Friedrich-Schiller-Universität Jena

Fakultät für Mathematik und Informatik

Projekt: Recommender Systems

Projektarbeit

Prüfungsleistung im Masterstudiengang Computational and Data Science im Modul

Statistische Lerntheorie (Lab)

Studentin: Dan Jia

Jena, 11.09.2018

Inhaltsverzeichnis

Implementierung eines Recommender Systems

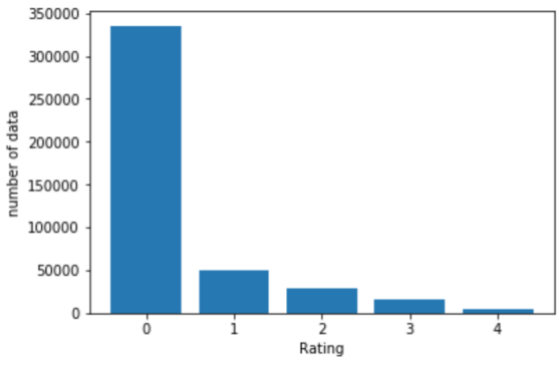
Dan Jia
Statistische Lerntheorie (Lab)
Friedrich-Schiller-Universität Jena

Einleitung

Mit der zunehmenden Bedeutung des Webs als Medium für elektronische und geschäftliche Transaktionen wird die Entwicklung von Recommender Systems immer wichtiger. Das Ziel hier ist es, mit unterschiedlichen Methoden ein Recommender System in Python zu implementieren.

Daten

Gegeben sind zwei Dateien, train.csv und qualifying.csv, bestehend aus 434641 bekannten Ratings und 108660 unbekannte Ratings (nur User und Item bekannt). Die entsprechenden Daten können als 2080×5499 (Items×User) oder 5499 × 2080 (User × Items) formatiert werden. Der Wertbereich der Ratings ist zwischen 0 und 4. Innerhalb der bekannten Ratings ist die Verteilung der Ratings von 0 bis 4 jeweils 335499, 50169, 29623, 15138, 4212, was hier als Histogramm gezeigt wird:



Um die Daten einfacher verarbeiten zu können, addieren wir erst 1 zu allen bekannten Ratings, dadurch wird der Wertbereich der Ratings in 1 bis 5 transformiert. Deswegen ist es auch nötig, dass von den vorhergesagten Ratings für qualifying.csv jeweils 1 abgezogen wird. Wir teilen die bekannten Daten in train.csv in zwei Teile auf, welche wir train_data und test_data nennen, damit wir die Verfahren trainieren und bewerten könnten. Bei der neighborhood-based Methode werden die Daten durch *scipy.sparse.coo_matrix()* zu einer 2080 × 5499 Matrix formiert.

Algorithmen

Algorithmus 1:

k-nearest neighbors algorithm (k-NN)

Nachbarschaftsbasierte kollaborative Filteralgorithmen basieren auf der Annahme, dass ähnliche User ähnliche Muster des Bewertungsverhaltens aufweisen und ähnliche Items ähnliche Rating erhalten. Es gibt zwei Haupttypen von Nachbarschafts-basierten Algorithmen: *User-based-* und *Item-based collaborative filtering*. [Agg16, ch.3]

Im Code wird *Item-based collaborative filtering* implementiert. Dafür werden folgende Schritte durchgeführt:

- Berechnen der Ähnlichkeitsmatrix.
- Suchen der k ähnlichsten Nachbarn für jedes Item.
- Vorhersagen aller unbekannten Ratings
- Berechnen von *RMSE (Root Mean Squared Error)* für die Abweichung zwischen den vorhergesagten Werten *r_hat* für test_data und den wahren Werten r aus test_data.
- Berechnen der Ähnlichkeitsmatrix mit allen bekannte Ratings aus train.csv und berechnen der vorhergesagter Ratings für qualifying.csv.

Normalerweise ist die Zahl der Items geringer als die Zahl der User, wie auch in der hier gegebenen Situation. D.h. die Ähnlichkeitsmatrix für *Item-based collaborative filtering (2080×2080)* wird kleiner als für *User-based collaborative filtering (5499×5499)*. Dies ist ein Grund warum wir nicht *User-based*, sondern *Item-based collaborative filtering v*erwenden, damit die Berechnung schneller ist.

Die Berechnung der Ähnlichkeit zwischen den unterschiedlichen Item-Vektoren (unterschiedlicher Zeile in der 2080 × 5499 Bewertungsmatrix) wird mittels Kosinus-Ähnlichkeit implementiert. In dem wir die Abweichung von der mittleren Bewertung von verschiedenen Users für das selbe Item berücksichtigt, zentrieren wir jede Zeile der Ratingsmatrix auf einen Mittelwert von Null, bevor die Ähnlichkeiten für diese Mittelwert-zentrierte Matrix berechnet werden.

$$s_{iu} = r_{iu} - \mu_i, \qquad s_{ju} = r_{ju} - \mu_j$$

$$sim(i,j) = \frac{\sum_{u \epsilon U_i \cap U_j} s_{iu} \cdot s_{ju}}{\sqrt{\sum_{u \epsilon U_i \cap U_j} s_{iu}^2} \cdot \sqrt{\sum_{u \epsilon U_i \cap U_j} s_{ju}^2}}$$

- r_{iu}: Rating Item i, User u
- r_{ju}: Rating Item j, User u
- μ_i: Mittelwert alle für Item i gegebenen Ratings
- μ_j: Mittelwert alle für Item j gegebenen Ratings
- U_i: Menge alle User welcher ein Rating für Item i abgegeben haben
- U_j: Menge aller User welcher ein Rating für Item j abgegeben haben

Für jedes Paar (User, Item) für welches ein Rating vorhergesagt werden soll, wird für das Item die Menge der k nächsten Nachbarn bestimmt, wobei nur Items berücksichtigt werden, für welche der User auch ein Rating abgegeben hat. Der gewichtete Mittelwert dieser bekannten Ratings wird als prognostizierte Rating verwendet. Als Gewichtung wird die Kosinusähnlichkeit des entsprechenden Items zum Zielitem verwendet. Es wird berücksichtigt, dass verschiedene User verschieden großzügig mit ihren Ratings sind, deshalb werden nicht die absoluten Ratings gewichtet, sondern die Abweichungen zum mittleren Rating für das Item:

$$\hat{r}_{iu} = \mu_i + \frac{\sum_{j \epsilon P} sim(i,j) \cdot s_{ju}}{\sum_{j \epsilon P} |sim(i,j)|}$$

- $\hat{r}_{iu}$: vorhergesagtes Rating für Item i, User u
- P: Menge der Items (k oder weniger) mit größter Ähnlichkeit

Um das trainierte Model zu bewerten, berechnen wir *RMSE (Root Mean Squared Error)* zwischen den vorhergesagten Werten *r_hat* und bekannten Werten r aus test_data. Anschließend wird das Vorgehen widerholt für qualify.csv, wobei die Ähnlichkeitsmatrix mittels aller Daten aus train.csv berechnet wird.

Algorithmus2:

Model-based Collaborative Filtering

Modellbasierte Recommender Systeme haben oft eine Reihe von Vorteilen gegenüber Nachbarschaftsmethoden, z.B. bessere Speichereffizienz, schnellere Trainingsgeschwindigkeit und Vorhersagegeschwindigkeit, Vermeiden von Overfitting usw.

Wenn wir Einträge einer Matrix vorhersagen wollen, nutzen wir aus, dass für jede m × n Matrix R ein Paar einer m × k Matrix P und einer n × k Matrix Q mit k≤ $rank(R)$existiert, so dass:

$$R = PQ^T$$

Das Ziel ist es daher, die Matrix $\boldsymbol{R}$ in zwei Matrizen mit kleinerem Rang zu zerlegen:

$$\boldsymbol{R} \approx \boldsymbol{P}\boldsymbol{Q}^T$$

Dieser Ansatz kann dahingehend interpretiert werden, dass ein latentes Faktormodell gefunden wird. Jede Bewertung r_{ui} in R kann näherungsweise als ein Skalarprodukt des i-ten Benutzerfaktors und des u-ten Elementfaktors ausgedrückt werden:

$$\hat{r}_{ui} = q_i^T p_u$$

Andererseits sind die Ratings nicht nur von latenten Faktoren abhängig, sondern auch davon

ob ein User streng oder großzügig ist, und von vorhandene Ratings von verschiedenen Usern für das selbe Item. Die Vorhergesagten Ratings sollen wie folgend berechnet werden:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

- $\hat{r}_{ui}$: vorhergesagtes Rating für User u, Item i
- μ: Mittelwert von alle bekannten Ratings
- b_u: Abweichung zwischen dem mittleren Rating von User u und Mittelwert μ
- b_i: Abweichung zwischen dem mittleren Rating von Item i und Mittelwert μ

Der Lernprozess für das Modell kann wie folgt zusammengefasst werden:

- Den Variablen b_u, b_i, p_u, q_i einen Startwert gegeben.
- Die Variablen werden zur Vorhersage verwendet.
- Das Vorhersageergebnis wird mit dem bekannten Ergebnis verglichen.
- Die Variablen werden gemäß dem Vergleichsergebnis korrigiert.
- Berechnen von *RMSE (Root Mean Squared Error)* für die Abweichung zwischen den vorhergesagten Werten *r_hat* für test_data und den wahren Werten r aus test_data.
- Trainieren alle bekannte Ratings in train.csv und berechnen der vorhergesagter Ratings für qualifying.csv.

Ziel ist es, die Werte der Parameter so anzupassen, dass die folgende Loss-Funktion (mit Regularisierungen) minimiert wird [Lu14]:

$$\min_{b_u, b_i, p_u, q_i} J = \sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2)$$

- r_{ui}: bekannte Rating für User u, Item i

$$e_{ui} = r_{ui} - \hat{r}_{ui}$$

Wir nutzen hier eine *Gradient Descent* Methode, um das Minimum zu finden. Die Variable, die wir iterativ optimieren sollen, sind:

$$b_u \leftarrow b_u + \gamma(e_{ui} - \lambda b_u)$$

$$b_i \leftarrow b_i + \gamma(e_{ui} - \lambda b_i)$$

$$p_u \leftarrow p_u + \gamma(e_{ui} \cdot q_i - \lambda p_u)$$

$$q_i \leftarrow q_i + \gamma(e_{ui} \cdot p_u - \lambda q_i)$$

Wenn die RMSE Wert nicht mehr sinkt, wird die Iteration abgebrochen.

Implementierung

KNN:

Durch Testen der Parameter k∈{2, 4, 5, 6, 10, 15} wird der geringste RMSE für k=5 gefunden, man wählt daher hier Parameter k=5.

Matrix-Faktorisierung:

Wir geben 0 als Startwerte für $b_u, b_i,$ und *0.1 * rand(K,1) / sqrt(K)* als Starwerte für p_u, q_i vor. K ist die Dimension der Matrizen P und Q. Durch testen der Wert von K∈{10, 20, 30, 40, 50, 60}, Gamma ∈ {0.02, 0.03, 0.04, 0.05, 0.06}, Lambda∈{0.04, 0.05, 0.06, 0.15, 1.15} wird das beste Ergebnis gefunden für K=50, Gamma=0.04, Lambda=0.05, es werden demensprechend diese Werte gewählt. Der minimale Wert der Loss-Funktion wird immer ungefähr bei der 20-ten Iteration gefunden.

Vergleich der Algorithmen anhand des RMSE:

Um die untersuchten Algorithmen bewerten zu können, berechnen wir RMSE für *test_data* und vergleichen die Ergebnisse:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(r_{ui} - \hat{r}_{ui})^2}{n}}$$

- n: Anzahl der Ratings in den Testdaten

Algorithmus	KNN	Matrix-Faktorisierung
RMSE	0.525399	0.506627

Referenz

[Agg16] Charu C Aggarwal. Recommender systems. Springer 2016.

[Lu14] Lu,C.; and Tang, J. Generalized nonconvex nonsmooth low-rank minimization. 2014.