```
      1 # run this cell
----> 2 plt.plot('year', 'max_snow', data=peaks_north);
      3 plt.plot('year', 'max_snow', 'r.', data=peaks_north);
      4 plt.legend();

NameError: name 'peaks_north' is not defined
```

Part 2: The IMDB (mini) Dataset

(Click here to jump back to the top of this notebook.)

We will explore a miniature version of the IMDb Dataset. This is the same dataset that we used for this week's lab. The remainder of this overview section is copied from this week's lab.

Let's load in the database in two ways (using both Python and cell magic) so that we can flexibly explore the SQL database.

A few reminders: * **Only SQL code written with `pd.read_sql` will be graded.** You should feel free to create `%%sql` cells **after** your Python answer + autograder cells to reduce debugging headaches, but you will still need to copy over any SQL to the Python answer cells. **Do not** add new cells betwen the question and the grading cells; it will cause errors when we run the autograder, and it will sometimes cause an error in generating the PDF file.

- **Caution: Be careful with large SQL queries!!** You may need to reboot your Jupyter Hub instance if it stops responding. **Use the LIMIT keyword** to avoid printing out 100k-sized tables (but remember to remove it).
- Films and movies are equivalent ways of expressing the condition that `titleType = 'movie'`, and they are used interchangeably throughout the assignment. They refer to the same thing!

[19]:
```python
# run this cell and the next one
engine = sqlalchemy.create_engine("sqlite:///data/imdbmini.db")
connection = engine.connect()
```

[20]:
```
%sql sqlite:///data/imdbmini.db
```

Let's take a look at the table schemas:

[21]:
```
%%sql
-- just run this cell --
SELECT * FROM sqlite_master WHERE type='table';
```

```
 * sqlite:///data/imdbmini.db
Done.
```

[21]:
```
[('table', 'Title', 'Title', 2, 'CREATE TABLE "Title" (\n"tconst" INTEGER,\n
"titleType" TEXT,\n  "primaryTitle" TEXT,\n  "originalTitle" TEXT,\n  "isAdult"
TEXT,\n  "startYear" TEXT,\n  "endYear" TEXT,\n  "runtimeMinutes" TEXT,\n
"genres" TEXT\n)'),
 ('table', 'Name', 'Name', 12, 'CREATE TABLE "Name" (\n"nconst" INTEGER,\n
"primaryName" TEXT,\n  "birthYear" TEXT,\n  "deathYear" TEXT,\n
```

```
"primaryProfession" TEXT\n)'),
 ('table', 'Role', 'Role', 70, 'CREATE TABLE "Role" (\ntconst INTEGER,\nordering
TEXT,\nnconst INTEGER,\ncategory TEXT,\njob TEXT,\ncharacters TEXT\n)'),
 ('table', 'Rating', 'Rating', 41, 'CREATE TABLE "Rating" (\ntconst
INTEGER,\naverageRating TEXT,\nnumVotes TEXT\n)')]
```

From running the above cell, we see the database has 4 tables: `Name`, `Role`, `Rating`, and `Title`.

[Click to Expand] See descriptions of each table's schema.

`Name` – Contains the following information for names of people.

- nconst (text) - alphanumeric unique identifier of the name/person
- primaryName (text)– name by which the person is most often credited
- birthYear (integer) – in YYYY format
- deathYear (integer) – in YYYY format

`Role` – Contains the principal cast/crew for titles.

- tconst (text) - alphanumeric unique identifier of the title
- ordering (integer) – a number to uniquely identify rows for a given tconst
- nconst (text) - alphanumeric unique identifier of the name/person
- category (text) - the category of job that person was in
- characters (text) - the name of the character played if applicable, else '\N'

`Rating` – Contains the IMDb rating and votes information for titles.

- tconst (integer) - alphanumeric unique identifier of the title
- averageRating (text) – weighted average of all the individual user ratings
- numVotes (text) - number of votes (i.e., ratings) the title has received

`Title` - Contains the following information for titles.

- tconst (text) - alphanumeric unique identifier of the title
- titleType (text) - the type/format of the title
- primaryTitle (text) - the more popular title / the title used by the filmmakers on promotional materials at the point of release
- isAdult (text) - 0: non-adult title; 1: adult title
- startYear (text) – represents the release year of a title.
- runtimeMinutes (integer) – primary runtime of the title, in minutes

From the above descriptions, we can conclude the following: * `Name.nconst` and `Title.tconst` are primary keys of the `Name` and `Title` tables, respectively. * `Role.nconst` and `Role.tconst` are **foreign keys** that point to `Name.nconst` and `Title.tconst`, respectively.

1.8 Question 4

1.8.1 Question 4a

How far back does our data go? Does it only include recent data, or do we have information about older movies and movie stars as well?

List the **10 oldest `movie` titles** by `startYear` and then `primaryTitle` both in **ascending** order. Do not include films where the `startYear` is `NULL`. The output should contain the `startYear`, primaryTitle, and titleType.

Remember, you can create a %%sql cell **after** the grader cell as scratch work. Just be sure to copy the query back into the Python cell to run the autograder.

[22]:
```
query_q4a = """
SELECT startYear, primaryTitle, titleType
FROM Title
WHERE startYear IS NOT NULL AND titleType == 'movie'
ORDER BY startYear, primaryTitle ASC
LIMIT 10;
"""


res_q4a = pd.read_sql(query_q4a, engine)
res_q4a
```

[22]:

	startYear	primaryTitle	titleType
0	1915	The Birth of a Nation	movie
1	1920	The Cabinet of Dr. Caligari	movie
2	1921	The Kid	movie
3	1922	Nosferatu	movie
4	1924	Sherlock Jr.	movie
5	1925	Battleship Potemkin	movie
6	1925	The Gold Rush	movie
7	1926	The General	movie
8	1927	Metropolis	movie
9	1927	Sunrise	movie

[23]:
```
grader.check("q4a")
```

[23]:
```
q4a results: All test cases passed!
```

[24]:
```
%%sql
SELECT startYear, primaryTitle, titleType
FROM Title
WHERE startYear IS NOT NULL AND titleType == 'movie'
ORDER BY startYear, primaryTitle ASC
LIMIT 10;
```

```
 * sqlite:///data/imdbmini.db
Done.
```

[24]:
```
[('1915', 'The Birth of a Nation', 'movie'),
 ('1920', 'The Cabinet of Dr. Caligari', 'movie'),
 ('1921', 'The Kid', 'movie'),
 ('1922', 'Nosferatu', 'movie'),
 ('1924', 'Sherlock Jr.', 'movie'),
```

```
('1925', 'Battleship Potemkin', 'movie'),
('1925', 'The Gold Rush', 'movie'),
('1926', 'The General', 'movie'),
('1927', 'Metropolis', 'movie'),
('1927', 'Sunrise', 'movie')]
```

1.8.2 Question 4b

Next, let's calculate the distribution of films by year. Write a query that returns the **total** `movie` titles for each `startYear` in the `Title` table as `total`. Keep in mind that some entries may not have a `startYear` listed – you should filter those out. Order your final results by the `startYear` in **ascending** order.

The first few records of the table should look like the following (but you should compute the entire table).

	startYear	total
0	1915	1
1	1920	1
2	1921	1
3	1922	1
...	...	...

[25]:
```
query_q4b = """
SELECT startYear, COUNT(*) AS total
FROM Title
WHERE startYear IS NOT NULL AND titleType LIKE 'movie'
GROUP BY startYear
ORDER BY startYear;
"""


res_q4b = pd.read_sql(query_q4b, engine)
res_q4b
```

[25]:
```
    startYear  total
0        1915      1
1        1920      1
2        1921      1
3        1922      1
4        1924      1
..        ...    ...
97       2017    213
98       2018    230
99       2019    194
100      2020    117
101      2021     85
```

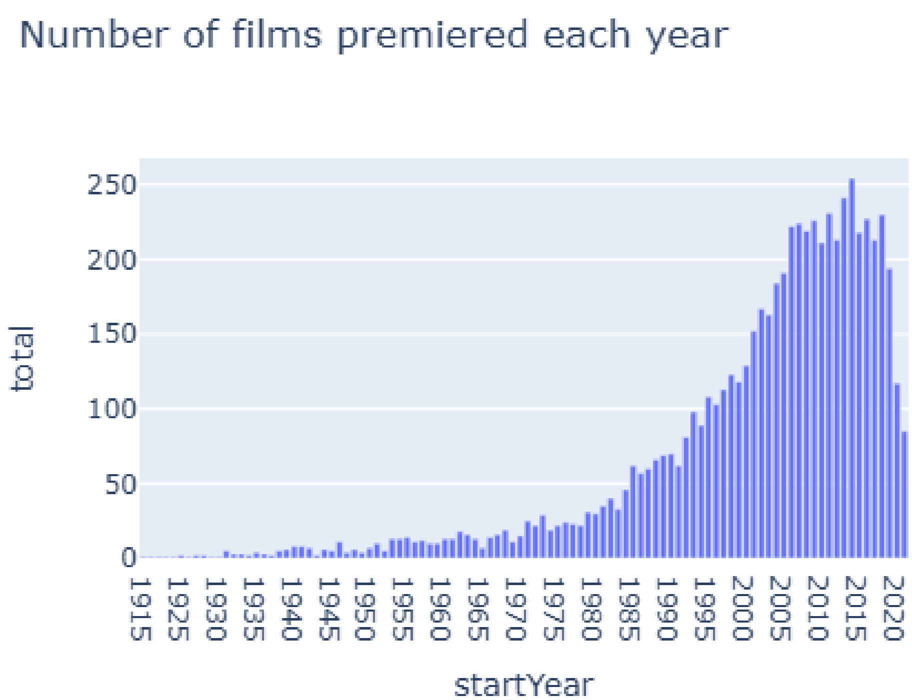
```
[102 rows x 2 columns]
```

[26]:
```python
grader.check("q4b")
```

[26]:
```
q4b results: All test cases passed!
```

The following should generate an interesting plot of the number of films that premiered each year. Notice there is a dip between the 1920s and late 1940s. Why might that be? *This question is rhetorical; you do not need to write your answer anywhere.*

[27]:
```python
# just run this cell
px.bar(res_q4b, x="startYear", y="total",
       title="Number of films premiered each year")
```



1.9 Question 5

Who are the **top 10 most prolific movie actors**?

Define the term "movie actor" is defined as anyone with an `actor` or `actress` job category role in a `movie` title.

Your SQL query should output exactly two fields named `name` (the movie actor name) and `total` (the number of movies the movie actor appears in). Order the records by `total` in descending order, and break ties by ordering by `name` in ascending order.

Your result should look something like the following, but without `????`:

	name	total
0	????	64
1	????	54
2	????	53
3	????	49
4	????	46
5	????	43
6	????	41
7	????	40
8	????	40
9	????	39

Some hints:

- ***The query should take $<$ 2 minutes to run.***
- Google the top of the list and see if it makes sense.
- If you want to include a non-aggregate field in the `SELECT` clause, it must also be included in the `GROUP BY` clause.

[28]:
```
query_q5 = """
SELECT primaryName as name, COUNT(*) AS total
FROM Role
JOIN Name ON Role.nconst = Name.nconst
JOIN Title ON Role.tconst = Title.tconst
WHERE (category LIKE '%actor%' OR category LIKE '%actress%') AND titleType LIKE␣
  ↪'%movie%'
GROUP BY primaryName
ORDER BY total DESC, name ASC
LIMIT 10;
"""


res_q5 = pd.read_sql(query_q5, engine)
res_q5
```

[28]:
```
                name  total
0     Robert De Niro     65
1  Samuel L. Jackson     55
2       Nicolas Cage     53
3      Bruce Willis     49
4          Tom Hanks     46
5        Johnny Depp     43
6      Mark Wahlberg     41
```

```
7        Liam Neeson     40
8     Morgan Freeman     40
9       Adam Sandler     39
```

[29]:
```
grader.check("q5")
```

[29]:
```
q5 results: All test cases passed!
```

1.10 Question 6: The `CASE` Keyword

The `Rating` table has the `numVotes` and the `averageRating` for each title. Which `movie` titles were **"big hits"**, defined as a movie with over 100,000 votes? Construct the following table:

	isBigHit	total
0	no	????
1	yes	????

Where `????` is replaced with the correct values. The row with `no` should have the count for how many movies **are not** big hits, and the row with `yes` should have the count of how many movies **are** big hits.

- Rating.numVotes currently consists of string objects, use `CAST(Rating.numVotes AS int)` to convert them to integer.
- You will need to use some type of `JOIN`.
- You may also consider using a `CASE WHEN ... IS ... THEN 'yes' ... ELSE ... END` statement. `CASE` statements are the SQL-equivalent of Python `if... elif... else` statements. To read up on `CASE`, take a look at the following links:
  - https://mode.com/sql-tutorial/sql-case/
  - https://www.w3schools.com/sql/sql_ref_case.asp

[30]:
```
query_q6 = """
SELECT CASE WHEN CAST(numVotes AS int) > 100000 THEN 'yes' ELSE 'no' END AS isBigHit, COUNT(*) AS total
FROM Rating
JOIN Title ON Rating.tconst = Title.tconst
WHERE titleType == 'movie'
GROUP BY isBigHit;
"""


res_q6 = pd.read_sql(query_q6, engine)
res_q6
```

[30]:

	isBigHit	total
0	no	4318
1	yes	2041

[31]:
```
grader.check("q6")
```

[31]: q6 results: All test cases passed!

1.11 Question 7

How does film length relate to ratings? To answer this question we want to bin `movie` titles by length and compute the average of the average ratings within each length bin. We will group movies by 10-minute increments – that is, one bin for movies [0, 10) minutes long, another for [10, 20) minutes, another for [20, 30) minutes, and so on. Use the following code snippet to help construct 10-minute bins:

```
ROUND(runtimeMinutes / 10.0 + 0.5) * 10 AS runtimeBin
```

Construct a table containing the **`runtimeBin`**, the **average** of the **average ratings** (as `averageRating`), the **average number of votes** (as `averageNumVotes`), and the number of `titles` in that **`runtimeBin`** (as `total`). Only include movies with **at least 10000 votes**. Order the final results by the value of `runtimeBin`.

[34]:
```
query_q7 = """
SELECT ROUND(runtimeMinutes / 10.0 + 0.5) * 10 AS runtimeBin,
    AVG(CAST(averageRating AS float)) AS averageRating,
    AVG(CAST(numVotes AS float)) AS averageNumVotes,
    COUNT(primaryTitle) AS total
FROM Title
JOIN Rating ON Title.tconst = Rating.tconst
WHERE titleType == 'movie' AND numVotes >= 10000
GROUP BY runtimeBin
ORDER BY runtimeBin;
"""



res_q7 = pd.read_sql(query_q7, engine)
res_q7.head()
```

[34]:

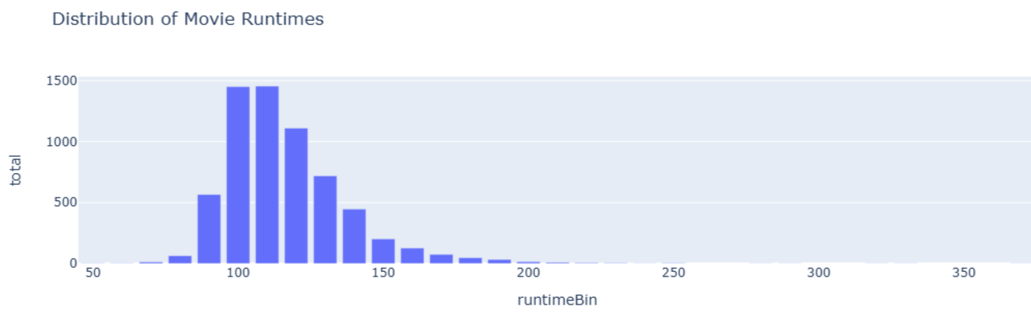
	runtimeBin	averageRating	averageNumVotes	total
0	50.0	7.850000	42535.000000	2
1	60.0	6.400000	30668.500000	2
2	70.0	7.600000	59822.000000	13
3	80.0	6.860937	67896.187500	64
4	90.0	6.283951	76907.608466	567

[35]:
```
grader.check("q7")
```

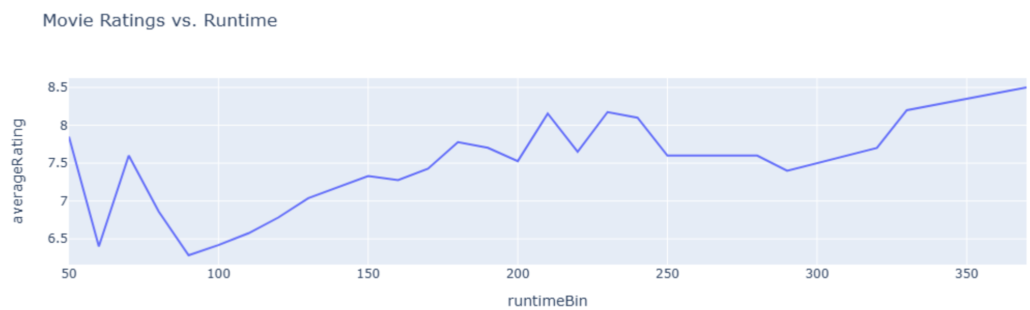
[35]: q7 results: All test cases passed!

If your SQL query is correct you should get some interesting plots below. This might explain why directors keep going a particular direction with film lengths.

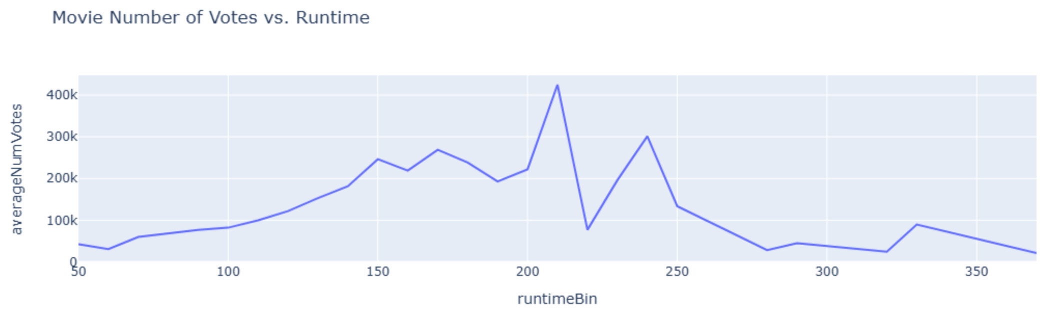
[36]:
```python
# just run this cell
px.bar(res_q7, x="runtimeBin", y="total",
       title="Distribution of Movie Runtimes")
```



[37]:
```python
# just run this cell
px.line(res_q7, x="runtimeBin", y="averageRating",
        title="Movie Ratings vs. Runtime")
```



[38]:
```python
px.line(res_q7, x="runtimeBin", y="averageNumVotes",
        title="Movie Number of Votes vs. Runtime")
```



1.12 Question 8

Which **movie actors** have the highest average ratings across all the **movies** in which they star? Again, define "movie actor" as anyone with an `actor` or `actress` job category role in a `movie` title.

Construct a table consisting of the **movie actor's name** (as `name`) and their **average actor rating** (as `actorRating`) computed by rescaling ratings for movies in which they had a role:

$$\text{actorRating} = \frac{\sum_m \text{averageRating}[m] * \text{numVotes}[m]}{\sum_m \text{numVotes}[m]}$$

Some notes: * Note that if an actor/actress has multiple `role` listings for a film then that film will have a bigger impact in the overall average (this is desired). * ***The query should take < 3 minutes to run.*** * Only consider ratings where there are **at least 1000** votes and only consider movie actors that have **at least 20 rated performances**. Present the movie actors with the **top 10** `actorRating` in descending order and break ties alphabetically using the movie actor's name.

The results should look something like this but without the `????`, and with higher rating precision.

	name	actorRating
0	????	8.4413...
1	????	8.2473...
2	????	8.1383...
3	????	8.1339...
4	????	8.0349...
5	????	7.9898...
6	????	7.9464...
7	????	7.9330...
8	????	7.9261...
9	????	7.8668...

[39]:
```
query_q8 = """
SELECT primaryName AS name, (SUM(averageRating * numVotes)/SUM(numVotes)) AS␣
↪actorRating
```

```
FROM Role
JOIN Title ON Role.tconst = Title.tconst
JOIN Name ON Role.nconst = Name.nconst
JOIN Rating ON Role.tconst = Rating.tconst
WHERE (category == 'actor' OR category == 'actress') AND titleType == 'movie'␣
  ↪AND numVotes >= 1000
GROUP BY name
HAVING COUNT(*) >= 20
ORDER BY actorRating DESC, name ASC
LIMIT 10;
"""


res_q8 = pd.read_sql(query_q8, engine)
res_q8
```

[39]:

	name	actorRating
0	Diane Keaton	8.441302
1	Tim Robbins	8.247318
2	Al Pacino	8.138361
3	Michael Caine	8.133915
4	Leonardo DiCaprio	8.034961
5	Christian Bale	7.989825
6	Robert Duvall	7.946483
7	Jack Nicholson	7.933034
8	Kevin Spacey	7.926158
9	Clint Eastwood	7.866839

[40]:
```
grader.check("q8")
```

[40]: q8 results: All test cases passed!

1.13 Congratulations!

Congrats! You are finished with this homework assignment.

1.14 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[ ]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(run_tests=True)
```