World Triathlon Elite Men Analysis

Jackson Konkin V00910699 SENG 474

Introduction

Triathlon is a sport that requires athletes to excel in three disciplines: swimming, cycling, and running, making it one of the most challenging endurance sports in the world. Elite triathletes are the best in the sport, competing at the highest level, and consistently pushing themselves to achieve new heights.

In this data analysis project, we aim to gain insights into the performance of Elite Men triathletes in World Championship Series events. To achieve this goal, we collected data from the World Triathlon API, which includes information on over 130,000 athletes and 5,000 events.

The World Triathlon API is a comprehensive data source that provides detailed information on triathlon events, athletes, and their performances. This API includes data on athlete details such as name, age, country, region, year, time, splits, and number of athletes, as well as event results for every competing athlete.

We believe that by processing and visualizing this data, we can gain a better understanding of the trends and patterns in Elite Men triathlon performances. We also plan to use machine learning algorithms to predict athlete performance and categorize them based on their abilities, as well as to identify factors that contribute to success in the sport.

As a triathlete myself, I possess a solid understanding of the sport, which helped me to determine which data was relevant and which was not. Throughout the data mining cycle, I will explain the reasons behind the choices I make, and include additional insights and observations that can be drawn from the data.

Our ultimate goal is to use the insights gained from this analysis to help Elite Men triathletes improve their performances and achieve their full potential. By identifying key performance indicators and factors that contribute to success, we aim to provide valuable information that can be used to develop training programs and strategies for athletes and coaches alike.

Data Collection

The project's data was sourced from the World Triathlon API, which contained information on over 130,000 athletes and 5,000 events. This data included athlete details such as name, age, country, region, year, time, splits, and number of athletes, as well as event results for every competing athlete.

Due to the vast amount of data available, it was necessary to narrow the focus of the analysis. In order to achieve this, the analysis only included athletes classified as Elite Men and events classified as World Championship Series. This decision was made to ensure that the machine learning algorithms were applied to the professional level of the sport, with the intention of subsequently applying them to Elite Women.

Here is a link to the API:

https://developers.triathlon.org/reference/live-timing

Data pre-processing and visualization

Imports

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

# suppress warnings
import warnings

warnings.filterwarnings('ignore')
```

Past Winners

To get started, let's take a look at all the past winners from 2009 up to the most recent race in Montreal. We will be examining the results of Elite Men in races classified as World Triathlon Championship Series races, which is the highest level of racing in the sport.

The first step is to make an api call and retreive a list of all the past WTCS races. We will store it in a JSON object called data.

```
# Set the headers to authenticate with the API
headers = {
    "accept": "application/json",
    "apikey": "2649776ef9ece4c391003b521cbfce7a"
}

# Get the data from the API
url = "https://api.triathlon.org/v1/events?
per_page=1000&category_id=351&order=asc"
response = requests.get(url, headers=headers).text

# Convert the data to a JSON object
data = json.loads(response)['data']

# Preview the data
data[0]
```


```
{'event_id': 5185,
 'event_title': '2009 Dextro Energy Triathlon - ITU World Championship
Series Tongyeong',
 'event_slug': '2009_dextro_energy_triathlon_-
```

```
_itu_world_championship_series_tongyeong',
 'event_edit_date': '2012-11-23T11:23:07+00:00',
 'event_venue': 'Tongyeong',
 'event_country': 'Korea, South',
 'event_latitude': 34.8544227,
 'event_longitude': 128.433182,
 'event_date': '2009-05-02',
 'event_finish_date': '2009-05-03',
 'event_country_isoa2': 'KR',
 'event_country_noc': 'KOR',
 'event_region_id': 13,
 'event_country_id': 198,
 'event_region_name': 'Asia',
 'event_website':
'https://web.archive.org/web/20091231100000/www.triathlon.or.kr',
 'event_status': 'Scheduled',
 'triathlonlive': False,
 'event_categories': [{'cat_name': 'World Championship Series',
   'cat_id': 351,
   'cat_parent_id': None}],
 'event_specifications': [{'cat_name': 'Triathlon',
   'cat_id': 357,
   'cat_parent_id': None},
  {'cat_name': 'Standard', 'cat_id': 377, 'cat_parent_id': 357}],
 'event_flag':
'https://triathlon-images.imgix.net/images/icons/kr.png',
 'event_listing':
'https://www.triathlon.org/events/event/2009_dextro_energy_triathlon_-
_itu_world_championship_series_tongyeong',
 'event_api_listing': 'https://api.triathlon.org/v1/events/5185'}
```


Next step is to retrieve some specific information from each event, most importantly the event_id. For each event, we make another api call using the event_id to retrieve the program id, which we will later use to retrieve the results.

```python
# get list of events name, event date, event id, and program id and
store in events list
events = []
for event in data:

    # Get the data from the API
    url = "https://api.triathlon.org/v1/events/" +
str(event['event_id']) + "/programs?is_race=true"
    response = requests.get(url, headers=headers)

    # convert response to json
    programData = response.json()["data"]

    # get the event name
    eventName = event['event_title']
```

```
    # get the event date
    eventDate = event['event_date']

    # go through each program and add the event name, event date, 
event id, and program id to the events list
    for program in programData:
        if program['prog_name'] == "Elite Men":
          # add tuple of event name, event id, and program id to 
events list
          events.append((eventName, eventDate, event['event_id'], 
program['prog_id']))

# Preview the events list
events[0]
```

```
('2009 Dextro Energy Triathlon - ITU World Championship Series 
Tongyeong',
 '2009-05-02',
 5185,
 4521)
```

With a list of every event and relevant information for each event, we now must make an api call using both the event and program id to retrieve the results from each event. We can then grab all the information we need and store it in a dataframe.

```
# create a list for the winner's data
winnerData = []

# go through each event and get the winner's data
for eventName, eventDate, event_id, program_id in events:

    # Get the data from the API
    url = "https://api.triathlon.org/v1/events/" + str(event_id) + 
"/programs/" + str(program_id) + "/results?per_page=1000"
    response = requests.get(url, headers=headers)

    # convert response to json
    resultsData = response.json()["data"]

    # check if results is empty
    if len(resultsData['results']) == 0:
        continue

    # get the winner of the event
    winner = resultsData['results'][0]

    # get the winner's id, name, age, splits, position, and time
    winner_id = winner['athlete_id']
    winner_name = winner['athlete_title']
```

```
    winner_age = winner['athlete_age']
    winner_splits = winner['splits']

    # separate the winner's splits into swim, t1, bike, t2, and run
    winner_swim = winner_splits[0]
    winner_t1 = winner_splits[1]
    winner_bike = winner_splits[2]
    winner_t2 = winner_splits[3]
    winner_run = winner_splits[4]

    # get the winner's total time
    winner_time = winner['total_time']

    # remove the commas from the event name
    eventName = eventName.replace(",", "")

    # add the winner's data to the winnerData list as a dictionary
    winnerData.append({
        "Event Name": eventName,
        "Event Date": eventDate,
        "Winner ID": winner_id,
        "Winner Name": winner_name,
        "Winner Age": winner_age,
        "Winner Swim": winner_swim,
        "Winner T1": winner_t1,
        "Winner Bike": winner_bike,
        "Winner T2": winner_t2,
        "Winner Run": winner_run,
        "Winner Time": winner_time
    })

# create a dataframe from the winnerData list
winner_df = pd.DataFrame(winnerData)

# preview the dataframe
winner_df.head()
```

```
                                          Event Name  Event Date  
Winner ID
0  2009 Dextro Energy Triathlon - ITU World Champ...  2009-05-02  
5297  \
1  2009 Dextro Energy Triathlon - ITU World Champ...  2009-05-31  
7788
2  2009 Dextro Energy Triathlon - ITU World Champ...  2009-06-21  
7788
3  2009 Dextro Energy Triathlon - ITU World Champ...  2009-07-11  
7788
4  2009 Dextro Energy Triathlon - ITU World Champ...  2009-07-25  
6442
```

```
         Winner Name  Winner Age Winner Swim Winner T1 Winner Bike
Winner T2
0     Bevan Docherty          46    00:18:36  00:00:48    01:00:18
00:00:24  \
1  Alistair Brownlee          35    00:17:57  00:01:08    01:01:27
00:00:26
2  Alistair Brownlee          35    00:20:06  00:00:33    00:57:01
00:00:20
3  Alistair Brownlee          35    00:16:15  00:00:53    00:55:07
00:00:24
4   Jarrod Shoemaker          40    00:16:40  00:00:44    00:56:46
00:00:20

  Winner Run Winner Time
0   00:30:20    01:50:25
1   00:30:31    01:51:26
2   00:31:00    01:48:58
3   00:30:36    01:43:13
4   00:29:37    01:44:06
```

We will want a new column with just the year so we can look at trends over the years and see if the times have changed at all

```
# create new attribute year from event date
winner_df['Year'] = winner_df['Event Date'].str[:4]

# look at the shape of the dataframe
winner_df.shape
```

```
(109, 12)
```

We need to do some data cleaning and processing if we want to do any analysis as the times are curently stored as strings. The easiest way to do this will be to convert all times to seconds, that way it is the same format over each discipline.

```
# convert the time to seconds for each split and total time
# swim
winner_df['Winner Swim'] = pd.to_datetime(winner_df['Winner Swim'],
format='%H:%M:%S')
winner_df['Winner Swim'] = winner_df['Winner Swim'].dt.hour * 3600 +
winner_df['Winner Swim'].dt.minute * 60 + winner_df['Winner
Swim'].dt.second

# t1
winner_df['Winner T1'] = pd.to_datetime(winner_df['Winner T1'],
format='%H:%M:%S')
winner_df['Winner T1'] = winner_df['Winner T1'].dt.hour * 3600 +
winner_df['Winner T1'].dt.minute * 60 + winner_df['Winner
T1'].dt.second
```

```python
# bike
winner_df['Winner Bike'] = pd.to_datetime(winner_df['Winner Bike'], format='%H:%M:%S')
winner_df['Winner Bike'] = winner_df['Winner Bike'].dt.hour * 3600 + winner_df['Winner Bike'].dt.minute * 60 + winner_df['Winner Bike'].dt.second

# t2
winner_df['Winner T2'] = pd.to_datetime(winner_df['Winner T2'], format='%H:%M:%S')
winner_df['Winner T2'] = winner_df['Winner T2'].dt.hour * 3600 + winner_df['Winner T2'].dt.minute * 60 + winner_df['Winner T2'].dt.second

# run
winner_df['Winner Run'] = pd.to_datetime(winner_df['Winner Run'], format='%H:%M:%S')
winner_df['Winner Run'] = winner_df['Winner Run'].dt.hour * 3600 + winner_df['Winner Run'].dt.minute * 60 + winner_df['Winner Run'].dt.second

# total time
winner_df['Winner Time'] = pd.to_datetime(winner_df['Winner Time'], format='%H:%M:%S')
winner_df['Winner Time'] = winner_df['Winner Time'].dt.hour * 3600 + winner_df['Winner Time'].dt.minute * 60 + winner_df['Winner Time'].dt.second

# convert the year to an integer
winner_df['Year'] = winner_df['Year'].astype(int)

# look at the data types
winner_df.dtypes
```

```
Event Name     object
Event Date     object
Winner ID       int64
Winner Name    object
Winner Age      int64
Winner Swim     int32
Winner T1       int32
Winner Bike     int32
Winner T2       int32
Winner Run      int32
Winner Time     int32
Year            int32
dtype: object
```

We are almost ready to use the data we collected and gain some insights on the past winners but there is just one more thing to do. In triathlon there are different race distances, standard, sprint, and super sprint, so we will need to create a new attribute based on the overall time.

```
# add new attribute called race type
# if total time is < 2000 race type is superSprint
# if 2000 < toatl time < 5000 race type is sprint
# if total time > 5000 race type is olympic

# create a list of our conditions
conditions = [
    (winner_df['Winner Time'] < 2000),
    (winner_df['Winner Time'] >= 2000) & (winner_df['Winner Time'] <
5000),
    (winner_df['Winner Time'] >= 5000)
    ]

# create a list of the values we want to assign for each condition
values = ['Super Sprint', 'Sprint', 'Olympic']

# create a new column and use np.select to assign values to it using
our lists as arguments
winner_df['Type'] = np.select(conditions, values)
```

Now lets look at the average winner times by each year for swim, bike, run, and overall.

```
# plot average winner time, swim, bike, and run time by year on
different graphs using subplots
figure, axes = plt.subplots(2, 2, figsize=(15,10))

# average winner time for olympic on top left
winner_df[winner_df['Type'] == 'Olympic'].groupby('Year')['Winner
Time'].mean().plot(ax=axes[0,0], title='Average Winner Time by Year')

# average swim time for olympic on top right
winner_df[winner_df['Type'] == 'Olympic'].groupby('Year')['Winner
Swim'].mean().plot(ax=axes[0,1], title='Average Swim Time by Year')

# average bike time for olympic on bottom left
winner_df[winner_df['Type'] == 'Olympic'].groupby('Year')['Winner
Bike'].mean().plot(ax=axes[1,0], title='Average Bike Time by Year')

# average run time for olympic on bottom right
winner_df[winner_df['Type'] == 'Olympic'].groupby('Year')['Winner
Run'].mean().plot(ax=axes[1,1], title='Average Run Time by Year')
```

```
<Axes: title={'center': 'Average Run Time by Year'}, xlabel='Year'>
```

Now we can make some insights from these graphs. It is clear to see that the avarage overall times and bike times have gotten faster. This makes sense as the bike technology has improved tremendously over the past 10 years. The swim times seem to be relatively similar. The run times seem to have gotten dramatically faster since about 2016, the year nike released its super shoe which saves loads of time.

Total time compared to swim, bike, and run times for each position

Lets now look at how the swim, bike, and run splits compare to the overall time for each position (1st, 2nd, etc...) over all WTCS Elite Men races

We will need to get the average times for each position over all the events.

```
# create a dictionary to store the average times for each position
time_dict = {}

# loop through each event
for eventName, eventDate, event_id, program_id in events:

    # Get the data from the API
    url = "https://api.triathlon.org/v1/events/" + str(event_id) +
"/programs/" + str(program_id) + "/results?per_page=1000"
    response = requests.get(url, headers=headers)

    # convert the response to json
    resultsData = response.json()["data"]
```

```
    # check if results is empty
    if len(resultsData['results']) == 0:
        continue

    # check if race is standard distance
    if resultsData['event']['event_specifications'][1]['cat_id'] !=
377:
        continue

    # get the results
    resultsData = resultsData['results']

    # loop through each athlete in the event
    for athlete in resultsData:

        # check if athlete DNF, DNS, DSQ or LAP
        if athlete['position'] == 'DNF' or athlete['position'] ==
'DNS' or athlete['position'] == 'DSQ' or athlete['position'] == 'LAP':
            continue

        # convert the times to seconds
        #swim
        swim = athlete['splits'][0]
        swim = pd.to_datetime(swim, format='%H:%M:%S')
        swim = swim.hour * 3600 + swim.minute * 60 + swim.second

        # bike
        bike = athlete['splits'][2]
        bike = pd.to_datetime(bike, format='%H:%M:%S')
        bike = bike.hour * 3600 + bike.minute * 60 + bike.second

        # run
        run = athlete['splits'][4]
        run = pd.to_datetime(run, format='%H:%M:%S')
        run = run.hour * 3600 + run.minute * 60 + run.second

        # total time
        time = athlete['total_time']
        time = pd.to_datetime(time, format='%H:%M:%S')
        time = time.hour * 3600 + time.minute * 60 + time.second

        # get the position
        position = athlete['position']

        # add info to dictionary
        # key is position
        # value is dictionary of a list of swim, bike, run, and total
time
        if position not in time_dict:
```

```
            time_dict[position] = {'swim': [], 'bike': [], 'run': [],
'total': []}

        time_dict[position]['swim'].append(swim)
        time_dict[position]['bike'].append(bike)
        time_dict[position]['run'].append(run)
        time_dict[position]['total'].append(time)
```

We need to do some data cleaning now by removing any results that slipped through due to mislabled data in the database. We will be focusing on olypic distances so remove any times that are too low

```
# clean up the data by removing low times that are probably from
sprint distances or super sprint distances
for position in time_dict:

    # remove swim times less than 800
    time_dict[position]['swim'] = [x for x in time_dict[position]
['swim'] if x > 800]

    # remove bike times less than 2500
    time_dict[position]['bike'] = [x for x in time_dict[position]
['bike'] if x > 2500]

    # remove run times less than 1500
    time_dict[position]['run'] = [x for x in time_dict[position]
['run'] if x > 1500]

    # remove total times less than 5000
    time_dict[position]['total'] = [x for x in time_dict[position]
['total'] if x > 5000]
```

Lastly, we must compute the average time for each position, then store it in a dataframe.

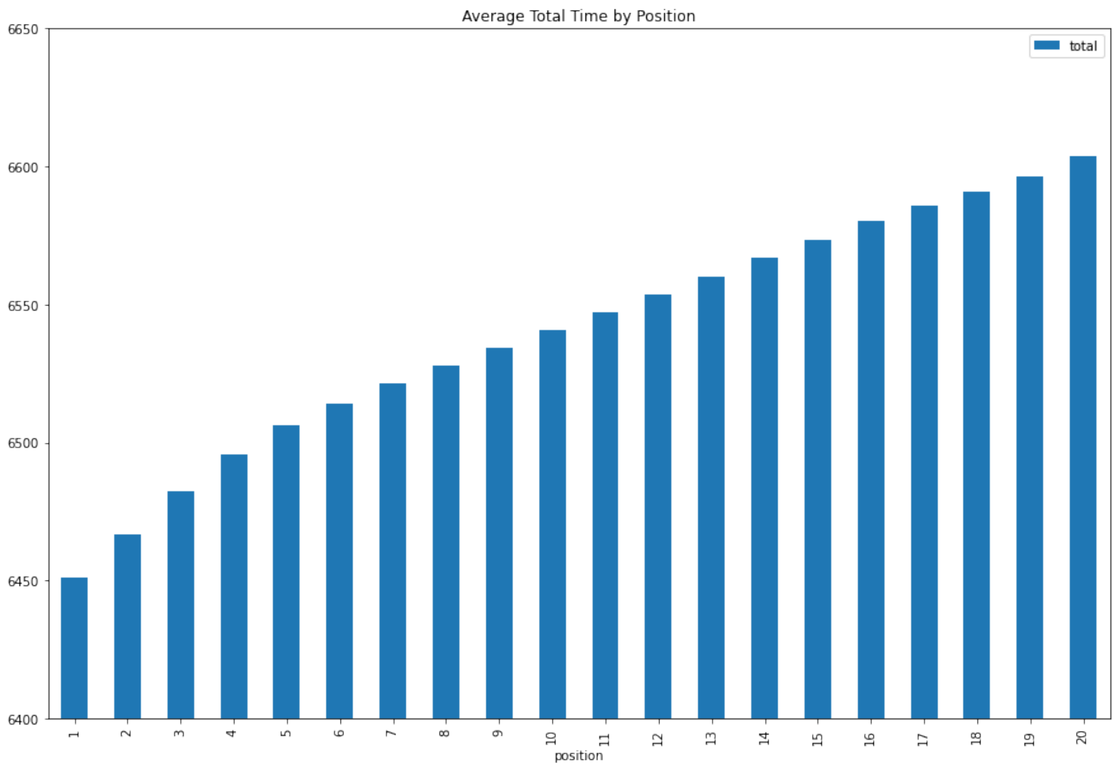
```
# compute the average times for each position and store in a list of
dictionaries
avg_times = []

for position in time_dict:
    avg_times.append({
        'position': position,
        'swim': sum(time_dict[position]['swim']) /
len(time_dict[position]['swim']),
        'bike': sum(time_dict[position]['bike']) /
len(time_dict[position]['bike']),
        'run': sum(time_dict[position]['run']) /
len(time_dict[position]['run']),
        'total': sum(time_dict[position]['total']) /
len(time_dict[position]['total'])
    })
```

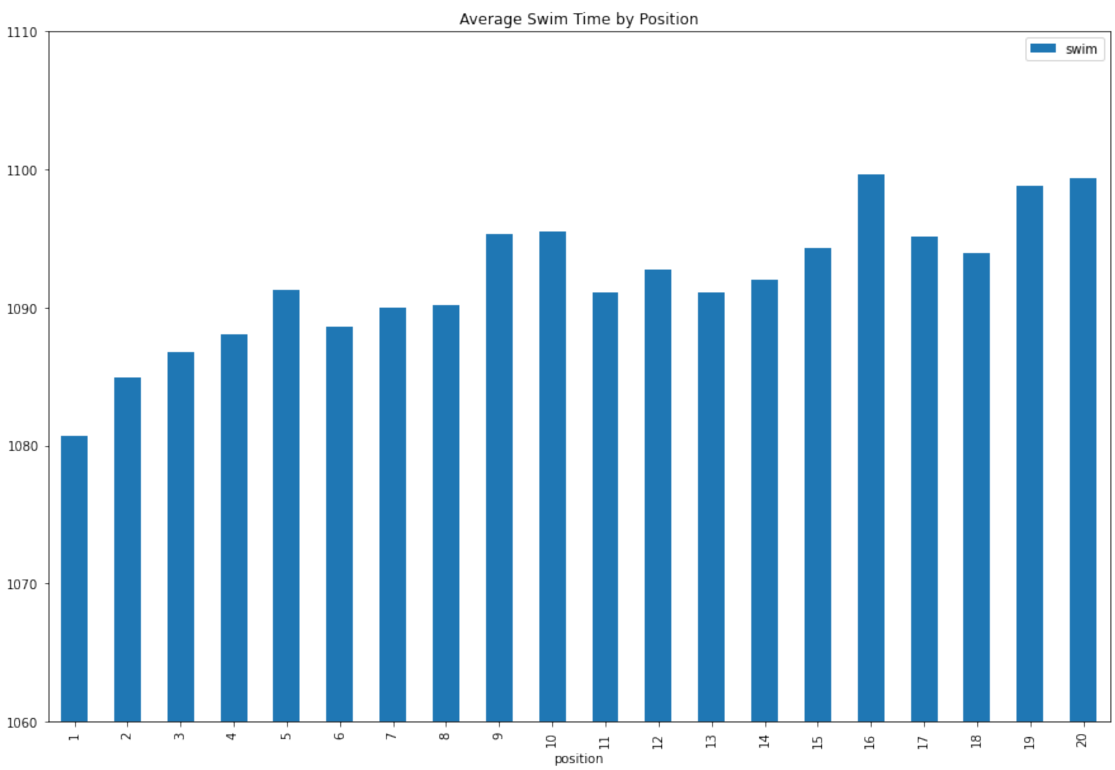
```
# create a dataframe from the list of dictionaries
avg_times_df = pd.DataFrame(avg_times)
```

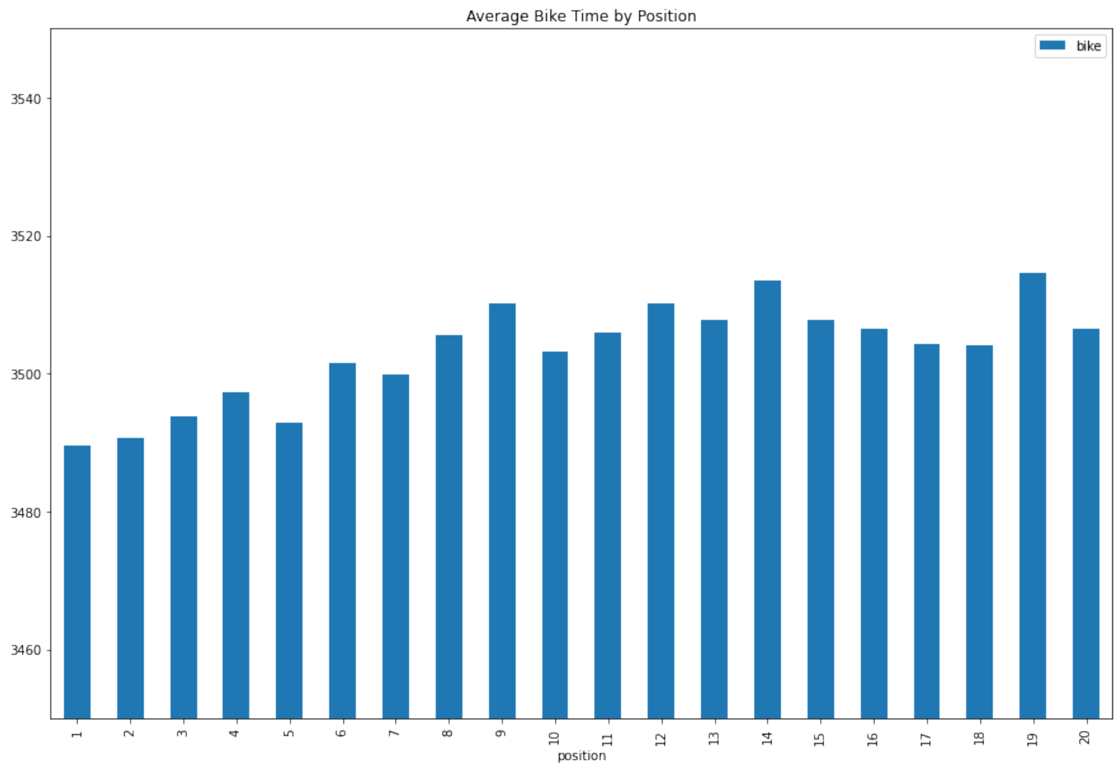
Lets now look at how the times compare over the positions.

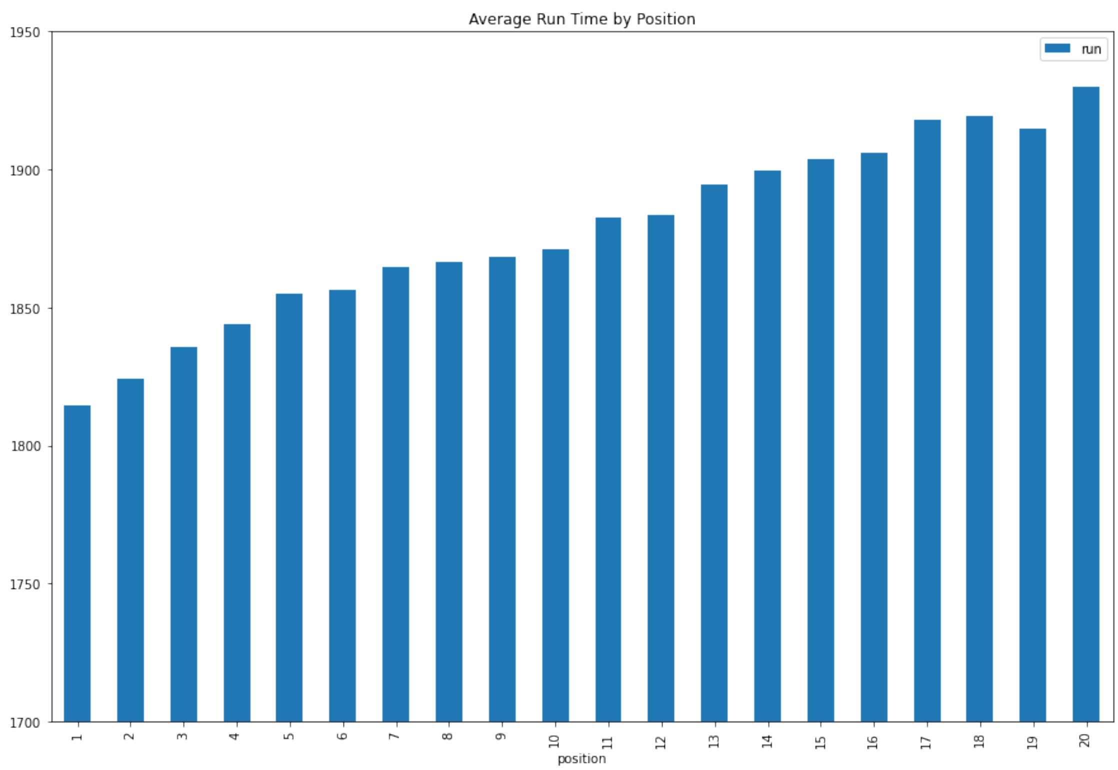
```
# plot position vs average total time as a bar chart for top 20
positions
avg_times_df.sort_values(by='position').head(20).plot.bar(x='position'
, y='total', figsize=(15,10), title='Average Total Time by Position')

# scale the y axis to be between 6000 and 7000
plt.ylim(6400, 6650)

# plot position vs average swim time as a bar chart for top 20
positions sorted by position
avg_times_df.sort_values(by='position').head(20).plot.bar(x='position'
, y='swim', figsize=(15,10), title='Average Swim Time by Position')

# scale the y axis to be between 1000 and 1500
plt.ylim(1060, 1110)

# plot position vs average bike time as a bar chart for top 20
positions sorted by position
avg_times_df.sort_values(by='position').head(20).plot.bar(x='position'
, y='bike', figsize=(15,10), title='Average Bike Time by Position')

# scale the y axis to be between 3000 and 3500
plt.ylim(3450, 3550)

# plot position vs average run time as a bar chart for top 20
positions sorted by position
avg_times_df.sort_values(by='position').head(20).plot.bar(x='position'
, y='run', figsize=(15,10), title='Average Run Time by Position')

# scale the y axis to be between 2000 and 2500
plt.ylim(1700, 1950)
```

```
(1700.0, 1950.0)
```


Average Total Time by Position
total
6650
6600
6550
6500
6450
6400
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
position


Average Swim Time by Position
swim
1110
1100
1090
1080
1070
1060
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
position





Looking at the graphs we can see obviously that the better positions have lower overall times, but what is interesting is that for the swim and bike, the finish position isn't necassarily the best predictor for the swim or bike time position. The run on the other hand is quite similar to the overall time.

Age vs Times

It would be interesting to see how the swim, bike, run, and overall times compare to the age of the athletes. To do this, we will need a dataframe with the average times for each age.

First we must collect all the swim, bike, run, and total times for each age.

```
# create a dictionary to store the average times for each age
age_dict = {}

# loop through each event
for eventName, eventDate, event_id, program_id in events:

    # Get the data from the API
    url = "https://api.triathlon.org/v1/events/" + str(event_id) +
"/programs/" + str(program_id) + "/results?per_page=1000"
    response = requests.get(url, headers=headers)

    # convert the response to json
    resultsData = response.json()["data"]

    # check if results is empty
    if len(resultsData['results']) == 0:
        continue

    # check if race is standard distance
    if resultsData['event']['event_specifications'][1]['cat_id'] !=
377:
        continue

    # get the year of the event from the event date
    eventYear = int(eventDate[:4])

    # get the results
    resultsData = resultsData['results']

    # loop through each athlete in the event
    for athlete in resultsData:

        # check if athlete DNF, DNS, DSQ or LAP
        if athlete['position'] == 'DNF' or athlete['position'] ==
'DNS' or athlete['position'] == 'DSQ' or athlete['position'] == 'LAP':
            continue

        # check if athlete age is available
        if athlete['dob'] == None:
            continue
```

```
        # convert the times to seconds
        #swim
        swim = athlete['splits'][0]
        swim = pd.to_datetime(swim, format='%H:%M:%S')
        swim = swim.hour * 3600 + swim.minute * 60 + swim.second

        # bike
        bike = athlete['splits'][2]
        bike = pd.to_datetime(bike, format='%H:%M:%S')
        bike = bike.hour * 3600 + bike.minute * 60 + bike.second

        # run
        run = athlete['splits'][4]
        run = pd.to_datetime(run, format='%H:%M:%S')
        run = run.hour * 3600 + run.minute * 60 + run.second

        # total time
        time = athlete['total_time']
        time = pd.to_datetime(time, format='%H:%M:%S')
        time = time.hour * 3600 + time.minute * 60 + time.second

        # get the age
        age = eventYear - int(athlete['dob'][:4])

        # add info to dictionary
        # key is age
        # value is dictionary of a list of swim, bike, run, and total time
        if age not in age_dict:
            age_dict[age] = {'swim': [], 'bike': [], 'run': [], 'total': []}

        age_dict[age]['swim'].append(swim)
        age_dict[age]['bike'].append(bike)
        age_dict[age]['run'].append(run)
        age_dict[age]['total'].append(time)
```

We then clean up the data by removing low times that are probably from sprint distances or super sprint distances.

```
# clean up the data by removing low times that are probably from sprint distances or super sprint distances
for age in age_dict:

    # remove swim times less than 800
    age_dict[age]['swim'] = [x for x in age_dict[age]['swim'] if x > 800]

    # remove bike times less than 2500
```

```
    age_dict[age]['bike'] = [x for x in age_dict[age]['bike'] if x >
2500]

    # remove run times less than 1500
    age_dict[age]['run'] = [x for x in age_dict[age]['run'] if x >
1500]

    # remove total times less than 5000
    age_dict[age]['total'] = [x for x in age_dict[age]['total'] if x >
5000]
```

We can now create a dataframe with the average times for each age.

```
# compute the average times for each age and store in a list of
dictionaries
avg_times = []

for age in age_dict:
    avg_times.append({
        'age': age,
        'swim': sum(age_dict[age]['swim']) / len(age_dict[age]
['swim']),
        'bike': sum(age_dict[age]['bike']) / len(age_dict[age]
['bike']),
        'run': sum(age_dict[age]['run']) / len(age_dict[age]['run']),
        'total': sum(age_dict[age]['total']) / len(age_dict[age]
['total'])
    })

# create a dataframe from the list of dictionaries
avg_times_df = pd.DataFrame(avg_times)
```

Let's take a look at how our dataframe looks.

```
# preview the dataframe
avg_times_df.head()
```

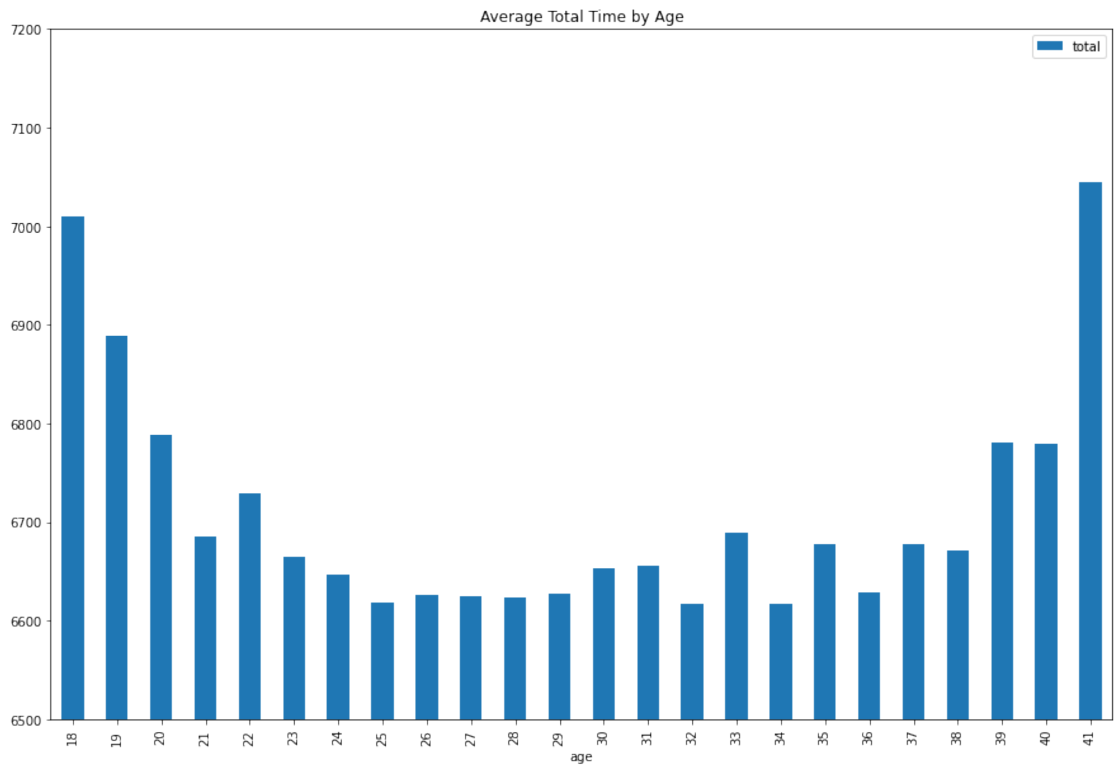
	age	swim	bike	run	total
0	32	1102.831776	3496.685185	1946.305556	6617.348624
1	30	1102.745665	3540.412429	1944.548023	6653.186441
2	23	1092.773504	3532.550847	1963.810127	6665.189873
3	27	1097.006734	3515.134426	1945.598684	6625.315789
4	22	1097.770950	3553.183784	1996.571429	6729.896739

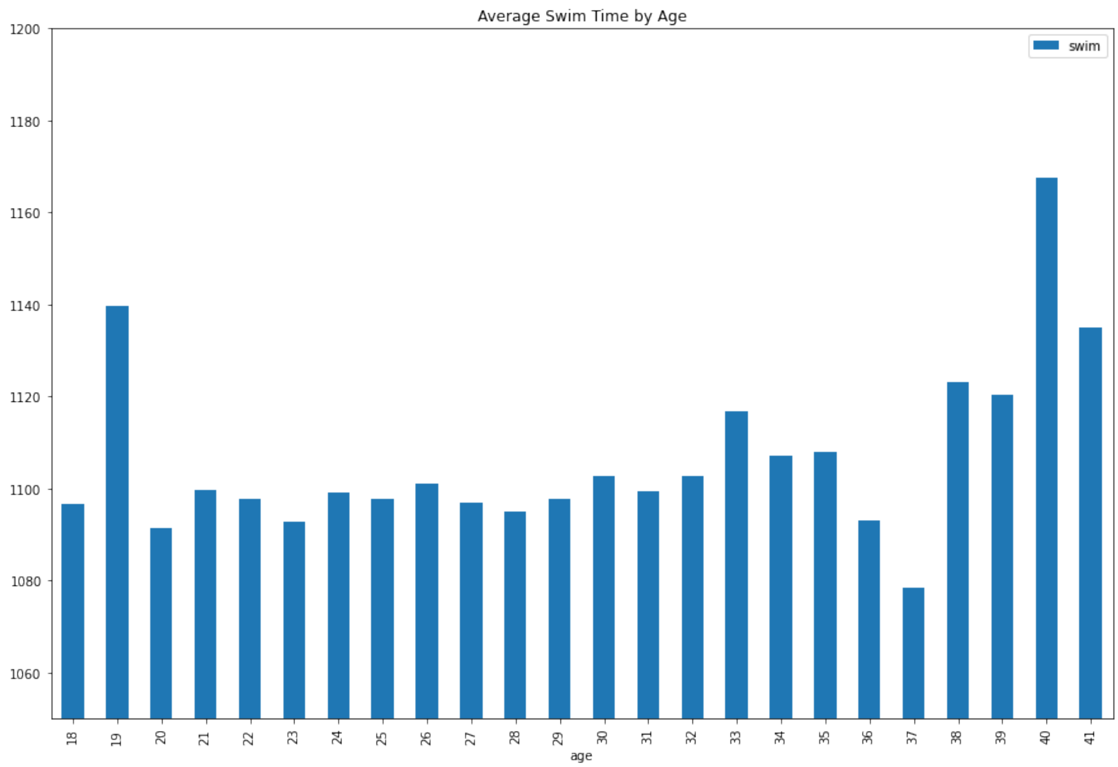
Now let's see what the fastest age is for each discipline

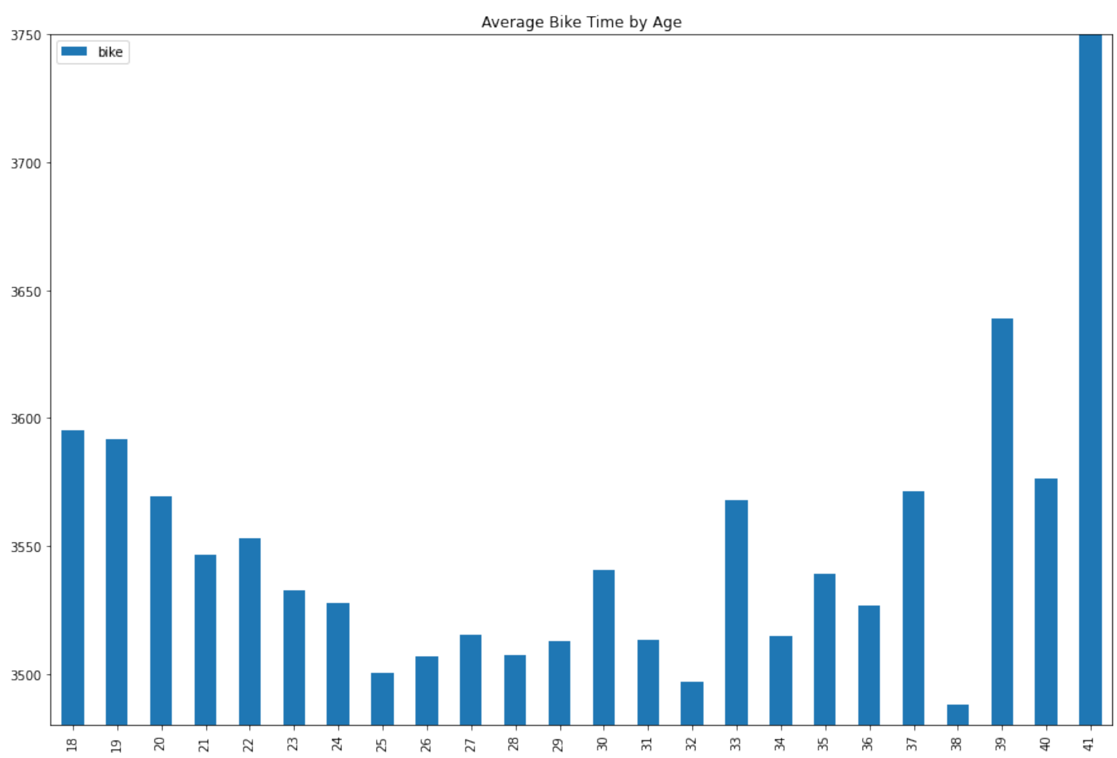
```
# plot age vs average total time as a bar chart sort by age
avg_times_df.sort_values(by='age').plot.bar(x='age', y='total',
figsize=(15,10), title='Average Total Time by Age')

# scale the y axis to be between 6000 and 7000
```

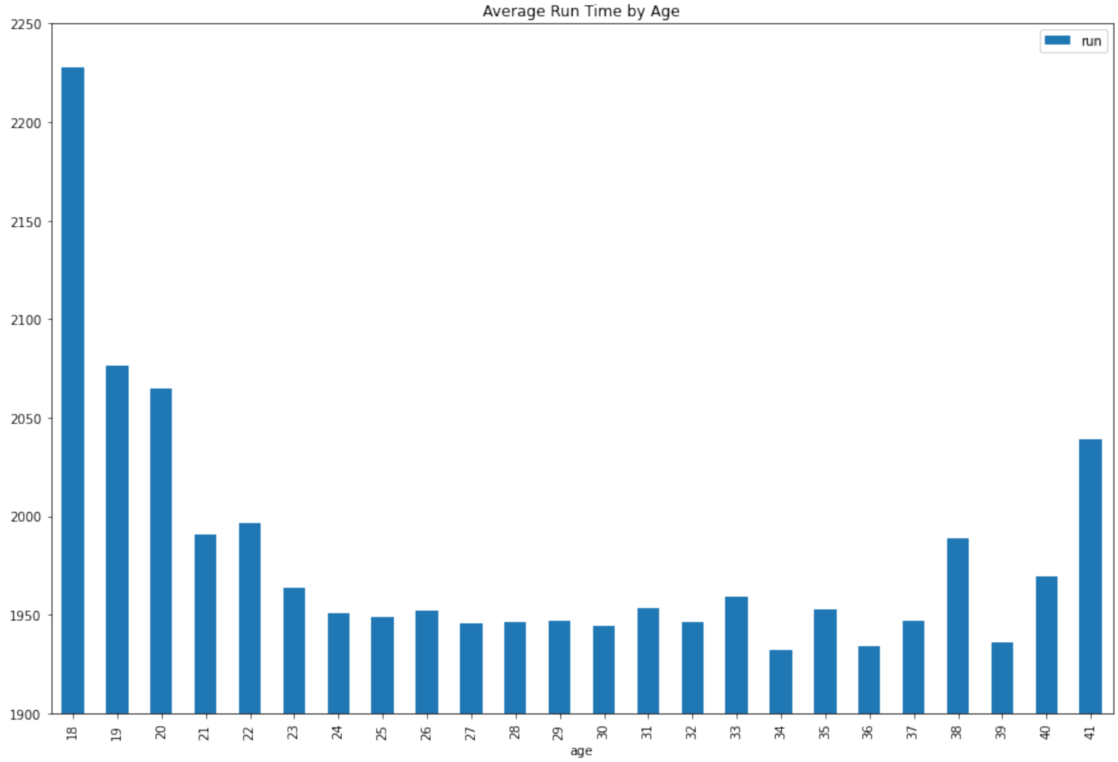
```
plt.ylim(6500, 7200)

# plot age vs average swim time as a bar chart sort by age
avg_times_df.sort_values(by='age').plot.bar(x='age', y='swim',
figsize=(15,10), title='Average Swim Time by Age')

# scale the y axis to be between 1000 and 1500
plt.ylim(1050, 1200)

# plot age vs average bike time as a bar chart sort by age
avg_times_df.sort_values(by='age').plot.bar(x='age', y='bike',
figsize=(15,10), title='Average Bike Time by Age')

# scale the y axis to be between 3000 and 3500
plt.ylim(3480, 3750)

# plot age vs average run time as a bar chart sort by age
avg_times_df.sort_values(by='age').plot.bar(x='age', y='run',
figsize=(15,10), title='Average Run Time by Age')

# scale the y axis to be between 2000 and 2500
plt.ylim(1900, 2250)
```

```
(1900.0, 2250.0)
```




Average Swim Time by Age
swim
1200
1180
1160
1140
1120
1100
1080
1060
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
age


Average Bike Time by Age
bike
3750
3700
3650
3600
3550
3500
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
age



As expected it looks like the fastest athletes are in the 30-38 age range with the slower athletes being the young, inexperienced ones.

Now lets get an exact number for each disipline.

```
# Get the fastest age group for overall, swim, bike, and run
# overall
fastest_age = avg_times_df.sort_values(by='total').iloc[0]['age']
print('Fastest Age Group Overall: ' + str(fastest_age))

# swim
fastest_age = avg_times_df.sort_values(by='swim').iloc[0]['age']
print('Fastest Age Group Swim: ' + str(fastest_age))

# bike
fastest_age = avg_times_df.sort_values(by='bike').iloc[0]['age']
print('Fastest Age Group Bike: ' + str(fastest_age))

# run
fastest_age = avg_times_df.sort_values(by='run').iloc[0]['age']
print('Fastest Age Group Run: ' + str(fastest_age))
```

```
Fastest Age Group Overall: 32.0
Fastest Age Group Swim: 37.0
Fastest Age Group Bike: 38.0
Fastest Age Group Run: 34.0
```

Data Mining

Now lets use some machine learning algorithms and see if we can predict podiums based times as well as classify athletes based on their swim, bike, and run times

Podium Predictor

Let's see if we can predict if an athlete has made a podium (1st, 2nd, or 3rd) over their proffesional career. We first need to collect the required data.

```
# For each athlete, get their average times, number of starts, and whether or not they have podiumed
# create a dictionary to store the data for each athlete with the key being the athlete id
athlete_dict = {}

# loop through each event
for eventName, eventDate, event_id, program_id in events:

    # Get the data from the API
    url = "https://api.triathlon.org/v1/events/" + str(event_id) + "/programs/" + str(program_id) + "/results?per_page=1000"
    response = requests.get(url, headers=headers)

    # convert response to json
    resultsData = response.json()["data"]

    # check if results is empty
    if len(resultsData['results']) == 0:
        continue

    # check if race is standard distance
    if resultsData['event']['event_specifications'][1]['cat_id'] != 377:
        continue

    # get the results
    results = resultsData['results']

    # loop through each athlete
    for athlete in results:
        # check if athlete is already in the dictionary
        if athlete['athlete_id'] not in athlete_dict:
            athlete_dict[athlete['athlete_id']] = {
                'swim': [],
                'bike': [],
                'run': [],
                'total': [],
                'starts': 0,
                'podiums': 0,
```

```python
            'DNFs': 0,
            'DNSs': 0,
            'DSQs': 0,
            'LAPs': 0
        }

    # check if athlete DNF, DNS, DSQ or LAP
    if athlete['position'] == 'DNF':
        athlete_dict[athlete['athlete_id']]['DNFs'] += 1
        continue

    if athlete['position'] == 'DNS':
        athlete_dict[athlete['athlete_id']]['DNSs'] += 1
        continue

    if athlete['position'] == 'DSQ':
        athlete_dict[athlete['athlete_id']]['DSQs'] += 1
        continue

    if athlete['position'] == 'LAP':
        athlete_dict[athlete['athlete_id']]['LAPs'] += 1
        continue

    # get the swim time
    swim = athlete['splits'][0]
    swim = pd.to_datetime(swim, format='%H:%M:%S')
    swim = swim.hour * 3600 + swim.minute * 60 + swim.second

    # get the bike time
    bike = athlete['splits'][2]
    bike = pd.to_datetime(bike, format='%H:%M:%S')
    bike = bike.hour * 3600 + bike.minute * 60 + bike.second

    # get the run time
    run = athlete['splits'][4]
    run = pd.to_datetime(run, format='%H:%M:%S')
    run = run.hour * 3600 + run.minute * 60 + run.second

    # get the total time
    time = athlete['total_time']
    time = pd.to_datetime(time, format='%H:%M:%S')
    time = time.hour * 3600 + time.minute * 60 + time.second

    # add the times to the dictionary
    athlete_dict[athlete['athlete_id']]['swim'].append(swim)
    athlete_dict[athlete['athlete_id']]['bike'].append(bike)
    athlete_dict[athlete['athlete_id']]['run'].append(run)
    athlete_dict[athlete['athlete_id']]['total'].append(time)
```

```
        # add to the number of starts
        athlete_dict[athlete['athlete_id']]['starts'] += 1

        # check if the athlete podiumed
        if athlete['position'] == 1 or athlete['position'] == 2 or
athlete['position'] == 3:
            athlete_dict[athlete['athlete_id']]['podiums'] += 1
```

We can do some tidying of the data now

```
# clean up the data to remove times that are 0
for athlete_id in athlete_dict:
    for time in athlete_dict[athlete_id]['swim']:
        if time == 0:
            athlete_dict[athlete_id]['swim'].remove(time)
    for time in athlete_dict[athlete_id]['bike']:
        if time == 0:
            athlete_dict[athlete_id]['bike'].remove(time)
    for time in athlete_dict[athlete_id]['run']:
        if time == 0:
            athlete_dict[athlete_id]['run'].remove(time)
    for time in athlete_dict[athlete_id]['total']:
        if time == 0:
            athlete_dict[athlete_id]['total'].remove(time)

# remove athletes that have less than 3 starts
for athlete_id in list(athlete_dict):
    if athlete_dict[athlete_id]['starts'] < 3:
        del athlete_dict[athlete_id]
```

We then need to put all of the data we have in a list of dicts where each dict is an athlete. Then we are able to create a dataframe from it.

```
# create a list of the athletes with the data
athlete_list = []

# loop through each athlete
for athlete_id in athlete_dict:

    # check if athlete has any starts
    if athlete_dict[athlete_id]['starts'] == 0:
        continue

    # calculate the average times
    swim_avg = sum(athlete_dict[athlete_id]['swim']) /
len(athlete_dict[athlete_id]['swim'])
    bike_avg = sum(athlete_dict[athlete_id]['bike']) /
len(athlete_dict[athlete_id]['bike'])
    run_avg = sum(athlete_dict[athlete_id]['run']) /
len(athlete_dict[athlete_id]['run'])
```

```
    total_avg = sum(athlete_dict[athlete_id]['total']) /
len(athlete_dict[athlete_id]['total'])

    # variable to store whether or not the athlete has podiumed
    if athlete_dict[athlete_id]['podiums'] > 0:
        podium = 1
    else:
        podium = 0

    # create a dictionary to store the data for the athlete
    athlete = {
        'athlete_id': athlete_id,
        'starts': athlete_dict[athlete_id]['starts'],
        'podiums': podium,
        'DNFs': athlete_dict[athlete_id]['DNFs'],
        'DNSs': athlete_dict[athlete_id]['DNSs'],
        'DSQs': athlete_dict[athlete_id]['DSQs'],
        'LAPs': athlete_dict[athlete_id]['LAPs'],
        'swim_avg': swim_avg,
        'bike_avg': bike_avg,
        'run_avg': run_avg,
        'total_avg': total_avg
    }

    # add the athlete to the list
    athlete_list.append(athlete)

# create a dataframe from the list of athletes
athlete_df = pd.DataFrame(athlete_list)
```

Get a feel for the data

Before preparing the data for machine learning algorithms, lets get a feel for the data and see if we need to do any further cleaning

```
# preview the dataframe
athlete_df.head()
```

```
   athlete_id  starts  podiums  DNFs  DNSs  DSQs  LAPs     swim_avg  \
0        5297      11        1     5     0     0     0  1098.100000
1        5711      14        1     4     0     0     0  1086.285714
2        7747      35        1     7     0     0     0  1060.228571
3        5296      13        1     2     0     0     0  1088.615385
4        5747      25        1     2     0     0     0  1081.720000

      bike_avg      run_avg    total_avg
0  3578.200000  1880.909091  6647.181818
1  3547.857143  1862.000000  6564.000000
2  3576.485714  1903.028571  6617.714286
```

```
3  3495.923077  1898.538462  6554.153846
4  3587.160000  1874.080000  6617.080000
```

```python
# Descriptive statistics for the dataframe
athlete_df.describe()
```

```
          athlete_id      starts     podiums        DNFs        DNSs

count     267.000000  267.000000  267.000000  267.000000
267.000000  \
mean    29979.348315   10.370787    0.183521    1.823970    0.018727

std     30899.016137    7.385286    0.387820    1.856432    0.135812

min      5296.000000    3.000000    0.000000    0.000000    0.000000

25%      6189.000000    4.000000    0.000000    0.000000    0.000000

50%     12816.000000    8.000000    0.000000    1.000000    0.000000

75%     46197.000000   14.000000    0.000000    3.000000    0.000000

max    158264.000000   35.000000    1.000000   11.000000    1.000000


             DSQs        LAPs     swim_avg     bike_avg      run_avg

count  267.000000  267.000000   267.000000   267.000000
267.000000  \
mean     0.056180    0.067416  1076.012358  3475.040092  1935.550263

std      0.358345    0.305258    52.815659   196.681108   119.594616

min      0.000000    0.000000   815.000000  2377.333333  1441.000000

25%      0.000000    0.000000  1055.833333  3423.752137  1866.427718

50%      0.000000    0.000000  1086.500000  3513.200000  1941.800000

75%      0.000000    0.000000  1110.857143  3572.463203  2010.085714

max      5.000000    3.000000  1179.000000  3857.666667  2311.666667


         total_avg
count   267.000000
mean   6565.398941
std     322.911839
min    4718.666667
```

```
25%    6454.884740
50%    6617.714286
75%    6752.750000
max    7161.000000
```

```
# Check for missing values
athlete_df.isnull().sum()
```

```
athlete_id    0
starts        0
podiums       0
DNFs          0
DNSs          0
DSQs          0
LAPs          0
swim_avg      0
bike_avg      0
run_avg       0
total_avg     0
dtype: int64
```

```
# check the data types
athlete_df.dtypes
```

```
athlete_id      int64
starts          int64
podiums         int64
DNFs            int64
DNSs            int64
DSQs            int64
LAPs            int64
swim_avg      float64
bike_avg      float64
run_avg       float64
total_avg     float64
dtype: object
```

Prepare the data for machine learning

Since there are many more non-podiums compared to podiums, we will want to do stratified sampling to ensure the training and test sets have similar ratios of podiums to non-podium instances

```
# Prepare the data for modeling
# create a copy of the dataframe
athlete_df_model = athlete_df.copy()

# drop the athlete_id column
athlete_df_model.drop('athlete_id', axis=1, inplace=True)

# use stratified sampling to split the data into training and testing
```

```
sets
from sklearn.model_selection import StratifiedShuffleSplit

# create the stratified shuffle split object
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
random_state=42)

# split the data into training and testing sets
for train_index, test_index in split.split(athlete_df_model,
athlete_df_model['podiums']):
    strat_train_set = athlete_df_model.loc[train_index]
    strat_test_set = athlete_df_model.loc[test_index]

# check the value counts for the podiums column in the training set
strat_train_set['podiums'].value_counts()
```

```
podiums
0    174
1     39
Name: count, dtype: int64
```

The next steps involve creating the label and training instances and then creating a pipeline to transform and fit the training instances.

```
# create the X and y variables for the training set
X_train = strat_train_set.drop('podiums', axis=1)
y_train = strat_train_set['podiums'].copy()

# create pipelines for the numerical and categorical attributes
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

# numerical pipeline
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('std_scaler', StandardScaler())
])

# fit the numerical pipeline to the training data
X_train_num = num_pipeline.fit_transform(X_train)
```

We can then train a logistic regression model

```
# train a logistic regression model
from sklearn.linear_model import LogisticRegression

# create the logistic regression model
log_reg = LogisticRegression()

# fit the model to the training data
```

```
log_reg.fit(X_train_num, y_train)

# create a function to display the results of the model
def display_scores(scores):
    print('Scores:', scores)
    print('Mean:', scores.mean())
    print('Standard Deviation:', scores.std())

# use cross validation to evaluate the model
from sklearn.model_selection import cross_val_score

# calculate the cross validation scores
scores = cross_val_score(log_reg, X_train_num, y_train,
scoring='accuracy', cv=10)

# display the scores
display_scores(scores)
```

```
Scores: [0.81818182 0.95454545 0.81818182 0.80952381 0.9047619
0.95238095
 0.9047619  0.95238095 0.85714286 0.85714286]
Mean: 0.8829004329004329
Standard Deviation: 0.05552654740134981
```

Let's see if we can improve the scores by using grid search to find the best paramaters for the model

```
# use grid search to find the best parameters for the model
from sklearn.model_selection import GridSearchCV

# create the parameter grid
param_grid = [
    {'penalty': ['l1', 'l2', 'elasticnet', 'none'], 'solver':
['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']}
]

# create the grid search object
grid_search = GridSearchCV(log_reg, param_grid, cv=10,
scoring='accuracy', return_train_score=True)

# fit the grid search object to the training data
grid_search.fit(X_train_num, y_train)

# display the best parameters
grid_search.best_params_
```

```
{'penalty': 'l2', 'solver': 'saga'}
```

Using the new parameters we get slightly better scores

```
# use the best parameters to create a new model
log_reg_best = LogisticRegression(penalty='l2', solver='saga')

# fit the new model to the training data
log_reg_best.fit(X_train_num, y_train)

# use cross validation to evaluate the model
scores = cross_val_score(log_reg_best, X_train_num, y_train,
scoring='accuracy', cv=10)

# display the scores
display_scores(scores)
```

```
Scores: [0.81818182 0.95454545 0.81818182 0.85714286 0.9047619
0.95238095
 0.9047619  0.95238095 0.85714286 0.85714286]
Mean: 0.8876623376623378
Standard Deviation: 0.050876852442227005
```

Next let's look at the confusion matrix as well as ROC, precision, and recall

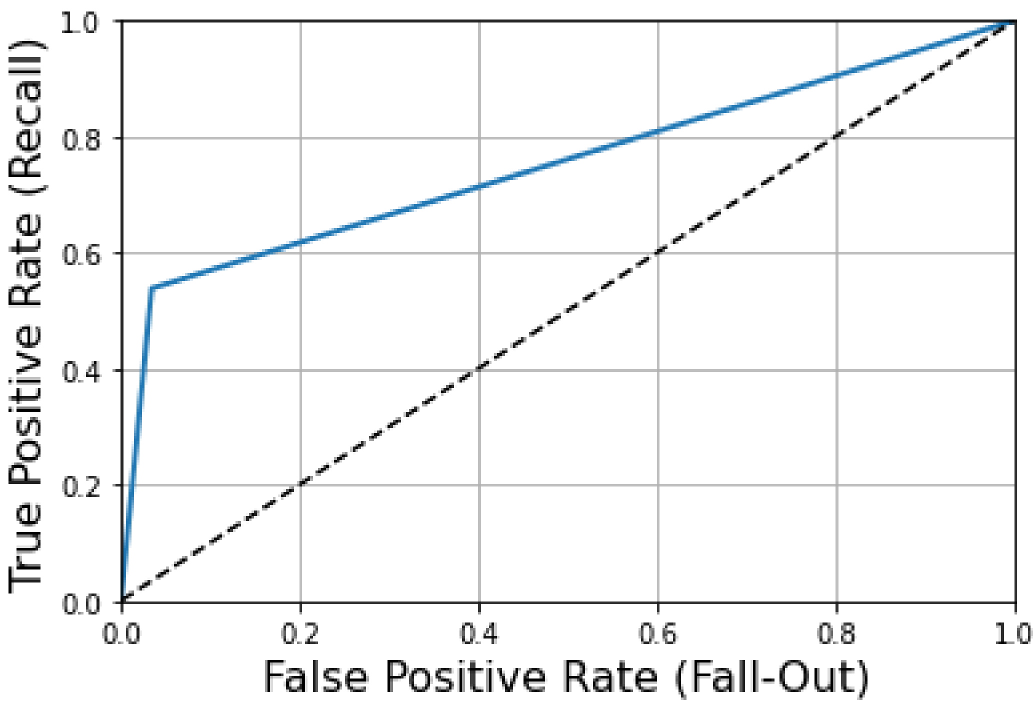
```
# compute the confusion matrix
from sklearn.model_selection import cross_val_predict

# make predictions using cross validation
y_train_pred = cross_val_predict(log_reg_best, X_train_num, y_train,
cv=10)

# create the confusion matrix
from sklearn.metrics import confusion_matrix

# create the confusion matrix
confusion_matrix(y_train, y_train_pred)
```

```
array([[168,   6],
       [ 18,  21]], dtype=int64)
```

```
# calculate the precision and recall
from sklearn.metrics import precision_score, recall_score

# calculate the precision score
p = precision_score(y_train, y_train_pred)

# calculate the recall score
r = recall_score(y_train, y_train_pred)

# print the precision and recall scores
print('Precision:', p)
print('Recall:', r)
```

```
Precision: 0.7777777777777778
Recall: 0.5384615384615384
```

```python
# calculate the roc curve
from sklearn.metrics import roc_curve

# calculate the fpr, tpr, and thresholds
fpr, tpr, thresholds = roc_curve(y_train, y_train_pred)

# plot the roc curve
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0,1], [0,1], 'k--') # dashed diagonal
    plt.axis([0,1,0,1])
    plt.xlabel('False Positive Rate (Fall-Out)', fontsize=16)
    plt.ylabel('True Positive Rate (Recall)', fontsize=16)
    plt.grid(True)

# plot the roc curve
plot_roc_curve(fpr, tpr)
```




```python
# use the roc auc score to evaluate the model
from sklearn.metrics import roc_auc_score

# calculate the roc auc score
roc_auc_score(y_train, y_train_pred)
```

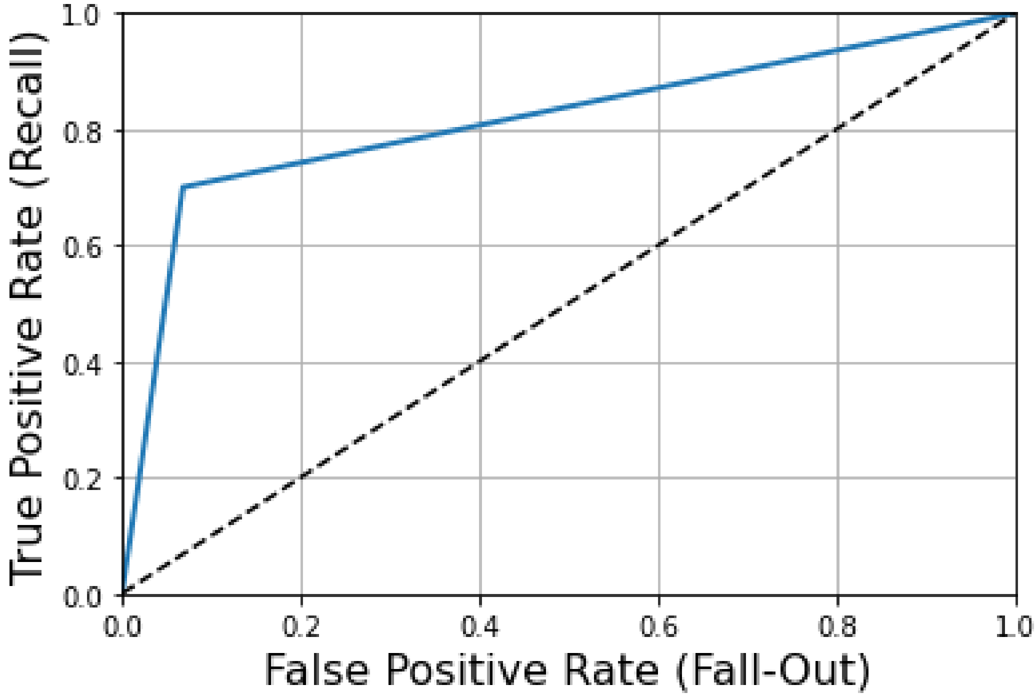
```
0.7519893899204244
```

The model is performing prett good, I prefer a higher precision rather than recall as I would rather the model ensure that athletes that haven't podiumed aren't mistaken for a podium

Now lets test the model on the test set

```
# test the model on the test set
# create the X and y variables for the test set
X_test = strat_test_set.drop('podiums', axis=1)
y_test = strat_test_set['podiums'].copy()

# transform the test set using the numerical pipeline
X_test_num = num_pipeline.transform(X_test)

# make predictions on the test set
y_test_pred = log_reg_best.predict(X_test_num)

# calculate the precision and recall scores
p = precision_score(y_test, y_test_pred)
r = recall_score(y_test, y_test_pred)

# print the precision and recall scores
print('Precision:', p)
print('Recall:', r)

# calculate the roc auc score
roc_auc_score(y_test, y_test_pred)

# calculate the fpr, tpr, and thresholds
fpr, tpr, thresholds = roc_curve(y_test, y_test_pred)

# plot the roc curve
plot_roc_curve(fpr, tpr)
```

```
Precision: 0.7
Recall: 0.7
```




```
# accuracy score
from sklearn.metrics import accuracy_score

# calculate the accuracy score
accuracy_score(y_test, y_test_pred)

0.8888888888888888

# which features are the most important?
# create a dataframe with the feature names and coefficients
feature_importances = pd.DataFrame(log_reg_best.coef_.T,
index=athlete_df_model.drop('podiums', axis=1).columns,
columns=['importance'])

# sort the dataframe by the importance
feature_importances.sort_values('importance', ascending=False)

           importance
starts       1.405046
swim_avg     1.365237
LAPs         0.031257
DNFs         0.011221
DNSs         0.003449
bike_avg    -0.133107
DSQs        -0.243026
total_avg   -0.528116
run_avg     -1.589112
```

Conclusion

The analysis of World Triathlon Elite Men provides valuable insights into the performance of athletes at the highest level of triathlon. Over the years, the winners' times have changed due to better technology and a better understanding of training. The sport's fastest age group overall is 32, with the fastest age group swimmer at 37, the fastest age group biker at 38, and the fastest age group runner at 34.

The analysis shows that the average overall and bike times have gotten faster, which can be attributed to the significant improvements in bike technology over the past decade. Swim times, on the other hand, have remained relatively similar. However, the most dramatic improvement in performance has been observed in run times since the release of Nike's super shoe in 2016.

The analysis also highlights the importance of different attributes in triathlon performance. Running is the most important factor for success, followed by the number of starts and swimming. The logistic regression model was able to predict podiums with 89% accuracy, which is a significant improvement in predicting athlete performance.

These insights are important not only for elite athletes but also for coaches and trainers. By identifying the key performance indicators and factors that contribute to success in the sport, coaches and trainers can develop training programs and strategies to help athletes improve their performance and achieve their full potential.

Overall, the analysis of World Triathlon Elite Men provides valuable insights into the trends and patterns in triathlon performance and can be used to guide training and development programs for athletes and coaches alike.