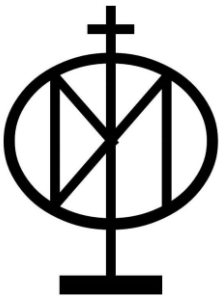
Софийски университет „Св. Климент Охридски“
Факултет по математика и информатика



Курсов проект

по

Интелигентни системи

на тема:

Recipes Content-Based Recommendation System

Изготвили:
Слави Кадиев – 61679
Милка Ферезлийска – 61710
Християн Димитров – 61669
Светимир Игнатов – 61726
Специалност:
Софтуерно инженерство
Курс: 4 курс

Проверил: проф. д-р Иван Койчев

гр. София
2017 г.

Съдържание

1. Мотивация и задача на курсовата работа 2
2. Кратък обзор 2
3. Нашето решение 3
    3.1. TF-IDF + Vector Space Model 3
        3.1.1. TF-IDF 3
        3.1.2. Vector Space Model 4
    3.2. kNN + k-d tree 5
        3.2.1. kNN 5
        3.2.2. k-d tree 6
    3.3. k-Means 7
    3.4. Naive Bayes 9
4. Програмна реализация 10
    4.1. TF-IDF 10
    4.2. Vector Space Model 11
    4.3. k-Means 14
    4.4. Naive Bayse 16
    4.5. kNN + k-d tree 12
5. Заключение 18
6. Литература 19
    6.1. Recommender systems: 19
    6.2. Имплементации с Python: 19
    6.3. Информация за TF-IDF & Space Vector Model 19
    6.4. Информация за k-Means алгоритъма: 19
    6.5. Информация за Naive Bayes 20
    6.6. Информация за kNN + k-d tree 20
    6.7. Hybrid Recommendation Filtering: 20

1. Мотивация и задача на курсовата работа

Проектът „Vkusotiiki-bg“ е добре известен сред преподавателите в нашия факултет. Причината е ясна – нашият екип иска традиционните български ястия да добият колкото се може по-голяма популярност сред нас – младите, както тук, така и в чужбина.

Създаването на vkusotiiki-bg.com предвижда все повече и повече хора да започнат да го използват. Това ще доведе до нуждата от съхранение и обработка на голямо количество данни в чист вид. Решихме да добавим изкуствен интелект към системата, което да подобри значително възможността потребителите да останат доволни, използвайки сайта, и да се "рекламират" възможно най-голям обем от рецепти.

През семестъра изучихме доста алгоритми за класификация и клъстеризация и опитахме някои от тях в домашните си. Постигнатите резултати ни дадоха идея и тласък да се опитаме да ги приложим в нашето ежедневие, за да видим как работят всъщност.

За настоящия проект беше необходимо създаването на функционалност за предлагане на рецепти. Това би могло да се случи чрез прости тагове, търсене на сходни думи и съставки в различните рецепти, подобни автори и други такива общи признаци, но е твърде лесна и не вдъхновяваща задача. Решихме да създадем изкуствен интелект, който да може да се самообучава и да дава все по-добри предположения. За целта сме използвали 4 различни алгоритъма.

За повече информация относно опитите, решенията и процесът на работа и реализация на проекта погледнете в следващата секция на настоящата документация „Кратък обзор“.

2. Кратък обзор

След направен кратък обзор на съществуващите системи за предложения, установихме, че е най-добре да подходим към проблема от различни страни и да видим къде възможните решения имат общи точки. Избрахме да използваме смесица от класификационни и клъстеризационни алгоритми, а именно kNN, Наивен Бейсов класификатор, kMeans и TF-IDF. В процеса на работа всеки от тях прояви своите силни и слаби черти и така се формира настоящата документация. Приложени са детайлни описания на извършените опити, снимки и обяснения, които да информират за крайния резултат от употребата на различните алгоритми.

Важно е да се отбележи какви данни използваме, за да тестваме алгоритмите. В началото се наложи да генерираме свои данни за потребителски харесвания на рецепти, с които да тестваме. В последствие се създаде начин да може да се събират потребителски харесвания в csv файл, той да се обработи и получените данни да се използват в системата. Така част от тестовете се провалиха заради липсата на логика при човешкия избор понякога, а други показаха силните страни на използваните алгоритми.

Работата по имплементацията им се разделихме, както следва:

- Христиан Димитров (61669) - TF-IDF
- Слави Кадиев (61679) – Naive Bayes classificator
- Милка Ферезлийска (61710) - kNN

- Светимир Игнатов (61726) - kMeans

За момента създадената система е извън https://vkusotiiki-bg.firebaseapp.com/, но данните и кодът са напълно съвместими с тези от системата, за да може да се интегрират на по-късен етап от развитието ѝ.

3.Нашето решение

3.1. TF-IDF + Vector Space Model

3.1.1. TF-IDF

TF-IDF (Term Frequency - Inverse Document Frequency) е метод, който има за цел да покаже колко е важна една дума в даден документ. Той често се използва като фактор за претегляне (weighting factor) при извличане на информация от текст. TF-IDF стойностите се увеличават пропорционално спрямо броя на срещанията на дадена думата в документа. Това се компенсира от честотата на поява на думата в корпуса (колекцията от документи) т.е. важността на някои думи се намалява спрямо това, колко често се появяват по-често като цяло.

В днешно време, TF-IDF е един от най-популярните методи за претегляне важността на дадена дума. Например, 83% от текст-базираните системи за препоръки в областта на програмните библиотеки използват този метод.

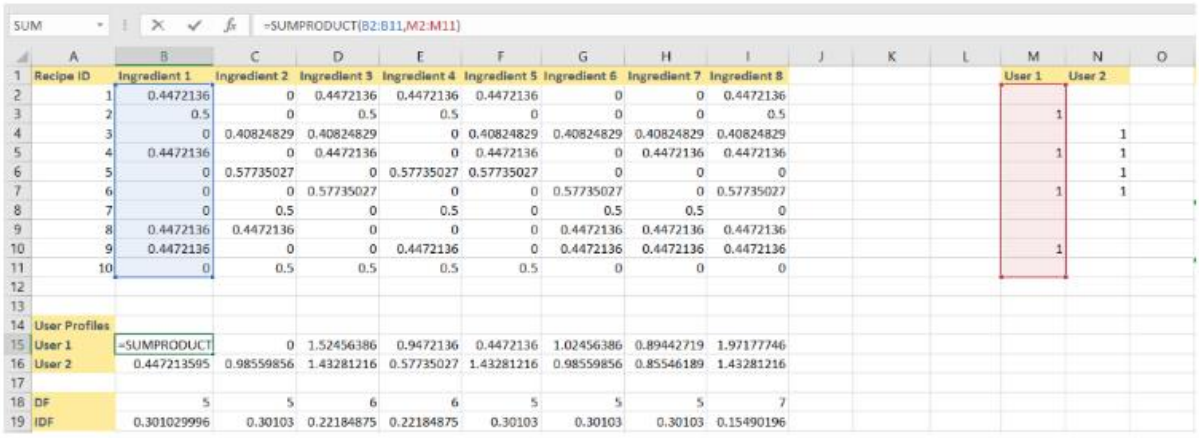
OneDrive на Michael ‣ project — td-idf

	A	B	C	D	E	F	G	H	I	M	N	O	P	Q	R	S
1	Recipe ID	Ingredient 1	Ingredient 2	Ingredient 3	Ingredient 4	Ingredient 5	Ingredient 6	Ingredient 7	Ingredient 8	User 1	User 2		TF-IDF		User 1 Pref	User 2 Pref
2	1	0,4472136	0	0,4472136	0,4472136	0,4472136	0	0	0,4472136				0,536951		62,98%	55,18%
3	2	0,5	0	0,5	0,5	0	0	0	0,5	1			0,449815		63,68%	40,13%
4	3	0	0,40824829	0,40824829	0	0,40824829	0,40824829	0,40824829	0,40824829		1		0,645388		55,36%	74,38%
5	4	0,4472136	0	0,4472136	0	0,4472136	0	0,4472136	0,4472136	1	1		0,572362		65,62%	60,97%
6	5	0	0,577350269	0	0,57735027	0,57735027	0	0	0		1		0,475684		19,90%	49,43%
7	6	0	0	0,57735027	0	0	0,577350269	0	0,57735027	1	1		0,391317		54,97%	48,30%
8	7	0	0,5	0	0,5	0	0,5	0,5	0				0,562469		39,39%	48,95%
9	8	0,4472136	0,447213595	0	0	0	0,447213595	0,4472136	0,4472136				0,607773		58,27%	54,00%
10	9	0,4472136	0	0	0,4472136	0	0,447213595	0,4472136	0,4472136	1			0,572362		67,66%	46,46%
11	10	0	0,5	0,5	0,5	0,5	0	0	0				0,522879		34,15%	58,70%
12																
13																
14	User Profiles														Cos(R4, R2)	Cos(R1, R2)
15	User 1	1,394427191	0	1,52456386	0,9472136	0,4472136	1,024563865	0,89442719	1,97177746						67,08%	89,44%
16	User 2	0,447213595	0,98559856	1,43281216	0,57735027	1,43281216	0,98559856	0,85546189	1,43281216							
17																
18	DF	5	5	6	6	5	5	5	7							
19	IDF	0,301029996	0,301029996	0,22184875	0,22184875	0,30103	0,301029996	0,30103	0,15490196							

- tf(t,d) = 1 ако продукта t се среща в рецептата d и 0 в противен случай
- user_likes(d) = 1 ако потребителят харесва рецептата d и 0 в противен случай

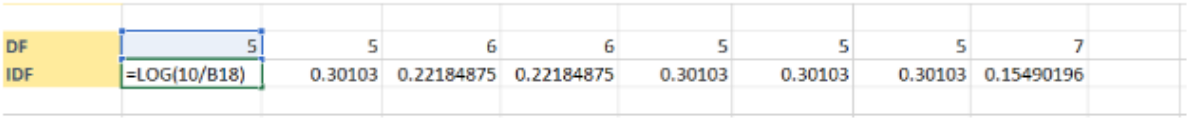
M	N
User 1	User 2
1	
	1
1	1
	1
1	1
1	

- tf(t,d) = tf(t,d) / sqrt(брой продукти в рецептата d) – нормализиране
- user_profile(t,d) = user_likes(d) . tf(d,t)



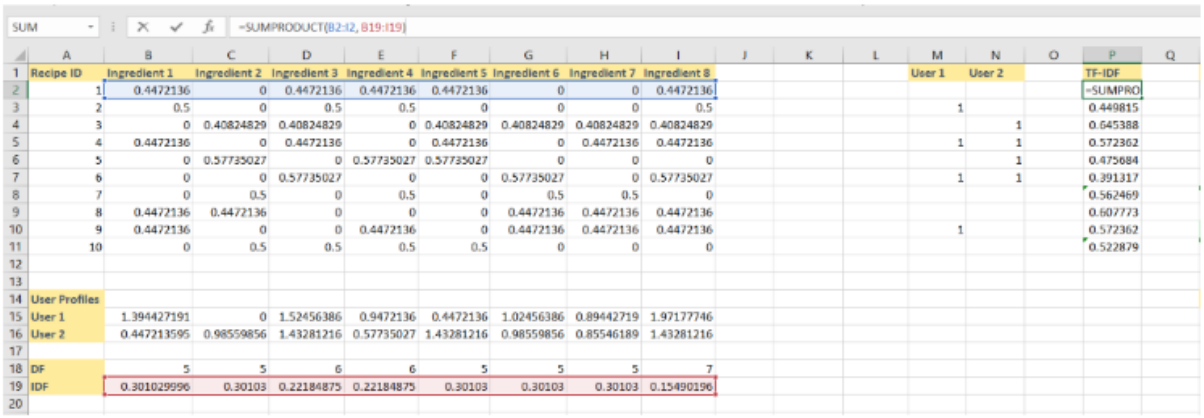
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Recipe ID	Ingredient 1	Ingredient 2	Ingredient 3	Ingredient 4	Ingredient 5	Ingredient 6	Ingredient 7	Ingredient 8				User 1	User 2	
2	1	0.4472136	0	0.4472136	0.4472136	0.4472136	0	0	0.4472136						
3	2	0.5	0	0.5	0.5	0	0	0	0.5				1		
4	3	0	0.40824829	0.40824829	0	0.40824829	0.40824829	0.40824829	0.40824829					1	
5	4	0.4472136	0	0.4472136	0	0.4472136	0	0.4472136	0.4472136				1	1	
6	5	0	0.57735027	0	0.57735027	0.57735027	0	0	0					1	
7	6	0	0	0.57735027	0	0	0.57735027	0	0.57735027				1	1	
8	7	0	0.5	0	0.5	0	0.5	0.5	0						
9	8	0.4472136	0.4472136	0	0	0	0.4472136	0.4472136	0.4472136						
10	9	0.4472136	0	0	0.4472136	0	0.4472136	0.4472136	0.4472136				1		
11	10	0	0.5	0.5	0.5	0.5	0	0	0						
12															
13															
14	User Profiles														
15	User 1	=SUMPRODUCT	0	1.52456386	0.9472136	0.4472136	1.02456386	0.89442719	1.97177746						
16	User 2	0.447213595	0.98559856	1.43281216	0.57735027	1.43281216	0.98559856	0.85546189	1.43281216						
17															
18	DF	5	5	6	6	5	5	5	7						
19	IDF	0.301029996	0.30103	0.22184875	0.22184875	0.30103	0.30103	0.30103	0.15490196						

- idf(t,D) = log(броя на всички рецепти D / (брой срещания на продукта t в множеството от всички рецепти D + 1))



DF	5	5	6	6	5	5	5	7
IDF	=LOG(10/B18)	0.30103	0.22184875	0.22184875	0.30103	0.30103	0.30103	0.15490196

- tf_idf(t,d,D) = tf(t,d) · idf(t,D)



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Recipe ID	Ingredient 1	Ingredient 2	Ingredient 3	Ingredient 4	Ingredient 5	Ingredient 6	Ingredient 7	Ingredient 8				User 1	User 2		TF-IDF	
2	1	0.4472136	0	0.4472136	0.4472136	0.4472136	0	0	0.4472136							=SUMPRO	
3	2	0.5	0	0.5	0.5	0	0	0	0.5				1			0.449815	
4	3	0	0.40824829	0.40824829	0	0.40824829	0.40824829	0.40824829	0.40824829					1		0.645388	
5	4	0.4472136	0	0.4472136	0	0.4472136	0	0.4472136	0.4472136				1	1		0.572362	
6	5	0	0.57735027	0	0.57735027	0.57735027	0	0	0					1		0.475684	
7	6	0	0	0.57735027	0	0	0.57735027	0	0.57735027				1	1		0.391317	
8	7	0	0.5	0	0.5	0	0.5	0.5	0							0.562469	
9	8	0.4472136	0.4472136	0	0	0	0.4472136	0.4472136	0.4472136							0.607773	
10	9	0.4472136	0	0	0.4472136	0	0.4472136	0.4472136	0.4472136				1			0.572362	
11	10	0	0.5	0.5	0.5	0.5	0	0	0							0.522879	
12																	
13																	
14	User Profiles																
15	User 1	1.394427191	0	1.52456386	0.9472136	0.4472136	1.02456386	0.89442719	1.97177746								
16	User 2	0.447213595	0.98559856	1.43281216	0.57735027	1.43281216	0.98559856	0.85546189	1.43281216								
17																	
18	DF	5	5	6	6	5	5	5	7								
19	IDF	0.301029996	0.30103	0.22184875	0.22184875	0.30103	0.30103	0.30103	0.15490196								
20																	

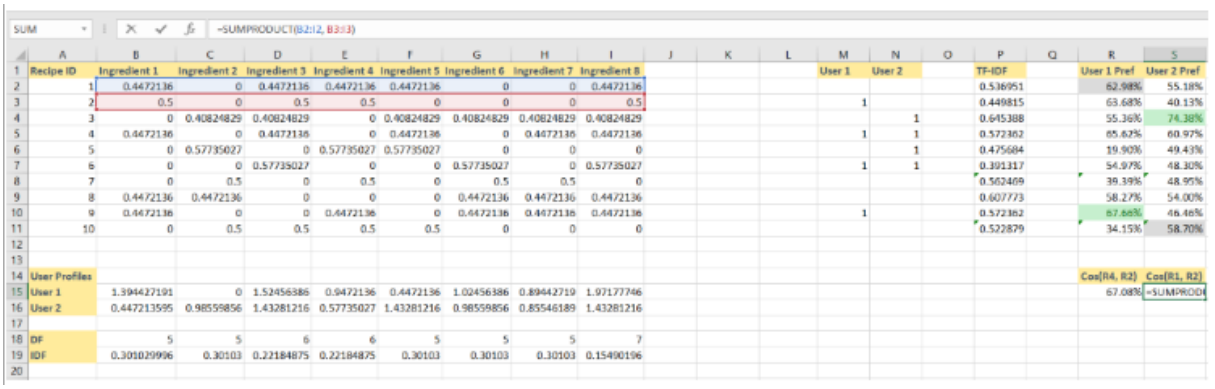
3.1.2. Vector Space Model

Vector Space Model (VSM) е алгебричен модел за представяне на текстови документи, както всякакви обекти, като вектори от идентификатори (атрибути). В нашият проект това са векторите от TF-IDF стойностите на различните рецепти.

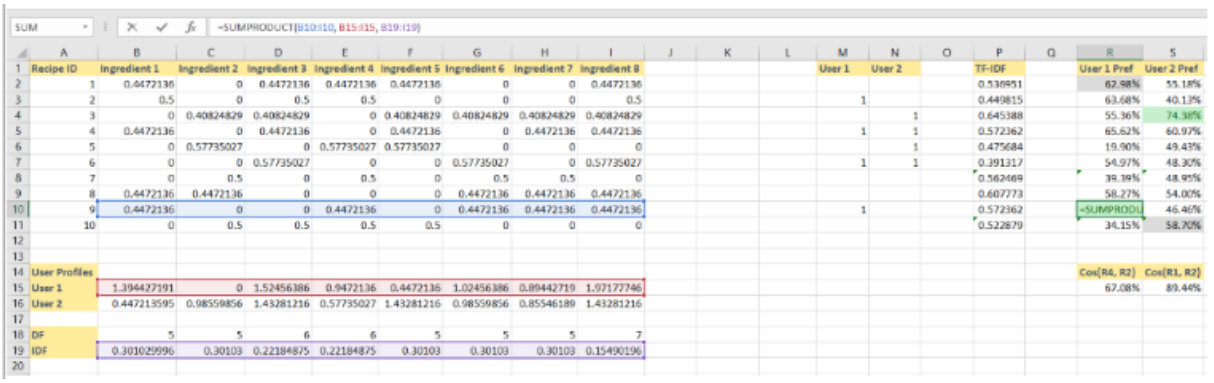
След образуването на тези вектори, те служат за намиране на близост между рецептите чрез сравняване на ъгъла между всеки два вектора. На практика, по-лесно е да се изчисли косинуса на ъгъла между тях θ, вместо самия ъгъл:

$$\cos(\theta) = \frac{\vec{a}.\vec{b}}{\left\|\vec{a}\right\|\left\|\vec{b}\right\|}$$

Поради факта, че TF-IDF векторите вече са нормализирани, то cos(θ) ще се изчисли, като се намери скаларното произведение на двата вектора.



- user_preference(t,d) = user_profile(t,d) · tf(t,d) · idf(t,D)



3.2. kNN + k-d tree

3.2.1.kNN

Един от най-простите класификационни алгоритми за обучение на машини е kNN (k Nearest Neighbours). Той е вид непараметричен алгоритъм, използван за класификация и регресия. kNN е локално търсещ мързелив алгоритъм, който търси най-близки съседи до търсения индивид в някакво пространство. Нарича се локално търсещ, защото търси индивиди само в част от пространството, и е мързелив, защото всички изчисления за отложени до момента, в който дойде новата инстанция и трябва да се класифицира.

По време на класификационната стъпка според въведено от потребителя k един некласифициран вектор (тестов индивид) се класифицира с най-често срещания клас от най-близките k обучаващи вектора (индивида). Обикновено k се избира да е нечетно число, за да може да има поне 1 клас, който се среща най-често. В случаите, когато всички класове се срещат равен брой пъти, се използва класа на най-близко разположения обучаващ индивид.

Най-често използваните метрики за изчисляване на разстояние между индивидите са Евклидово разстояние, разстояние на Хеминг (Hamming distance), коефициент на корелация на Пиърсън (Pearson correlation coefficient) и др. В контекста на текущия проект се използва Евклидово разстояние (Euclidean distance). Направени са и тестове със скаларно произведение на вектори, които дадоха добри резултати.

Недостатък от „мажоритарното гласуване“ при класификацията се появява, когато разпределението на класовете е изкривено. Например когато най-често срещаният клас доминира в предсказването за тестовата инстанция, защото той присъства твърде много пъти в множеството от съседи. Този проблем може да се превъзмогне като се създадат тегла за индивидите, взимайки предвид тяхното

разстояние до дадения индивид. Друг вариант за справяне с проблема е абстракция на представените данни, например самоорганизиращ се речник (SOM).

В настоящия проект се използва kNN, за да се открие кои рецепти са близки до рецептата, която потребителят разглежда в момента. Всяка рецепта се представя като вектор от tf стойности за всички дефинирани в системата съставки. Всеки вектор дефинира, че дадена съставка я има/няма в рецептата и има определена важност в нея. За да се определи близостта на търсената рецепта с някоя от съществуващите рецепти, които потребителят все още не е харесал, се използва Евклидово разстояние.

$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$, където $x = (x_1, x_2..., x_n)$, а $y = (y_1, y_2..., y_n)$

Създадена е и имплементация с употребата на скаларно произведение с цел да се провери точността на мерките за наличните данни. Тъй като всички рецепти-вектори са с еднаква дължина, то тя е равна на 1. Тогава скаларното произведение на два вектора, ще даде cos на ъгъла между тях. Колкото по-голяма е получената стойност за скаларното произведение, т.е. cos (a, b), то толкова по-близки са тези две рецепти. Това налага да се вземат като съвпадения първите k рецепти с най-голяма стойност на скаларното произведение с тествания индивид. За по-лесното и бързо откриване на най-близките рецепти се използва приоритетна опашка.

3.2.2.k-d tree

Системата предвижда обработката на голям брой данни и това налага прилагането на мерки за ускоряване на процеса по търсене сред тях. За целта се използва двоично разделяне на пространството. Това е процесът на рекурсивно разделяне на една сцена на две части, докато разделянето удовлетвори едно или повече условия и се реализира с помощта на k-d дървета.

k-d дървото е двоично дърво, където всеки възел е k-мерна точка. Всеки възел, различен от листо, може да се разглежда като разделяща хиперравнина, която разделя пространството на 2 части. Индивидите отляво на хиперравнината се съхраняват в лявото поддърво на този възел, а тези отдясно – в дясното поддърво. Посоката на хиперравнината се избира така че за всеки възел в дървото, свързан с едно от k на брой пространства, се създава хиперравнина, перпендикулярна на оста на тази величина. Така, например, ако за определен възел е избран "x" за ос, всички точки в поддървото с по-малка "x" стойност от възела ще се появят в лявото поддърво и всички точки с по-голяма "x" стойност ще бъдат в дясното поддърво. В такъв случай, хиперравнината ще бъде определена от x-стойността на точката и ще бъде успоредна на оста "y".

Недостатъкът на използването на k-d дървета е, че лесно може да се пропусне решение, когато търсим такова. Например – една точка може да се изключително близка по разстояние до тестовата точка, но няма да бъде взета предвид, защото се намира в съседно подпространство.

В настоящия проект съществува имплементация на k-d дърво от всички налични рецепти до определена дълбочина. Всяка рецепта в дървото се представя със своя вектор от tf стойности за продуктите. Самото дърво се построява като се намира за всеки атрибут на вектора (за всяка съставка на рецептата) медианата от стойностите за съответния атрибут на всяка рецепта. Кой атрибут се взима предвид, за да се направи избор на медиана, зависи от текущата дълбочина на дървото. Дървото се създава рекурсивно като елементът, който е избран за медиана според съответния атрибут, се поставя като възел. Спрямо възела останалите елементи се разпределят от лявата му

страна, ако са със стойност по-малка от медианата, или от дясната, ако стойността им е по-голяма. Така се създава дърво, което може да бъде дълбоко най-много log2(общ брой рецепти), тоест в случая log2(500) ~ 8.

Програмата предвижда да не се създава цялото дърво, за да може да се намери някаква част от цялото пространство, в което да се приложи kNN с помощта на скаларно произведение и да се намерят близки рецепти.

За съжаление така описаното решение не даде добри резултати. Причината се крие в това, че максималната дълбочина на дървото, което може да се създаде, е 8. В настоящите данни една рецепта се представя като вектор от tf-стойности с дължина, равна на броя на всички съществуващи съставки (малко над 400). Тоест, данните в настоящата система съдържат твърде много нули, които се разполагат във векторите. K-d дървото се създава като се сравняват първите 8 атрибута, което е крайно недостатъчно, за да се създаде дърво, което може да раздели рецептите в отделни пространства. След направени изследвания и тестове се установи, че създаването на k-d дърво не е подходящо решение за ускорение на анализа.

3.3. k-Means

Една от реализациите на Content Filtering, която използваме, е базирана на алгоритъма KMeans. Алгоритъма цели да изгради клъстъри от рецепти и при нужда да предложи на потребителя рецепти близки до посочена от него.

За реализацията му се спряхме на следните точки:

- Избиране на начални центроиди – началните центроиди се избират на случаен принцип от всички налични рецепти в базата данни. Броят им се подава като входен параметър.
- Трениране
  - ✓ избиране на нови центроиди – за всеки изграден клъстър се пресмята средната му стойност и се избира за нов центроид;
  - ✓ генериране на клъстърите – за всяка рецепта се изчислява разстоянието ѝ до новоизбраните центроиди. За изчисляване на разстоянието има две реализации: косинусова прилика и Жакардова прилика. Най-голяма прилика между две рецепти и за двата алгоритъма има при максимално големи стойности на разстоянието. Съответно всяка рецепта се причислява към клъстъра на центроида спрямо който има най-голямо разстояние;
  - ✓ край на етапа на трениране – за определяне края на итерирането, с цел оформянето на клъстърите, се използват две условия. В случай, че при изпълнението на дадена итерация елементите на всички клъстъри не се променят, то етапа на трениране се прекратява. В случай, че това не стане в обозрима граница има горна граница на броя итерации, които да извърши алгоритъма;
- Предсказване – изчислява се разстоянието на рецептата, за която ще се търсят подобни, спрямо всеки един от центроидите. На базата на тези изчисления се избира клъстъра към, който спада рецептата. От този клъстър се избират “n” на брой най-близки рецепти които се връщат като оценка от алгоритъма. Числото “n” се пресмята като корен квадратен от броя на рецептите в клъстъра, а разстоянието между рецептите пак се изчислява според един от двата типа прилика: косинусова или Жакардова;

```
Searched:
(191, 'Апетитна питка "Слънце"')
Suggested:
(447, 'Къпана питка (Анита)')
(393, 'Хляб по народна рецепта')
(401, 'Бързи содени питки')
(280, 'Великденски венец')
(218, 'Коледни колачета с мак и сусам')
(229, 'Пита с кашкавал')
(409, 'Пита с мед')
(176, 'Бързи мекички')
```

```
Searched:
(149, 'Тиквеник')
Suggested:
(457, 'Лесен и вкусен тиквеник')
(49, 'Тиквеник с аромат на канела')
(203, 'Лесен тиквеник')
(202, 'Тиквеник със сушени кайсии')
```

```
Searched:
(241, 'Карфиол с маслини на фурна')
Suggested:
(450, 'Тиквички със сос от печен чесън')
(434, 'Маринован зелен боб')
(455, 'Ароматна сьомга с 2 гарнитури')
(257, 'Супа с карфиол и броколи')
(336, 'Смесена туршия "Румяна"')
```

```
Searched:
(335, 'Супа от спанак с ориз')
Suggested:
(372, 'Супа топчета (Рафаела)')
(53, 'Постен ориз със спанак')
(298, 'Лозови сармички с агнешко')
(374, 'Постни пълнени чушки')
(315, 'Сарми с телешко месо')
(242, 'Зеленчукови кюфтенца с млечен сос')
(171, 'Мързеливи постни пълнени чушки')
```

Поради наличието на много нулеви данни, във векторите представляващи рецептите, решихме да направим експеримент използвайки Жакардово разстояние при изчисляване на разстоянията между различните рецепти. Експеримента може да се брои за успешен имайки предвид резултатите от проведените с него тестове. Стигнахме до извода, че за да имаме максимално ефективни резултати с него трябва да се смени условието за спиране на итерациите в KMeans алгоритъма. Понеже в клъстърите стават промени, много рядко се стига до условието за това елементите във всички клъстъри да не се променят между две итерации. В резултат на това твърде често се стигаше до уточнения максимален брой итерации за алгоритъма. Възможен вариант за промяна е да се търси минимална граница намеждуклъстърното пространство.

Резултати от алгоритъма при използвана Жакардово разстояние:

```
Searched:
(135, 'Клин със спанак')
Suggested:
(405, 'Спаначник')
(486, 'Сладка баница с ориз')
(86, 'Дърпана баница с кашкавал и сирене')
(394, 'Баница с късмети за Нова година')
(335, 'Супа от спанак с ориз')
(403, 'Спаначена пита')
(13, 'Плетен хляб със спанак')
(0, 'Солени тиганички с колбас')
(138, 'Яйца по панагюрски')
(397, 'Вита баница с кисело мляко')
(277, 'Хлебчета с прясна мая')
(55, 'Тутманик с кисело мляко')
(360, 'Пълнени тиквички със сирене и кашкавал')
```

```
Searched:
(127, 'Пилешко филе с лук, гъби и кашкавал')
Suggested:
(255, 'Пилешко с гъбки в 1 тиган')
(499, 'Пълнен патладжан с кашкавал')
(134, 'Пилешки сърчица със зеленчуци')
(360, 'Пълнени тиквички със сирене и кашкавал')
(151, 'Печен телешки джолан')
(272, 'Лесна гъбена супа')
(120, 'Патладжани с чесън и лук')
(180, 'Пилешки дробчета на тиган')
(189, 'Чесново пиле с кашкавал')
(495, 'Ечемичена супа с бекон')
(287, 'Миш-маш')
(289, 'Лятна зеленчукова манджа')
(110, 'Картофи в гювече')
(22, 'Пилешки гърди с 2 гарнитури')
```

3.4. Naive Bayes

Наивният Бейсов класификатор е интуитивен метод, който използва вероятностите на всеки атрибут, принадлежащ на всеки клас, за да направи предсказване. Той се базира на формулата за условна вероятност на Бейс:

$$P(c \mid d) = \frac{P(c)P(d \mid c)}{P(d)}$$

За наивния Бейсов класификатор се прави едно „наивно“ допускане, че стойността на всеки атрибут, принадлежащ на съответния клас, е независима от останалите. Това е силно предположение, но води до бърз и ефективен метод. Стойността, която се дава като вероятност на съответния атрибут от клас, се нарича условна вероятност. Умножавайки условните вероятности заедно за всеки атрибут, ние получаваме вероятността на данните за съответната инстанция на класа.

За да направим предположение, ние можем да изчислим вероятността за всяка инстанция на класа и да изберем тази стойност, която показва най-голяма вероятност. Често алгоритъма се описва, като се използва данни от тип категория, защото е лесно да се опише и изчисли чрез съотношения. По-използваната версия на алгоритъма за нашите предположения поддържа цифрови атрибути и приема, че тези стойности са нормално разпределени. В този случай се работи със параметрите на това разпределение:

- средна стойност $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$;
- стандартно отклонение $\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2}$;
- функция на плътността $f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$.

В нашия случай класовете, на които разделяме данните, са два – харесани рецепти и такива, които все още не са харесани. Алгоритъмът се пуска върху тренировъчните данни т.е. харесаните рецепти, а за тестови данни се избира част от не харесаните рецепти. Намира се вероятността за всеки атрибут от класа, а после се

умножават. Така получаваме вероятността за съответния клас. Избираме по-голямата вероятност т.е. връщаме като резултат дали потребителят би харесал тази рецепта или не.

4.Програмна реализация

4.1. TF-IDF

- **TF**

```
""" Gets the tf data from the json data """
def get_tf_data(ingredient_data, data):
    tf_data = []
    for ingredient_inner_data in ingredient_data.values():
        tf_inner_data = []
        count = sum(ingredient_inner_data)
        for item in ingredient_inner_data:
            if item == 0:
                tf_inner_data.append(0)
            else:
                tf_value = item / math.sqrt(count)
                tf_inner_data.append(tf_value)
        tf_data.append(tf_inner_data)
    return tf_data
```

- **IDF**

```
""" Gets the idf data from the json data """
def get_idf_data(ingredients_count_info, data_count):
    idf_data = []
    max_count = max(ingredients_count_info.values())
    for item in ingredients_count_info.values():
        if item == 0:
            idf_data.append(item)
        else:
            idf_data.append(math.log((data_count) / (item + 1)))
    return idf_data
```

- **TF-IDF**

```
""" Gets the tf-idf data from the tf_data and idf_data """
def get_tfidf_data(tf_data, idf_data):
    tfidf_data = list()
    for i in range(data_count):
        tf_list = tf_data[i]
        tfidf_data.append(np.dot(tf_list, idf_data))
    return tfidf_data
```

- **User likes**

```
""" Generates the users' favourite recipes with random data -
 likes for some recipes """
def generate_user_likes(data_count, binary_propabilities):
    user_likes = np.random.choice([0, 1], size=(data_count,),
p=binary_propabilities)
    return user_likes

 """ Generates the users' favourite recipes with particular recipe ids """
```

```
def generate_user_likes_by_recipes_ids(data_count, fav_recipe_ids):
    user_likes = [0] * data_count

    for recipe_id in fav_recipe_ids:
        user_likes[recipe_id] = 1

    return user_likes
```

- **User profile**

```
""" Generates the user profile using a dot product of tf value for
    each ingredient and the user's likes (favourite recipes)
"""
def generate_user_profile(tf_data, ingredients_count, user_likes):
    user_profile = []
    for i in range(ingredients_count):
        tf_list = []
        for item in tf_data:
            tf_list.append(item[i])
        user_profile_value = np.dot(user_likes, tf_list)
        user_profile.append(user_profile_value)

    return user_profile
```

4.2. Vector Space Model

- **User preference**

```
""" Generates the user preferences for the recipe using a dot product
    of user profile data, idf data and tf data
"""
def generate_user_pref(data_count, tf_data, idf_data, user_profile):
    user_pref = []

    for i in range(data_count):
        tf_list = tf_data[i]
        user_pref_value = round(sum(p * q * t for p, q, t in zip(user_profi
le, idf_data, tf_list)), 2)
        user_pref.append(user_pref_value)

    return user_pref
```

- **Близки рецепти до най-харесвана рецепта за даден потребител**

```
""" Gets the n closest recipes to the best prefered recipe with a given bes
t_recipe_pref_index """
def n_closest_recipes_to_best_recipe_pref(best_recipe_count, tf_data, user_
likes, best_recipe_pref_index):
    recipes_diff = dict()
    n_closest_recipe_indexes = []
    best_recipe_pref_tf_values = tf_data[best_recipe_pref_index]
    index = 0

    for tf_values in tf_data:
        if tf_values != best_recipe_pref_tf_values:
            recipes_diff[index]
= float("{0:.2f}".format(round(np.dot(tf_values, best_recipe_pref_tf_values
), 2)))
            index += 1
```

```
    recipe_diff_modified =
{index : item for index, item in recipes_diff.items() if user_likes[index]
== 0}
    n_closest_recipe_indexes = nlargest(best_recipe_count, recipe_diff_modi
fied, key=recipe_diff_modified.get)
    n_closest_recipes =
{index : recipes_diff[index] for index in n_closest_recipe_indexes}

    return n_closest_recipes, n_closest_recipe_indexes
```

- **Най-вероятните рецепти, които биха се харесали на даден потребител**

```
""" Gets the n largest user preferences """

def get_n_largest_user_pref(best_user_pref_count, best_recipe_pref_index,
user_pref, user_likes):

    max_user_pref = user_pref[best_recipe_pref_index]

    modified_user_pref = {index : float("{0:.2f}".format(round(item /
max_user_pref, 2))) for index, item in enumerate(user_pref) if
user_likes[user_pref.index(item)] == 0}

    n_largest_user_pref_indexes = nlargest(best_user_pref_count,
modified_user_pref, key=modified_user_pref.get)

    n_largest_user_pref = {index : modified_user_pref[index] for index in
n_largest_user_pref_indexes}

    return (n_largest_user_pref_indexes, n_largest_user_pref)
```

4.3. kNN + k-d tree

- **Tree**

Използва се named tuple, защото така по-лесно могат да се достъпват възела на дървото и двете му деца:

```
class Tree(namedtuple('Tree', 'root left_child right_child')):
    def __repr__(self):
        return pformat(tuple(self))
```

- **K-d tree**

K-d дървото се създава рекурсивно, като първо се строи лявото поддърво, а после и дясното:

```
def kdtree(point_list, depth=0):

    if not len(point_list):
        return None

    k = len(point_list[0])

    # Select attribute based on depth
    attr = depth % k

    # Sort point list and choose median as tree root
    point_list.sort(key=itemgetter(attr))
```

```
    median = len(point_list) // 2  # choose median

    # Create tree and construct subtrees
    return Tree(
        root=point_list[median],
        left_child=kdtree(point_list[:median], depth + 1),
        right_child=kdtree(point_list[median + 1:], depth + 1)
    )
```

- **Calculate Euclidean distance**

Основната част от kNN алгоритъма се съхранява във функцията generate_queue, която създава приоритетна опашка, за да може да се съхранят и извлекат възможно най-бързо стойнстите за разстоянията между рецептите с помощта на Евклидово разстояние:

```
def calculate_euclid_dist(item, recipe):
    current_sum = 0
    for i in range(len(recipe)):
        current_sum += (float(item[i]) - float(recipe[i])) ** 2
    return sqrt(current_sum)
```

- **kNN**

След като се вземат първите k рецепти с най-малко разстояние до текущо разглежданата рецепта, се избира клас. В текущата програма възможните стойности за класа на рецепта са две - 0 (не си приличат) и 1 (приличат си). Функцията връща резултата за това дали 2 рецепти си приличат или не.

```
def generate_queue(item, trainers_ids, tf_data, k, user_likes):
    queue = []

    for trainer_id in trainers_ids:
        recipe = tf_data[trainer_id]

        euclid_dist = calculate_euclid_dist(item, recipe)

        heappush(queue, (euclid_dist, trainer_id))

    queue.sort()
    new_elements = queue[:k]

    most_spread_class = group_classes(new_elements, user_likes)

    if len(set(most_spread_class.values())) ==
len(set(most_spread_class.keys())):
        found_class = sorted(list(most_spread_class.items()), key=lambda x:
x[1])[-1][0]
    else:
        k = max(most_spread_class.values())
        classes = [item for item, cl in most_spread_class.items() if cl ==
k]
        closest_cl = heappop(new_elements)
        found_class = closest_cl if closest_cl in classes else
choice(classes)

    return found_class
```

- **main**

Накрая, за да може да се тества създадената програма, се използва cross валидация. Това означава, че цялото пространство се разделя на 2 части - обучаващи и тестови. С помощта на train_recipe_ids, test_recipe_ids = train_test_split (sample(range(len(data)), set_length)) се създават 2 произволни множества от рецепти, с които съответно се обучава и тества kNN.

```
train_recipe_ids, test_recipe_ids = train_test_split
(sample(range(len(data)), set_length))
```

4.4. k-Means

- Генериране на начални центроиди:

```
def get_initial_centroids_indeces(filtered_tf_data, k):
    recipe_indeces = list(filtered_tf_data.keys())
    taken = []
    centroids = {}

    for x in range(0, k):
        elem = random_element(recipe_indeces, taken)
        taken.append(elem)
        centroids[x] = filtered_tf_data[elem]

    return centroids
```

- Функция реализираща реализираща content filtering-а:

```
def kmeans(tf_data, filtered_tf_data, k, best_recipe_pref_index,
train_likes):
    centroids = get_initial_centroids_indeces(filtered_tf_data, k)
    clusters = build_clusters(filtered_tf_data, centroids, k)
    iteration = 1

    while iteration < 40:
        centroids = next_centroids(clusters, filtered_tf_data)
        old_clusters = clusters
        clusters = build_clusters(filtered_tf_data, centroids, k)
        iteration += 1

        if static_clusters(old_clusters, clusters):
            print("Clusters are static after {}
iterations".format(iteration))
            break

    predicted_indeces = predict_cluster(tf_data[best_recipe_pref_index],
centroids, clusters)

    predicted_tf_data = list(map((lambda x: tf_data[x]),
predicted_indeces))

    k_for_closest = int(math.sqrt(len(predicted_indeces)))

    print("Searching for {} closest recipes".format(k_for_closest))

    k_closest = find_closest(tf_data, predicted_indeces, k_for_closest,
tf_data[best_recipe_pref_index], train_likes, best_recipe_pref_index)

    return k_closest
```

- Изграждане на клъстъри около центроиди:

```
def build_clusters(filtered_tf_data, centroids, k):
    clusters = {}
    all_indeces = list(filtered_tf_data.keys())

    """ Initialize clusters """
    for num in list(centroids.keys()):
        clusters[num] = []

    for index in filtered_tf_data:
        recipe = filtered_tf_data[index]
        clusters[choose_centroid(recipe, centroids)].append(index)

    return clusters
```

- Избиране на най-близкия центроид за дадена рецепта:

```
def choose_centroid(recipe, centroids):
    centroids_indeces = list(centroids.keys())
    proximities = list(map((lambda c: {
        "centroid": c,
        "proximity": proximity(recipe, centroids[c])
    }), centroids_indeces))

    return max_by_prop(proximities, "proximity")["centroid"]
```

- Намиране на косинусова прилика (cosine similarity):

```
def proximity(recipe, alternate_recipe):
    return np.dot(recipe, alternate_recipe)
```

- Жакардово разстояние (Jaccard distance):

```
def jaccard_distance(recipe, centroid):
    recipe_ingredients = [i for i, e in enumerate(recipe) if e != 0]
    centroid_ingredients = [i for i, e in enumerate(centroid) if e != 0]
    intersect =
set(recipe_ingredients).intersection(set(centroid_ingredients))
    union = set(recipe_ingredients).union(set(centroid_ingredients))

    return len(intersect) / len(union)
```

- Намиране на нови центроиди:

```
def next_centroids(clusters, filtered_tf_data):
    centroids = {}

    for cluster_id in clusters:
        cluster_recipes_indeces = list(filter((lambda x: x in
clusters[cluster_id]),
            list(filtered_tf_data.keys())))
        cluster_recipes = list(map((lambda x: filtered_tf_data[x]),
cluster_recipes_indeces))
```

- Проверка за наличието на движение на рецепти между клъстърите:

```
def static_clusters(old, new):
    clusters = list(old.keys())
```

```
    static_clusters = list(filter(lambda x: set(old[x]) == set(new[x]),
clusters))

    return len(static_clusters) == len(clusters)
```

- Намира рецептите от клъстъра към, който принадлежи дадена рецепта:

```
def predict_cluster(recipe, centroids, clusters):
    centroid = choose_centroid(recipe, centroids)

    return clusters[centroid]
```

- Намиране на най-близките “k” рецепти от даден клъстър до конкретна рецепта:

```
def find_closest(tf_data, recipes, k, recipe, train_likes, searched_index):
    proximities = {}

    for recipe_index in recipes:
        if (recipe_index not in train_likes and recipe_index !=
searched_index):
            proximities[recipe_index] = proximity(tf_data[recipe_index],
recipe)

    return nlargest(k, proximities, key=proximities.get)
```

4.5. Naive Bayes

- Групиране на tf стойностите спрямо класа им (харесванията на потребителя – 1 или 0)

```
def group_tf_data_by_class(tf_data, user_likes):
    grouped_tf_data = {}
    grouped_tf_data[0] = list()
    grouped_tf_data[1] = list()

    for tf_index, tf_value in enumerate(tf_data):
        tf_value_like = user_likes[tf_index]
        grouped_tf_data[tf_value_like].append(tf_value)

    return grouped_tf_data
```

- Намиране на стандартното отклонение на данните за всеки атрибут (продукт)

```
def standard_deviation(tf_values):
    avg = mean(tf_values)
    variance = sum([pow(x - avg,2) for x in tf_values])/ float
(len(tf_values)-1)
    deviation = math.sqrt(variance)
    return deviation
```

- Намиране на вероятността за всеки атрибут (продукт)

```
def get_attribute_probability(input_value, mean, standard_deviation):
    if standard_deviation != 0:
        input_value_diff = input_value - mean
        exponent = math.exp(-((input_value_diff * input_value_diff) / (2  *
standard_deviation * standard_deviation)))
```

```
        probability = (1 / (math.sqrt(2*math.pi) * standard_deviation)) *
exponent
    else:
        probability = 1

    return probability
```

- Намиране на вероятността за атрибутите чрез статистическите им данни (очакване/средно и стандартно отношение) спрямо класа

```
def get_class_probabilities(class_statistical_data, input_data):
    probabilities = dict()

    for class_id, class_statistical_value in
class_statistical_data.items():
        probabilities[class_id] = 1
        for index in range(len(class_statistical_value)):
            input_value = input_data[index]
            mean, standard_deviation = class_statistical_value[index]
            probabilities[class_id] *=
get_attribute_probability(input_value, mean, standard_deviation)

    return probabilities
```

- Намиране на статистическите данни (очакване/средно и стандартно отношение) на атрибутите (продуктите)

```
def get_statistical_data(dataset):
    statistical_data = [(mean(attribute), standard_deviation(attribute))
for attribute in zip(*dataset)]
    return statistical_data
```

- Намиране на статистическите данни (очакване/средно и стандартно отношение) на атрибутите спрямо класа

```
def get_statistical_data_by_class(tf_data, user_likes):
    statistical_data = dict()
    group_tf_data = group_tf_data_by_class(tf_data, user_likes)
    #print("group_tf_data: ", group_tf_data)

    for class_value, tf_value in group_tf_data.items():
        statistical_data[class_value] = get_statistical_data(tf_value)

    return statistical_data
```

- Намиране на вероятния клас на зададените входни данни

```
def predict_likes(class_statistical_data, input_data):
    probabilities = get_class_probabilities(class_statistical_data,
input_data)
    print("probabilities :", probabilities)

    if probabilities[0] > probabilities[1]:
        class_type = 0
    else:
        class_type = 1

    return class_type
```

- Намиране на вероятния клас спрямо тестовите данни

```
def get_predictions(class_statistical_data, test_data):
    predictions = []

    for test_likes in range(len(test_data)):
        result = predict_likes(class_statistical_data,
test_data[test_likes])
        predictions.append(result)

    return predictions
```

- Намиране на точността на предсказаните класове спрямо тестовите данни

```
def get_accuracy(test_likes, predictions):
    correct_predictions_count = 0

    for index in range(len(test_likes)):
        if test_likes[index] == predictions[index]:
            correct_predictions_count += 1

    accuracy = correct_predictions_count / float(len(test_likes))

    return accuracy * 100.0
```

5. Заключение

До момента имаме реализирани 4 различни алгоритъма, които може да използваме в нашата система, разбира се всеки със своите предимства и недостатъци.

В крайна сметка до момента имаме реализирани 4 различни алгоритъма, които може да използваме в нашата система, взимайки предвид предимствата и недостатъците им.

В процеса на имплементация открихме, че когато хората харесват рецепти без да има някаква връзка между тях, това понижава успеваемостта на алгоритмите. Въпреки това успяхме да получим много добри резултати с Наивния Бейсов класификатор. Той се справя изключително добре с около **90%** средна познаваемост.

Почти на същото ниво успява да познае и **TF-IDF** алгоритъма - със средна успеваемост от малко над **82%**.

kNN успя да постигне нелош резултат - около и над **78%** средно при 10 последователни пускания с различни данни с помощта на валидация на данните.

KMeans също успява да познае доста от близките до дадена рецепта рецепти като варира в резултата си около **50-60%.**

В бъдеще може да се позовем на изпълнението на следните задачи:

- Събиране и анализиране на по-голям набор от данни за рецепти
- Използване на повече информация за действията на потребителите:
  - разглеждани рецепти
  - кликвания по линкове
  - кликвания на страниците
- Използване на анкети за предпочитани рецепти/ястия на потребителите

6. Литература

6.1. Recommender systems:

- http://www.fxpal.com/publications/FXPAL-PR-06-383.pdf
- http://recommender-systems.org/information-filtering/
- https://www.toptal.com/algorithms/predicting-likes-inside-a-simple-recommendation-engine
- https://en.wikipedia.org/wiki/Recommender_system
- http://blog.untrod.com/2016/06/simple-similar-products-recommendation-engine-in-python.html
- https://en.wikipedia.org/wiki/Tf%E2%80%93idf
- https://en.wikipedia.org/wiki/K-means_clustering

6.2. Имплементации с Python:

- http://blog.untrod.com/2016/06/simple-similar-products-recommendation-engine-in-python.html
- https://www.analyticsvidhya.com/blog/2016/06/quick-guide-build-recommendation-engine-python/
- http://muricoca.github.io/crab/tutorial.html
- https://www.codementor.io/jadianes/tutorials/build-data-products-django-machine-learning-clustering-user-preferences-du107s5mk
- http://online.cambridgecoding.com/notebooks/eWReNYcAfB/implementing-your-own-recommender-systems-in-python-2
- http://dataaspirant.com/2015/05/25/collaborative-filtering-recommendation-engine-implementation-in-python/

6.3. Информация за TF-IDF & Space Vector Model

- https://en.wikipedia.org/wiki/Tf%E2%80%93idf
- http://www.tfidf.com/
- https://en.wikipedia.org/wiki/Vector_space_model
- https://www.youtube.com/watch?v=ZEkO8QSlynY – обяснение за Space Vector Model
- https://www.youtube.com/watch?v=E3shpvJUZ84
- https://www.analyticsvidhya.com/blog/2015/08/beginners-guide-learn-content-based-recommender-systems/ - пример за TF-IDF за предлагане на книги
- https://code.google.com/archive/p/tfidf/

6.4. Информация за k-Means алгоритъма:

- https://en.wikipedia.org/wiki/K-means_clustering
- http://www.galvanize.com/blog/introduction-k-means-cluster-analysis/
- http://www.onmyphd.com/?p=k-means.clustering
- http://aimotion.blogspot.bg/2009/08/data-mining-in-practice-learn-about-k.html
- http://www.ibm.com/developerworks/library/os-recommender2/
- http://www.johnwittenauer.net/machine-learning-exercises-in-python-part-7/

6.5.Информация за Naive Bayes

- http://machinelearningmastery.com/naive-bayes-classifier-scratch-python/
- https://en.wikipedia.org/wiki/Naive_Bayes_classifier
- http://scikit-learn.org/stable/modules/naive_bayes.html
- https://www.analyticsvidhya.com/blog/2015/09/naive-bayes-explained/
- http://machinelearningmastery.com/naive-bayes-for-machine-learning/

6.6. Информация за kNN + k-d tree

- https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
- https://en.wikipedia.org/wiki/K-d_tree

6.7. Hybrid Recommendation Filtering:

- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.789&rep=rep1&type=pdf
- https://pdfs.semanticscholar.org/

ПРИЛОЖЕНИЕ 1

Recipe ID	Ingredient 1	Ingredient 2	Ingredient 3	Ingredient 4	Ingredient 5	Ingredient 6	Ingredient 7	Ingredient 8	User 1	User 2	TF-IDF	User 1 Pref	User 2 Pref
1	0,4472136	0	0,4472136	0,4472136	0,4472136	0	0	0,4472136			0,536951	62,98%	55,18%
2	0,5	0	0,5	0,5	0	0	0	0,5	1		0,449815	63,68%	40,13%
3	0	0,40824829	0,4082483	0	0,40824829	0,4082483	0,4082483	0,4082483		1	0,645388	55,36%	**74,38%**
4	0,4472136	0	0,4472136	0	0,4472136	0	0,4472136	0,4472136	1	1	0,572362	65,62%	60,97%
5	0	0,57735027	0	0,5773503	0,57735027	0	0	0		1	0,475684	19,90%	49,43%
6	0	0	0,5773503	0	0	0,5773503	0	0,5773503	1	1	0,391317	54,97%	48,30%
7	0	0,5	0	0,5	0	0,5	0,5	0			0,562469	39,39%	48,95%
8	0,4472136	0,4472136	0	0	0	0,4472136	0,4472136	0,4472136			0,607773	58,27%	54,00%
9	0,4472136	0	0	0,4472136	0	0,4472136	0,4472136	0,4472136	1		0,572362	**67,66%**	46,46%
10	0	0,5	0,5	0,5	0,5	0	0	0			0,522879	34,15%	58,70%
User Profiles												**Cos (R4, R2)**	**Cos (R1, R2)**
User 1	1,394427191	0	1,5245639	0,9472136	0,4472136	1,0245639	0,8944272	1,9717775				67,08%	89,44%
User 2	0,447213595	0,98559856	1,4328122	0,5773503	1,43281216	0,9855986	0,8554619	1,4328122					
DF	5	5	6	6	5	5	5	7					
IDF	0,301029996	0,30103	0,2218487	0,2218487	0,30103	0,30103	0,30103	0,154902					

ПРИЛОЖЕНИЕ 2

Декларация за плагиатство

Тази курсова работа е наша работа, като всички изречения, илюстрации и програми от други хора, са изрично цитирани.

Тази курсова работа или нейна версия не са представени в друг университет или друга учебна институция.

Разбирам, че ако се установи плагиатство в работата ми ще получа оценка “Слаб”.

Слави Радостинов Кадиев …….

Милка Милчова Ферезлийска ……...

Христиан Венциславов Димитров…….

Светимир Цветанов Игнатов……..