# Recommendation systems

## Popularity based recommendation systems

* No need for historical data
* Can be used from day one, with no user’s previous data, no *cold-start*
* Not personalized

Google news, YouTube

## Classification models

* Uses product and users characteristics to predict if a user will be interested in a product

(1 – 0 output)

## Content-Based Recommendation System

* Recommend similar products (cos-distance, Euclidian distance, Jaccard distance)
* No need for user’s data, no cold start.

## K-means clustering

* Choose cluster centroids, assign points to clusters, calculate means,
* Reassign points to clusters, recalculate means.
* Find in which cluster is the item, recommend other points in cluster.

## Collaborative Filtering

* If user A and B have similar preferences, we recommend to A, items preferred by B.

**Data:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| User | Item1 | Item2 | Item3 | Item4 |
| U1 | 1 | 0 | 0 | 1 |
| U2 | 0 | 0 | 1 | 1 |
| U3 | 1 | 0 | 1 | 1 |
| U4 | 0 | 1 | 0 | 1 |
| U5 | 1 | 0 | 1 | 0 |

**Item-based nearest-neighbor collaborative filtering:** Calculate a similarity matrix.

Are items *i* and *j* preferred by similar users?

SM[i][j] = cos\_dist(DATA[][i], DATA[][j]).

– calculate distance (cosine similarity) between columns corresponding to Item1 and Item3

SM[1][3] = cos\_dist([1, 0, 1, 0, 1][0, 1, 1, 0, 1]) --

Sort SM[i][] and calculate neighbors for an item *i* (chose first *n* items in increasing order by SM[i][.] )

**User-based nearest-neighbor collaborative filtering**: System finds the users interested in similar products.

Normalize user ratings, users have different scales for ratings (between 1 and 5 or between 2 and 4 etc.)

NaN values in the matrix can be replaced with means.

Select only a list with fix number of users (*nearest neighbors*).

Calculate only for items *liked* by *nearest neighbors.*

Python useful libraries: pandas, numpy, sklearn, scipy [http://www.salemmarafi.com/code/collaborative-filtering-with-python/](https://neo4j.com/developer/python/)

## Graph database Neo4J

https://neo4j.com/download/

After install check: [http://localhost:7474/browser/](https://neo4j.com/blog/cypher-union-query-using-collect-clause/)

**Graph databases**:

**nodes**: entities in the graph, can hold attributes each node has a label identifying node’s role.

**relationships**: provide semantic connections between two node entities each relationship has a type, a start node and an end node can be navigated in both directions

**attribute**: properties of nodes or relationships. An attribute is given by a key-value pair. Relationship attributes are typically quantitative properties (weights, costs, ratings etc.).

Movie Tutorial :play movie graph

Cypher is the primary interface for Neo4j.

Neo4j users use Cypher to construct expressive and efficient queries to do any kind of create, read, update, or delete (CRUD) on their graph.

Cypher uses an ASCII-art type of syntax where (nodes)-[:ARE\_CONNECTED\_TO]->(otherNodes).

Cypher easily expresses path queries, which may be harder to express in SQL.

O imagine care conține text

Descriere generată automat

**Examples of queries in Cypher:**

* Find node label Person, with value for attribute name ‘Tom Hanks’:

MATCH (tom:Person {name: 'Tom Hanks'})

RETURN tom

* Find movies in which Tom Hanks played a role and directors.

MATCH (tom:Person {name: 'Tom Hanks'})

WITH tom as a

MATCH (a)-[:ACTED\_IN]->(m)<-[:DIRECTED]-(d)

RETURN a,m,d LIMIT 10;

* Find paths between Tom Hanks and coCoActors (of length 2)

MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED\_IN]->(movie1:Movie)<-[:ACTED\_IN]-(coActor:Person)-[:ACTED\_IN]->(movie2:Movie)<-[:ACTED\_IN]-(coCoActor:Person)

WHERE tom <> coCoActor

AND NOT (tom)-[:ACTED\_IN]->(:Movie)<-[:ACTED\_IN]-(coCoActor)

RETURN coCoActor.name

MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED\_IN]->(movie1:Movie)<-[:ACTED\_IN]-(coActor:Person)-[:ACTED\_IN]->(movie2:Movie)<-[:ACTED\_IN]-(cruise:Person {name: 'Tom Cruise'})

WHERE NOT (tom)-[:ACTED\_IN]->(:Movie)<-[:ACTED\_IN]-(cruise)

RETURN tom, movie1, coActor, movie2, cruise

MATCH (n:Person { name: 'Rami Malek ' })<-[:ACTED\_IN]-(m)

DETACH delete m

* Returning shortest path (BF or DF search)

MATCH (KevinB:Person {name: 'Kevin Bacon'} ),

(Al:Person {name: 'Al Pacino'}),

p = shortestPath((KevinB)-[:ACTED\_IN\*]-(Al))

WHERE all(r IN relationships(p) WHERE r.roles IS NOT NULL)

RETURN p

MATCH (KevinB:Person {name: 'Kevin Bacon'}),

(Al:Person {name: 'Al Pacino'}),

p = shortestPath((KevinB)-[\*]-(Al))

WHERE length(p) > 1

RETURN p

* *counting the number of paths between Tom Hanks and each coCoActor*

MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED\_IN]->(m)<-[:ACTED\_IN]-(coActors),

(coActors)-[:ACTED\_IN]->(m2)<-[:ACTED\_IN]-(cocoActors)

WHERE NOT (tom)-[:ACTED\_IN]->()<-[:ACTED\_IN]-(cocoActors) AND tom <> cocoActors

RETURN cocoActors.name AS Recommended, count(\*) AS Strength ORDER BY Strength DESC

**Connect with Python:**

>>pip install **neomodel**

from neomodel import StructuredNode, StringProperty, IntegerProperty, RelationshipTo, RelationshipFrom,db, config  
  
  
config.DATABASE\_URL = 'bolt://neo4juser:pass@localhost:7687'  
  
  
  
class Movie(StructuredNode):  
 tagline = StringProperty(unique\_index=True)  
 title = StringProperty(unique\_index=True)  
 released = IntegerProperty(unique\_index=False)  
  
  
  
class Person(StructuredNode):  
 name = StringProperty(unique\_index=True)  
 born = IntegerProperty(unique\_index=False)  
 moviesact = RelationshipFrom('Movie', 'ACTED\_IN')  
 moviesdir = RelationshipTo('Movie', 'DIRECTED')  
  
  
#rami\_malek10 = Person(name='Rami Malek ', born = 1981).save()  
#bohemian\_rhapsody10 = Movie(tagline='The story of the legendary British rock band Queen',title='Bohemian Rhapsody',released=2018).save()  
#rami\_malek10.moviesact.connect(bohemian\_rhapsody10)  
  
#for person in Person.nodes:  
# print(person.name)  
  
  
for person in Person.nodes.filter(born=1981):.  
 print(person.name)

## Context aware recommendation systems

User x Item x Context → Rating

**Context**

* information that characterizes an entity: who, what, when, where.
* Multidimensional approach

**Step1:** In *Neo4J Desktop*: Create project, create database *Restaurants*, start.

**Step2:** Add unique constraints:

CREATE CONSTRAINT ON ( rest:Restaurant ) ASSERT rest.pid IS UNIQUE;

CREATE CONSTRAINT ON ( company:Company ) ASSERT company.name IS UNIQUE;

CREATE CONSTRAINT ON ( usr:User ) ASSERT usr.UserID IS UNIQUE;

**Step 3:** Import users, create company, user, habits, personal\_info and cuisine nodes and relationships.

:auto USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM

"https://raw.githubusercontent.com/kaisesha/cdrgraph/master/Users\_50.csv"

As line

WITH line

MERGE (company:Company {name: "CompanyName"})

MERGE (usr:User {name: line.UserID, uid: line.UserID})

CREATE (company)-[:USER\_PROFILE]->(usr)

CREATE (prsnl:Personal {name: "Personal"})

CREATE (usr)-[:PERSONAL]->(prsnl)

CREATE (intrst:Interest {attr9: line.Interest})

CREATE (prsnl)-[:INTEREST]->(intrst)

CREATE (prsnlt:Persnlty {attr10: line.Personality})

CREATE (prsnl)-[:PERSONALITY]->(prsnlt)

CREATE (trnsp:Transp {attr5: line.Transport})

CREATE (prsnl)-[:TRANSPORT]->(trnsp)

CREATE (marrd:Married {attr6: line.Marital\_Status})

CREATE (prsnl)-[:MARITAL\_STATUS]->(marrd)

CREATE (brthyr:Byear {attr8: line.Birth\_Year})

CREATE (prsnl)-[:BIRTH\_YEAR]->(brthyr)

CREATE (relgn:Religion {attr11: line.Religion})

CREATE (prsnl)-[:RELIGION]->(relgn)

CREATE (habit:Habit {name: "Habits"})

CREATE (usr)-[:HABITS]->(habit)

CREATE (smkr:Smoker {attr1: line.Smoker})

CREATE (habit)-[:SMOKER]->(smkr)

CREATE (drnk:Drink {attr2: line.Drink\_Level})

CREATE (habit)-[:DRINK\_LEVEL]->(drnk)

CREATE (dress:Dress {attr3: line.Dress\_Pref})

CREATE (habit)-[:DRESS]->(dress)

CREATE (hijos:Hijos {attr7: line.Hijos})

CREATE (habit)-[:HIJOS]->(hijos)

CREATE (ambnc:Ambnce {attr4: line.Ambience})

CREATE (habit)-[:AMBIENCE]->(ambnc)

CREATE (actv:Activity {attr12: line.Activity})

CREATE (habit)-[:USER\_PROFILE]->(actv)

CREATE (budgt:Budget {attr13: line.Budget})

CREATE (habit)-[:BUDGET]->(budgt)

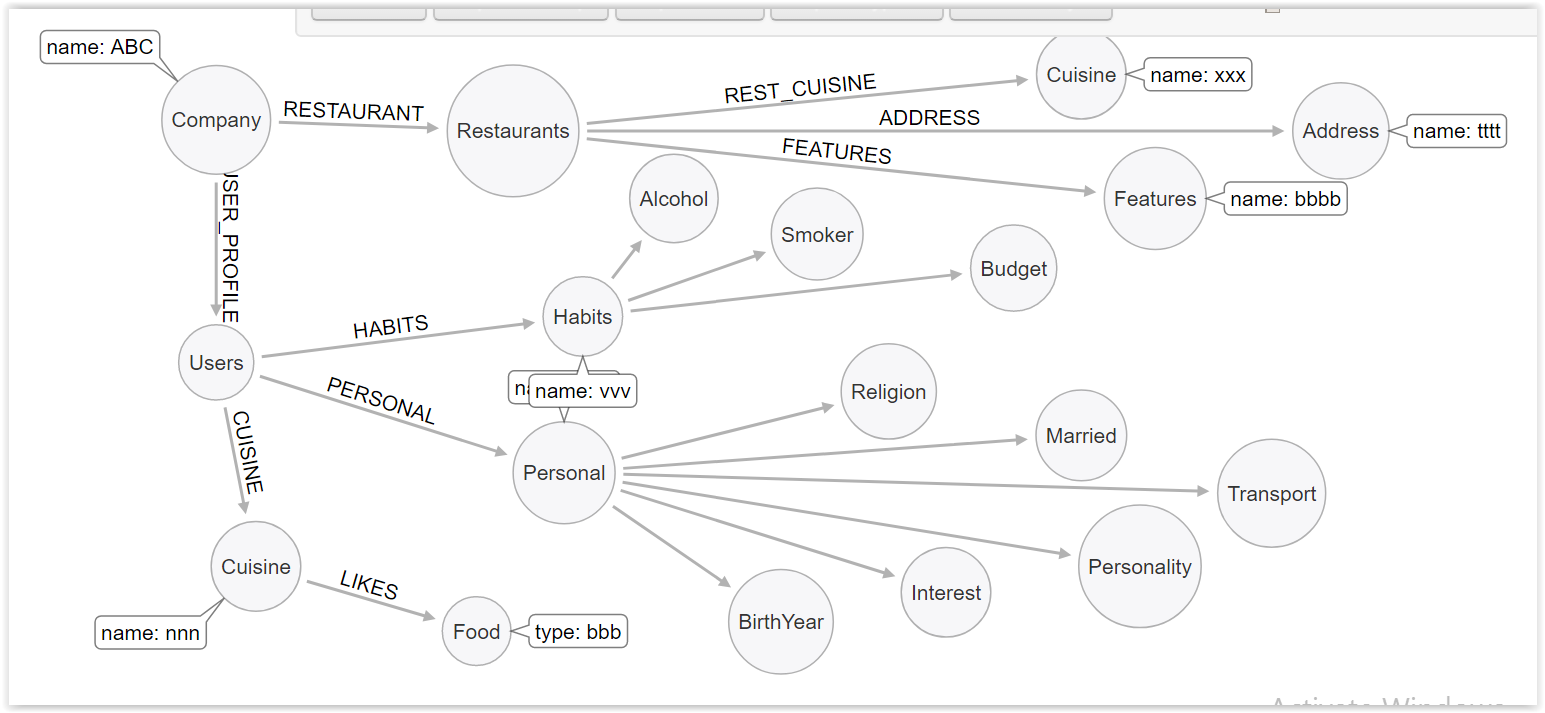
CREATE (m:Cuisine {name: "Cuisine"})

CREATE (usr)-[:CUISINE]->(m)

;

Observations:

* If the CSV file contains a significant number of rows (approaching hundreds of thousands or millions), USING PERIODIC COMMIT can be used to instruct Neo4j to perform a commit after a number of rows. This reduces the memory overhead of the transaction state. By default, the commit will happen every 1000 rows.
* The :auto command will send the Cypher query following it, in an auto committing transaction
* MERGE either matches existing nodes and binds them, or it creates new data and binds that. It’s like a combination of MATCH and CREATE that additionally allows you to specify what happens if the data was matched or created. [https://neo4j.com/docs/cypher-manual/current/clauses/merge/](https://neo4j.com/developer/cypher/)
* CREATE command creates nodes or relationships. [https://neo4j.com/docs/cypher-manual/current/clauses/create/](https://neo4j.com/developer/spring-data-neo4j/)



**Step 4:** Add User cuisine

:auto USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM

"https://raw.githubusercontent.com/kaisesha/cdrgraph/master/UCuisine.csv"

As line

WITH line

CREATE (f:Food {name: line.Cuisine})

WITH line, f

MATCH (n:User {uid:line.UserID})-[:CUISINE]->(m:Cuisine {name: 'Cuisine'})

MERGE (m)-[:LIKE]->(f);

Observations:

* MATCH command is used to search for patterns. [https://neo4j.com/docs/cypher-manual/current/clauses/match/](http://localhost:7474/browser/)

**Step 5**: Import restaurants

:auto USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM

"https://raw.githubusercontent.com/kaisesha/cdrgraph/master/Restaurants.csv"

As line

WITH line

MERGE (company:Company {name: "Ameyasoft"})

MERGE (rest:Restaurant {pid: toInteger(line.PlaceID), name: line.Name})

CREATE (company)-[:RESTAURANT]->(rest)

CREATE (addr:Addrs {street: line.Address, city: line.City, state: line.State, zip: line.Zip, country: line.Country})

CREATE (rest)-[:ADDRESS]->(addr)

CREATE (featrs:Features {alcohol: line.Alcohol, smoking: line.Smoking\_Area, dress: line.Dress\_Code, price: line.Price, ambience: line.Ambience})

CREATE (rest)-[:FEATURES]->(featrs)

;

Observations:

* For some calculations may use scalar functions: [https://neo4j.com/docs/cypher-manual/current/functions/scalar/](https://neo4j.com/developer/cypher/guide-build-a-recommendation-engine/)

**Step 6**: Import restaurant cuisine

:auto USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM

"https://raw.githubusercontent.com/kaisesha/cdrgraph/master/RestCuisine.csv"

As line

WITH line

MATCH (n:Restaurant {pid: toInteger(line.PlaceID)})

MERGE (n)-[:REST\_CUISINE]->(cuse:Cusine {name: line.Cuisine});

**Step 7**: Query nodes and relationships.

MATCH (c)-[r:USER\_PROFILE|RESTAURANT]->(n)-[]->(p)

WHERE n.uid IN['U1001', 'U1002', 'U1003'] or n.pid IN [132609, 132613, 132630]

RETURN c, n, p LIMIT 20;

**Step 8:** Query users: Mexican food, non-smoking, and medium priced

MATCH (c)-[]->(n)-[:CUISINE]->(r)-[:LIKE]->(t:Food {name: "Mexican"})

WITH COLLECT (n) AS nodes, t

UNWIND nodes AS n1

MATCH (c)-[]->(n1)-[:HABITS]->(q)-[:SMOKER]->(v:Smoker {attr1: "false"})

WITH COLLECT (n1) AS nodes, t, v

UNWIND nodes AS n2

MATCH (c)-[]->(n2)-[:HABITS]->(q)-[:BUDGET]->(v1:Budget {attr13: "medium"})

RETURN n2.uid as User, t.name as Cuisine, v.attr1 as Smoker, v1.attr13 as Budget;

**Step 9:** Query restaurants: Mexican food, non-smoking, and medium priced

MATCH (c)-[]->(n)-[:CUISINE]->(r)-[:LIKE]->(t:Food {name: "Mexican"})

WITH COLLECT (n) AS nodes, t

UNWIND nodes AS n1

MATCH (c)-[]->(n1)-[:HABITS]->(q)-[:SMOKER]->(v:Smoker {attr1: "false"})

WITH COLLECT (n1) AS nodes, t, v

UNWIND nodes AS n2

MATCH (c)-[]->(n2)-[:HABITS]->(q)-[:BUDGET]->(v1:Budget {attr13: "medium"})

WITH v1, t

MATCH (c)-[]->(n2)-[:REST\_CUISINE]->(p:Cusine {name: t.name})

WITH COLLECT(n2) AS pn, v1

UNWIND pn AS n3

MATCH (c)-[]->(n3)-[:FEATURES]->(q1:Features {price: v1.attr13, smoking: "none"})

WITH COLLECT(n3) AS pn

UNWIND pn AS n4

WITH DISTINCT n4

MATCH (c)-[]->(n4)-[:ADDRESS]-(k)

RETURN n4.name AS Restaurant,  k.city AS City;

[1] [https://raw.githubusercontent.com/kaisesha/cdrgraph/master/UCuisine.csv](https://neo4j.com/developer/guide-import-csv/)

[2] [https://raw.githubusercontent.com/kaisesha/cdrgraph/master/RestCuisine.csv](https://portal.graphgist.org/graph_gists/multidimensional-approach-to-recommender-systems/source)

[3] [https://towardsdatascience.com/recommendation-system-matrix-factorization-d61978660b4b](http://www.salemmarafi.com/code/collaborative-filtering-with-python/)

[4] [https://neo4j.com/developer/get-started/](https://neo4j.com/docs/cypher-manual/current/clauses/match/)

[5] [https://neo4j.com/developer/cypher/guide-build-a-recommendation-engine/](https://neo4j.com/docs/cypher-manual/current/clauses/merge/)

[6] [https://neo4j.com/developer/spring-data-neo4j/](https://raw.githubusercontent.com/kaisesha/cdrgraph/master/RestCuisine.csv)

[7] [https://neo4j.com/developer/python/](https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/)

[8] [https://neo4j.com/developer/dotnet/](https://neo4j.com/docs/cypher-manual/current/functions/scalar/)

[9] [https://neo4j.com/graphgist/multidimensional-approach-to-recommender-systems](https://raw.githubusercontent.com/kaisesha/cdrgraph/master/UCuisine.csv)

[10] [https://neo4j.com/developer/guide-import-csv/](https://neo4j.com/docs/cypher-manual/current/clauses/create/)

[11] [https://portal.graphgist.org/graph\_gists/multidimensional-approach-to-recommender-systems/source](https://towardsdatascience.com/recommendation-system-matrix-factorization-d61978660b4b)

[12] [https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/](https://neo4j.com/developer/get-started/)

[13] [https://neo4j.com/blog/cypher-union-query-using-collect-clause/](https://neo4j.com/developer/dotnet/)

[14] [https://neo4j.com/developer/cypher/](https://neo4j.com/graphgist/multidimensional-approach-to-recommender-systems)