Analiza postów z użyciem LDA i wybranych narzędzi Big Data

Krzysztof Wojdalski

2017-09-18

Spis treści

Wstęp

W ostatnich latach narzędzia **Big Data** rosną na popularności. Czynnikiem determinującym taki stan rzeczy jest przede wszystkim wzrost wolumenu danych, zarówno na poziomie makro (danych w ogóle), ale też mikro, na poziomie przedsiębiorstwa, co związane jest z tzw. digitalizacją. Zjawisko przyrostu cyfrowej informacji sprawia, że w firmach pojawia się problem jej składowania, przetwarzania, wyciągania wniosków biznesowych, czy tworzenia produktu w oparciu o nią. Paradygmat **MapReduce**, kluczowy z perspektywy **Big Data** nie jest nową koncepcją, jednak dopiero od około **15** lat projekty, które adresowałyby opisane powyżej wyzwania są prężnie rozwijane, przy czym w ostatnich **10** latach znalazły zastosowanie komercyjne. W niniejszym projekcie, zrealizowanym w ramach studiów **Big Data** na **Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej**, została omówiona implementacja narzędzi służących do interakcji z dużymi bazami danych. Wykorzystano w tym celu przede wszystkim **Apache Spark**. Finalnym celem było zbudowanie modelu **LDA** na podstawie danych z portalu **Stack Exchange**, który byłby w stanie wyodrębnić tematy i kluczowe słowa w nich występujące.

Konstrukcja pracy opiera się o opis kolejnych kroków, które posłużyły do otrzymania finalnych wyników. W pierwszym rozdziale przedstawiono użyte narzędzia i ich krótką charakterystykę. Zawarto nie tylko sekcje o oprogramowaniu stricte **Big Data**, ale też o innych elementach, takich jak np. **RStudio Server**. W drugiej części autor skupił się na przedstawieniu etapów analizy, czyli na generalnym procesie budowy rozwiązania opartego o inżynierię danych i data science. Rozpoczyna się od zebrania surowych danych, czyli plików zawierających informacje ze **Stack Exchange**. Następnie przetworzenia, co oznacza doprowadzenie zbioru z półustrukturyzowanych plików znacznikowych (**XML**) do postaci ustrukturyzowanej w formie tabel w **Apache Hive**. Trzecim krokiem była eksploracja danych, czyli sprawdzenie, co dokładnie zawierają analizowane posty. W kontekście modelowania bardzo ważnym krokiem jest także czyszczenie używanego zbioru, m.in. pod kątem błędnych zapisów, wartości odstających (tzw. outlierów). Gdy dane zostały usystematyzowane, oczyszczone i wiarygodne, przeprowadzono proces modelowania. Użyta metoda text miningowa (**Latent Dirichlet Allocation**) posłużyła do przypisania poszczególnych postów do tematów, a także słów do tematów. Ostatnim krokiem, który znajduje się w rozdziale drugim to komunikacja wyników w formie wizualizacji, tabel, a także wnioskowania autora.

Trzeci rozdział stanowi podsumowanie oraz rozważania na temat potencjalnych kierunków, możliwości rozwinięcia projektu w przyszłości. Autor poruszył także wybrane aspekty, ograniczenia i problemy, jakie napotkał w trakcie tworzenia pracy.

Narzędzia użyte w pracy

Głównym wykorzystanym narzędziem **Big Data** jest **Apache Spark**, wymagającym instalacji **Scali** (a tym samym **JVM**). Do analizy danych wykorzystano ponadto **Apache Hive**, język **R** wraz z **IDE RStudio Server**. Poza narzędziami stricte **Big Data**, w pracy do prezentacji danych użyto **RMarkdown**, substytut **Jupyter Notebooks** znanego z **Python**'a, oraz innych paczek do przetwarzania i wizualizacji danych.

Poniżej została przedstawiona krótka charakterystyka tego, co zostało użyte w pracy. Należy dodać, że nie są to oczywiście wszystkie możliwości, jakie posiadają poniższe narzędzia, a jedynie ich mały wycinek.

Apache Spark

Apache Spark to silnik do przetwarzania dużych zbiorów danych. Jego podstawowy atut w stosunku do głównej (ale nie jedynej) alternatywy, **Apache Hadoop**, to szybkość - potrafi wykonać to samo zadanie (tzw. **job**) nawet w **100**x mniejszym czasie (np. w przypadku algorytmów iteracyjnych). Powód, dla którego **Spark** jest aż tak wydajnym narzędziem, wynika z jego architektury. Zapewnia on przetwarzanie w pamięci (**in-memory**), redukując ilość czasochłonnych operacji typu **read/write**. **Hadoop** natomiast zawiera komponenty **HDFS (Hadoop Distributed File System)** oraz **MapReduce**, które działają w oparciu o zapis danych na fizycznych dyskach.

Spark stanowi ponadto zdecydowanie bardziej przyjazne użytkownikowi środowisko. Nie ogranicza interaktywnej eksploracji danych, wykorzystywania cząstkowych wyników (bez konieczności wcześniejszego ich zapisu i odczytu) Narzędzie zostało napisane w **Scali**, ale jest ono również dostępne w **Javie**, **Pythonie** (**PySpark**) i **R** (**SparkR**, **sparklyr**). **Apache Spark** jako projekt to nie tylko silnik, ale też **SQL**, w którym można pisać kwerendy na rozproszonych ramkach danych (**DataFrame**'ach), biblioteki do uczenia maszynowego (**ML/MLLib**) oraz **GraphX** do analizy grafów. Tego typu integracja pozwala na tworzenie produktów, np. systemów rekomendacyjnych, W przypadku uczenia maszynowego, które zostało wykorzystane w ramach projektu interesującym atrybutem są **pipeline**'y, które w łatwy sposób pozwalają od zera zbudować pełnoprawny model oparty o machine learning.

Apache Hive

Apache Hive to narzędzie służące do analizy i wydobywania danych z użyciem paradygmatu **MapReduce**. W swojej istocie podobne jest do języka zapytań **SQL** (używa **HiveQL**). Omawiany software został zaprojektowany w **Facebook**'u z myślą o pracownikach, którzy chcieliby w łatwy sposób uzyskać interesujące dane, a niekoniecznie są inżynierami. Dzięki integracji z ekosystemem **Apache Hadoop**, ale również ze **Sparkiem**, **Hive** zapewnia taki poziom abstrakcji, który pozwala w prosty sposób pisania kwerend, bez konieczności pisania dodatkowego kodu w **Javie**. Każde zapytanie przekształcane jest w zadanie **MapReduce** bądź **Apache Spark**. Z racji **fault tolerance**, **Hive** szczególnie dobrze nadaje się do długich procesów, gdzie zadanie zostanie zakończone nawet, gdy pojawi się np. błędny odczyt. Alternatywnym rozwiązaniem jest stworzona przez **Clouderę Impala**.

Apache Hue

Hue (**Hadoop User Experience**) to webowy interface, który agreguje narzędzia z ekosystemu hadoopowego. Służy do przeprowadzania analizy danych a także egzekucji zadań z poziomu przeglądarki, co stanowi alternatywę dla komend w terminalu. W obecnej wersji jest zintegrowany z bazami danych (m.in. **Hive**, **Impala**, **MySQL**, **PostgreSQL**, **Oracle**), umożliwia korzystanie z notebooków w **Pythonie**.

Apache YARN

Apache Hadoop YARN (**Yet Another Resource Negotiator**) służy do zarządzania klastrem - jego podstawowe funkcje to przydzielanie zasobów, monitorowanie zadań i harmonogramowanie. W kontekście pracy **YARN** jest o tyle istotny, że za jego pomoca można w prosty sposób łączyć się z danymi w **Hive** z poziomu sesji w **Apache Spark**, a także w razie problemów zarządzać istniejącymi procesami.

R i RStudio

R jest wysokopoziomowym językiem programowania służącym głównie do pracy z danymi. W ostatnich latach jest on niezwykle popularny wśród statystyków, data scientistów i analityków. Jego ewidentnym plusem jest jego ekspresywność i elastyczność, przez co prototypowanie jest wielokrotnie szybsze niż np. w **Javie**. Ponadto, z racji bazy użytkowników i ich specyfiki, implementacji najnowszych metod i modeli statystycznych pojawia się w **R** (obok **Python**'a) zwykle dużo szybciej niż w innych językach programowania. R posiada system bibliotek (paczek) - najpopularniejszy to **R CRAN** (**MRAN**) - dzięki którym można rozszerzyć możliwości podstawowej wersji o dodatkowe funkcjonalności, w tym także te wykorzystujące paradygmaty Big Data. Społeczność **R** stale się powiększa, co powoduje, że często zdarza się, że dany problem został rozwiązany (zaimplementowany) przynajmniej na kilka sposobów. Tak jest też ze **Sparkiem** - w tej chwili połączenie silnika **R** z silnikiem może się odbywać za pomocą dwóch głównych paczek:

- **SparkR** - oficjalnie wspieranej przez autorów oryginalnego projektu **Apache Spark**. **SparkR** jest konsystentny z analogicznymi pakietami dla **Scali** i **Python**'a, poprzez podobne funkcje i ich nazewnictwo.

Istotnym plusem jest możliwość pisania własnych funkcji, a także większa elastyczność, jeżeli chodzi o strukturę danych, z którą można pracować.
- **sparklyr** - paczka stworzona przez **RStudio Inc.** Sprawdza się lepiej w przypadku użytkowników pracujących wcześniej w **R**. Jest spójny z tym, co znane jest z innych popularnych pakietów, takich jak np. **dplyr**, **plyr**, **purrr**. Ze względu na ten właśnie fakt projekt będzie finalnie wykorzystywał pakiet **sparklyr**.

Elastic Map Reduce

Do obliczeń został wykorzystany serwis **Amazon EMR**, dostępny na **Amazon Web Services (AWS)**. Jest to rozwiązanie oparte o instancje **Amazon EC2**. Pozwala na wykorzystanie zdecydowanej większości znaczących projektów **Apache** stworzonych pod kątem **Big Data**. Nadaje się do niemal każdego przypadku, który wymaga infrastruktury do dużych zbiorów danych, np. web indexing, ETL, uczenia maszynowego, systemów finansowych, prac naukowych wymagających większej mocy obliczeniowej.

W ramach tego konkretnego projektu na klastrze zostały zainstalowane następujące narzędzia:

- Hadoop 2.7.3
- Hue 3.12.0
- Spark 2.2.0
- Hive 2.3.0
- Zeppelin 0.7.2
- R 3.4.0
- RStudio Server 1.0.153
- Scala 2.11.8

Proces budowy rozwiązania Big Data

W tym rozdziale został omówiony proces budowania rozwiązania - od konfiguracji klastra **Amazon EMR**, przez załadowanie danych, ich wyczyszczenie, doprowadzenie do ustrukturyzowanego typu, po budowę modelu text miningowego.

Konfiguracja Amazon Elastic Map Reduce

Klaster **Amazon EMR** wymagał następującej konfiguracji:

- Dobór narzędzi - tych wyszczególnionych w sekcji **Elastic Map Reduce**
- Wielkości klastra - użyto **1x master node m4.large**, **2x slave node m4.large**)
- Zabezpieczeń - komunikacja poprzez protokół **SSH**. Należało odblokować porty oraz dodać adres IP lokalnego komputera, który posłużył do łączenia się z klastrem

W znaczącej części konfiguracja została dokonana poprzez połączenie za pomocą dedykowanego narzędzia konsolowego. Jest ono szczególnie użyteczne, gdy występuje potrzeba zreplikowania klastra. W przypadku tego projektu klaster był tworzony trzykrotnie. Od uruchomienia poniższej komendy do instalacji całego oprogramowania, włącznie z tymi spoza domyślnego katalogu aplikacji **EMR** minęło około **40-50** minut. Niestety, na ten moment **Amazon** nie oferuje prostego rozwiązania typu black box, które tworzyłoby obrazy każdej z instancji, a następnie dystrybuowałoby je po węzłach. Proces przygotowania środowiska można mocno usprawnić poprzez napisanie odpowiednich skryptów w **bash**'u. W przypadku projektu jednak ten proces jednak podzielony na kroki, by kilkukrotnie przećwiczyć kolejne operacje w środowisku **Big Data**.

```
aws emr create-cluster --termination-protected --applications Name=Hadoop
Name=Hive Name=Pig Name=Hue Name=Zeppelin Name=Spark
Name=HCatalog --ec2-attributes
```

```
'{"KeyName":"emr_bigdata_pw","InstanceProfile":"EMR_EC2_DefaultRole",
"SubnetId":"subnet-0b368a46","EmrManagedSlaveSecurityGroup":"sg-9f38e5f7",
"EmrManagedMasterSecurityGroup":"sg-de3ee3b6"}' --release-label
emr-5.8.0 --log-uri 's3n://aws-logs-373664525226-us-east-2/elasticmapreduce/'
--instance-groups '[{"InstanceCount":1,
"EbsConfiguration":{"EbsBlockDeviceConfigs":
[{"VolumeSpecification":{"SizeInGB":32,"VolumeType":"gp2"},
"VolumesPerInstance":1}]},"InstanceGroupType":"CORE",
"InstanceType":"m4.large","Name":"Core - 2"},{"InstanceCount":1,
"EbsConfiguration":{"EbsBlockDeviceConfigs"
:[{"VolumeSpecification":{"SizeInGB":32,"VolumeType":"gp2"},"VolumesPerInstance":1}]},
"InstanceGroupType":"MASTER","InstanceType":"m4.large","Name":"Master - 1"}]'
--configurations '[{"Classification":"hive-site","Properties":
{"hive.metastore.client.factory.class":
"com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory"},
"Configurations":[]},{"Classification":"spark-hive-site",
"Properties":{"hive.metastore.client.factory.class":
"com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory"},
"Configurations":[]}]' --auto-scaling-role EMR_AutoScaling_DefaultRole
--ebs-root-volume-size 10 --service-role EMR_DefaultRole
--security-configuration 'Security_big_data' --enable-debugging
--name 'My cluster' --scale-down-behavior
TERMINATE_AT_INSTANCE_HOUR --region us-east-2
```

Podłączenie do klastra

Podłączenie do klastra nastąpiło poprzez nawiązanie połączenia za pomocą protokołu **SSH**. Poniżej znajduje się przykładowy kod, który uruchamia to połączenie. Istotny jest parametr `-i`, który specyfikuje lokalizację klucza do uwierzytelniania połączenia. Klucz ten został wytworzony w serwisie **Amazon**'a, po czym ściągniety na lokalny dysk.

```
ssh -i ~/Downloads/emr_bigdata_pw.pem hadoop@ec2-52-15-164-251.us-east-2.compute.amazonaws.com
```

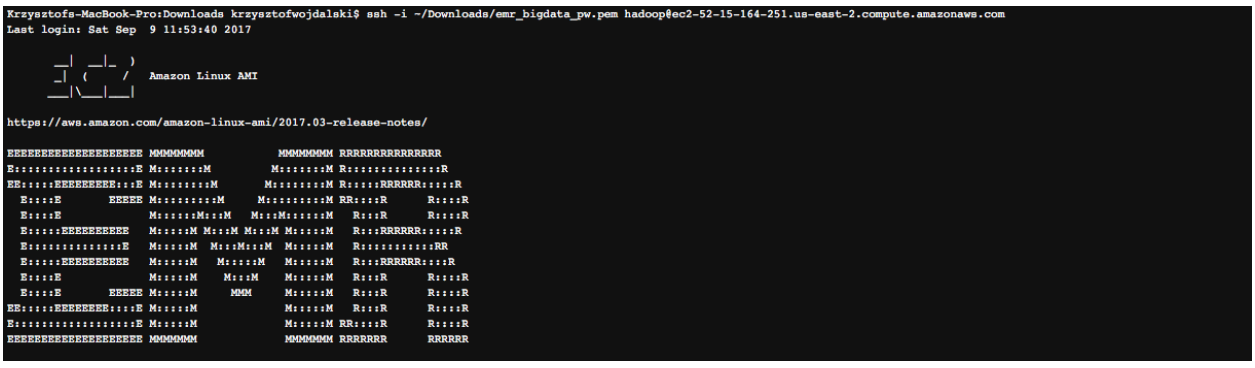

Figure 1: Konsola Elastic Map Reduce

Następnym krokiem była aktualizacja domyślnego systemu operacyjnego , na którym działa **EMR**. Odbyło się to poprzez system `yum`, służący do zarządzania pakietami. Ponadto należało zainstalować biblioteke, która wymagana jest do działania biblioteki `devtools` i **RStudio Server**'a w ogóle. Ostatnie linie kodu

stanowią komendy, które ściągały (`wget`), instalowały (`yum install`), a następnie dodawały użytkownika w systemie operacyjnym (`useradd`) i (`passwd`). Wszystkie operacje należało uruchamiać z użyciem `sudo` (super użytkownika z permisją m.in. do instalacji). Ponadto w systemie plików **HDFS** dodano folder, do którego trafiały nieobrobione dane - z liberalnymi uprawnieniami `-chmod 777`.

```
sudo yum update
sudo yum install libcurl-devel openssl-devel
sudo yum install R
wget https://download2.rstudio.org/rstudio-server-rhel-1.0.153-x86_64.rpm
sudo yum install --nogpgcheck rstudio-server-rhel-1.0.153-x86_64.rpm

sudo useradd -m kw
sudo passwd kw # Hasło

# Create new directory in hdfs
hadoop fs -mkdir /user/kw
hadoop fs -chmod 777 /user/kw
```

Transport danych do środowiska Big Data

Dane

Do stworzenia pracy posłużyły dane (w formacie **XML**) ściągniete ze **Stack Exchange**. Jest to serwis, w którym można tworzyć pytania a także odpowiadać na posty innych użytkowników w ramach różnych kategorii. Pomoc innym pozwala w zdobywaniu reputacji, która potem może przekładać się na poszanowanie wśród tej społeczności. Ponadto, tematy z wysoką ilością przydzielonych punktów pojawiają się wyżej w rankingu rekomendującym podobne pytania do wyszukiwanego, co powoduje, że użytkownicy z większym prawdopodobieństwem i szybciej znajdują odpowiedź na nurtujący ich problem. Serwis jest podzielony tematycznie, a każda ze stron posiada swój własny system do moderowania. W kontekście pracy zostały wykorzystane posty z tematu **music.stackexchange.com**, w którym udzielają się muzycy. W ściągniętych plikach najistotniejszy (i jedyny wykorzystany) to **Posts.xml**. Choć plik był w formie zbliżonej do **XML**'a, to jednak nie udało się go jednak sparsować poprzez funkcję w jednej z dedykowanych bibliotek (`spark-xml`). Struktura pliku została ustalona na podstawie znaczników i była taka, jak poniżej:

- `Id` - Id tematu
- `ParentId` - odniesienie do szerszego zagadnienia (tematu)
- `AcceptedAnswerId` - Id zaakceptowanej odpowiedzi innego użytkownika
- `CreationDate` - data utworzenia tematu w formacie zbliżonym do datetime (dotyczy też kolejnych zmiennych z nazwą *Date)
- `ViewCount` - ilość wyświetleń
- `Score` - ilość punktów przyznawanych w serwisie
- `Title` - tytuł zapytania
- `Body` - treść tematu, które zwykle zawiera skonkretyzowanie zapytania
- `LastEditorUserId` - Id użytkownika, który jako ostatni zedytował temat
- `LastEditDate` - data ostatniej edycji
- `LastActivityDate` - data ostatniej aktywności w temacie, czyli data ostatniego postu w temacie
- `Tags` - tagi, poprzez które można wyszukać interesujące zapytania
- `AnswerCount` - ilość odpowiedzi w temacie
- `CommentCount` - ilość komentarzy w temacie
- `FavoriteCount` - ilość polubien tematu, które zwykle oznaczają to, że dany użytkownik serwisu znalazł interesujące odpowiedzi w temacie bądź też po prostu uważa, że zagadnienie jest warte uwagi

Załadowanie danych do HDFS i Hive

Plik z postami należało najpierw umieścić na klastrze, wysyłając go uprzednio z użyciem protokołu **SSH** i poniższej pętli w **bash**'u.

```
# Klaster - stworzenie tymczasowego folderu dla danych
mkdir /tmp/music_datasets
# Lokalny dysk - przeniesienie danych z folderu na klaster
DIR=~/Downloads/music_datasets
for d in $(ls ${DIR}/);
do
scp -i ~/Downloads/emr_bigdata_pw.pem ${DIR}/${d} \
hadoop@ec2-52-15-164-251.us-east-2.compute.amazonaws.com:../../tmp/music_datasets;
# sciezka z automatu wskazuje na folder z hadoopem, stąd kropki
#
done;
```

```
Krzysztofs-MacBook-Pro:Downloads krzysztofwojdalski$ DIR=~/Downloads/music_datasets
Krzysztofs-MacBook-Pro:Downloads krzysztofwojdalski$ for d in $(ls ${DIR}/);
> do
> scp -i ~/Downloads/emr_bigdata_pw.pem ${DIR}/${d} hadoop@ec2-52-15-164-251.us-east-2.compute.amazonaws.com:../../tmp/music_datasets;
> # sciezka z automatu wskazuje na folder z hadoopem, stąd kropki
> done;
Badges.xml                                    100% 5207KB 288.6KB/s   00:18
Comments.xml                                  100%   18MB 331.4KB/s   00:54
PostHistory.xml                               100%   78MB 481.6KB/s   02:46
PostLinks.xml                                 100%  320KB 319.4KB/s   00:01
Posts.xml                                     100%   49MB 574.3KB/s   01:27
Posts_example.xml                             100% 3080     9.9KB/s   00:00
Tags.xml                                      100%   35KB  64.8KB/s   00:00
Users.xml                                     100% 9417KB 626.7KB/s   00:15
Votes.xml                                     100%   16MB 607.9KB/s   00:27
```

Figure 2: Przesłanie plików poprzez protokół SSH

Następny krok stanowiło przeniesienie danych do odpowiedniego folderu i innego systemu plików, czyli na **HDFS**.

```
# Stworzenie folderu kw na HDFS
hadoop fs -mkdir /user/kw/
# Kopia na HDFS z folderu tymczasowego
hadoop fs -put /tmp/music_datasets /user/kw/
```

W celu sprawdzenia, czy dane zostały poprawnie przeniesione, użyte został **Hue**, czyli graficzny interfejs do ekosystemu projektów Big Data.

```
http://52.15.164.251:8888/
kw / Kw8307@!
```

Kolejnym etapem było stworzenie tabeli w **Hive**. Celem takiego zabiegu było ustrukturyzowanie danych, by potem można było je wydajnie eksplorować. Do tego momentu bowiem struktura pliku załadowanego do **HDFS** była kolejnymi linijkami pliku **XML**, bez struktury z typami. Każda liczba, czy data i tak była traktowana jako `String`.

```
hive # uruchomienie konsoli Hive
-- Stworzenie pustej tabeli do przechowywania linii pliku XML
DROP TABLE temp_posts;
CREATE TABLE temp_posts (Text string);
LOAD DATA INPATH '/user/kw/music_datasets/Posts.xml' into TABLE temp_posts;

-- Stworzenie pustej tabeli wraz z typami spodziewanych danych
DROP TABLE POSTS;
CREATE EXTERNAL TABLE  posts
```

```
(
  Id int,
  ParentId int,
  AcceptedAnswerId int,
  CreationDate timestamp,
  ViewCount int,
  Score int,
  Title string,
  Body string,
  LastEditorUserId string,
  LastEditDate timestamp,
  LastActivityDate timestamp,
  Tags string,
  AnswerCount int,
  CommentCount int,
  FavouriteCount int
)

TBLPROPERTIES("skip.header.line.count"="2");

insert overwrite table posts
SELECT
regexp_extract(Text, 'Id="([0-9]+)"',1) Id,
regexp_extract(Text, 'ParentId="([0-9]+)"',1) ParentId,
regexp_extract(Text, 'AcceptedAnswerId="([0-9]+)"',1) AcceptedAnswerId,
cast(regexp_replace(regexp_extract(Text, 'CreationDate="(.*?)"',1), '[A-Z]',' ')
  as TIMESTAMP) CreationDate,
regexp_extract(Text, 'ViewCount="([0-9]+)"',1) ViewCount,
regexp_extract(Text, 'Score="([0-9]+)"',1) Score,
regexp_replace(regexp_extract(Text, 'Title="(.*?)"',1),'[^&;,.<>\\/\\\\A-Za-z 0-9]','') Title,
regexp_replace(regexp_extract(Text, 'Body="(.*?)"',1),'[^&;,.<>\\/\\\\A-Za-z 0-9]','') Body,
regexp_extract(Text, 'LastEditorUserId="([0-9]+)"',1) LastEditorUserId,
cast(regexp_replace(regexp_extract(Text, 'LastEditDate="(.*?)"',1), '[A-Z]', ' ')
  as TIMESTAMP) LastEditDate,
cast(regexp_replace(regexp_extract(Text, 'LastActivityDate="(.*?)"',1), '[A-Z]', ' ')
  as TIMESTAMP) LastActivityDate,
regexp_replace(regexp_extract(Text, 'Tags="(.*?)"',1),'[^&;,.<>\\/\\\\A-Za-z 0-9]','') Tags,
regexp_extract(Text, 'AnswerCount="([0-9]+)"',1) AnswerCount,
regexp_extract(Text, 'CommentCount="([0-9]+)"',1) CommentCount,
regexp_extract(Text, 'FavoriteCount="([0-9]+)"',1) FavouriteCount
from temp_posts;
```

Poniżej screenshot obrazujący podgląd danych w **Hive** za pomocą **Hue**.

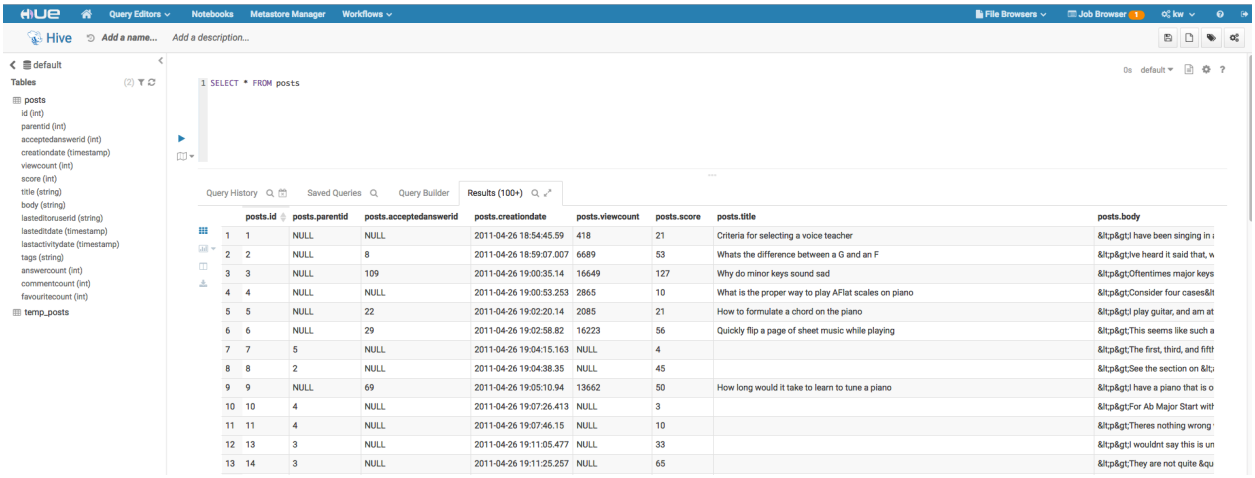

Figure 3: Interfejs Apache Hue

Analiza danych w Hive

W tej części zobrazowano wybrane możliwości **Hive** w postaci kwerend. Każda z nich jest de facto zadaniem - tekst z **HQL** zostaje przekształcony na **job** w **MapReduce**. Poniżej zostało napisane zapytanie, które podlicza posty z jakimś wynikiem (`score` inny niż `NULL`) i agreguje dane po godzinach.

```
SELECT hour(creationdate) AS hour,
       count(score) AS COUNT
FROM posts
WHERE score IS NOT NULL
GROUP BY hour(creationdate);
```

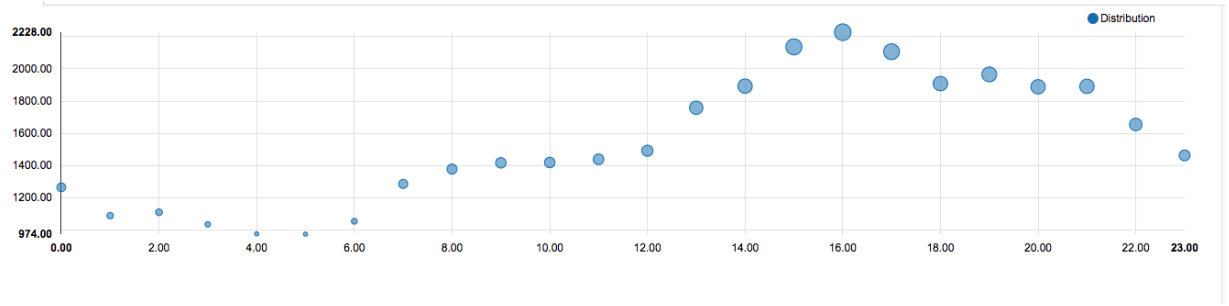

Figure 4: Rozkład postów w ciągu dnia

Z powyższego wykresu, można wywnioskować, że chociaż **Stack Exchange** jest serwisem globalnym, to jednak zdecydowana większość użytkowników pisze posty w godzinach popołudniowych czasu **UTC**.

Następne zapytanie **HQL** związane było z brakami w danych. Po wczesnej inspekcji części danych w **XML** można było stwierdzić, że występuje sporo wartości `NULL` bądź pustych danych typu `String`. By sprawdzić skalę zjawiska zostało utworzone poniższe zapytanie.

```
SELECT concat('Rows: ', cast(count(*) AS String)),
       concat('Nulls in viewcount: ', cast(cast((1.0 -count(viewcount)/count(*))
```

```
          * 100.0 AS decimal(4,1)) AS string), '%'),
       concat('Nulls in creationdate: ', cast(cast((1.0 -count(creationdate)/count(*))
          * 100.0 AS decimal(4,1)) AS string), '%'),
       concat('Zero length title: ',cast(sum(if(length(title)>0, 0,1))/count(*)
          * 100.0 as DECIMAL(4,1)), '%'),
       concat('Zero length body: ',cast(sum(if(length(body) >0, 0,1))/count(*)
          * 100.0 as DECIMAL(4,1)), '%'),
       concat('Zero length tags: ',cast(sum(if(length(tags) >0, 0,1))/count(*)
          * 100.0 as DECIMAL(4,1)), '%')


FROM posts;
```

Na jego podstawie zostało stwierdzone, że wśród **37652** wierszy:

- W **73.6%** przypadków (wierszy) brakowało wartości liczbowej w kolumnie `ViewCount`
- Timestampy dla wykreowanego tematu pojawiały się zawsze (**0%** `NULL`'i)
- Tematu posta brakowało w **73.6%**
- Treść posta występowała zawsze
- Tagów brakowało w **73.6%**

Należy zaznaczyć, że była to wstępna analiza - przyczyna występowania braku wartości niekoniecznie jest determinowana błędami w back-end **Stack Exchange**, a raczej specyfiką zapisu nowych rekordów do bazy (wystawionego pliku **XML**).

Kolejne zapytanie będzie służyło do policzenia słów w treści postów. W tym celu zostanie użyty `LATERAL VIEW` z funkcją `EXPLODE()`, który pozwala na analizę kolumny wektorów. Taka struktura danych jest użytecznym rozwiązaniem pomocnym w redukcji ilości wierszy i/lub kolumn. W przypadku z projektu jednak do zliczenia słów wymagana jest długa tablica (**long table**), gdzie każde słowo z każdego posta będzie reprezentowane przez jeden wiersz. Dodatkowo treść postów została oczyszczona ze zbędnych znaków, które zaburzały analizę.

```
SELECT lower(word) word ,
       count(word) as count
FROM posts
LATERAL VIEW explode(
split(
  regexp_replace(
  regexp_replace(BODY,
  '[0-9]|0[0-9]+|&amp|&xD|&lt(.*?)&gt|(&lt)|(&gt)|(&quot)|(;p)|(&xA)|(\/p)|(\/li)|(/\ol)|[;,.]',
  ' '),
  ' {2,}',' '),
  ' +|\/')) visitor AS word
WHERE length(word)>0
GROUP BY lower(word)
ORDER BY count desc
```

Z rezultatu zapytania wyszło, że zdecydowanie najpopularniejszym słowem jest **the** (**331** tys. wystąpień), następnie **a** (**185** tys.), **to** (**178** tys.), **and** (**133** tys.). Warto nadmienić, że z racji wyboru zbioru danych wiele słów jest jednoliterowa (nuty zapisuje się jako pojedyncze litery). Przykładowo najpopularniejszą występującą nutą jest **C** (**19** tys. wystąpień), poza **A**, które pojawia się wielokrotnie częściej, jednak w języku angielskim jest też przedimkiem. W części dotyczącej modelu wyszczególnione najpopularniejsze wyrazy, jak i wiele innych, należało wyrzucić z analizy. Było to determinowane ich neutralnością - przykładowo słowo **the** można przypisać, co do zasady, do jakiegokolwiek tematu.

Po napisaniu powyższego kodu **HQL** należało przejść do głównej części projektu, czyli modelu w **Spark**'u.

Logowanie do wcześniej zainstalowanego **RStudio Server**'a odbywa sie poprzez adres **http://18.220.255.**

40:8787/ . Po załadowaniu i zainstalowaniu bibliotek (paczek) do **R** (zostały wylistowane we wcześniejszej części pracy), połączenie ze **Sparkiem** (parametr master, czyli url klastra ustawiony na `yarn-client`) wskazało, że faktycznie tabela z użyciem **Hive** została utworzona.

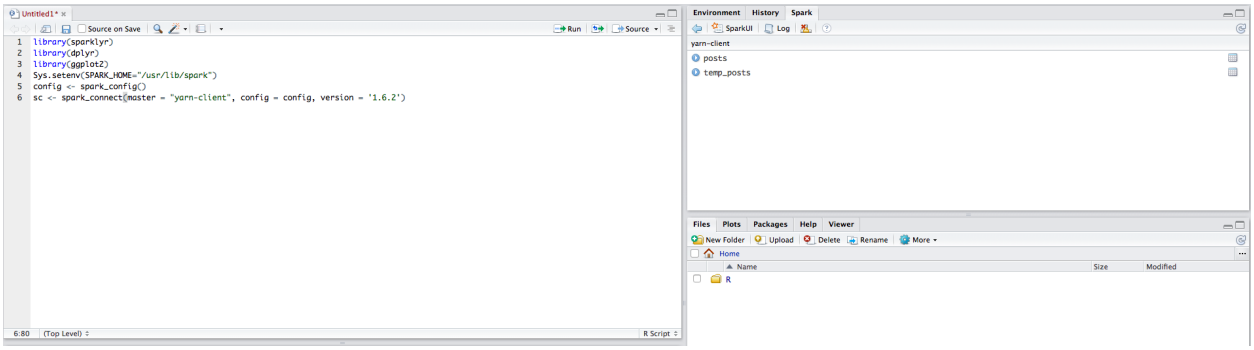

Figure 5: Interfejs RStudio Server'a

Model LDA w Apache Spark

Opis modelu

Latent Dirichlet Allocation to generatywny model probabilistyczny oparty o **rozkład Dirichleta**. Został on opublikowany w **2003** roku przez **Davida Blei**, **Andrew Ng** i **Michaela I. Jordana**. W swojej istocie służy do opisywania dokumentów, czyli np. tweetów, postów w internecie, książek. **LDA** opiera się na dwóch podstawowych założeniach:

- Każdy dokument jest zlepkiem tematów - oznacza to, że każdy dokument może zawierać słowa (w różnej proporcji) przypisane do różnych tematów. Przykładowo jeżeli post ze **Stack Exchange** brzmi: "Lubię jeść pomidory i lubię głaskać koty" to na podstawie modelu **LDA** można stwierdzić, że tekst w **50% traktuje o warzywach i w 50% o zwierzętach**
- Każdy temat jest zlepkiem słów - można sobie wyobrazić sytuacje, w której mamy dwa tematy - wcześniej wspomniane **zwierzęta** i **warzywa**. Najbardziej popularne słowa z tego pierwszego to np. **kot**, **pies**, zaś z drugiego **pomidor** i **ogórek**. Warto tu zaznaczyć, że słowa mogą przynależeć do obu grup.

LDA zawiera algorytm do estymacji obu tych zagadnień jednocześnie, czyli zasobu słów przynależących do tematów oraz tematów przynależących do dokumentu. Za pomocą implementacji modelu w **ML** (**sparklyr** wywołuje zadania z tej właśnie biblioteki) w dalszej części rozdziału został wymodelowany drugi z wyżej wymienionych.

W kontekście pracy ważnym postawionym pytaniem było to, dlaczego i do czego taki algorytm jest przydatny. Intuicyją odpowiedzią jest np. automatyzacja procesu tagowania postów w serwisie **Stack Exchange** bądź podobnym. Aplikacja takiego narzędzia jest jednak dużo szersza. Blei (2012) poruszył szerszy problem - obecny system wyszukiwania informacji w internecie jest niedoskonały. Opiera się o słowa klucze, które wpisywane są do silnika wyszukującego, który z kolei tworzy ranking i przedstawia zbiór dokumentów użytkownikowi (np. w formie linków). Autor stwierdził, że algorytmy do modelowania tematów mogą posłużyć do znacznie lepiej rozwiniętego mechanizmu, w którym eksploracja dokumentów opierałaby się o motywy (topics), które można generalizować bądź uszczegóławiać. Przykładowo, mając dokument o bioinformatyce można założyć, że zawiera on kilka motywów przewodnich. Blei wskazuje na **genetykę**, **ewolucję**, **chorobę** i **komputery** - jest to wybór arbitralny. Algorytm **LDA** działa w ten sposób, że zakłada najpierw pewien rozkład tematów - cały zbiór nie musi traktować o każdym z nich w równym stopniu. Następnie dla każdego słowa przydzielany jest temat, który go reprezentuje. Efektem jest macierz o rozmiarze $n \times k$, gdzie n to rozmiar słownika (ilość zanalizowanych słów), k liczba tematów, zaś wewnątrz jej są wartości prawdopodobieństwa $P(k = x|n)$. Na

tej podstawie można stwierdzić, że jeżeli słowo **człowiek**, **genom** i **dna** występują w części dokumentu, to można uznać z dużym ten fragment całego artykułu traktuje o genetyce. Taka wiedza pomogłaby odbiorcy, który może być zainteresowany pewnym wycinkiem artykułu, którego tematyka pokrywa się z jego potrzebą zdobycia informacji. Proste wyszukiwanie frazami bywa wysoce nieefektywne.

LDA jest więc algorytmem, który znajduje, bądź będzie znajdywać, zastosowanie w realnych aplikacjach i produktach biznesowych, stąd też decyzja o próbie zastosowania go w **Apache Spark**.

Implementacja modelu LDA

Latent Dirichlet Allocation został zaimplementowany w **Spark**'u poprzez dwie podstawowe biblioteki:

- **MLlib** - operacje są dokonywane na **RDD** (**Resilent Distributed Dataset**)
- **ML** - model działa na strukturze DataFrame. Ponadto biblioteka posiada możliwość tworzenia pipeline'ów, które w przejrzysty sposób pozwalają na przeprowadzenie procesu budowy modelu

Tak, jak zdecydowana większość funkcji, `LDA()` można używać we wszystkich **4** oficjalnie wspieranych językach programowania, to jest w **Scali**, **Javie**, **Pythonie** (**PySpark**) i **R** (**SparkR** i **sparklyr**). W projekcie implementacja miała nastąpiła w **R** - najnowszym języku wspieranym oficjalnie. Implementacja w sparklyr jest ograniczająca, jednak z perspektywy całego procesu, w przypadku autora (jego doświadczenia z tym językiem) było to optymalne.

Początkowo należało ściągnąć i zainstalować odpowiednie paczki w **R** (**RStudio Server**). Trzeba nadmienić, że **sparklyr** jest wciąż mocno dewelopowanym projektem, stąd instalacja wersji `0.7.0` bezpośrednio z repozytorium **GitHub**'a. Kolejnym krokiem było ustalenie zmiennej środowiskowej `SPARK_HOME`, która wskazywała ścieżkę do **Spark**'a. Gdy te kroki zostały pomyślnie zakończone, można było nawiązać sesję z użyciem menedżera zasobów **YARN** i podłączyć się do **Hive**, a także zcache'ować tabelę bezpośrednio do **Spark**'a. Tak, by można było na niej efektywnie pracować. Poniżej znajduje się kod obrazujący to zadanie.

```
# Screenshot z tabelami

require(pacman)
p_load(purrr, plyr, dplyr, sparklyr, dplyr, tidyr,
       magrittr, ggplot2, ggthemes, xtable, knitr,
       devtools, rmarkdown, formatR)


if('sparklyr'%in% installed.packages()) install_github('rstudio/sparklyr')
Sys.setenv(SPARK_HOME = "/usr/lib/spark")
config <- spark_config()
spark_disconnect_all()
sc <- spark_connect(master = "yarn-client", config = config, version = '2.2.0')
tbl_cache(sc, 'posts')
posts <- tbl(sc, 'posts')
```

```
## [1] 1
```

Drugie zadanie wymagało sprawdzenie danych i ich dalsze czyszczenie. Kwerendy w **Hive** miały charakter poglądowy - właściwy model wymagał usunięcia możliwie największej ilości zbędnego tekstu (tzw. stop words). Wyczyszczenie danych opierało się o usunięcie znaczników znanych z **HTML/XML**, znaków interpunkcyjnych, cyfr i zbędnych spacji. Warto nadmienić, że `mutate` faktycznie każdą operację przerabia na kwerendę w **HQL**, stąd mimo że funkcja `regexp_replace` nie występuje w **R**, to jednak kod jak poniżej będzie działać.

```
posts %>% head(5)
```

```
## # Source:   lazy query [?? x 15]
```

```
## # Database: spark_connection
##      Id ParentId AcceptedAnswerId            CreationDate ViewCount Score
##   <int>    <chr>            <chr>                   <chr>     <int> <int>
## 1    NA                                                        NA    NA
## 2    NA                                                        NA    NA
## 3     1                          2011-04-26T18:54:45.590       418    21
## 4     2                        8 2011-04-26T18:59:07.007      6689    53
## 5     3                      109 2011-04-26T19:00:35.140     16649   127
## # ... with 9 more variables: body <chr>, LastEditorUserId <chr>,
## #   LastEditDate <chr>, LastActivityDate <chr>, Title <chr>, Tags <chr>,
## #   AnswerCount <int>, CommentCount <int>, FavouriteCount <int>
```

```
posts %>% select(body) %>% head()
```

```
## # Source:   lazy query [?? x 1]
## # Database: spark_connection
##                                                                        body
##                                                                       <chr>
## 1
## 2
## 3 &lt;p&gt;I have been singing in a (classical) choir for several years, incl
## 4 &lt;p&gt;I've heard it said that, whilst on most instruments these notes ar
## 5 &lt;p&gt;Oftentimes major keys are called &quot;happy&quot; and minor keys
## 6 &lt;p&gt;Consider four cases:&lt;/p&gt;&#xA;&#xA;&lt;ul&gt;&#xA;&lt;li&gt;A
```

```
posts_clean <- posts %>%
  mutate(body = regexp_replace(
    body,
    "[0-9]|0[0-9]+|&#xA|&amp|&xD|&lt(.*?)&gt| (&lt)|(&gt)|(&quot)|
    (;p)|(&xA)|(\\/p)|(\\/li)|(\\/ol)|[;,.\\/\\(\\)\\?!'#]", " ")
  ) %>%
  mutate(body = regexp_replace(lower(body),'[^a-z\\-]', " ")) %>%
  mutate(body = regexp_replace(lower(body),'(?<= )[a-z]{1}(?= )', " "))
```

Model **LDA** wymaga ztokenizowanych danych, co oznacza, że każdy dokument (sentencja) musi zostać przerobiona na wektor wyrazów. Tego typu operacja jest wywoływana funkcją `ft_tokenizer` bądź `ft_regex_tokenizer`.

```
posts_tokenized <- posts_clean %>%
  select(body) %>%
  mutate(body = regexp_replace(body, '[^a-z ]',' ')) %>%
  mutate(body = regexp_replace(body, '(?<= )[a-z]{1,2}(?= )',' ')) %>%
  mutate(body = lower(regexp_replace(body, ' +', " "))) %>%
  ft_regex_tokenizer('body','tokenized','[^a-z]')


posts_tokenized %<>% ft_stop_words_remover('tokenized','tokenized_clean')
to_check <- posts_tokenized %>% select(tokenized_clean) %>% head(20) %>% collect()
```

Oprócz stricte oczywistych wykreśleń z tekstu za pomocą wyrażeń regularnych, istotne jest także wyrzucenie wyrazów o neutralnym wydźwięku, co dokonywane jest poprzez `ft_stop_words_remover`. Liczba wywołań operacji, które oczyszczają tekst wynika z faktu, że **sparklyr** jest wciąż narzędziem niedoskonałym, nie w pełni zgodnym z najpełniejszym API **Spark**'a w **Scali**. Ostatnim krokiem było sprawdzenie danych, czy są one w takiej formie, jak było to oczekiwane.

Inicjacja modelu **LDA** odbywa się poprzez dane wejściowe w postaci ztokenizowanego pliku przekształconego funkcją `ft_count_vectorizer` w **sparklyr**. Wśród opcjonalnych argumentów można ustalić między innymi

wielkość słownika, który ma posłużyć do wytworzenia odpowiedniego `DataFrame`. Jest to dość arbitralna liczba, tutaj ustalono ją na **2048**. Parametry z modelu zostały ustawione tak, by powstały **4** tematy (topic'i). Liczba iteracji wynosiła **50**, zaś algorytm optymalizujący na `em` (alternatywnie mogła to być metoda `online`). Taka konfiguracja wynikała z logiki modelu i została oparta o dokumentację dla funkcji. Ważne było to by każdą wartość liczbową podać explicite jako `integer`, inaczej funkcja nie zadziała. W kodzie poniżej pierwsze wywołanie funkcji miało wytworzyć tylko i wyłącznie słownik, zwracany jako wektor słów (`character`).

```
vocab <- posts_tokenized %>%
  ft_count_vectorizer('tokenized_clean', 'count_vec',
                      vocab.size = as.integer(2048), vocabulary.only = T)

vocab %>% {tibble(x = .)} %>%
  mutate(length = nchar(x)) %>%
  group_by(length) %>% summarize(n = n()) %>%
  ggplot(aes(x = as.factor(length), y = n)) +
  geom_bar(stat = 'identity') + theme_tufte() +
  labs(title = 'Rozkład długości poszczególnych słów', x = 'Długość słowa',
       subtitle = 'Analiza dotyczyła 2048 słów zdefiniowanych jako słownik')
```

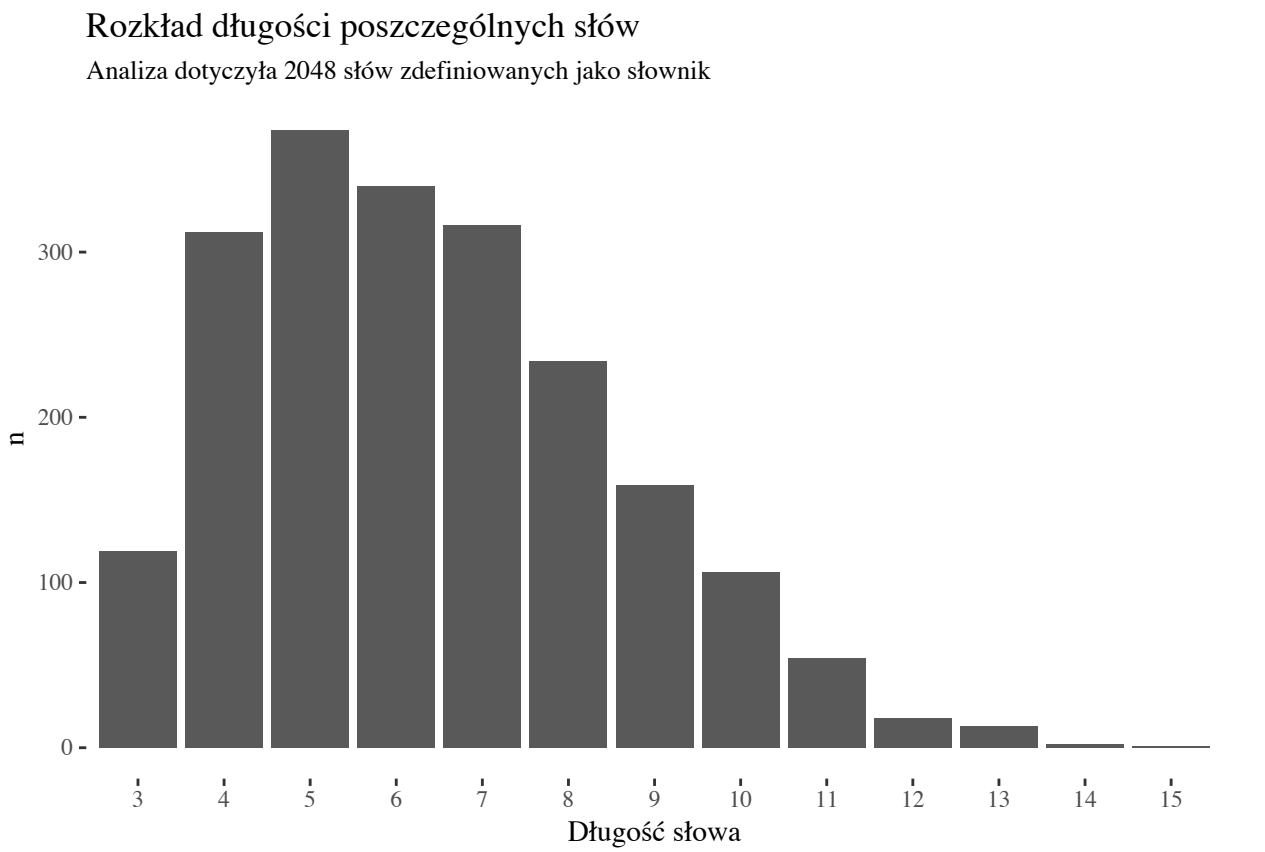

Figure 6: Rozkład długości poszczególnych słów

```
if(!'sparklyr.nested' %in% installed.packages()){
  devtools::install_github("mitre/sparklyr.nested")
}
```

```r
require(sparklyr.nested)

words_counted <- posts_tokenized %>%
  select(tokenized_clean) %>%
  sdf_explode(tokenized_clean) %>%
  group_by(tokenized_clean) %>%
  summarize(n = n()) %>%
  arrange(desc(n))
words_counted
```

```
## # Source:     lazy query [?? x 2]
## # Database:   spark_connection
## # Ordered by: desc(n)
##    tokenized_clean     n
##              <chr> <dbl>
##  1           music 23155
##  2            play 22555
##  3             one 22220
##  4           chord 21320
##  5          guitar 20185
##  6           notes 19964
##  7           would 19207
##  8            like 19082
##  9            note 18891
## 10           sound 16600
## # ... with more rows
```

Sam model **LDA** wywoływany jest poprzez `ml_lda`. Argumenty stanowią głównie parametry modelu, między innymi te omówione. Najszerszy opis znajduje się w **Scali**, tak jak poniżej.

```
scala> lda.explainParams()
params: String =
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1).
E.g. 10 means that the cache will get checkpointed every 10 iterations (default: 10)
docConcentration: Concentration parameter (commonly named "alpha") for the prior placed on
documents' distributions over topics ("theta"). (undefined)
featuresCol: features column name (default: features)
k: The number of topics (clusters) to infer. Must be > 1. (default: 10, current: 4)
keepLastCheckpoint: (For EM optimizer) If using checkpointing, this indicates whether
to keep the last checkpoint. If false, then the checkpoint will be deleted. Deleting the
checkpoint can cause failures if a data partition is lost, so set this bit with care.
(default: true) learningDecay: (For online optimizer) Learning rate,
set as an exponential decay rate. This should be between (0.5, 1.0]
to guarantee asymptotic convergence. (default: 0.51)
learningOffset: (For online optimizer) A (positive) learning parameter that downweights
early iterations. Larger values make early iterations count less. (default: 1024.0)
maxIter: maximum number of iterations (>= 0) (default: 20, current: 50)
optimizeDocConcentration: (For online optimizer only, currently) Indicates
whether the docConcentration (Dirichlet parameter for document-topic distribution)
will be optimized during training. (default: true)
optimizer: Optimizer or inference algorithm used to estimate the LDA model.
Supported: online, em (default: online, current: em)
seed: random seed (default: 1435876747)
subsamplingRate: (For online optimizer) Fraction of the corpus to be sampled and used in each
iteration of mini-batch gradient descent, in range (0, 1]. (default: 0.05)
```

```
topicConcentration: Concentration parameter (commonly named "beta" or "eta") for the
prior placed on topic' distributions over terms. (undefined)
topicDistributionCol: Output column with estimates of the topic mixture distribution for each
document (often called "theta" in the literature).
Returns a vector of zeros for an empty document. (default: topicDistribution)
```

```
model <- posts_tokenized %>%
   ft_count_vectorizer('tokenized_clean', 'count_vec', vocab.size = as.integer(2048)) %>%
   ml_lda('count_vec', k = 4, optimizer = 'em')
```

Kluczowe dane wyjściowe z perspektywy projektu składają się z wektorów umieszczonych w `data.frame`, stąd też należy je przekształcić używając wcześniej zdefiniowanego słownika, czyli zmiennej `vocab`.

```
topics_matrix <- data.frame(vocab = vocab, topics = model$topics.matrix) %>%
  as_tibble()
topics_description <- model$topics.description
topics_description_enhanced <- topics_matrix %>%
  mutate(vocab = as.character(vocab)) %>%
  left_join(collect(words_counted),c('vocab' = 'tokenized_clean'))
```

W kolejnym punkcie przekształcono dane tak, by nadawały się do analizy najbardziej charakterystycznych słów. Liczona statystyka to różnica w między największym i najmniejszym prawdopodobieństwem wystąpienia w danych dwóch skrajnych tematach.

```
p_load(purrrlyr, purrr, forcats, stringr)

topics_description_enhanced %<>% mutate(n = as.integer(n)) %>%
  mutate_if(funs(is.double(.)) , funs(./n))

topics_description_enhanced %<>% by_row(function(x) {
  max(x[,-c(1, ncol(x))]) - min(x[, -c(1, ncol(x))])
  }, .to ='diff',.collate = c('rows'))
topics_description_enhanced %<>%  arrange(desc(diff))
```

Poniżej statystyka została zwizualizowana z użyciem `ggplot2`. Jak widać skrajne słowa ciężko skojarzyć ze stricte tematem okołomuzycznym. Może to oznaczać, że faktycznie te, które są bardziej powiązane z muzyką występowały względnie po równo w każdym z tematów.

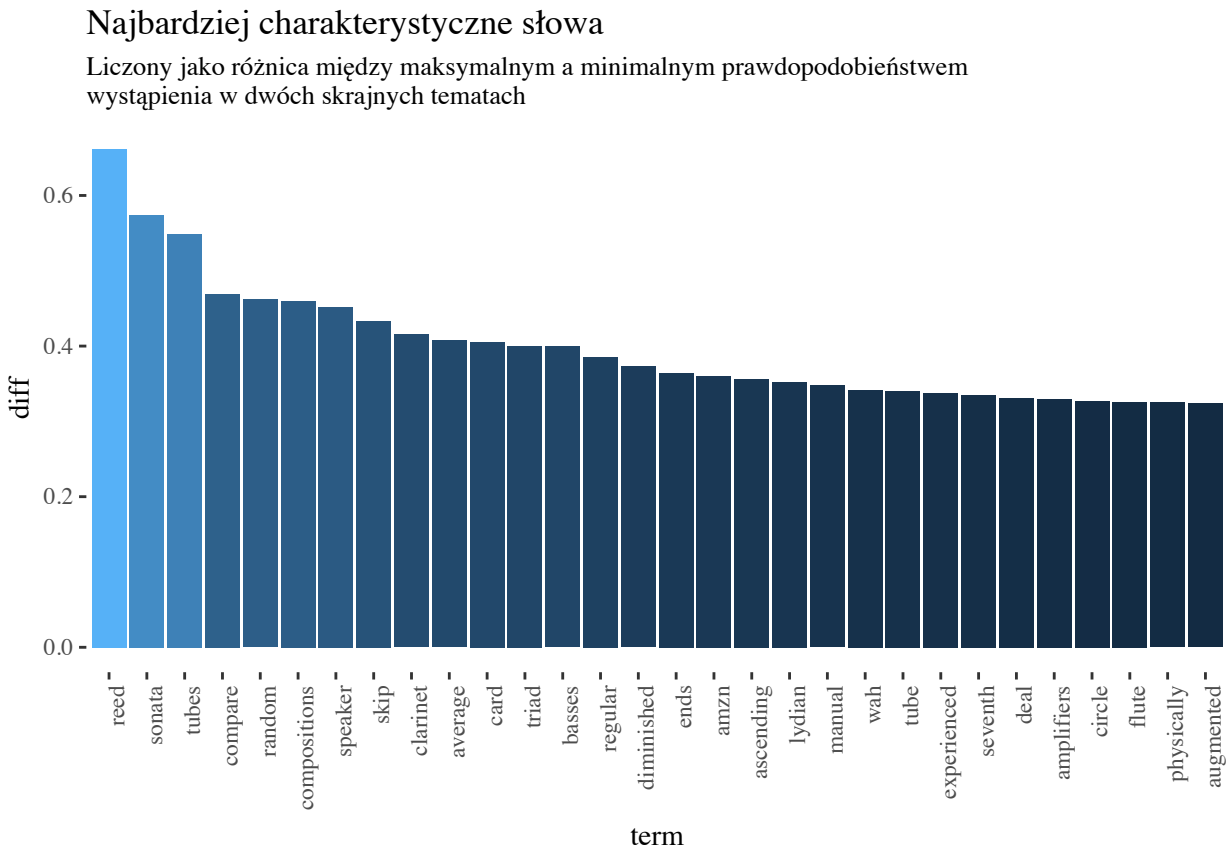

Figure 7: Najbardziej charakterystyczne słowa

```
### Words in topics
topics_description <- dmap_if(
    topics_description, is.list,
    function(x) llply(x, function(y) unlist(y))
  ) %>% tidyr::unnest(termIndices, termWeights) %>%
  mutate(terms = vocab[termIndices + 1])
```

Poniżej zostały wyszczególnione tematy i ich najczęstsze słowa. Jak widać klasteryzacja przebiegła poprawnie, gdyż dane w kolejnych tematach nieco się różnią, choć z drugiej strony należy przyznać, że nieznacznie.

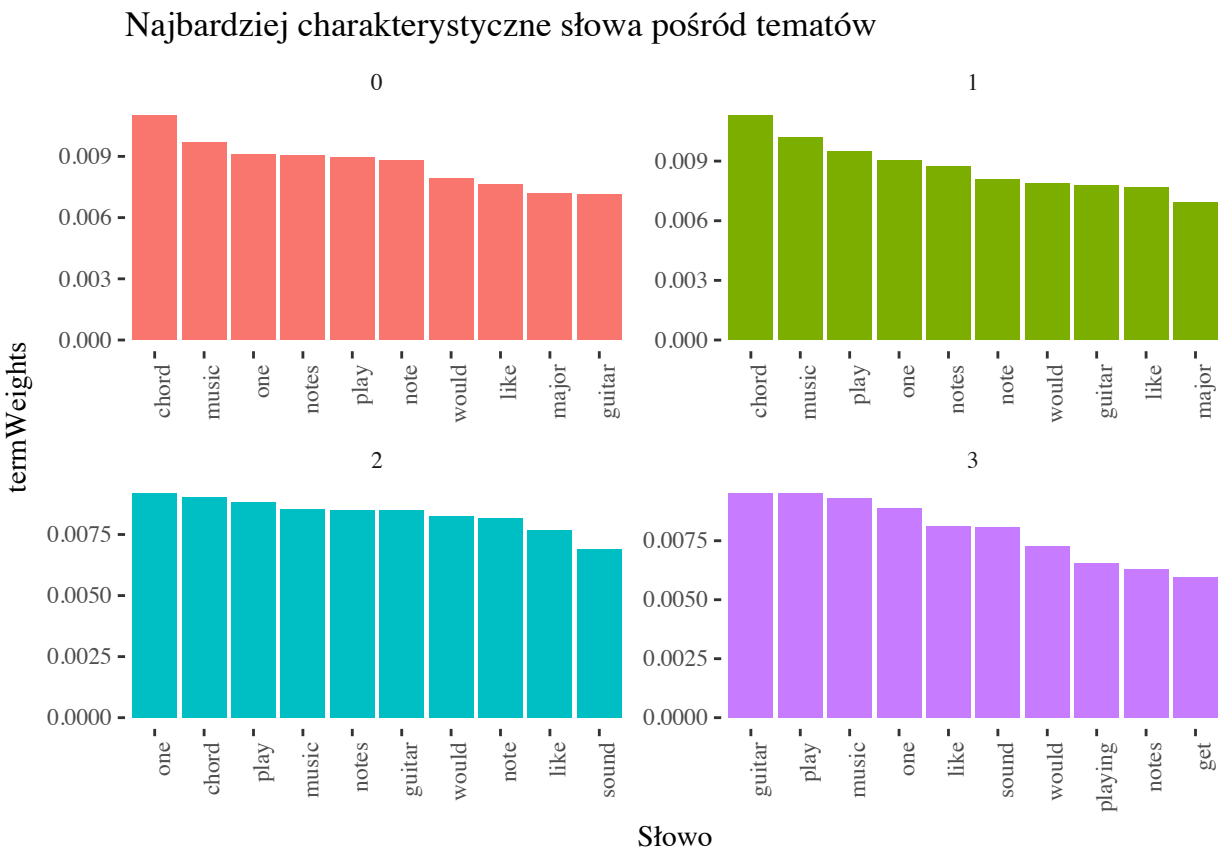

Figure 8: Najbardziej charakterystyczne słowa pośród tematów

Konkluzje

Dyskusja o problemach podczas projektu

Podczas tworzenia pracy autor borykał się z problemami natury technicznej, wynikających głównie z dynamiki projektu **Apache Spark**. Początkowo model miał być implementowany ściśle w **R**. Nie było to jednak optymalne rozwiązanie w kontekście użytego algorytmu. Dopiero obecna wersja biblioteki **sparklyr** jest w stanie zczytywać dane tekstowe do postaci `DataFrame` czy usuwać stop words za pomocą jednej funkcji. Wcześniej było to nietrywialne zadanie.

Później autor przestawił się na **Scalę**, implementacja znajduje się w załącznikach - plusem takiego rozwiązania było bardziej rozbudowane i konsystentne **API**. **sparklyr**, jak i alternatywa - **SparkR**, jest zdecydowanie mniej dojrzałe. Stąd część funkcjonalności należy zaimplementować samodzielnie (np. poprzez interfejs `invoke`), a w niektórych z nich pojawiają się błędy. Użycie **Scali** również było niebanalne, bo choć finalnie zakończyło się sukcesem (model działa), to jednak nie został wytworzony jednolity proces (od stworzenia klastra przez współpracę z **Hive** do zbudowania, interpretacji i wizualizacji modelu). Dopiero w trzecim kroku nastąpiła pomyślna próba sprowadzenia całego procesu do **R** i **RStudio Servera**, z użyciem **sparklyr 0.7.0**. Kolejnym ograniczeniem było szukanie dokumentacji do modelu **LDA**. **Spark** oferuje relatywnie ubogi zasób informacji w porównaniu do np. paczek w **R** służących do tego samego modelu i przetwarzania danych, tyle że lokalnie. Z tej właśnie przyczyny występowały trudności, by ustalić co oznaczają dane parametry, bądź jak działa wykorzystany algorytm. Chwilami problematyczny był też klaster, który autor konfigurował kilkukrotnie - z perspektywy czasu lepszym rozwiązaniem mogłoby się okazać wykorzystanie **docker**'a.

Podsumowanie

W projekcie został poruszany temat inżynierii danych. Jej celem było przetwarzanie danych oraz pomyślna implementacja modelu text miningowego w rozproszonym środowisku obliczeniowym. To założenie zostało zrealizowane. Użyto infrastruktury chmurowej **Amazon Elastic Map Reduce**, na który zostały wrzucone dane o postach z serwisu **Stack Exchange**. Ponadto dokonano ich eksploracji za pomocą kwerend **HQL**, które każde zapytanie przetwarzają na zadanie na klastrze. Końcowym elementem były zadania w **Sparku**, których rezultatem miał być model **Latent Dirichlet Allocation**. Po dokonaniu odpowiednich przeliczeń, został on zinterpretowany za pomocą języka **R**.

Potencjalne dalsze kierunki rozwoju podobnych projektów

W przyszłości podobny projekt mógłby być rozwijany przynajmniej w kilku kierunkach. Po pierwsze - pod kątem większej ilości danych. Choć użyte narzędzia służyły do dużych zbiorów danych, to jednak pierwotny plik miał jedynie nieco powyżej **35 mb**. Zaimplementowany model **LDA** miałby z pewnością większą wartość, gdyby ramka danych (**DataFrame**) w **Sparku** była obszerniejsza. Kolejna kwestia to sam model - ten został stworzony bardziej pod kątem inżynierii danych aniżeli data science. Celem było pokazanie, że faktycznie model, o którym mowa działa i jest w stanie wygenerować relatywnie sensowne wyniki. Jego tuning jednak został jedynie minimalnie poruszony w pracy. Trzeci aspekt to stworzenie gotowego produktu (np. w formie pliku **.jar**, który umieszcza się na klastrze, bez problematycznej konfiguracji), który mógłby zostać zaimplementowany w biznesie. Projekt opierał się o założenie, że dane są statyczne, a cały opisany proces ma zadziałać najwyżej kilkukrotnie. W praktyce jednak dane mogą być streamowane, np. za pomocą **Apache Kafka**, co oznaczałoby wielokrotne wywołanie funkcji modelującej, czy zawierać błędy - z tego względu potencjalnie należałoby zwiększyć elastyczność rozwiązania, a także zautomatyzować je. Tu należy wspomnieć o ograniczeniach użytych narzędzi - środowisko, w którym autor projektu czuł się najpewniej (**R**) pod kątem **Apache Spark**'a wciąż jest niedoskonałe i w kontekście zaadresowanych problemów często nieoptymalne. Być może przyszłe zmiany w bibliotekach **R**'owych doprowadzi do znacznego uproszczenia i poprawienia analogicznych projektów. W tym momencie jest to jednak zadanie nietrywialne.

Spis rysunków

Spis rysunków

Bibliografia

Alexander, Alvin. 2013. “Scala Cookbook.” O’Reilly Media.

Amazon Inc. n.d. “Getting Started: Analyzing Big Data with Amazon Emr.” http://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-gs.html.

Blei, David M. 2012. “Probabilistic Topic Models. Communications of the Acm.” Department of Computer Science. Princeton University.

Chen, Edwin. n.d. “Introduction to Latent Dirichlet Allocation.” http://blog.echen.me/2011/08/22/introduction-to-latent-dirichlet-allocation.

databricks. 2017. “Topic Modeling with Latent Dirichlet Allocation.” https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3741049972324885/3783546674231782/4413065072037724/latest.html.

David M. Blei, Michael I. Jordan, Andrew Ng. 2003. “Latent Dirichlet Allocation.” *Journal of Machine Learning Research 3* 3: 993–1022. http://tidytextmining.com.

Holden Karau, Patrick Wendell, Andy Konwinski. 2015. “Learning Spark.” O’Reilly Media.

Julia Silge, David Robinson. 2017. “Text Mining with R: A Tidy Approach.” http://tidytextmining.com/.

Labs, Zero Gravity. 2017. “LDA Topic Modeling in Spark Mllib.” https://zerogravitylabs.ca/lda-topic-modeling-spark-mllib.

RStudio Inc. n.d. “R Interface to Apache Spark.” https://spark.rstudio.com.

The Apache Software Foundation. 2017. “Apache Spark Documentation.” https://spark.apache.org/documentation.html.