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** w dalszej części rozdziału zostały wymodelowane te właśnie dwa problemy.

W kontekście pracy ważnym pytaniem było to, dlaczego taki algorytm jest ważny i do czego może się przydać? Intuicyjnym wydaje się być, np. automatyzacja procesu tagowania postów. Okazuje się, że aplikacja takiego narzędzia jest dużo szersza. Wedle Blei (2012) 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 twierdzi, że algorytmy do modelowania tematów mogą posłużyć do znacznie lepiej rozwiniętego mechanizmu, w którym eksploracja dokumentów byłaby oparta o motywy (topics), które można generalizować bądź uszczegóławiać. Przykładowo mając dokument o bioinformatyce, można założyć, że zawiera on kilka motywów przewodnich. Blei wskazuje przykładowo na **genetykę**, **ewolucję**, **chorobę** i **komputery**, choć jest to wybór arbitralny. Algorytm LDA 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.

LDA jest 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 przypadku tego projektu pierwotnie implementacja miała nastąpić w **R**, w najnowszym języku wspieranym oficjalnie. Okazało się to być jednak bardzo zawodne i ograniczone rozwiązanie. Przykładem jest `ft_stop_words_remover()` w paczce **sparklyr (0.7.0)**. Funkcja wywołuje metodę `StopWordsRemover` z biblioteki **ML** w **Scali.**. Służy ona do wyrzucania zbędnych słów (z perspektywy text miningu za takie można uznać np. **the**, **i**, **have**) ze ztokenizowanego tekstu w kolumnie **DataFrame**. Implementacja w sparklyr jest o tyle ograniczająca, że nie można w niej edytować listy wyrazów, a jedynie użyć domyślny zasób w języku angielskim. Z między innymi tego powodu ostateczny model został wywołany w Scali, jednak logika została oparta o przetwarzanie danych w Spark za pomocą **sparklyr** w **R**.

Na początku 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')
```

```
# 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=spark_install_dir())
config <- spark_config()
spark_disconnect_all()
sc <- spark_connect(master = "local[*]", config = config, version = '2.1.0')
posts <-
  spark_read_text(sc, 'posts', path = paste0(file.path(getwd(),'music_datasets/Posts.xml')))
posts <- tbl(sc, 'posts')
posts %>%
```

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)
posts %>% select(body) %>% head()
posts_clean <- posts %>%
  mutate(body = regexp_replace(body,
                               "[0-9]|0[0-9]+|&amp|&xD|&lt(.*?)&gt| (&lt)|(&gt)|(&quot)|
                               (;p)|(&xA)|(\\/p)|(\\/li)|(\\/ol)|[;,.\\/]",
                               " ")) %>%
```

```
  mutate(body = regexp_replace(body,'(?<= )[A-Z]{1}(?= )'," ")) %>%
  mutate(body = lower(regexp_replace(body,' +'," ")))
```

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) %>%
  ft_regex_tokenizer('body', 'tokenized', '((?<= )[a-z]{1}(?= ))|[^a-z]')

posts_tokenized %<>% ft_stop_words_remover('tokenized','tokenized_clean')
to_check <- posts_tokenized %>% select(tokenized_clean) %>% head(10) %>% 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)


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

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
#topics_matrix[1,-1] %>%sum()
```

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')
```

Strukturę otrzymanego modelu można sprawdzić poprzez `str`.

```
model_str <- str(model)
```

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'))
```

```
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))

topics_description_enhanced%>%filter(n > 200) %>% arrange(desc(diff))   %>%
  top_n(30) %>%
  ggplot(aes(x=fct_reorder(vocab, diff, .desc = T), y= diff, fill = diff)) +
  geom_bar(stat='identity') +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  theme_tufte() + guides(fill= FALSE)
```

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


### 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])

topics_description %>%
  mutate(index = seq(1, 40),
         label = paste('Topic', topic)) %>%
  ggplot(aes(x = fct_reorder(as.factor(paste0(terms,'-topic.', topic)), index),
             y= termWeights, fill = as.factor(topic)))+
  geom_bar(stat= 'identity') +
  scale_x_discrete(labels = function(x) str_replace_all(x,'-.*',''))+
  facet_wrap(~as.factor(topic), scales = 'free') +
  guides(fill = F) +
  theme_tufte()
```