University of Waterloo

2020 Election Project Report

ECE356 Databases
Prof. Paul Ward
Group 26

Github Repository: https://github.com/pwyq/ECE356

April 20th, 2021

Relational Schema

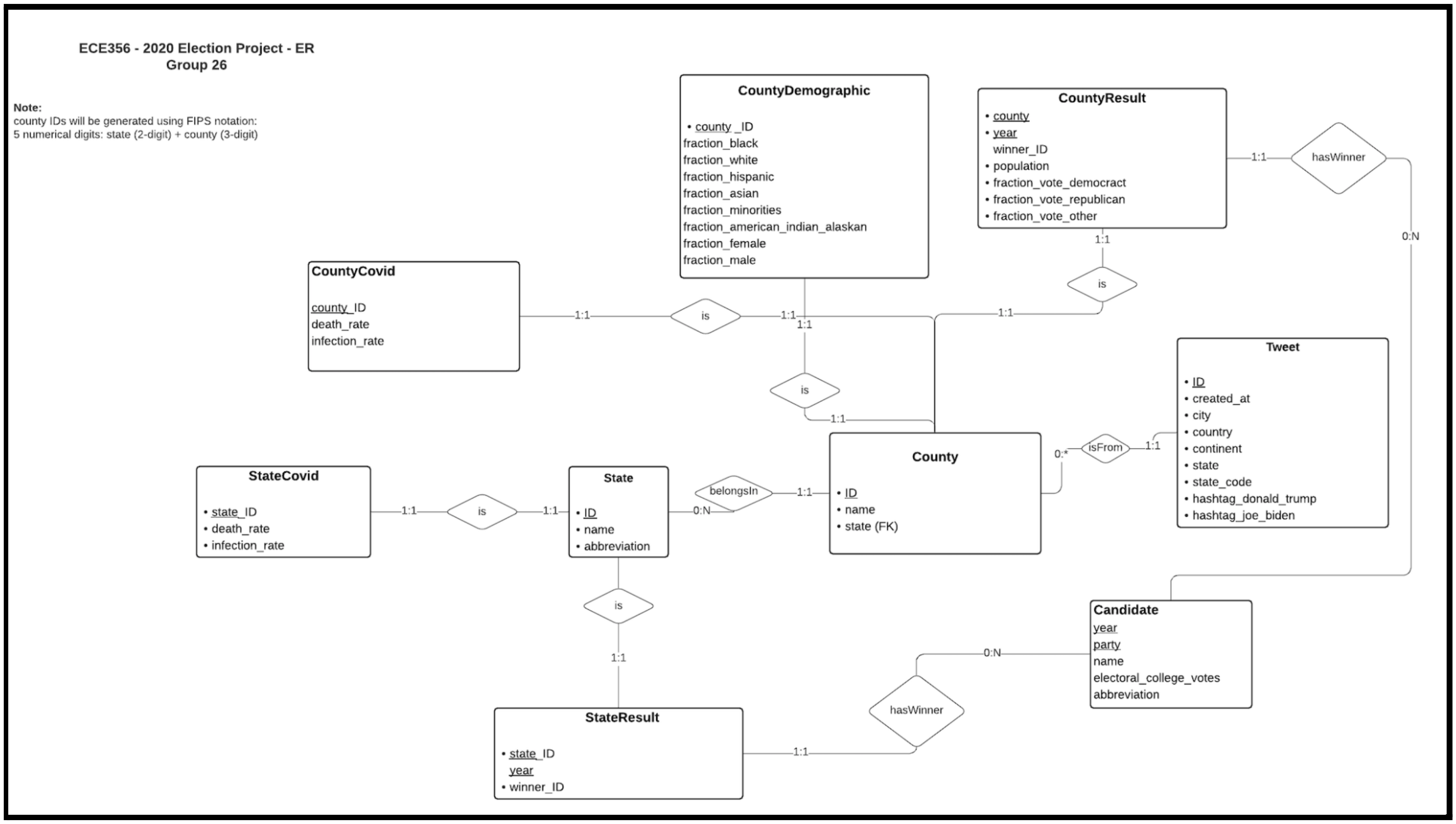


Figure 1. Relation Schema

Entity-Relation

The Entity-Relation diagram from our database is shown in the figure above. We have chosen the above entities for the purposes of our project as these were deemed necessary for the client application and data mining. Two of the entities, *State* and *County*, serve the purpose of “lookup” tables wherein they simply hold information on States and Counties as well as their FIPS codes, and act as Foreign Key references for the other tables in the database to preserve data integrity and consistency.

Most entities shown here were converted directly into tables using MySQL, specifically, *CountyCovid*, *CountyDemographic*, *CountyResult*, *Tweet*, *County*, *State*, and *Candidate*. The *StateResult* and *StateCovid* entities were implemented as Views in MySQL due to their nature: these two entities are simply aggregations of the rows in *CountyResult* and *CountyCovid* respectively. Thus, the views were created using the *CountyResult* and *CountyCovid* tables so that if these underlying tables ever change, the *StateResult* and *StateCovid* views change accordingly as well. More details on the creation of these tables and views is described below.

Table Creation and Population

To convert the entities into tables, first the data was pulled into the database from the Kaggle CSV files, for simplicity. Specifically, the following intermediate tables were created:

- _CountyStatistics
- _HashtagDonaldTrump
- _HashtagJoeBiden
- _PresidentCounty
- _PresidentCountyCandidate
- _PresidentState

And the data from the files “president_county.csv”, “president_county_candidate.csv”, “president_state.csv”, “county_statistics.csv”, “hashtag_donaldtrump.csv”, and “hashtag_joebiden.csv” was imported directly into these tables with minimal cleanup at this stage. Some fields from the “hashtag_donaldtrump.csv”, “hashtag_joebiden.csv” and “county_statistics.csv” were omitted during the file load using *@ignore* on those fields.

In order to determine the types of the columns in these intermediate tables, the CSV files were manually examined. For example, the *county* field in *_CountyStatistics* was given a type of *varchar(100)* to ensure that there is enough room to fit every county name from the Kaggle dataset since these were reported as strings.

Next, we created the State and County lookup tables. These were inserted using an external data source found here: https://raw.githubusercontent.com/kjhealy/fips-codes/master/state_and_county_fips_master.csv.

Next, these intermediate tables were used to populate our real tables by using “INSERT INTO…” and doing an necessary joins and cleanups. All of this can be seen in the “src/table_creation/create_tables_and_load_data.sql” file on our GitHub repository.

Constraints and Dependencies

Some obvious constraints include Primary and Foreign Keys added to all tables that share information. For example, any table that holds county data such as *CountyResult*, *CountyCovid*, *CountyDemographic* has a column called *county_ID* which is a foreign key referencing the *County* lookup table’s *ID* column, which is *County*’s primary key. These foreign keys and primary keys can be seen in our table creation script “load_data.sql” which can be seen in the GitHub repository.

Some other constraints and dependencies were enforced via “checks” added to the tables:

- In the *CountyResult* table, the *fraction_vote_other* column is determined by the *fraction_vote_dem* and *fraction_vote_rep* columns since all 3 columns must add to 1. In other words, the following dependency:
  (*fraction_vote_dem*, *fraction_vote_rep*) -> *fraction_vote_other*
  This was enforced by the following check:
  ```
  ALTER TABLE CountyResult ADD CONSTRAINT CHECK(
        fraction_vote_other = 1 - fraction_vote_dem - fraction_vote_rep
  );
  ```
- Similarly, all three ‘*fraction_vote_%*` columns in the *CountyResult* table have a check for their values to be <= 1
- The *total_votes* column in *CountyResult* has a check for being >= 0 (since it cannot be a negative number)
- The FIPS lookup tables *State* and *County* also have a check to ensure correct FIPS where the FIPS for a county is <2_digit_state_FIPS><3_digit_county_code>. In other words, the first 2 digits of each county’s

FIPS must match the code of the corresponding state's FIPS to which the county belongs. This is enforced by the following check:

```
ALTER TABLE County ADD CONSTRAINT CHECK(
      SUBSTRING(CAST(LPAD(ID, 5, 0) AS CHAR), 1, 2) =
      CAST(LPAD(state_ID, 2, 0) AS CHAR)
);
```

The LPAD function in this check is used because some FIPS codes have leading '0' which cannot be stored as such in a MySQL INT column.

Indexes

A composite index was added to the *Tweet* table on the *country* column since our data mining application requires some filtration of Tweets by country, as well as by country, then state. The composite index was created as follows:

```
CREATE INDEX tweet_country_state_idx ON Tweet (country, state);
```

This way, a query that only filters on *country* will be able to make use of this index. And a query that filters by both country and state will also be able to use this index. For example,

```
EXPLAIN ANALYZE SELECT * FROM Tweet WHERE country='Germany';
+-----------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                   |
+-----------------------------------------------------------------------------------------------------------+
| -> Index lookup on Tweet using tweet_country_state_idx (country='Germany')  (cost=... rows=138516) (actual time=...) |
+-----------------------------------------------------------------------------------------------------------+

mysql> EXPLAIN ANALYZE SELECT * FROM Tweet WHERE country='United States of America' AND state='Alabama';
+-----------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                     |
+-----------------------------------------------------------------------------------------------------------------------------+
| -> Index lookup on Tweet using tweet_country_state_idx (country='United States of America', state='Alabama')  (cost=...) (actual time=...) |
+-----------------------------------------------------------------------------------------------------------------------------+
```

Another index was added to the Candidate table, on the *abbreviation* column because our client application provides a user with the ability to filter election results by political party. The index was created as follows:

CREATE INDEX candidate_partyabbr_index ON Candidate (abbreviation);

And an example query that would make use of this index is the following query, where a user wants to see information on all the counties from all elections that have :

SELECT * FROM CountyResult INNER JOIN Candidate C ON C.ID = CountyResult.winner_ID WHERE abbreviation = 'DEM';

Prior to the addition of the index, the engine executed the query as follows:

```
+-----------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                     |
+-----------------------------------------------------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=832.78 rows=7441) (actual time=0.362..2.849 rows=977 loops=1)
|    -> Filter: (C.abbreviation = 'DEM')  (cost=...) (actual time=...)
|       -> Table scan on C  (cost=6.35 rows=61) (actual time=0.074...)
|    -> Index lookup on CountyResult using winner_ID (winner_ID=C.ID)  (cost=...) (actual time=...)
+-----------------------------------------------------------------------------------------------------------------------------+
```

After the addition of the index, the engine was able to skip the table scan on C

```
+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                    |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=407.49 rows=3659) (actual time=0.318..2.665 rows=977 loops=1)
|    -> Index lookup on C using candidate_partyabbr_index (abbreviation='DEM'), with index condition: (C.abbreviation = 'DEM')  (cost=1.05 rows=3) (actual time=0.046..)
|   -> Index lookup on CountyResult using winner_ID (winner_ID=C.ID)  (cost=...) (actual time=...)
+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

Although the performance boost was not very large, it could become more significant if the Candidate table ever grows to accommodate future election years. In addition, the actual time did decrease from 0.362 to 0.318 which can accumulate, especially if there are many concurrent queries.

Another popular query that we anticipate is filtering election results by state. However, we chose not to add an index to the *State* table since it is a small table (~50 rows) and we do not expect it to ever grow. Thus, an index on state name would not have much effect here.

It may also be common for a user to want to see the results of a specific county by filtering the election results by county name. The *County* table was created with the following unique constraint:

```
UNIQUE (name, state_ID)
```

since no state should have two counties with the same name. This unique constraint also acts as an index when filtering by County *name*, such as with the following query:

```
SELECT * FROM CountyResult
INNER JOIN County C
ON C.ID = CountyResult.county_ID
WHERE C.name = 'Bullock';
```

Running explain analyze on the above showed that the index on the name column was used:

```
+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                    |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=1.22 rows=2) (actual time=0.046..0.052 rows=2 loops=1)
   -> Index lookup on C using name (name='Bullock')  (cost=0.77 rows=1) (actual time=0.020..0.022 rows=1 loops=1)
   -> Index lookup on CountyResult using PRIMARY (county_ID=C.ID)  (cost=0.45 rows=2) (actual time=...)
+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

A benefit of the design of the table schemas was the way the Primary and Foreign Keys were created on our tables. These Primary and Foreign Keys are also used by the engine for quick lookups very often, and this allows our queries to be performant.

Client Application

The client application is a simple CLI application written in python using the peewee library. The client application accepts input from the user through the command line and interacts with the accepted input to provide useful information.

```
~/School/ECE356/project/ECE356/ClientApp master  ❯ python3.8 main.py
Please enter a user name (without spaces) so you can annotate!
Please enter your user name: manishjha

 Enter a number to select the option!
1) Get county stats from year 2016 or 2020
2) Get state stats from year 2016 or 2020
3) Get number of deaths and cases covid stats for counties
4) Get number of deaths and cases covid stats for states
5) Get election results from counties with county demographics for year 2020
6) Show all counties
7) Show all states
8) Show all candidates
9) Show all parties
10) Add annotations
0) Exit the program

 Please enter a value: 
```

As you can see above a series of print statements appear on the command line and prompt the user to input various commands. We allow the users of the client application to annotate whatever they'd like based on counties, states or neither and to be able to retrieve the annotations at a later date. To enable this we prompt the user to enter the username so we can retrieve them at a later time. We used peewee library to connect to the MYSQL client and enable the use of ORM to make database transactions simpler.

We allow users to filter on various facets of data silos as you can see above in the screenshot.
In the screenshot below we can see other screens that appear to let the user further filter on the selected dataset. For eg: for election results from counties with demographic filtering you can filter on unemployment rate, percent poverty, income, ethnicity, etc.

```
~/School/ECE356/project/ECE356/ClientApp master ❯ python3.8 main.py
Please enter a user name (without spaces) so you can annotate!
Please enter your user name: manishjha

 Enter a number to select the option!
1) Get county stats from year 2016 or 2020
2) Get state stats from year 2016 or 2020
3) Get number of deaths and cases covid stats for counties
4) Get number of deaths and cases covid stats for states
5) Get election results from counties with county demographics for year 2020
6) Show all counties
7) Show all states
8) Show all candidates
9) Show all parties
10) Add annotations
0) Exit the program

 Please enter a value: 5
1) Filter by unemployment rate
2) Filter by percent poverty
3) Filter by median income
4) Filter by percent white population
5) Filter by percent black population
6) Filter by percent hispanic population
7) Filter by percent native population
8) Filter by percent asian population
9) Filter by county name
0) Go back to main menu

 Please enter a value:
```

Test Cases

We wrote several test cases to test the input of the user using the unittest module in python.
We also did manual testing to ensure that the results from the database are valid.

Data Mining

In our MySQL database, we have data regarding tweet status and county demographics (including income, race ethnicity and covid statistics). We therefore want to study how to use the data to predict the state electoral result.

First, we investigated if there is a correlation between sender location and state electoral result. We examined the tweets in two parts, sender in US and sender outside US. For non-US senders, the top 5 non-us countries composite approximately 50% of all tweets, as shown below. Interestingly, senders from India are in particular favor for Joe Biden in terms of tweet count, which we believe is partially due to the fact that Kamala Harris, the running mate of Joe Biden, is from India.

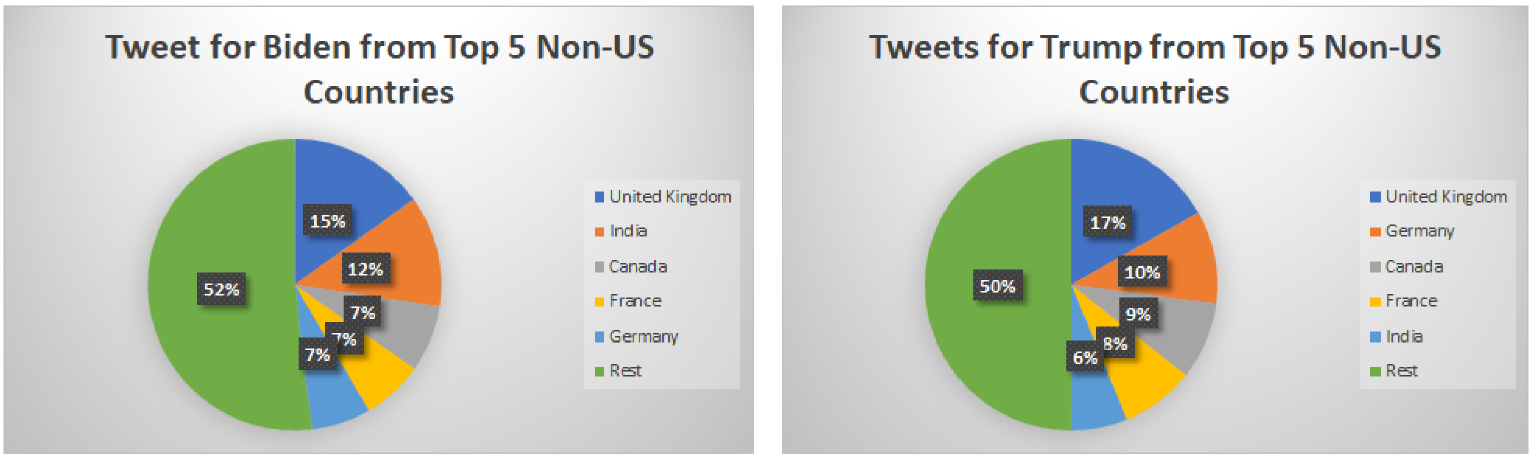


Figure 2. Tweet Count from Non-US Countries

For US senders, we calculate a score for each state based on a tweet hashtag (Trump vs. Biden). We used normalization to scale the data so we can compare all states. We experimented with two data normalization methods: (1) min-max normalization, (2) percentage normalization. A reduced sample output from (1) is shown below.

```
+----------------------+--------------+--------------+
| state                | ActualWinner | Winner       |
+----------------------+--------------+--------------+
| Alabama              | Donald Trump | Joe Biden    |
| Alaska               | Donald Trump | Joe Biden    |
| Arizona              | Joe Biden    | Joe Biden    |
| Arkansas             | Donald Trump | Donald Trump |
...
| Nevada               | Joe Biden    | Joe Biden    |
| New Hampshire        | Joe Biden    | Joe Biden    |
| New Jersey           | Joe Biden    | Donald Trump |
| New Mexico           | Joe Biden    | Joe Biden    |
```

New York	Joe Biden	Joe Biden
North Carolina	Donald Trump	Joe Biden
North Dakota	Donald Trump	Donald Trump
...		
Virginia	Joe Biden	Joe Biden
Washington	Joe Biden	Donald Trump
West Virginia	Donald Trump	Joe Biden
Wisconsin	Joe Biden	Joe Biden
Wyoming	Donald Trump	Donald Trump

The result is verified by comparing to the benchmark result. The performance is undesirable as the accuracy for the two methods are only 41% - 43% (even worse than flip a coin). We concluded that using tweet status (such as timestamp, count, and location) is insufficient to make a sound prediction.

We then experimented with more data, involving the *CountyDemographcis* table (including gender, income, race ethnicity, and professions). We pre-processed all the data to have the same presentation (i.e. percentage). We researched and implemented a weighted score system for the two candidates, where the weight is decided from a trustworthy national polls [1]. A full output form is shown below.

state	ActualWinner	Winner	TrumpScore	BidenScore
Alabama	Donald Trump	Joe Biden	184.8587	213.8914
Alaska	Donald Trump	Joe Biden	186.9053	199.9253
Arizona	Joe Biden	Joe Biden	183.8290	213.9812
Arkansas	Donald Trump	Joe Biden	191.1761	207.1833
California	Joe Biden	Joe Biden	172.8950	221.5412
Colorado	Joe Biden	Joe Biden	191.4815	204.7278
Connecticut	Joe Biden	Joe Biden	190.3888	210.1946
Delaware	Joe Biden	Joe Biden	184.3668	213.3948
District of Columbia	Joe Biden	Joe Biden	164.1645	233.7440
Florida	Donald Trump	Joe Biden	180.8688	217.4160
Georgia	Joe Biden	Joe Biden	177.0283	221.0074
Hawaii	Joe Biden	Joe Biden	160.4499	215.1392
Idaho	Donald Trump	Donald Trump	199.9543	198.0153
Illinois	Joe Biden	Joe Biden	185.5895	212.6136
Indiana	Donald Trump	Joe Biden	196.7199	201.4396
Iowa	Donald Trump	Donald Trump	202.3312	196.6802
Kansas	Donald Trump	Joe Biden	195.3123	200.8939
Kentucky	Donald Trump	Donald Trump	199.5372	197.7420

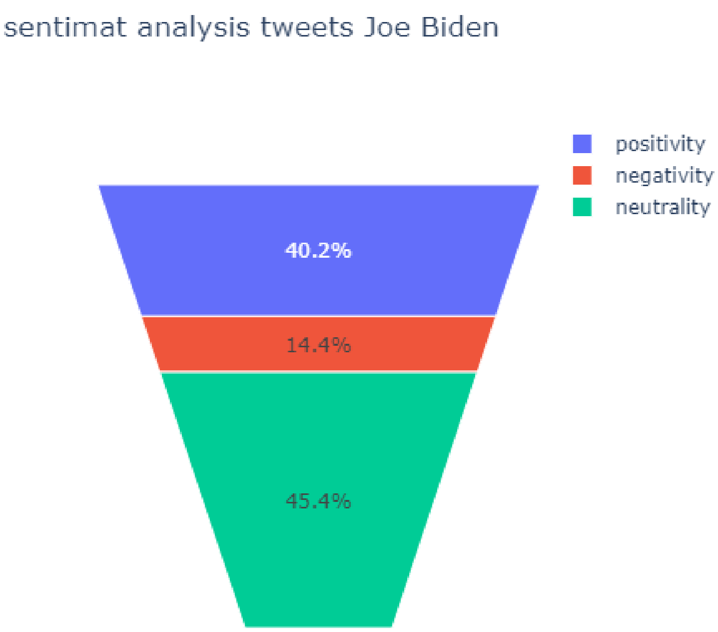
Maine	Joe Biden	Donald Trump	206.1669	190.2591
Maryland	Joe Biden	Joe Biden	176.4846	219.9760
Massachusetts	Joe Biden	Joe Biden	193.9614	206.6337
Michigan	Joe Biden	Joe Biden	193.1564	205.2879
Minnesota	Joe Biden	Joe Biden	198.3340	198.5675
Mississippi	Donald Trump	Joe Biden	177.7328	222.7619
Missouri	Donald Trump	Joe Biden	196.0558	201.1914
Montana	Donald Trump	Donald Trump	202.9257	194.2924
Nebraska	Donald Trump	Joe Biden	198.0876	199.4948
Nevada	Joe Biden	Joe Biden	179.6550	216.0065
New Hampshire	Joe Biden	Donald Trump	205.4636	193.3534
New Jersey	Joe Biden	Joe Biden	183.1272	218.2764
New Mexico	Joe Biden	Joe Biden	175.1327	220.9408
New York	Joe Biden	Joe Biden	182.4347	218.4253
North Carolina	Donald Trump	Joe Biden	184.6296	212.2295
North Dakota	Donald Trump	Donald Trump	203.0411	197.1777
Ohio	Donald Trump	Joe Biden	195.7284	201.4270
Oklahoma	Donald Trump	Joe Biden	188.1306	203.9315
Oregon	Joe Biden	Joe Biden	195.5080	198.4363
Pennsylvania	Joe Biden	Joe Biden	195.0966	203.8306
Rhode Island	Joe Biden	Joe Biden	193.7890	204.8503
South Carolina	Donald Trump	Joe Biden	183.6768	215.1163
South Dakota	Donald Trump	Donald Trump	201.8421	197.2111
Tennessee	Donald Trump	Joe Biden	191.8220	206.5175
Texas	Donald Trump	Joe Biden	174.9634	222.4089
Utah	Donald Trump	Joe Biden	197.8789	198.6639
Vermont	Joe Biden	Donald Trump	206.2913	190.4093
Virginia	Joe Biden	Joe Biden	184.4461	210.6716
Washington	Joe Biden	Joe Biden	191.3353	202.3643
West Virginia	Donald Trump	Donald Trump	204.2861	192.8732
Wisconsin	Joe Biden	Joe Biden	198.9393	199.2224
Wyoming	Donald Trump	Donald Trump	201.2644	194.8793

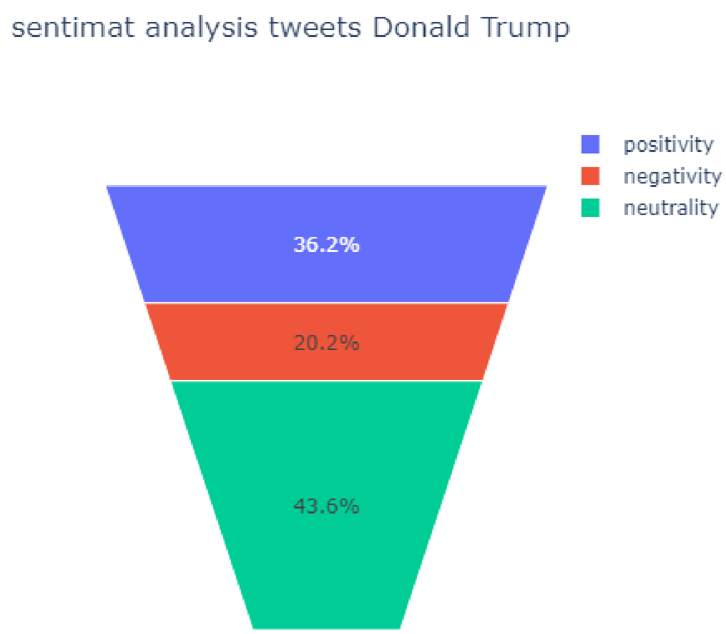
We validate the model by again comparing the prediction winner to the actual winner. With a weighted score system and more data attributes, the accuracy is improved from around 40% to 61%. Given that only a small set of attributes from the national polls is used, we conclude that if more attributes are provided, we can have much more accurate results.

In addition to mining in SQL, we experimented sentiment analysis in Python on Kaggle (the Notebook code is on our GitHub src/data_mining/election-tweet-sentiment-analysis.ipynb).

We used Python third-parties libraries Natural Language Toolkit (nltk) for the task. We measure each tweet by ‘subjectivity’ and ‘polarity’. Polarity is a float which lies in the range of [-1,1] where 1 means positive statement, -1 means a negative statement and 0 means neutral statement. Subjective sentences generally refer to personal opinions. Subjectivity is also a float which lies in the range of [0,1].

We again predicted the electoral result of each state, where the winner has higher positive sentiment than his opponent. Surprisingly, the accuracy is around 60%, which is similar to the performance of the weighted score system. However, while we examined from national level rather than state level, we found that Joe Biden received higher positive feedback (40.2%) than Donald Trump (36.2%).





[1] “National Exit Polls: How Different Groups Voted,” *The New York Times*, 05-Jan-2021. [Online]. Available: https://www.nytimes.com/interactive/2020/11/03/us/elections/exit-polls-president.html. [Accessed: 17-Apr-2021].