.33
8	24.41	26.17	155.20	55.07	29.80	172.30	174.67	2.37	23.70	1.91	0.43	13.91	0.70
9	24.96	23.64	135.30	49.19	27.79	170.23	172.67	2.43	21.36	1.79	0.43	11.55	0.62
10	23.74	22.90	132.95	48.00	28.81	168.33	173.00	4.67	20.33	2.00	0.52	11.05	0.48
11	23.35	23.95	126.65	55.74	28.88	166.20	169.77	3.57	21.20	2.17	0.42	13.70	0.65
12	25.08	27.79	165.13	49.67	29.97	165.28	168.45	3.17	23.21	3.51	1.03	14.18	0.77
13	25.59	19.18	102.82	50.43	28.03	163.54	166.86	3.32	16.85	1.90	0.36	10.31	0.56
14	22.58	20.42	116.31	48.29	27.78	161.50	163.23	1.73	16.25	3.28	0.75	9.17	1.17
15	23.26	19.88	119.91	40.49	26.82	159.82	162.62	2.80	16.21	3.09	0.53	8.09	0.94

Decreasing average age clearly seen as the weight classes get lower in weight. Interestingly only the penultimate weight category has a KO loss value above 1.

In [ ]:
```
median_by_division
```

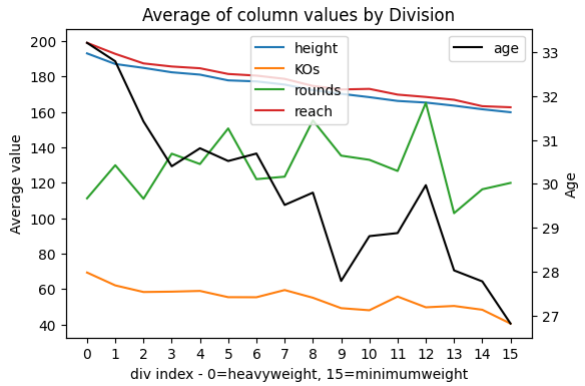
Out[ ]:

div index	division rating	bouts	rounds	KOs	age	height	reach	diff reach	wins	losses	draws	KO wins	KO losses
0	26.0	22.0	101.0	66.67	33.0	193.0	198.00	5.97	20.0	1.0	0.0	14.0	0.0
1	23.5	22.5	106.0	61.16	32.0	186.5	191.50	5.73	20.5	2.0	0.0	13.0	0.0
2	25.0	20.0	101.0	57.14	31.0	185.0	187.53	2.53	18.0	1.0	0.0	12.0	0.0
3	25.5	22.0	112.5	58.89	30.0	182.5	185.00	3.26	20.5	1.0	0.0	13.0	0.0
4	25.0	23.0	124.0	62.96	30.0	182.0	185.00	3.60	21.0	2.0	0.0	13.0	0.0
5	25.5	24.0	133.0	59.07	31.0	178.0	181.61	3.61	20.5	2.0	0.0	13.0	0.0
6	24.5	21.0	113.5	54.78	30.5	177.0	179.27	3.27	19.5	1.0	0.0	11.0	0.0
7	24.5	23.0	115.5	58.11	29.0	175.0	179.00	3.22	21.5	1.0	0.0	13.0	0.0
8	24.5	25.0	145.5	56.46	29.0	173.0	174.68	2.37	24.0	1.5	0.0	12.5	0.0
9	25.0	22.0	132.0	53.57	28.0	170.0	172.43	2.43	19.0	1.0	0.0	9.0	0.0
10	22.5	21.0	128.5	50.00	28.0	168.0	173.00	4.67	20.0	1.0	0.0	10.0	0.0
11	21.5	22.0	106.0	57.14	29.0	165.0	169.57	3.57	19.5	2.0	0.0	12.5	0.0
12	25.0	26.0	146.0	51.16	30.0	165.0	169.00	3.17	22.0	3.0	1.0	11.0	0.0
13	26.0	18.0	98.0	54.55	27.0	163.0	167.32	3.32	15.0	2.0	0.0	9.0	0.0
14	20.0	19.0	98.0	45.92	27.0	162.5	164.00	1.73	14.5	1.5	0.0	8.5	1.0
15	20.5	19.0	103.5	39.54	26.0	160.0	162.80	2.80	15.5	3.0	0.0	9.0	0.0

In [ ]:
```
fig, ax = plt.subplots()

width = 6
height = 4
fig.set_size_inches(width, height)
ax.plot(average_by_division['height'], label='height')
ax.plot(average_by_division['KOs'], label='KOs')
ax.plot(average_by_division['rounds'], label='rounds')
ax.plot(average_by_division['reach'], label='reach')

ax2 = ax.twinx()
ax2.plot(average_by_division['age'], label='age', color='black')
ax2.set_ylabel('Age')

ax.legend(loc=9)
ax2.legend(loc=0)
```

```
ax.set_xticks(np.arange(0, 16, 1))
ax.set_title('Average of column values by Division')
ax.set_xlabel('div index - 0=heavyweight, 15=minimumweight')
ax.set_ylabel('Average value')
```

Out[ ]: Text(0, 0.5, 'Average value')

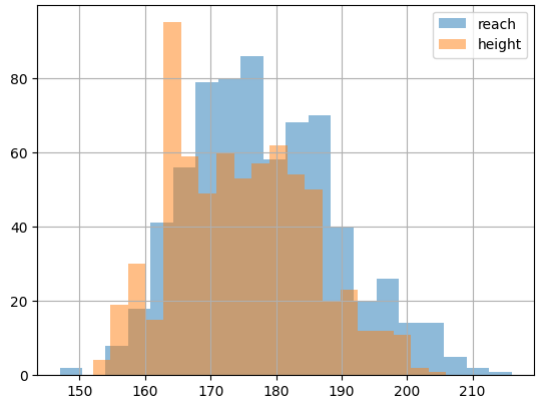


We use matplotlib to plot the line graphs with a secondary axis, all on a division basis.

Line graph here shows the similar relationship between height and reach having a linear relationship to a boxer's weight division. The black line represents Age on the second-y axis, and shows an interesting relationship where the boxer's in heavier weight classes are on average older than those in lighter weight classes. This makes sense as lighter divisions rely more on athleticism than raw power. This is somewhat seen in the fact that KO's are downwards trending on a scale of heavy to light divisions in the above graph.

In [ ]:
```
new_ds['reach'].hist(bins=20, alpha=0.5, legend=True)
new_ds['height'].hist(bins=20, alpha=0.5, legend=True)
```

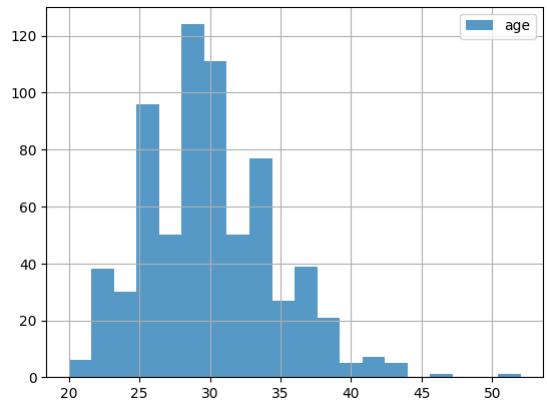
Out[ ]: <AxesSubplot: >



We can see a relationship where a boxer's height generally tends to be greater than their height.

In [ ]:
```
new_ds['age'].hist(bins=20, alpha=0.75, legend=True, )
```

Out[ ]: <AxesSubplot: >



Boxer age is clearly right-skewed, which again makes logical sense as boxing is a highly anaerobic sport, meaning successful practioners are likely to be fighting in and around their prime-years of between 25-30 years of age, which can also be seen in the data.

In [ ]:
```
#CLEANING REMAINING NULL VALUES
column_list = new_ds.columns

for x in column_list:
    if new_ds[x].isnull().sum() != 0:
        print(f'{x} null values: {new_ds[x].isnull().sum()}')
        # stance null values: 26
        # birth place null values: 28

stance_null = new_ds['stance'].isnull().index
birth_null = new_ds['birth place'].isnull().index
# birthplace nulls have no stance value, as there is no overlap and this isn't something that can be imputed, we are removing all nulls.
```

```
stance null values: 26
birth place null values: 28
```

The data contained more nulls which we counted with the for loop. The rows for which these values existed were the saved to the respective variable in preparation for removal.

In [ ]:
```
# removing stance and birthplace null values... will first remove stance nulls as they are more important, and then remove birth place nulls.
dropping_stance_ds = new_ds.dropna(subset=['stance'])
clean_ds = dropping_stance_ds.dropna(subset=['birth place'])
# running null check
for x in column_list:
    if clean_ds[x].isnull().sum() != 0:
        print(f'{x} null values: {clean_ds[x].isnull().sum()}')
```

We drop first the stance nulls and saving the output to a new dataframe, before dropping the remaining birth place nulls, some of which overlapped with the stance nulls. We then count the nulls remaining in the `clean_ds` and nothing is outputted, indicating no nulls present in the dataframe.

In [ ]:
```
from datetime import datetime

dates = clean_ds.debut.index
for x in dates:
    clean_ds.debut[x] = datetime.strptime(clean_ds.debut[x], '%d/%m/%Y').date()

clean_ds.debut = pd.to_datetime(clean_ds.debut)
```

```
C:\Users\smbal\AppData\Local\Temp\ipykernel_15596\4188815707.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  clean_ds.debut[x] = datetime.strptime(clean_ds.debut[x], '%d/%m/%Y').date()
C:\Users\smbal\AppData\Local\Temp\ipykernel_15596\4188815707.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  clean_ds.debut = pd.to_datetime(clean_ds.debut)
```

We have a column labelled debut in the dataframe but as we saved it to a csv and ingested it again into the dataframe, the resulting column was filled with string values. Here we use the datetime.strptime() function to convert the value to a date datatype.

In [ ]:
```
clean_ds.dtypes
```

Out[ ]:
```
name                      object
division rating            int64
division                  object
div index                 object
bouts                      int64
rounds                     int64
KOs                      float64
debut             datetime64[ns]
age                      float64
stance                    object
height                   float64
reach                    float64
diff reach               float64
residence                 object
birth place               object
wins                       int64
losses                     int64
draws                      int64
KO wins                    int64
KO losses                  int64
dtype: object
```

Checking it worked

In [ ]:
```
end_date = pd.to_datetime('2022/12/06')
days_active = end_date - clean_ds.debut
```

I decided it would be easier to use a time since debut value measured in the number of days since a boxer's debut for future analysis. I set a date for when I conducted the scraping of the data as the end_date, and then calculated the difference, saving the output to the days_active variable.

In [ ]:
```
clean_ds.insert(8, 'days active', days_active)
clean_ds['days active'] = clean_ds['days active'].astype(str)
clean_ds['days active'] = clean_ds['days active'].map(lambda x: x.rstrip(' days'))
clean_ds['days active'] = clean_ds['days active'].astype(int)
```

```
C:\Users\smbal\AppData\Local\Temp\ipykernel_15596\2307992778.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  clean_ds['days active'] = clean_ds['days active'].astype(str)
C:\Users\smbal\AppData\Local\Temp\ipykernel_15596\2307992778.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  clean_ds['days active'] = clean_ds['days active'].map(lambda x: x.rstrip(' days'))
C:\Users\smbal\AppData\Local\Temp\ipykernel_15596\2307992778.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  clean_ds['days active'] = clean_ds['days active'].astype(int)
```

Unfortunately this resulted in output such as "3923 days". This required some further wrangling to first convert the column to strings, then strip the `days` from the column, leaving only a number string which can be converted to an integer.

In [ ]:
```
bout_data = pd.read_csv('bout_data.csv')
bd_df = df = pd.DataFrame(bout_data)

clean_name_list = clean_ds['name'].to_list()

print(len(clean_name_list))
```

```
637
```

Importing the second dataset into the notebook. The structure of the second notebook was difficult to work with, and so I decided to simplify the project at this point. Here, I created a list of all the name values in the clean_ds dataframe which was from the first dataset.

In [ ]:
```
bd_df.columns
```

Out[ ]:
```
Index(['name', 'bout_num', 'bout_date', 'opp_url', 'opp_name', 'opp_wins',
       'opp_losses', 'opp_draws', 'bout_result'],
      dtype='object')
```

Checking the columns in the new dataframe of the second dataset

In [ ]:
```
filtered_df_x = bd_df[bd_df['name'].isin(clean_name_list)]

grouped_df_x = filtered_df_x.groupby('name')[['opp_wins', 'opp_losses', 'opp_draws']].sum()

name_counts = filtered_df_x['name'].value_counts()

grouped_df_x = grouped_df_x.assign(w_minus_l = grouped_df_x['opp_wins'] - grouped_df_x['opp_losses'])
grouped_df_x = grouped_df_x.assign(opp_win_pct = round(((2 * grouped_df_x['opp_wins'] + grouped_df_x['opp_draws']) / (2 * (grouped_df_x['opp_wins'] + grouped_df_x['opp_losses'] + grouped_df_x['opp_draws']
```

Creating a new dataframe with rows where the `name` in the bd_df are in the clean_name_list. I then group those values by name and sum all the wins, losses, and draws. I could have used the data in this dataset more thoroughly but as previously stated I found difficulties manipulating the dataframe and so opted for a more simplified version, hence summing the columns with the same name value. We then create some new columns `w_minus_l` which is a simple wins minus losses value, and an `opp_win_pct` which is essentially a scoring mechanism to identify the value of the boxer's opponents. A value closer to 1 would indicate the boxer has fought opponents with many more wins than losses, while a value closer to 0 would indicate they have fought opponents with significantly more losses than wins. We predict that most boxers would have values above 0.5, presumably around 0.6-0.8, with those with higher divisional rankings (i.e ranked 1-10) having the highest opp_win_pct values.

The formula is broken down as: opp_win_pct = (2 x oW + oD) / 2 * (oW + oL + oD)

In [ ]:
```
clean_ds = clean_ds.merge(grouped_df_x, left_on='name', right_on='name')
```

Merging the two dataframes on the name!

In [ ]:
```
preproc_ds = clean_ds
```

creating a new dataframe in order to preserve the clean, merged, dataframe.

In [ ]:
```
preproc_ds
```

Out[ ]:

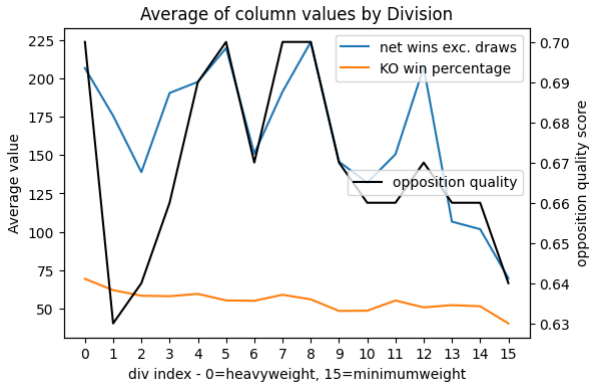
	name	division rating	division	div index	bouts	rounds	KOs	debut	days active	age	...	wins	losses	draws	KO wins	KO losses	opp_wins	opp_losses	opp_draws	w_minus_l	opp_win_pct
0	Oleksandr Usyk	1	heavy	0	20	168	65.00	2013-12-14	3279	35.0	...	20	0	0	13	0	473	79	9	394	0.85
1	Anthony Joshua	2	heavy	0	27	136	81.48	2013-10-26	3328	33.0	...	24	3	0	22	1	672	141	11	531	0.82
2	Andy Ruiz	4	heavy	0	37	187	59.46	2009-06-26	4911	33.0	...	35	2	0	22	0	576	168	33	408	0.76
3	Dillian Whyte	5	heavy	0	31	170	61.29	2011-09-16	4099	35.0	...	28	3	0	19	3	567	291	35	276	0.65
4	Joe Joyce	6	heavy	0	15	74	93.33	2018-02-16	1754	37.0	...	15	0	0	14	0	340	55	6	285	0.86
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
632	Yuni Takada	43	minimum	15	20	104	20.00	2015-08-08	2677	24.0	...	9	8	3	4	4	126	50	4	76	0.71
633	Tatsuro Nakashima	46	minimum	15	15	70	46.67	2015-11-15	2578	28.0	...	11	3	1	7	2	76	50	8	26	0.60
634	Yuri Kanaya	49	minimum	15	3	17	0.00	2021-10-20	412	26.0	...	3	0	0	0	0	27	12	1	15	0.69
635	Sora Takeda	50	minimum	15	10	43	10.00	2017-08-27	1927	22.0	...	6	4	0	1	2	37	10	1	27	0.78
636	Cris Ganoza	51	minimum	15	24	125	37.50	2014-01-25	3237	28.0	...	19	5	0	9	3	157	296	33	-139	0.36

637 rows × 26 columns

The merged dataframe.

In [ ]:
```
grouped_preproc = preproc_ds.groupby(['div index']).mean(numeric_only=True).round(2)

fig, ax = plt.subplots()

width = 6
height = 4
fig.set_size_inches(width, height)

ax.plot(grouped_preproc['w_minus_l'], label='net wins exc. draws')
ax.plot(grouped_preproc['KOs'], label='KO win percentage')

ax2 = ax.twinx()
ax2.plot(grouped_preproc['opp_win_pct'], label='opposition quality', color='black')
ax2.set_ylabel('opposition quality score')

ax.legend(loc=1)
ax2.legend(loc=5)
ax.set_xticks(np.arange(0, 16, 1))
ax.set_title('Average of column values by Division')
ax.set_xlabel('div index - 0=heavyweight, 15=minimumweight')
ax.set_ylabel('Average value')
```

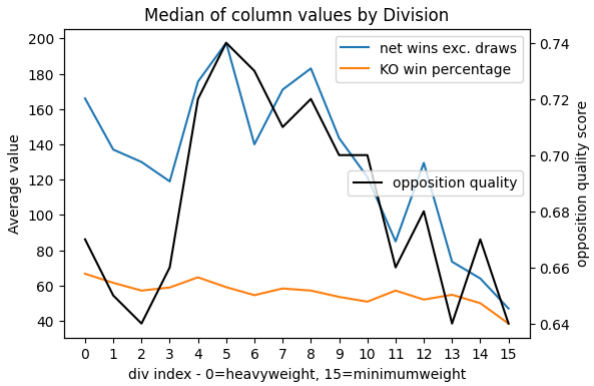
Out[ ]:
```
Text(0, 0.5, 'Average value')
```



Visualising the dataframe to identify trends, using the matplotlib library. We can see interestingly that cruiserweights have a low average opponent quality score while heavyweights have a higher quality score. As the weight classes go lower in weight towards minimumweight, quality scores drop, as do knockout win percentages.

In [ ]:
```
grouped_preproc = preproc_ds.groupby(['div index']).median(numeric_only=True).round(2)

fig, ax = plt.subplots()

width = 6
height = 4
fig.set_size_inches(width, height)

ax.plot(grouped_preproc['w_minus_l'], label='net wins exc. draws')
ax.plot(grouped_preproc['KOs'], label='KO win percentage')

ax2 = ax.twinx()
ax2.plot(grouped_preproc['opp_win_pct'], label='opposition quality', color='black')
ax2.set_ylabel('opposition quality score')

ax.legend(loc=1)
ax2.legend(loc=5)
ax.set_xticks(np.arange(0, 16, 1))
ax.set_title('Median of column values by Division')
ax.set_xlabel('div index - 0=heavyweight, 15=minimumweight')
ax.set_ylabel('Average value')
```

Out[ ]:
```
Text(0, 0.5, 'Average value')
```

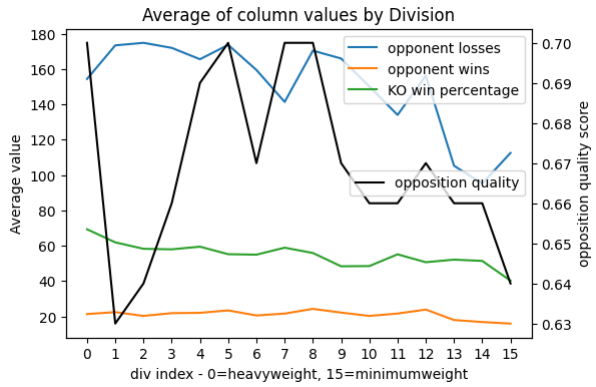


Same graph except using the median instead of the mean. It now looks as though most heavyweights actually have a much lower quality score, while middleweights maintained their strength, indicating that the middleweight division is generally quite competitive. This in some way makes logical sense as the middleweight would include males of close to average height and weight, and so offer a greater pool of boxers and likely increase competition. Quality scores

in the lower weight divisions remain relatively poor.

In [ ]:
```
grouped_preproc = preproc_ds.groupby(['div index']).mean(numeric_only=True).round(2)

fig, ax = plt.subplots()

width = 6
height = 4
fig.set_size_inches(width, height)

ax.plot(grouped_preproc['opp_losses'], label='opponent losses')
ax.plot(grouped_preproc['wins'], label='opponent wins')
ax.plot(grouped_preproc['KOs'], label='KO win percentage')

ax2 = ax.twinx()
ax2.plot(grouped_preproc['opp_win_pct'], label='opposition quality', color='black')
ax2.set_ylabel('opposition quality score')

ax.legend(loc=1)
ax2.legend(loc=5)
ax.set_xticks(np.arange(0, 16, 1))
ax.set_title('Average of column values by Division')
ax.set_xlabel('div index - 0=heavyweight, 15=minimumweight')
ax.set_ylabel('Average value')
```

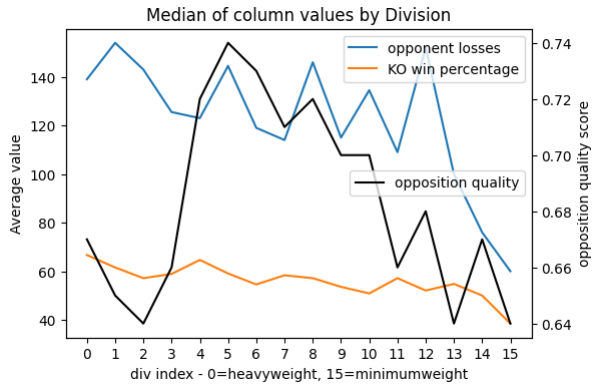
Out[ ]:
```
Text(0, 0.5, 'Average value')
```



Similar graph using average opponent losses and wins instead of a net value.

In [ ]:
```
grouped_preproc = preproc_ds.groupby(['div index']).median(numeric_only=True).round(2)

fig, ax = plt.subplots()

width = 6
height = 4
fig.set_size_inches(width, height)

ax.plot(grouped_preproc['opp_losses'], label='opponent losses')
ax.plot(grouped_preproc['KOs'], label='KO win percentage')

ax2 = ax.twinx()
ax2.plot(grouped_preproc['opp_win_pct'], label='opposition quality', color='black')
ax2.set_ylabel('opposition quality score')

ax.legend(loc=1)
ax2.legend(loc=5)
ax.set_xticks(np.arange(0, 16, 1))
ax.set_title('Median of column values by Division')
ax.set_xlabel('div index - 0=heavyweight, 15=minimumweight')
ax.set_ylabel('Average value')
```

Out[ ]:
```
Text(0, 0.5, 'Average value')
```



Similar graph using median opponent losses and wins instead of a net value.

In [ ]:
```
height_ds = preproc_ds[['division rating', 'div index', 'height', 'reach', 'wins']]

height_ds
```

Out[ ]:

	division rating	div index	height	reach	wins
0	1	0	191.0	198.0	20
1	2	0	198.0	208.0	24
2	4	0	188.0	188.0	35
3	5	0	193.0	198.0	28
4	6	0	198.0	203.0	15
...	...	...	...	...	...
632	43	15	156.0	158.8	9
633	46	15	165.0	167.8	11
634	49	15	155.0	157.8	3
635	50	15	162.0	164.8	6
636	51	15	165.0	167.8	19

637 rows × 5 columns

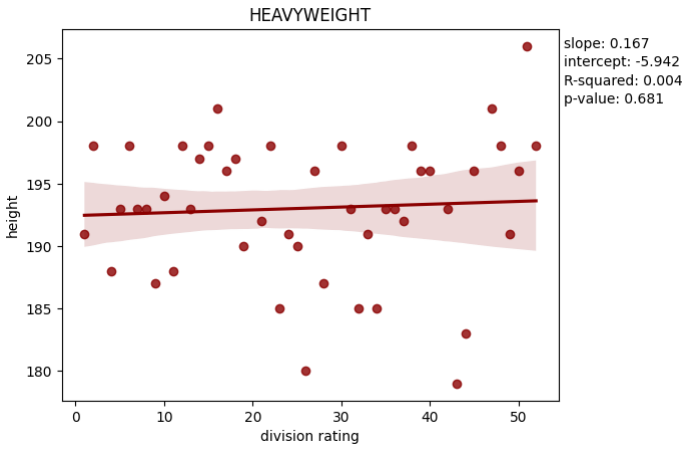
New dataframe called height_ds with several important statistics as we prepare to answer question relating to height and division rating

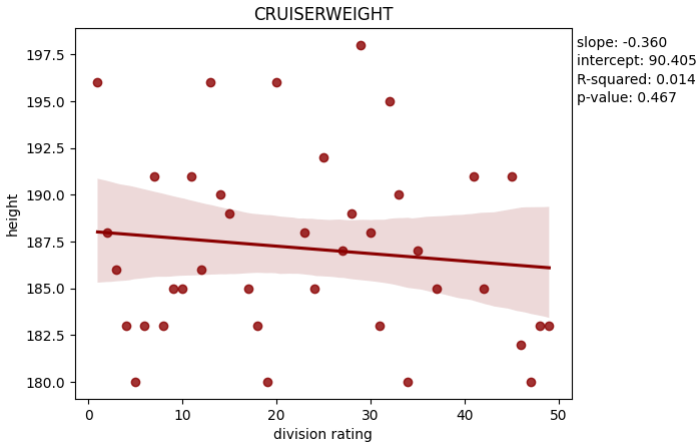
In [ ]:
```
from scipy import stats

# Calculate the t-test to determine whether there is a significant difference in the mean values of height and reach between boxers with higher and lower rankings
ranking_ttest = pd.DataFrame(columns=['t-value', 'p-value'])
for col in ['height', 'reach']:
    t_value, p_value = stats.ttest_ind(height_ds[height_ds['division rating'] > height_ds['division rating'].mean()][col], height_ds[height_ds['division rating'] <= height_ds['division rating'].mean()][co
    ranking_ttest.loc[col] = [t_value, p_value]

# Calculate the t-test to determine whether there is a significant difference in the mean values of height and reach between boxers with more and fewer wins
wins_ttest = pd.DataFrame(columns=['t-value', 'p-value'])
for col in ['height', 'reach']:
    t_value, p_value = stats.ttest_ind(height_ds[height_ds['wins'] > height_ds['wins'].mean()][col], height_ds[height_ds['wins'] <= height_ds['wins'].mean()][col])
    wins_ttest.loc[col] = [t_value, p_value]

print(ranking_ttest)
print(wins_ttest)
```

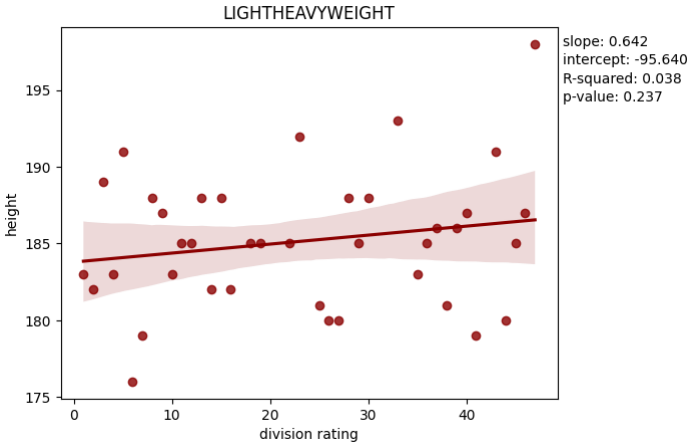
```
         t-value   p-value
height  1.558832  0.119534
reach   1.251941  0.211052
         t-value   p-value
height  0.800662  0.423627
reach   1.182535  0.237436
```

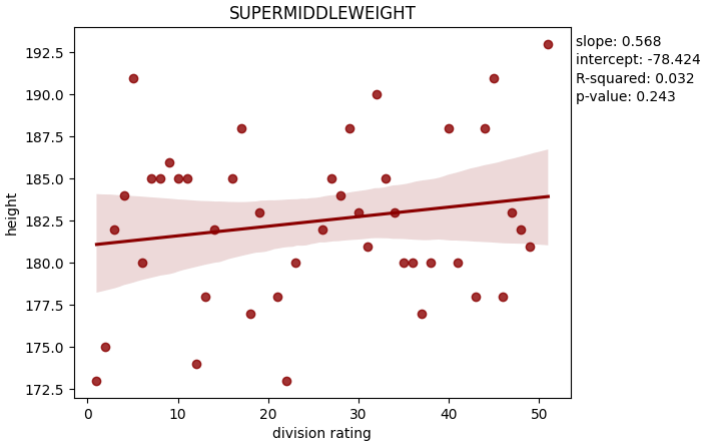
We use the stats module of the scipy library to identify whether there are statistically significant differences between height and reach of those boxers with higher and lower ranking and wins. The low p-values imply that these are not so important within the data.

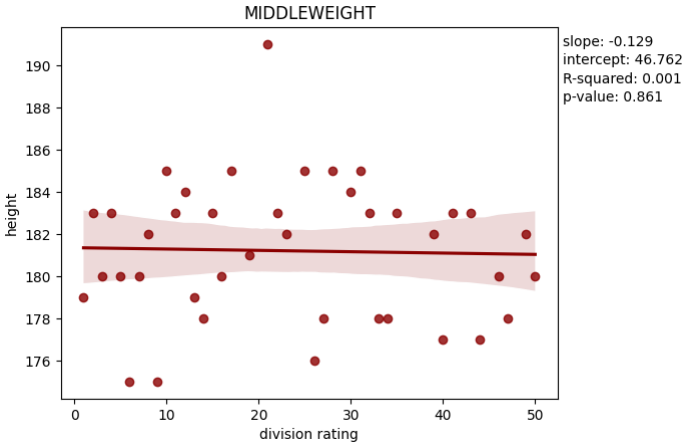
In [ ]:
```
import seaborn as sns

# Group the data by the div index column
groups = height_ds.groupby('div index')

# Iterate over the groups and create a separate plot for each group
for name, group in groups:
    # Fit a linear regression model to the data
    slope, intercept, r_value, p_value, std_err = stats.linregress(group['height'], group['division rating'])

    # Plot the regression line on top of a scatterplot of the data
    ax = sns.regplot(x='division rating', y='height', color='darkred', data=group)

    # Add text labels to the plot indicating the slope, intercept, R-squared value, and p-value for the regression
    ax.text(1.01, 0.95, f'slope: {slope:.3f}', transform=ax.transAxes)
    ax.text(1.01, 0.90, f'intercept: {intercept:.3f}', transform=ax.transAxes)
    ax.text(1.01, 0.85, f'R-squared: {r_value**2:.3f}', transform=ax.transAxes)
    ax.text(1.01, 0.80, f'p-value: {p_value:.3f}', transform=ax.transAxes)

    plt.title(f'{div_list[name].upper()}WEIGHT')
    plt.show()
```

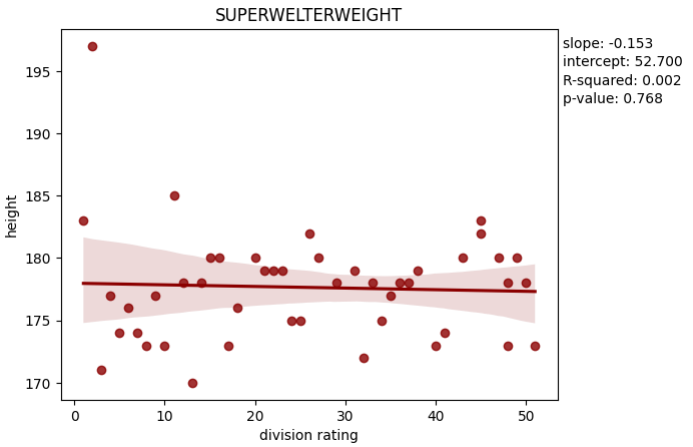




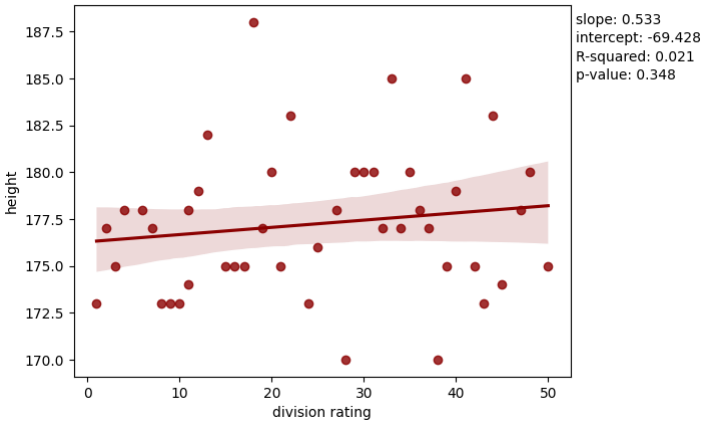

LIGHTHEAVYWEIGHT
slope: 0.642
intercept: -95.640
R-squared: 0.038
p-value: 0.237
height
division rating


SUPERMIDDLEWEIGHT
slope: 0.568
intercept: -78.424
R-squared: 0.032
p-value: 0.243
height
division rating

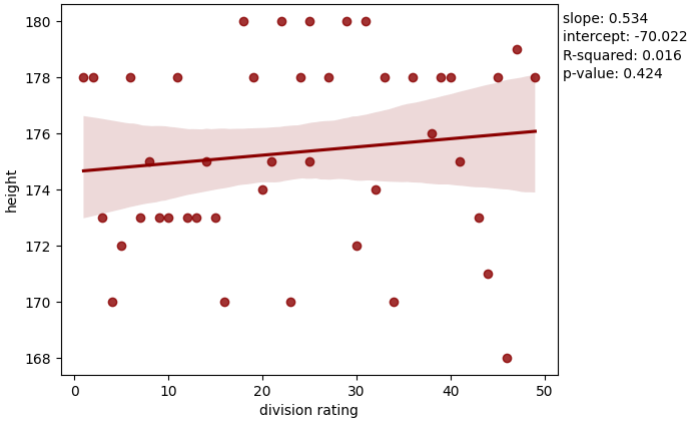

MIDDLEWEIGHT
slope: -0.129
intercept: 46.762
R-squared: 0.001
p-value: 0.861
height
division rating


SUPERWELTERWEIGHT
slope: -0.153
intercept: 52.700
R-squared: 0.002
p-value: 0.768
height
division rating

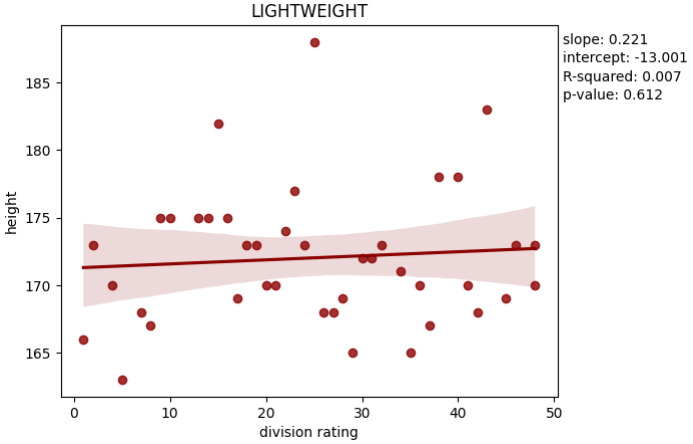
WELTERWEIGHT



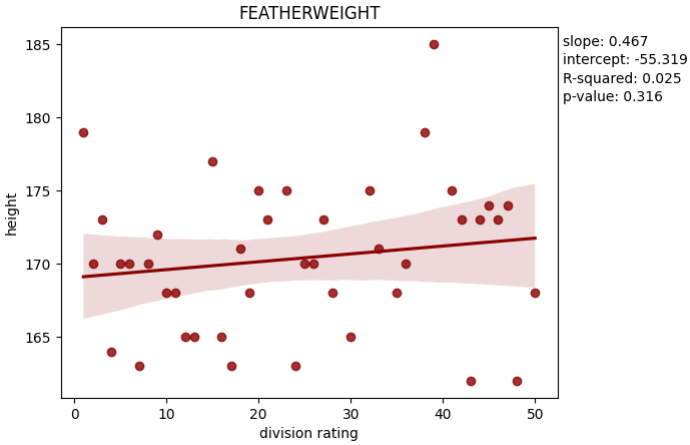
SUPERLIGHTWEIGHT



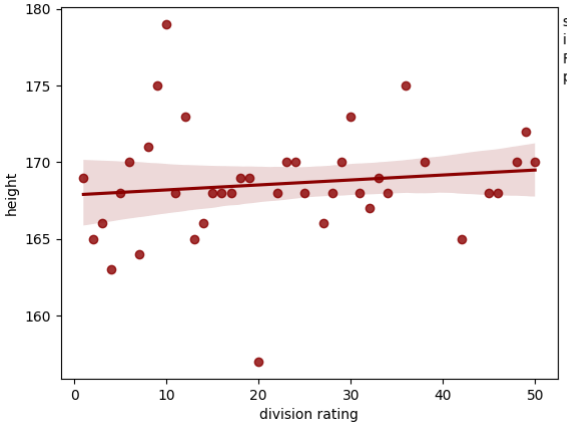
LIGHTWEIGHT



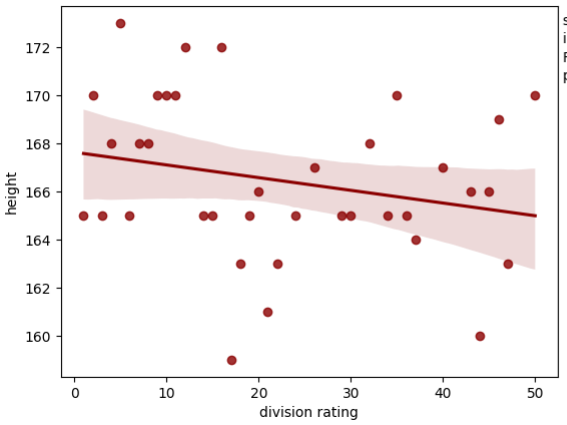
FEATHERWEIGHT



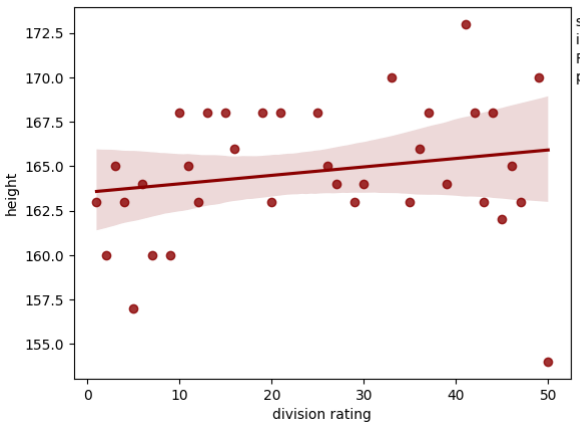
SUPERBANTAMWEIGHT



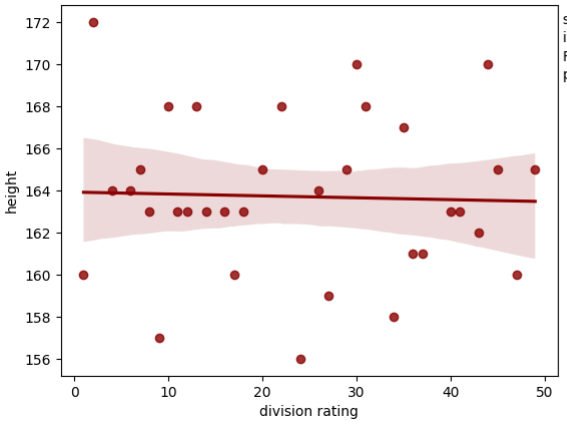
BANTAMWEIGHT

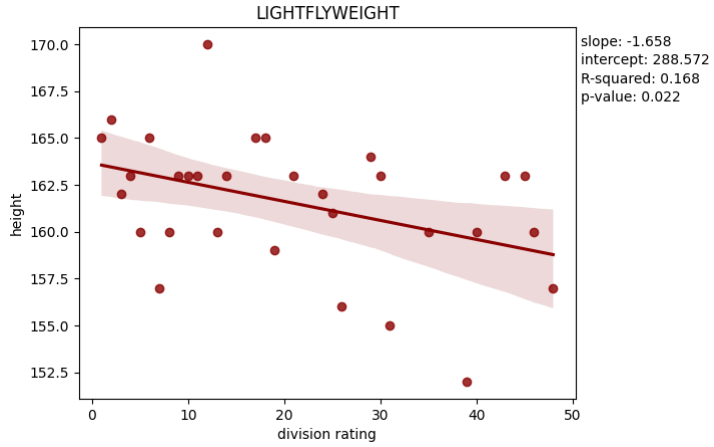


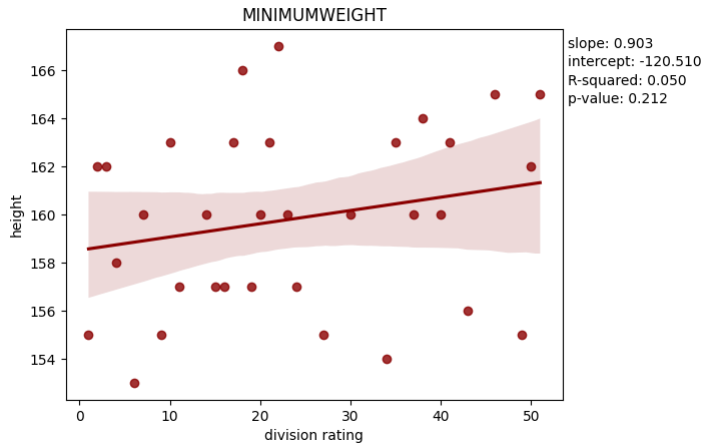
SUPERFLYWEIGHT



FLYWEIGHT







We use the seaborn library to plot some linear regressions conducted with the stats module to get an understanding of how height relates to divisional rating. We can see that in most cases there is little correlation between height and a fighter's division rating that would support the hypothesis of the wider boxing communtiy that good "bigger" boxers beat good "smaller fighters". Only in the lightflyweight division do we see a low p-value and a slope indicating that height in this division may be a factor towards a better divisional ranking

In [ ]:
```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()

#perform label encoding on 'stance' column
preproc_ds.stance= label_encoder.fit_transform(preproc_ds.stance)
```

Importing the LabelEncoder module from sklearn.preprocessing - this is to label encode the stance column which has only "orthodox" or "southpaw" values, where "southpaw" will be 1 and "orthodox" will be 0. Most boxers we expect to have a 0 value.

In [ ]:
```
preproc_ds = preproc_ds.drop('name', axis=1)
preproc_ds = preproc_ds.drop('division', axis=1)
preproc_ds = preproc_ds.drop('diff reach', axis=1)
preproc_ds = preproc_ds.drop('residence', axis=1)
preproc_ds = preproc_ds.drop('birth place', axis=1)
preproc_ds = preproc_ds.drop('debut', axis=1)
preproc_ds['div index'] = preproc_ds['div index'].astype(int)
```

Dropping columns which are no longer going to be needed in future analysis, such as name, division (which has been replaced with div index), diff reach, and debut (replaced with days active). We could have potentially used the residence or birthplace data to visualise where boxers in the top 50 per division are, and this may have led to interesting analysis. It would have likely shown that Asian and south American boxers dominate the lower weight categories, where as north American and European/CIS countries dominate the higher weight categories.

In [ ]:
```
preproc_ds.dtypes
```

Out[ ]:
```
division rating      int64
div index            int32
bouts                int64
rounds               int64
KOs                float64
days active          int32
age                float64
stance               int32
height             float64
reach              float64
wins                 int64
losses               int64
draws                int64
KO wins              int64
KO losses            int64
opp_wins             int64
opp_losses           int64
opp_draws            int64
w_minus_l            int64
opp_win_pct        float64
dtype: object
```

type checking, ensuring all are integers or floats

In [ ]:
```
six_ds = preproc_ds[['height', 'reach', 'age', 'days active', 'div index', 'division rating']]
```

New dataframe preparing for Kmeans clustering

In [ ]:
```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

Importing the necessary modules from sklean. Silhouette score will be used to identify the relevant K value to use prior to clustering. I opted for this method instead of the Elbow method to identify the optimal K value for its simplicity and ease.

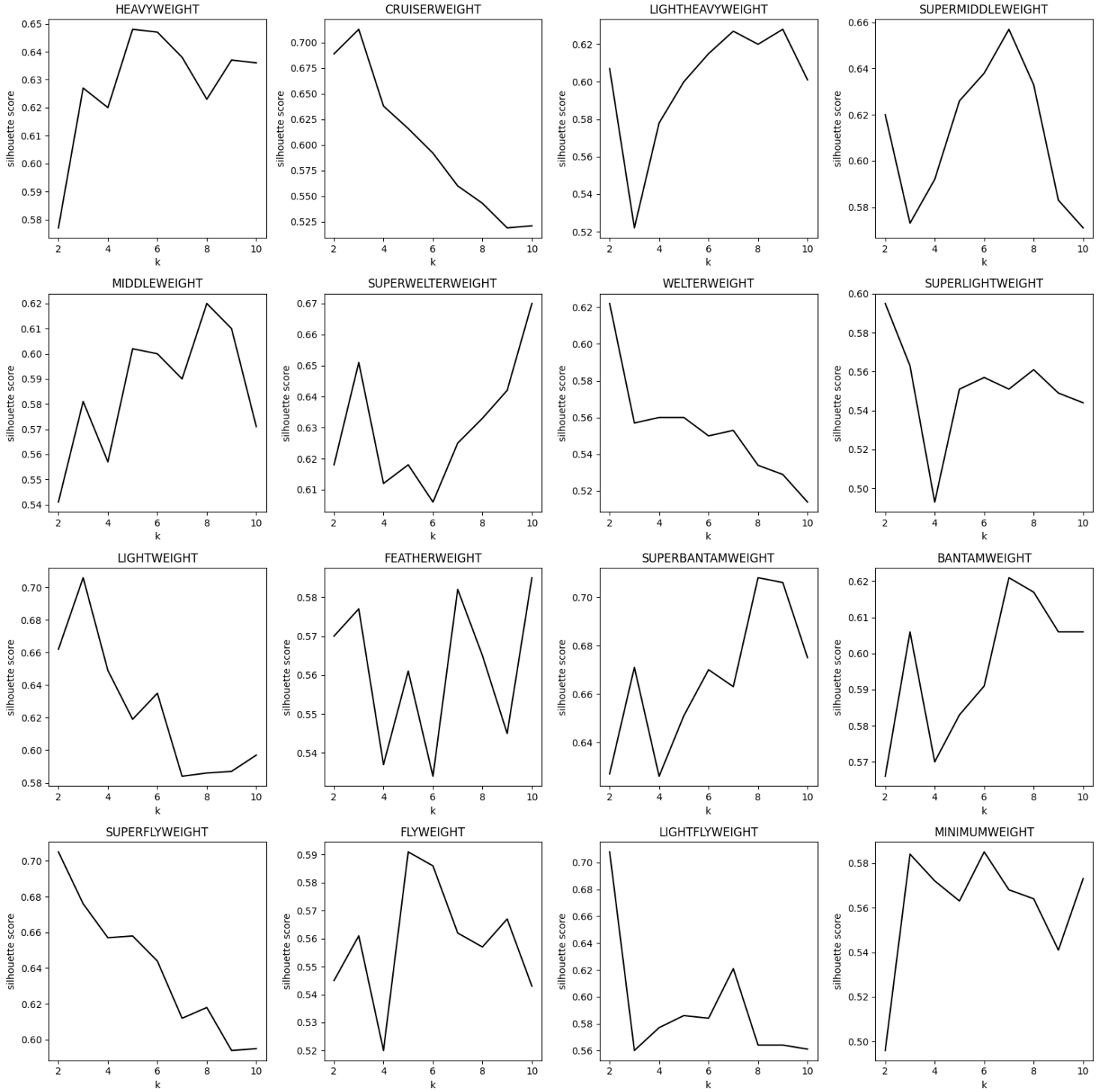
In [ ]:
```
k = 6
```

setting K to 6 to start.

In [ ]:
```
from statistics import mean

# Group the six_ds DataFrame by the div index column
grouped_df = six_ds.groupby('div index')

# Use the apply method to apply the k-means clustering algorithm to each group
```

```
grouped_df_clustered = grouped_df.apply(lambda group: KMeans(n_clusters=k).fit_predict(group))

# Create a figure with 16 subplots (one for each div index value)
fig, axes = plt.subplots(4, 4, figsize=(16, 16))

# Use a for loop to plot each group in a separate subplot
# Use a for loop to plot each group in a separate subplot
for i, (name, group) in enumerate(grouped_df):
    # Create an empty list to store the silhouette scores for this group
    silhouette_scores = []

    # Use a nested for loop to iterate over the range of k values from 2 to 8
    for k in range(2, 11):
        # Create a KMeans object
        model = KMeans(n_clusters=k)

        # Fit the model to the data in this group
        cluster_labels = model.fit_predict(group)

        # Calculate the silhouette score for this model
        score = silhouette_score(group, cluster_labels)

        # Append the silhouette score to the list
        silhouette_scores.append(round(score, 3))

    silhouette_scores_mean = round(mean(silhouette_scores),2)
    print(f'{name}: {silhouette_scores_mean}')

    # Use the plot method to plot the silhouette scores for this group
    axes[i // 4, i % 4].plot(range(2, 11), silhouette_scores, c='black')

    # Add x and y labels and a title to the subplot
    axes[i // 4, i % 4].set_xlabel('k')
    axes[i // 4, i % 4].set_ylabel('silhouette score')
    axes[i // 4, i % 4].set_title(f'{div_list[name].upper()}WEIGHT')

# Adjust the spacing between the subplots
fig.tight_layout()
```

```
0: 0.63
1: 0.6
2: 0.6
3: 0.61
4: 0.59
5: 0.63
6: 0.55
7: 0.55
8: 0.62
9: 0.56
10: 0.67
11: 0.6
12: 0.64
13: 0.56
14: 0.59
15: 0.56
```

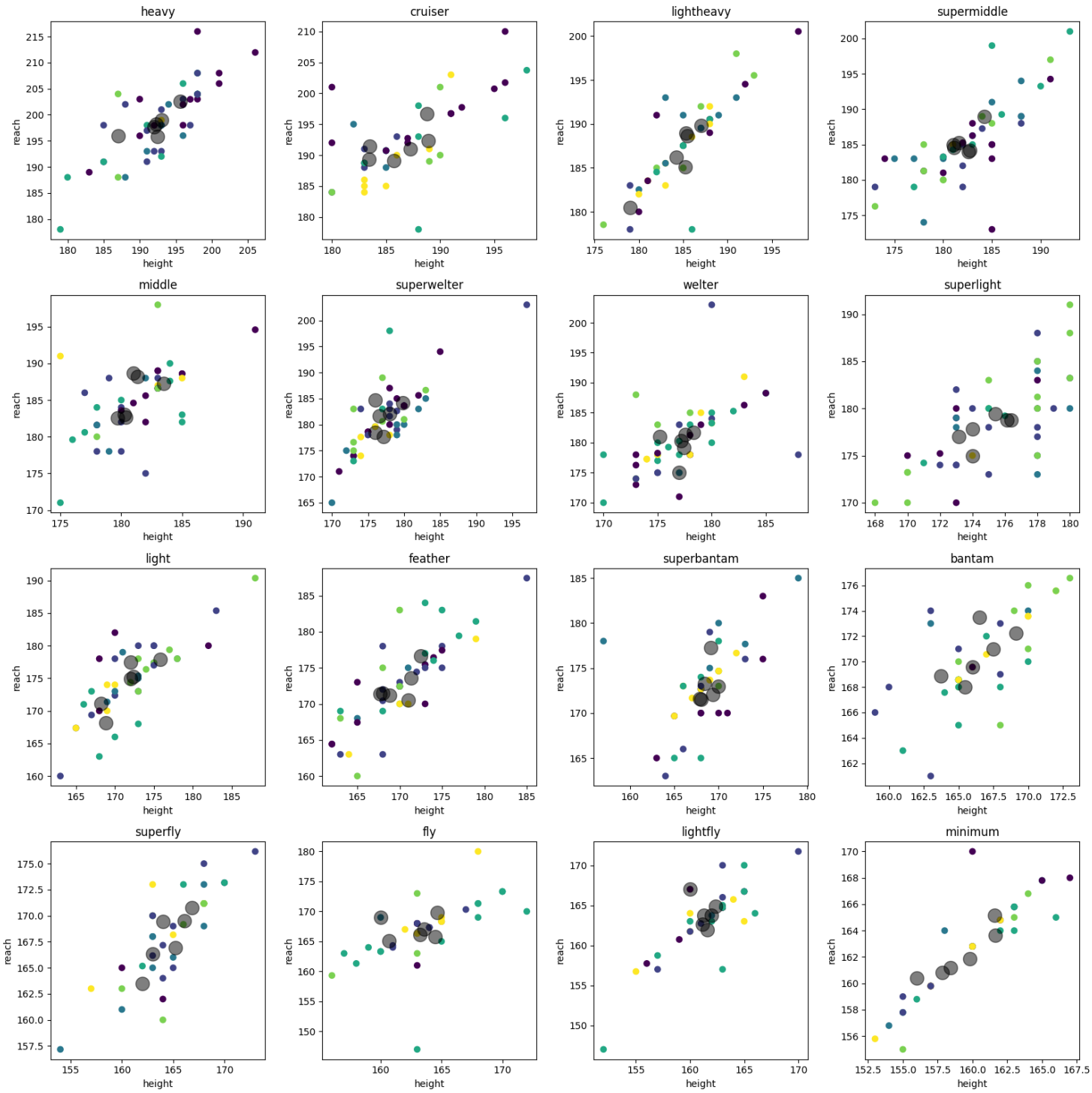


Identifying the silhouette scores. I save the k scores of each k value to a list of the scores by division and then calculate an average of them scores by division. I also visualise the K scores. Unfortunately it looks as though there is no one size fits all K value, and so I opted to use the mean value.

In [ ]:
```
k = 6
```

The mean K value.

In [ ]:
```
grouped_df = six_ds.groupby('div index')

# Create a figure with 16 subplots (one for each div index value)
fig, axes = plt.subplots(4, 4, figsize=(16, 16))

# Use a for loop to plot each group in a separate subplot
for i, (name, group) in enumerate(grouped_df):
    # Create a KMeans object
    model = KMeans(n_clusters=k)

    # Fit the model to the data in this group
    cluster_labels = model.fit_predict(group)

    # Use the scatter method to plot the data points
    axes[i // 4, i % 4].scatter(group['height'], group['reach'], c=cluster_labels, cmap='viridis')

    # Use the scatter method again to plot the cluster centers
    cluster_centers = model.cluster_centers_
    axes[i // 4, i % 4].scatter(cluster_centers[:, 0], cluster_centers[:, 1], c='black', s=200, alpha=0.5)

    # Add x and y labels and a title to the subplot
    axes[i // 4, i % 4].set_xlabel('height')
    axes[i // 4, i % 4].set_ylabel('reach')
    axes[i // 4, i % 4].set_title(f'{div_list[name]}')

# Adjust the spacing between the subplots
fig.tight_layout()
```



Visualising the K means clusters, can see in many cases there are overlapping clusters which imply I should continue to tune this model.

In [ ]:
```
grouped_df['height', 'age'].describe()
```

```
C:\Users\smbal\AppData\Local\Temp\ipykernel_15596\2799942490.py:1: FutureWarning: Indexing with multiple keys (implicitly converted to a tuple of keys) will be deprecated, use a list instead.
  grouped_df['height', 'age'].describe()
```

Out[ ]:

	height								age							
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max
div index																
0	47.0	193.042553	5.563692	179.0	190.50	193.0	197.5	206.0	47.0	33.212766	4.713238	23.0	30.00	33.0	35.50	43.0
1	39.0	187.128205	4.878414	180.0	183.00	186.0	190.5	198.0	39.0	32.717949	4.701336	24.0	30.00	32.0	34.50	52.0
2	39.0	185.153846	4.386213	176.0	182.00	185.0	188.0	198.0	39.0	31.512821	3.331443	24.0	29.50	31.0	33.00	39.0
3	44.0	182.477273	4.790925	173.0	180.00	182.5	185.0	193.0	44.0	30.386364	4.779880	21.0	27.00	30.0	33.25	43.0
4	40.0	181.200000	3.298795	175.0	178.75	182.0	183.0	191.0	40.0	30.675000	4.626498	23.0	27.75	30.0	33.00	43.0
5	46.0	177.652174	4.483141	170.0	174.25	178.0	180.0	197.0	46.0	30.434783	4.047782	21.0	27.00	31.0	33.75	37.0
6	45.0	177.244444	3.862145	170.0	175.00	177.0	180.0	188.0	45.0	30.800000	3.876503	23.0	28.00	31.0	34.00	38.0
7	43.0	175.325581	3.350344	168.0	173.00	175.0	178.0	180.0	43.0	29.813953	3.762318	23.0	27.00	30.0	33.00	39.0
8	41.0	172.024390	5.027364	163.0	169.00	172.0	175.0	188.0	41.0	30.048780	4.908927	23.0	27.00	29.0	33.00	46.0
9	42.0	170.357143	5.078993	162.0	168.00	170.0	173.0	185.0	42.0	27.976190	3.828571	21.0	25.00	28.0	30.75	38.0
10	40.0	168.600000	3.628943	157.0	167.75	168.0	170.0	179.0	40.0	28.925000	4.346454	22.0	26.00	28.0	32.00	43.0
11	37.0	166.432432	3.287523	159.0	165.00	166.0	169.0	173.0	37.0	29.243243	4.374201	22.0	26.00	29.0	31.00	42.0
12	36.0	164.722222	3.776704	154.0	163.00	164.5	168.0	173.0	36.0	30.138889	4.715644	22.0	26.75	30.5	34.00	40.0
13	34.0	163.705882	3.745943	156.0	161.25	163.0	165.0	172.0	34.0	28.352941	4.457764	23.0	25.00	27.0	31.50	40.0
14	31.0	161.548387	3.613297	152.0	160.00	163.0	163.0	170.0	31.0	27.677419	4.407472	20.0	25.50	27.0	29.50	42.0
15	33.0	159.818182	3.778678	153.0	157.00	160.0	163.0	167.0	33.0	26.787879	4.083179	20.0	24.00	26.0	28.00	37.0

Summary statistics by division for the height and age columns

In [ ]:
```
for name, group in grouped_df:
    height_q1 = group['height'].quantile(0.25)
    height_q3 = group['height'].quantile(0.75)
```

```
    age_q1 = group['age'].quantile(0.25)
    age_q3 = group['age'].quantile(0.75)

    height_IQR = height_q3 - height_q1
    age_IQR = age_q3 - age_q1

    height_outliers = group[( (group['height'] < (height_q1 - 1.5*height_IQR)) | (group['height'] > (height_q3 + 1.5*height_IQR)) )]
    age_outliers = group[( (group['age'] < (age_q1 - 1.5*age_IQR)) | (group['age'] > (age_q3 + 1.5*age_IQR)) )]

    # Count the number of outliers in the 'height' and 'age' columns
    height_outlier_count = height_outliers.count()
    age_outlier_count = age_outliers.count()

    print(f'{div_list[name]}: height outliers: {height_outlier_count.height} - age outliers: {age_outlier_count.age}')
```

```
heavy: height outliers: 1 - age outliers: 0
cruiser: height outliers: 0 - age outliers: 1
lightheavy: height outliers: 1 - age outliers: 3
supermiddle: height outliers: 1 - age outliers: 1
middle: height outliers: 1 - age outliers: 1
superwelter: height outliers: 1 - age outliers: 0
welter: height outliers: 1 - age outliers: 0
superlight: height outliers: 0 - age outliers: 0
light: height outliers: 1 - age outliers: 1
feather: height outliers: 1 - age outliers: 0
superbantam: height outliers: 6 - age outliers: 1
bantam: height outliers: 0 - age outliers: 3
superfly: height outliers: 1 - age outliers: 0
fly: height outliers: 1 - age outliers: 0
lightfly: height outliers: 3 - age outliers: 1
minimum: height outliers: 0 - age outliers: 2
```

During the K means cluster visualisations, I noticed there were many outliers, at least so I thought! Here, I am using the inter-quartile range method to identify outliers of age and height. However, the method did not return many outliers within the data.

In [ ]:
```
X = preproc_ds[['age', 'bouts', 'height', 'reach', 'stance', 'wins', 'losses', 'KO wins', 'days active', 'w_minus_l','opp_win_pct']]

X
```

Out[ ]:

	age	bouts	height	reach	stance	wins	losses	KO wins	days active	w_minus_l	opp_win_pct
0	35.0	20	191.0	198.0	1	20	0	13	3279	394	0.85
1	33.0	27	198.0	208.0	0	24	3	22	3328	531	0.82
2	33.0	37	188.0	188.0	0	35	2	22	4911	408	0.76
3	35.0	31	193.0	198.0	0	28	3	19	4099	276	0.65
4	37.0	15	198.0	203.0	0	15	0	14	1754	285	0.86
...	...	...	...	...	...	...	...	...	...	...	...
632	24.0	20	156.0	158.8	0	9	8	4	2677	76	0.71
633	28.0	15	165.0	167.8	0	11	3	7	2578	26	0.60
634	26.0	3	155.0	157.8	0	3	0	0	412	15	0.69
635	22.0	10	162.0	164.8	1	6	4	1	1927	27	0.78
636	28.0	24	165.0	167.8	1	19	5	9	3237	-139	0.36

637 rows × 11 columns

Creating a new K means cluster model this time with more columns for processing.

In [ ]:
```
kmeans = KMeans(n_clusters=16, random_state=42)
kmeans.fit(X)
```

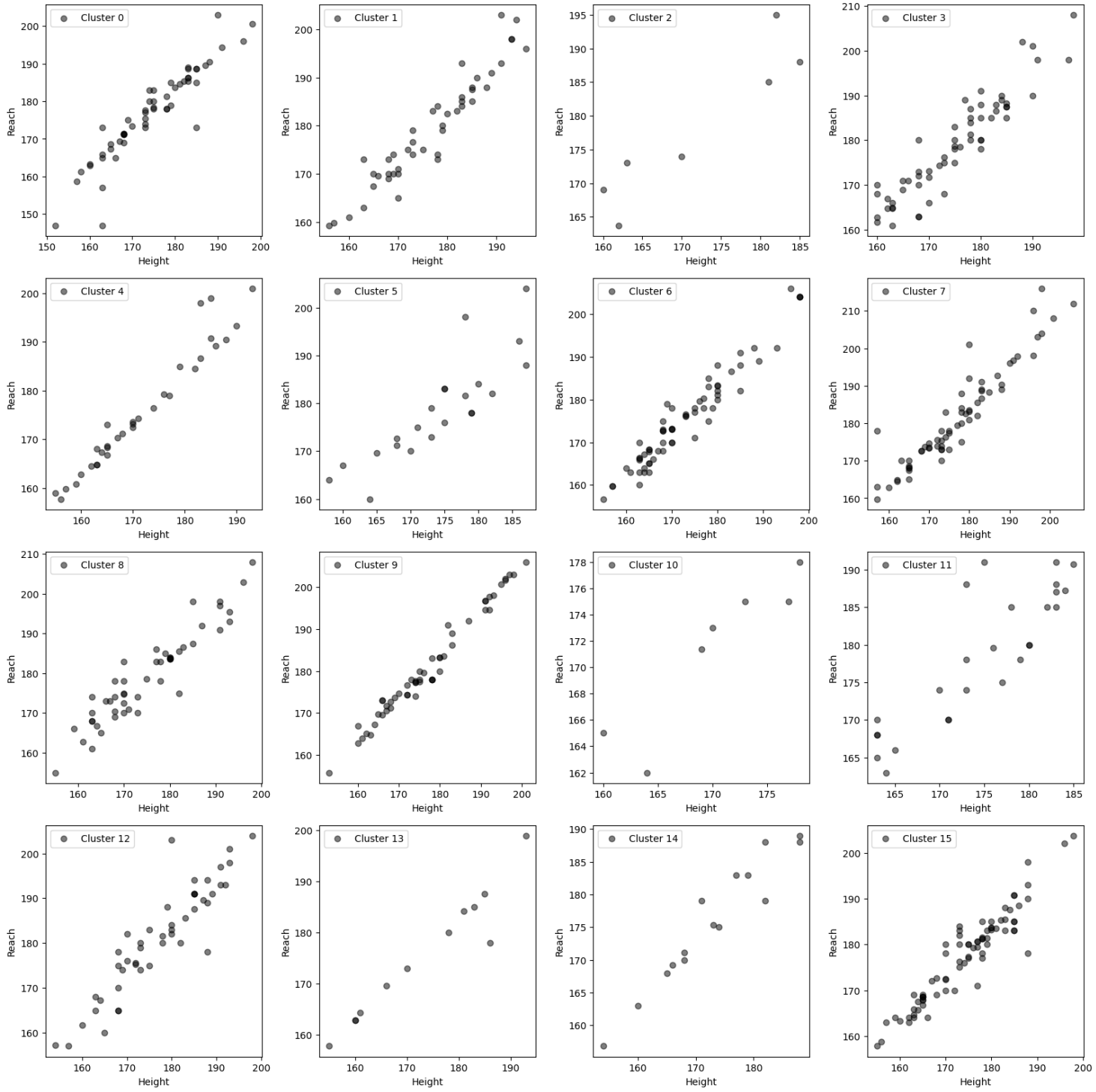
Out[ ]:
```
KMeans(n_clusters=16, random_state=42)
```

Fitting the model to X.

In [ ]:
```
# Predict the cluster labels for each boxer
cluster_labels = kmeans.predict(X)
```

Obtaining cluster labels for each boxer based off the data within the dataframe. These are then saved to the cluster_labels variable. The aim is to see whether the model can be used to predict hypothetical match ups...

In [ ]:
```
# Create a figure with 16 subplots (one for each cluster)
fig, axes = plt.subplots(4, 4, figsize=(16, 16))

# Loop over the clusters
for i in range(16):
    # Select the data for the current cluster
    x = preproc_ds.loc[cluster_labels == i, "height"]
    y = preproc_ds.loc[cluster_labels == i, "reach"]

    # Create a scatter plot for the current cluster
    axes[i // 4, i % 4].scatter(x, y, c='black', alpha=0.5, label=f"Cluster {i}")
    axes[i // 4, i % 4].set_xlabel("Height")
    axes[i // 4, i % 4].set_ylabel("Reach")
    axes[i // 4, i % 4].legend()

fig.tight_layout()
```



Plotting the respective groups. Some have very few points (clusters 2, 10, 13, 14) while others have many (clusters 0, 3, 7 ,15). It looks as though it will be unsuccesful.

In [ ]:
```
clustered_df = preproc_ds
```

New dataframe to add the clusters

In [ ]:
```
clustered_df['cluster label'] = cluster_labels
```

Adding the cluster labels to the cluster label column in the new dataframe

In [ ]:
```
clustered_df.columns
```

Out[ ]:
```
Index(['division rating', 'div index', 'bouts', 'rounds', 'KOs', 'days active',
       'age', 'stance', 'height', 'reach', 'wins', 'losses', 'draws',
       'KO wins', 'KO losses', 'opp_wins', 'opp_losses', 'opp_draws',
       'w_minus_l', 'opp_win_pct', 'cluster label'],
      dtype='object')
```

In [ ]:
```
clustered_df
```

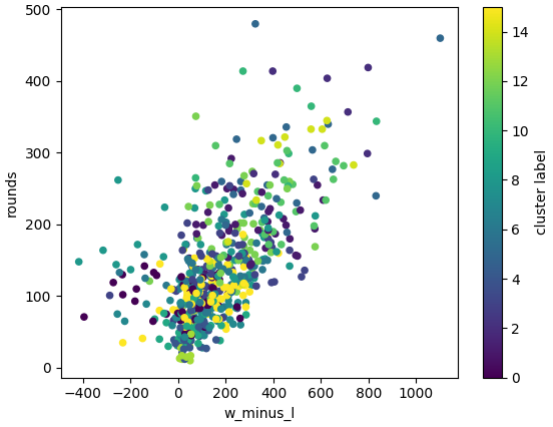
Out[ ]:

	division rating	div index	bouts	rounds	KOs	days active	age	stance	height	reach	...	losses	draws	KO wins	KO losses	opp_wins	opp_losses	opp_draws	w_minus_l	opp_win_pct	cluster label
0	1	0	20	168	65.00	3279	35.0	1	191.0	198.0	...	0	0	13	0	473	79	9	394	0.85	3
1	2	0	27	136	81.48	3328	33.0	0	198.0	208.0	...	3	0	22	1	672	141	11	531	0.82	3
2	4	0	37	187	59.46	4911	33.0	0	188.0	188.0	...	2	0	22	0	576	168	33	408	0.76	1
3	5	0	31	170	61.29	4099	35.0	0	193.0	198.0	...	3	0	19	3	567	291	35	276	0.65	12
4	6	0	15	74	93.33	1754	37.0	0	198.0	203.0	...	0	0	14	0	340	55	6	285	0.86	9
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
632	43	15	20	104	20.00	2677	24.0	0	156.0	158.8	...	8	3	4	4	126	50	4	76	0.71	15
633	46	15	15	70	46.67	2578	28.0	0	165.0	167.8	...	3	1	7	2	76	50	8	26	0.60	15
634	49	15	3	17	0.00	412	26.0	0	155.0	157.8	...	0	0	0	0	27	12	1	15	0.69	13
635	50	15	10	43	10.00	1927	22.0	1	162.0	164.8	...	4	0	1	2	37	10	1	27	0.78	7
636	51	15	24	125	37.50	3237	28.0	1	165.0	167.8	...	5	0	9	3	157	296	33	-139	0.36	6

637 rows × 21 columns

In [ ]:
```
clustered_df.plot.scatter(x='w_minus_l', y='rounds', c='cluster label', colormap='viridis')
```

Out[ ]:
```
<AxesSubplot: xlabel='w_minus_l', ylabel='rounds'>
```



Simple visualisation to see if there are any patterns - which is unclear.

In [ ]:
```
division_index = 0
boxer_x_rating = 1
boxer_y_rating = 30

# Selct the rows corresponding to boxer X and boxer_Y
boxer_x_row = clustered_df.loc[(clustered_df["div index"] == division_index) & (clustered_df["division rating"] == boxer_x_rating)]
boxer_y_row = clustered_df.loc[(clustered_df["div index"] == division_index) & (clustered_df["division rating"] == boxer_y_rating)]

# Get the indices of the rows
boxer_x_index = boxer_x_row.index[0]
boxer_y_index = boxer_y_row.index[0]

# Get the cluster labels for each boxer
boxer_x_cluster = clustered_df.loc[boxer_x_index, "cluster label"]
boxer_y_cluster = clustered_df.loc[boxer_y_index, "cluster label"]

# Get the number of bouts fought by each boxer
boxer_x_bouts = clustered_df.loc[boxer_x_index, "bouts"]
boxer_y_bouts = clustered_df.loc[boxer_y_index, "bouts"]

# Calculate the weighted average of the cluster labels
weighted_average = (boxer_x_cluster * boxer_x_bouts + boxer_y_cluster * boxer_y_bouts) / (boxer_x_bouts + boxer_y_bouts)
```

Attempting to create a weighted average value using the cluster values...

In [ ]:
```
print(f'FIGHTERS ARE IN THE {div_list[division_index].upper()}WEIGHT DIVISION.\n')
print(f'{clean_ds.name[boxer_x_index]} - Cluster #{boxer_x_cluster}')
print(f'{clean_ds.name[boxer_y_index]} - Cluster #{boxer_y_cluster}\n')

print('CLUSTER PREDICTION METHOD')
# Predict the outcome of the bout
if boxer_x_cluster > boxer_y_cluster:
  print(f'{clean_ds.name[boxer_x_index]} is predicted to win the bout.')
elif boxer_y_cluster > boxer_x_cluster:
  print(f'{clean_ds.name[boxer_y_index]} is predicted to win the bout.')
else:
  print("The outcome of the bout is uncertain.")

print('\nWEIGHTED AVERAGE METHOD')
print(f'wavg: {weighted_average}')
if weighted_average >= 2.5:
  print(f'{clean_ds.name[boxer_x_index]} is predicted to win the bout.')
elif weighted_average < 2.5:
  print(f'{clean_ds.name[boxer_y_index]} is predicted to win the bout.')
else:
  print("The outcome of the bout is uncertain.")
```

```
FIGHTERS ARE IN THE HEAVYWEIGHT DIVISION.

Oleksandr Usyk - Cluster #3
Efe Ajagba - Cluster #7

CLUSTER PREDICTION METHOD
Efe Ajagba is predicted to win the bout.

WEIGHTED AVERAGE METHOD
wavg: 4.837837837837838
Oleksandr Usyk is predicted to win the bout.
```

Because the K means cluster label definitions are unclear, the model believes that the boxer ranked 30th in the heavyweight division would beat the world champion heavyweight boxer. With the model embedded in the weighted average formula at least corrects this, however I used an arbitary 2.5 value to guage whether boxer X or Y would win the bout. Ultimately this needs much refining and the model should be trained on data within the original second dataset, where I do have data on boxers who have fought each other and their outcomes.

In [ ]:
```
preproc_ds
```

Out[ ]:

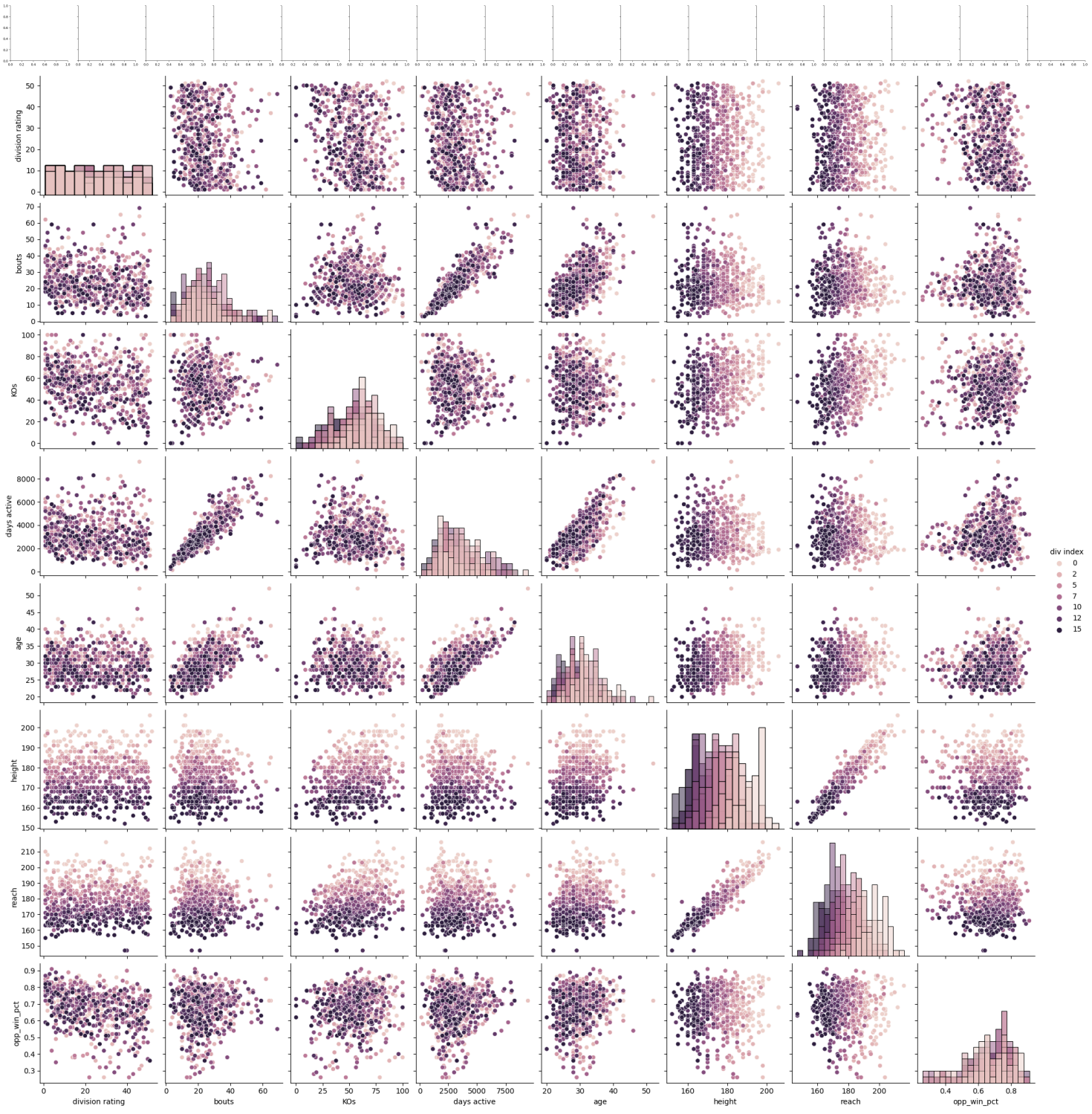
	division rating	div index	bouts	rounds	KOs	days active	age	stance	height	reach	...	losses	draws	KO wins	KO losses	opp_wins	opp_losses	opp_draws	w_minus_l	opp_win_pct	cluster label
0	1	0	20	168	65.00	3279	35.0	1	191.0	198.0	...	0	0	13	0	473	79	9	394	0.85	3
1	2	0	27	136	81.48	3328	33.0	0	198.0	208.0	...	3	0	22	1	672	141	11	531	0.82	3
2	4	0	37	187	59.46	4911	33.0	0	188.0	188.0	...	2	0	22	0	576	168	33	408	0.76	1
3	5	0	31	170	61.29	4099	35.0	0	193.0	198.0	...	3	0	19	3	567	291	35	276	0.65	12
4	6	0	15	74	93.33	1754	37.0	0	198.0	203.0	...	0	0	14	0	340	55	6	285	0.86	9
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
632	43	15	20	104	20.00	2677	24.0	0	156.0	158.8	...	8	3	4	4	126	50	4	76	0.71	15
633	46	15	15	70	46.67	2578	28.0	0	165.0	167.8	...	3	1	7	2	76	50	8	26	0.60	15
634	49	15	3	17	0.00	412	26.0	0	155.0	157.8	...	0	0	0	0	27	12	1	15	0.69	13
635	50	15	10	43	10.00	1927	22.0	1	162.0	164.8	...	4	0	1	2	37	10	1	27	0.78	7
636	51	15	24	125	37.50	3237	28.0	1	165.0	167.8	...	5	0	9	3	157	296	33	-139	0.36	6

637 rows × 21 columns

In [ ]:
```
preproc_ds.dtypes
```

Out[ ]:
```
division rating      int64
div index            int32
bouts                int64
rounds               int64
KOs                float64
days active          int32
age                float64
stance               int32
height             float64
reach              float64
wins                 int64
losses               int64
draws                int64
KO wins              int64
KO losses            int64
opp_wins             int64
opp_losses           int64
opp_draws            int64
w_minus_l            int64
opp_win_pct        float64
cluster label        int32
dtype: object
```

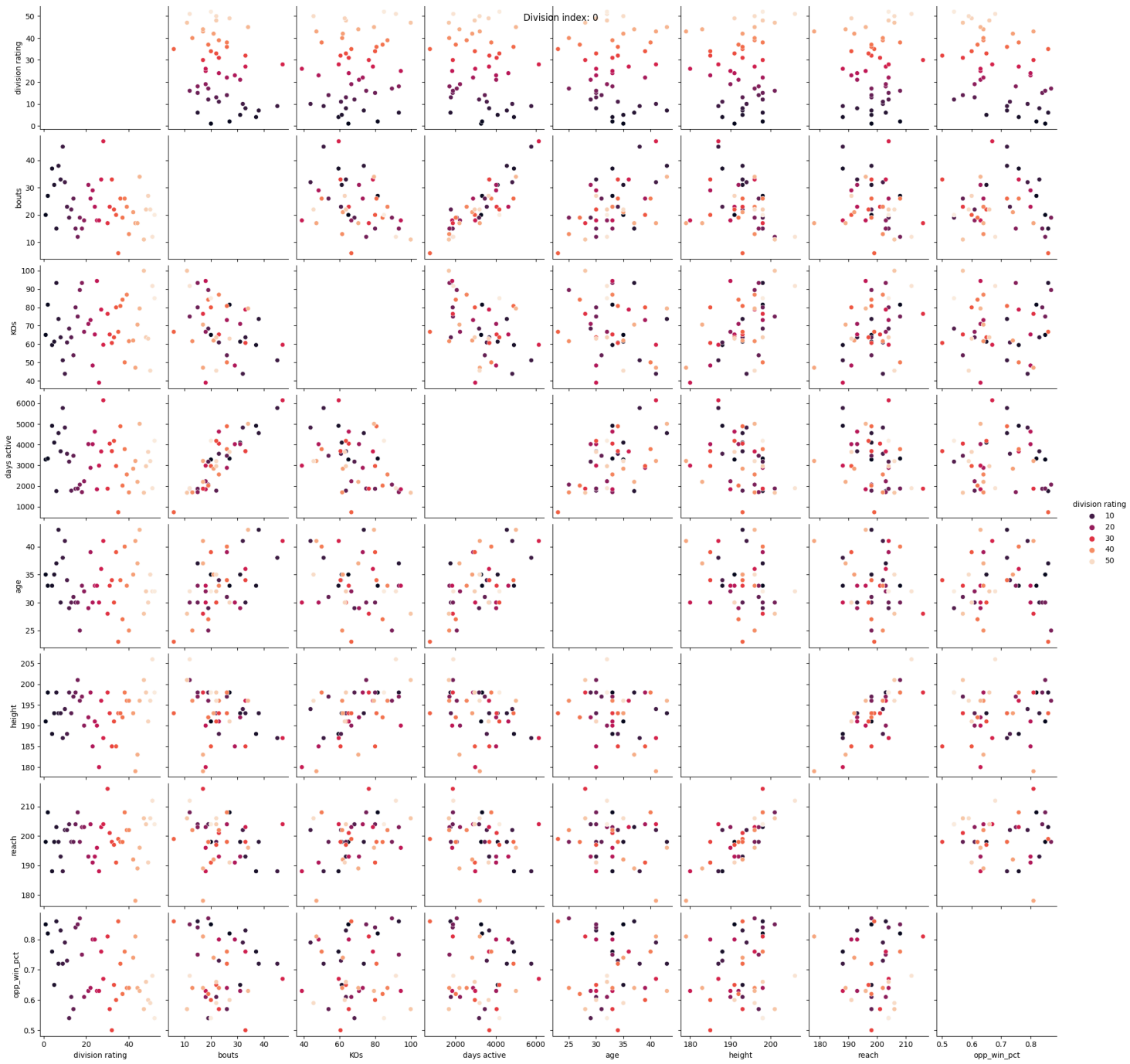
In [ ]:
```
import seaborn as sns

# Create a list of the columns you want to plot
plot_columns = ['division rating', 'bouts', 'KOs', 'days active','age', 'height', 'reach', 'opp_win_pct']

# Use the FacetGrid class to create a grid of plots, with div_index as the colour basis
g = sns.FacetGrid(preproc_ds, col='div index')

# Use map to apply the pairplot function to each subplot
g = sns.pairplot(preproc_ds, x_vars=plot_columns, y_vars=plot_columns, hue='div index', diag_kind='hist')
```

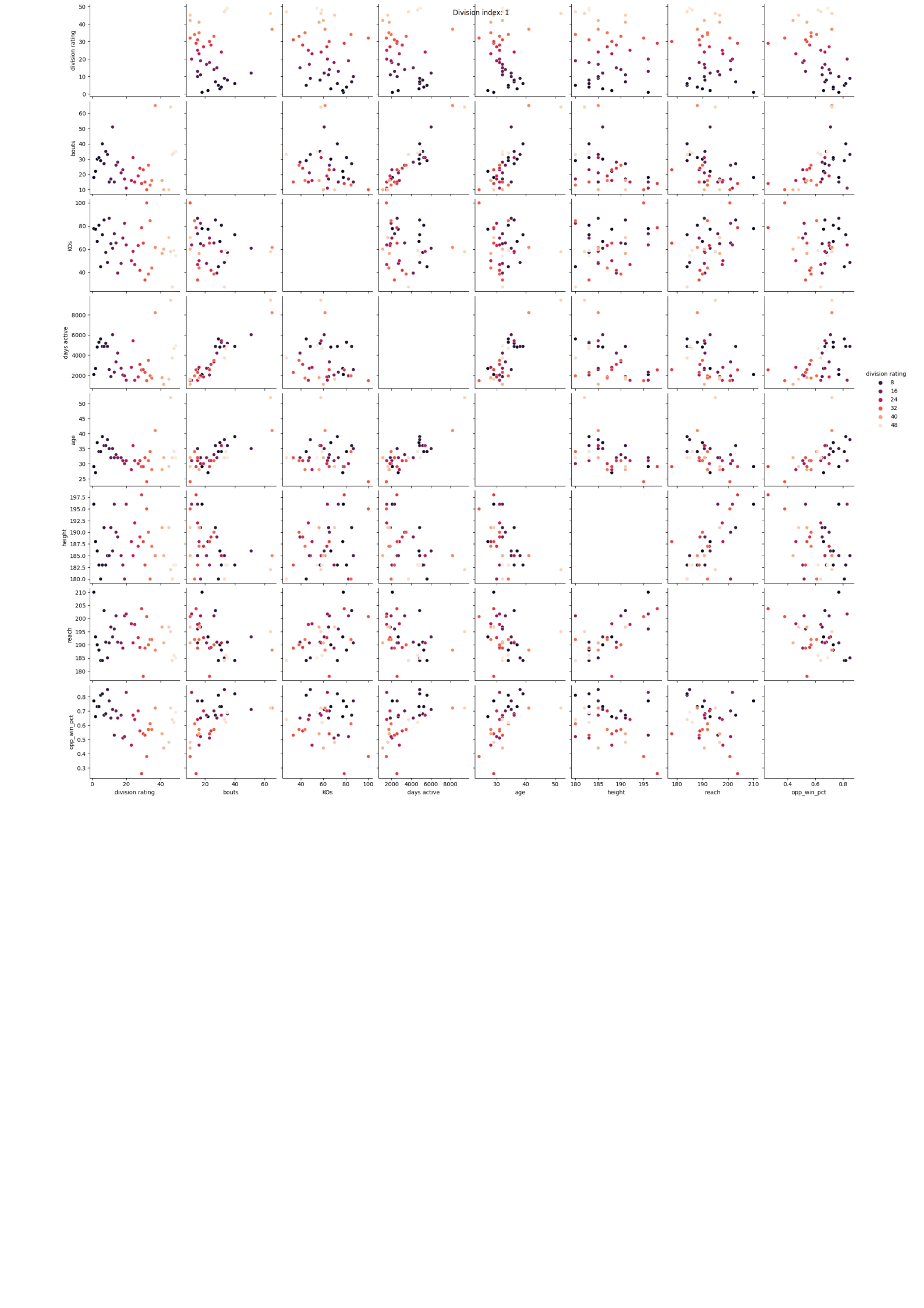


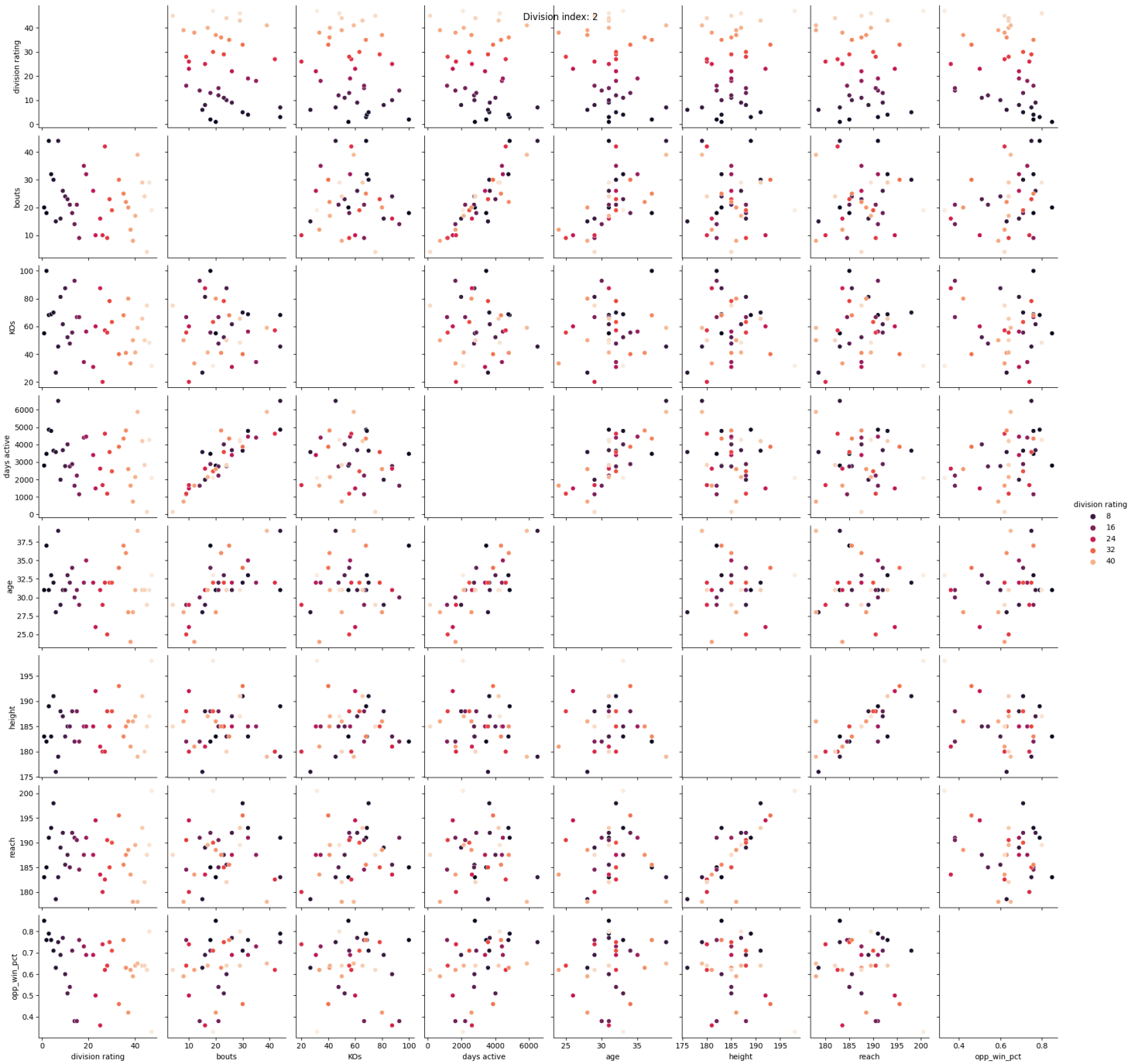
using the pairplot function to create paired plots of division rating, bouts, KOs, days active, age, height, reach, and opp_win_pct. Unfortunately this didn't lead to any clear relationships, with a lot of noise due to not being filtered for divisions.

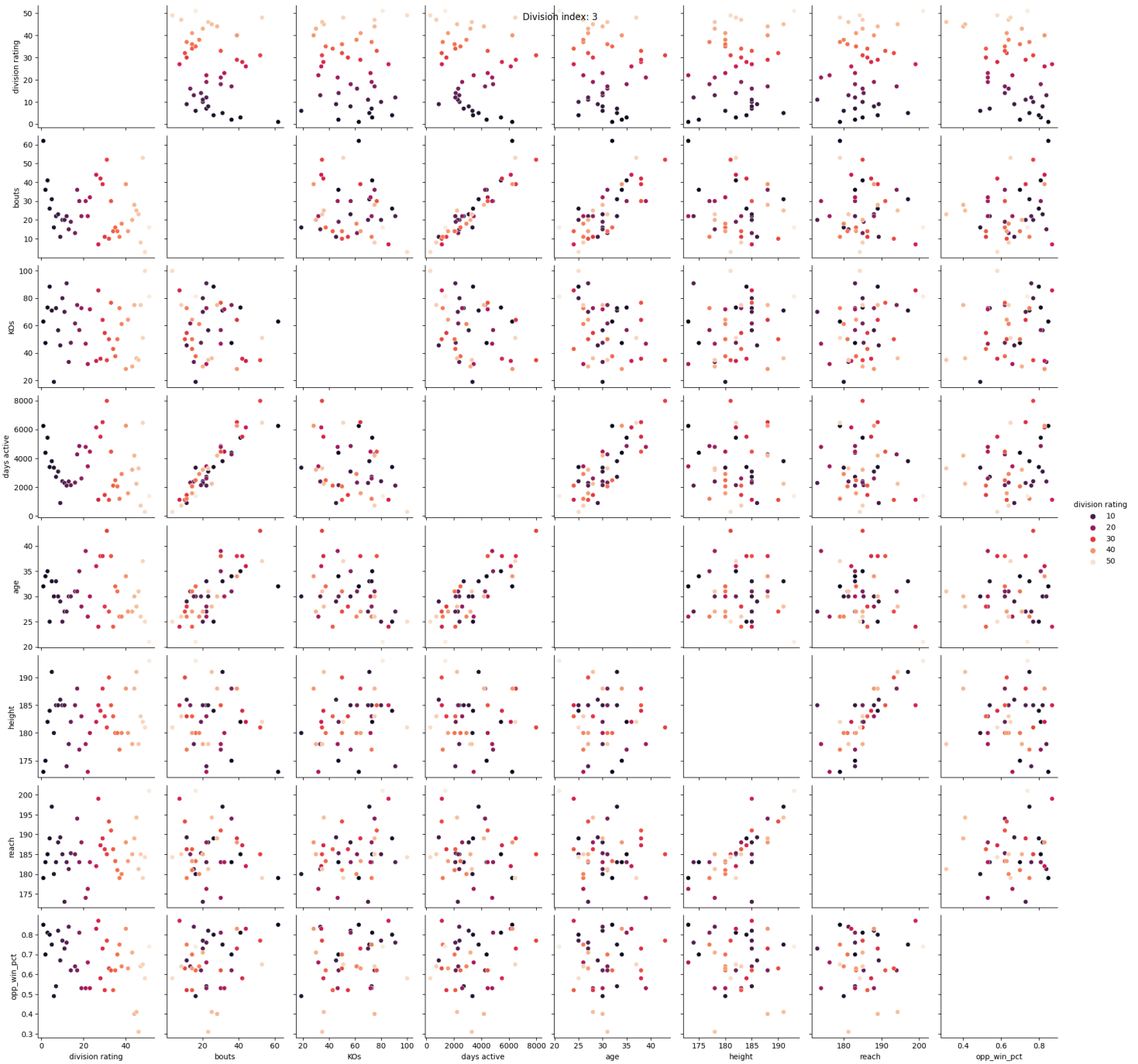
In [ ]:
```
# Get the unique values of the div index column
div_index_values = preproc_ds['div index'].unique()

# Loop over the unique values and create a pairplot for each one
for div_index in div_index_values:
```

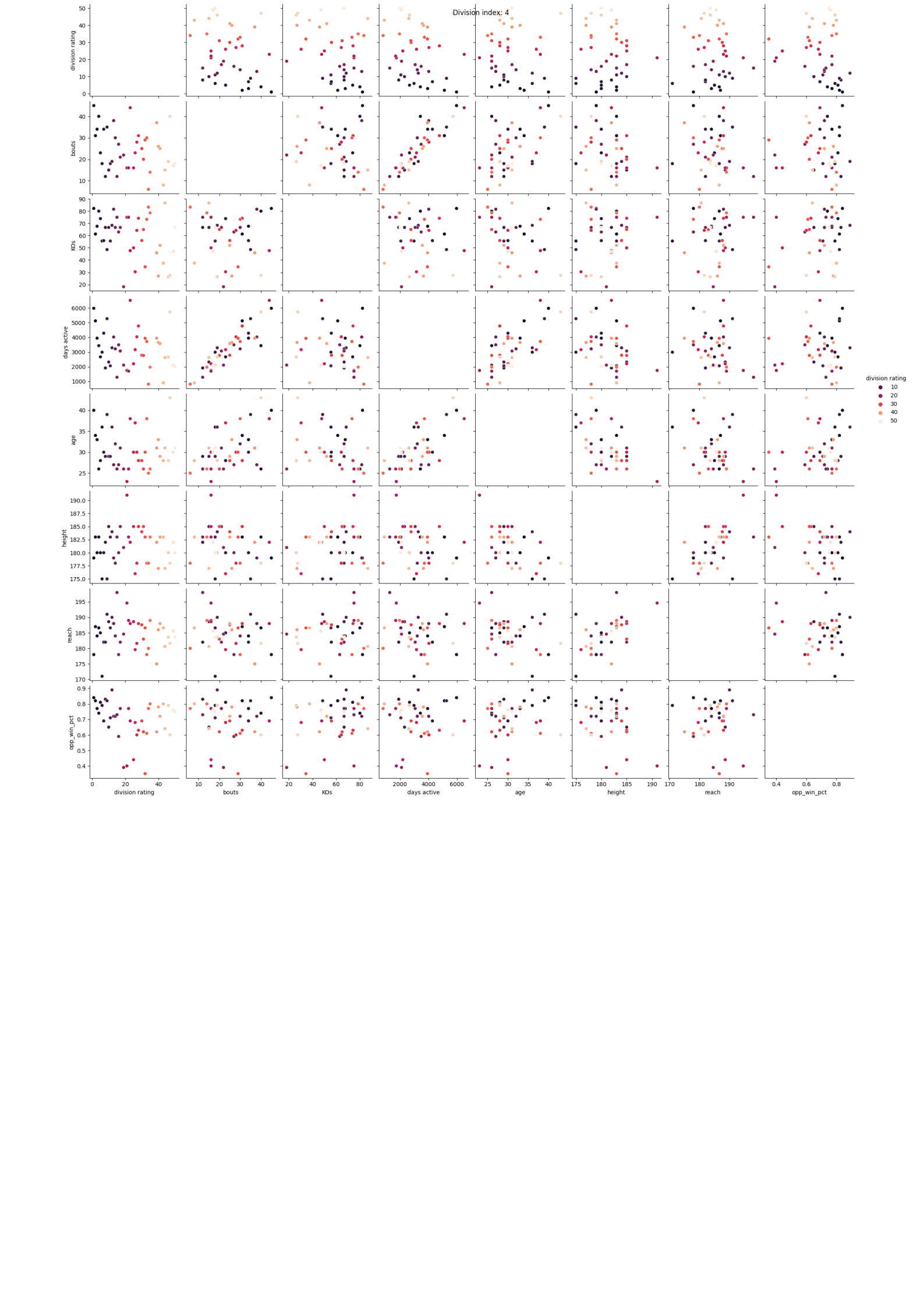
```
# Select the rows with the current div index value
df = preproc_ds[preproc_ds['div index'] == div_index]

# Create the pairplot
g = sns.pairplot(df, x_vars=plot_columns, y_vars=plot_columns, diag_kind='none', hue='division rating', palette="rocket")

# Set the title of the plot to the div index value
g.fig.suptitle(f'Division index: {div_index}')

# Show the plot
plt.show()
```

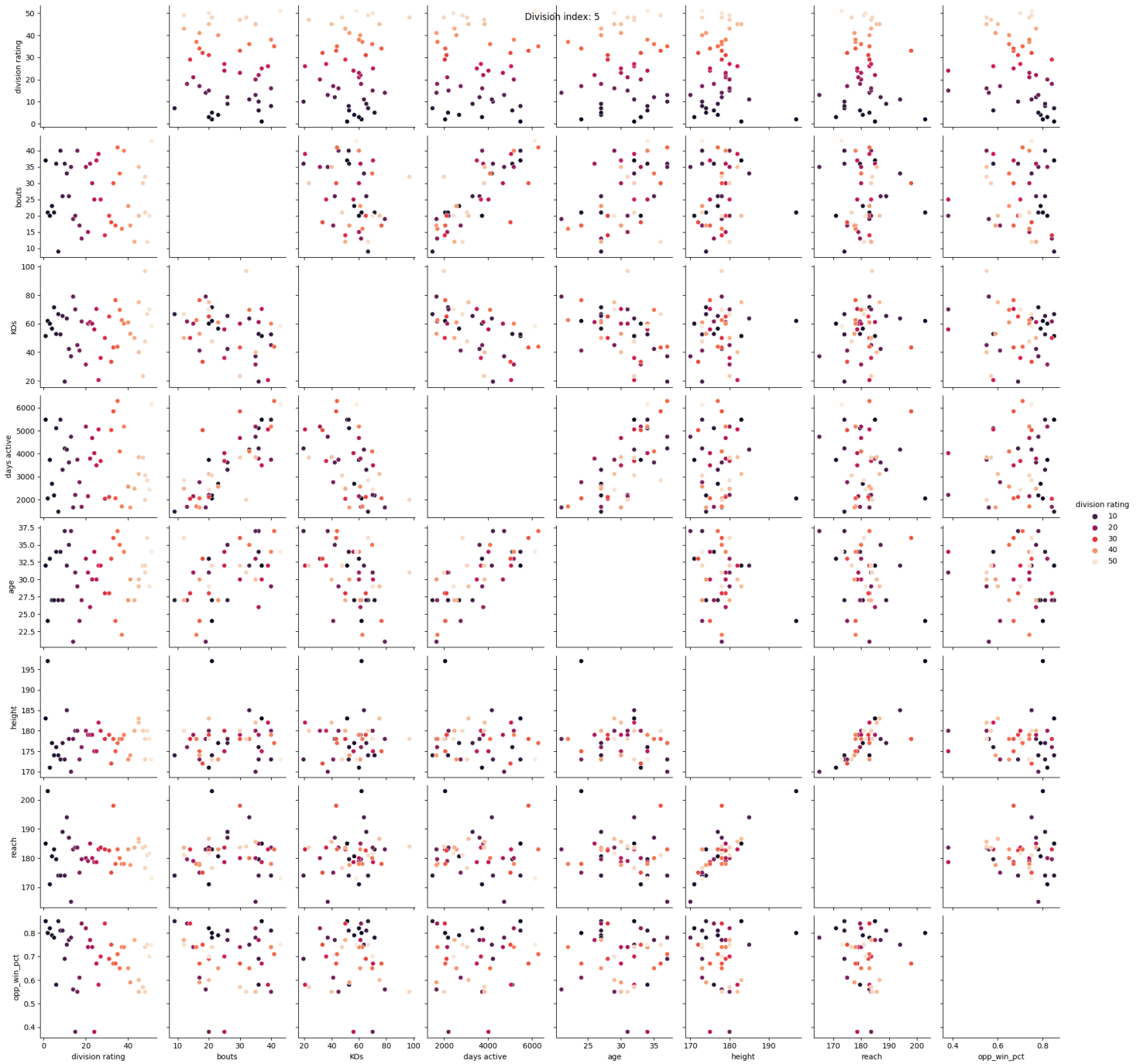


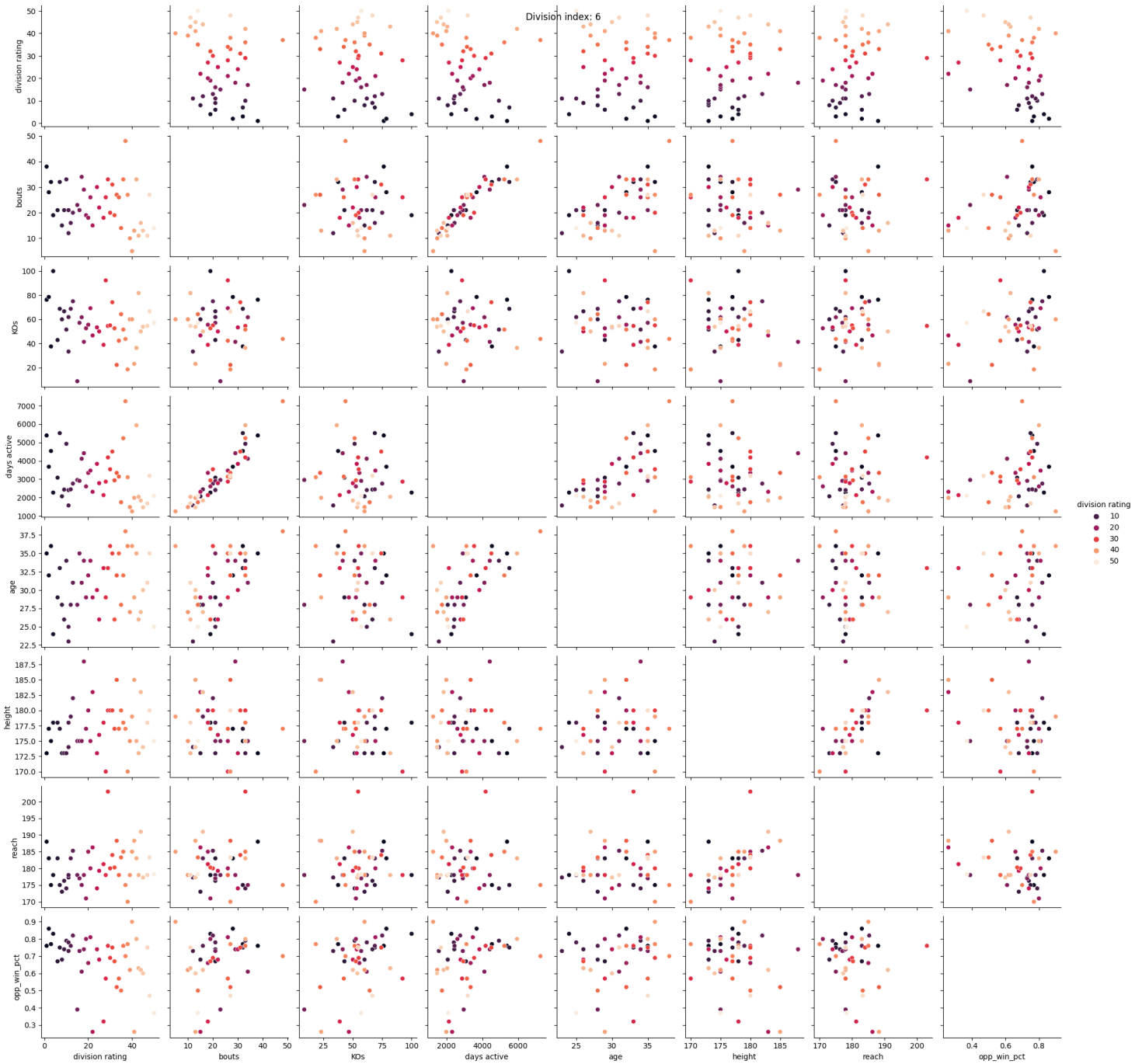

Division index: 1
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
8
16
24
32
40
48

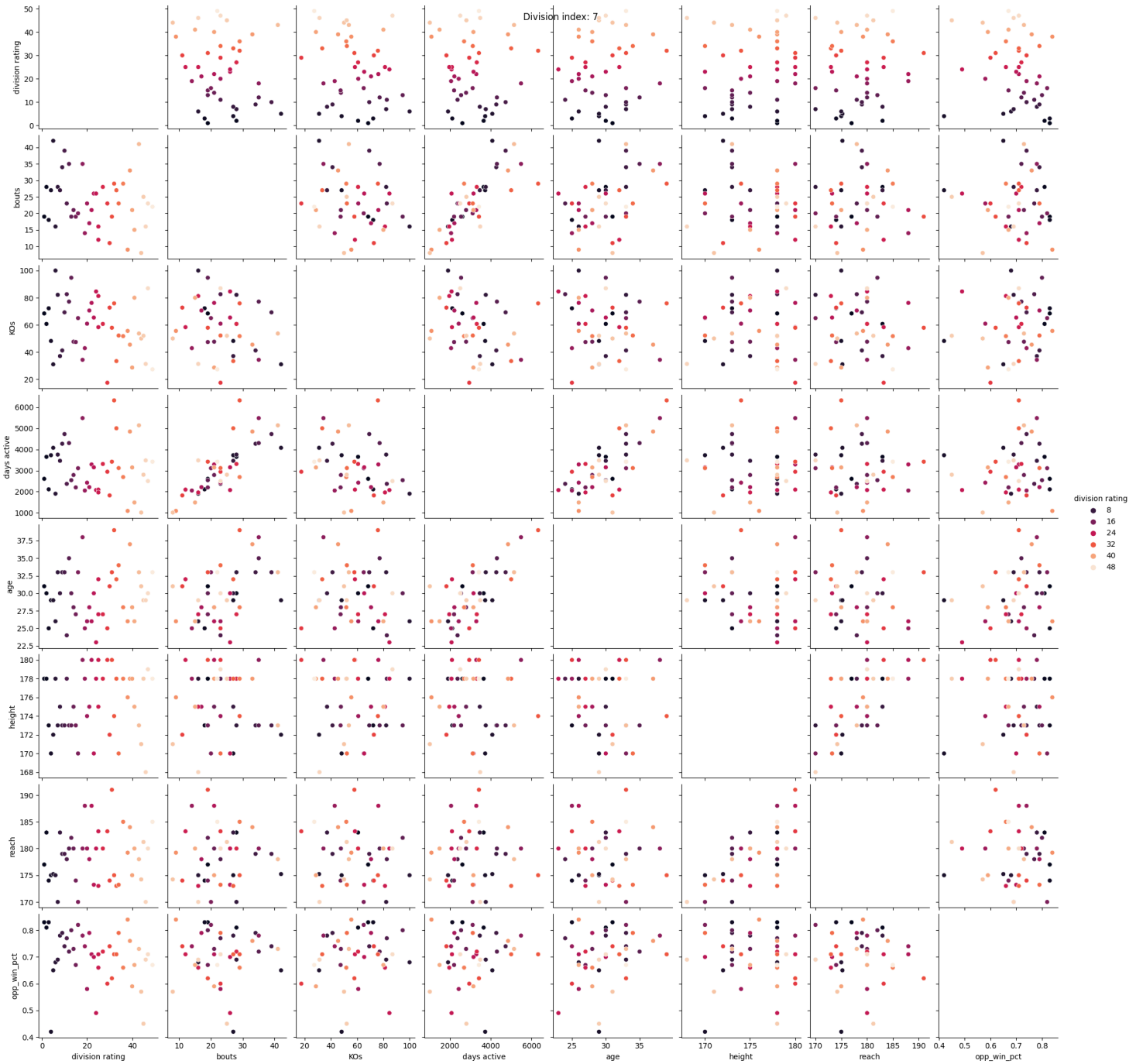

Division index: 2
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
8
16
24
32
40

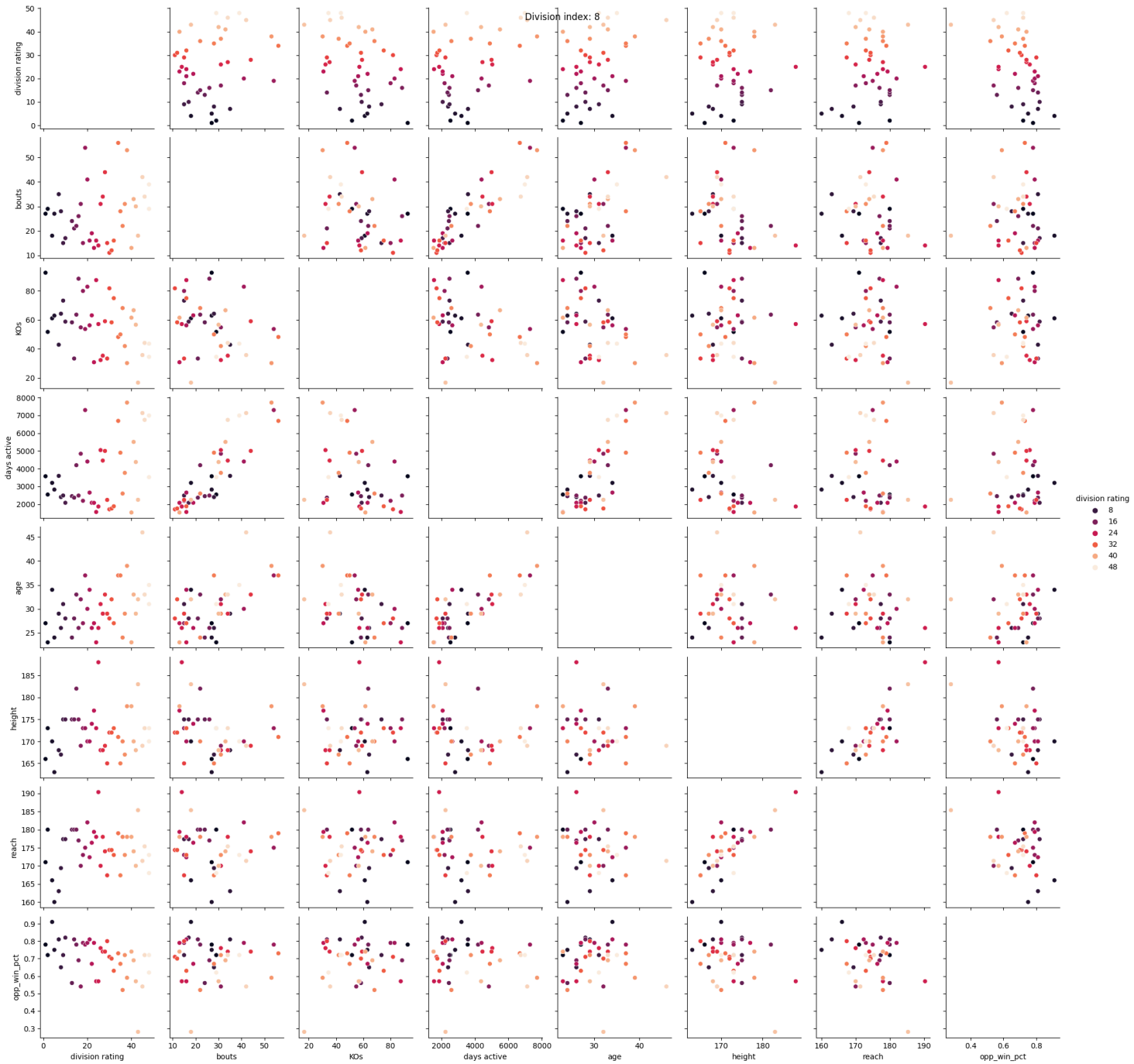

Division index: 3
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
10
20
30
40
50

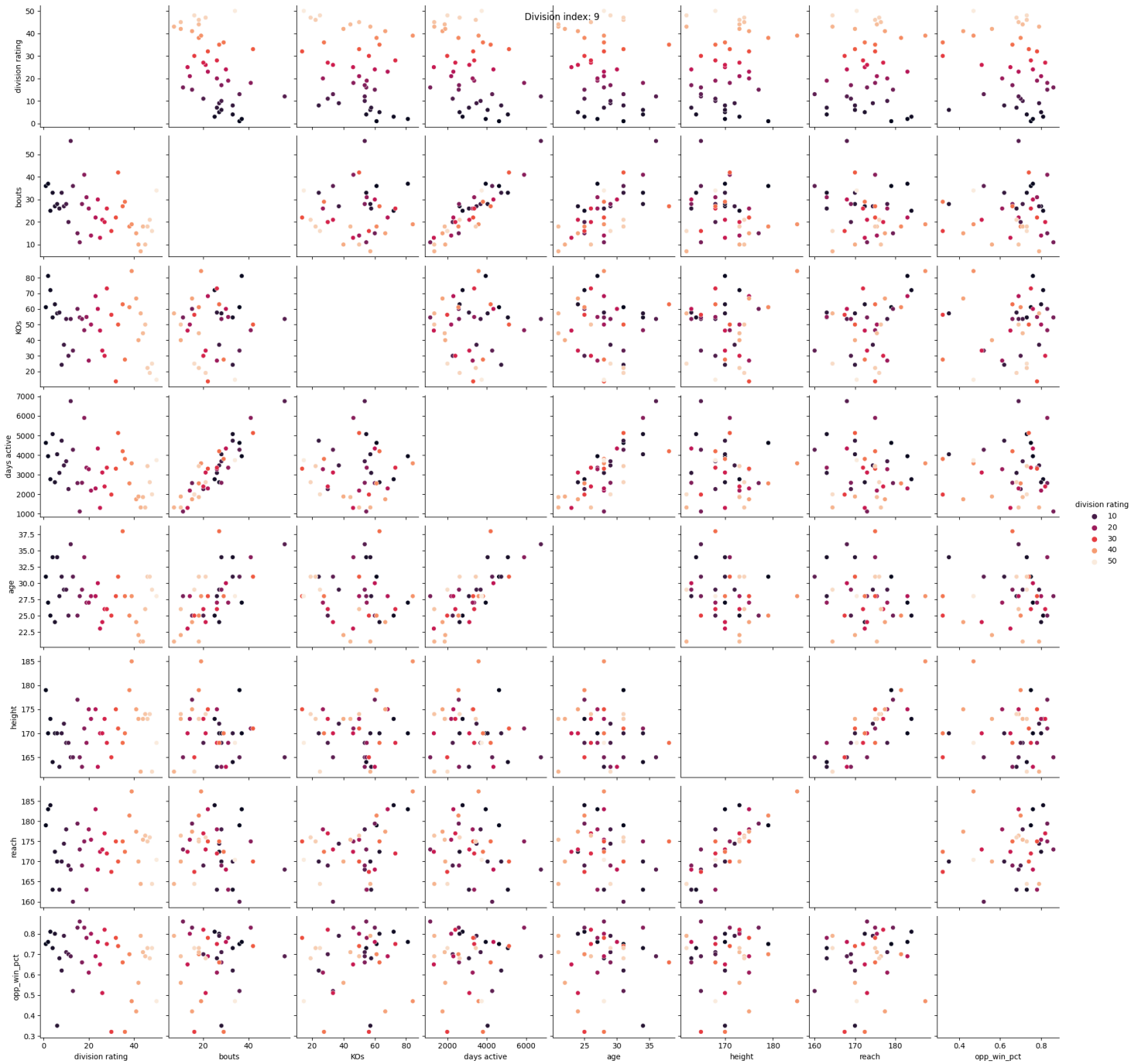

Division index: 4
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
10
20
30
40
50

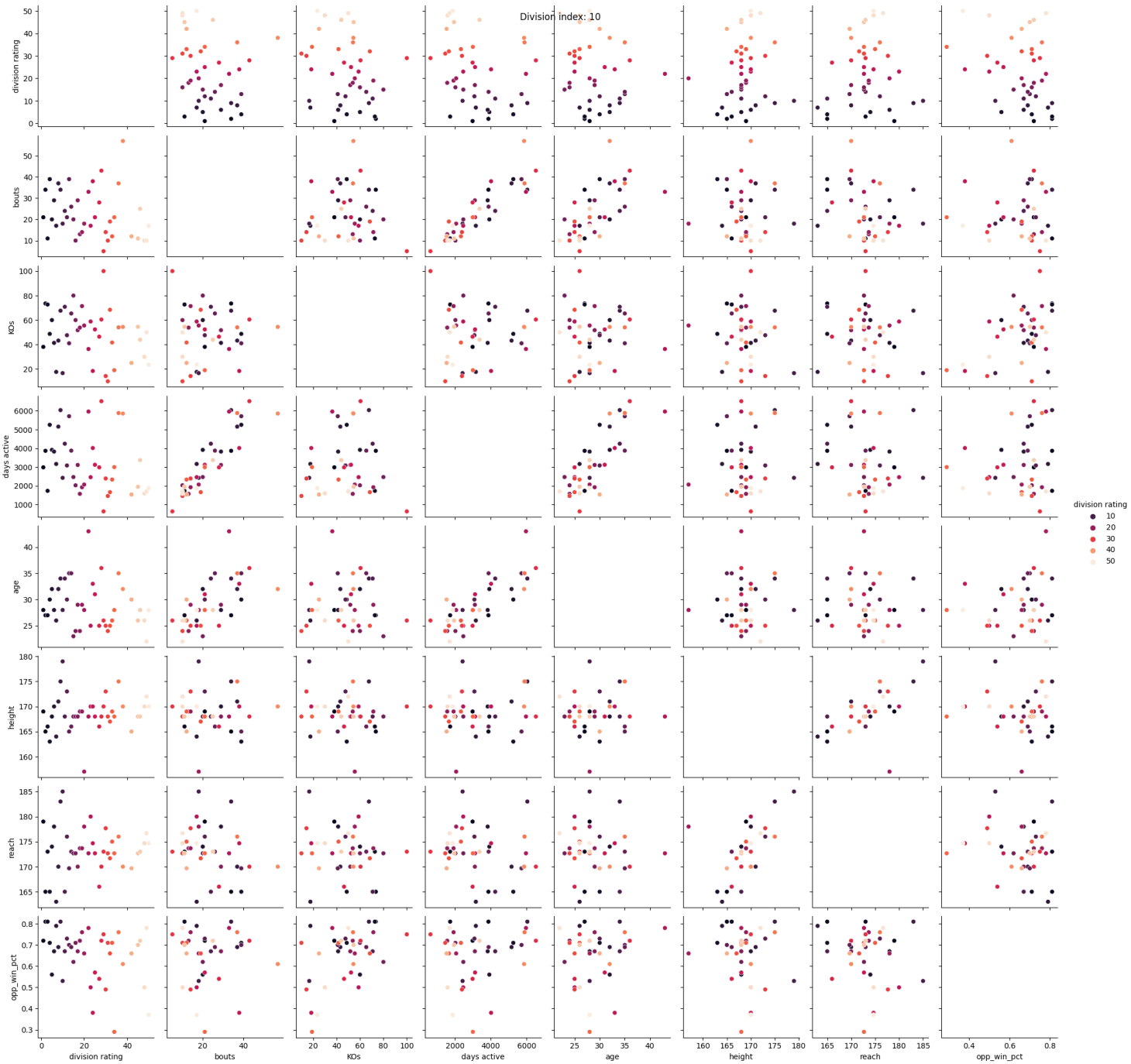

Division index: 5
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
10
20
30
40
50

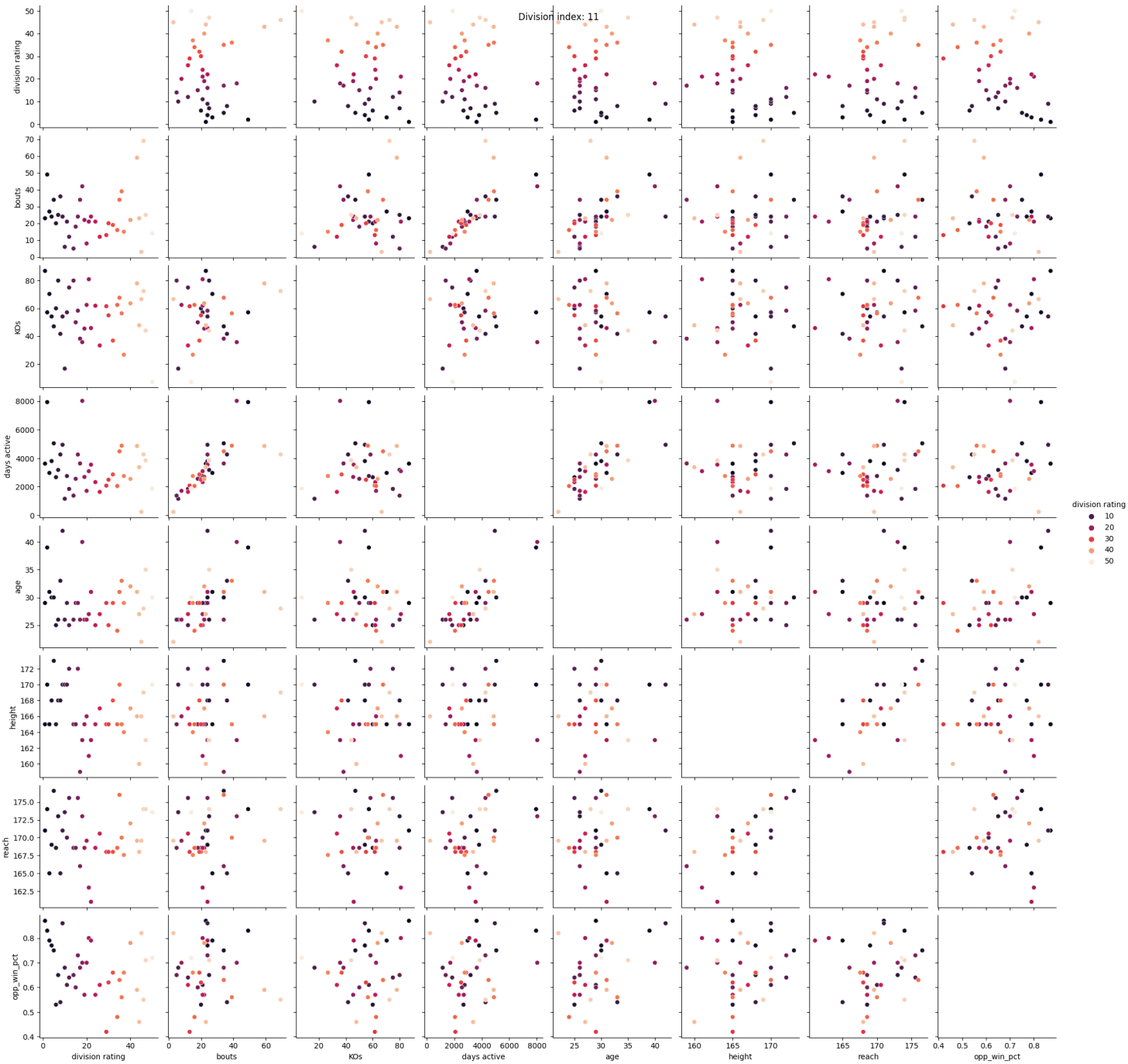

Division index: 6
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
10
20
30
40
50

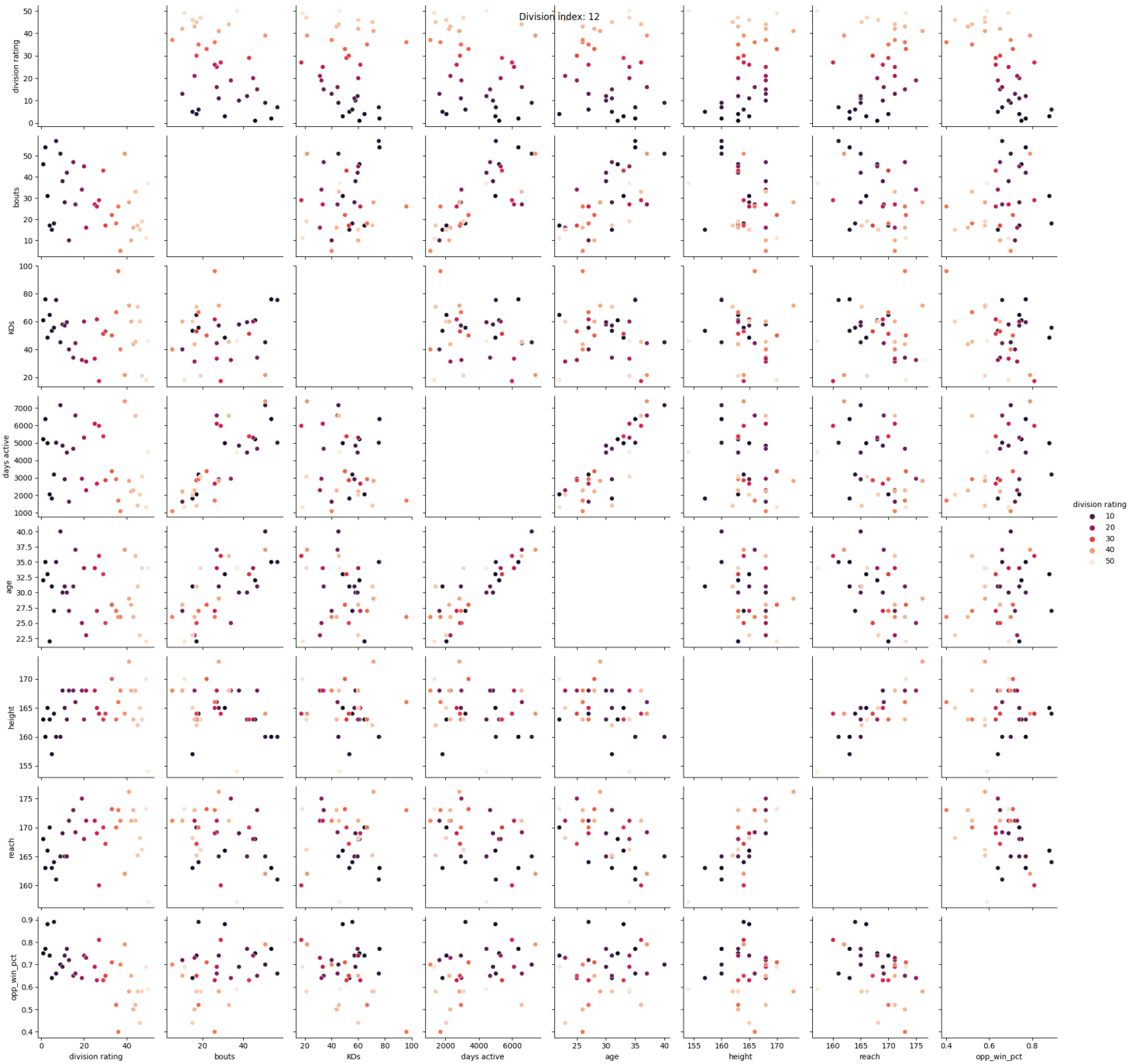

Division index: 7
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
8
16
24
32
40
48

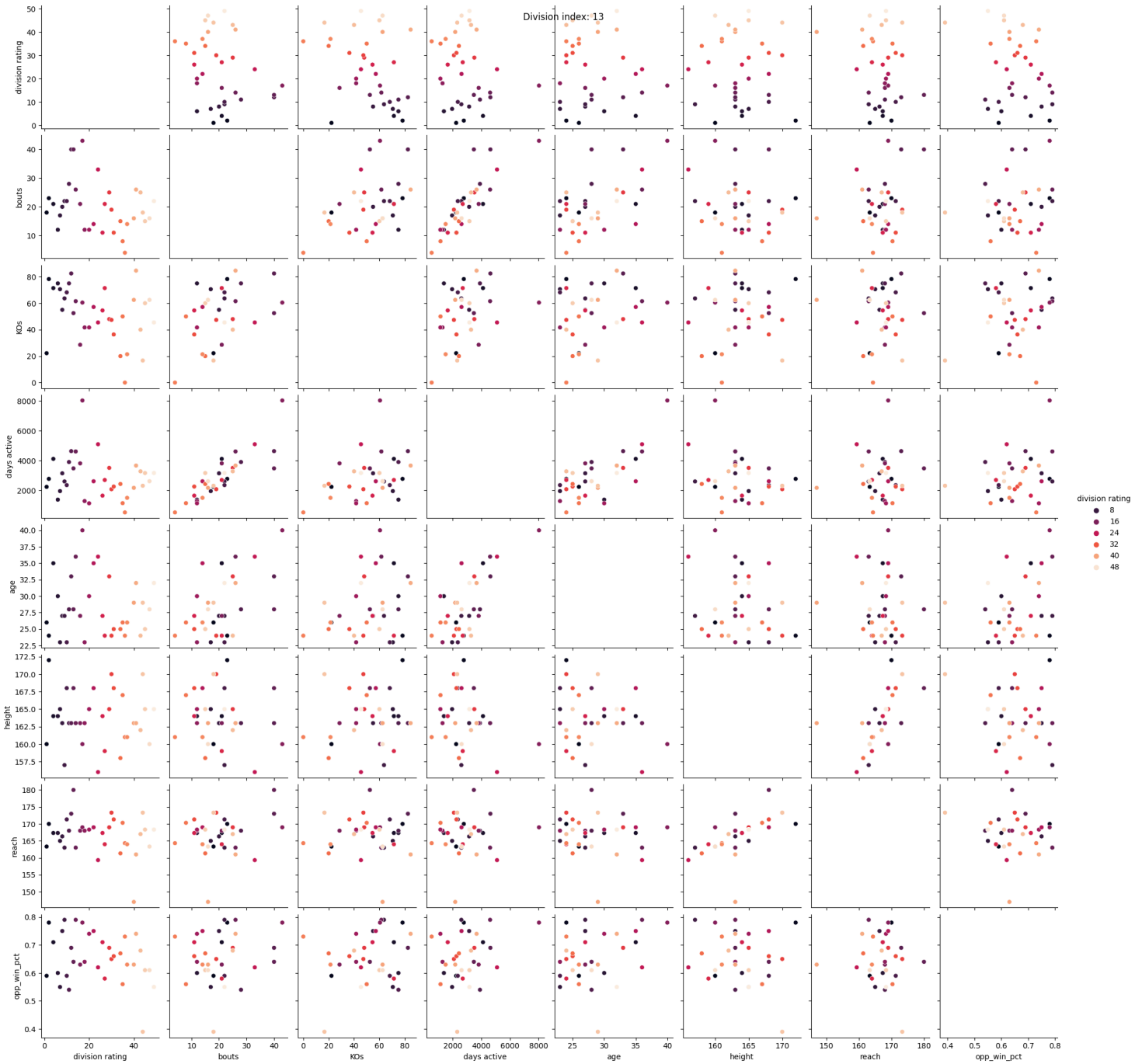

Division index: 8
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
8
16
24
32
40
48

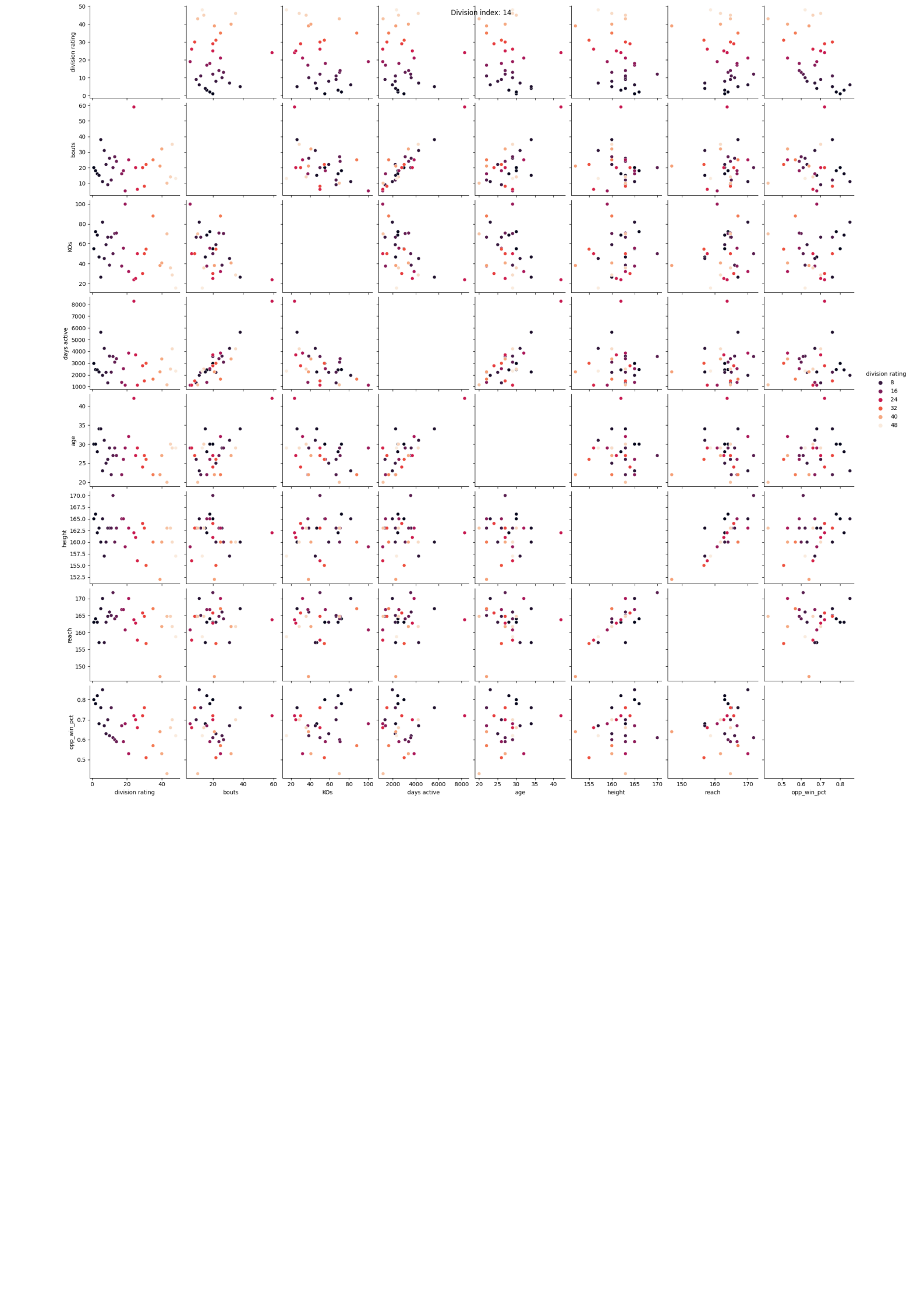

Division index: 9
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
10
20
30
40
50

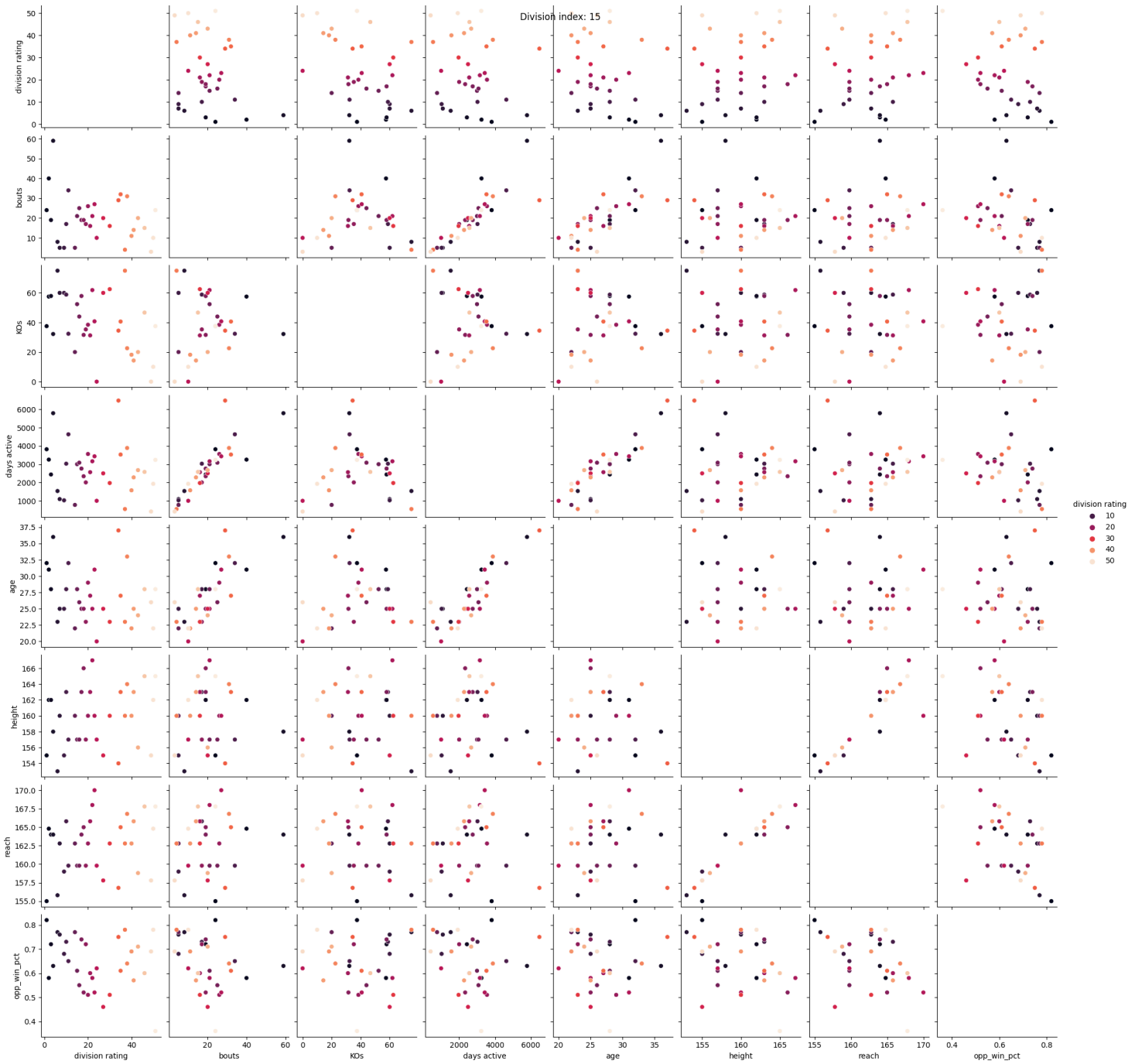

Division index: 10
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
10
20
30
40
50


Division index: 11
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
10
20
30
40
50


Division index: 12
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
10
20
30
40
50

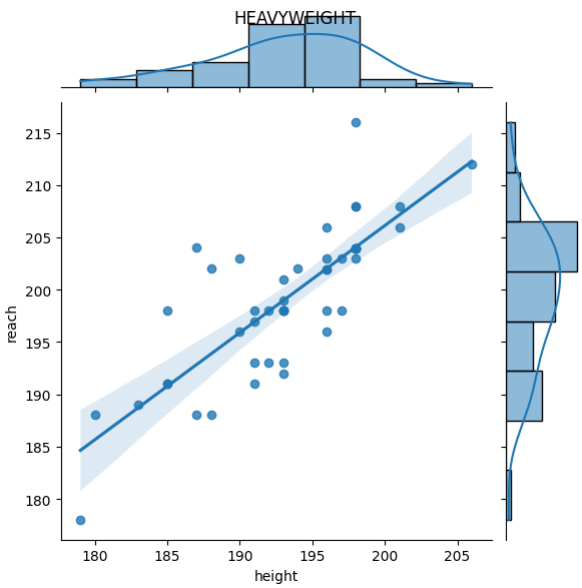

Division index: 13
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
8
16
24
32
40
48

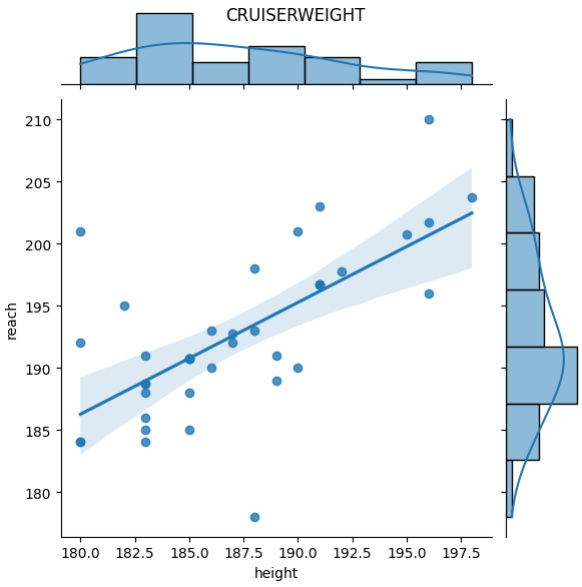

Division index: 14
division rating
bouts
KOs
days active
age
height
reach
opp_win_pct
division rating
8
16
24
32
40
48

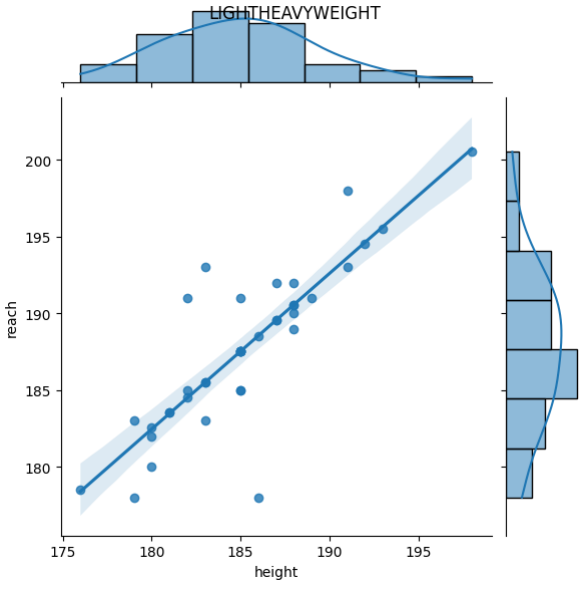


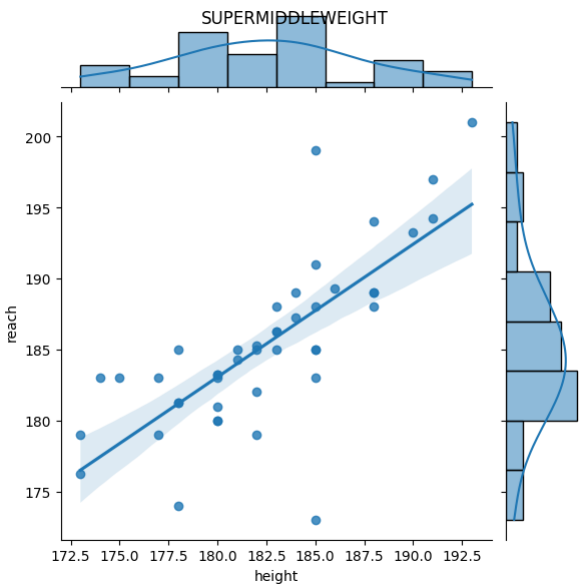
Using the same columns except this time we are seperating them by division to create separate pairplots. Still not many interesting insights!

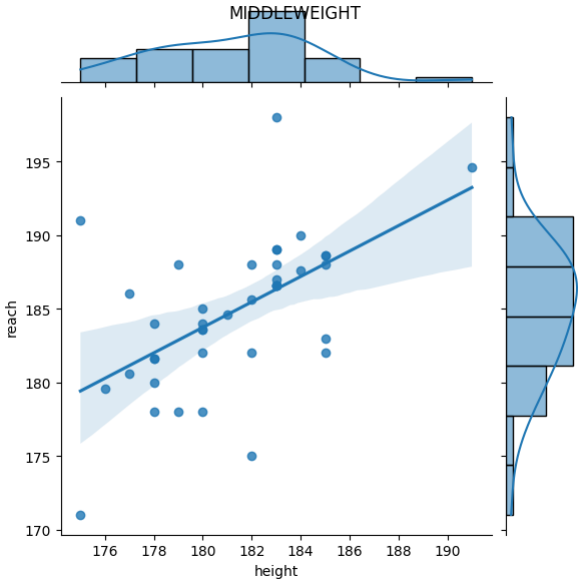
In [ ]:
```
# Loop over the unique values and create a pairplot for each one
for div_index in div_index_values:
    # Select the rows with the current div index value
    df = preproc_ds[preproc_ds['div index'] == div_index]

    # Create the pairplot
    g = sns.jointplot(data=df, x="height", y="reach", kind='reg')

    # Set the title of the plot to the div index value
    g.fig.suptitle(f'{div_list[div_index].upper()}WEIGHT')

    # Show the plot
    plt.show()
```

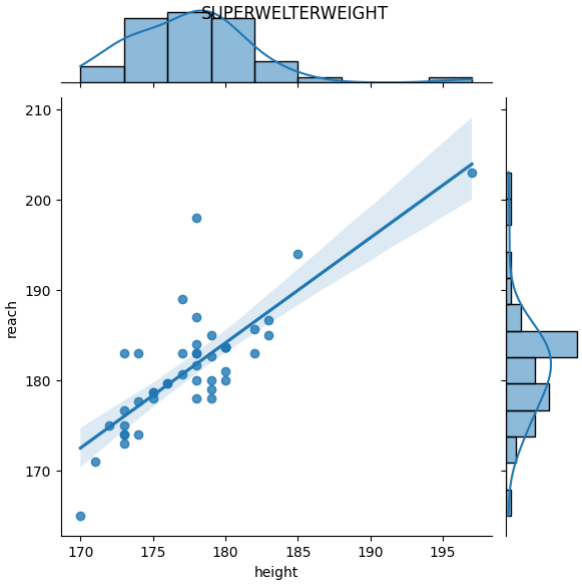

HEAVYWEIGHT
reach
height
180
185
190
195
200
205
210
215

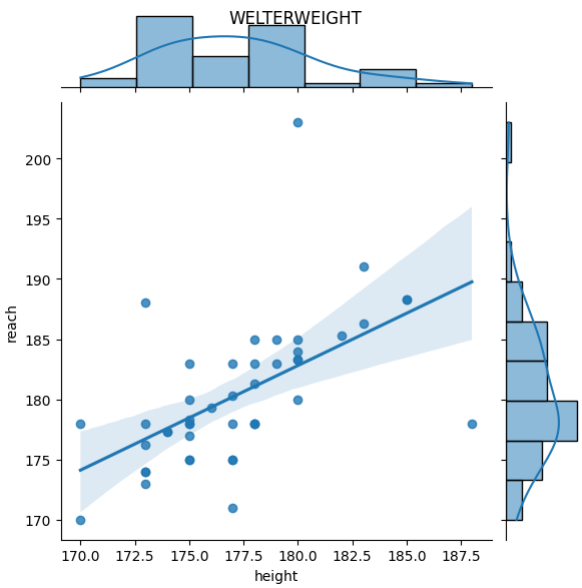

CRUISERWEIGHT
reach
height
180.0
182.5
185.0
187.5
190.0
192.5
195.0
197.5
180
185
190
195
200
205
210

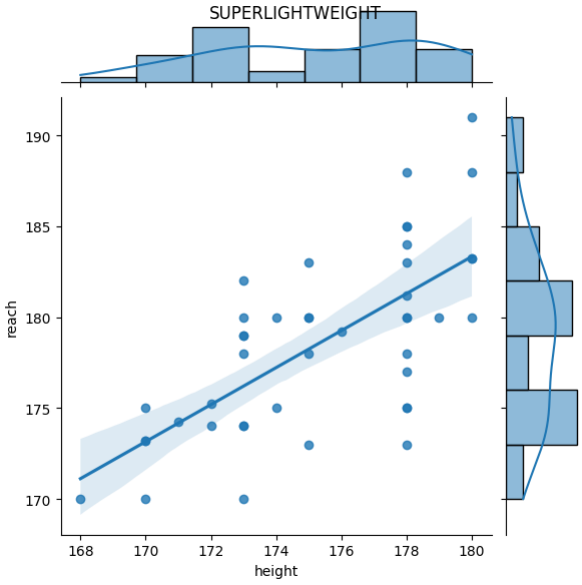

LIGHTHEAVYWEIGHT
reach
height
175
180
185
190
195
180
185
190
195
200

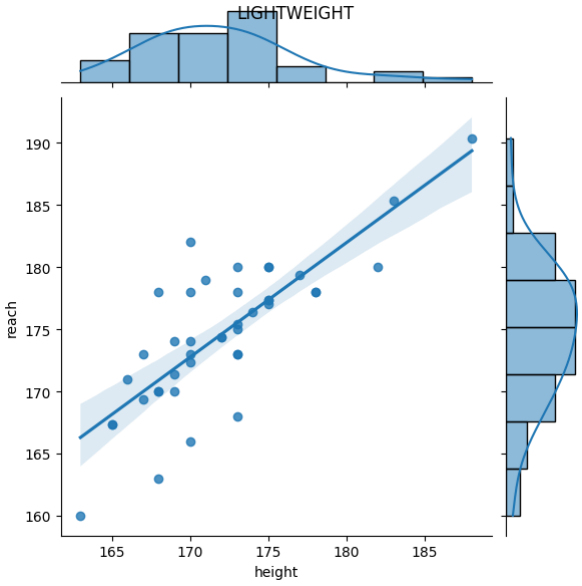

SUPERMIDDLEWEIGHT
reach
height
200
195
190
185
180
175
172.5
175.0
177.5
180.0
182.5
185.0
187.5
190.0
192.5

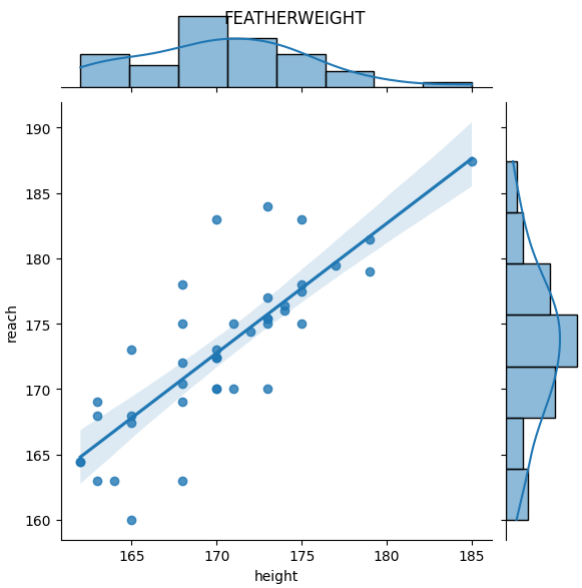

MIDDLEWEIGHT
reach
height
195
190
185
180
175
170
176
178
180
182
184
186
188
190

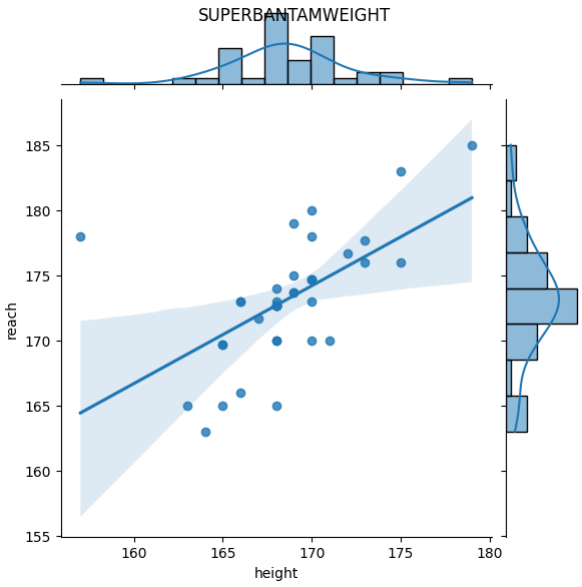

SUPERWELTERWEIGHT
reach
height
210
200
190
180
170
170
175
180
185
190
195

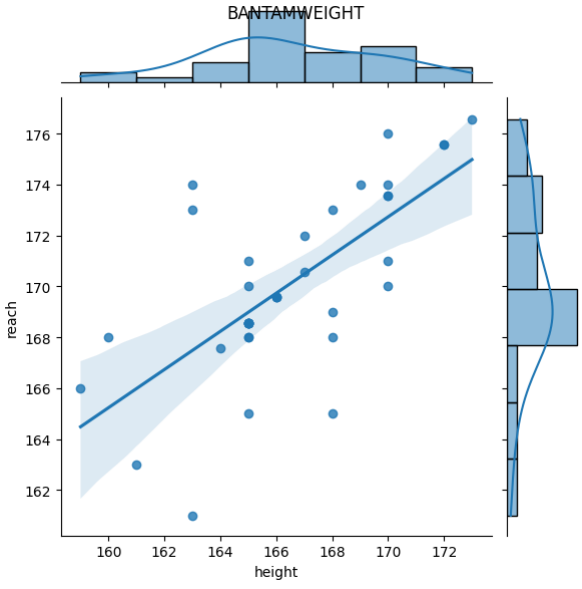

WELTERWEIGHT
reach
height
200
195
190
185
180
175
170
170.0
172.5
175.0
177.5
180.0
182.5
185.0
187.5


SUPERLIGHTWEIGHT
reach
height
190
185
180
175
170
168
170
172
174
176
178
180

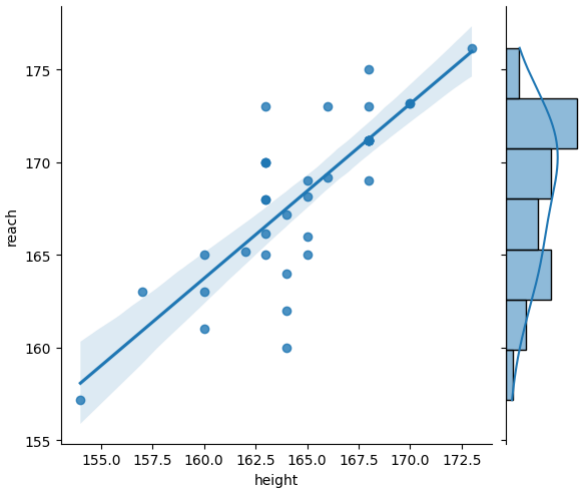

LIGHTWEIGHT
reach
height
190
185
180
175
170
165
160
165
170
175
180
185


FEATHERWEIGHT
reach
height

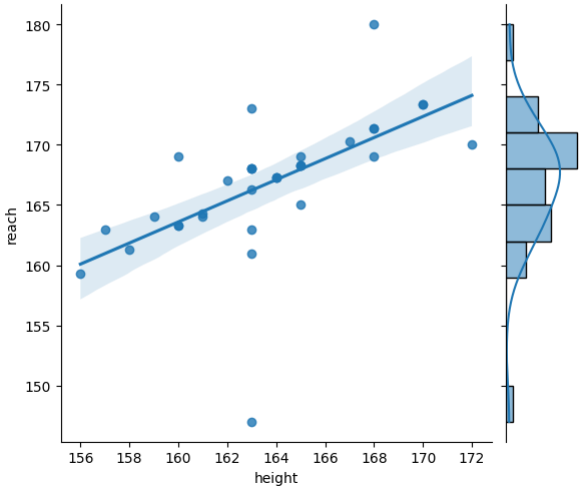

SUPERBANTAMWEIGHT
reach
height


BANTAMWEIGHT
reach
height

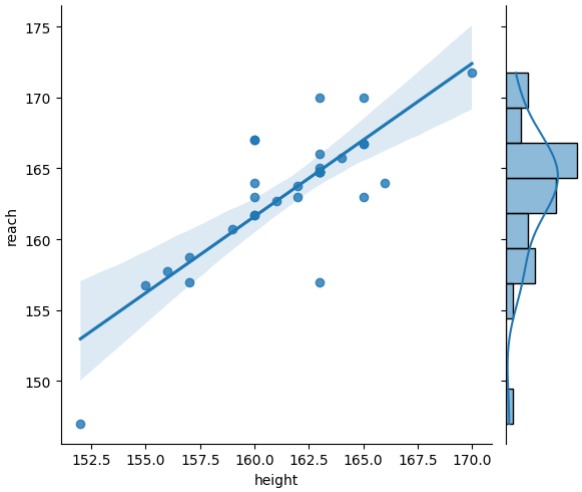
SUPERFLYWEIGHT

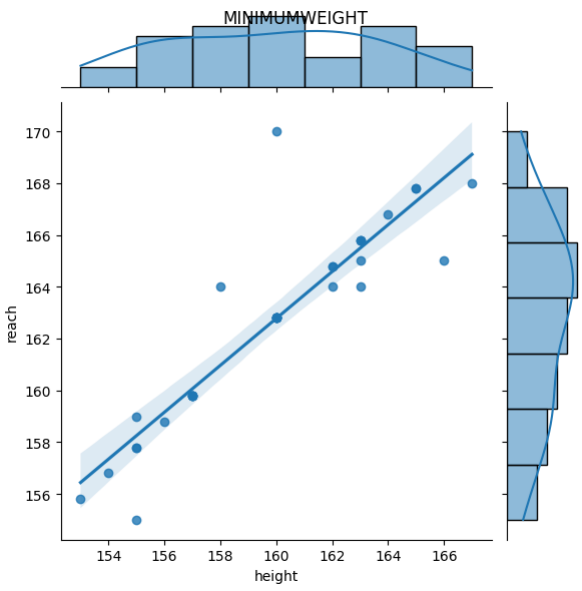


FLYWEIGHT



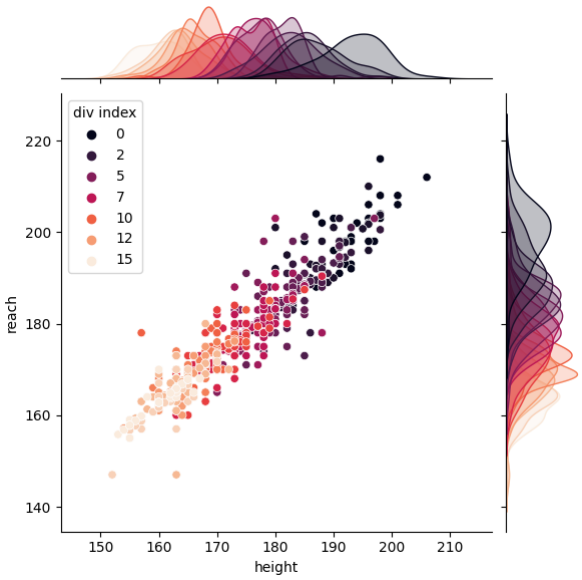
LIGHTFLYWEIGHT





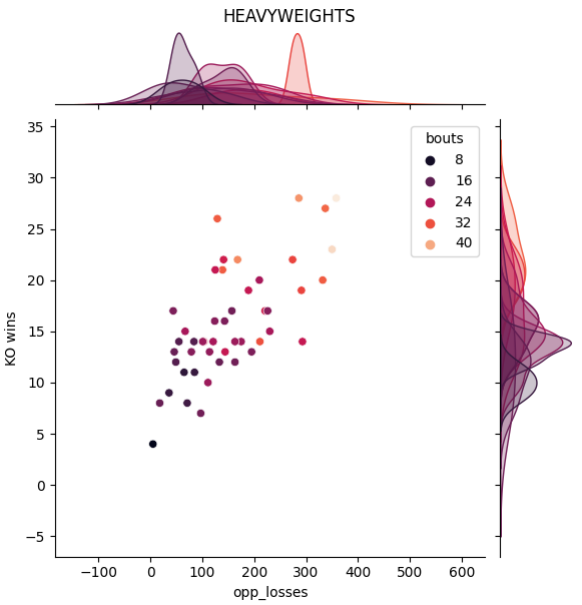
In [ ]:
```
sns.jointplot(data=preproc_ds, x="height", y="reach",hue='div index', palette='rocket')
```

Out[ ]:
```
<seaborn.axisgrid.JointGrid at 0x1aa51983160>
```

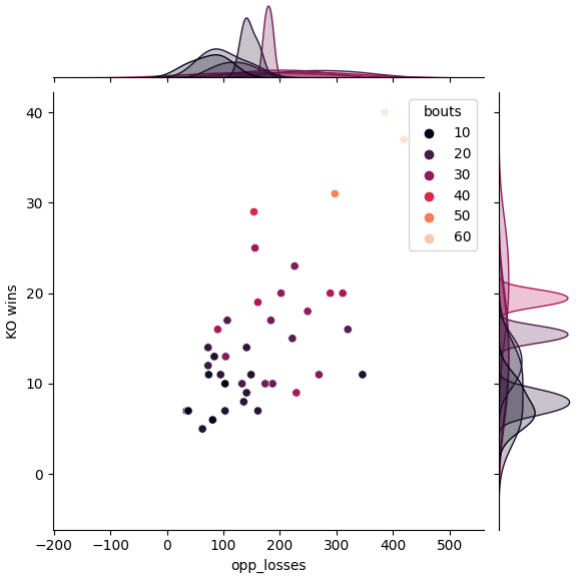


A pretty visualisation using the seaborn library and the jointplot function. Just shows height vs reach, but in a more visually appealing way, nice to learn for the future!

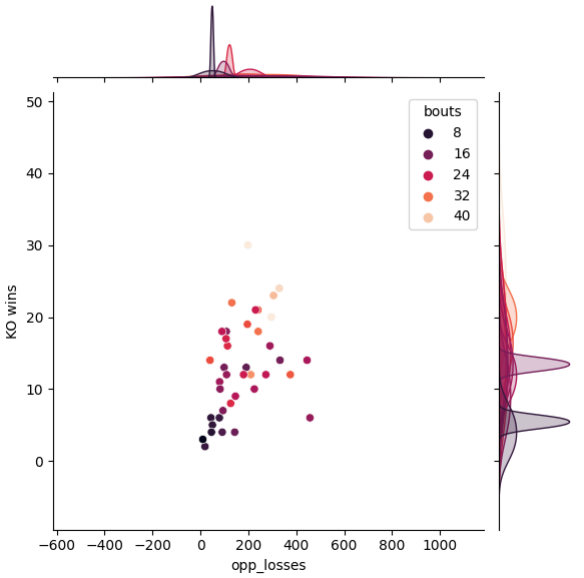
In [ ]:
```
for div_index in div_index_values:
    sns.jointplot(data= preproc_ds.groupby('div index').get_group(div_index), x="opp_losses", y="KO wins",hue='bouts', palette='rocket')
    plt.suptitle(f'{div_list[div_index].upper()}WEIGHTS', va='bottom')
```



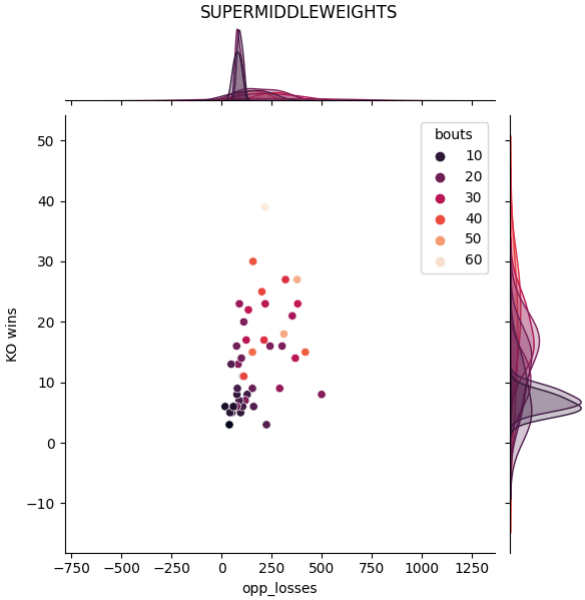
CRUISERWEIGHTS

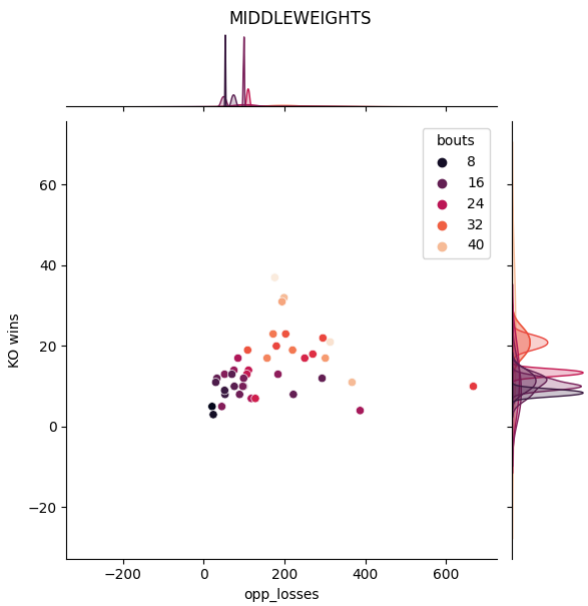


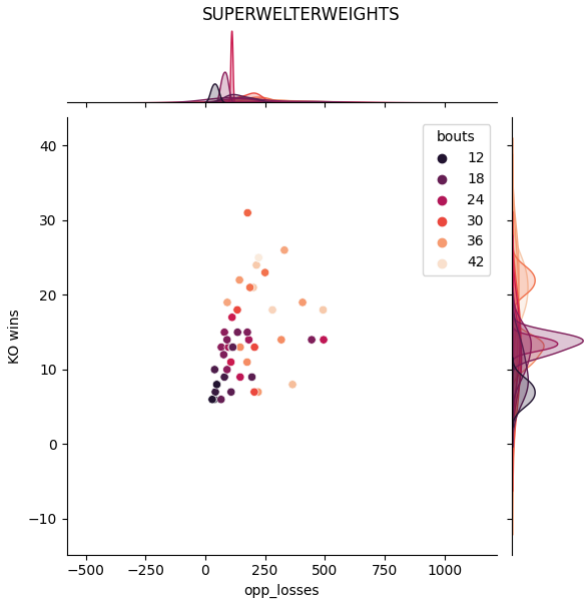
LIGHTHEAVYWEIGHTS

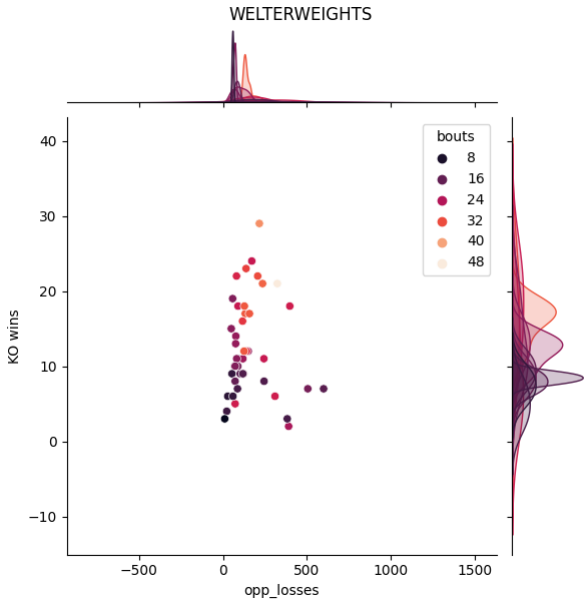


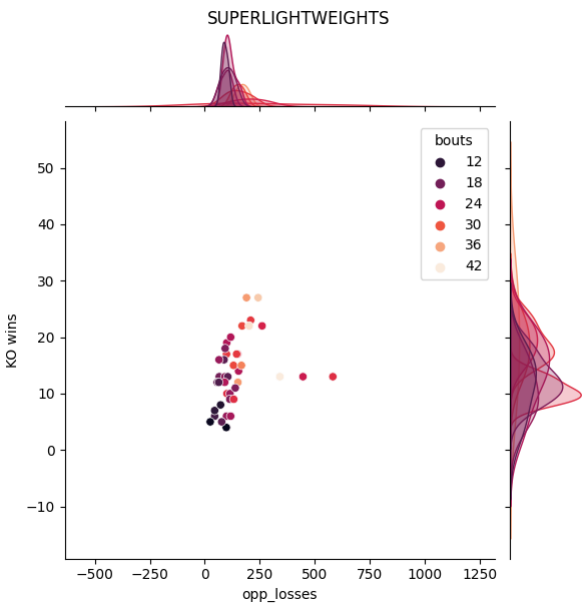
SUPERMIDDLEWEIGHTS

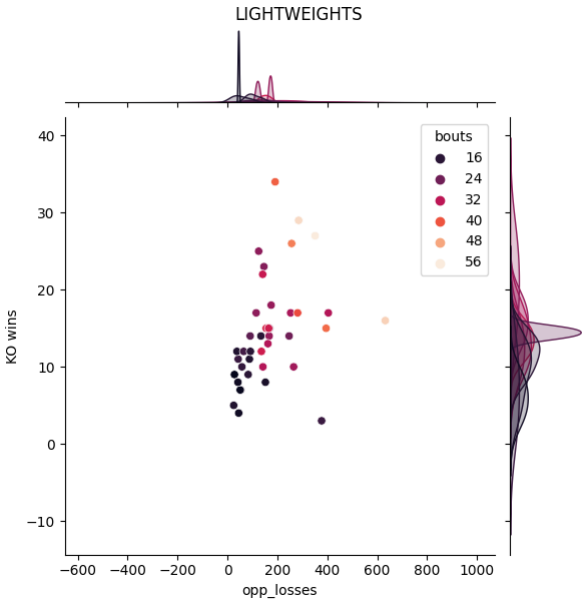


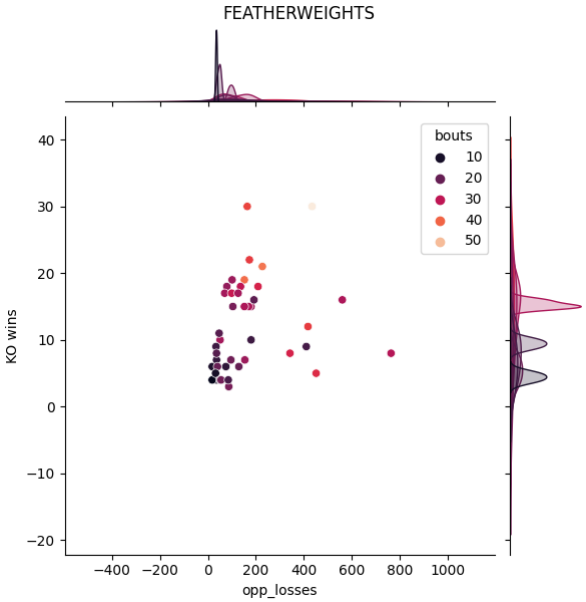

MIDDLEWEIGHTS
bouts
8
16
24
32
40
KO wins
opp_losses

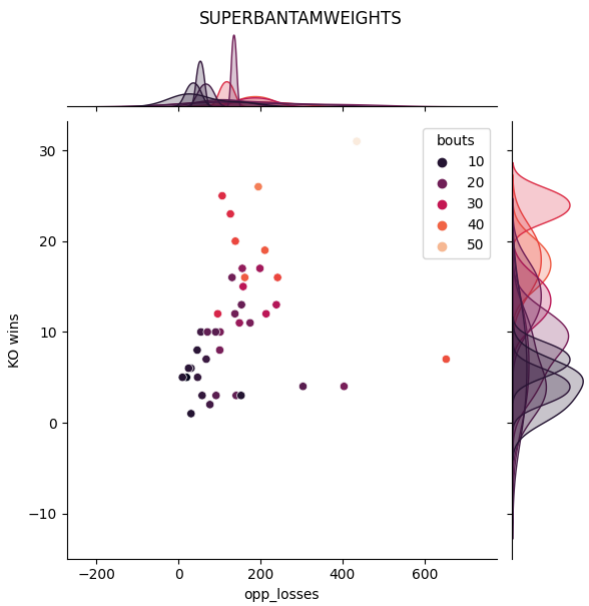

SUPERWELTERWEIGHTS
bouts
12
18
24
30
36
42
KO wins
opp_losses

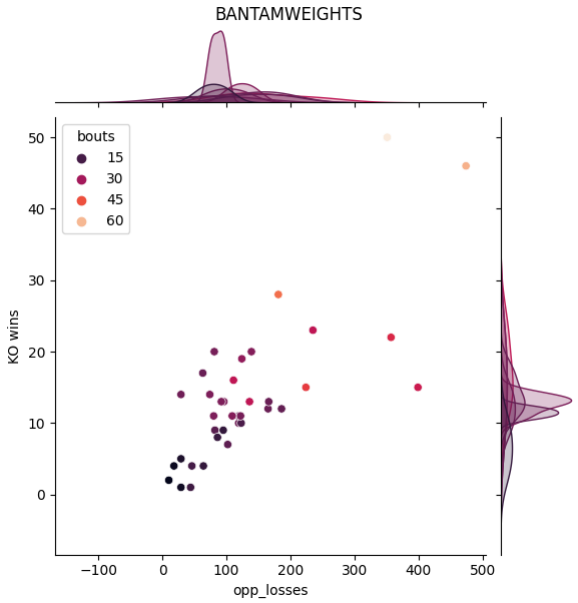

WELTERWEIGHTS
bouts
8
16
24
32
40
48
KO wins
opp_losses

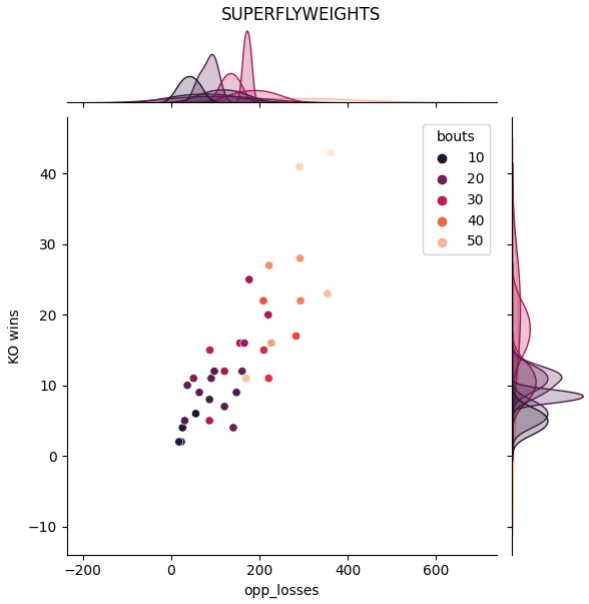

SUPERLIGHTWEIGHTS
bouts
12
18
24
30
36
42
KO wins
opp_losses

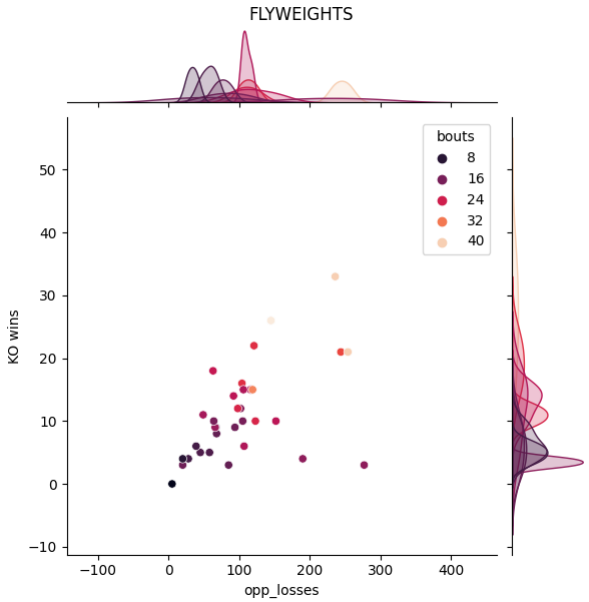

LIGHTWEIGHTS
bouts
16
24
32
40
48
56
KO wins
opp_losses

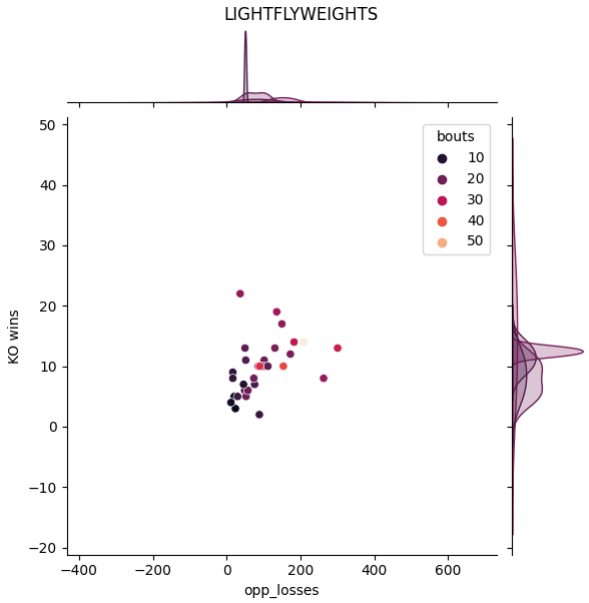

FEATHERWEIGHTS
bouts
10
20
30
40
50
KO wins
opp_losses

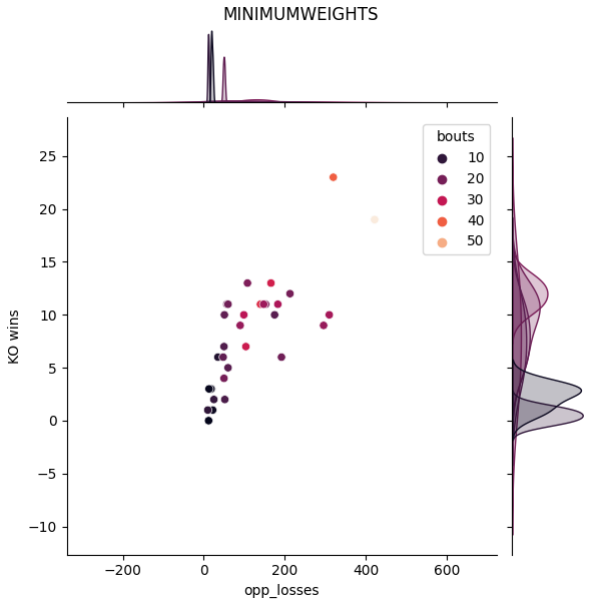

SUPERBANTAMWEIGHTS
bouts
10
20
30
40
50
KO wins
opp_losses


BANTAMWEIGHTS
bouts
15
30
45
60
KO wins
opp_losses


SUPERFLYWEIGHTS
bouts
10
20
30
40
50
KO wins
opp_losses



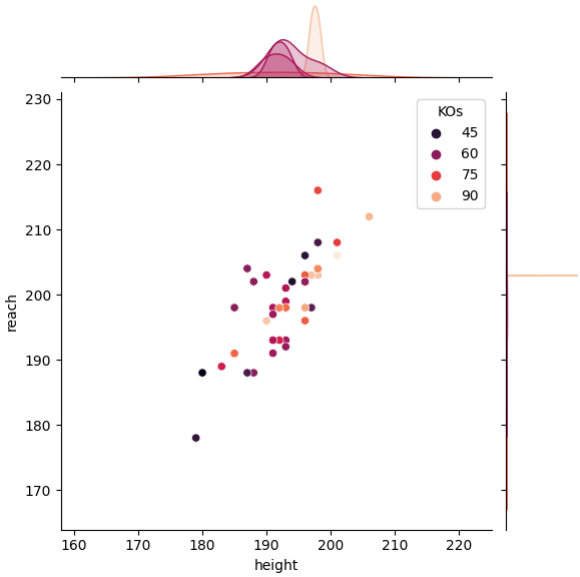




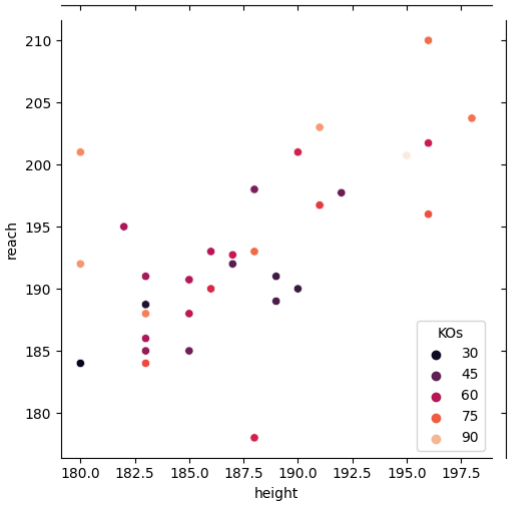
Using the same style of visualisation to see whether we can see any relationships between the number of KO wins of a boxer and their opp_loss value, coloured by the number of bouts they have had. As expected, the more boats, the more KO wins, and more opp_losses.

In [ ]:
```
for div_index in div_index_values:
    sns.jointplot(data= preproc_ds.groupby('div index').get_group(div_index), x="height", y="reach", hue='KOs', palette='rocket')
    plt.suptitle(f'{div_list[div_index].upper()}WEIGHTS', va='bottom')
```

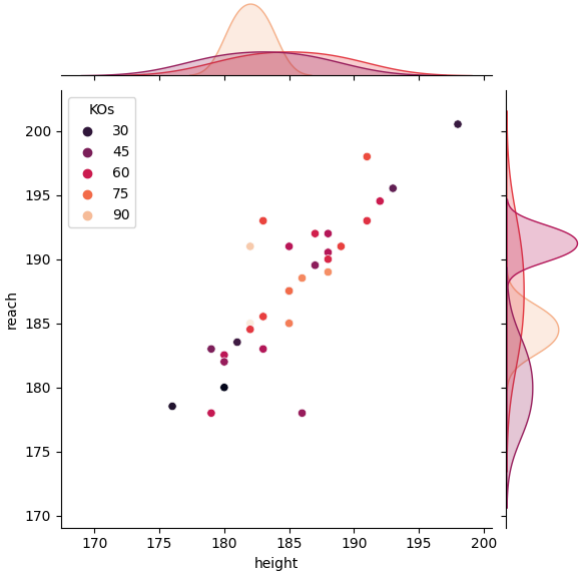
HEAVYWEIGHTS

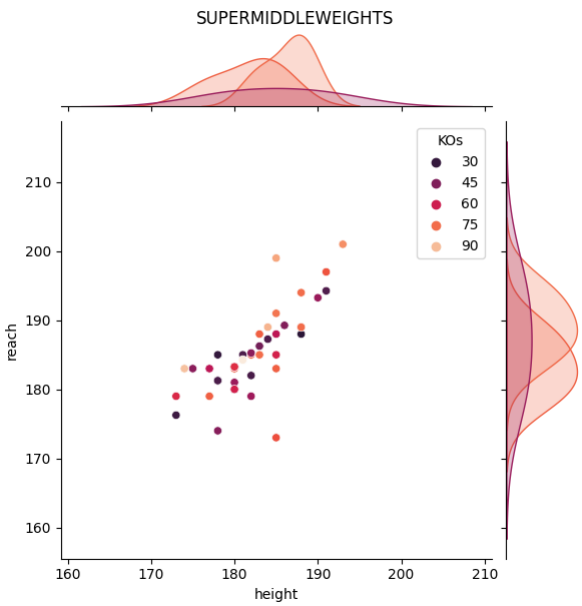


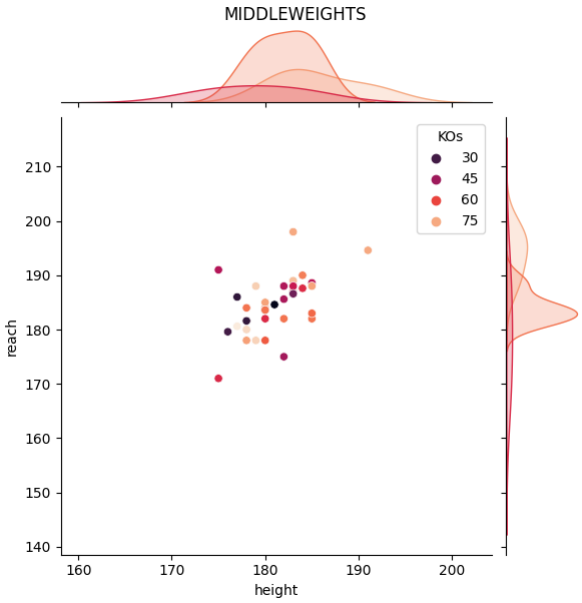
CRUISERWEIGHTS

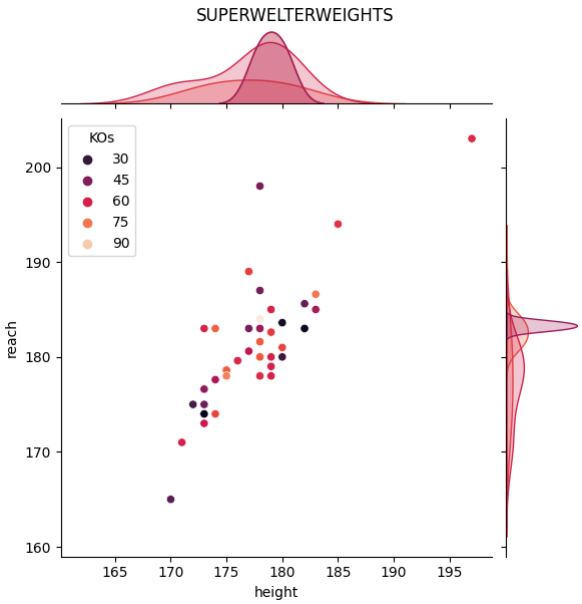


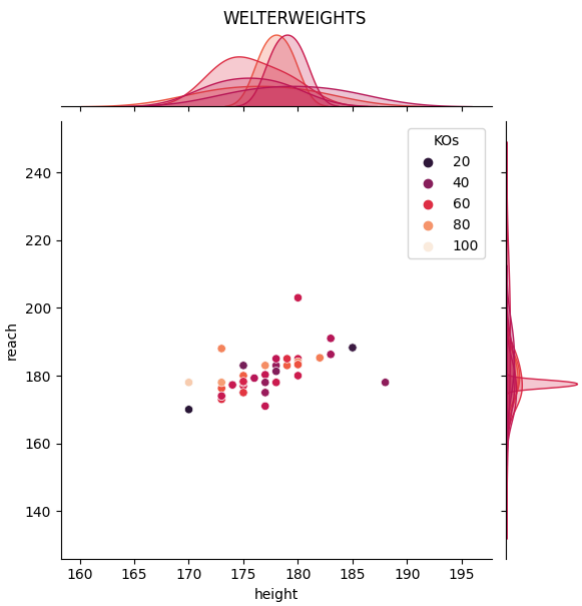
LIGHTHEAVYWEIGHTS

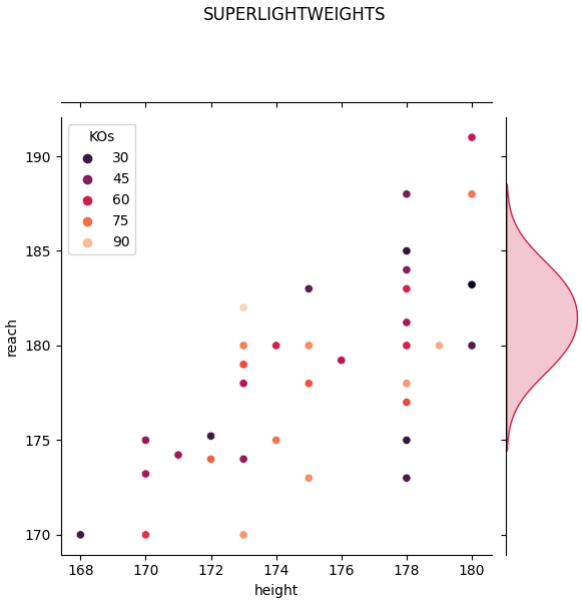


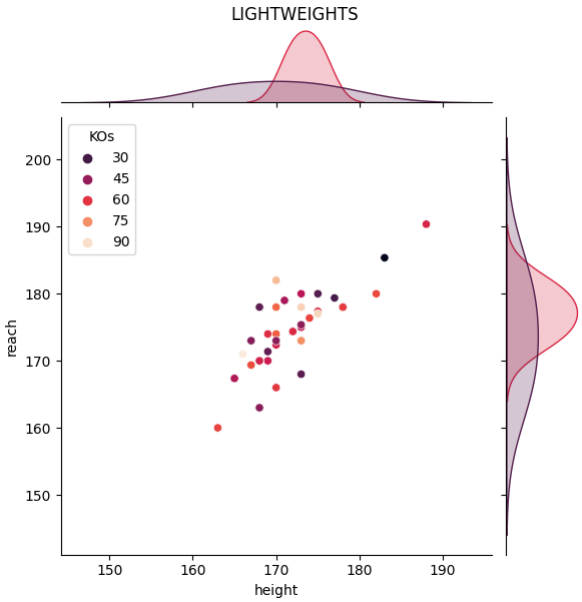

SUPERMIDDLEWEIGHTS
KOs
30
45
60
75
90
reach
height
160
170
180
190
200
210


MIDDLEWEIGHTS
KOs
30
45
60
75
reach
height
140
150
160
170
180
190
200
210

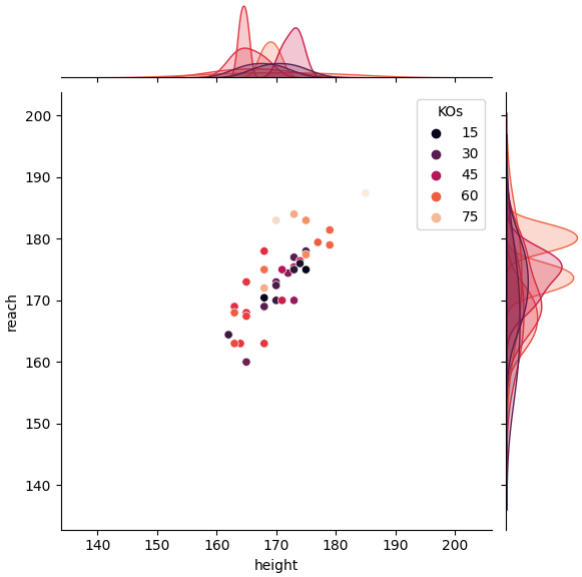

SUPERWELTERWEIGHTS
KOs
30
45
60
75
90
reach
height
160
165
170
175
180
185
190
195
200


WELTERWEIGHTS
KOs
20
40
60
80
100
reach
height

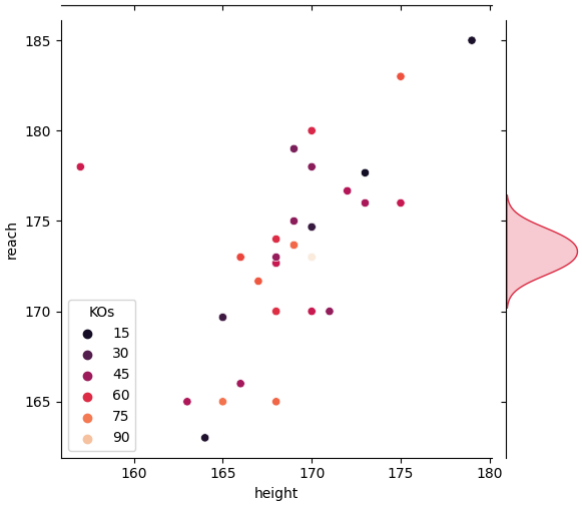

SUPERLIGHTWEIGHTS
KOs
30
45
60
75
90
reach
height


LIGHTWEIGHTS
KOs
30
45
60
75
90
reach
height

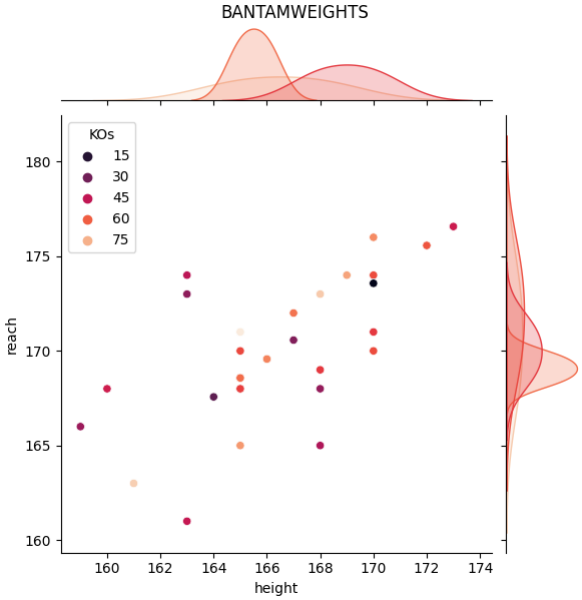
FEATHERWEIGHTS

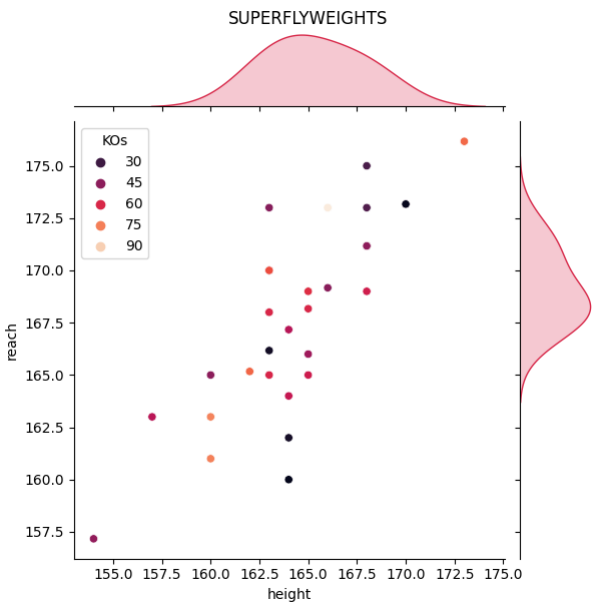


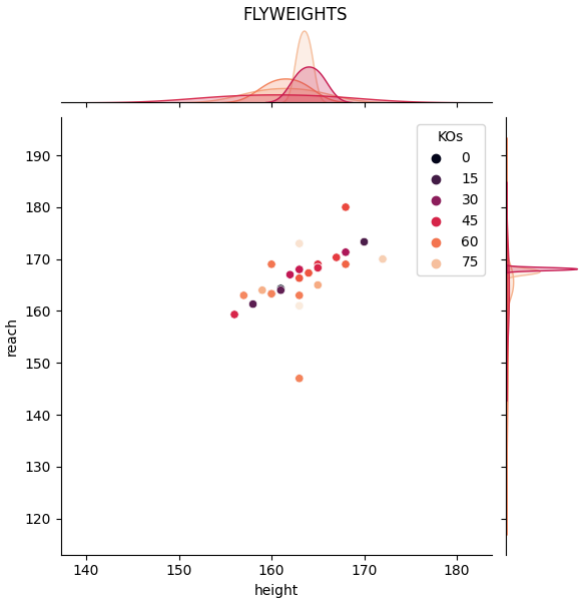
SUPERBANTAMWEIGHTS

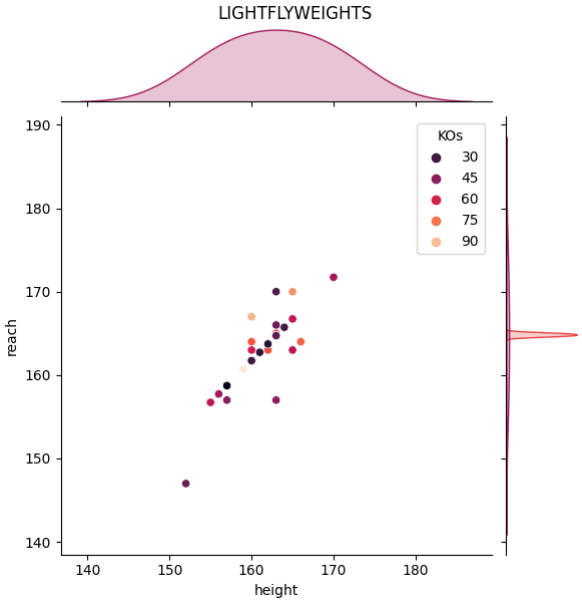


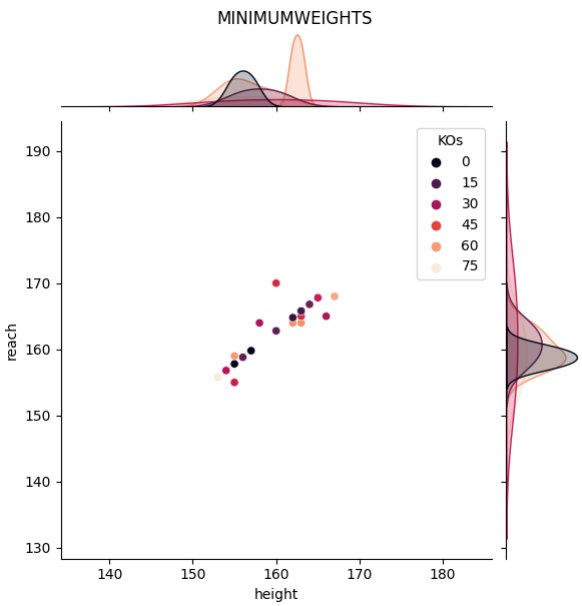
BANTAMWEIGHTS




SUPERFLYWEIGHTS
KOs
30
45
60
75
90
reach
height
175.0
172.5
170.0
167.5
165.0
162.5
160.0
157.5
155.0
157.5
160.0
162.5
165.0
167.5
170.0
172.5
175.0


FLYWEIGHTS
KOs
0
15
30
45
60
75
reach
height
190
180
170
160
150
140
130
120
140
150
160
170
180


LIGHTFLYWEIGHTS
KOs
30
45
60
75
90
reach
height
190
180
170
160
150
140
140
150
160
170
180



Using the same style of visualisation to see whether we can see any relationships between the reach of a boxer and their height, coloured by the KO percentage they have. Nothing too clear emerges from the visualisations, in some instances height and reach values which are higher have higher KO percentage, but this is not uniform across divisions.

In [ ]:
```
no_cluster = preproc_ds.drop(['cluster label'], axis=1)
```

dropping the cluster label column

In [ ]:
```
no_cluster
```

Out[ ]:

	division rating	div index	bouts	rounds	KOs	days active	age	stance	height	reach	wins	losses	draws	KO wins	KO losses	opp_wins	opp_losses	opp_draws	w_minus_l	opp_win_pct
0	1	0	20	168	65.00	3279	35.0	1	191.0	198.0	20	0	0	13	0	473	79	9	394	0.85
1	2	0	27	136	81.48	3328	33.0	0	198.0	208.0	24	3	0	22	1	672	141	11	531	0.82
2	4	0	37	187	59.46	4911	33.0	0	188.0	188.0	35	2	0	22	0	576	168	33	408	0.76
3	5	0	31	170	61.29	4099	35.0	0	193.0	198.0	28	3	0	19	3	567	291	35	276	0.65
4	6	0	15	74	93.33	1754	37.0	0	198.0	203.0	15	0	0	14	0	340	55	6	285	0.86
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
632	43	15	20	104	20.00	2677	24.0	0	156.0	158.8	9	8	3	4	4	126	50	4	76	0.71
633	46	15	15	70	46.67	2578	28.0	0	165.0	167.8	11	3	1	7	2	76	50	8	26	0.60
634	49	15	3	17	0.00	412	26.0	0	155.0	157.8	3	0	0	0	0	27	12	1	15	0.69
635	50	15	10	43	10.00	1927	22.0	1	162.0	164.8	6	4	0	1	2	37	10	1	27	0.78
636	51	15	24	125	37.50	3237	28.0	1	165.0	167.8	19	5	0	9	3	157	296	33	-139	0.36

637 rows × 20 columns

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
```

importing train_test_split and the RandomForestClassifier from sklearn to train a Random Forest model.

In [ ]:
```
groups = no_cluster
```

In [ ]:
```
X = groups.drop(columns='division rating')

# Select target variable
y = groups['division rating']
```

Aiming to predict the division rating of boxers using the data in the dataframe

In [ ]:
```
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Played around with many test_size vaules, 0.3 seemed to provide the highest (still negligable) accuracy score of 0.04.

In [ ]:
```
random_forest_model = RandomForestClassifier(n_estimators=400, random_state=42)
random_forest_model.fit(X_train, y_train)
```

Out[ ]:
```
RandomForestClassifier(n_estimators=400, random_state=42)
```

Played around with the number of estimators and a random state but again, nothing really improved the accuracy of the model.

In [ ]:
```
# Evaluate model
accuracy = random_forest_model.score(X_test, y_test)
print(f'Accuracy: {accuracy:.2f}')
```

```
Accuracy: 0.04
```

Ultimately I believe that the model failed because of a lack of data. If I had used to with a lot more data it may have likely been able to provide a greater accuracy on the test data.