

# Projekt z eksploracji danych

Filip Chudy, Marcin Grzywaczewski

19 lutego 2016

## Streszczenie

## 1 Opis danych i badanego zagadnienia

### 1.1 Dane

W projekcie wykorzystany został Million Songs Dataset (Last.fm). Składa się on z informacji na temat miliona piosenek. Każda piosenka reprezentowana jest jako JSON z następującymi elementami:

- `track_id` – string, unikalny identyfikator piosenki
- `similars` – lista par (`track_id`, float) określająca podobieństwa między piosenkami wraz z ich ważnością (0-1)
- `tags` – lista par (string, float) określająca tagi przypisane utworowi wraz z ich ważnością (0-100)
- `artist` – string, nazwa artysty
- `title` – string, tytuł piosenki
- `timestamp` – string, czas zebrania danych

Z naszego punktu widzenia pole **timestamp** jest nieistotne. Pola **artist** i **title** służą jedynie do prezentacji wyników. Liczba utworów podobnych i podanych tagów jest zmienna, w szczególności mogą się zdarzyć takie utwory, które nie mają podanych tagów. Liczba piosenek podanych w `similars` nie jest zbyt duża, co obrazuje histogram.

### 1.2 Cel

Celem projektu jest stworzenie systemu rekomendującego działającego na powyższych danych. Zadanie jest trudne ze względu na rzadkość i niekompletność macierzy tagów i podobieństwa.

## 2 Użyte metody

### 2.1 Wstęp

Dane są rzadkie, więc wektoryzacja piosenek na podstawie tagów nie wchodziła w grę. Struktura **similars** przywodzi na myśl graf, dzięki czemu można uzupełnić macierz podobieństwa choćby za pomocą obliczenia przechodniego domknięcia. Ponieważ w grafach naturalną miarą jest odległość, przeszliśmy z opisu opartego na podobieństwach na odległości według wzoru

$$distance(x, y) = \frac{1}{similarity(x, y)} \quad (1)$$

Metody wyznaczania przechodniego domknięcia grafu mają złożoność sześcienną względem liczby wierzchołków w grafie, więc aby którąś zastosować, należy zmniejszyć graf.

Pomysł na redukcję grafu wywodzi się z grupowania hierarchicznego. Gęste skupiska punktów można reprezentować przy pomocy pojedynczych superwierzchołków, w których rozwiązywany będzie podobny problem dla tego skupiska. Na tak wyznaczonych grafach hierarchicznych można już wyznaczyć przechodnie domknięcia.

### 2.2 Algorytm hierarchiczny

Algorytm kompresujący graf jest bardzo podobny w działaniu do algorytmów grupowania hierarchicznego. Dane będą reprezentowane przez graf. Jego wierzchołki odpowiadają piosenkom lub ich zbiorom, a krawędzie – odległościom między nimi. W wierzchołkach grafu będzie przechowywana mniejsza wersja rozwiązywanego problemu.

Graf  $G_1 = (V_1, E_1)$  jest przodkiem grafu  $G_2 = (V_2, E_2)$ , jeżeli istnieje taka krawędź  $e(v_i, v_j) \in E_1$ , że  $V_1 \setminus \{v_i, v_j\} \subseteq V_2$ ,  $V_2 \setminus V_1 = \{v_k\}$  i  $inner\_vertices(v_k) = inner\_vertices(v_i) \cup inner\_vertices(v_j)$ . Dodatkowo  $(\forall w \in \{v : e(v_i, v) \in E_1 \vee e(v_j, v) \in E_1\}) e(v_k, w) \in E_2 \wedge |e(v_k, w)| = \min\{|e(u, w)| : u \in \{v_i, v_j\}\}$ .  $G_2$  nazywamy potomkiem grafu  $G_1$ , jeśli  $G_1$  jest przodkiem  $G_2$ .

Innymi słowy dla każdej krawędzi  $e$  w grafie  $G_1$  istnieje potomek  $G_2$  grafu  $G_1$  powstały przez scalenie wierzchołków łączonych przez  $e$ . Oczywiście każda krawędź w grafie jednoznacznie wyznacza odpowiadającego jej potomka.

Każdemu grafowi przypisujemy koszt według pewnej funkcji błędu  $F$ . W każdej iteracji wybieramy pewnego potomka  $G_2$  grafu  $G_1$  pod warunkiem, że  $F(G_2) < F(G_1)$ . Ponieważ wyznaczanie wartości funkcji  $F$  dla każdego potomka grafu  $G_1$  byłoby kosztowne obliczeniowo, zastosowaliśmy wybór heurystyczny.

Wybieramy wierzchołek grafu o najmniejszym  $inner\_vertices$ . Spośród krawędzi incydentnych do tego wierzchołka wybieramy najkrótszą taką krawędź  $e(v_i, v_j)$ , że graf  $G_{ij}$  powstały przez scalenie  $v_i$  i  $v_j$  ma niższą funkcję błędu niż jego przodek. Jeśli żadna z krawędzi incydentnych do wierzchołka nie zmniejsza funkcji kosztu, wybieramy kolejny wierzchołek według liczności  $inner\_vertices$ .

Wybór kończymy, jeśli każdy potomek grafu ma nie mniejszy koszt niż jego przodek lub przekroczymy minimalną wielkość grafu (parametr algorytmu).

### 2.3 Funkcja kosztu

Funkcja kosztu składa się z trzech czynników: WCD (rozmiary wewnętrzne superwierzchołków), SVP (penalizacja superwierzchołków jednoelementowych), BCD (odległości między superwierzchołkami).

$$WCD(G) = \sum_{c \in C} \max_{e \in E(\text{graph}(c))} |e| \quad (2)$$

$$SVP(G) = |\{c \in C : \text{size}(c) = 1\}| \times \max_{e \in E(C)} |e| \quad (3)$$

$$BCD(G) = \sum_{c_1 \in C} \sum_{c_2 \in C} |e(c_1, c_2)| \quad (4)$$

Sama funkcja kosztu jest iloczynem tych trzech czynników:

$$F(G) = (WCD(G) + SVP(G)) \times BCD(G) \quad (5)$$

Obliczanie takiej funkcji jest kosztowne, jednak jeśli znamy wartości wszystkich czynników składających się na funkcję kosztu przodka, możemy na ich podstawie efektywnie wyznaczyć czynniki dla potomka.

Niech  $G_2$  będzie potomkiem  $G_1$  łączącym  $c_i, c_j \in G_1$  w  $c_k \in G_2$ . Można zauważyć, że:

$$\begin{aligned} \Delta WCD(G_2) &= WCD(G_2) - WCD(G_1) \\ &= \sum_{c \in C(G_2)} \max_{e \in E(\text{graph}(c))} |e| - \sum_{c \in C(G_1)} \max_{e \in E(\text{graph}(c))} |e| \\ &= \max_{e \in E(\text{graph}(v_k))} |e| - \max_{e \in E(\text{graph}(v_i))} |e| - \max_{e \in E(\text{graph}(v_j))} |e| \end{aligned} \quad (6)$$

$$\begin{aligned} \Delta SVP(G_2) &= SVP(G_2) - SVP(G_1) \\ &= |\{c \in \{v_i, v_j\} : \text{size}(c) = 1\}| \times \max_{e \in E(G)} |e| \end{aligned} \quad (7)$$

$$\begin{aligned} \Delta BCD(G_2) &= BCD(G_2) - BCD(G_1) \\ &= \sum_{c_1 \in C_2} \sum_{c_2 \in C_2} |e(c_1, c_2)| - \sum_{c_1 \in C_1} \sum_{c_2 \in C_1} |e(c_1, c_2)| \\ &= \sum_{c \in C_2} |e(v_k, c)| - \sum_{c \in C_1} |e(v_i, c)| - \sum_{c \in C_1} |e(v_j, c)| + |e(v_i, v_j)| \end{aligned} \quad (8)$$

Dzięki temu obliczanie funkcji kosztu dla kolejnych potomków grafu jest znacznie szybsze.

### 2.4 Ogląd globalny

Początkowy graf dzielimy na spójne składowe algorytmem BFS. W każdej spójnej składowej wywołujemy opisany powyżej algorytm hierarchiczny. Grafy na różnych poziomach hierarchii są na tyle małe, żeby dopełnić je do klikli za pomocą algorytmu Floyd-Warshalla. Na tak dopełnionych grafach uruchomiony zostanie system rekomendujący.

## 2.5 System rekomendujący

System rekomendujący jako argument przyjmie piosenki **polubione** przez użytkownika. System odnajduje zadane piosenki w grafie i zapisuje sobie ich sąsiadów. Jeśli sąsiadem jest superwierzchołek, do listy sąsiedztwa zostaje dodana cała jego zawartość (z odległością równą odległości między superwierzchołkami). Mamy więc listę par piosenka-odległość (piosenka może na liście występować wielokrotnie, o tym za chwilę). Na podstawie takiej listy zbudujemy dla każdej piosenki występującej na liście wartość liczbową mówiącą, jak dobrze nadaje się ona do rekomendacji. Niech  $distances_s$  oznacza [multi]zbiór odległości dla piosenki  $s$ .

$$weight(s) = \sum_{d \in distances(s)} d^{-1} \quad (9)$$

W ten sposób dla każdej piosenki wyznaczamy odpowiadającą jej wartość liczbową. Rekomendacji dokonujemy losując bez zwracania odpowiednio wiele piosenek z zadanego przez  $weight$  rozkładu dyskretnego z wagami.

## 3 Implementacja

Implementacja została podzielona na dwie części:

- część przygotowania danych – na tym etapie następuje normalizacja danych, utworzenie lasu z podanych wierzchołków, zastosowanie algorytmu hierarchicznego oraz eksport powstałego wyniku do plików;
- część rekomendacji – na tym etapie korzystamy z danych powstałych podczas poprzedniej fazy, aby dla zadanego przez użytkownika zbioru utworów otrzymanym wynikiem były utwory zarekomendowane przez system.

Wyzwaniem okazała się złożoność pamięciowa. Istotnie trudnym problemem implementacyjnym okazało się opracowanie efektywnej procedury łączenia dwóch podgrafów w jeden – jest to istotą naszego algorytmu hierarchicznego.

### 3.1 Założenia na temat danych

Przy analizie danych niezbędne okazały się następujące założenia na ich temat:

- relacja podobieństwa danych powinna być symetryczna, więc dane zostały odpowiednio zmodyfikowane, aby założenie to spełnić,
- jeżeli podobieństwo utworu  $A$  z  $B$  wynosi  $x$ , podobieństwo utworu  $B$  z  $A$  również powinno wynosić  $x$ ,
- grafy należące do lasu będące izolowanym wierzchołkiem nie niosą istotnej informacji dla systemu rekomendującego – zostały one odrzucone,
- jeżeli w grafie istnieje krawędź  $(v, w)$  to  $v$  oraz  $w$  istnieje w zbiorze danych.

Aby spełnić te założenia w implementacji uwzględniono odpowiednie procedury modyfikujące wejściowy zbiór danych.

### 3.2 Poprawa symetryczności miary podobieństwa w zbiorze danych

W celu poprawienia krawędzi zastosowano procedurę działającą w następujący sposób:

- dla każdego wierzchołka  $v$  występującego w grafie znajdujemy wszystkie krawędzie  $(v, w)$ ,
- dla każdej krawędzi  $(v, w)$  sprawdzamy jej miarę podobieństwa  $p_{vw}$ ; jeżeli nie istnieje w grafie krawędź  $(w, v)$ , tworzymy ją nadając jej miarę podobieństwa  $p_{vw}$ ,
- jeżeli krawędź  $(w, v)$  istnieje, sprawdzamy jej miarę podobieństwa  $p_{wv}$ ; ustalamy miarę podobieństwa obu krawędzi  $(v, w)$  oraz  $(w, v)$  na  $p_m$ , gdzie  $p_m$  jest zdefiniowane jako  $\max\{p_{vw}, p_{wv}\}$ ,

Powyższa procedura jest zaimplementowana jako `fix_similarity_symmetry` w module `load_data` implementacji. Jej złożoność obliczeniowa to  $\mathcal{O}(|V| + |E|)$ . Jej wynikiem jest zbiór danych, który spełnia pierwsze i drugie założenie na temat danych.

### 3.3 Usunięcie wierzchołków izolowanych krawędzi zewnętrznych zbioru danych

Po przeprowadzeniu procedury poprawy symetryczności krawędzi wierzchołki i krawędzie nieprawidłowe zostają usunięte z grafu. Za nieprawidłowe wierzchołki uważamy wierzchołki izolowane (tj. grafy w powstałym lesie zawierające dokładnie jeden wierzchołek). Za nieprawidłowe krawędzie uważamy krawędzie, których końce nie zawierają się w zbiorze danych.

Sposób działania procedury podobny do działania procedury poprawy symetryczności. Przechodzimy wszystkie wierzchołki  $v$  występujące w grafie. Jeżeli liczba krawędzi wychodzących z tego wierzchołka to 0, usuwamy taki wierzchołek. W przeciwnym wypadku przeglądamy wszystkie krawędzie  $(v, w)$ . Jeżeli wierzchołek  $w$  nie istnieje w grafie, krawędź ta zostaje usunięta.

Odpowiadanie na pytanie „zawierania wierzchołka” jest zrealizowane w czasie stałym (struktura słownika). Całkowita złożoność procedury to  $\mathcal{O}(|V| + |E|)$ . Wynikiem działania procedury jest zbiór danych spełniający trzecie i czwarte założenie dotyczące danych.

Procedura jest zaimplementowana jako `purge_invalid_vertices` w module `load_data`.

### 3.4 Poprawa złożoności pamięciowej: obcięcie danych

W celu zaoszczędzenia pamięci wczytany zbiór danych został pozbawiony struktury `tags` zawierającej tagi utworów. Identyfikatory utworów, będące łańcuchami znaków, zostały odwzorowane na zbiór  $\{0, 1, \dots, n\}$ . Wygenerowana mapa oraz mapa odwrotna tego odwzorowania jest zapisywana jako plik.

Oba te kroki są częścią procedury `load_data` w module `load_data`.

### 3.5 Algorytm wyznaczania lasu ze zbioru danych

W celu wyznaczenia lasu ze zbioru danych zastosowany został klasyczny algorytm wyznaczania spójnych składowych. Wybrany sposobem przeglądania grafu został algorytm BFS. Całkowita złożoność wyznaczania obliczeniowa lasu to  $\mathcal{O}(|V| + |E|)$ . Jako wejście procedura akceptuje zbiór danych w postaci słownika odwzorowującego identyfikatory utworów w postaci liczb na obcięte dane zgodne z formatem plików oryginalnego zbioru danych. Wynikiem jest kolekcja obiektów typu `Graph` realizujących logikę redukowania grafów i wyznaczania macierzy odległości.

Algorytm ten został zrealizowany jako obiekt `Forest`. Dostarczona została też procedura generująca reprezentację plikową już policzonego i zredukowanego lasu.

### 3.6 Reprezentacja grafu

Reprezentacja grafu musiała spełniać następujące warunki: w sposób efektywny obliczeniowo obsługiwać operacje wymagane przez algorytm redukcji, będąc jednocześnie odpowiednio wydajną pamięciowo. **Każdy wierzchołek w grafie jest podgrafem.**

Ostatecznym rozwiązaniem okazała się następująca implementacja:

- w celu szybkiego dostępu do podgrafów grafu utrzymujemy słownik `subgraphs` będący odwzorowaniem identyfikatorów grafu (liczby  $\{-1, -2, \dots, -n\}$ ) na obiekty podgrafów,
- w celu szybkiego przeglądania krawędzi w grafie stosujemy strukturę `edges` będącą dwuwymiarowym słownikiem odwzorowującym parę identyfikatorów grafów  $(v, w)$  na posortowaną rosnąco względem odległości listę krotek w postaci  $(a, b, p)$  gdzie  $a$  i  $b$  to krawędź pomiędzy rzeczywistymi wierzchołkami, a  $p$  to miara odległości (odwrotność miary podobieństwa). Z racji dwuwymiarowości możemy szybko odpowiadać na pytania typu *wszystkie krawędzie incydentne z  $v$*  kosztem dwukrotnego zwiększenia złożoności pamięciowej dla kluczy,
- w celu wydajnego przeprowadzania redukcji utrzymujemy posortowany zbiór identyfikatorów podgrafów, zrealizowany jako drzewo czerwono-czarne; kluczem sortowania jest wielkość grafu – preferujemy łączyć mniejsze grafy w większe.

### 3.7 Redukcja grafu

W celu redukowania grafu implementujemy algorytm opisany w poprzedniej sekcji raportu. Procedury `loss_after_merge` oraz `loss` są implementacjami odpowiednio: wartości funkcji błędu i wartości funkcji błędu po połączeniu pewnych podgrafów  $v$  i  $w$ .

W procedurze `reduce` realizujemy następujący algorytm.

- dopóki znajdujemy wierzchołek  $v$  mogący poprawić graf i liczba wierzchołków grafu jest większa niż pewna stała  $M$ :
  - przeglądamy posortowaną rosnąco względem liczby krawędzi w podgrafie listę wierzchołków  $v$ :

- \* przeglądamy listę krawędzi  $(v, w)$ . Obliczamy funkcję błędu i funkcję błędu po scaleniu wierzchołków  $v$  i  $w$ ,
- \* jeżeli scalenie  $(v, w)$  zmniejsza funkcję błędu, scalamy te wierzchołki w jeden podgraf. Oznaczamy iterację jako "udaną",
- \* jeżeli nie znajdujemy żadnej pary  $(v, w)$  polepszającej funkcję błędu, przerywamy iterację.

### 3.8 Łączenie podgrafów

Łączenie podgrafów  $(v, w)$  polega na stworzeniu nowego wierzchołka  $G$  będącego nowym podgrafen aktualnie przetwarzanego grafu. Do tego podgrafu przenoszone są:

- wszystkie wierzchołki należące do  $v$  i  $w$ ,
- wszystkie krawędzie od  $v$  do  $w$  (mamy tu na myśli listę połączeń realnych wierzchołków z wagami),
- wszystkie krawędzie wewnętrzne w  $v$  i w  $w$ .

Poza tym należy przeprowadzić następujące kroki na poziomie aktualnie przetwarzanego grafu:

- usunięcie wierzchołków  $v$  i  $w$ ,
- usunięcie krawędzi incydentnych z  $v$  i  $w$  (łączymy je z nowym grafem  $G$ ),
- usunięcie krawędzi pomiędzy  $v$  i  $w$ ,
- przekierowanie wszystkich krawędzi  $(v, x)$  i  $(w, y)$  na krawędzie w postaci  $(G, x)$ ,  $(G, y)$ , uwzględniając krawędzie odwrotne.

W celu łączenia dwóch posortowanych list będących wartością dwuwymiarowego słownika `edges` wprowadzona została metoda `merge_sorted_lists`. Operację łączenia podgrafów realizuje metoda `merge_subgraphs` obiektu `Graph`.

### 3.9 Obliczanie macierzy odległości

Efektom redukcji grafu jest struktura, której liczba wierzchołków na pojedynczym poziomie jest odpowiednio zredukowana. Dzięki temu możemy policzyć macierz odległości. Macierz odległości liczona jest klasycznym algorytmem Floyda-Warshalla, działającym w czasie sześciennym od liczby wierzchołków w grafie.

Procedura ta jest zaimplementowana w obiekcie `Graph` jako `distance_matrix`.

### 3.10 Optymalizacja pamięciowa: Usuwanie nadmiarowych krawędzi

Po zredukowaniu grafu przechwytywanie listy krawędzi z rzeczywistych wierzchołków i ich wag przestaje być potrzebne. Procedura `simplify_edges` redukuje listę do jedynego interesującego nas w przypadku macierzy odległości elementu - krotki z najmniejszą odległością.

### 3.11 Format wyjściowy

Wyjściowym formatem dla grafów jest plik posiadający wewnątrz siebie listę identyfikatorów wierzchołków oraz macierz odległości między nimi. Procedura generowania tych plików (rekurencyjna) jest zrealizowana w obiekcie `Graph` jako `pickle_graph`.

### 3.12 System rekomendujący

System rekomendujący realizowany jest przez klasę `Recommender`. Korzysta ona z wcześniej wyliczonych grafów leniwie odczytując je z dysku. Do samej rekomendacji służy metoda `recommend`. Przyjmuje ona listę utworów polubionych przez użytkownika i liczbę piosenek, którą system powinien dostarczyć. Dla każdego polubionego utworu (i grafów leżących wyżej w hierarchii, które go zawierają) wyznaczamy listę kandydatów do rekomendacji i ich wagi. Następnie losujemy bez zwracania potrzebną liczbę elementów.

## 4 Wyniki

Przedstawione tutaj wyniki są wytworzone poprzez analizę mniejszego zbioru danych (*subset*) zawierającego około 9500 utworów muzycznych. Niestety nie ma matematycznej miary poprawności rekomendacji. Niemniej jednak po ręcznym przeanalizowaniu otrzymanych wyników uznajemy efekty za zadowalające. System rekomenduje inne utwory tego samego artysty oraz artystów tworzących swoje albumy w podobnych czasach i stylach muzycznych. Nie jest to wystarczające do upewnienia się co do jakości systemu. Stąd też zostało zastosowane inne rozwiązanie.

### 4.1 Przykłady

Prezentujemy kilka przykładów symulowanych przez losowanie danych wejściowych i powiązanych z nimi rekomendacji:

Zapytany o piosenkę Scott Wesley Brown – All Rise system podał:

- Mötley Crüe – Merry-Go-Round
- Ricky Martin – She’s All I Ever Had
- Erja Lyytinen – Soul Of A Man
- Santana – Rock Me
- Scott Wesley Brown – I Stand Here Forgiven

Zapytany o piosenkę Jimmy Page – Liquid Mercury system podał:

- Incubus – Just a Phase
- Beastie Boys – Flute Loop (2009 Digital Remaster)
- Yami Bolo – The Wonders
- The Police – Voices Inside My Head
- Otis Taylor – Round and Round



## 4.2 Analiza jakości rekomendacji

Z racji braku danych na temat wkładu użytkowników (odsluchań i polubień) utworów klasyczne metody sprawdzania systemu rekomendacji nie mogły zostać tutaj użyte. Jedyne dostępne nam dane pozwalające ocenić rozwiązanie to tagi utworów. Przypomnijmy - tagi prezentowane są w postaci listy krotek  $(n, p)$  gdzie  $n$  jest etykietą danego tagu, a  $p$  - wyrażoną od 1 do 100 "pewnością" należenia danego utworu do podanego tagu. Nasze rozwiązanie problemu walidacji rekomendacji oparliśmy na następującym pomysśle:

- wytwarzamy *sztucznego użytkownika* według następującej procedury: losujemy pewien utwór ze zbioru danych, nazwijmy go  $T$ . W kolejnym kroku algorytmu prosimy o 5 rekomendacji dla utworu  $T$ , po czym otrzymane rekomendacje dodajemy do zbioru 'wejściowego' dla kolejnych prób rekomendacji. W przypadku, gdy system nie potrafi zarekomendować pięciu utworów dla danego  $T$ , losujemy kolejny utwór, tworząc zbiór  $\{T, T2\}$  i ponawiamy próbę. Proces kończymy, gdy uda nam się zgromadzić 100 utworów.
- dla wytworzonego *sztucznego użytkownika* reprezentowanego przez zbiór utworów  $\{t_1, t_2, \dots, t_{100}\}$  prosimy o kolejne 5 rekomendowanych utworów.
- Przez  $C$  oznaczmy zbiór utworów stanowiący sztucznego użytkownika. Przez  $R$  zaś - zbiór utworów zarekomendowanych przez tego użytkownika. Na zbiorach  $C$  i  $R$  wykonujemy procedurę *gromadzenia tagów*.

## 4.3 Gromadzenie tagów

Dla danego zbioru  $X$  utworów muzycznych gromadzenie tagów wytwarza pewien agregat  $aggregated\_tags(X)$  zdefiniowany następująco:

- jeżeli dla każdego  $t \in X$  jest oznaczone pewnym tagiem o etykiecie  $e$ , agregat zawiera tag o tej etykiecie.
- dla każdego tagu o etykiecie  $e$  w agregacie, jego "pewność" wyrażona jest jako  $\frac{\sum_{i=1}^n p_i}{N}$  gdzie  $n$  to liczba utworów oznaczonych tagiem z etykietą  $e$ ,  $N$  - liczba utworów muzycznych o niepustej liście tagów, zaś  $p_i$  jest "pewnością" tagu o etykiecie  $e$  dla  $i$ -tego utworu oznaczonego tagiem z etykietą  $e$ .

Tak przygotowany agregat tagów jest wykorzystywany w dalszej analizie poprawności systemu.

## 4.4 Funkcja błędu

Naszym założeniem przy ocenie systemu rekomendującego jest to, że dla dowolnego utworu muzycznego  $t$  jego rekomendacja  $t'$  jest oznaczana przez podobne tagi. Aby "mierzyć" to podobieństwo wykorzystaliśmy dwie funkcje błędu:

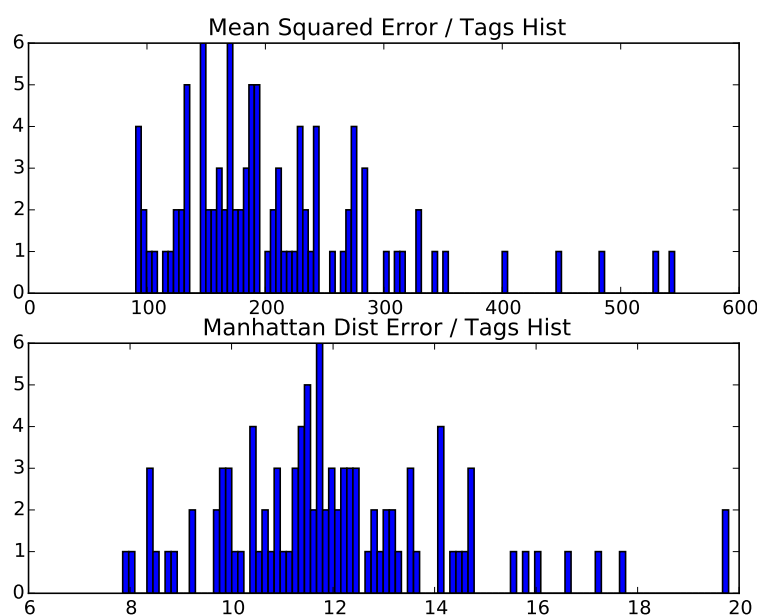
- Funkcja błędu oparta o odległość Manhattan - naturalna i prosta do analizy miara błędu.

- Funkcja błędu średniokwadratowego - w celu porównania z poprzednią funkcją błędu. Nie bardzo wiadomo co oznaczają wartości tej funkcji.

W przypadku tych błędów "odległością" mierzona przez funkcję błędu jest różnica "pewności" dla tagów z agregatu tagów dla *sztucznego użytkownika* i kolejnych pięciu rekomendacji. Przy obliczaniu funkcji błędu nie bierzemy pod uwagę tagów o bardzo małej pewności - zakładamy, że są zaburzeniami i odrzucamy je. W przypadku, gdy agregat  $X$  jest oznaczany tagiem o etykiecie  $e$ , a drugi agregat nie - zakładamy że "pewność" dla drugiego agregatu tagu o etykiecie  $e$  wynosi 0.

## 4.5 Wyniki testowania

W celu przeprowadzenia analizy wylosowaliśmy 100 par sztucznych użytkowników i ich kolejnych pięciu rekomendacji. Dla każdej z tych par wyliczyliśmy funkcję błędu MDE (*Manhattan Distance Error*) oraz MSE (*Mean Squared Error*). Poniżej przedstawiamy histogram wartości obu funkcji błędów, podzielonych przez liczbę tagów w agregatach:



Rysunek 1: Histogram zastosowanych funkcji błędu

Jak można zauważyć, wartości funkcji MDE wskazują na wysoką skuteczność systemu - wartości oscylują w okolicach wartości 10. Przy zakresie od zera do stu "pewności" tagów, których może być wiele jest to wartość pożądana - oznacza to, że wśród WSZYSTKICH istotnych tagów łączna różnica ich "pewności" oscyluje w granicach tej wartości. Oczywiście wszystko zależy od liczności agregatów tagów. W przypadku sztucznych użytkowników uśredniona liczność odpowiadającego mu agregatu oscylowała dookoła wartości 2000. W przypadku

agregatu kolejnych rekomendacji było to 160. Liczby te uwzględniają tagi nieistotne, wycięte podczas liczenia funkcji błędu - niemniej jednak pozwalają mieć ogólny wgląd w to, jak liczne są zbiory.

Warto wspomnieć o pomysłach, które okazały się nieudane - pierwszym naszym pomysłem na funkcję błędu była f. błędu bazująca na indeksie Giniego. Z racji jej własności (etykiety tagów byłyby nierozróżnialne) i trudności w zdefiniowaniu prawdopodobieństwa należenia do analizowanego przez indeks zbioru odrzuciliśmy ten pomysł jako nietrafiony.

## 5 Wnioski końcowe, podsumowanie, perspektywy rozwoju

Pomimo posiadania wiedzy na temat lokalnego podobieństwa utworów muzycznych, utworzenie miar przechodniego (tudzież "globalnego") podobieństwa pewnych utworów muzycznych okazało się wyzwaniem. System rekomendujący przedstawiony w tym projekcie jest przykładem systemu opartego na idei *content-based-filtering*. Wiedza, którą generuje z informacji zadanych w zbiorze danych, jest wiedzą wynikającą wyłącznie z własności przedmiotów rekomendowanych (w tym wypadku – utworów muzycznych). Z racji operowania pojęciem macierzy podobieństwa (która jest obiektem pamięciowo rozległym) wymagane były skomplikowane operacje grupowania danych.

Wielkość opracowywanego zbioru danych sugerowałaby użycie podejść typu *collaborative filtering*. Niestety nie posiadaliśmy danych o użytkownikach i ich ocenach/preferencjach dotyczących utworów – stąd też rozwiązanie jest oparte na potencjalnie "gorszej" idei. Niemniej jednak systemy tego typu sprawdzają się lepiej w przypadku, gdy zgromadzonych danych jest niewiele – nie zależą też od ocen użytkowników.

Perspektywy rozwoju tego systemu leżą w ulepszeniu istniejącej implementacji (usprawnienie obliczeń, zmniejszenie zużycia pamięci) oraz implementacji nowych elementów mających wpływ na wyniki rekomendacji:

1. Możliwe jest wprowadzenie wyborów użytkownika – mógłby on oznaczać pewne utwory jako nie lubiane. System rekomendacyjny karałby wtedy utwory znajdujące się w bezpośrednim otoczeniu (odległości) od takich utworów.
2. Możliwa jest także dalsza regularyzacja funkcji błędu. Rozważanym przez nas pomysłem było m.in. wprowadzenie czynnika *maksymalnej liczności wierzchołków grafu*. System rekomendacyjny karałby wtedy za podgrafy zawierające zbyt wiele elementów.