David Burton & Adam Casenas
4/18/23
SI 206
Barbara Ericson

SI 206 Project Report

Goals:

The primary objective of our project was to analyze the correlation between the popularity of artists on Spotify and their live event performances, as indicated by the Ticketmaster API. To accomplish this, we decided to work with two widely recognized APIs, namely the Spotify API and the Ticketmaster API. We believed Spotify API would provide us with valuable insights into the artist's popularity, and the Ticketmaster API allows us to gather comprehensive information about the live events.

In order to draw meaningful comparisons and conclusions, we gathered data on a diverse range of artists from various genres and levels of popularity. Our intention was to examine whether there is a direct relationship between an artist's popularity on Spotify and the frequency and scale of their live performances. We hypothesized that artists with higher popularity on Spotify would have a larger number of live events. By analyzing this data, we aimed to understand the dynamics between an artist's digital presence and their live performance success, shedding light on the evolving landscape of the music industry in the digital age.

Achieved Goals:

Upon implementing the Spotify API and the Ticketmaster API, we successfully achieved some of our primary goals. We were able to generate a table that displayed key information on ten popular artists, including their average popularity score (based on their top 10 songs) and the number of concerts they performed. This data enabled us to conduct a preliminary analysis of the correlation between an artist's popularity on Spotify and their frequency of live events. Additionally, we were able to gather information on the regions and cities where these concerts took place, providing a geographical perspective on the artists' performance trends

However, there were certain aspects of our project where we did not fully meet our objectives. Specifically, we were unable to obtain data on the number of attendees for each concert and the number of listeners per song, which limited the depth of our analysis. These missing data points would have provided us with a better understanding of the relationship between an artist's digital presence and the scale of their live performances. It is worth noting that the APIs we utilized either did not provide this information, or we were unable to successfully find it.

Problems:

One problem we faced was that in setting our goals we assumed we would be able to obtaining specific attendance of concerts with the Ticket Master API and number of plays per song with the Spotify API, but once we actually began working with the APIs we found that those values were not explicitly available so we had to find alternatives. For the Spotify API we found that the

API provided a value ‘popularity’ which indicated a given song's popularity based on an algorithm developed by Spotify and instead of attendance numbers we could defer to the number of different cities each artist was scheduled to perform in on Ticketmaster. We also found that the spotify api itself was difficult to work with in Python so we resorted to the python library ‘Spotipy’ to make the API easier to work with.

Calculations:

Whole calculation processing file:

```python
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
import sqlite3

def main():
    # set up found in spotipy guide
    client_credentials_manager = SpotifyClientCredentials()
    sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)

    conn = sqlite3.connect('finalDatabase.db')
    cursor = conn.cursor()

    # artists we are using
    artists = ["Taylor Swift", "Ed Sheeran", "Beyonce", "Bruno Mars", "Drake", "Khalid", "Lady Gaga", "Katy Perry", "Bono", "Adele"]
    # create tables artists, songs, and spotify
    cursor.execute('''CREATE TABLE IF NOT EXISTS artists
                (id integer PRIMARY KEY, name text)''')
    cursor.execute('''CREATE TABLE IF NOT EXISTS songs
                (id integer PRIMARY KEY, name text)''')
    cursor.execute('''CREATE TABLE IF NOT EXISTS spotify
                (id integer PRIMARY KEY, artist_id integer, song_id integer, popularity integer,
                FOREIGN KEY(artist_id) REFERENCES artists(id), FOREIGN KEY(song_id) REFERENCES songs(id))''')

    # Check if the artist table is empty
    cursor.execute("SELECT COUNT(*) FROM artists")
    numArtists = cursor.fetchone()[0]
    if numArtists == 0:
        #on the first run only fill artists table
        for artist in artists:
            cursor.execute("INSERT INTO artists (name) VALUES (?)", (artist,))
            conn.commit()
    else:
    # Loop through the artists and get their top tracks
        cursor.execute("SELECT COUNT(*) FROM spotify")
        numSongs = cursor.fetchone()[0]
        artist_id = numSongs // 10 +1
        artist = artists[artist_id - 1]

        #taken from spotipy guide
        info = sp.search(q=artist, type="artist")
        if info["artists"]["items"]:
            # Insert the artist data into the database
            songs = sp.artist_top_tracks(info["artists"]["items"][0]["id"])["tracks"][:10]
            for song in songs:
                name = song['name']
                popularity = song['popularity']

                #check if the song already occurs
                cursor.execute("SELECT COUNT(*) FROM spotify")
                numSongs = cursor.fetchone()[0]
                song_id = numSongs + 1
                cursor.execute("SELECT id FROM songs WHERE name=?", (name,))
                songNumber = cursor.fetchone()
                if songNumber is not None:
                    song_id = songNumber[0]

                # Insert songs and data into spotify table
                cursor.execute("INSERT OR IGNORE INTO songs (name) VALUES (?)", (name,))
                cursor.execute("INSERT INTO spotify (artist_id, song_id, popularity) VALUES (?, ?, ?)", (artist_id, song_id, popularity))
                conn.commit()
        else:
            print(f"Error finding songs for artist {artist}")

    # Close the database connection
    conn.close()

if __name__ == "__main__":
    main()
```

We had a table in our database named "spotify" (as displayed in the left-hand table below), which featured 10 songs per artist. The data below illustrates songs by Ed Sheeran (artist_id 1) and Taylor Swift (artist_id 2). To calculate the average popularity, we grouped the data by

artist_id and computed the average, resulting in the average popularity for each artist. Additionally, we had another table in our database called "ticket_master" (visible in the right-hand table below), which provided information on individual concerts. The screenshot displays all the concerts performed by Ed Sheeran (artist_id 1). To determine the number of concerts performed by an artist, we grouped the data by artist_id and then calculated the count for each group. The final table, along with the code used to generate it, is presented below:

artist_id	song_id	popularity
Filter	Filter	Filter
1	1	95
1	2	91
1	3	89
1	4	80
1	5	89
1	6	88
1	7	79
1	8	88
1	9	86
1	10	79
2	11	79
2	12	90
2	13	88
2	14	88
2	15	87
2	16	67
2	17	80
2	18	66
2	19	83
2	20	75

artist_id	city_id	region_country_id
Filter	Filter	Filter
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6
1	7	7
1	8	8
1	9	9
1	10	10
1	11	11
1	12	12
1	13	13
1	14	14
1	15	6
1	16	6
1	17	15
1	18	16
1	19	17

```
# SQL statement to get the artist name and average popularity
c.execute("""
SELECT artists.name, AVG(spotify.popularity) as popularity
FROM artists JOIN spotify ON artists.id = spotify.artist_id
GROUP BY artists.name""")
```

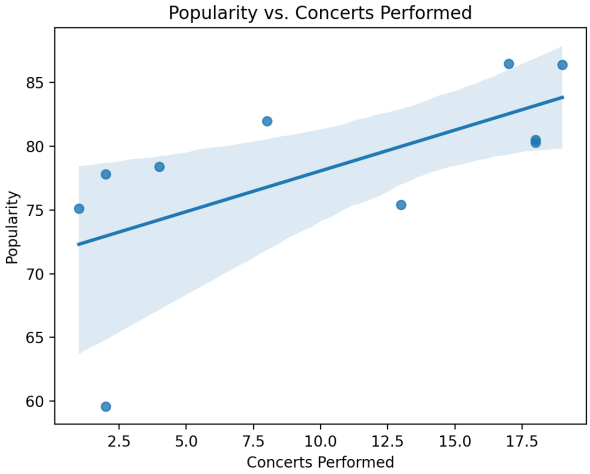
```
# SQL statement to get the concerts_performed column
c.execute("""SELECT ticket_master.artist_id, COUNT(*) as concerts_performed
FROM ticket_master
GROUP BY ticket_master.artist_id""")
```

	artist	popularity	concerts_performed
0	Adele	78.4	4
1	Beyonce	75.4	13
2	Bono	59.6	2
3	Bruno Mars	82.0	8
4	Drake	86.5	17
5	Ed Sheeran	80.3	18
6	Katy Perry	75.1	1
7	Khalid	80.5	18
8	Lady Gaga	77.8	2
9	Taylor Swift	86.4	19

Whole CSV FILE ^

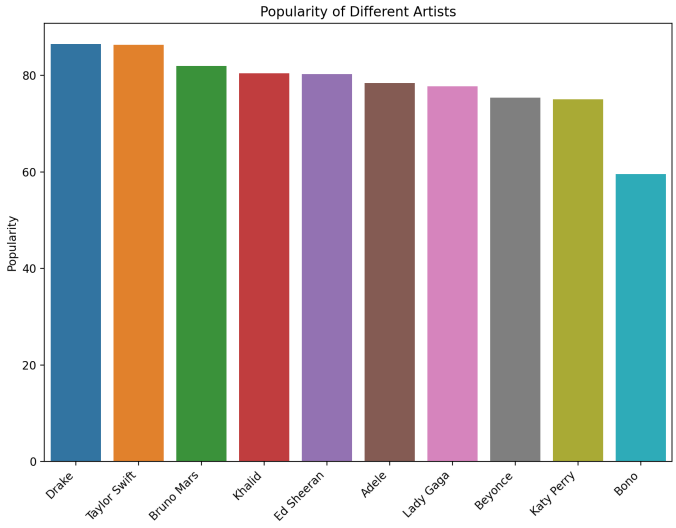
Visualizations:

There were several intriguing methods to visualize the data gathered from the tables mentioned above. First, we examined the correlation between an artist's popularity and the number of concerts they performed:

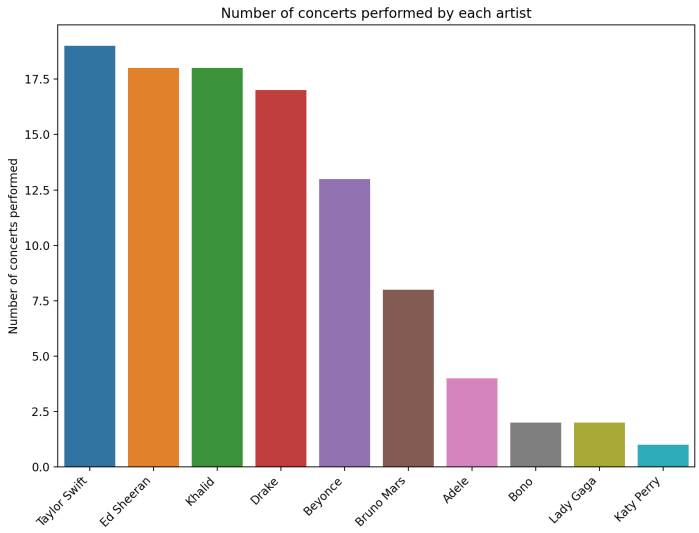


This visualization reveals a positive relationship between the number of concerts performed and the popularity of the artist. Intuitively, this makes sense because as an artist becomes more popular, more people are likely to be willing to pay money to watch them perform live.

Second, we looked at a bar chart of the artist and their respective popularity in descending order:



Third, we looked at a bar chart of the artist and the amount of concerts they performed in:



Instructions:
We stored our data in a database titled ‘finalDatabase.db’. In order to gather data from the Spotify and Ticketmaster API’s we wrote files ‘spotifyapi.py’ and ‘final_project.py’ for each API respectively that would pull requests from each API and store them into the database. We also wrote a file called ‘visualizations.py’ that would select the data from the database and create three distinct visualizations and write the data to a csv file.

To fill the database:

Start with the file ‘spotifyapi.py’ and enter a Spotify Client ID into the command line using the command
‘export SPOTIPY_CLIENT_ID=9511b86fc99b4f10a5fa20cb16f21b64’ and a Spotify Client Secret using the command
‘export SPOTIPY_CLIENT_SECRET=acd3f41233494703b66ce40e5b6264ed’
then you must hit the run button 1 time to fill the table ‘artists’ and 10 times to commit the data from the spotify api to the database (one for each artist committing 10 lines at a time).

Now go to the file ‘final_project.py’. You must run the code 10 times. One run for each artist which will commit no more than 20 lines for each run.

Lastly to process the data and create visualizations:

Go to the file ‘visualizations.py’ and run the code once to create the visualizations and write out the data to a csv file.

Code Documentation:

7. Documentation for each function that you wrote. This includes describing the input and output for each function (20 points)

In spotifyapi.py:

```
def gather_and_store(cursor, conn, sp, artists):
    # Check if the artist table is empty
    cursor.execute("SELECT COUNT(*) FROM artists")
    numArtists = cursor.fetchone()[0]
    if numArtists == 0:
        #on the first run only fill artists table
        for artist in artists:
            cursor.execute("INSERT INTO artists (name) VALUES (?)", (artist,))
            conn.commit()
    else:
    # Loop through the artists and get their top tracks
        cursor.execute("SELECT COUNT(*) FROM spotify")
        numSongs = cursor.fetchone()[0]
        artist_id = numSongs // 10 +1
        artist = artists[artist_id - 1]

        #taken from spotipy guide
        info = sp.search(q=artist, type="artist")
        if info["artists"]["items"]:
            # Insert the artist data into the database
            songs = sp.artist_top_tracks(info["artists"]["items"][0]["id"])["tracks"][:10]
            for song in songs:
                name = song['name']
                popularity = song['popularity']

                #check if the song already occurs
                cursor.execute("SELECT COUNT(*) FROM spotify")
                numSongs = cursor.fetchone()[0]
                song_id = numSongs + 1
                cursor.execute("SELECT id FROM songs WHERE name=?", (name,))
                songNumber = cursor.fetchone()
                if songNumber is not None:
                    song_id = songNumber[0]

                # Insert songs and data into spotify table
                cursor.execute("INSERT OR IGNORE INTO songs (name) VALUES (?)", (name,))
                cursor.execute("INSERT INTO spotify (artist_id, song_id, popularity) VALUES (?, ?, ?)", (artist_id, song_id, popularity))
                conn.commit()
        else:
            print(f"Error finding songs for artist {artist}")
```

We wrote a function called gather_and_store which takes in a cursor: c, a connection object: conn, a spotipy object: sp, and a list of artists: artists. This establishes what run of the code we are on by checking how many lines have been added to the various tables. The function first checks if any lines have been added to the table artists and if the function is in its first run it fills the helper table ‘artists’. For each concurrent run it checks the line count of the main table spotify and based on how many lines have been added decides which artist needs to be processed during that run. It then uses the spotipy helper library to pull from the spotify api the top 10 songs from each artist

detailing the name and popularity for each track. It then fills the tables ‘songs’ and ‘spotify’ with the data accordingly/

```
def main():
    # set up found in spotipy guide
    client_credentials_manager = SpotifyClientCredentials()
    sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)

    conn = sqlite3.connect('finalDatabase.db')
    cursor = conn.cursor()

    # artists we are using
    artists = ["Taylor Swift", "Ed Sheeran", "Beyonce", "Bruno Mars", "Drake", "Khalid", "Lady Gaga", "Katy Perry", "Bono", "Adele"]
    # create tables artists, songs, and spotify
    cursor.execute('''CREATE TABLE IF NOT EXISTS artists
                (id integer PRIMARY KEY, name text)''')
    cursor.execute('''CREATE TABLE IF NOT EXISTS songs
                (id integer PRIMARY KEY, name text)''')
    cursor.execute('''CREATE TABLE IF NOT EXISTS spotify
                (id integer PRIMARY KEY, artist_id integer, song_id integer, popularity integer,
                FOREIGN KEY(artist_id) REFERENCES artists(id), FOREIGN KEY(song_id) REFERENCES songs(id))''')
    gather_and_store(cursor, conn, sp, artists)

    conn.close()
if __name__ == "__main__":
    main()
```

The main function takes no parameters. It establishes the spotipy library helper item sp, the lists of artists we are to work with, and sets up a connection and cursor. It also sets up the tables then calls gather_and_store to fill them.
In final_project.py:

```
def gather_and_store(c, conn, apikey, artists):
    c.execute("SELECT COUNT(*) FROM ticket_master")
    numArtists = c.fetchone()[0]
    artist = ""
    artist_id = 1
    if numArtists == 0:
        artist = artists[0]
    else:
        c.execute("SELECT artist_id FROM ticket_master ORDER BY id DESC LIMIT 1")
        artist_id = c.fetchone()[0]
        artist_id += 1
        c.execute("SELECT name FROM artists WHERE id=?", (artist_id,))
        artist = c.fetchone()[0]

    print(f"This is {artist} cities")
    search_response: Any
    master.com/discovery/v2/events.json?apikey={apikey}&keyword={artist}"
    search_response = requests.get(search_url).json()

    # Get the cities where the artist performed and retrieve the pricing details for each event
    if "_embedded" in search_response:
        for event in search_response["_embedded"]["events"]:
            city = event["_embedded"]["venues"][0]["city"]["name"]
            region = event["_embedded"]["venues"][0]["state"]["name"]
            country = event["_embedded"]["venues"][0]["country"]["name"]
            region_country = region + "," + country
            c.execute("SELECT COUNT(*) FROM ticket_master")
            max_id = c.fetchone()[0]
            curr_id = max_id + 1
            c.execute('INSERT OR IGNORE INTO region_country (name) VALUES (?)', (region_country,))
            c.execute('INSERT OR IGNORE INTO cities (name) VALUES (?)', (city,))
            # Insert the data into the artist_cities table
            c.execute("""
                INSERT OR IGNORE INTO ticket_master (id, artist_id, city_id, region_country_id)
                VALUES (?, ?, (SELECT id FROM cities WHERE name = ?), (SELECT id FROM region_country WHERE name = ?) )""", (curr_id, artist_id, city, region_country))


            conn.commit()
    else:
        print(f"No events found for {artist}")
```

We wrote a function called gather and store which takes in a cursor: c, a connection object: conn, an apikey: apikey, and a list of artists: artists. This function procured which artist from the list artists we work with on that specific run. It then pulled the city, region,

country from the api for each show that artist had and stored those values into the helper tables which prevented any possible duplicate string data region_country and cities and the overarching table tiket_master.

```python
def main():
    # Set up API and database credentials
    apikey = "wxye7zQTvRKhs2EZabVNoOeOVcWAh1gU"

    # Define the artists you want to search for
    artists = ["Taylor Swift", "Ed Sheeran", "Beyonce", "Bruno Mars", "Drake", "Khalid", "Lady Gaga", "Katy Perry", "Bono", "Adele"]

    # Create a SQLite database and table to store the artist-city data
    conn = sqlite3.connect('finalDatabase.db')
    c = conn.cursor()
    # c.execute('''DROP TABLE IF EXISTS ticket_master''')
    # c.execute('''DROP TABLE IF EXISTS region_country''')
    # c.execute('''DROP TABLE IF EXISTS cities''')
    c.execute('''CREATE TABLE IF NOT EXISTS region_country
                 (id integer PRIMARY KEY, name text , UNIQUE(name))''')
    c.execute('''CREATE TABLE IF NOT EXISTS cities
                 (id integer PRIMARY KEY, name text , UNIQUE(name))''')
    c.execute("""
        CREATE TABLE IF NOT EXISTS ticket_master (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            artist_id INTEGER,
            city_id INTEGER,
            region_country_id INTEGER,
            FOREIGN KEY(artist_id) REFERENCES artists(id),
            FOREIGN KEY(region_country_id) REFERENCES region_country(id),
            FOREIGN KEY(city_id) REFERENCES cities(id)
            UNIQUE(artist_id, city_id)
        )
    """)

    gather_and_store(c, conn, apikey, artists)
    # Close the database connection
    conn.close()
if __name__ == "__main__":
    main()
```

The main function takes no parameters. It establishes the apikey, the lists of artists we are to work with, and sets up a connection and cursor. It also sets up the tables then calls gather_and_store to fill them.

In visualizations.py:

```python
def relevant_df(c):
    # SQL statement to get the concerts_performed column
    c.execute("""SELECT ticket_master.artist_id, COUNT(*) as concerts_performed
    FROM ticket_master
    GROUP BY ticket_master.artist_id""")

    concerts_performed_data = c.fetchall()

    # SQL statement to get the artist name and average popularity
    c.execute("""
    SELECT artists.name, AVG(spotify.popularity) as popularity
    FROM artists JOIN spotify ON artists.id = spotify.artist_id
    GROUP BY artists.name""")

    artist_data = c.fetchall()

    # Merge the two data sets on artist_id
    result = []
    for artist in artist_data:
        artist_id = c.execute("SELECT id FROM artists WHERE name = ?", (artist[0],)).fetchone()[0]
        concerts_performed = [row[1] for row in concerts_performed_data if row[0] == artist_id][0]
        result.append((artist[0], artist[1], concerts_performed))

    df = pd.DataFrame(result, columns=['artist', 'popularity', 'concerts_performed'])
    return df
```

This code defines a function called relevant_df that takes a SQLite cursor object as input. It then retrieves data from two different tables in the SQLite database and creates a Pandas dataframe that combines this data. The first SQL query selects the count of the concerts performed by each artist, and the second SQL query selects the average popularity of each artist. The two datasets are merged based on the artist ID using a for loop that matches the artist name from the second query with the artist ID from the first query. It then combines the data using a for loop and stores the result in a list. Finally, the function converts the list to a Pandas dataframe and returns it. The resulting Pandas dataframe contains three columns: artist name, average popularity, and the number of concerts performed.

```
def write_csv(df):
    df.to_csv('output.csv', index=False)
```

This code defines a function called write_csv that takes a dataframe as input and simply writes the dataframe to a csv file.

```
def linear_regression(df):
    sns.regplot(x='concerts_performed', y='popularity', data=df)
    plt.xlabel('Concerts Performed')
    plt.ylabel('Popularity')
    plt.title('Popularity vs. Concerts Performed')

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

This code defines a function called linear_regression that takes a dataframe as input and creates a scatterplot with a linear regression line. The function uses the Seaborn library's regplot function to plot the relationship between 'concerts_performed' (x-axis) and 'popularity' (y-axis). After setting the axis labels and title, it displays the plot using plt.show().

```python
def artist_popularity(df):
    # sort the DataFrame by popularity in descending order
    sorted_df = df.sort_values(by='popularity', ascending=False)

    # create Barchart of the different artists and their popularity
    plt.subplots(figsize=(10, 7))
    sns.barplot(x='artist', y='popularity', data=sorted_df)
    plt.xticks(rotation=45, ha='right')
    plt.xlabel('Artist')
    plt.ylabel('Popularity')
    plt.title('Popularity of Different Artists')
    plt.show()
```

This code defines a function called artist_popularity that takes a dataframe as input and creates a bar chart depicting the popularity of different artists. First, it sorts the dataframe by popularity in descending order, and then it uses the Seaborn library's barplot function to create the chart with 'artist' on the x-axis and 'popularity' on the y-axis. After customizing the axis labels, title, and x-axis tick rotation, it displays the plot using plt.show().

```python
def artist_concerts(df):
    new_sorted_df = df.sort_values(by='concerts_performed', ascending=False)
    #Creating bar chart of the different artists and amount of concerts they played
    plt.subplots(figsize=(10, 7))
    sns.barplot(x='artist', y='concerts_performed', data=new_sorted_df)
    plt.xticks(rotation=45, ha='right')
    plt.xlabel('Artist')
    plt.ylabel('Number of concerts performed')
    plt.title('Number of concerts performed by each artist')
    # Show the plot
    plt.show()
```

This code defines a function called artist_concerts that takes a dataframe as input and creates a bar chart displaying the number of concerts performed by each artist. It sorts the dataframe by 'concerts_performed' in descending order, and then uses the Seaborn library's barplot function to create the chart with 'artist' on the x-axis and 'concerts_performed' on the y-axis. After customizing the axis labels, title, and x-axis tick rotation, it displays the plot using plt.show().

Documentation:

8. You must also clearly document all resources you used. The documentation should be of the following form (20 points)

Date	Issue Description	Location Resource	Result
4/15	Issue Working With Spotify API	https://spotipy.readthedocs.io/en/2.22.1/	Able to use the spotipy library to easily work with the spotify api.
4/15-4/19	Various coding confusions and troubleshoots	https://stackoverflow.com	Helpful resource to debug code and learn from people who made similar mistakes and encountered similar issues.

REPORT	**100**	
Goals	_____ 20	-10 if missing original goals -10 if missing achieved goals
Problems that you faced	_____ 10	-10 if missing
Include the calculation file	_____ 10	-10 if not included physically in the report
Include the visualizations created	_____ 10	-10 if no visualizations physically included -5 if some but not all visualizations included
Instructions for running the code	_____ 10	-5 if unclear -5 if instructions didn't work
Code documentation (explain what each function does including describing its input and output)	_____ 20	-5 for each missing function explanation (up to -20)
Documentation of resources used	_____ 20	-10 for unclear resource documentation (not following the format) -20 for no documentation