WOJSKOWA AKADEMIA TECHNICZNA
im. Jarosława Dąbrowskiego
WYDZIAŁ CYBERNETYKI

SPRAWOZDANIE

Z REALIZACJI PROJEKTU ZESPOŁOWEGO - BIG DATA

Grupa: **K9B2S4**

Skład zespołu:

Karol Baranowski
Wojciech Grzyb
Patryk Nędzi

Warszawa 2020

Spis treści

1. Wstęp

1.1. Opis zadania do realizacji

W ramach projektu zespołowego z przedmiotu Big Data należy w grupach maksymalnie 3-osobowych przygotować i opracować system przetwarzania i analizy danych należących do pewnego (wybranego przez poszczególne grupy) zagadnienia, zwanego dalej obszarem biznesowym. W określonym przez poszczególne grupy obszarze biznesowym należy zdefiniować cele do jakich ma być wykorzystywany projektowany system.
W celu realizacji zadania projektowego wymagane jest wykorzystanie technologii Big Data, technik przetwarzania języka naturalnego (ang. Natural Language Processing – NLP) oraz algorytmów uczenia maszynowego (ang. Machine Learning – ML).
Dane w zaprojektowanym systemie powinny być przetwarzane zarówno w sposób wsadowy jak i strumieniowy, z kolei sam proces przetwarzania danych powinien składać się z wielu cykli.
Wyniki przetworzenia danych należy zwizualizować w formie wykresów, grafów, bądź danych wyjściowych konsoli. Dodatkowo, w celu możliwości lepszego sterowania systemem, należy również składować informacje dotyczące przetwarzania danych (m.in. czasu przetworzenia danych) przez poszczególne komponenty systemu.

1.2. Wybór obszaru biznesowego

Jako obszar biznesowy zdecydowano się na: przetwarzanie wiadomości pochodzących z serwisów społecznościowych (takich jak Discord czy Twitter).

1.3. Cel projektowanego systemu

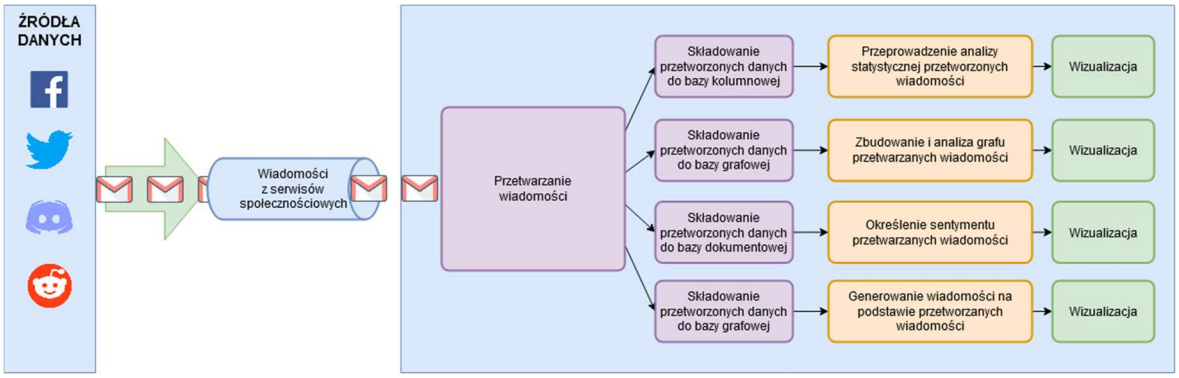
Przewidziano, że projektowany system będzie wykorzystywany do następujących celów:

- przeprowadzanie analiz statystycznych dla przetworzonych danych
- budowanie i analiza grafu przetworzonych wiadomości
- określanie sentymentu dla przetwarzanych wiadomości
- generowanie wiadomości na podstawie przetworzonych danych

2. Projekt systemu

2.1. Ogólna koncepcja systemu

Ogólna koncepcja projektowanego systemu została przedstawiona przy użyciu poniższego diagramu:



W projektowanym systemie wiadomości będą pobierane z wielu serwisów społecznościowych jednocześnie, a następnie zostaną one przetworzone w taki sposób, aby można było na ich podstawie zrealizować poszczególne cele projektu.

2.2. Dodatkowe założenia związane z realizacją projektu

W związku z wymaganiami zadania projektowego założono, że:

- ze względu na konieczność przetwarzania danych w sposób strumieniowy, pobrane z serwisów społecznościowych dane będą filtrowane „w locie”;
- ze względu na pobieranie wiadomości z wielu źródeł danych, pobrane dane będą kolejkowane;
- ze względu na spodziewany duży wolumen wiadomości do przetworzenia oraz konieczność skorzystania z technologii Big Data, przetworzone wiadomości zostaną odłożone do baz NoSQL;
- ze względu na konieczność skorzystania z technik NLP, generowanie wiadomości odbywać się będzie na podstawie łańcuchów Markowa;
- ze względu na konieczność skorzystania z algorytmów uczenia maszynowego, dla każdej pobranej wiadomości zostanie wykryty język;
- ze względu na konieczność skorzystania z algorytmów ML, określanie sentymentu wiadomości odbywać się będzie na podstawie wytrenowanego modelu;

- ze względu na konieczność przetwarzania danych w sposób wsadowy, model do określania sentymentu wiadomości zostanie wytrenowany przy użyciu przygotowanego wcześniej skończonego zbioru danych;

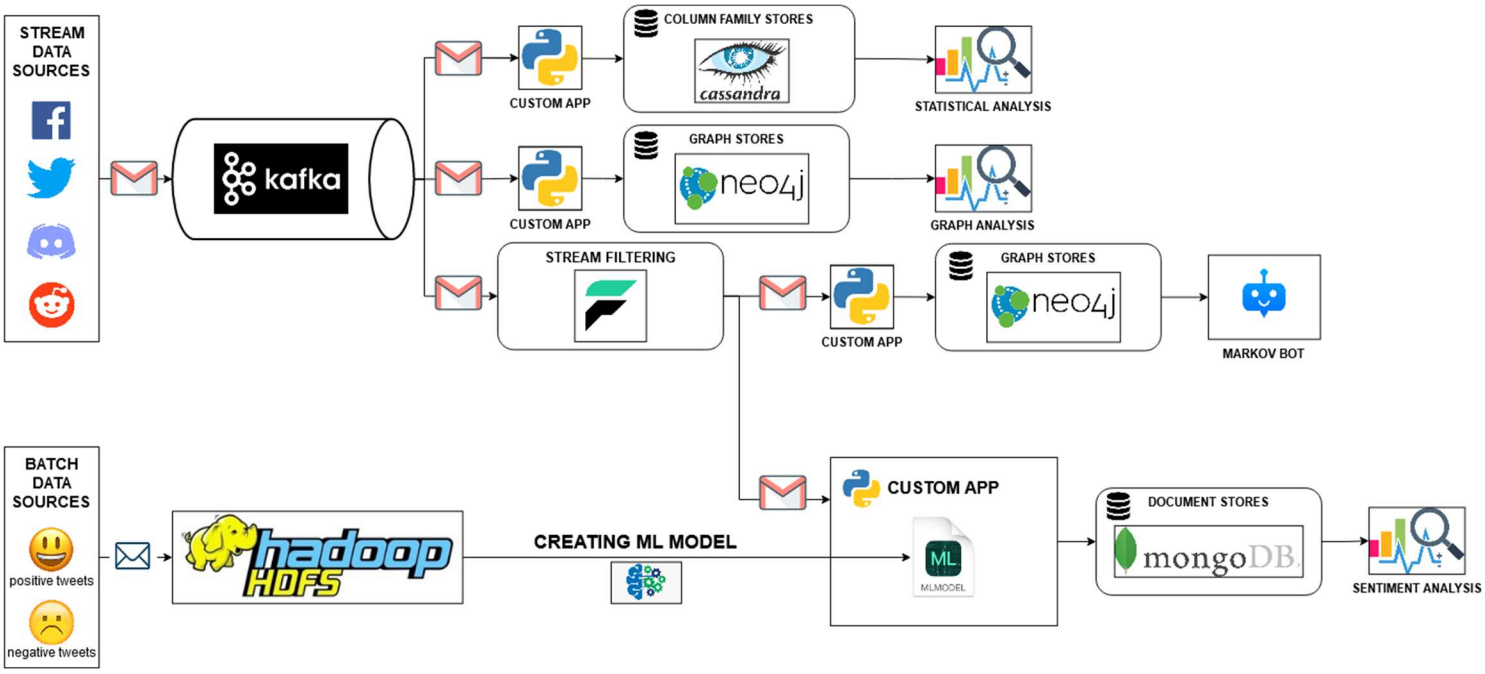
Dodatkowo założono również, że:

- na początku wiadomości będą pobierane tylko z Discord-a oraz Twitter-a, jednakże w trakcie działania systemu mogą pojawić się kolejne źródła danych, w związku z czym dodanie ich do systemu nie powinno wymagać przerwania obecnie działających procesów;
- filtrowanie wiadomości odbywać się będzie na podstawie kilku warunków takich jak:
  - serwis, z którego pochodzi wiadomość,
  - wykryty dla wiadomości język,
  - zawartość wulgaryzmów,
  - zawartość adresów URL.
- dane do wytrenowania modelu (określającego sentyment wiadomości) przechowywane będą w rozproszonym systemie plików;
- analiza statystyczna przetworzonych wiadomości będzie obejmować:
  - analizę liczby wiadomości z poszczególnych serwisów społecznościowych,
  - analizę najczęściej używanych hasztag-ów w poszczególnych serwisach społecznościowych,
  - analizę użytkowników wysyłających najwięcej wiadomości w poszczególnych serwisach społecznościowych.

2.3. Szczegółowy opis projektowanego systemu

System, przy wykorzystaniu dedykowanych programów, w pierwszym kroku będzie łączyć się z API serwisów społecznościowych w celu pobrania z nich wiadomości. W następnym etapie zostanie przeprowadzona wstępna obróbka pobranych wiadomości w celu sprowadzenia ich do jednej, wspólnej postaci. Następnie na przygotowanych w ten sposób wiadomościach przeprowadzone zostanie wykrywanie języka przy użyciu algorytmów uczenia maszynowego. W przypadku, gdy język wiadomości nie zostanie rozpoznany, jako język wiadomości zostanie ustawiona wartość ‘Not recognized’. W kolejnym kroku wiadomości (wraz z wykrytym językiem) zostaną przesłane do brokera kolejki pod pewien określony topic. Dane z tego topic-u (w niezmienionej formie) zostaną

odłożone do bazy kolumnowej oraz bazy grafowej, a także zostaną przesłane do przefiltrowania. Dane odłożone do bazy kolumnowej będą wykorzystywane do przeprowadzania analiz statystycznych, natomiast dane odłożone do bazy grafowej posłużą do zbudowania grafu przetwarzanych wiadomości. Filtrowanie wiadomości będzie odbywać się na podstawie zdefiniowanych w poprzednim rozdziale kryteriów. Przefiltrowane wiadomości będą wysyłane do brokera kolejki pod dwa topic-i: ‘valid-msgs’ oraz ‘invalid-msgs’. Do topic-u ‘valid-msgs’ trafią te wiadomości, które spełnią wszystkie zdefiniowane kryteria, pozostałe wiadomości trafią natomiast do topic-u ‘invalid-msgs’. Następnie wiadomości z topic-u ‘valid-msgs’ zostaną odłożone do bazy grafowej, a także zostaną wysłane do określenia ich sentymentu. Wiadomości odłożone do bazy grafowej posłużą do zbudowania łańcuchów Markowa, a następnie na ich podstawie do generowania wiadomości. Określanie sentymentu wiadomości odbywać się będzie przy wykorzystaniu wytrenowanego wcześniej modelu. Trenowanie modelu będzie odbywać się automatycznie w regularnych odstępach czasu. Dane wykorzystywane do trenowania modelu będą przechowywane w rozproszonym systemie plików. Wiadomości wraz z przyporządkowanym sentymentem będą zapisywane do bazy dokumentowej, skąd będą później pobierane w celu analizy liczby wiadomości z określonym sentymentem.



2.4. Wyróżnione procesy projektowanego systemu

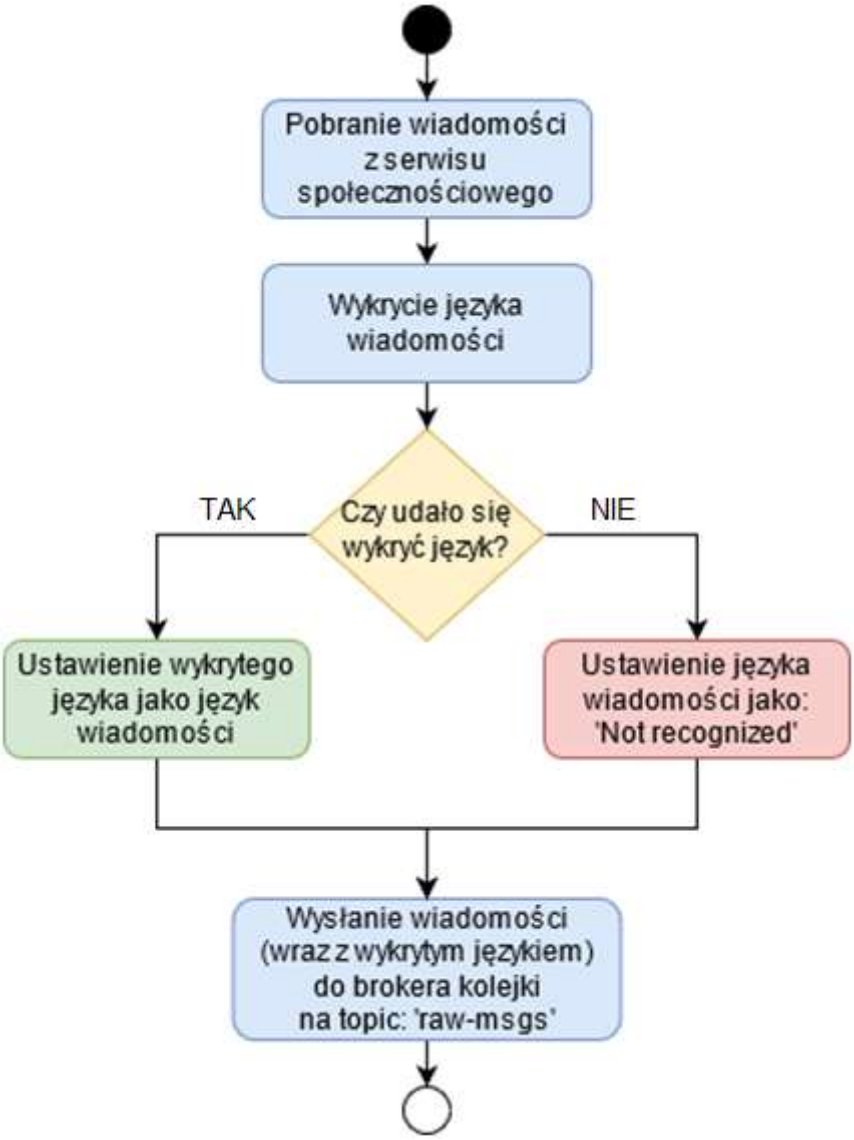
W projektowanym systemie wyróżniono następujące procesy:

- proces pobierania wiadomości z serwisów społecznościowych;
- proces zapisu pobranych wiadomości do bazy kolumnowej;
- proces przeprowadzenia analizy statystycznej dla przetworzonych wiadomości;
- proces zapisu pobranych wiadomości do bazy grafowej;
- proces analizy grafu przetworzonych wiadomości;
- proces filtrowania pobranych wiadomości;
- proces zapisu przefiltrowanych wiadomości do bazy grafowej;
- proces generowania wiadomości na podstawie danych z bazy grafowej;
- proces zapisu danych (do trenowania modelu) do rozproszonego systemu plików;
- proces trenowania modelu do określania sentymentu wiadomości na podstawie danych zapisanych do rozproszonego systemu plików;
- proces określania sentymentu dla przefiltrowanych wiadomości;
- proces analizy wiadomości z przydzielonym sentymentem;

Każdy z wymienionych procesów został szeroko omówiony i opisany w kolejnych podrozdziałach tego rozdziału.

2.4.1. Proces pobierania wiadomości z serwisów społecznościowych

Proces pobierania wiadomości z serwisów społecznościowych będzie polegał na: równoległym pobraniu przez dedykowane aplikacje wiadomości z serwisów społecznościowych, przydzieleniu pobranym wiadomościom języka, a następnie przesłaniu ich (wraz z wykrytym językiem) do brokera kolejki na topic ‘raw-msgs’.
Proces pobierania wiadomości z konkretnego serwisu społecznościowego przez dedykowaną dla niego aplikacje przedstawiono za pomocą poniższego diagramu:



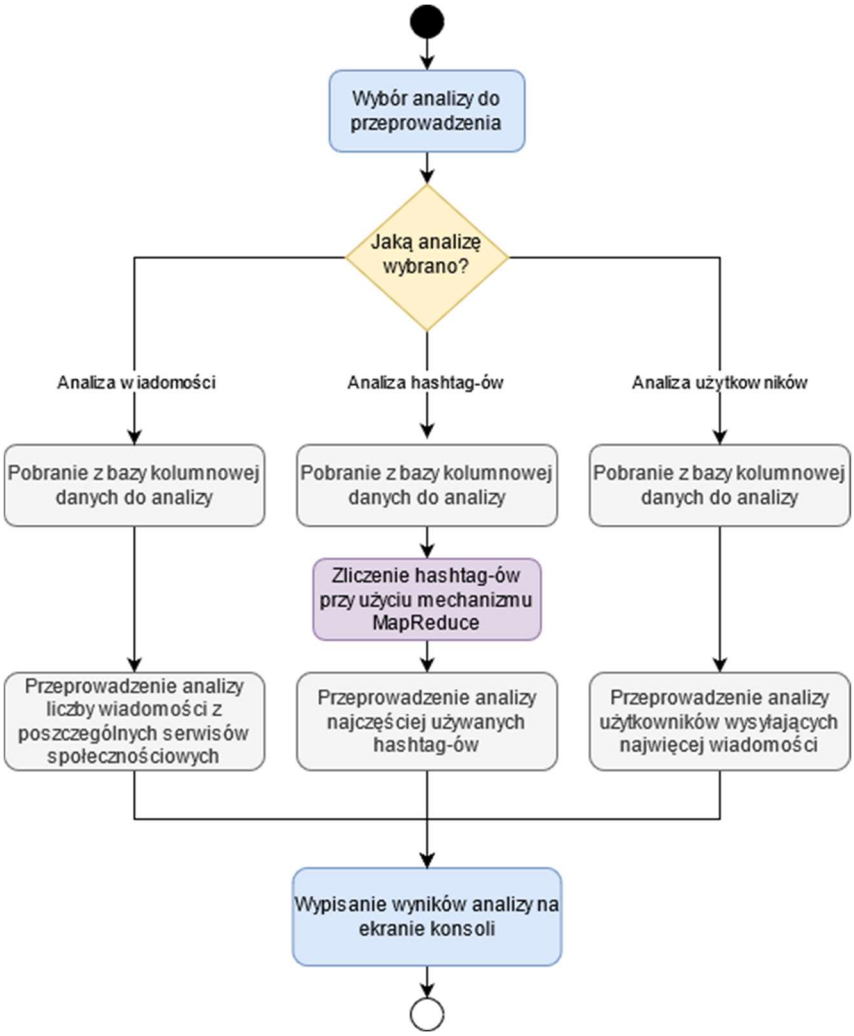
Do wykrywania języka wiadomości zaplanowano wykorzystać jeden z algorytmów uczenia maszynowego.

2.4.2. Proces zapisu pobranych wiadomości do bazy kolumnowej

Proces zapisu pobranych wiadomości do bazy kolumnowej będzie polegał na: pobraniu wiadomości od brokera kolejki z topic-u ‘raw-msgs’ i odłożeniu ich do bazy kolumnowej.

2.4.3. Proces przeprowadzenia analizy statystycznej

Proces przeprowadzenia analizy statystycznej będzie polegał na: pobraniu z bazy kolumnowej danych do analizy, a następnie przeprowadzeniu na pobranych danych zdefiniowanych wcześniej analiz statystycznych.



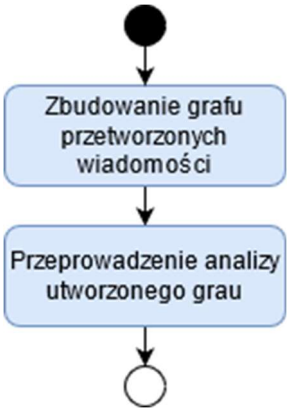
W celu przeprowadzenia analizy najczęściej używanych hasztag-ów zaplanowano skorzystać z mechanizmu MapReduce do zliczenia poszczególnych hasztag-ów.

2.4.4. Proces zapisu pobranych wiadomości do bazy grafowej

Proces zapisu pobranych wiadomości do bazy grafowej będzie polegał na: pobraniu wiadomości od brokera kolejki z topic-u ‘raw-msgs’ i odłożeniu ich do bazy grafowej.

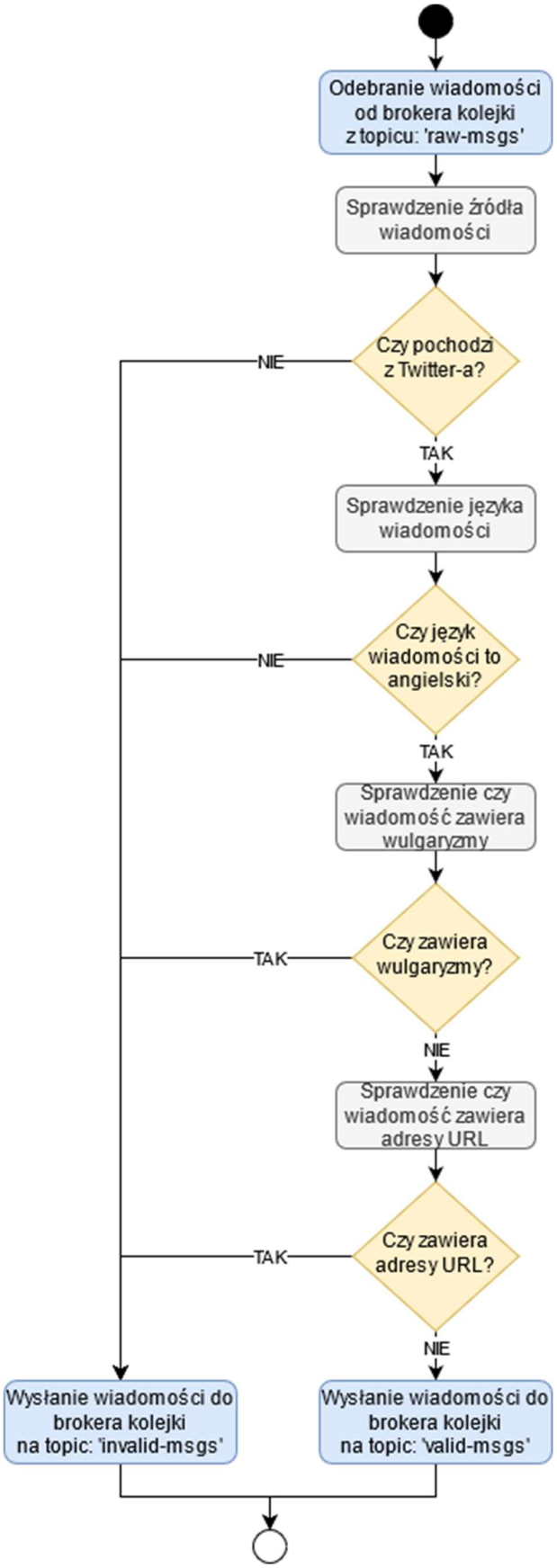
2.4.5. Proces analizy grafu przetworzonych wiadomości

Proces analizy grafu przetworzonych wiadomości będzie polegał na: zbudowaniu grafu wiadomości na podstawie danych umieszczonych do bazy grafowej, a następnie przeprowadzenia analizy utworzonego grafu poprzez wykorzystanie odpowiednich zapytań oraz miar odległości.



2.4.6. Proces filtrowania pobranych wiadomości

Proces filtrowania pobranych wiadomości będzie polegał na: pobraniu wiadomości od brokera kolejki z topic-u ‘raw-msgs’ i przefiltrowaniu ich na podstawie zdefiniowanych kryteriów. Jeśli wiadomość spełni wszystkie kryteria wówczas zostanie wysłana do brokera kolejki na topic ‘valid-msgs’, w przeciwnym wypadku zostanie wysłana do brokera kolejki na topic ‘invalid-msgs’.


Odebranie wiadomości od brokera kolejki z topicu: 'raw-msgs'
Sprawdzenie źródła wiadomości
Czy pochodzi z Twitter-a?
NIE
TAK
Sprawdzenie języka wiadomości
Czy język wiadomości to angielski?
NIE
TAK
Sprawdzenie czy wiadomość zawiera wulgaryzmy
Czy zawiera wulgaryzmy?
TAK
NIE
Sprawdzenie czy wiadomość zawiera adresy URL
Czy zawiera adresy URL?
TAK
NIE
Wysłanie wiadomości do brokera kolejki na topic: 'invalid-msgs'
Wysłanie wiadomości do brokera kolejki na topic: 'valid-msgs'

2.4.7. Proces zapisu przefiltrowanych wiadomości do bazy grafowej

Proces zapisu przefiltrowanych wiadomości do bazy grafowej będzie polegał na: pobraniu wiadomości od brokera kolejki z topic-u ‘valid-msgs’ i odłożeniu ich do bazy grafowej.

2.4.8. Proces generowania wiadomości

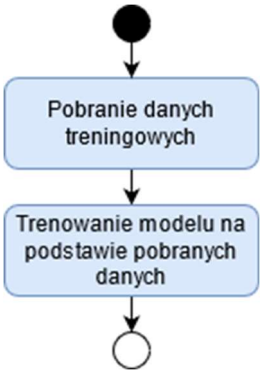
Proces generowania wiadomości na podstawie przefiltrowanych danych zapisanych do bazy grafowej będzie polegał na: utworzeniu łańcuchów Markowa, a następnie na ich podstawie wygenerowaniu wiadomości.

2.4.9. Proces zapisu danych do trenowania modelu

Proces zapisu danych do trenowania modelu będzie polegał na: przygotowaniu danych do trenowania modelu, a następnie zapisaniu ich do rozproszonego systemu plików.

2.4.10. Proces trenowania modelu do określania sentymentu wiadomości

Proces trenowania modelu do określania sentymentu wiadomości będzie polegał na: pobraniu danych treningowych z rozproszonego systemu plików, a następnie wykorzystaniu ich do wytrenowania modelu.

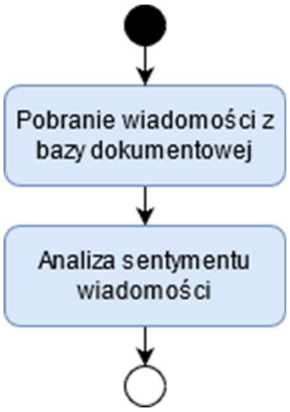


2.4.11. Proces określania sentymentu dla przefiltrowanych wiadomości

Proces określania sentymentu przefiltrowanych wiadomości będzie polegał na: pobraniu wiadomości od brokera kolejki z topic-u ‘valid-msgs’, a następnie określeniu im sentymentu na podstawie wytrenowanego wcześniej modelu. Wiadomości wraz z przydzielonym sentymentem zostaną zapisane do bazy dokumentowej.

2.4.12. Proces analizy wiadomości z przydzielonym sentymentem

Proces analizy wiadomości z przydzielonym sentymentem będzie polegał na: pobraniu danych z bazy dokumentowej, a następnie przeprowadzeniu analizy liczby wiadomości z konkretnym sentymentem.



2.5. Wybór technologii

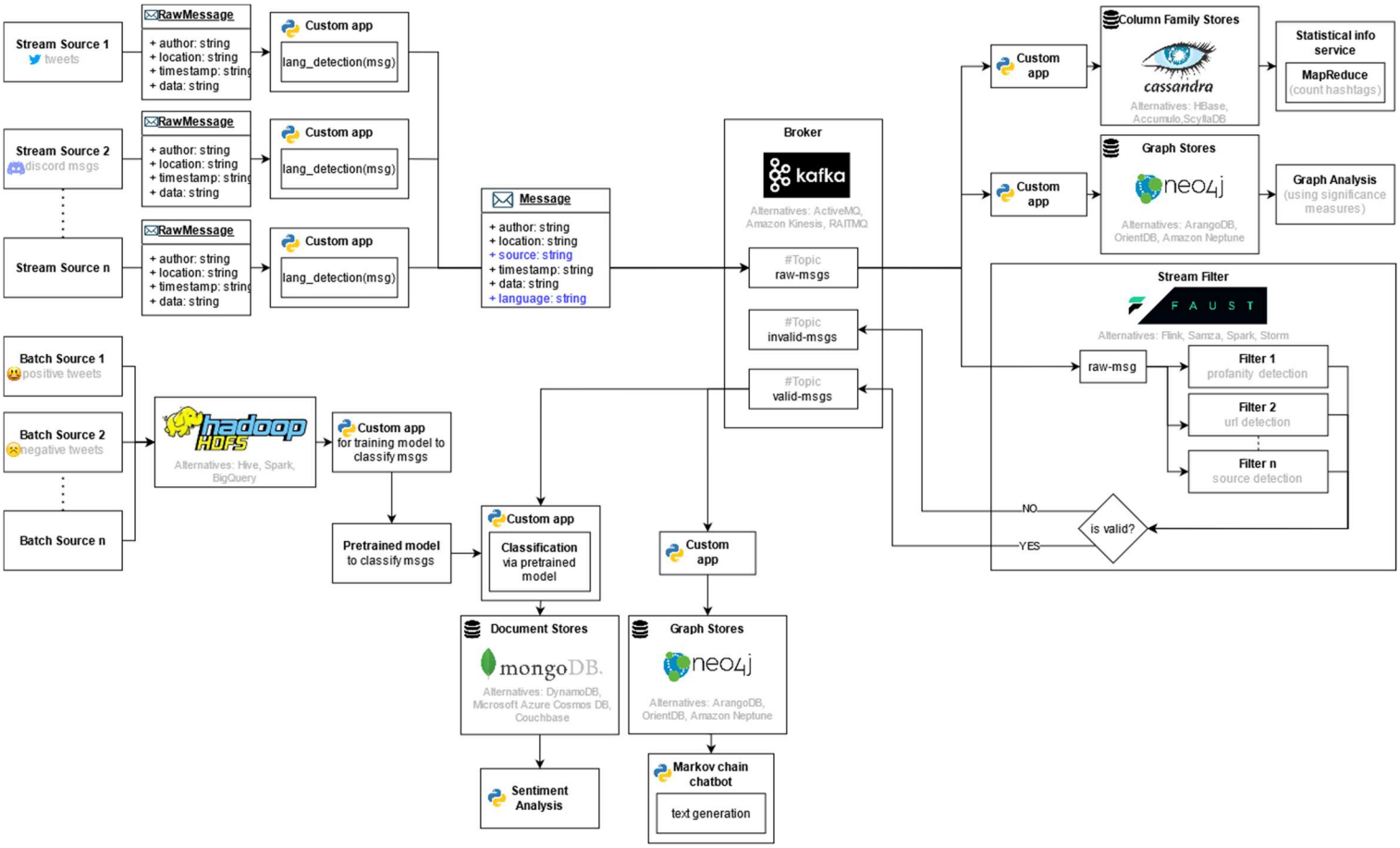
Po zdefiniowaniu procesów systemu, określone zostały technologie, które będą wykorzystywane w projektowanym systemie do realizacji poszczególnych procesów. Do wyboru technologii wzięto pod uwagę m.in. rankingu popularności danego rozwiązania oraz posiadane doświadczenie w pracy z daną technologią. Wybrane technologie przedstawiono w formie poniższej tabeli:

Kategoria	**Wybrane rozwiązanie**	**Alternatywne rozwiązania**
Język programowania	Python 3.9	C++, C#, Go, Java, Perl, Ruby
Broker kolejki	Kafka 5.4	ActiveMQ, Amazon Kinesis, RAITMQ
Baza kolumnowa	Cassandra 3.11	HBase, Accumulo, ScyllaDB
Baza grafowa	Neo4j 3.5	ArangoDB, OrientDB, Amazon Neptune
Baza dokumentowa	MongoDB 4.4	Amazon DynamoDB, Couchbase, Microsoft AzureCosmos DB
Rozproszony system plików	HDFS 2.9.1	IPFS, Hive, Spark, BigQuery
Narzędzie do przetwarzania strumieniowego	Faust 1.10.4	Flink, Samza, Spark, Storm

W tabeli wymienione zostały również alternatywne technologie, które były brane pod uwagę podczas wyboru konkretnego rozwiązania.

2.6. Kompletny cykl procesów

Całokształt procesów zachodzących w zaprojektowanym systemie przedstawiono w formie poniższego diagramu:

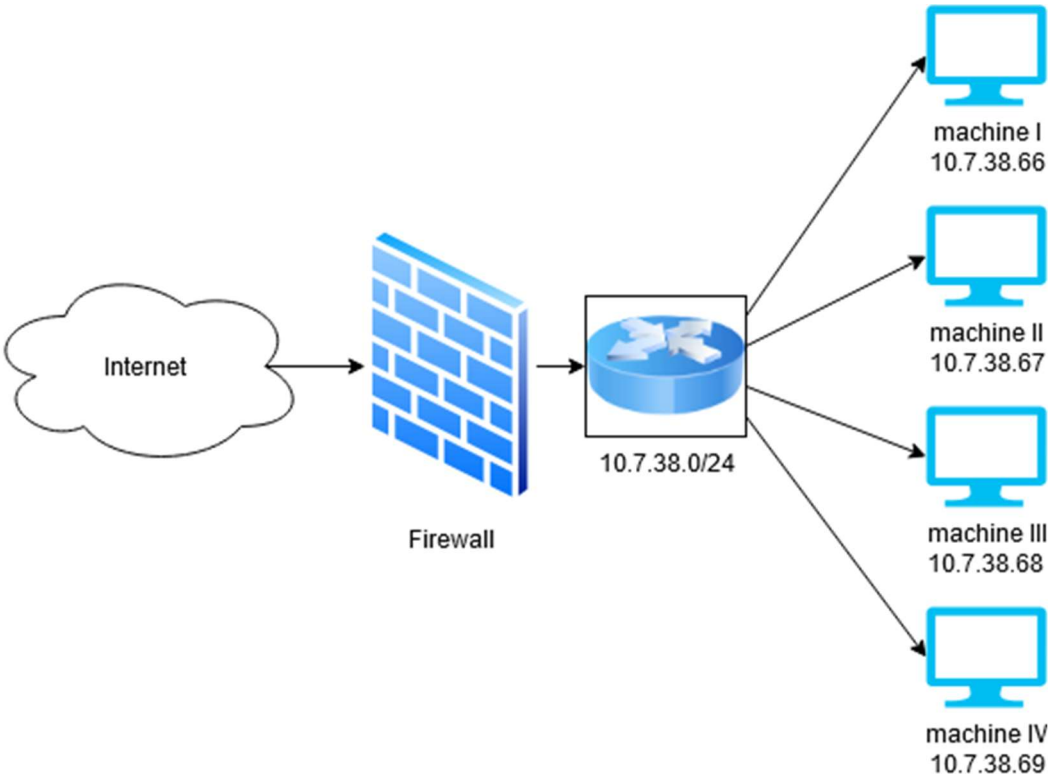


2.7. Architektura rozwiązania

Podczas tworzenia architektury rozwiązania wyróżniono 4 maszyny fizyczne o następującej adresacji sieciowej:

- maszyna I – 10.7.68.66
- maszyna II – 10.7.68.67
- maszyna III – 10.7.68.68
- maszyna IV – 10.7.68.69

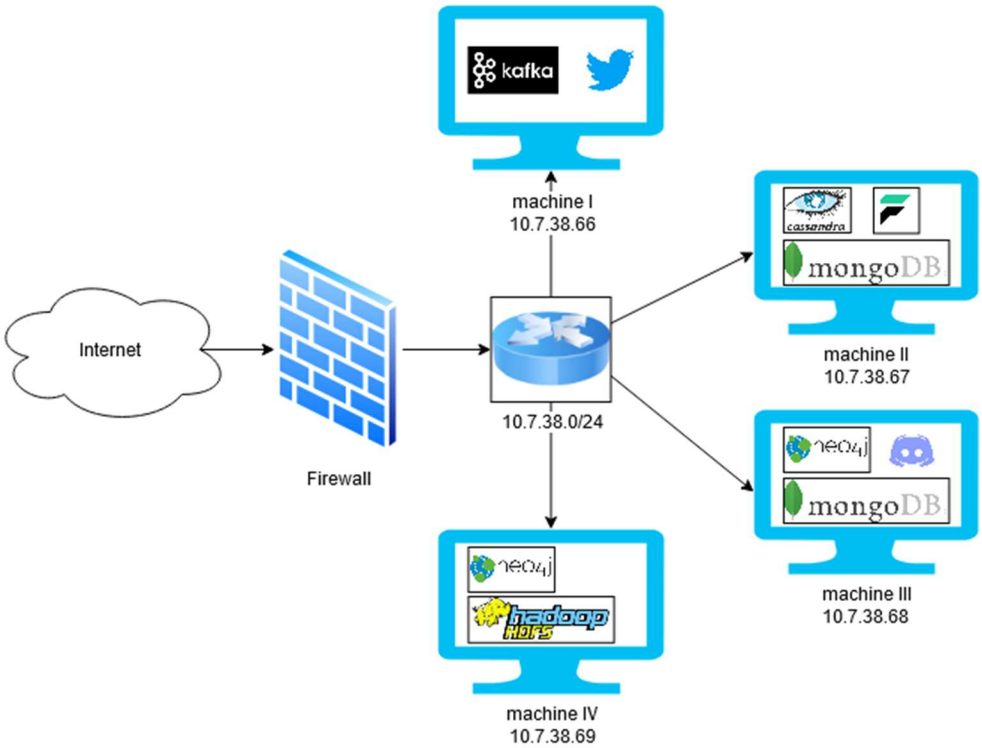
Koniecznym jest, aby każda z ww. maszyn „widziała się” nawzajem w sieci.

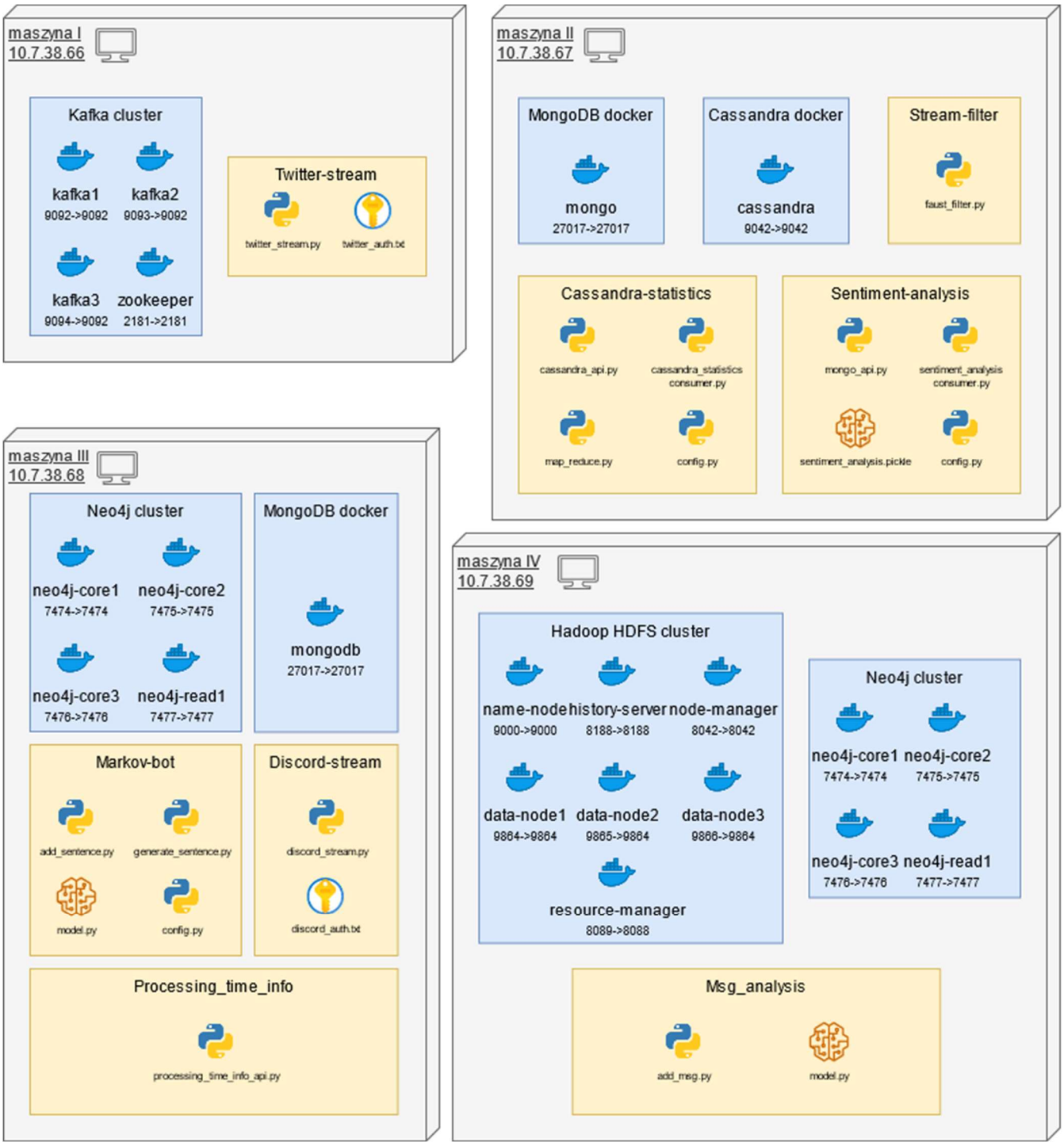


Na każdej z maszyn przewidziano:

- umieszczenie zaimplementowanych komponentów systemu;
- zainstalowanie i skonfigurowanie środowisk wykonawczych wyróżnionych w projekcie systemu;
-

Architekturę rozwiązania przedstawiono w formie poniższych diagramów:





Procedura konfiguracji i instalacji systemu została dołączono w formie następującego załącznika:

Konfiguracja.docx

3. Implementacja komponentów systemu

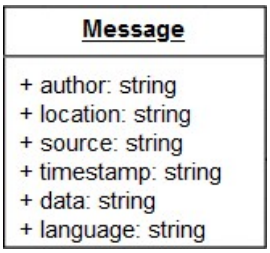
3.1. Pobieranie i początkowe przetwarzanie wiadomości (dane strumieniowe)

W niniejszym rozdziale opisano realizację pobierania i początkowej obróbki wiadomości z wielu źródeł danych.

Zgodnie z założeniami projektowymi wiadomości pochodzą z n źródeł, zaś w trakcie działania systemu mogą pojawić się nowe źródła wiadomości.

W celu realizacji tego założenia zdecydowano zaimplementować osobne serwisy dla każdego źródła danych. Każdy z serwisów pobiera wiadomości z danego źródła danych, przetwarza je do pewnej ustalonej postaci **Message**, a następnie wysyła przetworzone wiadomości do brokera kolejki. Dzięki takiemu podejściu, aby dodać nowe źródło danych, wystarczy tylko utworzyć dla niego serwis.

Obiekty przesyłane do brokera kolejki mają następującą strukturę:



W projekcie zaimplementowano dwa serwisy:

- jeden dla wiadomości pochodzących z Twittera (opisany w podrozdziale **3.1.1. Wiadomości z Twittera**),
- drugi dla wiadomości pochodzących z Discorda (opisany w podrozdziale **3.1.2. Wiadomości z Discorda**).

3.1.1. Wiadomości z Twittera

W celu pobierania i początkowego przetwarzania wiadomości z Twittera utworzono skrypt o nazwie: twitter_stream.py

W utworzonym skrypcie do pobierania wiadomości z Twittera wykorzystywana jest biblioteka ‘tweepy’. Biblioteka ta wymaga uprzednio otrzymania tokena autoryzującego od zespołu Twittera. W tym celu:

- napisano wniosek o otrzymanie tokena dla deweloperów do pełnego przetwarzania tweetów,
- wygenerowany token zapisano do pliku: ‘twitter_auth.txt’.

Procedura uwierzytelnienia w API Twittera zaimplementowana skrypcie wygląda następująco:

```
def twitter_auth():
    with open('twitter_auth.txt', 'r') as f:
        contents = [line.strip() for line in f if line.strip()]

    auth = tweepy.OAuthHandler(contents[0],
                               contents[1])
auth.set_access_token(contents[2],
contents[3])
api = tweepy.API(auth, wait_on_rate_limit=True)
try:
api.verify_credentials()
print("Authentication OK")
except:
print("Error during authentication")
return api
```

Do pobierania i wstępnego przetwarzania strumienia wiadomości zaimplementowano klasę StreamListener z metodą on_status():

```
class StreamListener(tweepy.StreamListener):
    def on_status(self, status):
        is_retweet= hasattr(status, "retweeted_status")
        if hasattr(status, "extended_tweet"):
            text = status.extended_tweet["full_text"]
        else:
            text= status.text
        text = (re.sub(r'<U+.*?>', '', text)).replace('\n', ' ').strip()
        lang = get_language(text)
        msg = {
            'author': status.user.screen_name,
            'location': status.user.location,
            'source': 'twitter',
            'timestamp': datetime.datetime.now(),
            'data': text,
            'language': lang
        }
        kafka_producer.send('raw-msgs', value=msg)
        kafka_producer.flush()
        stream = tweepy.Stream(auth=api.auth, listener=streamListener,
tweet_mode='extended')
        tags = ["football", "cryptocurrencies", "coronavirus"]
        stream.filter(track=tags)
```

Tworzony jest obiekt stream przyjmujący powyższą klasę oraz token jako parametry. W metodzie on_status() zapętlone jest pobieranie wiadomości z twittera, które mają tagi: "football", "cryptocurrencies", "coronavirus". Wiadomość msgma strukturę **Message** (opisaną w rozdziale **3.1. Pobieranie i początkowe przetwarzanie wiadomości**). Do wykrywania języka wiadomości wykorzystywana jest funkcja get_language(text), która

została opisana w rozdziale **3.2. Wykrywanie języka wiadomości**. Utworzony obiekt msg wysyłany jest do brokera kolejki (Kafki) przy użyciu metody send na topic ‘raw-msgs’. Broker kolejki został zdefiniowany w następujący sposób:

```
# KAFKA #
KAFKA_CLUSTER_IP = "10.7.38.66"
bootstrap_servers = [KAFKA_CLUSTER_IP + ":9092", KAFKA_CLUSTER_IP +
":9093", KAFKA_CLUSTER_IP + ":9094"]
kafka_producer=KafkaProducer(bootstrap_servers=bootstrap_servers,
value_serializer=lambda m: json.dumps(m,
default=datetime_converter).encode('ascii'))
```

3.1.2. Wiadomości z Discorda

W celu pobierania i początkowego przetwarzania wiadomości z Discorda utworzono skrypt o nazwie: discord_stream.py

W utworzonym skrypcie do pobierania wiadomości z Discorda wykorzystywana jest biblioteka ‘discord’. Biblioteka ta wymaga uprzednio skonfigurowania konta bota. W tym celu:

- stworzono w aplikacji Discord nowe konto bota o nazwie ‘big_data_project_bot’,
- utworzone konto dodano do serwera ‘Big Data Project’ (z którego będą pobierane wiadomości),
- dodanemu kontu nadano na serwerze odpowiednie uprawnienia (m.in. do odczytywania wiadomości wysyłanych przez użytkowników na kanale: ‘channel_stream’),
- wygenerowano token do uwierzytelnienia konta w implementowanej aplikacji,
- wygenerowany token zapisano do pliku: ‘discord_auth.txt’.

Procedura połączenia się z kontem bota ‘big_data_project_bot’ w zaimplementowany skrypcie wygląda następująco:

```
# DISCORD #
client = discord.Client()
with open('discord_auth.txt', 'r') as f:
    token = f.readline().strip()
client.run(token)
```

Do pobierania i wstępnego przetwarzania wiadomości przez bota zaimplementowano funkcję on_message(message).

```
@client.event
async def on_message(message):
if message.author == client.user:
return

    if message.channel.name == 'channel_stream':
        text = message.content
        lang = get_language(text)
msg = {
'author': message.author.name,
'location': None,
```

```
'source': 'discord',
'timestamp': datetime.datetime.now(),
'data': text,
'language': lang
        }
kafka_producer.send('raw-msgs', value=msg)
kafka_producer.flush()
        print(msg)
```

Po połączeniu się z kontem bota, bot nasłuchuje czy na serwerach (na których został dodany) nie pojawiła się nowa wiadomość. Jeśli na którymś z serwerów bota pojawi się nowa wiadomość, to bot na początku sprawdzi, kto jest jej autorem, oraz na jakim kanale wiadomość została wysłana. Jeśli wiadomość została wysłana przez użytkownika na kanale ‘channel_stream’, to wówczas bot przetworzy taką wiadomość. W przeciwnym wypadku wiadomość zostanie zignorowana.

W sytuacji, gdy wiadomość jest przetwarzana, bot tworzy nowy obiekt msg o strukturze **Message** (opisanej w rozdziale **3.1. Pobieranie i początkowe przetwarzanie wiadomości**). Do wykrywania języka wiadomości wykorzystywana jest funkcja get_language(text), która została opisana w rozdziale **3.2. Wykrywanie języka wiadomości**. Utworzony obiekt msg wysyłany jest do brokera kolejki (Kafki) przy użyciu metody send na topic ‘raw-msgs’.

Broker kolejki został zdefiniowany w następujący sposób:

```
# KAFKA #
KAFKA_CLUSTER_IP = "10.7.38.66"
bootstrap_servers = [KAFKA_CLUSTER_IP + ":9092", KAFKA_CLUSTER_IP +
":9093", KAFKA_CLUSTER_IP + ":9094"]
kafka_producer=KafkaProducer(bootstrap_servers=bootstrap_servers,
value_serializer=lambda m:json.dumps(m,
default=datetime_converter).encode('ascii'))
```

3.2. Wykrywanie języka wiadomości (ML)

Wykrywanie języka wiadomości wykorzystywane jest przez serwisy źródeł danych strumieniowych. Każdy z tych serwisów odwołuje się do funkcji get_language(msg) w celu wykrycia języka dla przetwarzanej wiadomości.

Funkcja get_language(msg) przyjmuje jeden parametr wejściowy (tekst wiadomości), natomiast zwraca stringa reprezentującego wykryty język.

```
from langid.langidimport LanguageIdentifier, model

identifier = LanguageIdentifier.from_modelstring(model, norm_probs=True)

def get_language(msg):
    predictions = identifier.classify(msg)
# Językrozpoznany z prawdopodobieństwemwiększymniż 80%
if predictions[1] >0.8:
        lang = predictions[0]
```

```
else:
        lang = "Not recognized"
return lang
```

Funkcja get_language(msg) działa w oparciu o wytrenowany wcześniej model pochodzący z biblioteki langid. Przy użyciu tego modelu dokonywana jest klasyfikacja wiadomości. Każdej (zdefiniowanej w modelu) klasie języka przyporządkowywana jest pewna wartość liczbowa odzwierciedlająca stopień prawdopodobieństwa tego, iż przetwarzana wiadomość pochodzi z danego języka. Im większa jest przypisana liczba, tym większe prawdopodobieństwo tego, że dana wiadomość pochodzi z danego języka. Przewidywanym dla wiadomości językiem jest ten, dla którego przyporządkowana liczba jest największa.

Zaimplementowana funkcja get_language(msg) przed zwróceniem przewidzianego języka sprawdza wartość prawdopodobieństwa, z jakim zostało stwierdzone, że dana wiadomość pochodzi z danego języka. Jeśli wartość prawdopodobieństwa nie jest większa od 80%, to wówczas, jako zwracany język, zwracana jest wartość „Not recognized”. Przewidziany dla wiadomości przy użyciu wytrenowanego modelu język zwracany jest tylko wtedy, gdy wartość prawdopodobieństwa jest większa niż 0.8.

3.3. Filtrowanie wiadomości

Proces filtrowania wiadomości dotyczy wszystkich wiadomości wysłanych do brokera kolejki (w tym przypadku Kafki)na topic ‘raw-msgs’. Filtrowanie wiadomości odbywa się w sposób ciągły - analizowane są wszystkie wiadomości z topicu ‘raw-msgs’ wiadomość po wiadomości. Każda wiadomość sprawdzana jest pod wieloma kryteriami. Zgodnie z założeniami projektu, w trakcie działania systemu mogą pojawić się nowe kryteria, a ich wdrożenie powinno być możliwie najprostsze.

Jeśli chociaż jedno z określonych kryteriów nie zostanie spełnione przez wiadomość, to wówczas wiadomość ta zostanie odfiltrowana, uznana za nieprawidłową i wysłana do brokera kolejki na topic ‘invalid-msgs’.Z kolei na topic ‘valid-msgs’ wysłane zostaną tylko te wiadomości, które spełnią wszystkie zdefiniowane kryteria jednocześnie.

Do filtrowania wiadomości skorzystano z narzędzia Faust. Narzędzie to wymaga zdefiniowania czynności (które będą wykonywane przez agentów Fausta) przy użyciu pythonowego skryptu. W związku z powyższym utworzono skrypt o nazwie: ‘faust_filter.py’, w którym zdefiniowano zachowanie agentów Fausta.

W utworzonym skrypcie skorzystano z biblioteki: faust. Określono, aby Faust subskrybował topic ‘raw-msgs’ z brokera kolejki Kafka.

```
import faust
app = faust.App(
'faust_filter',
broker=[f'kafka://{KAFKA_CLUSTER_IP}:9092',
f'kafka://{KAFKA_CLUSTER_IP}:9093',
f'kafka://{KAFKA_CLUSTER_IP}:9094'],
value_serializer="json"
)
raw_msgs_topic = app.topic('raw-msgs')
```

W skrypcie ‘faust_filter.py’ zdefiniowano zachowanie dla jednego agenta Fausta:

```
@app.agent(raw_msgs_topic)
async def filter_msg(msgs):
async for msginmsgs:
filter_values = await asyncio.gather(
            *[filter_language(msg), filter_profanity(msg),
filter_source(msg), filter_url(msg)])
is_valid = all(filter_values)
        topic = 'valid-msgs'ifis_validelse'invalid-msgs'
kafka_producer.send(topic, value=msg)
kafka_producer.flush()
print(f'The message has been sent to the topic: "{topic}"')
```

Zdefiniowany agent Fausta sprawdza, czy wszystkie kryteria są spełnione dla pewnej wiadomości msg. Jeśli wszystkie kryteria są spełnione, to wówczas wiadomość ta zostaje wysłana do brokera kolejki na topic ‘valid-msgs’, w przeciwnym wypadku wiadomość zostaje odfiltrowana i wysłana do brokera kolejki na topic ‘invalid-msgs’.

Obecnie zdefiniowane są 4 kryteria do filtrowania wiadomości:

- sprawdzanie czy wiadomość jest napisana w języku angielskim:

```
async def filter_language(msg):
return msg.get('language') == 'en'
```

- sprawdzanie czy wiadomość pochodzi z Twitter-a:

```
async def filter_source(msg):
return msg.get('source') == 'twitter'
```

- sprawdzanie czy wiadomość nie zawiera wulgaryzmów:

```
from profanityfilterimportProfanityFilter

async def filter_profanity(msg):
return ProfanityFilter().is_clean(msg.get('data'))
```

- sprawdzanie czy wiadomość nie zawiera adresów URL:

```
from urlextractimportURLExtract

async def filter_url(msg):
return not URLExtract().has_urls(msg.get('data'))
```

Do zdefiniowania kryterium weryfikującego, czy wiadomość nie zawiera wulgaryzmów, użyto biblioteki ProfanityFilter, która działa w oparciu o własną listę słów cenzurowanych. Jeśli w badanej wiadomości nie znaleziono słów z listy cenzurowanych słów, to wówczas rozpatrywane kryterium zostaje spełnione (zakładane jest, iż badana wiadomość nie zawiera wulgaryzmów).

Do zdefiniowania kryterium weryfikującego, czy wiadomość nie zawiera adresów URL, użyto biblioteki URLExtract, która w badanej wiadomości próbuje znaleźć jakiekolwiek wystąpienie domeny najwyższego poziomu (ang. TLD – Top-Level Domain). Jeśli w badanej wiadomości nie zostanie znalezione TLD, to wówczas rozpatrywane kryterium zostaje spełnione (zakładane jest, iż badana wiadomość nie zawiera adresów URL).

W celu uruchomienia narzędzia Faust należy przejść do folderu ze skryptem ‘faust_filter.py’ i wykonać następujące polecenie:

```
faust -A faust_filter worker -l info
```

3.4. Analiza grafowa

Do przeprowadzenia analizy grafowej wiadomości skorzystano z narzędzia: Neo4j Browser dostępnego pod adresem: http://10.7.38.69:7474/browser/.

W celu dostrzeżenia ukrytych zależności między węzłami grafu przeprowadzono analizę przy wykorzystaniu zapytania Cypher:

```
MATCH (n)
RETURN
DISTINCT labels(n),
count(*) AS SampleSize,
avg(size(keys(n))) as AvgPropCount,
min(size(keys(n))) as MinPropCount,
max(size(keys(n))) as MaxPropCount,
avg(size( (n)-[]-() ) ) as AvgRelCount,
min(size( (n)-[]-() ) ) as MinRelCount,
max(size( (n)-[]-() ) ) as MaxRelCount
```

Zapytanie zwróciło poniższą tabelę:

"labels(n)"	"SampleSize"	"PropCount"	"AvgRelCount"	"MinRelCount"	"MaxRelCount"
["User"]	178673	2.0	2.2964465811846195	2	531
["Location"]	50666	1.0	3.5284016894959396	1	59947
["Message"]	134806	2.0	1.7176015904336364	1	3541

Widać trzy rodzaje węzłów: User, Location, Message oraz ich liczebności w bazie. Każdy użytkownik oraz wiadomość mają dwa atrybuty, a lokacja jeden. W chwili pisania raportu średnio każdy użytkownik ma nieco ponad dwa połączenia z innymi węzłami, lokacja

~3,5, a wiadomość ~1,7 połączeń. Widać również, jaka jest najmniejsza i największa liczba relacji każdego z rodzaju węzła.

Użyto również biblioteki GDS do wyznaczenia centralności stopnia użytkownika oraz lokacji. Po analizie okazało się, że miary centralności są liczone tylko dla grafów, które wszystkie węzły mają takiego samego typu w tej bibliotece. W związku z tym, że w bazie Neo4j przechowywane są trzy typy węzłów, zaimplementowane własną kwerendę do wyznaczenia centralności stopnia użytkownika:

```
MATCH (u:User) RETURN u.nick, size((u)-[:AUTHOR]->()) as score ORDER BY
score DESC limit 20
```

Kwerenda ta zwraca 20 najważniejszych użytkowników w kontekście liczby napisanych wiadomości:

"u.nick"	"score"
"CoronaUpdateBot"	530
"ebenesport"	247
"Bot_Corona_V"	236
"Football2ch"	140
"bitcoinconnect"	138
"vmrwanda"	119
"jeremy_hume"	116
"Daminous_Purity"	114
"batallas_jgr"	102
"LivEchoLFC"	93

Zauważyć można, że w rankingu tym wygrywają boty, gdyż one generują najwięcej wiadomości.

Analogicznie napisano kwerendę do znalezienia najważniejszych lokalizacji, czyli takich, z których najwięcej użytkowników pisze wiadomości:

```
MATCH (l:Location) RETURN l.name, size((l)<-[:LIVES]-()) as score ORDER
BY score DESC limit 20
```

"l.name"	"score"
"Unknown"	69560
"United States"	2315
"London, England"	1271
"USA"	942
"London"	939
"United Kingdom"	847
"Washington, DC"	736
"California, USA"	708
"Los Angeles, CA"	683
"Florida, USA"	636

Zdecydowana większość użytkowników nie podaje lokalizacji, a w wiadomościach z tagami “coronavirus”, “football”, “cryptocurrencies” przodują Stany Zjednoczone oraz Londyn.

Do przeprowadzenia analizy grafowej łańcuchów wiadomości skorzystano z narzędzia: Neo4j Browser dostępnego pod adresem: http://10.7.38.68:7474/browser/.

Wyliczono ważność węzła poprzez centralność stopnia z użyciem gds (graph data science):

```
CALL gds.alpha.degree.stream({
nodeProjection: 'Word',
relationshipProjection: 'NEXT'
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC limit 20
```

"name"	"score"
"RT"	30166.0
"the"	12792.0
":"	11079.0
"."	10599.0
","	8875.0
"and"	7989.0
"a"	7758.0
"of"	6540.0
"to"	6219.0
"in"	4288.0

Widać, że najważniejszym wyrazem jest RT czyli znacznik, że wiadomość była retweet'em, czyli udostępnieniem wypowiedzi kogoś innego. Następnie pojawiają się typowe angielskie słowa a, the, and itp. Oraz znaki interpunkcyjne.

Wyliczono centralność również algorytmem ArticleRank, zwracającym węzły, które najczęściej występują po jak największej ilości innych węzłów. Wykorzystano kwerendę:

```
CALL gds.alpha.articleRank.stream({
nodeProjection: 'Word',
relationshipProjection: 'NEXT',
dampingFactor: 0.85
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS msg, score
ORDER BY score DESC
```

Jej wynik przedstawia tabela:

"msg"	"score"
":"	625.9774756729895
"."	271.6334697869024
"…"	227.86027348500212
","	219.3094687535078
"and"	164.6719175867853
"in"	118.08184140207301
"to"	112.91325430075815
"is"	110.16814705284489
"!"	102.40909149476502
"for"	94.0818927766988
"of"	84.76136723406671
"the"	80.82422762624923
"on"	77.21792979139138
"?"	76.99849075801323
"I"	73.4525110158982
"football"	67.80565967877337

Przodują znaki interpunkcyjne, ponieważ mogą wystąpić po bardzo dużej liczbie wyrazów, większej niż jakiekolwiek słowo. Następnie są typowe angielskie słowa, które pojawiają się niemal w każdym zdaniu. Między innymi dzięki tym zależnościom bot będzie używał ich często w generacji zdań - mają dużo połączeń co zwiększa prawdopodobieństwo wylosowania tego słowa.

3.5. Pobieranie danych statystycznych

3.5.1. Zapisywanie do bazy danych

W pierwszym kroku ustanawiane jest połączenie z bazą danych Cassandra, co przedstawiono poniżej.

```
from cassandra.clusterimportCluster

cass_cluster = Cluster([CASSANDRA_IP])
cass_session = cass_cluster.connect('statistics')
```

Następnie tworzony i uruchamiany jest wątek, który pobiera wiadomości z wybranego tematu Kafki, w tym przypadku raw-msgs.

```
if__name__ == "__main__":
# Utworzenie wątków - każdy wątek to osobny consumer

thread1 = Thread('raw-msgs', 'cassandra-statistics', KAFKA_BOOTSTRAP_SERVERS)
# thread2 = Thread('invalid-msgs', 'faust-filter', KAFKA_BOOTSTRAP_SERVERS)
    # Rozpoczęcie wątków
thread1.start()
# thread2.start()
print("Wątek został uruchomiony")
```

Konstruktor wątku tworzy połączenie z Kafką przy wykorzystaniu obiektu klasy KafkaConsumer:

```
class Thread(threading.Thread):
def __init__(self, topic, group_id, brokers):
        threading.Thread.__init__(self)
# Utworzenie consumer-a do odczytywaniadanych z topic-u registration-logs
self.consumer = KafkaConsumer(topic, group_id=group_id,
bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
value_deserializer=lambda m: json.loads(m.decode('ascii')))
```

Główna funkcja wątku, run, iteruje po wiadomościach przesyłanych za pośrednictwem Kafki i dla każdej z nich tworzy słownik message, stanowiący reprezentacje obiektu Message opisanego we wcześniejszych rozdziałach.

```
def run(self):
for msginself.consumer:

message = {
'author': msg.value.get('author'),
'location': msg.value.get('location'),
'source': msg.value.get('source'),
'timestamp': msg.value.get('timestamp'),
'data': msg.value.get('data'),
'language': msg.value.get('language'),
}
```

Następnie każda wiadomość jest umieszczana w bazie przy wykorzystaniu polecenia INSERT.

```
stmt = cass_session.prepare(
"INSERT INTO messages (author, location, source, time, msg_data,
language, msg_day) VALUES (?,?,?,?,?,?,?)")
qry = stmt.bind(
    [message['author'], message['location'], message['source'],
datetime.strptime(message['timestamp'], '%Y-%m-%d %H:%M:%S.%f'),
message['data'], message['language'],
message['timestamp'].split()[0].replace('-', '')])
cass_session.execute(qry)
```

3.5.2. Pobieranie danych statystycznych

W celu pobierania statystyk utworzono skrypt cassandra_api.py. Do kontaktu z bazą danych wykorzystuje się bibliotekę Cassandra i klasę Cluster. Procedurę łączenia się z bazą przedstawiono poniżej:

```
from cassandra.clusterimportCluster
from config import CASSANDRA_IP

cass_cluster = Cluster([CASSANDRA_IP])
cass_session = cass_cluster.connect('statistics')
```

Narzędzie przyjmuje argumenty z wiersza poleceń w następujący sposób:

```
[root@rpd-10 cassandra_statistics]# python3 cassandra_api.py --help
usage: cassandra_api.py [-h] [-sd STARTDATE] [-ed ENDDATE] [-t TOP]
                        [-s SOURCE]
                        action

positional arguments:
  action                available: service, user, hashtag

optional arguments:
  -h, --help            show this help message and exit
  -sd STARTDATE, --startdate STARTDATE
                        start date - format %Y%m%d (if left empty, data from
                        today is returned
  -ed ENDDATE, --enddate ENDDATE
                        end date - format %Y%m%d (usable alongside start date
                        only)
  -t TOP, --top TOP     top X users or top X hashtags
  -s SOURCE, --source SOURCE
                        available: twitter, discord (default: twitter)
```

Istnieje możliwość określenia dat początkowej (-sd) i końcowej (-ed). Uruchomienie narzędzia bez zakresu dat skutkuje zliczeniem tweetów z aktualnego dnia. Ponadto można wskazać źródło oraz liczbę pobieranych wyników, jako odpowiednio source (-s) i top (-t). Do przetwarzania argumentów służy biblioteka argparse.

```
parser = argparse.ArgumentParser()
parser.add_argument("action", help="available: service, user, hashtag")
parser.add_argument("-sd", "--startdate", help="start date - format
%%Y%%m%%d (if left empty, data from today is returned")
parser.add_argument("-ed", "--enddate", help="end date - format %%Y%%m%%d
(usable alongside start date only)")
parser.add_argument("-t", "--top", type=int, help="top X users or top X
hashtags")
parser.add_argument("-s", "--source", help="available: twitter, discord
(default: twitter)")

args = parser.parse_args()
```

Przykładowy kod, który wskazuje, jacy użytkownicy w określonym zakresie dat wysłali najwięcej wiadomości przedstawiono poniżej. Jeżeli nie wskazano liczby, zwraca się 5 pierwszych nazw użytkowników (linijka sorted_users[:5]). Na podstawie wskazanych dat tworzona jest lista dni (msg_day_list), która przekazywana jest w kwerendzie.

```
if args.action == "user":
if args.source:
        source = args.source
else:
        source = 'twitter'
if args.startdate:
if args.enddate:
start_date = datetime.strptime(args.startdate, "%Y%m%d")
end_date = datetime.strptime(args.enddate, "%Y%m%d")
numdays = (end_date - start_date).days
msg_day_list = tuple([(end_date - timedelta(days=x)).strftime("%Y%m%d")
for x in range(numdays + 1)])

stmt = cass_session.prepare(
"SELECT author, count(*) FROM messages WHERE msg_day in ? AND source = ?
GROUP BY msg_day, source, author;")
qry = stmt.bind([msg_day_list, source])
twitter_rows = cass_session.execute(qry)
sorted_users = sorted([tuple(row) for row in list(twitter_rows)],
key=lambda x: x[1], reverse=True)
if args.top:
print(sorted_users[:args.top])
else:
print(sorted_users[:5])
```

3.5.3. Wykorzystanie mechanizmu MapReduce

Mechanizm MapReduce wykorzystywany jest w projekcie do zliczania hashtagów w pewnym okrojonym zbiorze wiadomości. Użytkownik definiuje zbiór wiadomości poprzez określenie parametrów: -sd (START DATE) oraz -ed (END DATE).

Do zaimplementowania mechanizmu MapReduce skorzystano z implementacji opartej na bibliotece multiprocessing udostępnionej w serwisie pymotw.com[1]:

```
import collections
import itertools
import multiprocessing


class MapReduce(object):

def __init__(self, map_func, reduce_func, num_workers=None):
"""
map_func

          Function to map inputs to intermediate data. Takes as
          argument one input value and returns a tuple with the key
```

[1] Link do publikacji: https://pymotw.com/2/multiprocessing/mapreduce.html

```
          and a value to be reduced.

reduce_func

          Function to reduce partitioned version of intermediate data
          to final output. Takes as argument a key as produced by
map_func and a sequence of the values associated with that
          key.

num_workers

          The number of workers to create in the pool. Defaults to the
          number of CPUs available on the current host.
        """
self.map_func = map_func
self.reduce_func = reduce_func
self.pool = multiprocessing.Pool(num_workers)

def partition(self, mapped_values):
"""Organize the mapped values by their key.
        Returns an unsorted sequence of tuples with a key and a sequence
of values.
        """
partitioned_data = collections.defaultdict(list)
for key, value in mapped_values:
partitioned_data[key].append(value)
return partitioned_data.items()

def __call__(self, inputs, chunksize=1):
"""Process the inputs through the map and reduce functions given.

        inputs
          An iterable containing the input data to be processed.

chunksize=1
          The portion of the input data to hand to each worker.  This
          can be used to tune performance during the mapping phase.
        """
map_responses = self.pool.map(self.map_func, inputs, chunksize=chunksize)
partitioned_data = self.partition(itertools.chain(*map_responses))
reduced_values = self.pool.map(self.reduce_func, partitioned_data)
return reduced_values
```

Wykorzystana implementacja wymaga zdefiniowania funkcji mapowania oraz funkcji redukcji:

- funkcja mapowania ‘hashtag_mapper(msg_text)’ wyodrębnia użyte w wiadomości hashtagi:

```
def hashtag_mapper(msg_text):
hashtag_list = []
for part in msg_text.split():
if part.startswith('#'):
hashtag_list.append((part[1:], 1))
return hashtag_list
```

- funkcja redukcji zdefiniowana jako ‘hashtag_reducer(item)’ redukuje liczbę wyodrębnionych hashtagów poprzez zsumowanie identycznych tagów:

```
def hashtag_reducer(item):
    word, occurrences = item
return word, sum(occurrences)
```

Mechanizm MapReduce wywoływany jest na zbiorze wiadomości pochodzących z pewnego, określonego przez użytkownika zakresu. Z każdej wiadomości pochodzącej z tego zbioru wyodrębniane są użyte w wiadomości hashtagi. Następnie wyodrębnione hashtagi są redukowane poprzez sumowanie identycznych tagów. W ten sposób uzyskiwana jest lista hashtagów wraz z ich licznością.

```
start_date = datetime.strptime(args.startdate, "%Y%m%d")
numdays = (today - start_date).days
msg_day_list = tuple([(today - timedelta(days=x)).strftime("%Y%m%d") for
x in range(numdays + 1)])

stmt = cass_session.prepare(
"SELECT msg_data FROM messages WHERE msg_day in ? AND source = ?;")
qry = stmt.bind([msg_day_list, source])
twitter_rows = cass_session.execute(qry)
message_texts = [row[0] for row in twitter_rows]

# Wywołaniemechanizmu MapReduce
mapper = MapReduce(hashtag_mapper, hashtag_reducer, 3)
hashtag_counts = mapper(message_texts)
```

Na standardowe wyjście wypisywanych jest n najczęściej użytych hashtagów (w okrojonym przez użytkownika zbiorze wiadomości), gdzie n to liczba określona przez użytkownika przy pomocy parametru -t (TOP). Jeśli użytkownik nie określi tego parametru, to wówczas na standardowe wyjście zostanie wypisanych 5 najpopularniejszych w danym okresie hashtagów.

```
hashtag_counts = mapper(message_texts)
hashtag_counts.sort(key=operator.itemgetter(1))
hashtag_counts.reverse()
if args.top:
print(hashtag_counts[:args.top])
else:
print(hashtag_counts[:5])
```

3.6. Analiza sentymentu

Do analizy sentymentu zaimplementowano model uczenia maszynowego Naive Bayes. Jako danych treningowych użyto dwóch plików zawierających tweety określone jako pozytywne i negatywne. Są to dane wsadowe umieszczone na HDFS.

3.6.1. Przetwarzanie wsadowe zestawów danych

Pliki do przetworzenia mają strukturę json, gdzie każda linia zawiera wszystkie możliwe parametry tweeta dostępne z API Twittera. Do analizy sentymentu istotne jest pole “text” zawierajace treść wiadomości”. Przykładowa struktura jednej linii pliku z pominięciem większości pól prezentuje się następująco:

{"contributors":..., "text": "Treść wiadomości", "user": {"time_zone": ... }

W celu przeniesienia zestawów danych do HDFS należy w pierwszej kolejności umieścić je wewnątrz dockera namenode:

```
docker cp positive_tweets.jsonnamenode:/docker cp
negative_tweets.jsonnamenode:/
```

Następnie wejść do dockera namenode:

```
Dockerexec -itnamenodebash
```

Umieścić pliki na HDFS poleceniami:

```
hdfsdfs -put positive_tweets.json input/twitter_samples
hdfsdfs -put negative_tweets.json input/twitter_samples
```

Jeżeli należałoby zwiększyć pliki o dodatkowe rekordy należy wykonać polecenie:

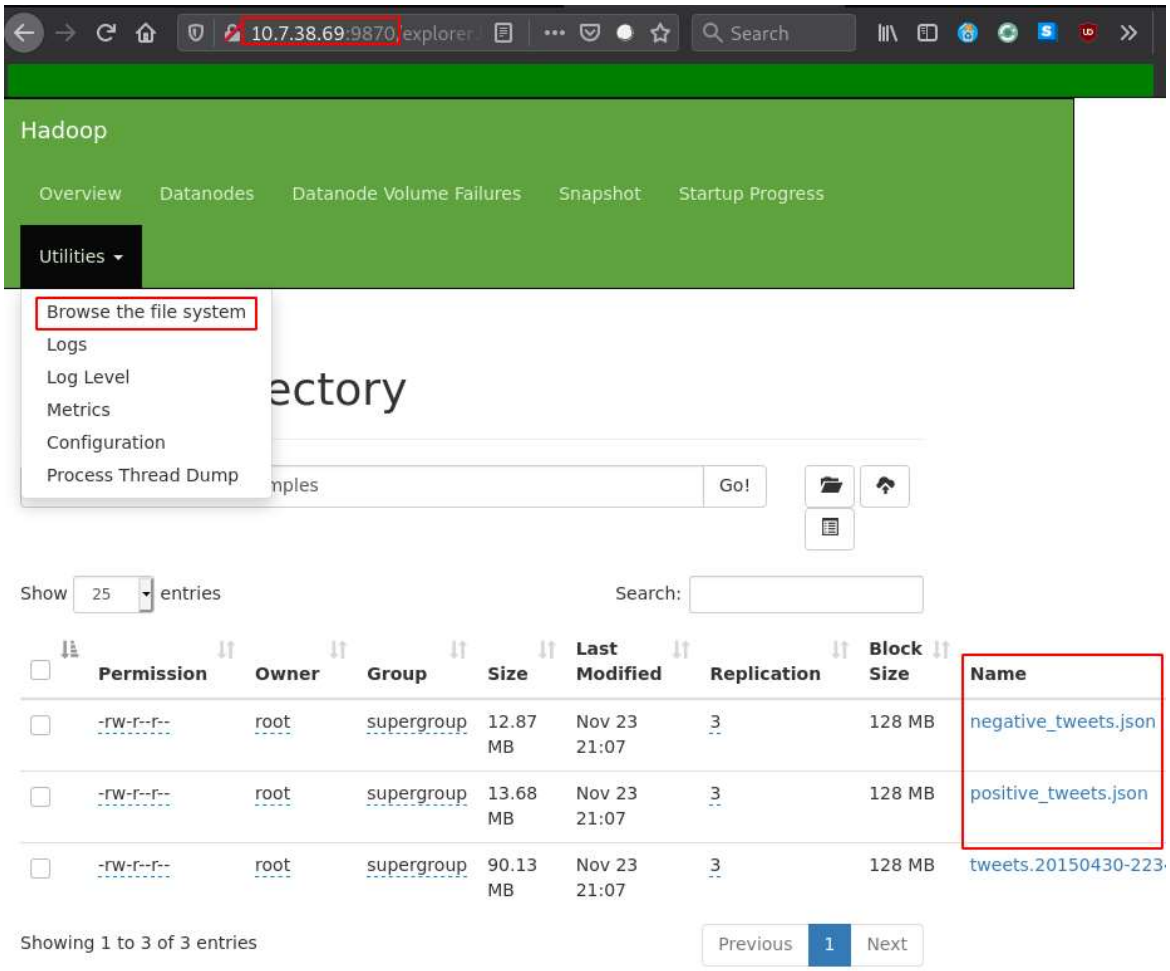
```
hadoopdfs -appendToFile<localsrc><dst>
```

Które dodaje do pliku <dst> na HDFS plik <localsrc>.

Tym sposobem NameNode umieścił pliki na wszystkich trzech DataNode’ach czyniąc je dostępnymi do dalszego przetwarzania. W aplikacjach na innych maszynach, aby czytać pliki z HDFS wykonywany jest poniższy kod:

```
import hdfs
client = hdfs.InsecureClient('http://10.7.38.69:9870', user='root')
with client.read('input/twitter_samples/positive_tweets.json',
encoding='utf-8') as reader:
# Właściwe przetwarzanie pliku
```

Przetwarzanie tych plików opisane jest dokładnie w następnym podrozdziale. Zarządzać HDFS można również poprzez WebHDFS wpisując adres maszyny i port, na których wdrożony jest docker z Hadoop, a następnie wejść w Utilities ->Browse the file system:



3.6.2. Tworzenie modelu Naive Bayes

W pierwszym kroku ustanawia się połączenie z HDFS, co widać na poniższym fragmencie kodu.

```
import hdfs

client = hdfs.InsecureClient('http://10.7.38.69:9870', user='root')
```

Podczas tworzenia modelu wykorzystuje się bibliotekę NLTK. Na początku głównej funkcji pobierane są potrzebne dane.

```
def create_model():
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
```

Następnie pobiera się zestawy pozytywnych oraz negatywnych tweetów z HDFS:

```
with client.read('input/twitter_samples/positive_tweets.json',
encoding='utf-8') as reader:
positive_tweets_hdfs = [json.loads(line) for line in reader]
```

```
with client.read('input/twitter_samples/negative_tweets.json',
encoding='utf-8') as reader:
negative_tweets_hdfs = [json.loads(line) for line in reader]
```

W następnym kroku tweety są dzielone na odpowiednie części mowy i wykorzystuje się pobraną wcześniej stoplistę.

```
positive_tweets_tokens = [tweet["text"].split() for tweet in
positive_tweets_hdfs]
negative_tweets_tokens = [tweet["text"].split() for tweet in
negative_tweets_hdfs]


stop_words = stopwords.words('english')

positive_tweets_tokens_cleaned =
[remove_stop_words(lemmatization(clean_data(token)), stop_words) for
token in positive_tweets_tokens]

negative_tweets_tokens_cleaned =

[remove_stop_words(lemmatization(clean_data(token)), stop_words) for
token in negative_tweets_tokens]
```

Następnie tworzony jest zestaw danych, na który składają się pomieszane pozytywne i negatywne tweety.

```
dataset = positive_tweets_tokens_transformed +
negative_tweets_tokens_transformed
random.shuffle(dataset)

train_data = dataset[:7000]
```

Klasyfikator jest trenowany na podstawie ww. danych.

```
classifier = NaiveBayesClassifier.train(train_data)
```

Stan klasyfikatora jest zapisywany do pliku z rozszerzeniem .pickle:

```
f = open(MODEL_PICKLE_FILENAME, 'wb')
pickle.dump(classifier, f)
f.close()
```

3.6.3. Przydzielanie sentymentu

W pierwszym kroku ustanawiane jest połączenie z bazą danych MongoDB, co przedstawiono poniżej.

```
# MONGO
mongo_client = MongoClient(MONGO_IP)
mongo_db = mongo_client["sentiment"]
mongo_col = mongo_db["sentiment"]
```

Następnie tworzony i uruchamiany jest wątek, który pobiera wiadomości z wybranego tematu.

```
if__name__ == "__main__":
# Utworzenie wątków - każdy wątek to osobny consumer

thread1 = Thread('valid-msgs', 'faust-filter', KAFKA_BOOTSTRAP_SERVERS)
# Rozpoczęcie wątków
thread1.start()
print("Wątek został uruchomiony")
```

Konstruktor wątku jest widoczny poniżej. Na początku wywołuje funkcję, która pobiera lub tworzy klasyfikator oraz tworzy połączenie z Kafką.

```
class Thread(threading.Thread):
def __init__(self, topic, group_id, brokers):
        threading.Thread.__init__(self)

self.classifier = get_model()
self.tokenizer = TweetTokenizer()

# Utworzenie consumer-a do odczytywaniadanych z topic-u registration-logs
self.consumer = KafkaConsumer(topic, group_id=group_id,
bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
value_deserializer=lambda m: json.loads(m.decode('ascii')))
```

Funkcja pobierania modelu początkowo próbuje znaleźć plik z rozszerzeniem .pickle. Jeżeli takowy nie istnieje, następuje utworzenie modelu. Kod wygląda następująco:

```
def get_model():
if not os.path.isfile(MODEL_PICKLE_FILENAME):
create_model()
try:
        f = open(MODEL_PICKLE_FILENAME, 'rb')
        classifier = pickle.load(f)
f.close()
except Exception as e:
print(f"Wystąpiłbłądpodczasładowaniamodelu: {e}")
return None
return classifier
```

Funkcja przypisująca sentyment znajduje się w kodzie wątku - klasy Thread. Program podczas działania iteruje po wiadomościach znajdujących się w konsumencie oraz zapisuje je w postaci słowników. Data wysłania wiadomości jest konwertowana na format rozpoznawany przez MongoDB - IsoDate.

```
for msginself.consumer:

message = {
'author': msg.value.get('author'),
'location': msg.value.get('location'),
'source': msg.value.get('source'),
'timestamp':
datetime.strptime(datetime.strptime(msg.value.get('timestamp'), '%Y-%m-%d
%H:%M:%S.%f').strftime("%Y-%m-%dT%H:%M:%S.000Z"), "%Y-%m-
%dT%H:%M:%S.000Z"),
```

```
'data': msg.value.get('data'),
'language': msg.value.get('language'),
}
```

Treść każdej wiadomości jest dzielona na wyrazy oraz umieszczana w modelu:

```
custom_tokens = self.tokenizer.tokenize(message['data'])
prob_dict = self.classifier.prob_classify(dict([token, True] for token in
custom_tokens))
```

Jeżeli model z prawdopodobieństwem większym niż 85% stwierdzi pozytywny lub negatywny sentyment, wiadomość ma przypisana odpowiednią kategorię. Jeżeli prawdopodobieństwo nie znajduje się we wskazanym przedziale, wiadomość nie jest zapisywana do bazy.

```
if prob_dict.prob('Negative') <0.15:
    category = 'Positive'
elifprob_dict.prob('Negative') >0.85:
    category = 'Negative'
else:
    category = "Inconclusive"
```

```
message["sentiment"] = category

if category != "Inconclusive":
mongo_col.insert_one(message)
```

Regularnie w pewnych odstępach czasowych następuje ponowne trenowanie modelu. Do tego celu wykorzystuje się bibliotekę schedule.

```
schedule.every(1).minutes.do(update_model)
```

Wywoływana funkcja update_model wymusza ponowne utworzenie i wytrenowanie modelu na podstawie tweetów umieszczonych w HDFS. Proces tworzenia modelu opisany jest w poprzednim podrozdziale.

```
def update_model():
print(f"Aktualizuję model")
create_model()
try:
        f = open(MODEL_PICKLE_FILENAME, 'rb')
classifier = pickle.load(f)
f.close()
exceptExceptionase:
print(f"Wystąpił błąd podczas aktualizowania modelu: {e}")
return None
return classifier
```

3.6.4. Połączenie z bazą danych

W celu pobierania liczby tweetów, dla których określono sentyment utworzono skrypt mongo_api.py. Do kontaktu z bazą danych wykorzystuje się bibliotekę PyMongo. Procedurę łączenia się z bazą przedstawiono poniżej:

```
from pymongo importMongoClient

# MONGO
mongo_client = MongoClient(MONGO_IP)
mongo_db = mongo_client["sentiment"]
mongo_col = mongo_db["sentiment"]
```

Narzędzie przyjmuje argumenty z wiersza poleceń w następujący sposób:

```
[root@rpd-10 sentiment_analysis]# python3 mongo_api.py --help
usage: mongo_api.py [-h] [-sd STARTDATE] [-ed ENDDATE] action

positional arguments:
  action                available: count

optional arguments:
  -h, --help            show this help message and exit
  -sd STARTDATE, --startdate STARTDATE
                        start date - format %Y%m%d (if left empty, data from
                        today is returned
  -ed ENDDATE, --enddate ENDDATE
                        end date - format %Y%m%d (usable alongside start date
                        only)
```

Istnieje możliwość określenia dat początkowej i końcowej. Uruchomienie narzędzia tylko z argumentem count skutkuje zliczeniem tweetów z aktualnego dnia. Do przetwarzania argumentów służy biblioteka argparse.

```
parser = argparse.ArgumentParser()

parser.add_argument("action", help="available: count")

parser.add_argument("-sd", "--startdate", help="start date - format
%%Y%%m%%d (if left empty, data from today is returned")

parser.add_argument("-ed", "--enddate", help="end date - format %%Y%%m%%d
(usable alongside start date only)")

args = parser.parse_args()
```

Przykładowy kod dla wywołania z określonym zakresem dat przedstawiono poniżej.

```
if args.action == "count":
if args.startdate:
if args.enddate:

start_date = datetime.strptime(datetime.strptime(args.startdate, "%Y%m%d").strftime("%Y-%m-
%dT%H:%M:%S.000Z"), "%Y-%m-%dT%H:%M:%S.000Z")

end_date = datetime.strptime((datetime.strptime(args.enddate, "%Y%m%d") +
timedelta(days=1)).date().strftime("%Y-%m-%dT%H:%M:%S.000Z"), "%Y-%m-%dT%H:%M:%S.000Z")
```

```
            positive = mongo_col.find({"sentiment": "Positive", "timestamp": {"$gte":
start_date, "$lte": end_date}}).count()

            negative = mongo_col.find({"sentiment": "Negative", "timestamp": {"$gte":
start_date, "$lte": end_date}}).count()

print(f"Positive tweets: {positive}\nNegative tweets: {negative}")
```

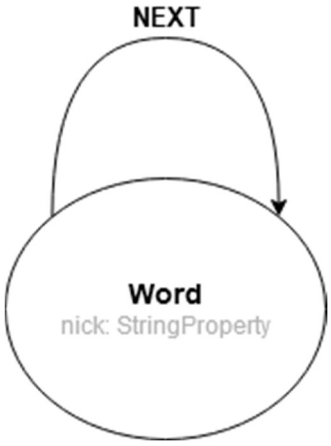
Do pobierania tweetów z bazy służy funkcja find, która w argumencie otrzymuje wskazany sentyment oraz przedział czasu określony za pomocą operatorów logicznych $gte (greater than or equal) oraz $lte (less than or equal). Działanie na datach w bazie MongoDB wymaga wykorzystania określonego formatu IsoDate, który widać w kodzie powyżej.

3.7. Generowanie wiadomości przy wykorzystaniu łańcuchów Markowa

Jednym z celów projektu jest generowanie wiadomości. Do tego celu postanowiono wykorzystać łańcuchy Markowa. Założono, iż wiadomości będą generowane na podstawie przefiltrowanych wiadomości wysłanych do brokera kolejki na topic ‘valid-msgs’ (opis filtrowania wiadomości opisano w rozdziale **3.3. Filtrowanie wiadomości**). Innymi słowy wiadomości będę generowane tylko na podstawie anglojęzycznych wiadomości z Twittera, w których nie ma wulgaryzmów ani adresów URL.

3.7.1. Model danych

Model, w oparciu o który będą generowane tweety, wygląda następująco:



Zdefiniowany model danych wdrożono do bazy grafowej Neo4j.
W modelu wyróżniono tylko jedną klasę (klasę wyrazu **Word**), a także zdefiniowano jedną relację: **NEXT** (relacja ta określa, jakie wyrazy mogą nastąpić po danym wyrazie).

3.7.2. Zapisywanie danych do modelu

W celu zapisywania danych do zdefiniowanego w bazie Neo4j modelu danych utworzono skrypt o nazwie: add_sentence.py

Skrypt ten w celu zapisania danych do bazy Neo4j korzysta z biblioteki neomodel. Biblioteka ta wymaga zdefiniowania w kodzie modelu danych znajdującego się w bazie, do której będą wysyłane dane.

```
from neomodel import StructuredNode, StringProperty, Relationship, config

NEO4J_CLUSTER_IP = "10.7.38.68"
config.DATABASE_URL = f'bolt://:@{NEO4J_CLUSTER_IP}:7687'


class Word(StructuredNode):
    name = StringProperty(unique_index=True, required=True)
next_words = Relationship('Word', 'NEXT')
```

Skrypt add_sentence.py nasłuchujena topic: ‘valid-msgs’. Dla każdej wiadomości wysłanej na ten topic wykonywana jest funkcja add_sentence(sentence).

```
KAFKA_CLUSTER_IP = "10.7.38.66"
bootstrap_servers = [KAFKA_CLUSTER_IP + ":9092", KAFKA_CLUSTER_IP +
":9093", KAFKA_CLUSTER_IP + ":9094"]
consumer = KafkaConsumer('valid-msgs',
bootstrap_servers=bootstrap_servers,
value_deserializer=lambda m: json.loads(m.decode('ascii')))

for message in consumer:
add_sentence(message.value.get('data'))
```

Funkcja add_sentence(sentence) dzieli wiadomość na wyrazy (tokeny). Następnie dla każdego wyodrębnionego wyrazu pobierany jest za pomocą funkcji get_word(word_name) node z bazy Neo4j reprezentujący dany wyraz. Jeśli w bazie Neo4j nie istnieje node dla jakiegoś wyrazu, to wówczas jest on tworzony. Następnie określane są relacje **NEXT** między pobranymi node'ami.

```
def get_word(word_name):
    word = Word.nodes.first_or_none(name=word_name)
if word is None:
        word = Word(name=word_name).save()
return word


def add_sentence(sentence):
token_list = TweetTokenizer().tokenize(sentence)
for iinrange(1, len(token_list)):
prev_word = get_word(token_list[i - 1])
curr_word = get_word(token_list[i])
prev_word.next_words.connect(curr_word)
    print('Sentence has been added.')
```

3.7.3. Generowanie tweetów

W celu wygenerowania tweetu utworzono api: ‘generate_sentence.py’

```
[root@rpd-11 markov-bot]# python3 generate_sentence.py --help
usage: generate_sentence.py [-h] [n_words]

Markov-bot

positional arguments:
  n_words     Maksymalna długość generowanej sentencji.

optional arguments:
  -h, --help  show this help message and exit
```

Utworzone api służy do generowani sentencji składających się z n wyrazów, gdzie n to parametr podany przez użytkownika. Jeśli użytkownik nie określi tego parametru, to wówczas zostanie wygenerowany tweet składający się z 20 wyrazów.

```
parser = argparse.ArgumentParser(description='Markov-bot')
parser.add_argument('n_words', nargs='?', type=int, default=20,
help='Maksymalna długość generowanej sentencji.')
args = parser.parse_args()
```

Generowanie tweetu odbywa się w następujący sposób:

- losowany jest pierwszy wyraz sentencji:

```
def get_random_first_word():
    words = Word.nodes.filter(name__regex='^[A-Z].*')
return random.choice(words)
```

- po wylosowaniu pierwszego wyrazu, losowane są kolejne wyrazy, które są w relacji NEXT z poprzednim wyrazem:

```
def get_random_next_word(word):
return random.choice(word.next_words)
```

- proces losowanie wyrazów jest powtarzany do momentu wygenerowania n wyrazów bądź do momentu gdy wylosowany wyraz nie ma następników:

```
def generate_sentence(n_words):
    word = get_random_first_word()
    text = word.name
for iinrange(1, n_words):
        word = get_random_next_word(word)
        text += ' ' + word.name
if word.next_wordsis None:
return text
return text
```

- Wygenerowany wyraz wypisywany jest na standardowe wyjście:

```
print(generate_sentence(args.n_words))
```

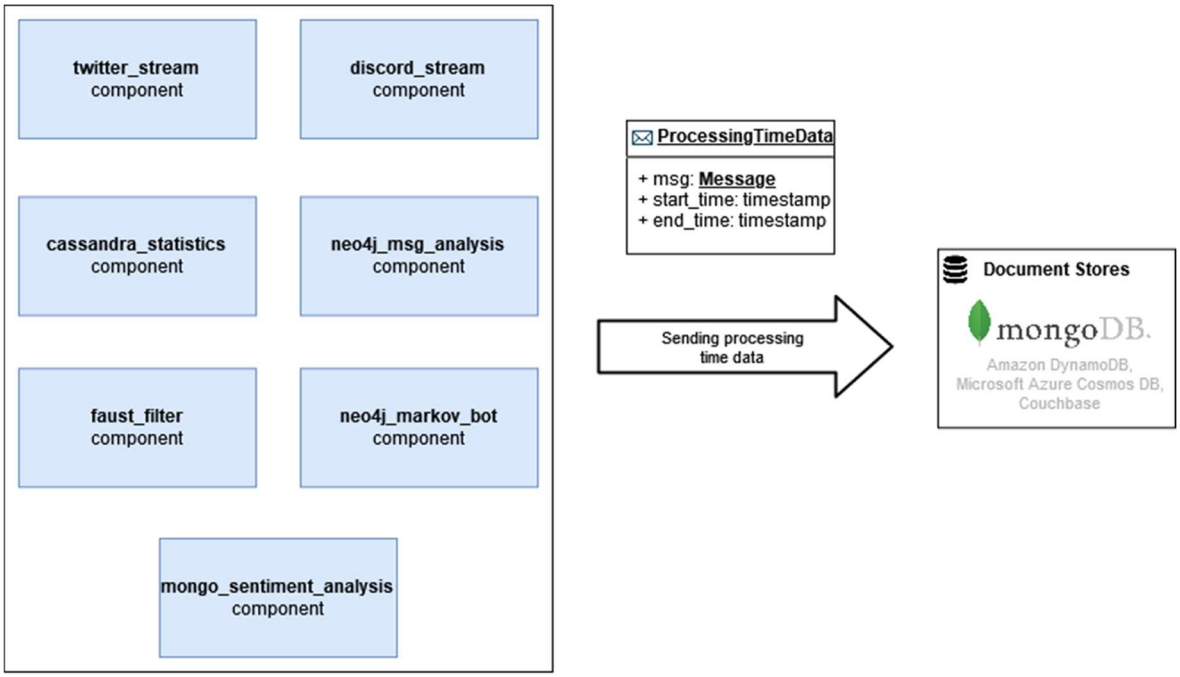
Poniżej przedstawiono kilka wygenerowanych tweetów przy użyciu zaimplementowanego api:

```
[root@rpd-11 markov-bot]# python3 ./generate_sentence.py
Kido Taylor-Hart Kido Taylor-Hart has Gifford collided Gifford Frank Worthington
 Kilbourne Worthington . @MayorBowser Sounds : @JoshfromBoston1 : ☹ vaccines
[root@rpd-11 markov-bot]# python3 ./generate_sentence.py 20
Obeying tyrannical that established It @_andrewalan It @jaromjordan It @TDs_Tang
ents It s last international flights 35 ( inadequate scandalously low
[root@rpd-11 markov-bot]# python3 ./generate_sentence.py 20
Cairngorms-valid in famous old act balancing about job Klopps job for Plans are
Trent and responsible during sandwiches with passengers
[root@rpd-11 markov-bot]# python3 ./generate_sentence.py 30
Wyre Constituency his bring him making pancakes , sick s call @NickOlsonNFL call
 yourselves Brace for presenting is ramping is Cerebroné is AD #Aztecs looking )
 Singleton ( stato )
[root@rpd-11 markov-bot]# python3 ./generate_sentence.py 30
LCFC on  #LHSAA . sleeping while … @ZachKrantz … Nova … republic … indica … Bun
des … hostels open Spartans are places where Colin Chun Colin O'Riordan , Fever
[root@rpd-11 markov-bot]# python3 ./generate_sentence.py 50
Discussion Today's of Sheriff in paying Mexico telling WR Josh you blaming Musli
ms all dey them voted 3rd for dining has Arabia Saudi patient contact #Worcester
shireHour ! Kibera in penny #keepitupwithbrexitpovertyinfo penny in Highs . Glob
alists wanted Mikel @AfcNoah14 Mikel ... Century .. joined to CDC @GOPChairwoman
 CDC " Rugby
[root@rpd-11 markov-bot]# python3 ./generate_sentence.py 50
Honored & futures . recommend don't pls help solve his politics twitter one Roan
 Sixth day will 70 minutes 630 Football FIFA Suspends CAF Champions Complete " A
lcohol ... First Responders During Pep v 5-1 v #Temple v leagues won hurlers @of
ficialdonegal football French tested improperly the virtues of symptoms
```

4. Weryfikacja czasu przetwarzania danych przez poszczególne komponenty systemu

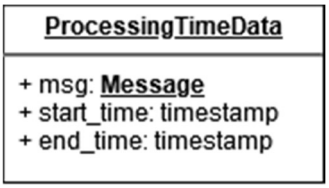
4.1. Opis procesu

W celu weryfikacji czasu przetwarzania danych przez poszczególne komponenty systemu utworzono następujący diagram:



W utworzonym diagramie wyróżniono 7 komponentów z zaprojektowanego systemu. W celu weryfikacji czasu przetwarzania danych przez te komponenty przewidziano, że każdy nich będzie zapisywał do bazy dokumentowej MongoDB czasy przetwarzania poszczególnych wiadomości. Następnie na podstawie przesłanych danych dla każdego komponentu wyznaczany będzie jego średni czas przetwarzania danych w pewnym określonym przez użytkownika zakresie dni.

Założono, że przesyłane przez komponenty dane do bazy MongoDB będą miały następującą strukturę:



4.2. Konfiguracja środowiska

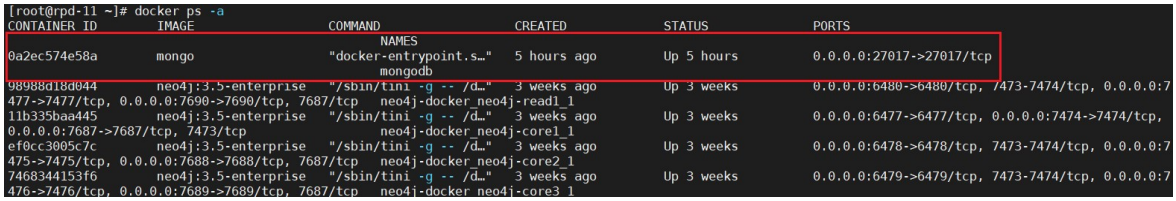
W celu przygotowania bazy MongoDB (do składowania czasów przetwarzania wiadomości przez poszczególne komponenty systemu) wykonano następujące polecenie:

```
docker run -d --name mongodb -p 27017:27017 mongo
```

W wyniku wykonania się polecenia, powinien zostać utworzony docker o nazwie mongodb. Aby to zweryfikować należy wykonać z linii poleceń następującą komendę:

```
dockerps -a
```

Powinien zostać zwrócony wynik podobny do:



Następnie w stworzonej bazie MongoDB utworzono db o nazwie ‘processing_time’ oraz tabele dla każdego komponentu:

```
docker exec -it mongodb mongo
use time_processing
db.createCollection('twitter_stream')
db.createCollection('discord_stream')
db.createCollection('cassandra_statistics')
db.createCollection('neo4j_msg_analysis')
db.createCollection('faust_filter')
db.createCollection('neo4j_markov_bot')
db.createCollection('mongo_sentiment_analysis')
```

4.3. Zapis do bazy

W celu zapisu danych do bazy danych przez poszczególne komponenty systemu, w każdym komponencie:

- zdefiniowano parametry połączeniowe do utworzonej bazy MongoDB:
  ```
  # MONGO PT
  MONGO_PT_IP = '10.7.38.68'
  MONGO_PT_PORT = 27017
  ```
- przed rozpoczęciem przetwarzania wiadomości przez dany komponent, utworzono obiekt ‘pt_data’ o strukturze zgodnej z ProcessingTimeData oraz uzupełniono jego parametry ‘msg’ oraz ‘start_time’:
  ```
  pt_data = {
  'msg': msg,
  'start_time': datetime.utcnow()
  }
  ```
- po zakończeniu przetwarzania wiadomości przez dany komponent, uzupełniono obiekt ‘pt_data’ o brakujący parametr ‘end_time’:
  ```
  pt_data['end_time'] = datetime.utcnow()
  ```
- kompletny obiekt ‘pt_data’ przesłano do bazy MongoDB:

```
mongo_pt_client = MongoClient(MONGO_PT_IP, MONGO_PT_PORT)
mongo_pt_db = mongo_pt_client["processing_time"]
mongo_pt_col = mongo_pt_db[component_name]
mongo_pt_col.insert_one(pt_data)
```

4.4. Wyznaczanie średniego czasu przetwarzania danych przez poszczególne komponenty systemu

Do weryfikacji czasu przetwarzania danych przez poszczególne komponenty systemu przygotowano api o nazwie ‘processing_time_info_api.py’:

```
[root@rpd-11 processing_time_info]# python3 processing_time_info_api.py -h
usage: processing_time_info_api.py [-h] [-sd STARTDATE] [-ed ENDDATE]
                                   collection_name

positional arguments:
  collection_name       available: ['cassandra_statistics', 'discord_stream',
                        'faust_filter', 'mongo_sentiment_analysis',
                        'neo4j_markov_bot', 'neo4j_msg_analysis',
                        'twitter_stream']

optional arguments:
  -h, --help            show this help message and exit
  -sd STARTDATE, --startdate STARTDATE
                        start date - format %Y%m%d (if left empty, data from
                        today is returned
  -ed ENDDATE, --enddate ENDDATE
                        end date - format %Y%m%d (usable alongside start date
                        only)
```

Utworzone api zwraca średni czas przetwarzania danych przez wskazany (jako parametr ‘collection_name’) komponent w zadanym odstępie czasu.

Jeśli użytkownik nie określi zakresu czasu, wówczas api zwróci średni czas przetwarzania danych w dniu dzisiejszym.

Przykładowy output:

```
[root@rpd-11 processing_time_info]# python3 processing_time_info_api.py discord_stream
Mean processing time: 0.009s
[root@rpd-11 processing_time_info]# python3 processing_time_info_api.py twitter_stream
Mean processing time: 0.005192846065537451s
[root@rpd-11 processing_time_info]# python3 processing_time_info_api.py cassandra_statistics
Mean processing time: 0.0030659966971442383s
[root@rpd-11 processing_time_info]# python3 processing_time_info_api.py neo4j_msg_analysis
Mean processing time: 0.3047249538261868s
[root@rpd-11 processing_time_info]# python3 processing_time_info_api.py faust_filter
Mean processing time: 0.1956417039925241s
[root@rpd-11 processing_time_info]# python3 processing_time_info_api.py neo4j_markov_bot
Mean processing time: 0.30843343826512765s
[root@rpd-11 processing_time_info]# python3 processing_time_info_api.py mongo_sentiment_analysis
Mean processing time: 0.0021930423870607925s
```

5. Wyniki

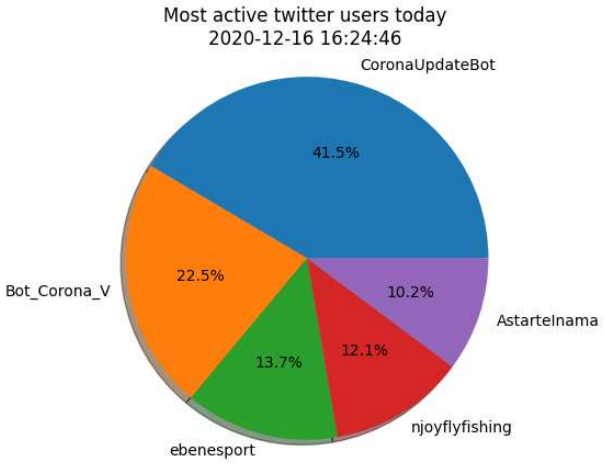
W niniejszym rozdziale przedstawiono wyniki przetwarzania danych przez wdrożony system z okresu 5 dni jego funkcjonowania: od 12.12.2020 r. do 16.12.2020r.
W trakcie działania systemu pobranych zostało 342383 wiadomości z różnych serwisów społecznościowych.

```
[root@rpd-9 ~]# docker exec -it kafka1 bash
root@kafka1:/# for i in `kafka-run-class kafka.tools.GetOffsetShell --broker-lis
t kafka1:19092,kafka2:19093,kafka3:19094 --time -1 --topic raw-msgs| awk -F : '{
print $3}'`; do sum=$(($sum+$i)); done; echo $sum
342383
root@kafka1:/#
```

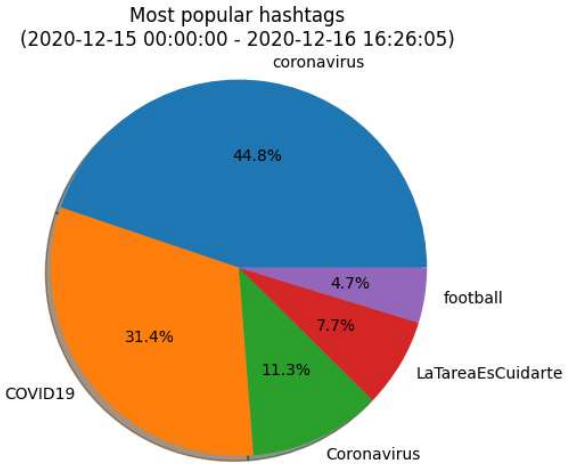
Poniżej przedstawiono tabelę zawierającą informacje o zajętym miejscu w użytych bazach danych, po wykonaniu przetwarzania i zeskładowaniu w nich danych wynikowych.

Tabela wolumetryczna		
Baza	Liczba rekordów	Zajęte miejsce (GB)
Cassandra	351311	1,112
Mongo	628173	2,014
Neo4j 1 – Tweets	816525	1,273
Neo4j 2 – Markov Bot	173919	0,784

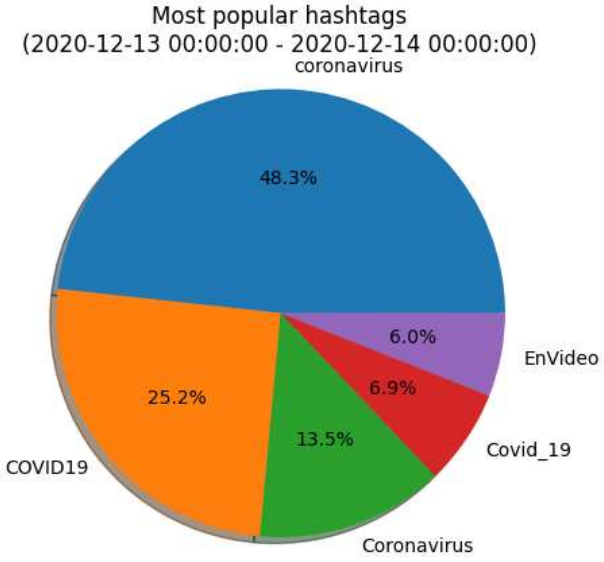
Na wykresie kołowym poniżej zwizualizowano najbardziej aktywnych użytkowników w dniu 16 grudnia 2020 do godz. 16:24.



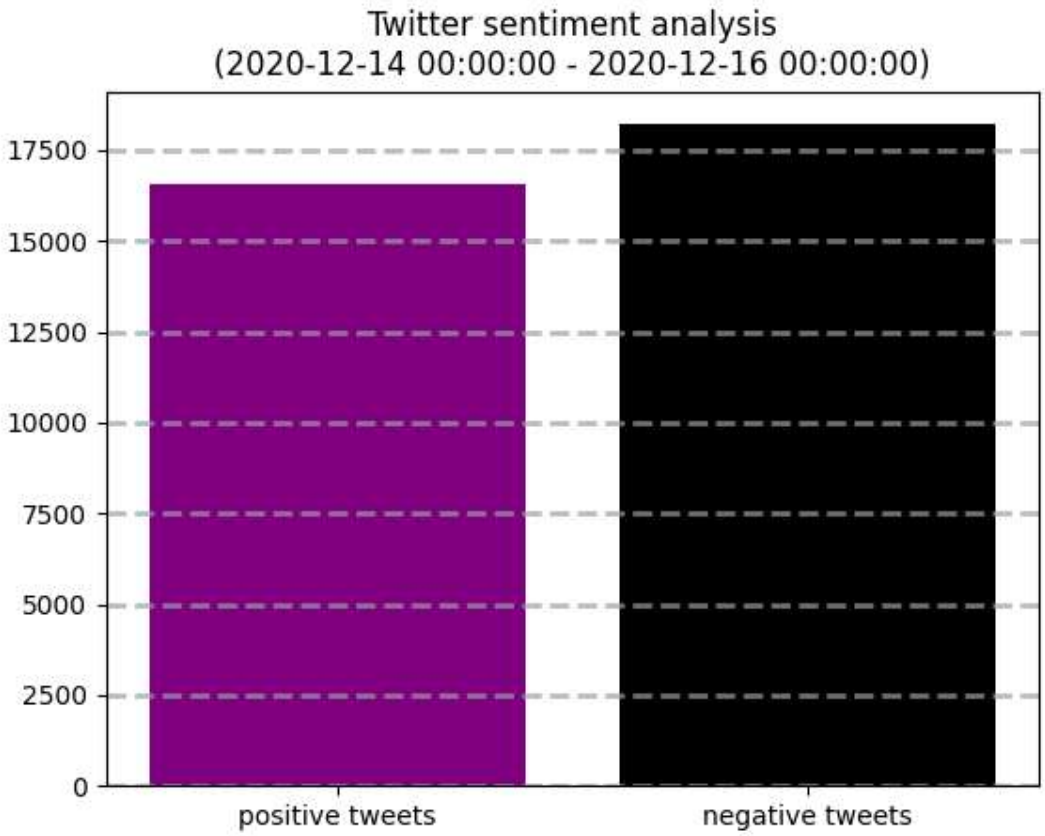
Poniższy wykres kołowy przedstawia najczęściej używane hashtagi w określonym przedziale czasowym.



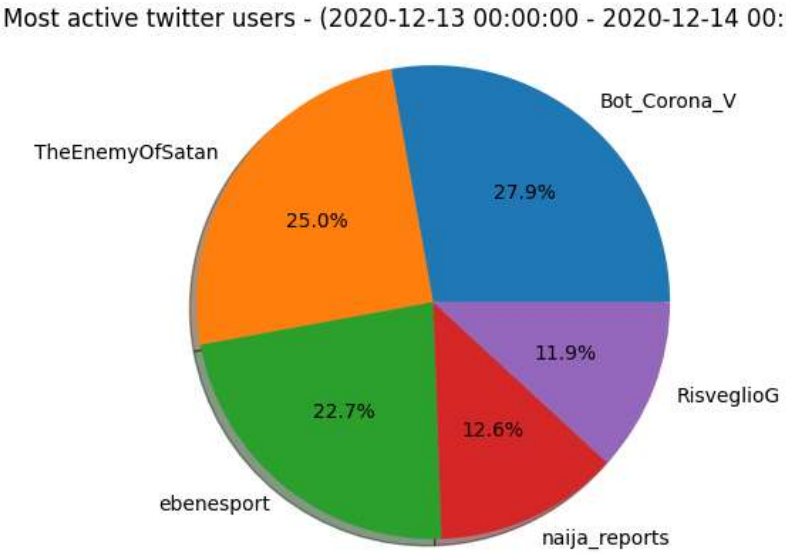
Dla porównania, wykres poniżej przedstawia hashtagi dla wcześniejszego przedziału czasowego.



Na poniższym wykresie słupkowym widać liczbę tweetów pozytywnych i negatywnych przyporządkowanych przez model analizy sentymentu w dniach 14, 15 i 16 grudnia 2020 r.



Najbardziej aktywnych użytkowników na Twitterze między 13 a 15 grudnia 2020 r. przedstawia poniższy wykres kołowy.



6. Podsumowanie

Cel zadania został osiągnięty. Udało się zrealizować wszystkie cele oraz wdrożyć wymagane przez prowadzącego elementy. W projekcie wdrożono wiele cykli zarówno przetwarzania strumieniowego, jak i wsadowego. Do kolejkowania danych wykorzystano Kafkę. Przetworzone dane zapisano do baz NoSQL – baz rodziny kolumn (Cassandra), baz dokumentowych (MongoDB) oraz baz grafowych (Neo4j). W projekcie wykorzystano mechanizm MapReduce do pobierania danych statystycznych odnośnie hashtagów. Ponadto w projekcie użyto również technik NLP, do przetwarzania tekstu, oraz ML, do wykrywania języka wiadomości oraz przewidywania sentymentu wiadomości.

W przyszłości należy skupić się na bezpieczeństwie aplikacji i całego sposobu konfiguracji. Każde dane pochodzące od użytkownika aplikacji powinny być odpowiednio modyfikowane po stronie serwera, przed dopuszczeniem ich do kontaktu z bazami danych. Ponadto należy zaimplementować dodatkowe zabezpieczenia przy przetwarzaniu danych np. szyfrowanie i odszyfrowywanie danych przez węzły przetwarzania oraz autoryzację i uwierzytelnianie każdego węzła wysyłającego dane przez węzeł odbierający. Należałoby ustawić loginy i hasła dla każdego komponentu przetwarzania, które byłyby zapisane w zmiennych systemowych, bądź w pliku konfiguracyjnym dockera, do którego należałoby ograniczyć dostęp. W ten sposób wrażliwe dane nie byłyby przechowywane w kodzie, a dodatkowo komponenty uwierzytelniałyby się automatycznie przy interakcji między sobą, pobierając dane zapisane w zmiennych systemowych, do których potencjalny atakujący nie miałby praw dostępu, nie uzyskując wcześniej uprawnień uprzywilejowanej grupy.

Kolejnym obszarem prac jest dodanie większej liczby źródeł przetwarzania strumieniowego, aby móc znajdywać cechy wspólne, jak i wykrywać różnice wiadomości z wielu źródeł. Należy też powiększać dane wsadowe o kolejne wiadomości do trenowania sentymentu, gdyż zwiększy to skuteczność obliczania sentymentu przez model. Oprócz tego należałoby dostosować wygląd aplikacji, która w tej chwili w wielu miejscach nie posiada zaimplementowanej wizualizacji. Analiza danych stałaby się prostsza, jednak wszystkie istotne dane można wyczytać z terminali maszyn poprzez uruchamianie przygotowanych do tego skryptów.