Search Write Sign up Sign in

Sentiment Analysis Part 2

Exploring Text Data with MongoDB Using PyMongo

Mike Harmon · Follow
8 min read · Nov 12, 2022

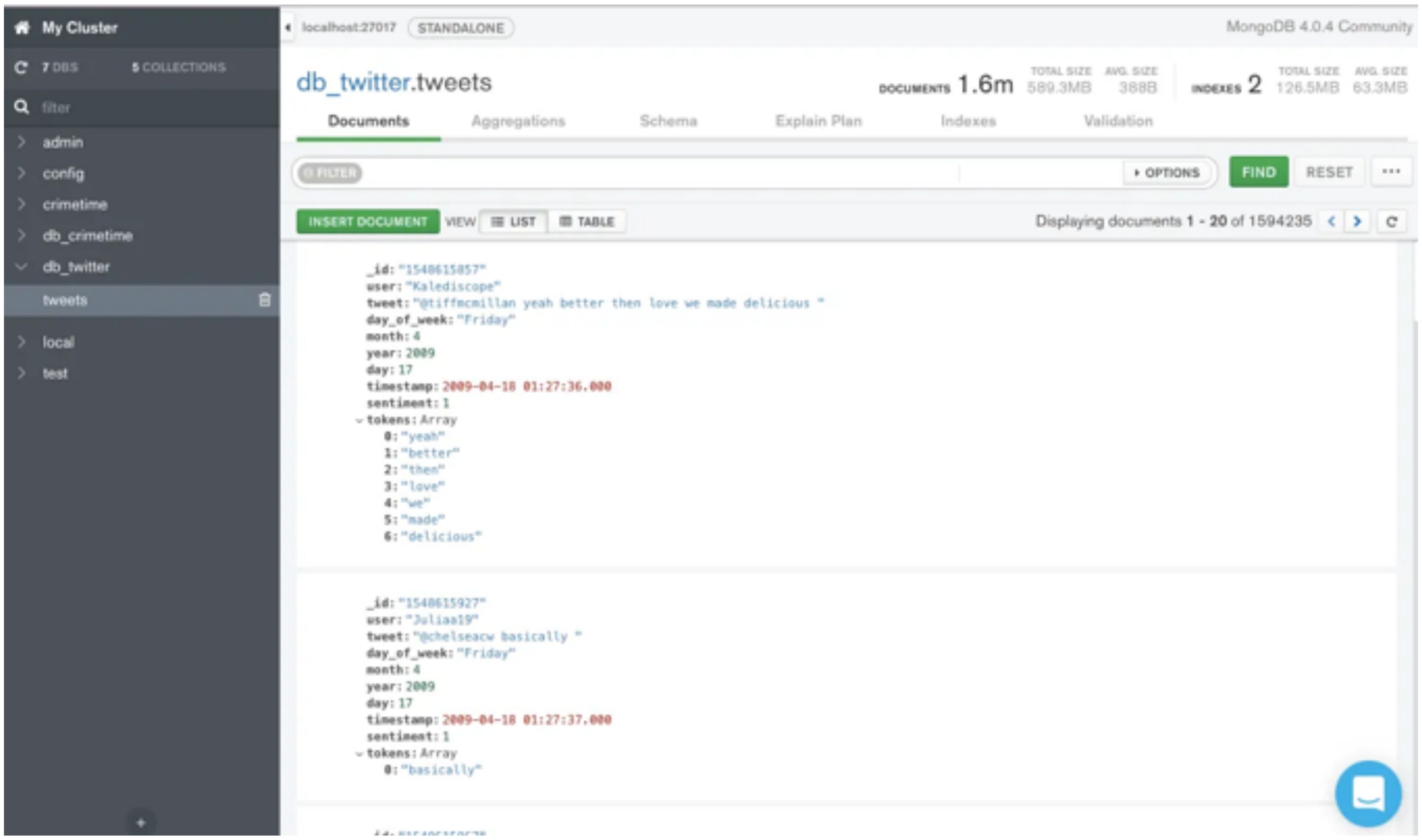
4

Originally published in 2019.

In the last post we covered how to preform ETL on text data using PySpark and MongoDB. In this post we learn how to use PyMongo to explore text in our NoSQL database. MongoDB is a document based NoSQL database that is fast, easy to use and allows for flexible schemas. I used MongoDB in this blog post since it is document based and is perfect for working with text data like tweets that may have inconsistent schemas.

Each database in MongoDB contains **collections**, each collection contains a set of **documents** that are stored as JSON objects. A collection is equivalent to a table in SQL and a document is equivalent to one record or row in that table.

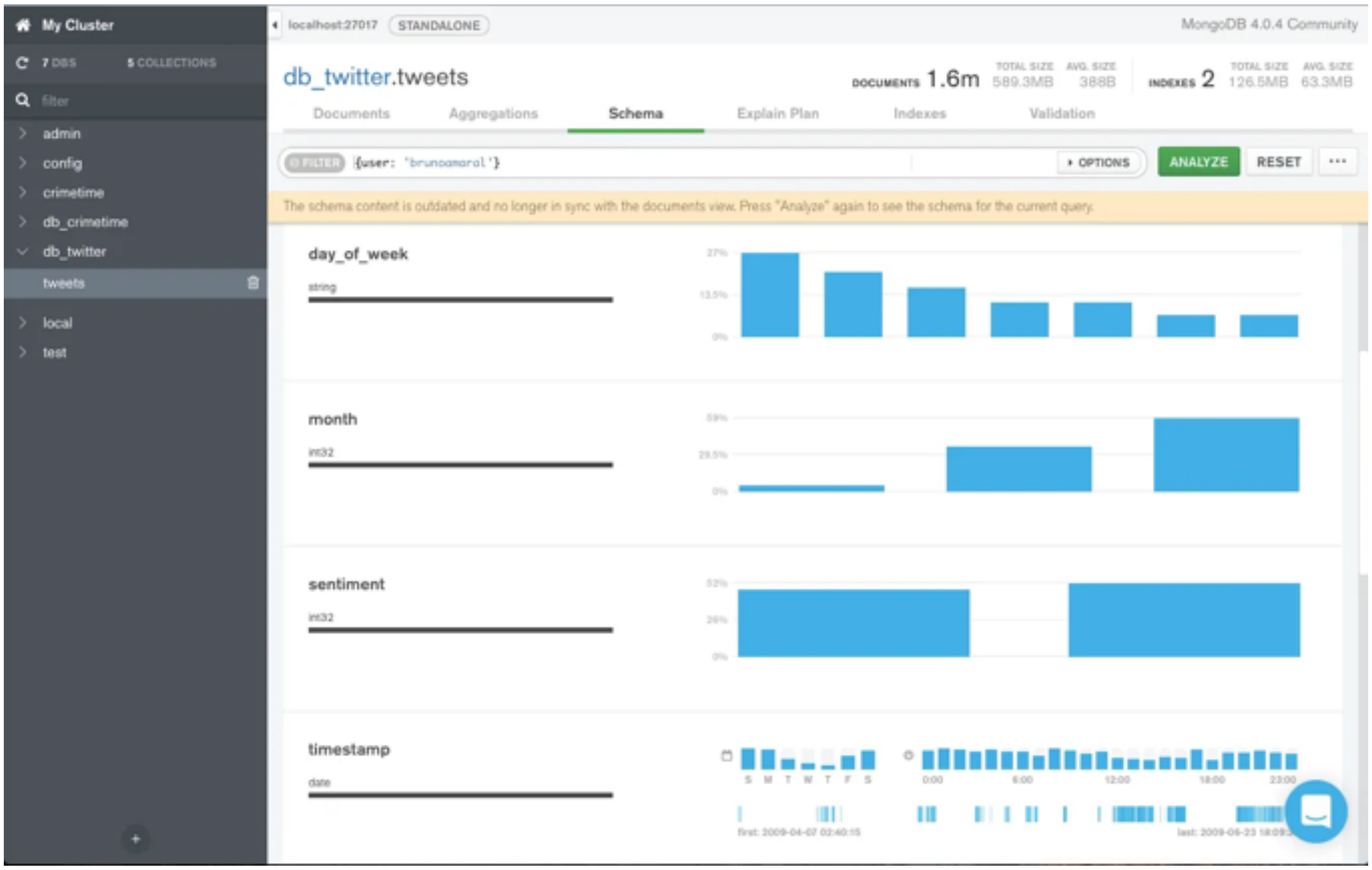
In our current example each tweet is a document in our `tweets` collection in the `db_twitter` database. MongoDB has nice GUI called Compass. An example view of our `tweets` collection using Compass is shown below:



```
_id: "1548615857"
user: "Kalediscope"
tweet: "@tiffmcmillan yeah better then love we made delicious "
day_of_week: "Friday"
month: 4
year: 2009
day: 17
timestamp: 2009-04-18 01:27:36.000
sentiment: 1
tokens: Array
  0: "yeah"
  1: "better"
  2: "then"
  3: "love"
  4: "we"
  5: "made"
  6: "delicious"
```

```
_id: "1548615927"
user: "Juliaa19"
tweet: "@chelseacw basically "
day_of_week: "Friday"
month: 4
year: 2009
day: 17
timestamp: 2009-04-18 01:27:37.000
sentiment: 1
tokens: Array
  0: "basically"
```

Compass gives a nice interface to our database. It allows us to run interactive queries on collections and displays easy to read results. One of the most useful features is the ability to analyze your schema; as shown below:



This utility samples our collection to determine the datatypes and values of the fields in your documents. The ability to discern datatypes is especially useful for this type of NoSQL database because fields can have multiple different datatypes. This is in contrast to traditional SQL databases where the entries in tables must rigidly adhere to the defined datatype of that field.

Besides, interacting with a MongoDB database through Compass one can instead use the Mongo Shell, however we will not go over in this feature in this post (except for using it to create an index). Instead we'll use the PyMongo driver which allows us to connect to our Mongo server using Python.

First we import the PyMongo module:

```
import pymongo
```

Then we can connect to our Mongo server and `db_twitter` database:

```
# connect to the mongo
conn = pymongo.MongoClient('mongodb://localhost:27017')

# connect to the twitter database
db = conn.db_twitter
```

We can now get the `tweets` collections with the following:

```
tweets = db.tweets
```

Mongo uses JavaScript as it's query language so our queries input and outputs are JSON. In Python we will use dictionaries as the equivalent data structure to JSON. We can run a first example query below:

```
query      = {"day_of_week": "Monday"}
```

This is a simple filter query where we scan `tweets` collection and only return documents that occurred on `Monday`,

```
presults = tweets.find(query)
```

The returned `results` variable is an **iterator** of **documents** (a cursor in Mongo terminology). We can iterate through the first 3 documents/tweets with the following:

```
for res in results.limit(2):
    print("Document = {}\n".format(res))
```

```
Document = {'_id': '1467810369', 'user': '_TheSpecialOne_', 'tweet': "@switchfo

Document = {'_id': '1467810672', 'user': 'scotthamilton', 'tweet': "is upset th
```

The `_id` field is the unique identifier for each document and has been set to the tweet id in this case. As you can see the resulting documents contain all the fields, if instead we wanted only a subset of the fields we can use a projection. Projections list fields to show with the name followed by a 1 and the those not to show with their name followed by a 0:

```
projection = {"_id":0, "user":1, "tweet":1}

# use the same query as before, but with a projection operator as second arguem
results = tweets.find(query, projection)

# print first three results again
for res in results.limit(3):
    print("Document = {}\n".format(res))
```

```
Document = {'user': '_TheSpecialOne_', 'tweet': "@switchfoot http://twitpic.com

Document = {'user': 'scotthamilton', 'tweet': "is upset that he can't update hi

Document = {'user': 'mattycus', 'tweet': '@Kenichan I dived many times for the
```

With a projection the `_id` field is shown by default and needs to be explicitly suppressed. All other fields in the document are by default suppressed and need a 1 after them to be displayed.

The filter above was just a simple string matching query. If we wanted to find those tweets that occurred on a Monday and in the months before June I would write:

```
# filter tweets to be on Monday and months before June.
query      = {"day_of_week":"Monday", "month":{"$lt":6}}

# only include the user name, month and text
projection = {"_id":0, "user":1, "month":1, "tweet":1}
```

```
results = tweets.find(query, projection)

for res in results.limit(3):
    print("Document = {}\n".format(res))
```

```
Document = {'user': '_TheSpecialOne_', 'tweet': "@switchfoot http://twitpic.com

Document = {'user': 'scotthamilton', 'tweet': "is upset that he can't update hi

Document = {'user': 'mattycus', 'tweet': '@Kenichan I dived many times for the
```

We can perform a simple aggregation such as finding out the number of tweets that correspond to each sentiment. To do so we use a `$group` operator in the first line in our query. The field that we group by in this case is the `sentiment` field:

```
# Simple groupby example query
count_sentiment = {"$group":
                      {"_id" : {"sentiment":"$sentiment"},  # note use a $ on th
                       "ct"  : {"$sum":1}
                      }
                  }
```

The result of this query will be a new document, and this is the reason we need a `_id`, with the resulting value of the key-value pairs `{'sentiment': value}` and `{'ct': value}`. The first key-value pair's value will either be 0 or 1 and the second key-value pair's value will be the number of tweets with that sentiment.

We can then run the query using the aggregate method.

```
results = tweets.aggregate([count_sentiment], allowDiskUse=True)

for res in results:
    print(res)
```

```
{'_id': {'sentiment': 0}, 'ct': 797066}
{'_id': {'sentiment': 1}, 'ct': 797169}
```

First notice that the query is within an array; this allows us to run multiple aggregation queries or '*stages in our aggregation pipeline*'. **By default each stage in the pipeline can only use 180mb of memory, so in order to run larger queries we must set `allowDiskUse=True` to allow the calculations to spill over onto disk.**

From this query we can see that, **our data set is actually quite well balanced, meaning the number of positive and negative tweets are about the same.**

Next we show an example of an aggregation pipeline. The first stage in the pipeline groups the months, and therefore gets the unique months in our dataset:

```
# get the unique months
get_months = {"$group": {"_id": "$month"} }
```

The second stage in the pipeline is a projection, which changes the structure of the resulting document:

```
rename_id  = {"$project":
                  {"_id":0,
                   "month":"$_id"
                }
             }
```

In this case the projection operator (`$project`) suppresses the original `_id` field for the resulting document of stage one and instead defines the new `month` key which uses the `_id` value (`$_id` operator). We can run this query and see the results:

```
results = tweets.aggregate([get_months, rename_id])

for res in results:
    print(res)
```

```
{'month': 5}
{'month': 6}
{'month': 4}
```

We can see our Twitter database only has the months: April, May and June.

Another example of multiple aggregations is to use the same group-by-count query as above, but filtering first on the month. This dataset only has 3 months of tweets in it and we can use a `$match` operator to first filter our data to only consider the month of June and then count the number of tweets that occurred in June by using the same query (`count_sentiment`) as above. The query which performs the match is:

```
match_query = {"$match": {"month":6}}
```

We can then run the full pipeline and print the number of tweets that were positive and negative in the month of June:

```
results = tweets.aggregate([match_query, count_sentiment], allowDiskUse=True)

for res in results:
    print(res)
```

```
{'_id': {'sentiment': 1}, 'ct': 388416}
{'_id': {'sentiment': 0}, 'ct': 531999}
```

The last topic I will discuss in the Mongo query language is the topic of indexing. Indexing a collection allows for more efficient queries against it. For instance if we wanted to find tweets which occurred on a certain date we could write a filter query for it. To execute the query Mongo has to scan the entire collection to find tweets that occurred on that day. If we create an index for our `tweets` collection by the date we create a natural ordering on the date field. Queries on that field will be much faster since there is now an ordering and entire collection scan is no longer needed. **You can have indexes on all sorts of fields, however,** ***your index must be able to fit into memory or else it defeats the purpose of having fast look ups.***

One extremely useful indexing scheme is indexing on the text of the documents in your collection. We can index the `tweet` field our `tweets` collection as shown from the Mongo shell below, Once the index is created you will see:

Notice that before creating the above index we actually had one index, but after we have two. The reason is that we have index before indexing our collection is because every collection is by default indexed on the `_id` field. This also shows us that collections can have multiple indices.

We can now search all our tweets for the phrase "Obama" relatively quickly using the query format:

```
{"$text": {"$search": phrase_to_search}}
```

We note that the search capabilities ignore stop words, capitalization and punctuation. We show the results below.

```
search_query = {"$text": {"$search":"obama"} }
projection   = {"_id":0, "user":1, "tweet":1}

# sort the results based on the timestamp
results = db.tweets.find(search_query, projection)\
                       .sort('timestamp', pymongo.ASCENDING)

print("Number of tweets with obama is great: ", results.count())
```

```
Number of tweets with obama is great:  448
```

```
for res in results.limit(2):
    print("Document = {}".format(res) + "\n")
```

```
Document = {'user': 'brasten', 'tweet': "@dezine it's also amusing how many peo

Document = {'user': 'haveyoumettony', 'tweet': '@GrantACummings Nah. Obama had
```

In this post we showed how one can explore the loaded data in the Mongo database using Compass and PyMongo. In the next blog post will be looking into using PySpark to model the sentiment of these tweets using PySpark!

Python Mongodb NoSQL Pymongo NLP

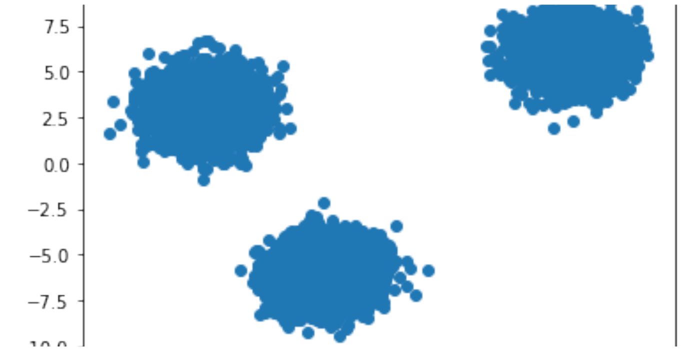
Written by Mike Harmon

Follow

11 Followers

Machine Learning Engineer, Apache Spark Lover, NLP Enthusiast, PhD Applied Math, Former Athlete

More from Mike Harmon



Mike Harmon

Scikit-Learn Compatible KMeans Esimator From Scratch

15 min read · Dec 3, 2022

6 1

date	
5 PDT 2009	[Mon, Apr, (
9 PDT 2009	[Mon, Apr, (

Mike Harmon

Sentiment Analysis, Part1: ETL With PySpark and MongoDB

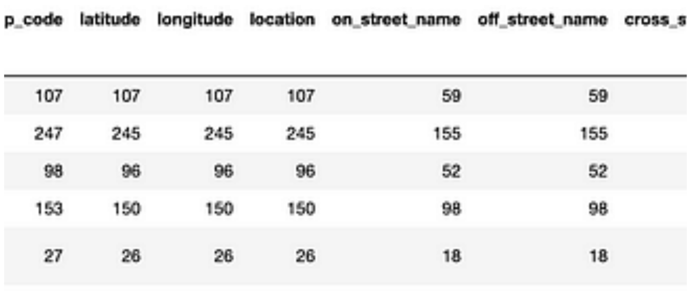
12 min read · Nov 12, 2022

7 1

```
e \

-standard-2 \
-standard-4 \

ES="python==3.5"' \
kages=org.mongodb.spark:mongo-spark-connector_2.11:2.
 gs://mikesparkinitializer/sparkenvprep.sh,gs://mikes
```



p_code	latitude	longitude	location	on_street_name	off_street_name	cross_s
107	107	107	107	59	59	
247	245	245	245	155	155	
98	96	96	96	52	52	
153	150	150	150	98	98	
27	26	26	26	18	18	

Mike Harmon

Sentiment Analysis Part 3

Machine Learning With Spark On Google Cloud

25 min read · Nov 29, 2022

2 1

Mike Harmon

Polars & DuckDB

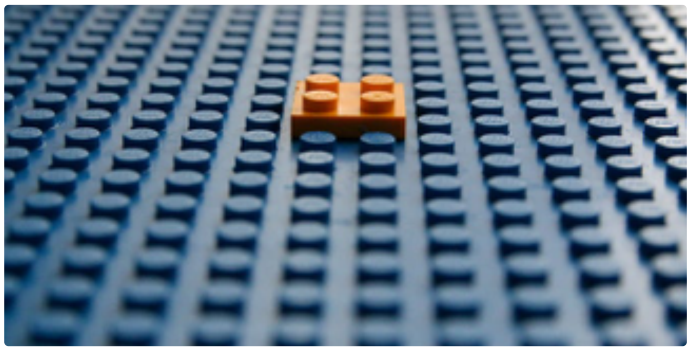
DataFrames and SQL For Python Without Pandas

17 min read · Jul 16, 2023

23

See all from Mike Harmon

Recommended from Medium



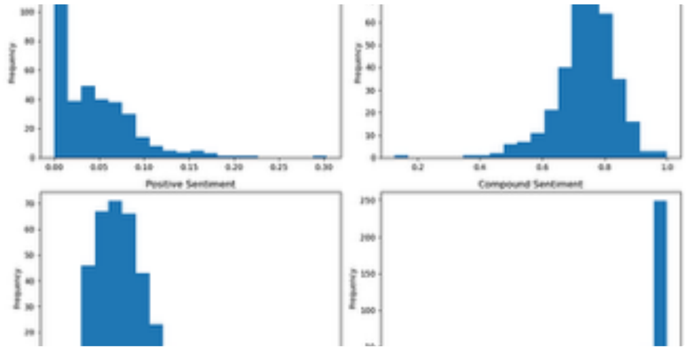
Awaldeep Singh

Understanding the Essentials: NLP Text Preprocessing Steps!

Introduction

8 min read · Dec 30, 2023

4



Chiemezie Agbo

Sentiment Analysis using Python (Automated IMDb Movie Review...

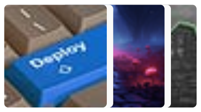
In today's digital world, there's a sea of information—reviews, comments, emails,...

8 min read · Jan 1, 2024

Lists

Coding & Development

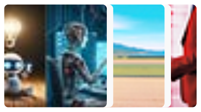
11 stories · 644 saves


Predictive Modeling w/ Python

20 stories · 1267 saves

Practical Guides to Machine Learning

10 stories · 1526 saves


Natural Language Processing

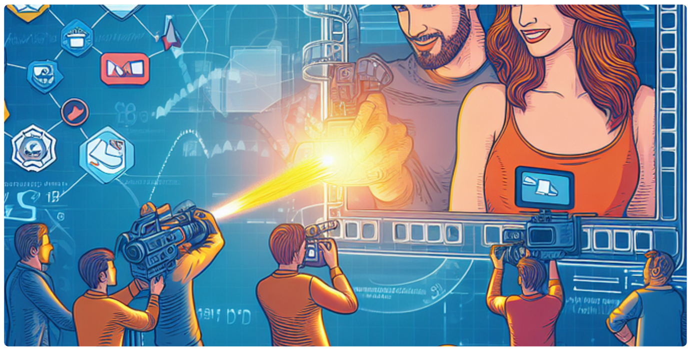
1499 stories · 1024 saves

Eulene

Sentiment Analysis of Amazon Reviews Using Natural Language...

11 min read · Feb 8, 2024

26 1

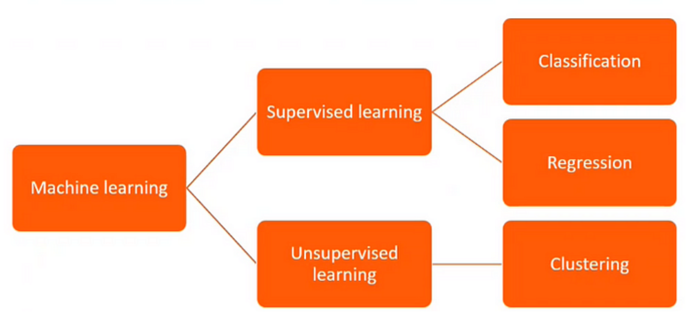

MD Khaleel Ahamed

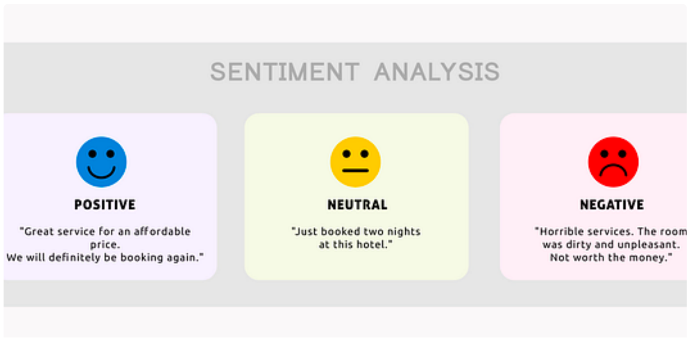
Sentiment Analysis in Action: A Case Study with Movie Reviews...

In the ever-evolving landscape of Natural Language Processing (NLP), sentiment...

7 min read · Jan 21, 2024

14




SENTIMENT ANALYSIS
POSITIVE
"Great service for an affordable price. We will definitely be booking again."
NEUTRAL
"Just booked two nights at this hotel."
NEGATIVE
"Horrible services. The room was dirty and unpleasant. Not worth the money."

Pooja

Sentiment Analysis using NLP

Sentiment Analysis helps in understanding the positive and negative intent of the…

10 min read · Jan 10, 2024

2

Anirudh Sekar

An implementation of Text and Audio Sentiment Analysis using AI

Emotion AI is a growing field, and the main forms of Emotion AI are through text and…

10 min read · Jan 27, 2024

See more recommendations

Help Status About Careers Press Blog Privacy Terms Text to speech Teams