Projekt z eksploracji danych

Filip Chudy, Marcin Grzywaczewski

18 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

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

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 $V_1 \backslash \{v_i, v_j\} \subseteq V_2$, $V_2 \backslash V_1 = \{v_k\}$ 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).

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

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

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

Sama funkcja kosztu jest iloczynem tych trzech czynników:

$$F(G) = (WCD(G) + SVP(G)) \times BCD(G) \tag{5}$$

.

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:

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

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

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

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 kliki za pomocą algorytmu Floyda-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.

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

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 rekomendacyjnego – 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 ńieprawidłoweźostają 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 źawierania wierzchołka"jest zrealizowane w czasie stałym (struktura słownika). Całkowita złożoność prodedury 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 $\{0, 1, \ldots, n\}$. 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. Wybranym 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 $\{-1, -2, ..., -n\}$) 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 podgrafem 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

Efektem redukcji grafu jest struktura, której liczba wierzchołków na pojedyńczym 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 przechywywanie 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. Prodecedura generowania tych plików (rekurencyjny) jest zrealizowany w obiekcie `Graph` jako `pickle_graph`.

4 Wyniki

5 Wnioski końcowe, podsumowanie, perspektywy rozwoju