

Sentiment Analysis Part 1

Extract-Transform-Load with PySpark and MongoDB



Mike Harmon · [Follow](#)

12 min read · Nov 12, 2022



7



1



Originally published in 2019.

Introduction

I've been itching to learn some more Natural Language Processing and thought I might try my hand at the classic problem of Twitter sentiment analysis. I found labeled twitter data with 1.6 million tweets on the Kaggle website [here](#). While 1.6 million tweets is not a substantial amount of data and does not require working with Spark, I wanted to use Spark for ETL as well as modeling since I haven't seen too many examples of how to do so in the context of Sentiment Analysis. In addition, since I was working with text data I thought I would use MongoDB, since it allows for flexible data models and is easy to use. Luckily Spark and MongoDB work well together and I'll show how to work with both later.

At first I figured I would make this one blog post, but after getting started I realized it was a significant amount of material and would break it into two posts. This first post covers the topics of ETL working with Spark and MongoDB. The second post will deal with the actual modeling of sentiment analysis using Spark. The source code for this post can be found [here](#).

ETL With PySpark

Spark is a parallel processing framework that has become the de-facto

standard in data engineering for **extract-transform-load (ETL)** operations. It has a number of features that make it great for working with large data sets including:

- Natural integration with Hadoop for working with large distributed datasets
- Fault tolerance
- Lazy evaluation that allows for behind the scenes optimizations

Spark is also great because it enables developers to use a signal framework for working with structured and unstructured data, machine learning, graph computations and even streaming. Some references that I have used for working with Spark in the past include:

- Learning Spark
- Advanced Analytics with Spark
- High Performance Spark
- This post
- The documentation webpage is pretty extensive as well

In this blog post I will **NOT** be covering the basics of Spark, there are plenty of other resources (like those above) that will a better job than I can. Instead, I want to cover the basics of working with Spark for ETL on text data. I'll explain the steps of ETL I took in detail in this post. While I used a

notebook for development, in practice I wrote a Python script that I used to perform batch analysis. You can find that script [here](#). The script was used to connect to my [Atlas MongoDB](#) cluster and I had to change the normalize UDF so that the results are strings instead of arrays of string. This was necessary so that the resulting collection was within the storage limits of the free tier.

Now let's dive into the extract-transform-load operations in Spark and MongoDB!

First we download and extract the dataset from the Kaggle website:

```
import os
os.system("kaggle datasets download -d kazanova/sentiment140")
os.system("unzip sentiment140.zip")
```

Next we import the datatypes that we will need for ETL and the functions module from `spark.sql`

```
from pyspark.sql.types import (IntegerType, StringType,
                                TimestampType, StructType,
                                StructField, ArrayType,
                                TimestampType)

import pyspark.sql.functions as F
```

Extract

Now we need to define the schema of the CSV file we want to read. Alternately, we could have Spark infer the schema, however, this would take longer since Spark would have to scan the file twice: once to infer the schema and once to read in the data.

```
schema = StructType([StructField("target", StringType()),
                        StructField("id", StringType()),
                        StructField("date", StringType()),
                        StructField("flag", StringType()),
                        StructField("user", StringType()),
                        StructField("text", StringType())
                    ])
```

Now we can define the path to the file, specify its format, schema and then “read” it in as a Dataframe. Since I am working in standalone mode on my local machine I’ll use the address of the csv in my local filesystem:

```
path = "training.1600000.processed.noemoticon.csv"

# read in the csv as a dataframe
df = (spark.read.format("csv")
      .schema(schema)
      .load(path))
```

I put read in quotations since Spark uses a **lazy-evaluation** model for computation. This means that *the csv is not actually read into the worker nodes* (see [this](#) for definition) until we perform an action on it. An action is any operation that,

- writes to disk
- brings results back to the **driver** (see [this](#) for definition), i.e. count, show, collect, toPandas,

Even though we have not read in the data, we can still obtain metadata on the dataframe such as its schema:

```
df.printSchema()
```

```
root
|-- target: string (nullable = true)
|-- id: string (nullable = true)
|-- date: string (nullable = true)
|-- flag: string (nullable = true)
|-- user: string (nullable = true)
|-- text: string (nullable = true)
```

Let's take a look at the first few rows in our dataframe:

```
df.show(3)
```

```
+-----+-----+-----+-----+-----+
|target|      id|          date|    flag|      user|
+-----+-----+-----+-----+-----+
|      0|1467810369|Mon Apr 06 22:19:...|NO_QUERY|_TheSpecialOne_|@switchfoot ht
|      0|1467810672|Mon Apr 06 22:19:...|NO_QUERY|  scotthamilton|is upset that
|      0|1467810917|Mon Apr 06 22:19:...|NO_QUERY|      mattycus|@Kenichan I di
+-----+-----+-----+-----+-----+
only showing top 3 rows
```

We can see that the table has a `target` field which is the label of whether the sentiment was positive or negative, an `id` which is a unique number for the tweet, a `date` field, a `flag` field (which we will not use), the `user` field which is the twitter user's handle and the actual tweet which is labeled as `text`. We'll have to do transformations on all the fields (except `flag` which we will drop) in order to get them into the correct format. Specifically, we will:

1. Extract relevant fields information the `date` field
2. Clean and transform the `text` field

Transformations in Spark are computed on **executor nodes** (computations in Spark occur where the data is in memory/disk which is the data nodes in Hadoop) and use lazy evaluation. The fact transformations are lazy is a very useful aspect of Spark because we can chain transformations together into **Directed Acyclic Graphs (DAG)**. Because the transformations are lazy, Spark

can see the entire pipeline of transformations and optimize the execution of operations in the DAG.

Transform

We perform most of our transformations on our Spark dataframes in this post by using **User Defined Functions or UDFs**. UDFs allow us to transform one Spark dataframe into another. UDFs act on one or more columns in a dataframe and return a column vector that we can assign as a new column to our dataframe. We'll first show how we define UDFs to extract relevant date-time information from the `date` field in our dataframe. First let's take a look at the actual date field:

```
df.select("date").take(2)
```

```
[Row(date='Mon Apr 06 22:19:45 PDT 2009'),  
 Row(date='Mon Apr 06 22:19:49 PDT 2009')]
```

Note that we could not use the `.show(N)` method and had to use the `.take(N)` method. This returns the first N rows in our dataframe as a list of Row objects; we used this method because it allows us to see the entire

string in the date field while `.show(N)` would not.

Our first transformation will take the above strings and return the day of the week associated with the date-time in that string. We write a Python function to do that:

```
def get_day_of_week(s : str) -> str:
    """
    Converts the string from the tweets to day of week by
    extracting the first three characters from the string.

    """
    day      = s[:3]
    new_day  = ""

    if day == "Sun":
        new_day = "Sunday"
    elif day == "Mon":
        new_day = "Monday"
    elif day == "Tue":
        new_day = "Tuesday"
    elif day == "Wed":
        new_day = "Wednesday"
    elif day == "Thu":
        new_day = "Thursday"
    elif day == "Fri":
        new_day = "Friday"
    else:
        new_day = "Saturday"

    return new_day
```

Next we define the desired transformation on the dataframe using a Spark UDF. UDFs look like wrappers around our Python functions with the format:

```
UDF_Name = F.udf(python_function, return_type)
```

Note that specifying the return type is not entirely necessary since Spark can infer this at runtime, however, explicitly declaring the return type does improve performance by allowing the return type to be known at compile time.

In our case the UDF for the above function becomes:

```
getDayOfWeekUDF = F.udf(get_day_of_week, StringType())
```

Now we apply the UDF to columns to our dataframes and the results are appended as a new column to our dataframe. This is efficient since Spark dataframes use column-based storage. In general we would write the transformation as:

```
df = df.withColumn("output_col", UDF_Name(df["input_col"]) )
```

With the above UDF our example becomes:

```
df = df.withColumn("day_of_week", getDayOfWeekUDF(df["date"]))
```

We can now see the results of this transformation:

```
df.select(["date", "day_of_week"]).show(3)
```

```
+-----+-----+
|          date|day_of_week|
+-----+-----+
|Mon Apr 06 22:19:...|    Monday|
|Mon Apr 06 22:19:...|    Monday|
|Mon Apr 06 22:19:...|    Monday|
+-----+-----+
only showing top 3 rows
```

(**Note:** If this were an extremely big table, a more efficient way to do this operation would be to use a regular expression to pull out the day of the week abbreviation as a new column. Then we could use a broadcast join with a look-up-table which has the day abbreviation to day of week.)

Another way to define UDFs is by defining them on Lambda functions. An example is shown below:

```
dateToArrayUDF = F.udf(lambda s : s.split(" "), ArrayType(StringType()))
```

This UDF takes the `date` field which is a string and splits the string into an array using white space as the delimiter. This was the easiest way I could think of to get the month, year, day and time information from the string in the `date` field. Notice that while the return type of the Python function is a simple list, in Spark we have to be more specific and declare the return type to be an array of strings.

We can define a new dataframe which is the result of appending this new array column:

```
df2 = df.withColumn("date_array", dateToArrayUDF(df["date"]))
```

We can see the result of this transformation below by using the `toPandas()` function to help with the formatting

```
(df2.select(["date", "date_array"])  
     .limit(2)  
     .toPandas())
```

	date	date_array
0	Mon Apr 06 22:19:45 PDT 2009	[Mon, Apr, 06, 22:19:45, PDT, 2009]
1	Mon Apr 06 22:19:49 PDT 2009	[Mon, Apr, 06, 22:19:49, PDT, 2009]

One other thing to note is that Spark Dataframes are based on Resilient Distributed Datasets (RDDs) which are immutable, distributed Java objects. It is preferred when using structured data to use dataframes over RDDs since the former has built-in optimizations and compression. The fact RDDs are immutable means that Dataframes are immutable. While we can still call the resulting dataframe from transformations with the same variable name `df`, the new dataframe is actually pointing to a completely new object under-the-hood. Many times it is desirable to call the resulting dataframes by the same name, but sometimes we have to give the new dataframe a different variable name like we did in the previous cell. We do this for convenience sometimes and other times because we do not want to violate the acyclic nature of DAGS.

Next let's define a more few functions to extract the day, month, year, time and create a timestamp for the tweet. The functions will take as an input the `date_array` column. That is it will take as input the array of strings that results from the delimiting of the `date` field by whitespaces. We don't show how these functions are defined (see [source code](#)), but rather import them from `ETL.src`:

```
from ETL.src.date_utility_functions import (get_month,
                                           get_year,
```

```
get_day,  
create_timestamp)
```

Now we create UDFs around these functions as well as creating them around lambda functions to change the `target` field from 0 to 1:

```
getYearUDF      = F.udf(get_year, IntegerType())  
getDayUDF       = F.udf(get_day, IntegerType())  
getMonthUDF     = F.udf(get_month, IntegerType())  
getTimeUDF      = F.udf(lambda a : a[3], StringType())  
timestampUDF    = F.udf(create_timestamp, TimestampType())  
targetUDF       = F.udf(lambda x: 1 if x == "4" else 0, IntegerType())
```

Now we apply the above UDFs just as we did before. We can get the `month` of the tweet from the `date_array` column by applying the `getMonthUDF` function with the following:

```
df2 = df2.withColumn("month", getMonthUDF(F.col("date_array")))
```

Note that we had to use the notation `F.col('input_col')` instead of

`df['input_col']` . This is because the column `date_array` is a derived column from the original dataframe/csv. In order for Spark to be able to act on derived columns we need to use the `F.col` to access the column instead of using the dataframe name itself.

Note: If I working on a cluster, the functions in `ETL.src.date_utility_functions` would not exist on the executors (or the datanodes) of the cluster and I would get an error. I would have to send these functions to the executors using the `addPyFile` if working from a notebook or adding them to the command when submitting the batch job.

Now we want to apply multiple different UDFs (`getYearUDF` , `getDayUDF` , `getTimeUDF`) to the same `date_array` column. We could list these operations all out individually as we did before, but since the input is not changing we can group all the UDFs as well as their output column names into a list,

```
list_udf = [getYearUDF, getDayUDF, getTimeUDF]
list_cols = ["year", "day", "time"]
```

and then iterate through that list applying the UDFS to the single input column,

```
for udf, output in zip(list_udf, list_cols) :
    df2 = df2.withColumn(output, udf(F.col("date_array")))
```

Now we want to store an actual datetime object for the tweet and use the `timeStampUDF` function to do so. Notice how easy it is to use UDFs that have multiple input columns, we just list them out!

```
# now we create a time stamp of the extracted data
df2 = df2.withColumn("timestamp", timeStampUDF(F.col("year"),
                                              F.col("month"),
                                              F.col("day"),
                                              F.col("time")))
```

Now we have finished getting the date-time information from the `date` column on our dataframe. We now rename some of the columns and prepare to transform the text data.

```
# convert the target to a numeric 0 if negative, 1 if positive
df2 = df2.withColumn("sentiment", targetUDF(df2["target"]))

# Drop the columns we no longer care about
df3 = df2.drop("flag", "date", "date_array", "time", "target")

# rename the tweet id as _id which is the unique identifier in MongoDB
df3 = df3.withColumnRenamed("id", "_id")

# rename the text as tweet so we can write a text index without confusion
df3 = df3.withColumnRenamed("text", "tweet")
```


We can take a look at our dataframe's entries by running,

```
df3.limit(2).toPandas()
```

	_id	user	tweet	day_of_week	month	year	day	timestamp	sentiment
0	1467810369	_TheSpecialOne_	@switchfoot http://twitpic.com/2y1zl - Awww, t...	Monday	4	2009	6	2009-04-06 22:19:45	0
1	1467810672	scotthamilton	is upset that he can't update his Facebook by ...	Monday	4	2009	6	2009-04-06 22:19:49	0

In order to clean the text data we first tokenize our strings. This means we create an array from the text where each entry in the array is an element in the string that was separated by white space. For example, the sentence,

```
"Hello my name is Mike"
```

becomes,

```
["Hello", "my", "name", "is", "Mike"]
```

The reason we need to tokenize is two part. The first reason is because we want to build up arrays of tokens to use in our **bag-of-words model**. The second reason is because it allows us to apply regular-expressions to individual words/tokens and gives us a finer granularity on cleaning our

text.

We use the `Tokenizer` class in Spark to create a new column of arrays of tokens:

```
from pyspark.ml.feature import Tokenizer

# use PySpark's built-in tokenizer to tokenize tweets
tokenizer = Tokenizer(inputCol = "tweet",
                      outputCol = "token")

df4 = tokenizer.transform(df3)
```

We can take a look at the results again:

```
df4.limit(2).toPandas()
```

	_id	user	tweet	day_of_week	month	year	day	timestamp	sentiment	token
0	1467810369	_TheSpecialOne_	@switchfoot http://twitpic.com/2y1zl - Awww, t...	Monday	4	2009	6	2009-04-06 22:19:45	0	[@switchfoot, http://twitpic.com/2y1zl, -, aww...
1	1467810672	scotthamilton	is upset that he can't update his Facebook by ...	Monday	4	2009	6	2009-04-06 22:19:49	0	[is, upset, that, he, can't, update, his, face...

Now we want to clean up the tweets. This means we want to remove any web addresses, call outs and hashtags. We do this by defining a Python function that takes in a list of tokens and performs regular expressions on each token to remove the unwanted characters and returns the list of clean

tokens:

```
import re

def removeRegex(tokens: list) -> list:
    """
    Removes hashtags, call outs and web addresses from tokens.
    """
    expr = '(@[A-Za-z0-9_]+)|(#[A-Za-z0-9_]+)|'+\
           '(https?://[^\s<>"]+|www\.[^\s<>"]+)'

    regex = re.compile(expr)

    cleaned = [t for t in tokens if not(regex.search(t)) if len(t) > 0]

    return list(filter(None, cleaned))
```

Now we write a UDF around this function:

```
removeWEBUDF = F.udf(removeRegex, ArrayType(StringType()))
```

Next we define our last function which removes any non-english characters from the tokens and wrap it in a Spark UDF just as we did above.

```
def normalize(tokens : list) -> list:
    """
    Removes non-english characters and returns lower case versions of words.
```

```

"""
subbed    = [re.sub("[^a-zA-Z]+", "", s).lower() for s in tokens]

filtered = filter(None, subbed)

return list(filtered)

normalizeUDF = F.udf(normalize, ArrayType(StringType()))

```



 Search

 Write

Sign up

Sign in



Now we apply these UDFs and remove any tweets that would result in an empty array of tokens after cleaning.

```

# remove hashtags, call outs and web addresses
df4 = df4.withColumn("tokens_re", removeWEBUDF(df4["token"]))

# remove non english characters
df4 = df4.withColumn("tokens_clean", normalizeUDF(df4["tokens_re"]))

# rename columns
df5 = df4.drop("token", "tokens_re")
df5 = df5.withColumnRenamed("tokens_clean", "tokens")

# remove tweets where the tokens array is empty, i.e. where it was just
# a hashtag, callout, numbers, web adress etc.
df6 = df5.where(F.size(F.col("tokens")) > 0)

```

Looking at the results:

```
df6.limit(2).toPandas()
```

	_id	user	tweet	day_of_week	month	year	day	timestamp	sentiment	tokens
0	1467810369	_TheSpecialOne_	@switchfoot http://twitpic.com/2y1zI - Awww, t...	Monday	4	2009	6	2009-04-06 22:19:45	0	[awww, thats, a, bummer, you, shoulda, got, da...
1	1467810672	scotthamilton	is upset that he can't update his Facebook by ...	Monday	4	2009	6	2009-04-06 22:19:49	0	[is, upset, that, he, cant, update, his, faceb...

LOAD

Now we come to the last stage in ETL, i.e. the stage where we write the data into our database. Spark and MongoDB work well together and writing the dataframe to a collection is as easy as declaring the format and passing in the names of the database and collection you want to write to:

```
db_name          = "db_twitter"
collection_name  = "tweets"

# write the dataframe to the specified database and collection
(df6.write.format("com.mongodb.spark.sql.DefaultSource")
    .option("database", db_name)
    .option("collection", collection_name)
    .mode("overwrite")
    .save())
```

That's it for the part on ETL with Spark. Spark is a great platform from doing batch ETL work on both structured and unstructured data. MongoDB is a document based NoSQL database that is fast, easy to use, allows for flexible schemas and perfect for working with text data. PySpark and MongoDB work well together allowing for fast, flexible ETL pipelines on large semi-structured data like those coming from tweets.

In the next part we'll take a look at working with our MongoDB database next!

[Pyspark](#)[Data Science](#)[Spark](#)[Big Data](#)[Data Engineering](#)



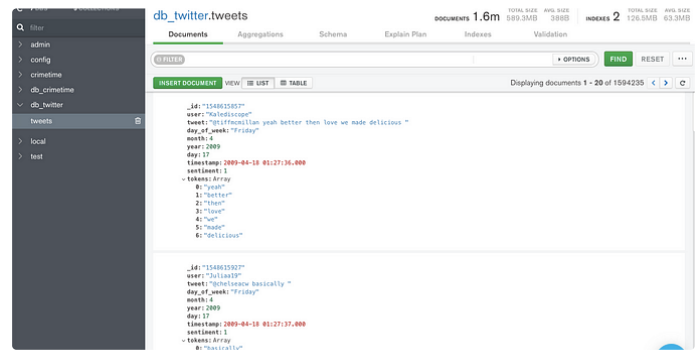
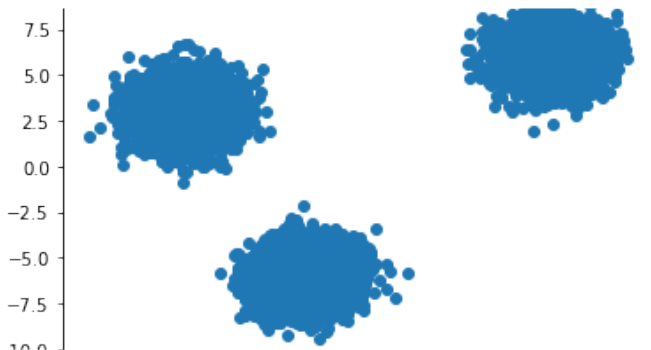
Written by Mike Harmon

11 Followers

Follow

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

More from Mike Harmon



Mike Harmon

Scikit-Learn Compatible KMeans Estimator From Scratch

15 min read · Dec 3, 2022



6

1



4



Mike Harmon

Sentiment Analysis Part 2

Exploring Text Data with MongoDB Using PyMongo

8 min read · Nov 12, 2022

code

code latitude longitude location on street name off street name cross s



 Mike Harmon

Sentiment Analysis Part 3

Machine Learning With Spark On Google Cloud

25 min read · Nov 29, 2022

 2  1 

See all from Mike Harmon

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

Polars & DuckDB

DataFrames and SQL For Python Without Pandas

17 min read · Jul 16, 2023

 23  

Recommended from Medium



Navdeep Sidana

Real Time Sentiment Analysis of Twitter using Spark Tweepy: Big...

<https://es.m.wikipedia.org/wiki/Archivo:Twitter-logo.svg>

7 min read · Feb 3, 2024



47



Eulene

Sentiment Analysis of Amazon Reviews Using Natural Language...

11 min read · Feb 8, 2024



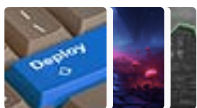
26



1



Lists



Predictive Modeling w/ Python

20 stories · 1267 saves



Practical Guides to Machine Learning

10 stories · 1526 saves



Coding & Development

11 stories · 644 saves



ChatGPT prompts

48 stories · 1653 saves

PySpark	Pandas
Large datasets in a distributed computing environment	Smaller, tabular datasets on a single machine
Resilient Distributed Datasets (RDD)	Dataframes
Parallel processing	Memory constraints



Integration with Hadoop and other big data tools	Integration possible with some tools but tedious
Spark SQL for table structure	-
Streaming data handled easily	Streaming difficult
Steep learning curve	Easy to learn

M

Mrunmayee Dhapre

Natural Language Processing (NLP) with Spark (Python)

Apache Spark is an open-source, distributed computing system that has emerged as a...

8 min read · Jan 30, 2024

 87

 1







Torsten Walbaum in Towards Data Science

What 10 Years at Uber, Meta and Startups Taught Me About Data...

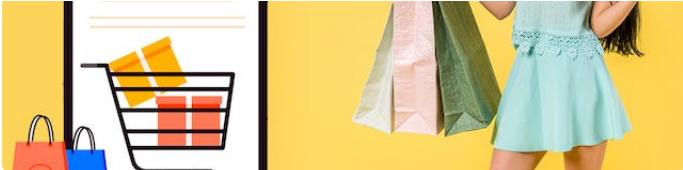
Advice for Data Scientists and Managers


9 min read · May 30, 2024

 4.4K

 76







Sithi Asma Basheer

Data Analysis Project:E-Commerce Reviews Analysis.

Finding insights from the customer reviews for a women’s clothing company.


4 min read · May 27, 2024

 42

 1







Joseph M. Tandiallo

Exploratory Data Analysis (EDA) Using Python

Basic Examples about exploratory data analysis and data visualization in Python

12 min read · Jan 21, 2024

 254

 1



[See more recommendations](#)

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Terms](#) [Text to speech](#) [Teams](#)