Matchning av Produkter mellan Produktdatabaser

Mattias Frisk

Department of Electrical and Information Technology
Lund University

Advisor: Rasmus Sundberg och Fredrik Andersson

12 september 2013

Printed in Sweden
E-huset, Lund, 2013

Abstract

This master thesis examines how to make a computer automatically find identical products between product descriptions from different online stores. Solutions are done by using a known technique for matching text documents: tf-idf and modifying it to exploit the unique structure of product descriptions from online stores. The thesis focuses on making as accurate solutions as possible. The modified tf-idf solutions in this thesis vastly outperforms the regular tf-idf in terms of accuracy.

Keywords: tf-idf, matching

Förord

Jag vill ge ett stort tack till mina handledare på Apptus, Rasmus Sundberg och Fredrik Andersson för deras feedback och hjälp under hela examensarbetet.

Innehåll

1 Introduktion **1**
- 1.1 Bakgrund . . . 1
- 1.2 Problembeskrivning . . . 3
- 1.3 Metod . . . 3
- 1.4 Begränsningar . . . 5
- 1.5 Teori . . . 5

2 Implementering **9**
- 2.1 Testdata . . . 9
- 2.2 Parsning . . . 10
- 2.3 Metoder . . . 11

3 Resultat **17**

4 Diskussion **19**
- 4.1 Slumpmatchning . . . 19
- 4.2 Tf-idf . . . 20
- 4.3 Tf-idf med bijektiv matchning . . . 22
- 4.4 Boolesk tf-idf med bijektiv matchning . . . 23
- 4.5 Kolumnbaserad tf-idf . . . 23
- 4.6 Bijektiv kolumnbaserad . . . 24
- 4.7 Strikt kolumnbaserad . . . 24
- 4.8 Normaliserad strikt kolumnbaserad . . . 26
- 4.9 Strikt kolumnbaserad med tf-idf vikter . . . 26

5 Slutsatser **29**
- 5.1 Slumpmatchning . . . 29
- 5.2 Tf-idf . . . 29
- 5.3 Tf-Idf med bijektiv matchning . . . 30
- 5.4 Booleansk tf-idf med bijektiv matchning . . . 30
- 5.5 Kolumnbaserad tf-idf . . . 30
- 5.6 Bijektiv kolumnbaserad . . . 30
- 5.7 Strikt kolumnbaserad . . . 30
- 5.8 Normaliserad strikt kolumnbaserad . . . 31

5.9 Strikt kolumnbaserad med tf-idf vikter 31
5.10 Rekommendationer 31

6 Appendix resultat **33**
6.1 Slumpmatchning 33
6.2 Tf-idf 34
6.3 Tf-idf med bijektiv matchning 34
6.4 Booleansk tf-idf med bijektiv matchning 35
6.5 Kolumnbaserad tf-idf 36
6.6 Bijektiv kolumnbaserad 36
6.7 Strikt kolumnbaserad 37
6.8 Normaliserad strikt kolumnbaserad 39
6.9 Strikt kolumnbaserad med tf-idf vikter 40

Litteraturförteckning **43**

Kapitel 1

Introduktion

1.1 Bakgrund

Nätbutiker har oftast väldigt stora produktdatabaser över vad de säljer och i dessa kan det samma produkt vara inlagd två eller flera gånger med liten variation på produktbeskrivningen. Att hitta samma eller liknade produkter är svårt och tar lång tid att göra manuellt över produktdatabaser som har tiotusentals produkter. Det finns även en marknad för företag som sammanställer eller analyserar data för produkter från flera olika nätbutiker (t.ex. Prisjakt.se eller GfK). Företag som sammanställer produktinformation från flera olika nätbutiker måste kunna identifiera samma produkt över två produktdatabaser och matcha ihop dem.

Det finns inte alltid unika ID som EAN-kod för att trivialt kunna se vilka produkter som är likadana. EAN-kod står för "European Article Number" och är en standard för att märka varor som används över hela Europa [1].

Även om EAN-kod eller något liknade finns tillgängligt är det inte alltid säkert att de är korrekta. I detta examensarbete kommer det endast diskuteras hur man matchar par över två olika databaser och inte i enskilda.

I nätbutiker beskrivs ofta en produkt med ett antal kolumner och kolumnvärden. Nedanför är fyra tabeller med produktinformation från två olika nätbutiker som vi kallar A & B. Kolumnerna är till vänster i tabellen och kolumnvärdena till höger.

Nätbutik A, produkt 1

Namn	Apple MacBook Air 11" (Modell 5) 128GB
Färg	Silver
4G stöd	Nej
Grafikkort	Intel HD Graphics 4000
Processor hastighet (GHz)	1.7
RAM minne (GB)	4
Vikt (g)	1080

Nätbutik A, produkt 2	
Namn	Assassin's Creed II
Format	MAC
Antal spelare	1
Genre	Action
Releasedatum	2010-03-04
Åldersgräns	18 år
Distributör	Ubisoft

Nätbutik B, produkt 1	
Namn	MacBook Air 11.6" MD223
Allmän egenskap - Färg	Silver
Grafik - Grafikkortsmodell	Intel HD Graphics 4000
Processor hastighet	1.7 GHz
RAM	4GB
Vikt	1.08kg

Nätbutik B, produkt 2	
Namn	Assassins Creed 2
Kompatibel med	PC
Typ av spel	Action
Releasedatum	04-03-2010
Åldersgräns	18

Syftet med examensarbetet är att utveckla ett program som kan läsa in produkter från två godtyckliga nätbutiker där informationen för varje produkt beskrivs som i en relationsdatabas, och sedan på ett så optimalt sätt som möjligt skriva ut en sorterad lista med par (potentiellt ekvivalenta produkter) mellan de två nätbutikerna. Programmet ska sortera paren i listan så att de par som programmet tror är mest sannolikt är samma kommer först. Examensarbetet kommer även undersöka kvantitativa sätt för att mäta hur optimal en lösning är jämfört med en manuellt fördefinierad lista av par med ekvivalenta produkter. Hur man på ett bra sätt ska avgöra hur optimal en lösning är också ett problem som kommer tas upp i examensarbetet.

I exemplet ovan ser man uppenbart att produkten i "Nätbutik A, produkt 1" är samma som "Nätbutik B, produkt 1" men att "Nätbutik A, produkt 2" och "Nätbutik B, produkt 2" inte är det även om de är lika då de är till två olika format, PC och Mac. I detta fallet skulle en optimal lösning skriva ut "Produkt 2 i nätbutik A matchar produkt 1 i nätbutik B"

1.2 Problembeskrivning

Givet en databas av produktinformation från en nätbutik som beskriver data genom en matris $A = a(i, j)$ där värdemängden för varje element är en mängd termer (inklusive tomma mängden). En produkt definieras som en radvektor i matrisen $a(i, k)$ för ett konstant k. Varje kolumnvektor har också en korresponderande mängd termer som beskriver vad för data kolumnvektorn innehåller.

Med termer menas enskilda ord, datum eller nummer och antal gånger de förekommer. Strängen "Borderlands (Xbox 360) till Xbox 360" skulle ge termerna $\{(\text{Borderlands}, 1), (360, 2), (\text{till}, 1)(\text{Xbox}, 2)\}$

För två produktdatabaser $A_1 = a_1(i_1, j_1)$ och $A_2 = a_2(i_2, j_2)$ skall programmet för varje radvektor mellan de två databaserna ge ett "likhetsvärde" inom $\{e : e \in \mathbf{R} \wedge 0 \leq e \leq 1\}$ där 1 är att paren är som mest lika medan 0 är att de är som minst lika. Programmet skall sedan göra en samling av varje potentiellt par av produkter mellan de två databaserna som $\{(i_1, i_2, e)\}$. Samlingen ska stämma så bra överens som möjligt med en fördefinierad lista med par av ekvivalenta produkter. Den fördefinierade listan är byggd manuellt. Hur man ska testa hur bra en lösning är kommer också att undersökas i examensarbetet

1.3 Metod

Under examensarbetet undersöktes först lämpliga sätt att mäta hur bra en lösning var. När detta var evaluerat och en specifik mätningsmetod var vald testades en del olika lösningar. Lösningarna som testades förbättrades kontinuerligt genom analys av vilken typ av data de matchade bättre eller sämre. Huvudsakligen implementerades samt testades alla lösningar i C# 4.5 [2].

För att testa bra hur en lösning är så optimerar man över ett tröskelvärde k så att listan av par $P = \{(i_1, i_2, e) : e < k\}$ är så lik som möjligt gentemot den manuellt fördefinierad listan av par F. Tröskelvärdet k är ett värde inom $0 \ldots 1$. En naiv metod för att mäta hur optimal en lösning är vore att helt enkelt räkna andelen av par i F som överensstämmer med den fördefinierade listan

$$\frac{|\{(i_1, i_2, e) \in P : (i_1, i_2) \in F\}|}{|F|}$$

Nackdelen är att en lista med alla möjliga kombinationer av par får full poäng. En bättre lösning hade varit att ge poäng efter

$$\frac{\sum_{(i_1,i_2)\in P} f(i_1, i_2)}{2|P|} + 0.5$$

där f är definierat som

$$f(i_1, i_2) = \begin{cases} 1 & \text{if } (i_1, i_2) \in F \\ -1 & \text{if } (i_1, i_2) \notin F \end{cases}$$

Dividerar med 2 och adderar med 0.5 för att normalisera värdemängden till $[0 \dots 1]$. Dock fungerar inte den lösningen helt optimalt då den ger lika mycket poäng till en lösning som hittar $|F|$ antal par varav hälften är korrekta (skulle ge 0 poäng) gentemot en lösningen som inte hittar några par alls. En lösning som hittar korrekt par hälften av gångerna är självfallet bättre än en lösning som inte hittar något överhuvudtaget. Man hade kunnat tänka sig att justera poängsättningen (1 och -1) så att en korrekt matchning ger mer poäng för att lösa problemet. Ett sätt att skriva det mer generellt är att definiera f som

$$f(i_1, i_2) = \begin{cases} p & \text{if } (i_1, i_2) \in F \\ -q & \text{if } (i_1, i_2) \notin F \end{cases}$$

där

$$\frac{\sum_{(i_1, i_2, e) \in P} f(i_1, i_2)}{(p+q)|P|} + \frac{q}{p+q}$$

Om man sätter $p = 2, q = 1$ blir en lösning som matchar en tredjedel korrekt lika bra som en lösning som inte matchar något alls. Oavsett vad man sätter för värden på p och q kommer man ha problematiken att en tom lösning får lika mycket poäng som en lösning med mer än en korrekt matchning. Formeln tar heller inte hänsyn till hur stor andel av alla element faktiskt är korrekta vilket också är relevant då det är svårare att hitta en korrekt matchning utav 100 produkter än en korrekt matchning utav 10 produkter.

Istället används F_1-poäng för att mäta hur bra en lösning är, F_1-poäng är definierat som

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Där precision är hur stor andel av de matchade paren i P som är i den fördefinierade listan F. Recall är hur stor andel av par från den fördefinierade listan F som är med i P. F_1-poäng är en välbeprövad metod och används i många liknade sammanhang till att mäta hur bra en lösning presterar [5]. F_1-poäng förklaras med mer detalj i sektion 1.5.4.

Man måste även ta upp hur man väljer värdet tröskelvärdet k. Man skulle kunna tänka sig att man helt enkelt sätter det till ett värde t.ex. 0.5 (vilket visade sig vara ett väldigt bra värde för de två produktdatabaserna som användes under examensarbetet). Korsvalidering används för att bestämma tröskelvärdet k då man vill mäta hur bra en lösning presterar med okänd eller ny data [3]. Hur korsvalidering fungerar förklaras i sektion 1.5.5. Ens data delas således upp i två delar, en

träningsdel som man optimerar tröskelvärdet över och en testdel där man testar hur hög F_1-poäng man får med tröskelvärdet från träningsdelen.

Man vill att produkterna som väljs till tränings- och testdata väljs slumpmässigt och för att göra detta sorterar man produkterna för varje produktdatabas slumpmässigt till en lista. Man går sedan iterativt genom den slumpade listan där man lägger produkterna som antingen träningsdata eller testdata. För en tvåfaldig korsvalidering så läggs varannan produkt från den slumpade listan som testdata och de resterande som träningsdata. Man vill inte att den slumpade listan blir annorlunda varje gång man testar en lösning, då små varianter kan ge metoder för låga eller höga poäng beroende på hur fördelaktig den slumpade listan blev. För att skapa en slumplista används en hash-funktion på strängen man får utav att summera alla kolumnvärden för en produkt. $s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-1}$ Hash-funktionen används bland annat i Java [4]. Man hade även kunnat tänka sig att använda en pseudo-random generator med samma "seed" men om man hade lagt till nya produkter i en av databaserna hade det potentiellt ändrat den slumpade listan drastiskt, då om en produkt läggs till först får alla produkter en helt ny placering i den slumpade listan.

1.4 Begränsningar

Lösningen ska endast fungera på välformaterade produktdatabaser där stavfel och strukturändringar för produkter inom samma kategori är sällsynta. Examensarbetet kommer huvudsakligen att fokusera på att optimera och mäta olika lösningar så att de matchar så bra som möjligt automatiskt och inte som om de vore ett hjälpverktyg för att hitta alla ekvivalenta produkter. Examensarbetet kommer endast fokusera på textbeskrivningar av produkter och inte eventuella bilder.

1.5 Teori

Här beskrivs de olika algoritmer eller tekniker som har använts i examensarbetet

1.5.1 Tf-idf

Term frequency-inverse document frequency är en metod för att avgöra hur viktigt ett ord (eller "term" som det kallas här) är i ett textdokument relativt en samling av textdokument. Tillämpningar där tf-idf används är t.ex. sökmotorer eller textanalys [8].

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

Där t är en term, d är ett textdokument (samling av termer t) och D är en samling av alla textdokument d

Med inverse document frequency som

$$\mathrm{idf}(t, D) = \log \frac{|D| + 1}{|\{d \in D : t \in d\}|}$$

och Term Frequency

$$\mathrm{tf}(t, d) = |t \in d|$$

där $|D|$ är antal textdokument i samlingen och $|\{d \in D : t \in d\}|$ är antal textdokument i samlingen där termen t förekommer

Om vi som exempel räknar ut tf-idf för två textdokument: ”Idag var det varmt” och ”Igår var det kallt”. I de två textdokumenten har vi följande termer: ”Idag”, ”Igår”, ”var”, ”det”, ”varmt” och ”kallt”. Termerna ”var” och ”det” finns i båda textdokumenten vilket ger dem en idf-faktor på

$$\mathrm{idf}(t, D) = \log \frac{2 + 1}{2} = 0.4$$

I fallet ovan används den naturliga logaritmen men det spelar ingen roll vilken bas man använder för logaritmen då det endast kommer ändra värdet på alla resultat med en konstant faktor. Idf-faktorn för de resterande termerna blir

$$\mathrm{idf}(t, D) = \log \frac{2 + 1}{1} = 1$$

Eftersom alla termer bara förekommer en gång i varje textdokument så blir deras tf-idf värden för alla termer 0.4 eller 1. Därmed enligt tf-idf är termerna ”Idag” och ”varmt” viktigast för det första dokumentet medan ”Igår” och ”kallt” är viktigast för det andra. I vanliga fall har man självfallet mycket fler och längre textdokument men resultatet för exemplet hade nog inte ändrats jättemycket då termerna ”var” och ”det” är väldigt vanliga i svensk text, och de resterande fyra termerna antagligen förekommer lika ofta.

1.5.2 Vector space model

Vector space model är ett sätt representera textdokument (kan användas för att representera andra saker också) som vektorer, där man använder tf-idf för själva värdet i vektorn medan termen är dimensionen [7]. Man använder vector space model för att räkna ut hur lika två textdokument är. Värdet på likheten mellan två vektorer i vector space model räknas ut genom cosinus-vinkeln mellan de två vektorerna på följande sätt

$$\mathrm{sim}(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|}$$

Som räkneexempel tar vi dokumenten från sektion 1.5.1 och räknar ut vinkeln för dem. De termerna som har samma dimension är ”var” och ”det” vilket ger täljaren till ”sim”-funktionen $(0.4, 0.4) \cdot (0.4, 0.4) = 0.32$.

Nämnaren blir $\sqrt{2 \cdot 1^2 + 2 \cdot 0.4^2}\sqrt{2 \cdot 1^2 + 2 \cdot 0.4^2} = 2.16$ vilket ger ett värde $\frac{0.32}{2.32} = 0.14$ alltså har vektorerna 82 grader mellan varandra. Värdet blir lågt (eller vinkeln stor) eftersom de två viktigaste orden i de båda dokumentet är olika.

1.5.3 Longest common subsequence

Longest common subsequence (LCS) är ett problem där man försöker hitta den längsta sekvensen som överensstämmer mellan två mängder. Att lösa detta problem kan vara intressant för att mäta hur lika två strängar är genom att räkna ut längden på strängen man får utav LCS, och sen ge poäng efter längden på LCS-strängen dividerat med längden för de båda strängarna. Som exempel har vi två par av strängar

Universal Serial Bus
USB

ttttteeeee
dettatest

För det första paret blir LCS ”USB”, där första strängen är 17 bokstäver lång (mellanrum exkluderat) och den andra strängen är på tre vilket ger ett likhetsvärde på $2\frac{3}{17+3} = 0.30$. Det andra parets LCS är ”te” där den första strängen är 10 bokstäver lång och den andra strängen är på 9 bokstäver vilket resulterar i ett likhetsvärde $2\frac{2}{10+9} = 0.21$

1.5.4 F_1-poäng

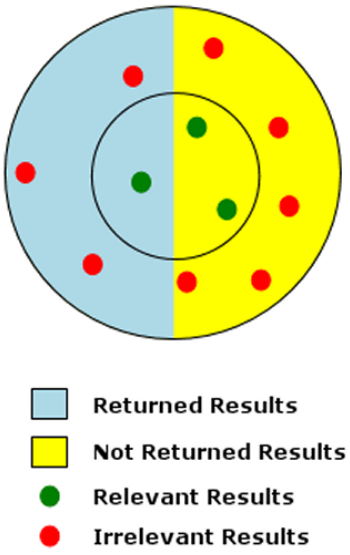
F_1-poäng är ett sätt att mäta hur väl ett test fungerar och används ofta för att mäta hur bra en lösning presterar inom fält som sökmotorer [5]. F_1-poäng är definierat som

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Precision är hur stor andel av de valda dokumenten som är korrekta och recall är hur stor andel av de korrekta dokumenten som är valda. Både precision och recall har en värdemängd inom $[0 \ldots 1]$. I Figur 1.1 ser man elva dokument varav de fyra till höger är valda och de gröna dokumenten är korrekta varav de röda inte är korrekta. I exemplet är det ett korrekt valt dokument av fyra vilket ger en precision på $\frac{1}{4}$. Man ser också att utav de korrekta dokumenten är ett valt vilket ger en recall på $\frac{1}{3}$.

Från exemplet i Figur 1.1 får man en F_1-poäng på

$$F_1 = 2 \cdot \frac{\frac{1}{4} \cdot \frac{1}{3}}{\frac{1}{4} + \frac{1}{3}} = 0.286$$



Figur 1.1: Precision & recall exempel

1.5.5 Korsvalidering

Korsvalidering är en teknik att mäta hur bra en lösning fungerar emot okänd eller ny data [3]. I korsvalidering delar man upp datan i en tränings- och testdel. Man optimerar först parametrarna i lösningen efter träningsdatan för att sedan köra med de optimerade parametrarna emot testdelen. Resultatet man får från testdelen säger hur bra lösningen fungerar.

Det finns en rad olika varianter på korsvalidering men den man är intresserad av i detta examensarbete är K-faldig. K:et står för hur många delar man delar upp datan. Om man har K-delar så blir träningsdatan $K-1$ av delarna och testdatan blir en, med denna uppdelning kan man köra lösning K gånger med unik testdata.

Kapitel 2

Implementering

För att lösa problemet undersöktes två olika metoder: tf-idf som är en beprövad teknik för att matcha textdokument och en kolumnbaserad lösning som i stor del är tf-idf som tar hänsyn till kolumnstrukturen i en produktdatabas. Den kolumnbaserade lösningen (och i viss mån den vanliga tf-idf) utvecklades för att få så bra resultat som möjligt på den testdatan som fanns tillgänglig.

2.1 Testdata

Som testdata till hjälp av utveckling och evaluering av examensarbetet användes produkter från två olika nätbutiker till att bygga två relationsdatabaser. Nätbutikerna som valdes användes för att de båda hade ett stort utbud av produkter och att de visade EAN-koden för varje produkt, vilket hjälpte till att göra matchningar mellan produktdatabaserna då det är säkert att två produkter är identiska om de har samma EAN-kod.

Två produkter från olika affärer med olika EAN-koder kan dock fortfarande vara identiska. Att två produkter skulle vara identiska om de hade samma EAN-kod visade sig inte vara helt sant då en av de två nätbutikerna hade flera produkter med samma EAN-kod.

Produkterna från nätbutikerna extraherades genom att spindla (låta ett datorpogram gå igenom alla produkter på deras hemsida) deras hemsidor och i mån av tid extraherades endast produkter inom kategorierna: TV, Spel, Surfplattor, Bärbara datorer, Videokameror, Minneskort och Kompaktkameror. För att hitta vilka produkter som var samma över de två databaserna matchades först de produkter som hade samma EAN-koder och därefter kontrollerades resterande produkter manuellt.

Produktkategori	Nätbutik A	Nätbutik B	Matchningar
TV	161 st.	48 st.	5 st.
Spel	597 st.	3499 st.	409 st.
Kompaktkamera	59 st.	67 st.	9 st.
Laptop	74 st.	23 st.	16 st.
Surfplatta	60 st.	41 st.	35 st.
Videokamera	15 st.	11 st.	1 st.
Minneskort	54 st.	27 st.	0 st.
Totalt	1020 st.	3716 st.	475 st.

Tabell 2.1: Antal produkter hämtade och antal korrekta matchningar mellan databaserna

2.2 Parsning

När rådata ska läsas in till Tf-Idf eller den kolumnbaserade matchningen räknas en ”term” som antingen en sträng – eller ett nummer eller datum. I Listing 2.1 defineras hur rådata läses in. ”StringTerms” har lägre prioritet än de de andra fyra typerna. Efter parsningen kommer som exempel två ”Integer” och ”Decimal” räknas som lika fastän om de var olika i rådatan t.ex. ”15” eller ”15.0”. Om man istället hade använt alla specialtecken som termbrytning hade ”15.0” parsats som {”15”,”0”} vilket hade hade gjort ”0”-termen omatchad. En annan anledning är för att falska matchningar såsom ”2012-02-01” i tf-idf hade gett full poäng emot ”2012-01-02”. Produkterna i Listing 2.1 fungerar bara för länder som har samma nummer- och datumstandard som i Sverige.

```
Document → Term (<Seperator> Term)*
Term → Integer | Decimal | Date | StringTerms
Integer → <Integer>
Decimal → <Decimal>
Date → <Date>
StringTerms → <String> (<FinalSeperator> <String>)*
```

Listing 2.1: CFG för parsningen

```
<String> = [a–zA–Z0–9]*
<Integer> = [0–9. ]*
<Decimal> = [0–9. ]+,[0–9]*
<Date> = [0–9]{4}(.|/| |–)[0–9]{2}(.|/| |–)[0–9]{2}
<Seperator> = ^[a–zA–Z0–9,/–.]
<FinalSeperator> = ,/–.
```

Listing 2.2: Tokens för CFG till svenska sidor

Man hade kunnat utveckla parsningen så att den normaliserar enheter vilket innebär att ”100 cm” hade räknas som att vara ekvivalent med ”1 m” vilket inte

hade matchats med parsningen ovan. Men med den testdatan som användes var det väldigt få kolumnvärden som beskrevs med annorlunda enheter, och när det väl gjordes stod ibland enheten i kolumnnamn och i andra fall kolumnvärdet vilket gör det lite svårare att implementera.

2.3 Metoder

I denna sektion förklaras hur de olika metoderna som har testas under examensarbetet är implementerade. Hur bra metoderna presterar och varför de är implementerade som de är diskuteras längre fram. Varje metod utvärderas efter tvåfaldig korsvalidering på tre olika dataset: spel, eleketronik och spel samt elektronik (detta datasetet kallas ”Allt”). Anledningen att bara köra tvåfaldig korsvalidering beror delvis på att körtiden för metoderna var väldigt hög, dvs, det hade tagit för lång tid att testa med tiofaldig. Den främsta anledningen är dock att idf-värdena blir felaktiga ifall de byggs upp med för lite data, vilket leder till konstiga resultat (testdelen för Eleketronik skulle bestå av ungefär 10 produkter för en tiofaldig korsvalidering).

Träningsdatan optimerar över tröskelvärdet k efter så hög F_1-poäng som möjligt. Testdatan körs sedan med tröskelvärdet k från träningsdatan och det är hur bra F_1-poäng träningsdatan får som man är intresserad av. I nedanstående sektioner beskrivs hur varje metod får fram poängen e för varje par mellan de två produktdatabaserna. Alla produkter mellan de två produktdatabaserna jämförs med varandra, vilket ger en komplexitet på $\mathrm{O}(nm)$ där n är antalet produkter i den första produktdatabasen och m antalet i den andra.

I produktdatabaserna $a_1(i, j)$ och $a_2(i, j)$ är varje värde ett element en vektor av termer och värden. Så om $a_1(0, 0)$ bestod av strängen ”Bildskärm 30 x 30 x 5” hade det efter parsningen blivit en vektor med dimensionerna: ”Bildskärm”, ”30”, ”x” och ”5” med värdena $(1, 2, 2, 1)$. Varje unik term i de båda produktdatabaserna blir en egen dimension för vektorerna.

2.3.1 Slumpmatchning

Första metoden som testades var att matcha produkter mellan databaserna helt slumpmässigt. För varje par $P = \{(i_1, i_2, e)\}$ sätts värdet e till ett slumpmässigt värde mellan $[0 \ldots 1]$ genom att använda ”Mersenne twister” som är en pseudorandom algoritm och beskrivs i mer detalj i [6].

2.3.2 Tf-idf

Denna metod försöker matcha produkter genom att använda sig av ren tf-idf som förklaras i sektion 1.5.1. Först definieras en samling vektorer från de båda produktdatabaserna $\mathbf{tfidf}_{1,i}$ från nätbutik A och $\mathbf{tfidf}_{2,i}$ från nätbutik B.

$$\mathbf{d_{1,i}} = \sum_j a_1(i,j) \text{ och } D_1 = \{\mathbf{d_{1,0}} \dots \mathbf{d_{1,n}}\}$$

$$\mathbf{tfidf}_{1,i} = \mathrm{tf}(t, \mathbf{d_{1,i}}) \times \mathrm{idf}(t, D_1)$$

$$\mathbf{d_{2,i}} = \sum_j a_2(i,j) \text{ och } D_2 = \{\mathbf{d_{2,0}} \dots \mathbf{d_{2,n}}\}$$

$$\mathbf{tfidf}_{2,i} = \mathrm{tf}(t, \mathbf{d_{1,2}}) \times \mathrm{idf}(t, D_2)$$

där $\mathrm{idf}(t, D)$ och $\mathrm{tf}(t, d)$ är definierade som

$$\mathrm{idf}(t, D) = \log \frac{|D| + 1}{|\{d \in D : t \in d\}|}$$

$$\mathrm{tf}(t, d) = |t \in d|$$

$\mathrm{idf}(t, D)$ returnerar antalet gånger en term förekommer i ett visst antal produkter och $\mathrm{idf}(t, D)$ ger ett värde på hur vanlig termen är för alla produkter. $\mathrm{idf}(t, D)$ adderar 1 till $|D|$ för att undvika att undvika division med noll i senare steg.

Poängen mellan två produkters vektorer räknas ut genom vector space model som beskrivs i sektion 1.5.2

$$\mathrm{sim}(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|}$$

Poängen e för varje par i_1, i_2 är $e = \mathrm{sim}(\mathbf{tfidf}_{1,i_1}, \mathbf{tfidf}_{2,i_2})$

2.3.3 Tf-idf med bijektiv matchning

Istället för att matcha alla möjliga kombinationer av par så matchas bara par som har sitt högsta värde på e emot varandra. På detta sätt kommer varje produkt bara vara matchade med en produkt i den andra produktdatabasen.

Alltså så tas det endast med par som uppfyller kravet

$$\{(i_1, i_2, e) : e = \max_{k=0\dots n}(\mathrm{sim}(\mathbf{tfidf}_{1,k}, \mathbf{tfidf}_{2,i_2})) \text{ och } e = \max_{k=0\dots n}(\mathrm{sim}(\mathbf{tfidf}_{1,i_1}, \mathbf{tfidf}_{2,k}))\}$$

Utöver detta krav fungerar lösningen ekvivalent med lösningen i föregående sektion. Man kan även matcha par som inte är strikt bijektiva om alla produkter i den andra databasen som matchar bättre redan är matchade emot en produkt i sin egen databas, detta kallar vi ”semi-bijektiv”.

2.3.4 Booleansk tf-idf med bijektiv matchning

Denna metod använder sig av bijektiv matchning som beskrevs i föregående sektion samt att $\mathrm{tf}(t, d)$ returnerar värdet 1 om produkten har en eller fler av termen t och annars 0.

$$\mathrm{tf}(t, d) = \begin{cases} 1 & \text{if } |t \in d| > 0 \\ 0 & \text{if } |t \in d| = 0 \end{cases}$$

2.3.5 Kolumnbaserad tf-idf

I de föregående metoderna tog man inte hänsyn till att varje kolumnvärde har en kolumn och kolumnnamn. Här försöker man ta hänsyn till kolumnstrukturen så att man endast matchar termer från liknade kolumner mellan de två produktdatabaserna. Utöver att bara matcha termer från liknade kolumner fungerar denna metod likadant som den i sektion 2.3.4

Nedan definieras först vektorer genom tf-idf för varje kolumn i produktdatabaserna.

$$\mathbf{c_{1,j}} = \sum_i a_1(i, j) \text{ och } C_1 = \{\mathbf{c_{1,0}} \dots \mathbf{c_{1,n}}\}$$

$$\mathbf{column}_{1,i} = \mathrm{tf}(t, \mathbf{c_{1,i}}) \times \mathrm{idf}(t, C_1)$$

och

$$\mathbf{c_{2,j}} = \sum_i a_2(i, j) \text{ och } C_2 = \{\mathbf{c_{2,0}} \dots \mathbf{c_{2,n}}\}$$

$$\mathbf{column}_{2,i} = \mathrm{tf}(t, \mathbf{c_{1,2}}) \times \mathrm{idf}(t, C_2)$$

Där poängen e_k för ett kolumnpar (j_1, j_2) är $e_k = \mathrm{sim}(\mathbf{column}_{1,j_1}, \mathbf{column}_{2,j_2}) \cdot L$ där L är ett värde $[1 \dots 2]$ baserat på hur lika strängarna i kolumnnamnen för kolumnparen är.

Om båda kolumnnamnen är helt ekvivalenta blir $L = 2$. Om ett av kolumnnamnen är en substräng av det andra kolumnnamnet och båda kolumnnamnen är längre än tre bokstäver blir $L = 1.75$. Om inget av de två föregående fallen stämmer används ”longest common subsequence” som beskrivs i sektion 1.5.3. Longest common subsequence ger som högst $L = 1.5$ (om de båda strängarna är helt ekvivalenta) och som minst $L = 1.0$ (om strängarna inte har en enda gemensam bokstav)

Vi definerar en samling kolumnpar som $K = \{(j_1, j_2, e) : e_k > 0.5\}$. Alltså alla kolumnpar som har en poäng högre än 0.5 kommer ha sina termer med i poängsummeringen.

och d-variablarna blir istället

$$\mathbf{d_{1,i}} = \sum_{(j_1,j_2,e)\in K} a_1(i,j_1)$$

$$\mathbf{tfidf}_{1,i} = \mathrm{tf}(t, \mathbf{d_{1,i}}) \times \mathrm{idf}(t, \mathbf{c_{1,i}})$$

och

$$\mathbf{d_{2,i}} = \sum_{(j_1,j_2,e)\in K} a_2(i,j_2)$$

$$\mathbf{tfidf}_{2,i} = \mathrm{tf}(t, \mathbf{d_{1,2}}) \times \mathrm{idf}(t, \mathbf{c_{2,i}})$$

2.3.6 Bijektiv kolumnbaserad

Denna metod kör bijektiv matchning även på kolumnmatchningar på samma sätt som det görs för produkterna och fungerar i stort sett annars ekvivalent med det som är beskrivet i sektion 2.3.5.

$$P = \{(i_1, i_2, e) : e = \max_{k=0\ldots n}(\mathrm{sim}(\mathbf{tfidf}_{1,k}, \mathbf{tfidf}_{2,i_2})) \text{ och } e = \max_{k=0\ldots n}(\mathrm{sim}(\mathbf{tfidf}_{1,i_1}, \mathbf{tfidf}_{2,k}))\}$$

Bijektiv matchning över kolumnerna fungerar inte helt felfritt då till skillnad från produkter kan en kolumn matcha flera kolumner samtidigt som t.ex. ”Hårddisk lagringskapacitet” i en produktdatabas kan korrespondera emot både, ”Hårddisk (GB)” och ”Hårddisk (TB)”, i den andra produktdatabasen. För att lösa detta kontrolleras det vilka kolumner som förekom tillsammans i produktdatabaserna och om de två kolumnerna ”Hårddisk (GB)” och ”Hårddisk (TB)” inte förekom samtidigt i någon produkt så kunde de fortfarande matchas till samma kolumn även om det inte längre är strikt bijektivt. Tröskelvärdet e_k sänks även från 0.5 till 0.2 då man inte längre behöver vara lika orolig över att kolumner matchas fel när de bara matchas en gång.

2.3.7 Strikt kolumnbaserad

Den strikt kolumnbaserade lösningen fungerar som i sektion 2.3.6 förutom att man nu endast matchar kolumnvärdet för en produkt emot dess korresponderande kolumnvärde i den andra produkten. Man summerar alltså inte ihop termerna över kolumnvärdena. Poängen räknas ut genom

$$\mathbf{s_{i_1,i_2}} = \sum_{(j_1,j_2,e)\in K} [\mathrm{tf}(t, a_1(i_1,j_1)) \times \mathrm{idf}(t, \mathbf{c_{1,j_1}})] \cdot [\mathrm{tf}(t, a_2(i_1,j_2)) \times \mathrm{idf}(t, \mathbf{c_{2,j_2}})]$$

$$\mathbf{m_{i_1,i_2}} = \sum_{(j_1,j_2,e)\in K} \|\mathrm{tf}(t, a_1(i_1, j_1)) \times \mathrm{idf}(t, \mathbf{c_{1,j_1}})\| \cdot \|\mathrm{tf}(t, a_2(i_2, j_2)) \times \mathrm{idf}(t, \mathbf{c_{2,j_2}})\|$$

där e för ett par i_1, i_2 blir

$$e = \frac{\mathbf{s_{i_1,i_2}}}{\mathbf{m_{i_1,i_2}}}$$

2.3.8 Normaliserad strikt kolumnbaserad

Här gör man så att varje kolumnvärde ger lika mycket poäng oavsett hur hög idf-poäng termerna har

$$\mathbf{s_{i_1,i_2}} = \sum_{(j_1,j_2,e)\in K} \mathrm{sim}[\mathrm{tf}(t, a_1(i_1, j_1)) \times \mathrm{idf}(t, \mathbf{c_{1,j_1}}), \mathrm{tf}(t, a_2(i_2, j_2)) \times \mathrm{idf}(t, \mathbf{c_{2,j_2}})]$$

där e blir summan av alla element i $\mathbf{s_{i_1,i_2}}$ dividerat med antal dimensioner i $\mathbf{s_{i_1,i_2}}$

Strikt kolumnbaserad med tf-idf vikter

$$\mathbf{s_{i_1,i_2}} = \sum_{(j_1,j_2,e)\in K} \mathrm{sim}[\mathrm{tf}(t, a_1(i, j_1))\times\mathrm{idf}(t, \mathbf{c_{1,j_1}}), \mathrm{tf}(t, a_2(i, j_2))\times\mathrm{idf}(t, \mathbf{c_{2,j_2}})]\cdot(w_{1,j_1}+w_{2,j_2})$$

där e blir summan av alla element i $\mathbf{s_{i_1,i_2}}$ dividerat med summan av $(w_{1,i}+w_{2,i})$ där w bestäms efter medelvärdet av tf-idf på kolumntabellen K. Så $w_{1,i}$ är medelvärdet av elementen i vektorn $\mathbf{column}_{1,i}$

2.3.9 Metoder utvärderade på alla strikt kolumnbaserade lösningar

- Semi-bijektiv. som beskrivs i slutet av sektion 2.3.3 (utvärderas även på icke kolumnbaserade lösningar)
- Normalisera poäng. Avrunda poängen för varje kolumnmatchning till ett av de fem talen: 0, 0.25, 0.5, 0.75, 1
- Uteblivna termer. Ge poäng för termer som inte förekommer alls i den andra produktens kolumn (Åldersgräns i en produktdatabas skriver ålder som ”18 år” medan den andra bara skriver ”18” så termen ”år” kommer aldrig matchas). Poängen som ges är tf-idf värdet upphöjt till två gånger 0.5
- Fullpoängsbelöning. Ge bonuspoäng om två kolumnvärden matchar till 100% (så t.ex. kolumner som bara har JA/NEJ-värden får högre poäng). Poängen adderar 1 till både nämnaren och täljaren för varje fullpoängskolumn när man räknar ut poängen i ”sim”-funktionen

- Icke booleansk. Använda sig av en linjär tf-funktion istället för den booleanska som används för alla strikt kolumnbaserade
- Matcha över kolumner. Om termen finns i en annan kolumn tas termen med som partiell matchning där poängen är tf-idf värdet upphöjt till två gånger 0.5

Kapitel 3

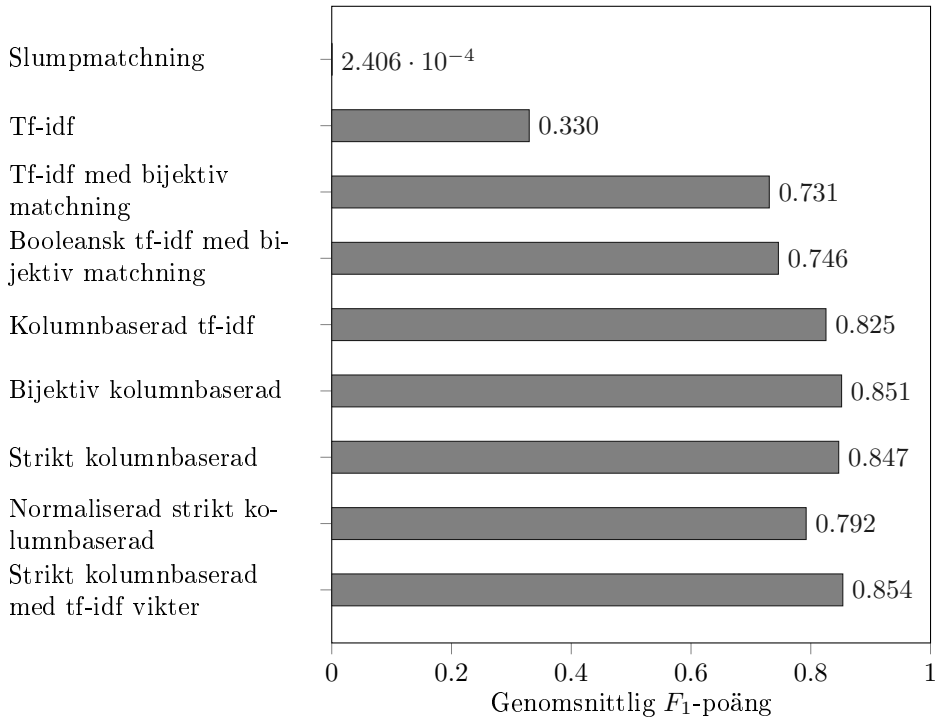
Resultat

För att mäta hur en lösning presterar tar man medelvärdet av F_1-poängen av en tvåfaldig korsvalidering. Varje lösning kommer mätas över tre olika dataset: Elektronik, Spel och båda två (som kallas ”Allt”). Elektronik räknas som TV, Kompaktkamera, Laptop, Surfplatta, Videokamera och Minneskort från Tabell 2.1 i sektion 2.1. F1-poäng förklaras i sektion 1.5.4 och korsvalidering i sektion 1.5.5. Flera av de olika metoderna som är listade i föregående sektion kan kombineras med varandra vilket leder till ett väldigt stort antal olika lösningar. Eftersom det blir ett väldigt stort antal olika lösningar presenteras endast de som är mest intressanta ur ut ett diskussions-perspektiv.

Tabellerna i ”Appendix resultat” listas resultat från de båda körningarna (tränings- och testkörningen) och deras medelvärde där F_1 är F_1-poängen för testdatan, F_1^{T} är den optimerade F_1-poängen för träningsdatan och k är det optimala tröskelvärdet från träningsdatan. I graferna visas endast den genomsnittliga F_1-poängen för testdatan.

Värdena som blev ”NaN” (Not a Number) beror på att lösningen inte hittade någon matchning vilket gav en recall på noll som leder till division med noll när man ska räkna ut F_1-poängen. För att räkna ut medelvärdet för de dataseten med ”NaN” ersattes det som noll istället.

I tabellen nedan visas medelvärdet för F_1-poängen för datasetet ”Allt” för alla de olika lösningarna. För de strikt kolumnbaserade lösningarna visas resultatet för den kombination av metoder från sektion 2.3.9 ”Metoder utvärderade på alla strikt kolumnbaserade lösningar” som gav högst medelvärde för F_1-poängen.


Slumpmatchning
Tf-idf
Tf-idf med bijektiv matchning
Booleansk tf-idf med bijektiv matchning
Kolumnbaserad tf-idf
Bijektiv kolumnbaserad
Strikt kolumnbaserad
Normaliserad strikt kolumnbaserad
Strikt kolumnbaserad med tf-idf vikter
$2.406 \cdot 10^{-4}$
0.330
0.731
0.746
0.825
0.851
0.847
0.792
0.854
0
0.2
0.4
0.6
0.8
1
Genomsnittlig F_1-poäng

Kapitel 4

Diskussion

I denna sektion diskuteras varför lösningarna är implementerade som de är och resultaten för varje lösning. Som det togs upp i början av sektion 3 så kommer inte alla möjliga kombinationer av lösningar diskuteras utom bara de som gav bra eller intressanta resultat

Överlag fick de kolumnbaserade lösningarna en F_1-poäng på ungefär 0.85 för alla dataseten. Det är svårt att konstatera styrkor och svagheter för de olika kolumnbaserade lösningarna då de alla fick ungefär samma poäng.

En del av produkterna mellan de två produktdatabaserna är väldigt svåra att matcha som t.ex. spel som har sina titlar på svenska i den ena produktdatabasen och på engelska i den andra. Under den manuella matchningen var det en del produkter jag själv var osäker på om de verkligen var detsamma, om man skulle räkna ett spel som det stod var "Limited Edition" so m samma som ett annat ekvivalent spel i den andra produktdatabasen men som inte var "Limited Edition". Det var också ett par stycken datorer som jag inte kunde avgöra om de var samma produkt eller inte. Jag skulle gissa på att ungefär 5 till 10% av antalet korrekt matchande produkter mellan de två produktdatabaserna inte går att förvänta sig vara möjliga att matchas av algoritmer som diskuteras i detta examensarbete.

Resultat-tabellerna för varje lösning går att hitta i kapitel 6 "Appendix resultat"

4.1 Slumpmatchning

Denna metod testades då det är intressant att se hur bra de andra lösningarna presterar jämfört med en lösning som väljer helt på måfå. Rent teoretiskt kan man se att det antagligen inte kommer gå vidare bra för denna lösning då utav de $1020 \cdot 3716 = 3790320$ möjliga paren i datasetet "Allt" är endast korrekta 475 matchningar (1020 produkter från nätbutik A och 3716 från nätbutik B). Om man väljer allt (tröskelvärdet till 0, $k = 0$) för datasetet "Allt" får man en recall på 1 och precision till $\frac{475}{3790320} = 0.0001253$, vilket ger en F_1-poäng på ungefär 0.00025.

Den minsta F_1^{T}-poängen slumpmatchingen kan ge för datasetet ”Allt” är alltså 0.00025.

Vi ser att tröskelvärdet för alla körningar och dataseten (utom Elektronik) är höga. I de fallen där tröskelvärdet är högt beror på att de bara tog ett fåtal par för att få en hög precision, och i många fall lämnar det testkörningen utan någon matchning alls. I andra fall gav det testdatan någon enstaka korrekt matchning men i överlag handlar det om att testdatan får en poäng på en faktor fyra mindre (går inte att se då det endast visas 3 decimaler i resultat-tabellen).

Till skillnad från de andra lösningarna hjälper inte tröskelvärdet man får från träningsdatan att förbättra resultatet till testdatan. Detta då slumpmatchning av ren definition ger oberoende resultat vilket gör det lite märkligt att testa korsvalidering på just denna lösning. Som vi såg fick träningsdatan aldrig poäng över 0.00025 så för slumpmatchning hade det antagligen varit bättre att sätta tröskelvärdet för testdatan till 0 och skippa testfasen helt.

4.2 Tf-idf

Tf-idf med Vector space model verkar rätt lovande då det använts mycket till att mäta hur lika textdokument är. Man kan se en produkt som ett textdokument om man bara adderar ihop alla termerna i alla kolumnvärden. Unika kolumnvärden för en produkt som t.ex. processorhastigheten för en dator eller surfplatta kommer få hög poäng av tf-idf, och om en produkt i den andra produktdatabasen har samma processorhastighet blir likhetsvärdet mellan de två mycket högre. När den manuella matchningen mellan produkter gjordes matchades produkter på liknade sätt. I den manuella matchningen försökte man hitta unika kolumnvärden för varje produkt och om någon annan produkt i den andra produktdatabasen hade mycket liknade unika kolumnvärden så sattes de som en matchning.

Som vi ser i tabellen blir tröskelvärdet för varje dataset och körning ungefär lika (som lägst 0.438 och som högst 0.574) vilket förklarar varför F_1-poängen är nästan lika hög som F_1^{T} (t.o.m. högre för datasetet ”Allt” i körning 1). Tittar man på tröskelvärdet i körning 1 för t.ex. ”Elektronik” så är tröskelvärdet 0.466 jämfört med det optimala tröskelvärdet man ser i körning två: 0.438, att F_1-poängen för träningsdatan nästan är samma som för testdatan är inte konstigt då tröskelvärdet skiljer sig endast med 0.028 jämfört med det optimala tröskelvärdet. Att tröskelvärdet även är rätt nära för de två rätt skilda kategorierna ”Elektronik” och ”Spel” kan tyda på att lösningen skulle prestera likvärdigt för ännu ett dataset om man skippar träningsfasen helt och bara sätter tröskelvärdet till 0.5.

Tf-idf presterade väldigt mycket bättre än slumpmatchning men det är fortfarande rätt låga resultat. För detta examensarbetet valdes tf-idf som grund och lösningar byggdes vidare baserade på tf-idf. Tf-idf valdes som grund då det är en välbeprövad teknik som använts i många liknade fall [8]. Varje lösning som testades skapades efter att åtgärda något eller några av problemen som var identifierade med tf-idf. I sektionerna nedan beskriver en del av problemen med ”tf-idf”-lösningen.

4.2.1 Sammanfogning

Som man ser i resultatet presterar tf-idf inte särskilt bra när man slog ihop (eller sammanfogade) elektronik och spel (datasetet ”Allt”). Om lösningen hade kunnat hantera sammanslagningar perfekt hade resultatet åtminstone blivit högre än 0.507 då spel har denna poäng och elektronik har högre. Så varför presterar lösningen sämre när man matchar över två sammanfogade dataset? En av anledningarna är att idf-värdet för varje term blir annorlunda än när dataseten var skilda. Om vi t.ex. tittar på termen ”NEJ” som i princip förekommer för varje elektronikprodukt och aldrig förekommer för spelprodukter. När elektronik var ett eget dataset var idf-värdet för ”NEJ” noll, men i det sammanfogade datasetet blir idf-värdet

$$\log \frac{3716}{217} = 2.84$$

då det är 3716 produkter varav 217 är elektronik i datasetet spel enligt Tabell 1. Om man jämför det relativa värdet av termen ”NEJ” emot en unik term (idf-värde på $\log \frac{3716}{1} = 8.22$) har nu ”NEJ”-termerna värdet av en unik term på en faktor 0.35. Inom TV-kategorin där varje produkt ett tjugotal JA/NEJ-kolumner blir matchningar med andra TV-apparater väldigt hög även om de är helt skilda modeller. Problemet är att en term som förekommer väldigt ofta för alla elektronikprodukter nu väger mycket och höjer poängen för alla elektronikprodukter avsevärt. De högst värderade paren mellan de två produktdataseten är nu endast elektronikprodukter, nästan varenda möjligt elektronikpar kommer högre upp än spelparen.

4.2.2 Kritisk information ignoreras & matchningar över kolumner

En del kolumnvärden för en produkt kan vara helt avgörande för att se om två produkter är korrekt matchade eller inte. Som t.ex. är två tv-apparater antagligen inte samma om båda har kolumnen ”HD-TV” och det är ”NEJ” för den ena och ”JA” för den andra. I denna lösning tittar man inte alls på kolumnen utom mäter bara om de två produkterna har lika många ”JA” som ”NEJ”. Två produkter med två kolumner ”HD-TV” och ”Widescreen” får full poäng om den den första produkten har ”NEJ” på ”HD-TV” och ”JA” på ”Widescreen” och den andra produkten har tvärtom.

4.2.3 Matchar produkter flera gånger

Som det togs upp i Begränsningar (sektion 1.4) antar man att produktdatabaserna är välformaterade vilket innebär att en produkt inte finns flera gånger inom samma produktdatabas. Detta betyder att det högst ska finnas en korrekt par för varje produkt mellan två produktdatabaser. I denna lösning matchar man varje produkt emot alla produkter i den andra databasen, även om två felaktigt matchade produkter borde få lägre poäng kan två liknade produkter få högre poäng

än två korrekt matchade produkterna om de beskrivs på annorlunda sätt i de två produktdatabaserna (som t.ex. kan bero på det som beskrivs i sektion 4.2.3 eller sektion 4.2.1).

Problem med utebliven information

Ett problem med tf-idf är när produkter visade mycket mer information gentemot dess par i den andra databasen. Ett konkret exempel på detta är att en av databaserna visade rekommenderad datorprestanda för sina spel vilket bestod av ungefär hälften av alla kolumner för datorspel. Den andra databasen har inte alls med rekommenderad datorprestanda för sina datorspel vilket gör att datorspel från ena databasen matchades bäst emot konsolspel från den andra (då konsolspel inte har rekommenderad prestanda).

4.3 Tf-idf med bijektiv matchning

Att matcha bijektivt gjordes för att åtgärda att produkter matchas flera gånger som beskrivs i sektion 4.2.3. Båda produktdatabaserna har produkter som är väldigt lika som t.ex. två modeller av identitiska surfplattor där endast lagringsutrymmet skiljer sig åt, detta ger $2 \cdot 2 = 4$ matchningar med hög poäng varav endast hälften är korrekta. Det blir ännu värre med identiska spel som kommer ut till olika format då de har samma namn, releasedatum, genre, producent och distributör, där endast format skiljer sig. Ett spel som kommer till PC, Xbox 360, PS3 och Wii ger $4 \cdot 4 = 16$ matchningar med hög poäng varav endast fyra är korrekta.

Som det beskrivs i metoder kommer denna lösning max ha en matchning per produkt vilket leder till att resultatlistan P högst ha en samma längd som antalet produkter i den minsta databasen oberoende av tröskelvärdet k. Nackdelen med denna lösning är att det inte är säkert att alla korrekta matchningar är med, så om man sätter tröskelvärdet k till 0 är det inte säkert att man får en recall på 1, vilket man alltid får med den föregående lösningen.

Som vi ser för resultatet blir det en förbättring för alla dataset vilket var förväntat. Datasetet ”Allt” förbättras markant då att matcha bijektivt också indirekt löser en del av problemet med att elektronikprodukter får högre poäng än spelprodukter som beskrivs i sektion 4.2.3. Denna lösning fixar inte det bakomliggande problemet med sammanfogning utom endast reducerar hur mycket ”skada” det gör.

Man ser också att det optimala tröskelvärdet k för träningsdatan är mycket lägre än vad det var för den föregående lösningen. Skillnaden mellan det optimala tröskelvärdet k är fortfarande liten mellan de olika dataseten och körningar, vilket är positivt som det diskuteras om i föregående sektion 4.2.

Att köra semi-bijektivt höjer recall om man sätter tröskelvärdet k till 0. Det är fortfarande inte säkert att man får en recall på 1 med tröskelvärdet k till 0 då två felaktigt matchade produkter kan fortfarande matcha varandra bijektivt även

om det finns en korrekt matchning för vardera av produkterna. Resultatet för semi-bijektiv är något sämre för dataseten ”Allt” och ”Elektronik” men en liten förbättring för ”Spel”.

Att det blir bättre att matcha spel semi-bijektivt beror antagligen på att det löser lite av problemet som i sektion 4.2.3 då om man har ett spel som släpps både till dator och en konsol kommer konsolversionerna över de båda produktdatabaserna matcha varandra högst, och datorversionen kommer antagligen också matcha konsolversionen högst på grund av ”rekommenderad datorprestanda”-kolumnen men matcha datorversionen som näst högst.

Tröskelvärdena k blir lite högre för den semi-bijektiva jämfört med den bijektiva vilket antagligen beror på att den helt enkelt skapar en resultatlista med fler matchningar.

4.4 Boolesk tf-idf med bijektiv matchning

Att använda boolesk tf-funktion istället för linjär testades för att det skulle reducera skadan av exempelproblemet i sektion 4.2.1 om sammanfogning (JA/NEJ-termerna räknas bara en gång nu). På samma sätt borde lösningen också reducera skadan från problemet med matchningar över kolumner som beskrivs i sektion 4.2.2.

Som förväntat blir resultatet bättre för dataseten ”Allt” och ”Elektronik” men marginellt sämre för ”Spel”. Att det inte blir bättre för ”Spel” beror antagligen på att det datasetet inte lider av problemet som beskrivs i sektion 4.2.2 eller exempelproblemet från sektion 4.2.1 då produktbeskrivningen för spel inte har JA/NEJ-kolumner.

4.5 Kolumnbaserad tf-idf

Denna metod skapades främst för att åtgärda sammanfogningsproblemet och problemet med utebliven information som beskrivs i sektion 4.2.1 och sektion 4.2.3. Problemet med sammanfogning löses genom att låta varje kolumn ha sina egna idf-värden vilket gör att idf-värdena för varje term i en kolumn kommer vara detsamma även när man sammanfogar dataset (om dataseten inte har kolumner med samma namn). Problemet med utebliven information kommer också åtgärdas då om en produktdatabas har en kolumn som inte liknar någon kolumn i den andra produktdatabasen kommer denna kolumn att ignoreras.

Självfallet leder detta till potentiella nya problem såsom att information som egentligen borde tas med ignoreras eller att kolumn som faktiskt finns representerad i båda produktdatabaserna ignoreras för de råkar ha mycket skilda värden. Att man matchar kolumner på ett bra sätt är väldigt kritiskt för att denna lösning ska fungera bra.

Vi ser en tydlig förbättring för dataseten ”Allt” och ”Spel”. Att datasetet ”Allt” ligger såpass nära ”Spel” tyder på att sammanfogningsproblemet i princip är löst. ”Elektronik” har dock försämrats markant vilket beror på att kolumnmatchningen inte fungerar, elektronik är den kategori med flest kolumner där varje produkt i snitt har 20 kolumner medan spel ligger på runt 6-9 kolumner.

Tröskelvärdena k är mycket högre än för de två föregående lösningar vilket delvis beror på att man har löst problemet med utebliven information (man matchar bara matchande kolumner) och att idf-värdena nu är annorlunda då de tas direkt från sin egen kolumn. Tröskelvärdena är också väldigt nära varandra över de tre dataseten vilket verkar lovande för om man skulle vilja testa lösningen på data utan att träna upp tröskelvärdet först (bara sätta det till 0.5), det är dock ingen garanti att det kommer fungera lika bra för annan data.

4.6 Bijektiv kolumnbaserad

För att fixa problemen med kolumnmatchningen valdes det att även matcha kolumner bijektivt, detta var för att kunna sätta lägre värde för att räkna kolumner som matchade. Att lättare matcha kolumner borde lösa problemet för datasetet ”Elektronik”.

Som vi ser i resultatet blir det en markant förbättring för ”Elektronik” som fick t.om. högre poäng än den booleska tf-idf med bijektiv matchning. ”Spel” blir marginellt sämre och ”Allt” blir bättre. Att ”Allt” inte blir mycket bättre även fast ”Elektronik” förbättrades markant beror på att det är mycket färre elektronikprodukter än spel.

Tröskelvärdena k är ungefär detsamma som i den föregående metoden men värdet varierar mer mellan dataseten.

4.7 Strikt kolumnbaserad

Denna metod gjordes för att lösa problemen att kritisk information ignoreras & matchningar över kolumner som beskrivs i sektion 4.2.2. Metoden löser inte problemet med att kritisk information ignoreras av sig självt men med ”Fullpoängsbelöning” åtgärdas det problemet.

I resultaten ser man en direkt försämring för alla dataset utom spel (som blir marginellt bättre) jämfört med föregående lösning. elektronikprodukterna försämras markant vilket beror huvudsakligen på inkonsekventa kolumner i en av produktdatabaserna. I ena produktdatabasen har de flesta elektronikprodukter tre kolumner ”Höjd”, ”Bredd” och ”Djup” som specificerar dimensionerna för produkten. ”Höjd” och ”Djup” byter ofta mening för produkterna så att ”Djup” beskriver höjden istället medan ”Höjd” beskriver djupet. Dimensionskolumnerna väger rätt mycket då storleken på en elektronikprodukt är oftast rätt unika.

Tröskelvärdet k blir mycket lägre för elektronikprodukter medan marginellt så för spel. Att det blir mycket lägre för elektronikprodukter beror på de inkonsekventa kolumnerna. När man matchar strikt kommer alltid poängen man får bli samma eller lägre som poängen hade blivit för den vanliga kolumnbaserade då enda skillnaden mellan dem är att den strikta kanske inte matchar två termer om de råkar ligga i olika kolumner.

Om man kör med ”Icke boolesk”-matchning för strikt kolumnbaserad borde det inte göra resultatet sämre då saker som JA/NEJ-termer oftast bara förekommer en gång per kolumn. Som man ser i grafen ”Strikt kolumnbaserad med Icke boolesk” stämmer detta då resultatet är identiskt förutom att det blir marginellt sämre för elektronikprodukter.

För att åtgärda problemet med inkonsekventa kolumner testades det med att ge partiell matchning för termer som finns i någon annan kolumn, även om detta gör att det inte längre är strikt kolumnbaserat. Som vi ser för ”Strikt kolumnbaserad med Matcha över kolumner” blir resultatet och tröskelvärdena likvärdiga som för den bijektivt kolumnbaserade.

Testar även att matcha ge partiell matchning för uteblivna termer för att reducera problemet med utebliven information. För t.ex. spel i ena produktdatabasen listar ”Åldersgräns”-kolumnen åldersgränsen som exempel ”18 år”, ”15 år” eller ”barnspel” medan den andra produktdatabasen bara skriver ”18”, ”15” eller ”barnspel”. Genom att ge en partiell matchning om en term inte alls förekommer i den ena kolumnen reduceras problemet. Som man ser i ”Strikt kolumnbaserad med Matcha över kolumner & Uteblivna termer” blir resultatet marginellt sämre. Detta beror delvis på att produktdatabaserna är relativt små så att många termer blir unika för en kolumn när de egentligen inte kanske hade varit så för en större produktdatabas. En produkt som bara har unika termer i sin kolumner hade fått 0.5 som poäng emot alla andra produkter vilket såklart inte är rimligt.

För att lösa problemet med att kritisk information ignoreras som beskrivs i sektion 4.2.2 ges en minimumpoäng för en kolumn om termerna matchar perfekt. Detta ger simpla JA/NEJ-kolumner en faktiskt vikt vilket borde hjälpa elektronikprodukter. Man ser i ”Strikt kolumnbaserad med Matcha över kolumner & Fullpoängsbelöning” att resultatet faktiskt blev sämre för elektronikprodukter vilket kan ses som förvånade. Anledningen för detta verkar bero på att för detta dataset inte lönar sig att titta på JA/NEJ-kolumner överhuvudtaget och istället låta de mer specifika kolumnerna såsom processorhastighet för datorer eller dimensioner för en TV vara de kolumner som bestämmer hur lika två produkter är. JA/NEJ-kolumner är svårare att kolumnmatcha just på grund av att det finns så många av dem och att de är så lika vilket leder till att kolumnmatchningen är mer osäker än för andra kolumner.

4.8 Normaliserad strikt kolumnbaserad

Att göra så att alla kolumner gav ungefär lika mycket poäng gjordes för att åtgärda att kritisk information ignoreras i sektion 4.2.2. Till skillnad från den föregående sektion borde denna metod åtgärda problemet med att kritisk information ignoreras.

Som det diskuterades för ”Fullpoängsbelöning” i föregående sektion så verkar det inte vara så effektivt att faktiskt ta hänsyn till JA/NEJ-kolumner vilket denna metod är specialdesignad för. Som vi ser i resultat får denna lösning sämre resultat än den föregående för alla dataset. Skillnaden för tröskelvärdet mellan spel- och elektronikprodukter är mycket större än för de föregående lösningarna vilket inte verkar lovande för om man ska köra denna metod på data utan att träna den först. Denna metod känns inte helt klockren då om två matchade JA/NEJ-kolumner för två produkter råkar stämma överens väger det högre än låt säga en kolumn med unikt modellnamn hade matchat, vilket också är anledningen till att elektronikprodukter presterar så pass mycket sämre.

Som för den föregående lösningen presterar denna bättre för dataseten ”Allt” och ”Elektronik” om man matchar över kolumner. Skillnaden mellan tröskelvärdena är fortfarande höga och spelprodukterna fick exakt samma poäng som innan.

4.9 Strikt kolumnbaserad med tf-idf vikter

Denna metod är en utbyggnad på föregående metod men till skillnad från den föregående tar denna inte hänsyn till problemet med att kritisk information ignoreras. Denna metod ska lösa problemet med om två matchade JA/NEJ-kolumner för två produkter råkar stämma överens väger det högre än låt säga en kolumn med unikt modellnamn hade matchat. Att matcha på detta sätt kan tyckas vara identiskt med den strikt kolumnbaserade metoden men med denna metod ger varje kolumn lika mycket poäng oavsett vad det är för termer man matchar. Denna metod bör prestera lika bra som den strikt kolumnbaserade om man inte har en stor spridning i tf-idf värdena mellan elementen i kolumnerna då medelvärdet av tf-idf värdena är sannolikt nära ett specifikt värde från mängden (om spridningen är låg).

Man ser att denna metod presterar bättre än den föregående men marginellt sämre än den strikt kolumnbaserade och till skillnad från den föregående metoden är differensen mellan tröskelvärdena för de olika dataseten relativt låg. Som de andra strikta metoderna får denna metod högre poäng för de två dataseten ”Allt” och ”Elektronik” om man matchar över kolumner.

I några fall kan man tycka att två kolumnvärden ska ge lika mycket poäng som t.ex. ”Borderlands 2” emot ”Borderlands 2 PS3” och ”Borderlands 2 Wii” men om det finns färre kolumnvärden med ”Wii” än med ”PS3” får kolumnvärdet högre poäng emot ”Borderlands 2 PS3”. Det kan även självfallet vara önskvärt att kolumnvärdet ska matcha emot PS3 bättre då det finns fler PS3-spel än Wii-spel vilket leder till att det är högre sannolikhet att det faktiskt är till PS3. När man normaliserar

poängen för kolumnmatchningarna får man en marginellt sämre resultat och även mer spridning för tröskelvärdena mellan dataseten.

Kapitel 5

Slutsatser

I denna sektion diskuteras det kort om varje lösning, om det gick som förväntat och om potentiella vidareutvecklingar. Om man ska utveckla vidare på vad som är gjort under detta examensarbete borde man nog börja med att hitta fler och större produktdatabaser då många av lösningarna presterade jämlikt och med större eller fler dataset kan man med större säkerhet säga vilken lösning som är generellt bättre. De flesta av de sista lösningarna presterade ungefär lika bra och det hade varit intressant att se hur de presterar för större produktdatabaser. En intressant observation är att det verkar bäst att helt ignorera JA/NEJ-kolumner.

I exmanesarbetet har det tagits fram en rad lösningar som med bra noggrannhet löser problemet från problembeskrivningen

5.1 Slumpmatchning

Som förväntat presterade denna lösning extremt dåligt men visade samtidigt att antalet kombinationer av matchningar är på en annan nivå än för vanliga sökproblem. Detta säger också att det inte är speciellt sannolikt att en algoritm är bättre än någon annan baserat på ren tur. Har svårt att tänka mig att man skulle kunna att bygga vidare på denna lösning.

5.2 Tf-idf

Tf-idf var intressant att använda som en baseline då det ofta används till liknade problem. Att hitta just matchningar mellan två produktdatabaser hade självfallet några speciella egenskaper som standard tf-idf inte tar hänsyn till, vilket lede till ett någorlunda dåligt resultat för alla dataset utom Elektronik.

Under examensarbetet byggdes det vidare på tf-idf under författningen att produktdatabaserna var välformaterade. Om databaserna inte vore välformaterade

och det hade förekommit saker såsom stavfel hade nog de andra lösningarna blivit rätt annorlunda.

5.3 Tf-Idf med bijektiv matchning

Som förväntat blev resultatet avsevärt mycket bättre när man körde bijektiv matchning. Semi-bijektiv presterade också bra men inte riktigt i samma nivå som den bijektiva. En förbättring vore att göra en lite smartare semi-bijektiv vilket kan leda till att den presterar bättre än den rent bijektiva

5.4 Booleansk tf-idf med bijektiv matchning

Även här blev resultatet som förväntat och annars finns det inte så mycket att säga om just denna lösning.

5.5 Kolumnbaserad tf-idf

Kolumnbaserad tf-idf fungerade bra förutom just för datasetet ”Elektronik”. Att matcha kolumner blev inte helt lyckat för elektronikprodukterna och det finns mycket att potentiellt arbeta vidare på för att matcha kolumner bättre.

5.6 Bijektiv kolumnbaserad

Som förväntat blev resultatet mycket bättre för elektronikprodukterna men denna lösning tar inte hänsyn till att en kolumn kan matcha flera kolumner. Att den inte kunde matcha en kolumn emot flera var inget problem med dataseten som användes men det kan säkerligen vara ett problem för andra produktdatabaser. Denna metod var en av de bästa enligt testerna.

5.7 Strikt kolumnbaserad

Den strikt kolumnbaserade presterade något sämre än den bijektivt kolumnbaserade vilket var lite förvånade. Denna metod finns det mycket att utveckla vidare på som t.ex. förbättra ”Uteblivna termer” eller förbättra matchningen för kolumner.

5.8 Normaliserad strikt kolumnbaserad

Denna lösning presterade klart sämre än alla de andra kolumnbaserade lösningarna men visar att det kan vara intressant att matcha med vikter (i detta fall var vikterna 1 för alla kolumner) på kolumner.

5.9 Strikt kolumnbaserad med tf-idf vikter

Som utbyggnad på Normaliserad strikt kolumnbaserad presterar denna metod nästan lika bra som den bijektiva kolumnbaserade. Denna lösning kan vara intressant att bygga vidare på och eventuellt sätta minimumpoäng för varje kolumn om man har data där JA/NEJ-kolumner är mycket viktiga.

5.10 Rekommendationer

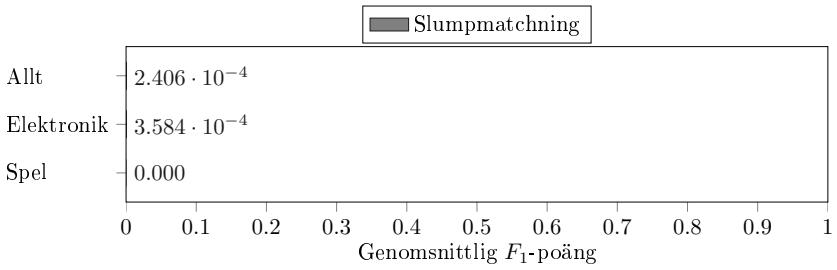
Som det togs upp i början av "Slutsatser" är det svårt att säga vilken av lösningarna som skulle prestera bäst på ny data då många utav dem presterade likvärdigt. Men utav de fyra lösningarna som fick över 0.8 i F_1-poäng skulle jag inte rekommendera "Kolumnbaserad tf-idf" då "Bijektiv kolumnbaserad" var en rakt förbättring som presterade bättre på alla dataset och borde således väljas före. Strikt kolumnbaserad och Strikt kolumnbaserad med tf-idf vikter må prestera bättre än Bijektiv kolumnbaserad om man har data där JA/NEJ-kolumner är väldigt avgörande.

Överlag är nog den Bijektivt kolumnbaserade den bästa lösningen för de flesta produktdatabaser, och även den lösning som är lättast att utveckla vidare på.

Kapitel 6

Appendix resultat

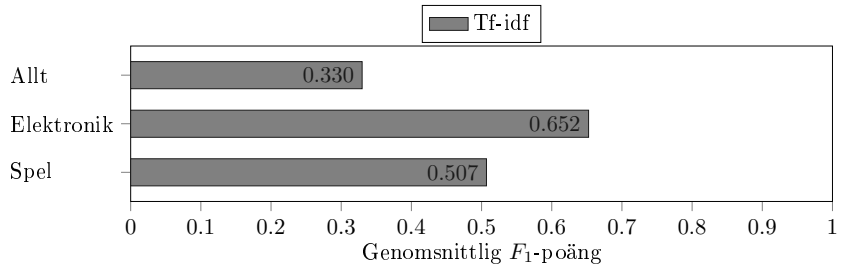
6.1 Slumpmatchning



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	NaN	0.008	1.000	0.000	0.002	0.999	0.000	0.005	0.994
Elektronik	NaN	0.007	0.988	0.001	0.004	0.758	0.000	0.006	0.873
Spel	NaN	0.003	0.999	0.000	0.001	0.988	0.000	0.002	0.993

Tabell 6.1: Värden för slumpmatchningen

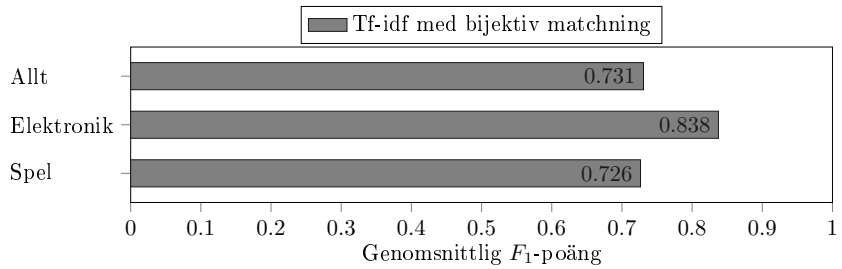
6.2 Tf-idf



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.361	0.338	0.531	0.299	0.373	0.574	0.330	0.355	0.552
Elektronik	0.638	0.680	0.466	0.667	0.704	0.438	0.652	0.692	0.452
Spel	0.503	0.531	0.466	0.511	0.521	0.511	0.507	0.526	0.487

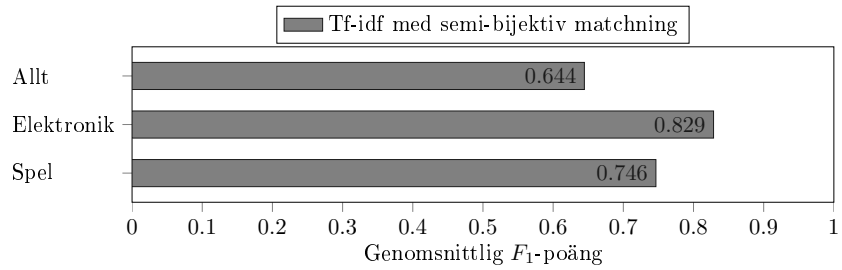
Tabell 6.2: Värden för tf-idf

6.3 Tf-idf med bijektiv matchning



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.740	0.729	0.291	0.722	0.748	0.334	0.731	0.739	0.312
Elektronik	0.836	0.839	0.346	0.839	0.862	0.321	0.838	0.850	0.333
Spel	0.73	0.729	0.198	0.723	0.740	0.245	0.726	0.734	0.221

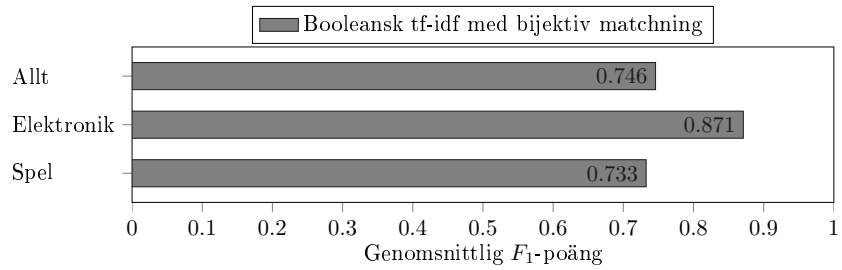
Tabell 6.3: Värden för strikt tf-idf med bijektiv matchning



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.653	0.651	0.439	0.644	0.684	0.334	0.644	0.668	0.387
Elektronik	0.857	0.839	0.346	0.8	0.857	0.356	0.829	0.848	0.351
Spel	0.762	0.731	0.345	0.731	0.765	0.344	0.746	0.748	0.344

Tabell 6.4: Värden för strikt tf-idf med semi-bijektiv matchning

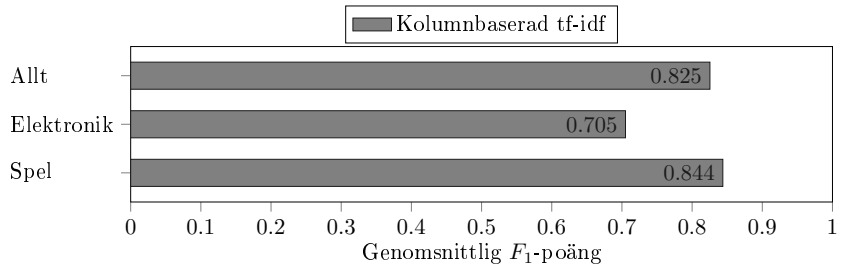
6.4 Booleansk tf-idf med bijektiv matchning



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.757	0.738	0.276	0.735	0.766	0.271	0.746	0.752	0.274
Elektronik	0.871	0.892	0.243	0.871	0.882	0.266	0.871	0.887	0.254
Spel	0.744	0.732	0.277	0.733	0.754	0.243	0.733	0.743	0.260

Tabell 6.5: Värden för booleansk tf-idf med bijektiv matchning

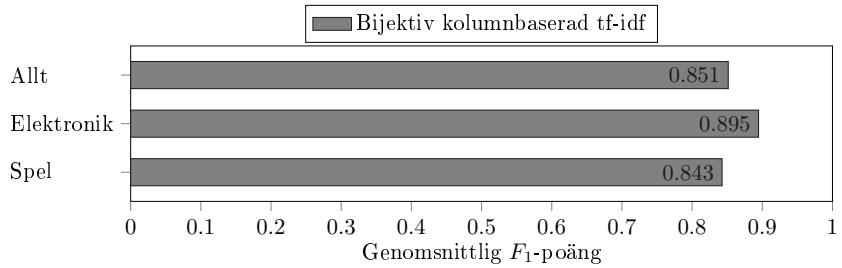
6.5 Kolumnbaserad tf-idf



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.838	0.818	0.491	0.812	0.840	0.493	0.825	0.829	0.492
Elektronik	0.730	0.746	0.483	0.680	0.759	0.545	0.705	0.752	0.514
Spel	0.857	0.842	0.487	0.830	0.862	0.527	0.844	0.852	0.507

Tabell 6.6: Värden för kolumnbaserad tf-idf

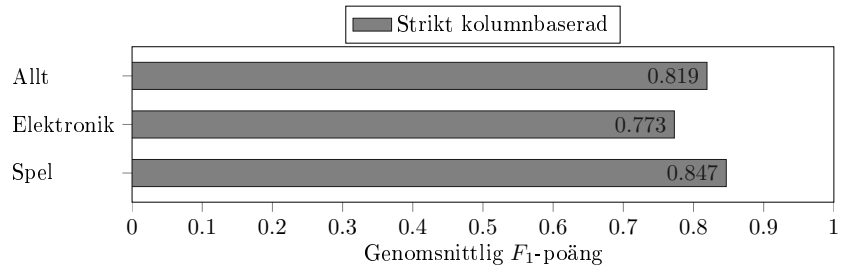
6.6 Bijektiv kolumnbaserad



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.854	0.838	0.507	0.832	0.861	0.534	0.843	0.85	0.52
Elektronik	0.897	0.912	0.552	0.893	0.897	0.553	0.895	0.904	0.553
Spel	0.852	0.855	0.475	0.851	0.855	0.454	0.851	0.855	0.465

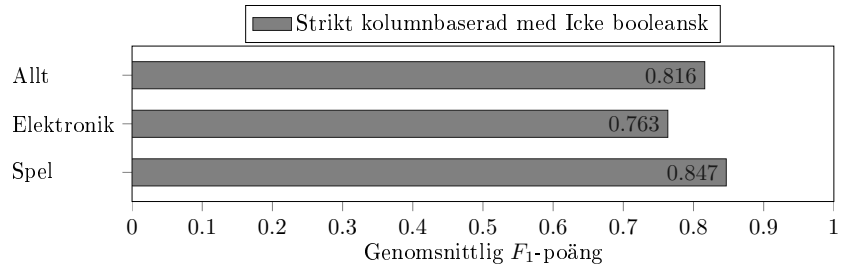
Tabell 6.7: Värden för bijektiv kolumnbaserad tf-idf

6.7 Strikt kolumnbaserad



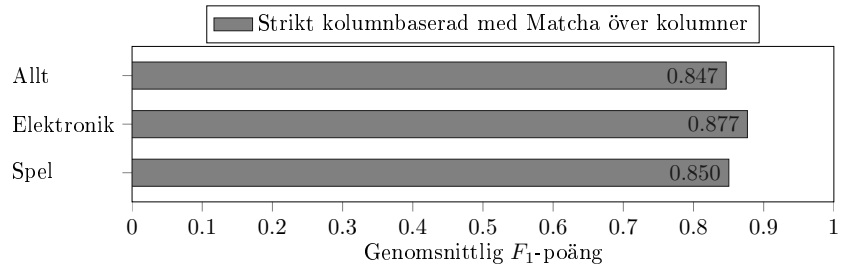
	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.829	0.814	0.352	0.809	0.837	0.378	0.819	0.825	0.365
Elektronik	0.788	0.8	0.387	0.758	0.812	0.407	0.773	0.806	0.397
Spel	0.846	0.852	0.403	0.847	0.852	0.452	0.847	0.852	0.428

Tabell 6.8: Värden för strikt kolumnbaserad



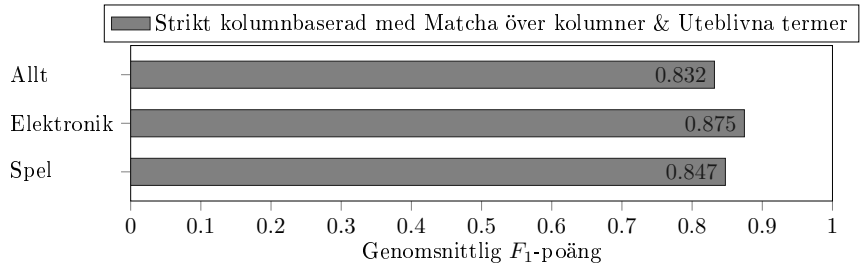
	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.822	0.812	0.352	0.81	0.833	0.441	0.816	0.822	0.397
Elektronik	0.769	0.8	0.387	0.758	0.794	0.407	0.763	0.797	0.397
Spel	0.846	0.852	0.403	0.847	0.852	0.452	0.847	0.852	0.428

Tabell 6.9: Värden för strikt kolumnbaserad med Icke booleansk



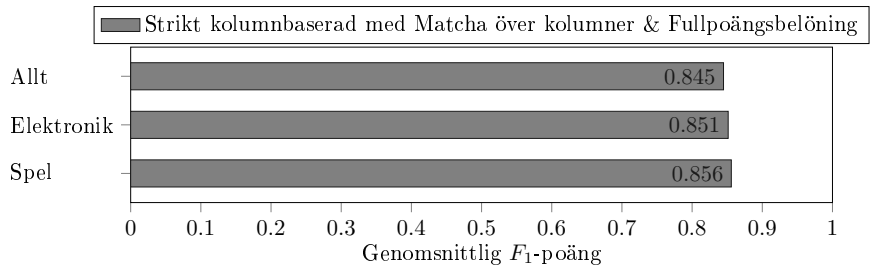
	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.848	0.847	0.553	0.845	0.854	0.538	0.847	0.85	0.546
Elektronik	0.857	0.912	0.572	0.897	0.881	0.542	0.877	0.897	0.557
Spel	0.849	0.861	0.482	0.852	0.853	0.42	0.85	0.857	0.451

Tabell 6.10: Värden för strikt kolumnbaserad med matcha över kolumner



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.83	0.847	0.645	0.833	0.842	0.626	0.832	0.844	0.635
Elektronik	0.852	0.912	0.663	0.897	0.871	0.661	0.875	0.892	0.662
Spel	0.846	0.861	0.615	0.849	0.854	0.574	0.847	0.857	0.595

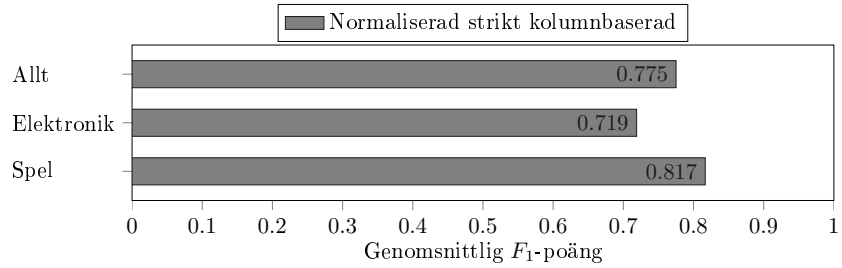
Tabell 6.11: Värden för strikt kolumnbaserad med Matcha över kolumner & Uteblivna termer



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.848	0.844	0.561	0.842	0.855	0.535	0.845	0.849	0.548
Elektronik	0.821	0.893	0.599	0.881	0.862	0.584	0.851	0.877	0.592
Spel	0.851	0.861	0.518	0.861	0.852	0.465	0.856	0.856	0.492

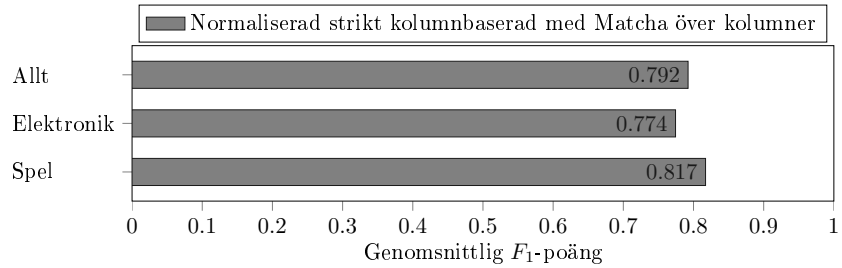
Tabell 6.12: Värden för strikt kolumnbaserad med Matcha över kolumner & Fullpoängsbelöning

6.8 Normaliserad strikt kolumnbaserad



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.764	0.798	0.36	0.786	0.783	0.485	0.775	0.79	0.422
Elektronik	0.679	0.759	0.661	0.759	0.727	0.656	0.719	0.743	0.658
Spel	0.813	0.835	0.359	0.82	0.819	0.423	0.817	0.827	0.391

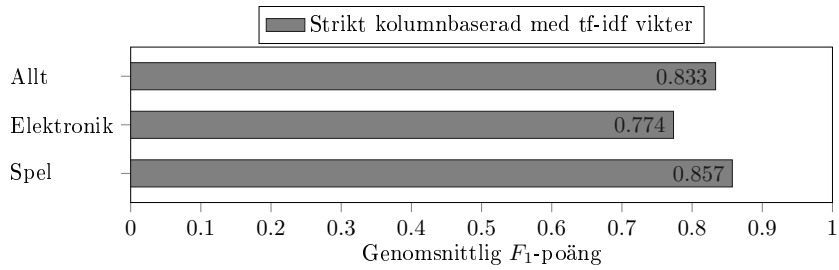
Tabell 6.13: Värden för normaliserad strikt kolumnbaserad



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.792	0.799	0.532	0.793	0.8	0.485	0.792	0.799	0.508
Elektronik	0.762	0.812	0.665	0.787	0.762	0.681	0.774	0.787	0.673
Spel	0.821	0.824	0.36	0.814	0.823	0.399	0.817	0.823	0.379

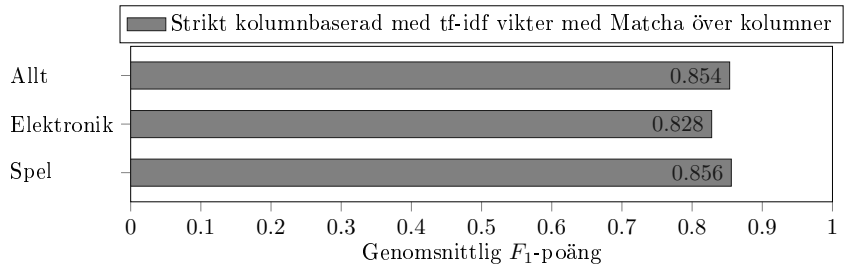
Tabell 6.14: Värden för normaliserad strikt kolumnbaserad med Matcha över kolumner

6.9 Strikt kolumnbaserad med tf-idf vikter



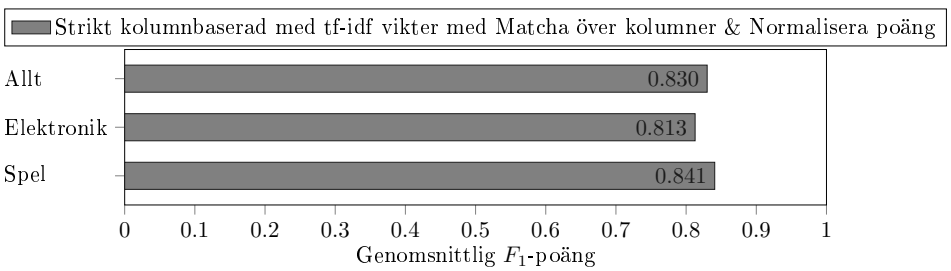
	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.837	0.844	0.437	0.83	0.842	0.512	0.833	0.843	0.475
Elektronik	0.769	0.831	0.467	0.778	0.794	0.423	0.774	0.812	0.445
Spel	0.852	0.863	0.421	0.863	0.855	0.405	0.857	0.859	0.413

Tabell 6.15: Strikt kolumnbaserad med tf-idf vikter



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.858	0.851	0.522	0.849	0.862	0.533	0.854	0.857	0.528
Elektronik	0.794	0.881	0.564	0.862	0.847	0.589	0.828	0.864	0.576
Spel	0.859	0.861	0.521	0.852	0.864	0.552	0.856	0.863	0.537

Tabell 6.16: Strikt kolumnbaserad med tf-idf vikter med Matcha över kolumner



	Körning 1			Körning 2			Genomsnitt		
Dataset	F_1	F_1^{T}	k	F_1	F_1^{T}	k	F_1	F_1^{T}	k
Allt	0.832	0.828	0.586	0.828	0.841	0.575	0.83	0.835	0.58
Elektronik	0.806	0.853	0.519	0.82	0.839	0.579	0.813	0.846	0.549
Spel	0.848	0.844	0.601	0.833	0.857	0.486	0.841	0.851	0.543

Tabell 6.17: Strikt kolumnbaserad med tf-idf vikter med Matcha över kolumner & Normalisera poäng

Litteraturförteckning

[1] http://www.gs1.org/about. [Online; accessed 2013-Augusti-15].

[2] http://msdn.microsoft.com/en-us/library/w0x726c2.aspx. [Online; accessed 2013-Augusti-15].

[3] http://www.cs.cmu.edu/~schneide/tut5/node42.html. [Online; accessed 2013-Juni-15].

[4] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html#hashCode. [Online; accessed 2013-Juni-10].

[5] Steven M. Beitzel. On understanding and classifying web queries, 2006.

[6] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[7] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[8] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.