Hjälpmedel för regelkonstruktion

- verktyg för att underlätta skapande av regler till Granska

Aids for rule construction

- tools to simplify creation of rules for Granska

Examensarbete i datalogi av Magnus Johansson (d97-mjo@nada.kth.se)

Handledare: Viggo Kann

Examinator: Stefan Arnborg

2002-08-06

Hjälpmedel för regelkonstruktion - verktyg för att underlätta skapande av regler till Granska

Sammanfattning

Granska är ett system för grammatikkontroll utvecklat vid KTH. Granska försöker finna mönster i text, exempelvis språkliga felaktigheter, med hjälp av regler. Reglerna konstrueras manuellt, vilket är tidskrävande. Detta examensarbete beskriver ett antal verktyg i form av datorprogram som kan underlätta arbetet och hjälpa den som skall konstruera regler.

Uppsatsen beskriver implementationen av de datorprogram som tagits fram. Det första verktyget är ett verktyg för felannotering av texter. Annoteringen utförs för att man skall få en korpus som man senare kan utvärdera sina regler gentemot. Dessutom beskrivs ett verktyg för utvärdering av regler mot en annoterad korpus. Verktyget presenterar ett antal rapporter där regelkonstruktören kan analysera sina regler och se hur de kan förbättras.

Vidare undersöks hur man med hjälp av maskininlärning kan låta datorn själv konstruera regler endast med tillgång till en felannoterad korpus. Metoden som analyseras bygger på att man söker meningar med liknande uppbyggnad och skapar regler som känner igen dessa likheter.

Aids for rule construction – tools to simplify creation of rules for Granska

Abstract

Granska is a system for performing grammatical analysis and checking of texts in Swedish. Granska was developed at KTH. The grammatical checking tries to find patterns in the text, for instance syntactic errors, by using rules. These rules are constructed manually, which takes time and resources. This master thesis describes some computer programs that can aid a person in constructing rules for Granska.

The implementation of the computer programs developed is described. The first program is a tool for annotating texts with grammatical errors. The annotation is done in order to get a corpus that can be used to evaluate the rules constructed for Granska. Aside from the annotation tool, a tool for evaluating rules is described. This tool presents a number of reports that can be used by the user to analyze his rules and perhaps give a hint on how to improve them.

Finally, this paper examines how machine learning techniques can be used to automatically generate rules for use with Granska. The computer only has access to a large corpus annotated with grammatical errors. The method used is based on alignment of sentences. Two sentences with a similar structure is found, and rules are created that recognize these sentences.

Innehåll

Inledning....5
Bakgrund....6
Utdata från Granska....8
Hjälpmedel för regelkonstruktion....10
Annoteringsverktyg....11
Statistik och utvärdering....17
Automatiseringsverktyg....22
Administrationskonsol....24
Automatisk konstruktion av regler....26
Transformationsbaserad inlärning....27
Alignment based training....29
Slutsats....35
Referenser....36

Inledning

Datoriserade verktyg för naturliga språk blir allt vanligare; både rättstavning och grammatikkontroll finns i moderna ordbehandlingsprogram. Granska är ett system för grammatikkontroll som är utvecklat vid KTH. Granska bygger på system av regler som skall känna igen felaktiga, men även korrekta, grammatiska konstruktioner. Reglerna ger även möjlighet till att generera förslag på hur felaktigheterna skall rättas. Granskas regler konstrueras idag för hand. Det manuella arbetet är tidskrävande och det är detta som detta examensarbete skall försöka lösa.

Syftet med detta exjobb är att utveckla prototyper av några verktyg som skall hjälpa den person som konstruerar regler med konstruktionsarbetet. Dessa verktyg kan användas för att snabbare förbättra de regler som idag redan finns konstruerade för det svenska språket, dels kan verktygen användas för att bygga ut Granska med stöd för andra språk än svenska. Dessutom utvärderas huruvida en enkel metod att låta datorn själv konstruera regler kan fungera. Det är svårt att få en helt automatisk konstruktion av regler att fungera lika bra och lika effektiv som manuellt konstruerade regler, men de automatiskt konstruerade reglerna skulle kunna hjälpa regelkonstruktören att finna alla de regler som behövs för att detektera alla typer av grammatiska fel som är vanliga.

Det viktigaste problemet som detta examensarbete försöker lösa är felannotering av texter och utvärdering av regler. Felannoteringen krävs för att man på ett korrekt sätt skall kunna utvärdera reglerna. Att utvärdera regler är viktigt för att kunna se vilka regler som saknas och hur man skall kunna förbättra redan existerande regler.

De verktyg som implementeras är för det första ett annoteringsverktyg för att markera felaktigheter i databaser med texter, så kallade korpusar. Dessa markeringar kan sedan användas för att verifiera hur bra grammatikgranskningen i Granska fungerar. Det andra verktyget är ett statistikverktyg som utvärderar hur bra grammatikgranskningen fungerar i förhållande till de annoterade felen. Dessutom implementeras ett verktyg som automatiserar arbetet med att testa olika regler och varianter på regler, samt ett verktyg som används för att starta och konfigurera de andra verktygen.

Bakgrund

Granska är ett verktyg för grammatikgranskning av främst svenska texter (se t.ex. Carlberger et al. 2000 eller Domeij et al. 1999). Granska försöker känna igen felaktiga och korrekta meningar med hjälp av speciella regler. Reglerna bygger på mönster som eftersöks i meningarna. Mönstren kan vara enkla, som specifika ord, ordsekvenser eller reguljära uttryck, men även mer komplicerade och grunda sig på olika grammatiska attribut hos orden i texten, såsom ordklass, numerus och species.

Granska består huvudsakligen av fyra moduler. Den första modulen är en tokeniserare, vars uppgift är att dela upp texten i *tokens*, bland annat ord, meningar och skiljetecken. Tokeniseraren rensar även bort skräptecken och levererar en tydlig representation av texten till nästa modul, taggaren.

Taggarens uppgift är att tilldela värden på olika attribut till orden i texten (Carlberger & Kann, 1999 samt Carlberger & Kann, 2000). Attributen som tilldelas är bland annat ordklass, numerus och species. Taggaren använder ett lexikon där många ords attribut finns att slå upp, men flertalet ord kan dock ha olika attribut beroende hur ordet används i meningen. Betrakta exempelvis ordet *en* i meningen *En grön en stod i trädgården*. Det första *en* beskriver ett antal och det andra är en trädart. Taggaren gör en statistisk analys för att hitta den tilldelning av attribut till orden i meningen som är den mest sannolika. Analysen bygger på känd statistik om hur sannolikt det är att exempelvis ett substantiv i singular följs av ett verb i preteritum. Många ord saknas helt i lexikonet, det kan exempelvis röra sig om sammansättningar, felstavningar, facktermer eller egennamn. Taggaren försöker dock tilldela värden på attributen även till dessa ord genom att bland annat studera ordets olika ändelser.

De attribut som taggaren tilldelar orden används sedan av nästa modul, regel-matcharen. Regelmatcharens uppgift är att med hjälp av en uppsättning speciella regler leta efter meningar som känns igen av någon av dessa regler. Reglerna kan matcha meningar genom att titta på bland annat sekvenser av ord, ordstammar och alla de andra attribut som taggaren har tilldelat (se t.ex. Knutsson, 2001).

Slutligen finns en modul för att utföra rättstavning. Rättstavningsmodulen i Granska är en version av ett fristående rättstavningssystem för svenska, också utvecklat vid KTH, som heter Stava (Kann et al. 1998). Stava används för att hitta ord som är felstavade. Stava hanterar även förslag på hur felstavningarna skall rättas.

Regler

Låt oss studera hur en regel i Granska kan se ut. Nedan visas en regel som känner igen felaktiga meningar, såsom *Kom hit snabbt som bara möjligt* där ordet *så* saknas. Regeln föreslår också en förändring till *Kom hit så snabbt som bara möjligt*. Regeln är tagen ur Granskas regelsamling.

```
som_möjligt@adjektiv
{
    X1(sed=sen),
    X2(text!="så")*,
    X3(wordcl=jj)+,
    X4(wordcl=nn)*,
    X5(text="som"),
    X6(wordcl=ab)*,
    X7(text="möjligt")
-->
    mark(X3 X5 X7)
    corr(X3.insert("så"))
    info("Bortglömt ord" italics("så") "?")
    detect("Kom hit snabbt som bara möjligt")
    action(scrutinizing)
}
```

Vi ser att regeln har ett namn, som består av ett regelnamn och en regelkategori, uppdelat med hjälp av ett snabel-a. Regeln består av två delar, dels en matchningsdel, dels en åtgärdsdel. Delarna är avgränsade med en stiliserad pil, `-->`.

Matchningsdelen består av ett antal matchningsvariabler, som här heter X1, X2, …, X7. Efter varje matchningsvariabel följer, inom parentes, ett matchningsvillkor. Exempelvis matchar `sed=sen` meningsbörjan, `text!="så"` alla ord som inte är ordet *så* och `wordcl=jj` alla ord som är taggade med ordklassen adjektiv. Efter matchningsuttrycket följer ibland en markering som markerar hur många ord som förväntas matchas, * betyder att noll eller flera ord förväntas matchas, medan + betyder att ett eller flera ord förväntas matchas. Sammanfattningsvis kan man säga att matchningsdelen i regeln ovan letar efter meningar där meningsstart följs av ett antal ord som inte är ordet *så*. Därefter följer ett eller flera adjektiv, följt av noll eller flera substantiv, följt av ordet *som*. Sedan följer noll eller flera adverb följt av ordet *möjligt*.

Åtgärdsdelen består av ett antal instruktioner som exempelvis markerar ord för presentation för användaren eller genererar rättningsförslag. I regeln ovan börjar åtgärdsdelen med att de ord som matchats av matchningsvariablerna X3, X5 och X7 markeras. Därefter genereras ett rättningsförslag där ordet *så* infogas före de adjektiv som matchats av X3. Efter rättningsförslaget finns en instruktion som talar om vilket meddelande som skall presenteras för användaren. Slutligen finns instruktionen `action(scrutinizing)` som berättar för Granska att regeln är en granskningsregel som försöker finna grammatiska fel. Det finns även regler som försöker hitta korrekt konstruerade meningar och på så sätt förhindra att andra regler detekterar meningen som en potentiell felaktighet.

Utdata från Granska

Traditionellt sett har den enda möjligheten att ta del av Granskas logik och kunskap om en text varit att bygga program och applikationer som moduler i Granska. Delvis beroende på detta exjobbs behov av att kommunicera med Granska har Johnny Bigert under hösten 2001 implementerat ett exportformat av Granskas fulla kunskap om en text till ett XML-format. Med hjälp av detta data och denna kunskap har jag på enklare sätt kunnat bygga mina prototyper utan att behöva förstå allt om hur Granska är implementerat och fungerar.

För att se vilken information som finns att tillgå visas här ett exempel på utdata från Granska för en liten text med endast en mening.

```
<Root>
    <scrutinizer>
        <s ref="400">
            <tokens>19</tokens>
            <text>Denna rapport handlar om hur SEB:s
                  arbetsmiljöprofil,
                  deras stadgar och [policies ]ser ut.</text>
            <heading />
            <contents>
                <w no="0" tag="sen.per" lemma="$.">$.</w>
                <w no="1" tag="sen.per" lemma="$.">$.</w>
                <w no="2" ref="0"
                   tag="dt.utr.sin.def"
                   lemma="denna">Denna</w>
                       .
                       .
                <w no="16" ref="87" tag="mad" lemma=".">.</w>
                <w no="17" tag="sen.per" lemma="$.">$.</w>
            </contents>
            <gramerrors>
                <gramerror>
                    <marked>[policies ]</marked>
                    <info>Okänt ord</info>
                    <rule>stav1@stavning</rule>
                    <url>http://www.nada.kth.se/~viggo...</url>
                    <urltext>Stava</urltext>
                    <suggestions>
                        <sugg>[polisciss]</sugg>
                        <sugg>[policiss]</sugg>
                        <sugg>[polisids]</sugg>
                        <sugg>[polisils]</sugg>
                    </suggestions>
                    <marked_section>
                        <mark begin="13" end="13" />
                    </marked_section>
                </gramerror>
            </gramerrors>
        </s>
    </scrutinizer>
</Root>
```

Man kan se hur Granskas taggare taggar en text, exempelvis kan vi se att ordet *Denna* taggas som en determinerare i utrum, singular och definit form. Vi ser också alla potentiella grammatiska fel som Granska hittar och rättningsförslag. I exemplet ovan har Granska hittat vad den tror är ett stavfel av ordet *policies*. Vi ser att regeln stav1@stavning har använts, samt att det är ord nummer 13 som har markerats. Vidare ser vi att Granska föreslår att stavfelet skall ändras till antingen *polisciss*, *policiss*, *polisids* eller *polissils*.

Informationsmängden är väldigt stor. Utdata från lite längre texter ger snabbt upphov till väldigt stora filer. XML-filerna blir någonstans mellan femton och tjugo gånger så stora som korresponderande text. Mängden data gör att prototyperna i detta exjobb fungerar bäst med texter som är mindre än cirka hundra sidor.

Hjälpmedel för regelkonstruktion

Det finns många delar i regelkonstruktionsarbetet som kan förenklas med hjälp av datorverktyg. Regelkonstruktionen består förenklat av tre moment. Först författas regeln. Detta görs vanligtvis med hjälp av ett vanligt textredigeringsvektyg. Därefter vill man verifiera att regeln fungerar som den ska. Verifieringen har hittills främst fungerat så att man testar sina regler med Granska på ett antal texter av olika typ och sedan manuellt verifierat att regeln hittar de fel som den är ämnad att hitta. Slutligen kommer fasen där man vill förbättra sin regel med hjälp av det man lärt sig av test-körningen. Man förbättrar och testar sedan sin regel tills man är nöjd med vilka meningar som matchas.

Till hjälp med författandet av regler skulle man kunna tänka sig ett mer intelligent redigeringsverktyg för granskningsregler än en vanlig textredigerare. En annan sak som kan underlätta författandet är tillgång till all den information om en text som Granska besitter; det är framförallt intressant hur Granska har valt att tagga de meningar som man vill känna igen med sin regel.

Vad gäller verifieringsfasen kan datorverktyg exempelvis sammanställa statistik om hur många meningar som Granska detekterat med hjälp av vilka granskningsregler. Om man även hade ett facit, det vill säga hade definierat vilka meningar som Granskas regler skulle matcha, så kan också statistik beräknas om vilka meningar som reglerna matchade, men som egentligen inte är felaktiga. En analys av varför en given mening detekterats av en given regel kan också vara intressant för bland annat felsökning.

Den sista fasen, förbättringsfasen, skulle kunna underlättas med hjälp av ett verktyg som analyserar vilka regler i en regelsamling som verkligen används. En annan möjlighet vore ett program som undersöker små variationer av en regel för att se om en alternativ formulering kan innebära en förbättring. Ett alternativ till detta vore att ha ett verktyg som snabbt kan söka igenom alternativa formuleringar av en regel, men att dessa formuleringar inte skapas automatiskt, utan föreslås av regel-konstruktören.

I detta exjobb har jag valt att koncentrera mig på endast en del av dessa verktyg. Jag upplevde efter diskussioner med Ola Knutsson att den viktigaste biten att automatisera är att kontrollera resultatet av regelsamlingen gentemot ett facit, en korpus med annoterade fel. För att detta skall vara möjligt krävdes tre verktyg, ett

program för att annotera texter med fel, samt ett program som analyserar Granskas resultat gentemot den annoterade texten och skapar en rapport. Dessutom behövs, som vi skall se senare, ett program för att koppla de annoterade texterna mot Granska. Annoteringsverktyget kan också vara intressant för andra projekt, dels inom ramen för Granska, dels för andra språkteknologiska tillämpningar, eftersom datorläsbara annoteringar är användbara och ofta nödvändiga för exempelvis fortsatt utveckling inom automatisk regelkonstruktion samt för att utvärdera andra metoder för att detektera fel än med hjälp av granskningsregler.

Jag har också valt att implementera ett verktyg för att regelkonstruktören snabbt skall kunna testa olika regelformuleringar. Konstruktören beskriver vilka varianter av regler som skall testas varvid det på ett enkelt sätt kan prövas olika formuleringar av regeln på olika typer av texter. Med hjälp av statistikverktyget kan man sedan analysera vilka formuleringar som fungerar bäst.

Till sist har jag också implementerat ett gemensamt grafiskt gränssnitt för att hantera de olika applikationerna. Programmet hanterar allt data som applikationerna behöver och allt data de genererar. Dessutom kan man starta applikationerna. Systemet erbjuder också en enkel funktion för att hantera flera projekt.

Eftersom Granska genererar XML som utdata har jag valt att använda XSLT[1] som en bas i alla mina prototyper. XSLT är ett funktionellt programspråk vars syfte är att översätta från ett XML-format till ett annat. Det är alltså mycket enkelt att sammanställa information från flera XML-filer och skapa exempelvis webbsidor (som också är en form av XML) som resultat. Eftersom data är tillgängligt som XML och XSLT lämpar sig för att transformera från XML till HTML så är de största delarna av verktygen byggda som webbsidor.

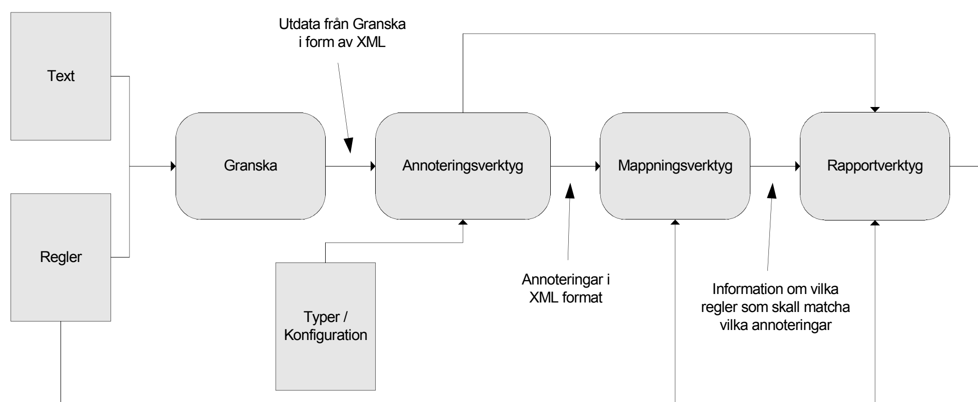
I figur 1 visas hur de olika verktygen samverkar och vilket data som används.

Annoteringsverktyg

Basen för många tillämpningar inom naturliga språk är tillgång till stora mängder text eller tal. Dessutom krävs ofta att texten är uppmärkt för den speciella tillämpningen. Om man exempelvis vill utvärdera rättstavningssystem kan det vara lämpligt att märka vilka ord som är rättstavade och vilka som är felstavade.

För att kunna utvärdera grammatikgranskning vill man markera vilka meningar som är grammatiskt korrekta och vilka meningar och ord som innehåller felaktiga grammatiska konstruktioner. Med både texten och annoteringarna är det lätt att utvärdera hur pass bra grammatikgranskningen fungerar, man jämför helt enkelt de annoterade felen med de fel som grammatikgranskningen funnit.

1 XSLT står för eXtensible Stylesheet Language Transformation.



Figur 1 Figuren visar hur verktygen används tillsammans. Granska använder sig av regler och text som skall grammatikgranskas. Annoteringsverktyget använder sig av utdata från Granska samt information om vilka feltyper som skall annoteras. Utdata från annoteringsverktyget kan tillsammans med utdata från Granska och mappningsverktyget användas för att skapa statistik för att kunna analysera täckning och precision hos regeluppsättningen.

För det svenska språket är tillgången till stora mängder text, som är annoterad för grammatikgranskning, begränsad. Detta gör att det varit svårt och tidsödande att utvärdera system såsom Granska.

För att snabbare kunna skapa annoterade texter behöver man ha ett verktyg som kan användas för att konstruera datorläsbara annoteringar. Ett önskemål är att annoteringen enkelt skall kunna utföras av ett flertal olika personer för att snabbare kunna få tillgång till stora mängder annoterat material. Man vill därför ha ett enkelt sätt att distribuera annoteringsarbetet. Det innebär också att annoteringsverktyget helst inte skall kräva att annoteraren har en möjlighet att köra Granska eller att kräva att annoteraren har en mängd andra program installerade på sin dator.

Det finns inget vitt spritt format på hur annoteringarna skall lagras elektroniskt som annoteringsverktyget skall rätta sig efter. Det finns inte heller någon entydig feltypologi, det vill säga uppdelning av grammatiska fel som annoteringarna skall följa.

Konstruktion

I den prototyp av ett annoteringsverktyg som jag implementerat så valde jag att en annotering pekar ut ett antal ord i en mening. Samma ord kan vara del i flera annoteringar. En annotering kategoriseras av en typ som är en kombination av del-typer från flera olika kategorier. Vidare innehåller också en annotering vanligtvis en referens till vem som skapat annoteringen, ett ersättningsförslag samt en kommentar.

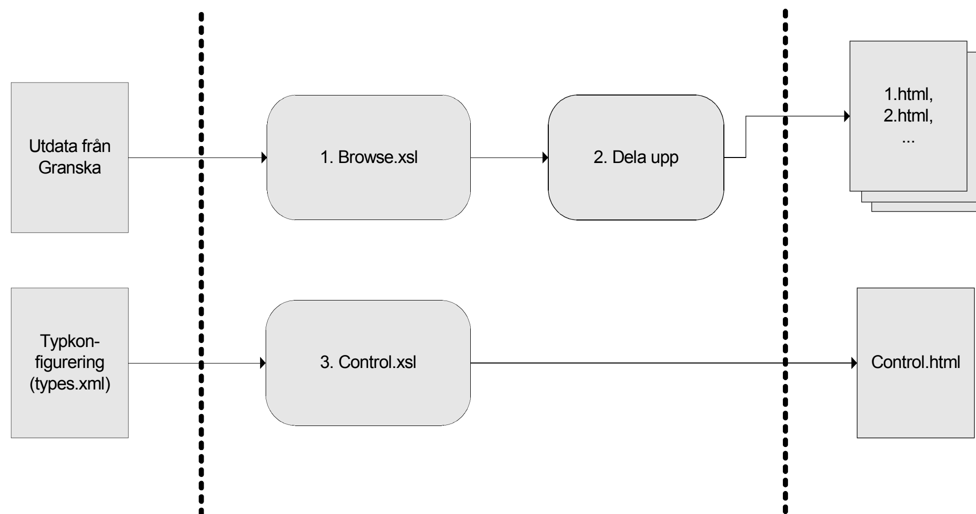
Om man önskar ytterligare attribut såsom exempelvis annoteringsdatum så tillåter systemet att användaren definierar egna attribut. Användaren kan också enkelt definiera vilka kategorier och deltyper som skall finnas.

Annoteringarna lagras i ett XML-format och hanteras separat från själva texten. Texten behålls helt intakt. Annoteringarna refererar till meningar med hjälp av meningens position i texten räknat som antalet tecken från textens början. Annoteringarna pekar ut ord i meningarna genom att numrera meningens ord från ett och uppåt. Ett exempel på en annoteringsfil visas nedan.

```
<Root>
    <s ref="4952">
        <annot>
            <position pos="2" />
            <position pos="3" />
            <type>DELETION</type>
            <text>Jag ser polis åka förbi</text>
            <comment></comment>
            <suggestion>ser en polis åka förbi</suggestion>
            <annotatedWords>ser polis</annotatedWords>
        </annot>
    </s>
    <s ref="7563">
        <annot>
            <position pos="33" />
            <type>SUBSTITUTION</type>
            <type>SPELL</type>
            <text>
                samt fortlöpande informera arbetsmiljööombudet
                om förhållanden
            </text>
            <comment>arbetsmiljöombudet är felstavat</comment>
            <suggestion>arbetsmiljöombudet</suggestion>
            <annotatedWords>arbetsmiljööombudet</annotatedWords>
        </annot>
    </s>
</Root>
```

Vi ser att två felaktigheter är annoterade. Dels innehåller mening på position 4952 ett fel där man glömt ordet *en* i meningen *ser polis åka förbi*. Nästa fel finns i mening på position 7563 där ordet *arbetsmiljöombudet* är felstavat. Den andra meningen kategoriseras av en feltyp som består av deltyperna *substitution* och *spell*.

Vi har sett ovan att annoteringarna refererar till ord och meningar. För att annoteringarna skall vara användbara måste dessa begrepp vara entydigt definierade. Eftersom annoteringarna primärt skall användas för att utvärdera Granska valdes att annoteringsverktyget skall använda samma definitioner som Granska använder. Genereringen av annoteringsverktyget startar därför med att vi applicerar Granska på den text som skall annoteras. På så sätt kan vi använda Granskas kunskap om textens struktur i meningar och ord. Dessutom hanterar Granska strukturer såsom paragrafer och rubriker som används av annoteringsverktyget för att skapa en mer attraktiv och överskådlig presentation av texten som skall annoteras.



Figur 2 Annoteringsverktyget byggs upp med hjälp av utdata från Granska samt information om vilka feltyper som skall annoteras. Utdata från Granska används för att bygga upp de HTML-sidor som innehåller själva texterna, steg 1 i figuren. Eftersom man med XSLT endast kan generera en utdatafil, så måste man dela upp utdata i flera filer med ett speciellt program, implementerat i Java. Detta illustreras med steg 2 i figuren. Slutligen transformeras informationen om vilka feltyper som skall finnas till en egen HTML-fil som man i figur 3 kan se som ett formulär i nedre vänstra hörnet.

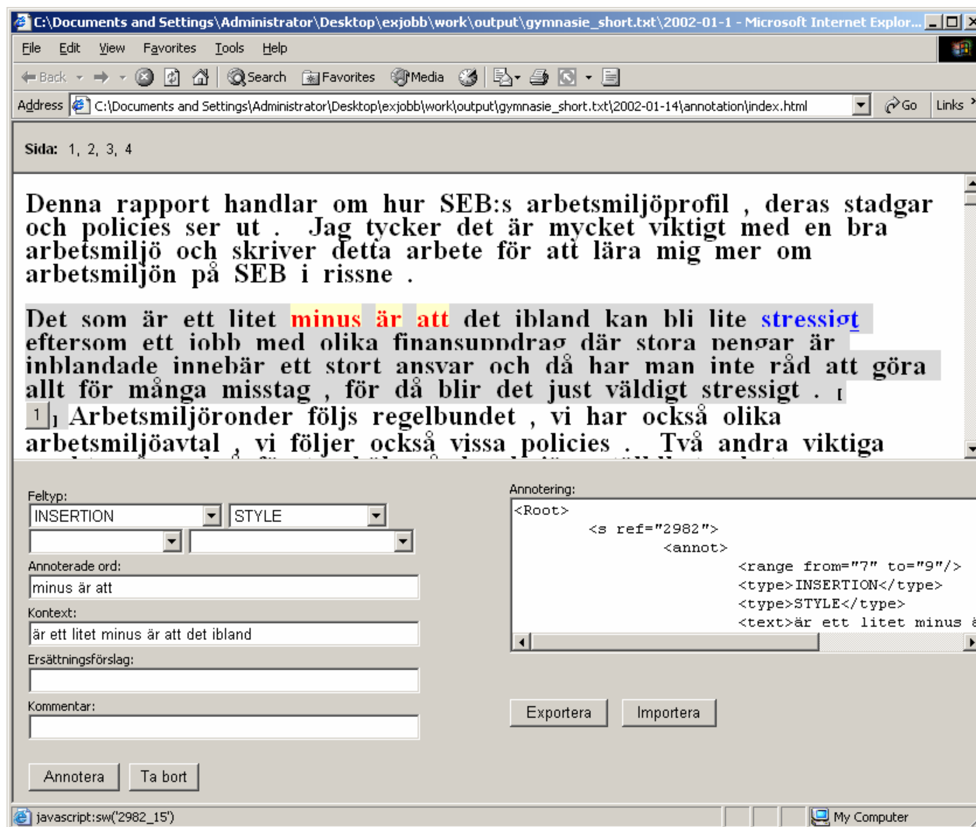
Annoteringsverktyget är byggt med hjälp av HTML och JavaScript. På så sätt kan annoteringsverktyget användas utan tillgång till något annat än en modern webb-läsare. Verktyget använder endast logik på klientsidan och kräver ingen logik på serversidan.

För varje text och varje uppsättning deltyper så måste man generera ett nytt annoteringsverktyg. Genereringen skapar de HTML-sidor som behövs för att man ska kunna annotera just den texten. Denna process kräver dock tillgång till Granska, en fungerande XSLT-processor samt en fungerande Javamiljö. När man applicerat Granska på texten som skall annoteras får man en XML-fil som innehåller Granskas fulla kunskap om texten. Tillsammans med en XML-fil som beskriver vilka feltyper som skall kunna annoteras, samt en XML-fil som beskriver eventuella extra attribut som skall vara del av en annotering så appliceras en rad XSLT-transformationer för att generera de HTML-filer som utgör annoteringsverktyget.

Genereringen av ett annoteringsverktyg för en specifik text och en specifik uppsättning feltyper sammanfattas i figur 2.

Att använda annoteringsverktyget

Annoteringsverktyget är ämnat att kunna användas av en normalt datorkunnig person som har en god förståelse för själva annoteringen. Ett exempel på hur annoteringsverktyget kan se ut visas i figur 3.



Figur 3 Ett exempel på hur det ser ut när man använder annoteringsverktyget. Annoteringarna markeras med hjälp av röd färg. Ovan är "minus är att" markerat. Annoteringens inställningar visas i nedre vänstra hörnet. I nedre högra hörnet visas annoteringarna kodat som XML. Lägg också märke till sidvalen i fönstrets översta del.

Texten som skall annoteras är uppdelad i ett lämpligt antal meningar per sida, vanligtvis finns omkring 100 meningar per sida för att annoteringsarbetet skall bli mer överskådligt. Annoteraren skapar en annotering genom att klicka på de ord i meningen som skall ingå i annoteringen. När användaren valt vilka ord som skall ingå så kan han välja de attribut som skall höra till annoteringen, exempelvis vilken typ av fel som skall annoteras, vem som har annoterat samt eventuellt rättningsförslag. När detta är färdigt trycker användaren på knappen "Annotera". Det dyker då upp en knapp invid den mening som just annoterats. Om användaren senare vill ändra eller ta bort sin annotering, så kan han trycka på knappen invid meningen för att markera annoteringen. Man kan också välja att ta bort en markerad annotering.

När alla fel har annoterats så kan användaren spara sina annoteringar i en XML-fil. Annoteringarna i form av XML skrivs då ut i ett textfält på sidan. Innehållet kan sedan kopieras för att klippas in i en fil för att sparas lokalt på användarens dator. Annoteringsverktyget kan också läsa in annoteringar som gjorts tidigare genom att importera annoteringar sparade i XML-format.

Andra möjligheter

Det finns som tidigare nämnts andra behov av annotering än annotering av grammatiska fel. Annoteringsverktyget är tillsynes hårt knutet till Granska, men annoteringsverktyget kan med små förändringar användas för andra saker än annotering av grammatiska fel.

Man kan tänka sig annotering för andra tillämpningar inom språkteknologi. Exempelvis skulle man kunna annotera ord med ordklasser, ordformer och andra grammatiska attribut för att separat kunna utvärdera Granskas taggare. De flesta typer av annoteringar som görs på mening eller ordnivå kan utföras med hjälp av annoteringsverktyget. Eftersom bland annat feltyperna är konfigurerbara så kan man lätt göra enkla anpassningar av annoteringsverktyget till den speciella tillämpningen. Dessutom kan man enkelt lägga till extra attribut som man vill koppla till annoteringen och på så sätt utföra lite mer avancerade anpassningar.

Annoteringsverktyget använder sig av utdata från Granska vilket skulle kunna innebära att det är svårt att använda systemet utan Granska. Systemet nyttjar dock ingen information som är unik för Granska. Den enda information som används är vilka meningar och ord som finns. Nedan visas en XML-fil som innehåller allt det data som krävs av annoteringsverktyget.

```
<Root>
    <scrutinizer>
        <s ref="19">
            <words>12</words>
            <heading />
            <paragraph />
            <contents>
                <w no="2" ref="0" >Arbetsmiljö</w>
                <w no="3" ref="11">,</w>
                <w no="4" ref="13>SEB</w>
                <w no="5" ref="17">-</w>
                <w no="6" ref="19">Ett</w>
                <w no="7" ref="23">arbete</w>
                <w no="8" ref="30">på</w>
                <w no="9" ref="33">Skolarkivet</w>
            </contents>
        </s>
        <s ref="105">
            <words>7</words>
            <paragraph />
            <contents>
                <w no="2" ref="0">Arbetsmiljö</w>
                <w no="3" ref="11">,</w>
                <w no="4" ref="13">SEB</w>
            </contents>
        </s>
    </scrutinizer>
</Root>
```

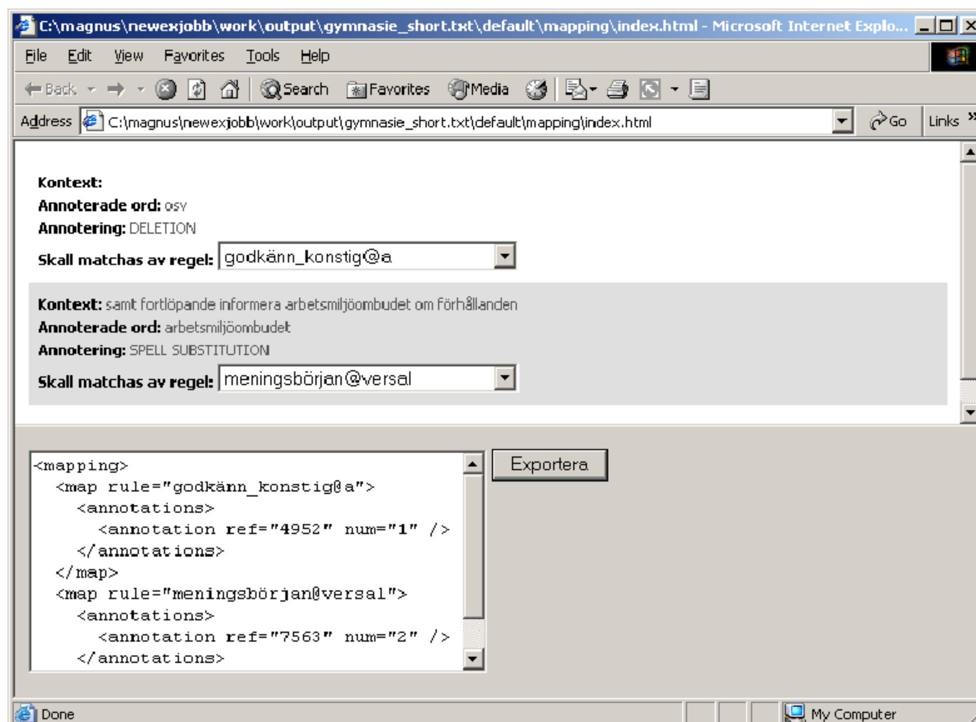
Som man ser ovan kräver annoteringen endast tillgång till uppdelning i ord och meningar. Information om stycken och rubriker, `<paragraph />` och `<heading />`, används om det är tillgängligt men är ej nödvändigt. Information om exempelvis taggarens taggning av orden i meningarna, eller eventuellt matchade fel används inte. Med hjälp av XSLT bör det vara enkelt att transformera nästan vilket annat XML-format som helst som innehåller en liknande uppdelning av ord och meningar till det format som krävs av annoteringsverktyget.

Annoteringsverktyget kan även användas till andra saker än annotering för direkta tillämpningar inom språkteknologi. Ett exempel skulle kunna vara ett datorsystem för skollärare att rätta uppsatser som är skrivna av elever. Läraren skulle på så sätt kunna analysera vilka typer av fel som eleverna gör och anpassa undervisningen därefter. Eleverna skulle också kunna se vilka fel som de har gjort och se förslag på rättningar som läraren givit. Även eleven skulle kunna studera vilka typer av fel som han eller hon gör oftast.

Statistik och utvärdering

Om man har en korpus annoterad med fel och ett grammatikgranskningsverktyg så behöver man ett verktyg för att utvärdera sitt system. Man behöver ett program som genererar statistik och talar om vilka regler som är bra (matchar många felaktiga meningar och få korrekta meningar) samt vilka som är dåliga (matchar för många korrekta meningar). Vilka typer av grammatiska fel detekteras väl av grammatik-granskaren och vilka missas? Vanligtvis vill man mäta detta i begreppen täckning och precision. Begreppet täckning visar hur pass stor del av de annoterade felen som upptäcks och beräknas som antalet upptäckta fel dividerat med antalet annoterade fel. Begreppet precision syftar till att visa hur många fel som upptäcks verkligen är riktiga (annoterade) fel. Precision mäts som antalet upptäckta annoterade fel dividerat med antalet upptäckta fel. Man önskar både hög täckning och hög precision. Precisionen kan göras godtyckligt hög om man är beredd att göra avkall på täckning. Även täckningen kan göras godtyckligt hög men med minskande precision som följd. Vanligtvis vill man prioritera en hög precision med en minskande täckning som följd.

För att kunna utvärdera Granska med hjälp av annoterade texter så måste man först bestämma huruvida ett av Granska upptäckt fel korresponderar till en annotering eller ej. Om man bestämmer sig för att det upptäckta felet korresponderar till en annotering så är det upptäckta felet korrekt, annars är det felaktigt. Det finns två svårigheter i att avgöra om Granskas markeringar korresponderar till annoteringarna. Man måste man bestämma sig för om annoteringarna och markeringarna måste markera exakt samma ord, delvis samma ord eller eventuellt bara samma mening. I statistikverktyget kan användaren själv bestämma vilken av dessa ”överensstämmelse-typer” som skall gälla.

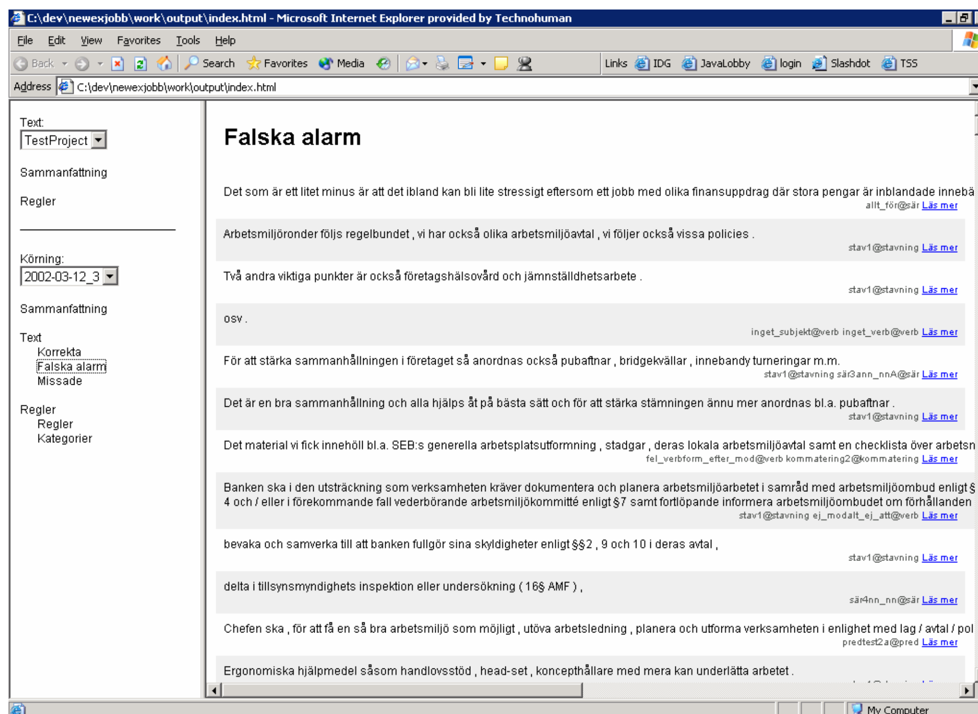


Figur 4 Överst presenteras alla tillgängliga annoteringar. Vid varje annotering visas en lista med alla regler som användaren kan välja bland. För varje annotering skall användaren välja en lämplig regel. Nederst visas den mappning som användaren just skapat som en XML-fil.

Man måste också verifiera att Granska markerar fel av samma typ som annoteringen. Eftersom annoteringen är så pass frikopplad från Granskas regler så kan man inte generellt säga att Granskas regler korresponderar direkt till de typer som används av annoteringarna. Statistikverktyget kräver istället att man för varje annotering bestämmer vilken regel som förväntas detektera det felet. Denna del är ett separat verktyg och fungerar på samma sätt som annoteringsverktyget via en HTML-sida.

Mappningsverktyget presenterar en lista på alla annoteringar som gjorts. Allt data som finns tillgängligt om annoteringen visas här; vilka ord som ingår, vilken feltyp och så vidare. Användaren ges möjlighet att för varje annotering välja, i en lista med befintliga regler, vilken regel som skall matcha den annoteringen. För att underlätta arbetet för användaren så kan man i ett moment markera att alla annoteringar av en given typ skall detekteras av en och samma regel. När användaren valt regel för varje annotering så kan han på liknande sätt som i annoteringsverktyget välja att exportera sina val till en XML-fil. I figur 4 visas ett exempel på hur mappningen mellan annoteringar och regler kan se ut.

När användaren bestämt sig för vilka regler som skall hitta vilka annoteringar så kan man generera en rapport. Rapporten innehåller information om vilka annoteringar som Granska detekterade (korrekta), vilka annoteringar som Granska inte hittade (missade) samt alla markeringar som Granska gjorde, men som inte korresponderade till någon annotering (falska alarm). Rapporten innehåller också samma information



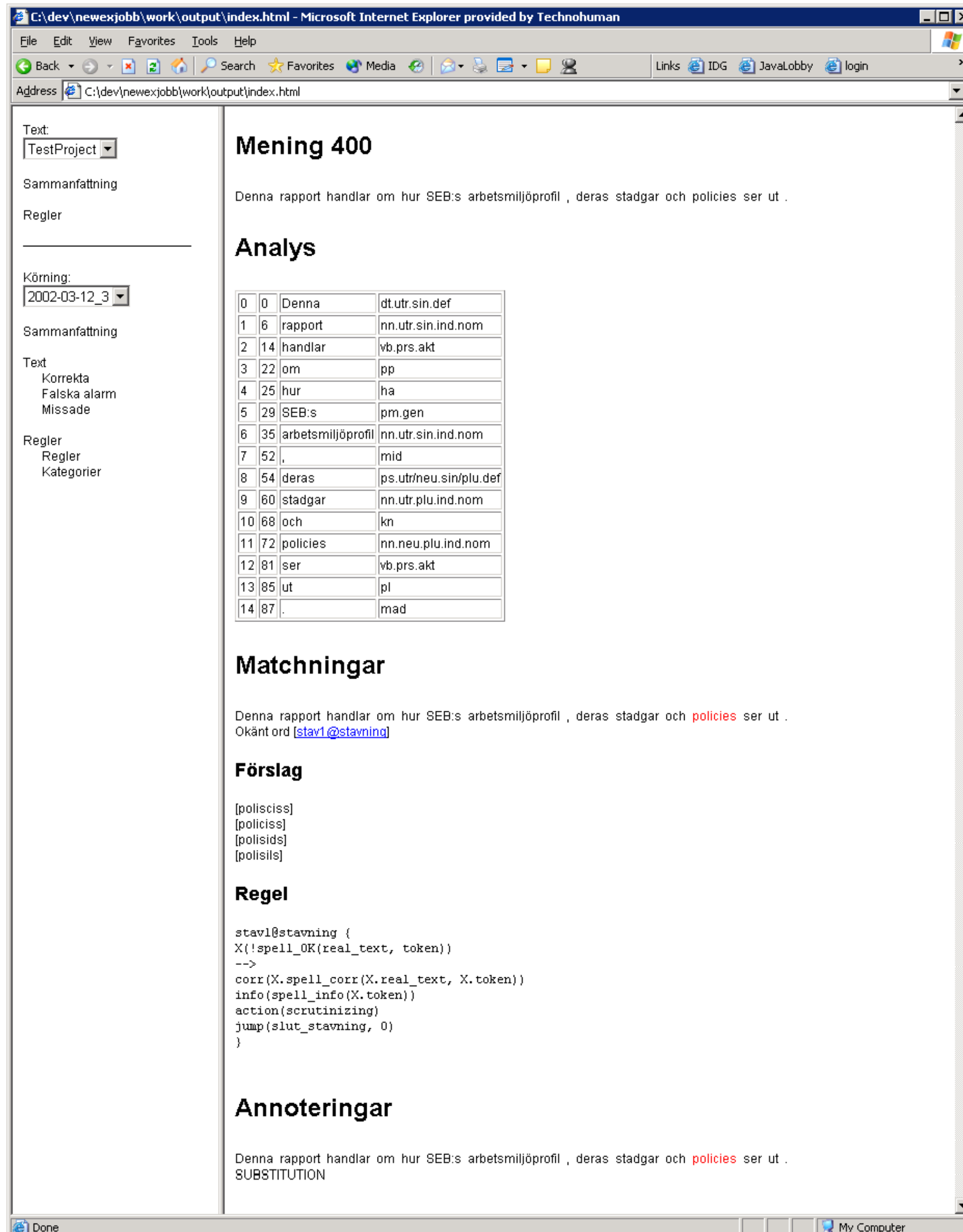
Figur 5 Till vänster ses menyn. Här kan man byta projekt och körning. Man kan också välja att få sammanställningar över vilka meningar som Granska hittat korrekt, vilka som inte hittats samt vilka som Granska har hittat men som ej var annoterade (falska alarm). Till höger kan man dels se förteckningar över meningar, dels detaljer om en specifik mening. I bilden ovan visas till höger en lista med meningar som detekterats som falska alarm.

grupperat på annoteringarnas typ eller regler i Granska. Man kan med hjälp av detta ta reda på precisionen hos regeln *subj_efter_prep@pronomen*, samt enkelt besvara frågor såsom ”Är precisionen högre vad gäller kongruensfel än stavfel?”.

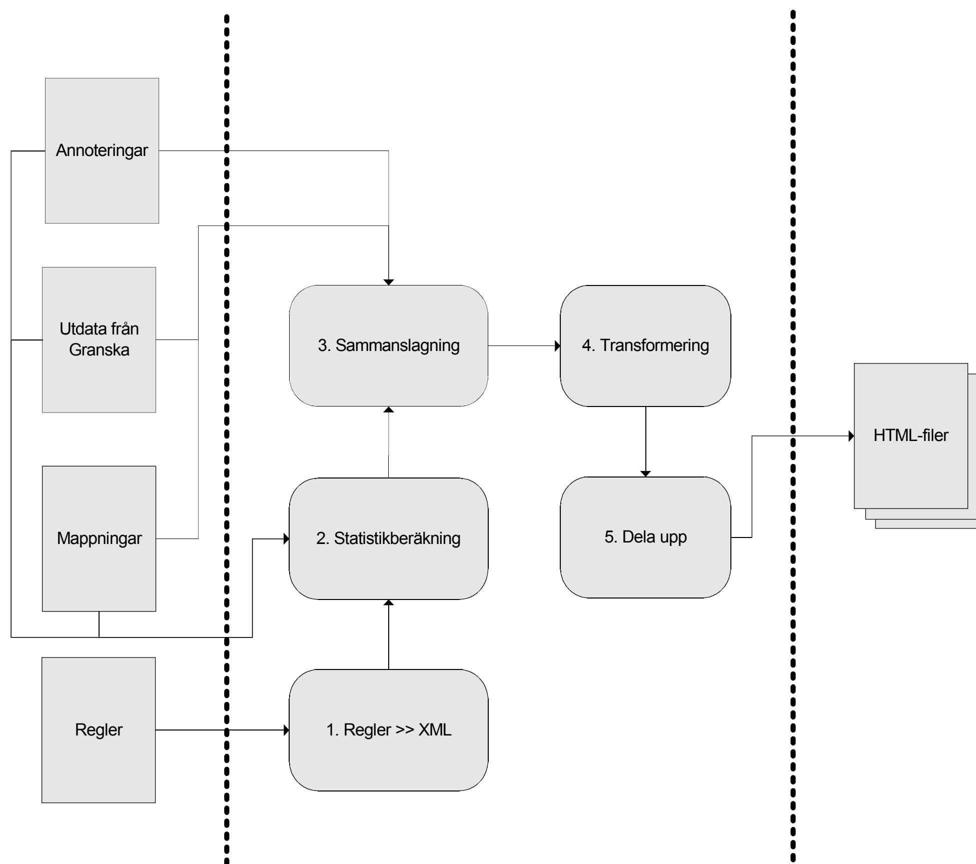
Rapporten ger också möjlighet att studera nästan allt som Granska och annoteringarna ”vet” om en given mening. Man kan se hur Granska har taggat orden i meningen, vilka regler som matchat vilka ord i meningen, hur den regeln är konstruerad samt vilka ersättningsförlag som finns. Slutligen visas också annoteringar som annoterar något ord i meningen. Exempel på hur rapporterna kan se ut visas i figur 5 och 6.

Jämförande studier

En vanlig fråga som man vill besvara är hur precisionen eller täckningen utvecklas när man ändrar på sin regelsamling, gör korrigeringar i annoteringarna eller ändrar hur pass strikt man vill vara när man bedömer om en annotering detekterats av Granska eller ej. Statistikverktyget stödjer denna typ av utvärderingar genom att i samma rapport visa alla körningar på alla texter som gjorts. Varje gång man genererar statistik så väljer man dels den text man vill analysera och dels ett namn på den körning man gör (exempelvis dagens datum). Exempelvis så kan man den 5 mars göra en körning på texten gymnasie.txt och den 7 mars gör man en annan körning på samma text fast med några nya regler. Vi får två körningar 2002-03-05



Figur 6 Här syns en annan vy i rapportverktyget. Till höger visas nu detaljer för en specifik mening. Överst ser vi en analys som visar vilka särdrag som Granska har tilldelat orden i meningen. Vi ser sedan alla matchningar som Granska har gjort, i det här fallet har Granska hittat en felstavning av ordet *policies*. Slutligen syns alla annoteringar som gjorts till den valda meningen. Vi ser här att *policies* även är annoterat, varför matchningen kan anses vara korrekt.



Figur 7 Rapportverktyget kräver utdata från Granska och mappningsverktyget. Dessutom behövs regler och annoteringar. Först konverteras regelfilen till XML. Därefter används ett Javaprogram för att med hjälp av regler, mappningar, annoteringar och utdata från Granska beräkna statistik. Utdata från detta program tillsammans med tidigare data sätts sedan samman till en enda stor fil för att XSLT-transformeringen till HTML-filer skall gå snabbare. Slutligen används XSLT för att skapa HTML-filer. Eftersom XSLT endast kan skapa en fil som utdata krävs ofta ett extra steg som delar upp en XML-fil i flera mindre.

och 2002-03-07 av texten gymnasie.txt. Man kan då i rapporten se en komparativ analys av den totala täckningen och precisionen samt se en analys för varje enskild regel. Man kan exempelvis avgöra om man lyckats öka täckning över tiden eller ej.

Konstruktion

Statistikverktyget använder sig till skillnad från annoteringsverktyget inte endast av XSLT för att generera rapporter. Uppgiften att för varje markering gjord av Granska ta reda på om det finns en matchande annotering eller ej implementerades initialt i XSLT. Körtiden blev dock alldeles för lång och implementationen blev alldeles för komplicerad då möjligheterna att bygga lämpliga datastrukturer samt göra många korsrefererande uppslagningar i XML-data med hjälp av XSLT är väldigt begränsade.

En andra implementation där ovanstående uppgift implementerades i Java gjordes. Javaprogrammet går igenom annoteringarna och utdatafilen från Granska och lägger till information om vilka annoteringar som matchats av Granska och vilka matchningar som korresponderar mot annoteringar. Programmet lägger också till summeringar över missade annoteringar, korrekta matchningar och falska alarm per regel och annoteringstyp. Ett sista steg är att läsa in regelsamlingen och konvertera den till ett XML-format så att informationen enkelt kan användas av XSLT-transformationer.

Det sista steget att generera lämpliga HTML-sidor sköts fortfarande av en serie XSLT-transformationer av annoteringarna, Granskas utdata, den extra information som Javaprogrammet genererat samt reglerna i XML-format.

Samverkan mellan de olika delarna i genereringen av statistikrapporterna samt det data som krävs visas schematiskt i figur 7.

Automatiseringsverktyg

När man konstruerar regler vill man ofta undersöka hur alternativa formuleringar av en regel påverkar resultatet. Det är också intressant att studera vad som händer om regler tas bort eller läggs till. För att slippa att göra dessa modifieringar för hand på regeluppsättningen så har jag implementerat ett enkelt automatiseringsverktyg för att underlätta detta.

Automatiseringsverktyget är starkt knutet till statistikverktyget. Varje modifiering som automatiseringsverktyget gör ger upphov till en ny körning. Statistikverktygets möjlighet att göra jämförande analyser kan sedan användas för att enkelt studera hur olika modifikationer påverkar resultatet.

Regelkonstruktören använder sig av automatiseringsverktyget genom att specificera ett antal modifikationer av regeluppsättningen i en XML-fil. Därefter kan automatiseringsverktyget utföra dessa modifikationer och efter varje modifikation köra Granska med den nya regeluppsättningen för att kunna utvärdera modifikationen. XML-filen erbjuder regelkonstruktören att göra tre typer av förändringar av regeluppsättningen. Man kan lägga till nya regler samt ta bort befintliga regler. Slutligen kan man också modifiera redan befintliga regler. Modifikationerna grupperas i namngivna grupper som sedan ger upphov till olika körningar.

Nedan visas ett exempel på hur en automatiseringsfil kan se ut:

```
<batch>
    <run name="2002-03-12#1">
        <replace-rule name="härom@sär">
            X(text="härom"),
            Y(text="kvällen" | text="dagen" | text="aftonen" |
              text="dan" | text="natten" | text="sistens" |
              text="veckan" | text="året")
-->
            corr(X.join(Y.real_text))
            detect("Härom kvällen var jag på bio.")
            action(scrutinizing)
        </replace-rule>
        <delete-rule name="xxx@yyy" />
        <insert-rule name="zzz@kkk" after-rule="stav1@stavning">
            --- regelformulering ---
        </insert-rule>
    </run>
    <run name="2002-03-12#2">
        <replace-rule name="stav1@stavning">
            --- regelformulering ---
        </replace-rule>
        <delete-rule name="xxx@yyy" />
        <insert-rule name="xx2@kkk" after-rule="stav1@stavning">
            --- regelformulering ---
        </insert-rule>
    </run>
</batch>
```

Vi ser ovan två körningar. I den första körningen görs följande:

1. Regeln *härom@sär* formuleras om. Den nya formulering syns i automatiseringsfilen ovan.

2. Vi tar bort regeln *xxx@yyy*.

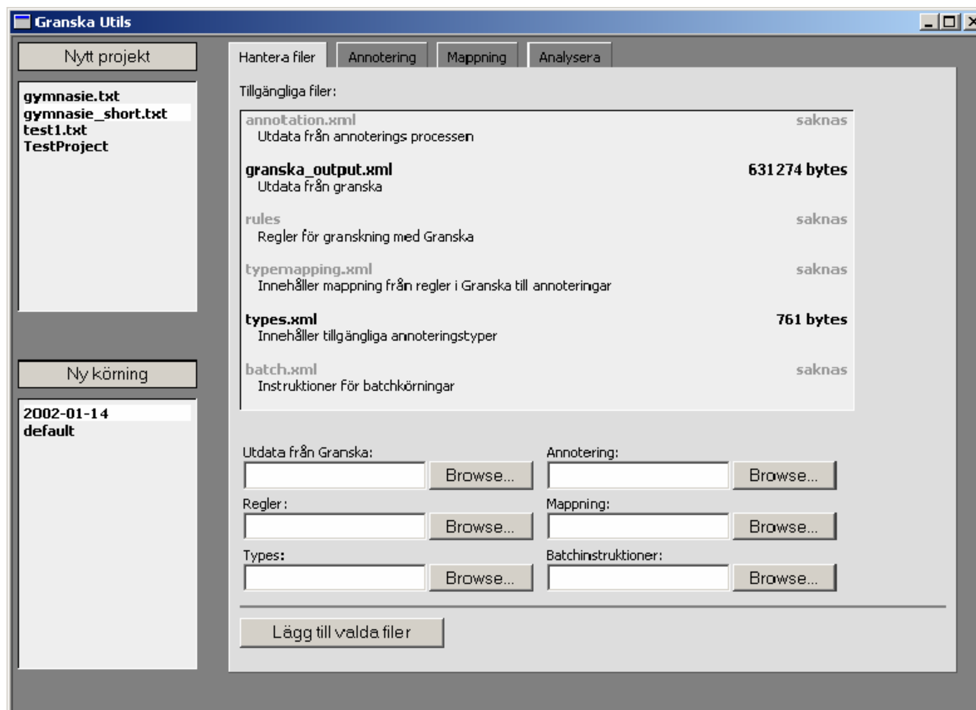
3. Vi skapar en ny regel, *zzz@kkk*, som placeras i ordning efter regeln *stav1@stavning*.

I nästa körning, som vi kallar 2002-03-12#2, gör vi följande:

1. Vi provar en ny formulering av regeln *stav1@stavning*.

2. Vi provar att ta bort regeln *xxx@yyy*.

3. Vi skapar en ny regel *xx2@kkk*, som placeras i ordning efter regeln *zzz@kkk* som lades till i den första körningen.



Figur 8 Verktyg för att generera verktyg. Till vänster syns de tillgängliga projekten och de körningar som finns tillgängliga för det aktiva projektet. Till höger syns "Hantera filer" där man kan välja de filer som behövs för de olika verktygen.

Administrationskonsol

När verktygen skall genereras så krävs att användaren har tillgång till all indata som krävs, bland annat krävs ofta utdata från Granska. All generering görs med hjälp av batchskript och startas således med en rad kommandoradsparametrar från kommandoraden. Dessutom måste indatafilerna placeras i speciella bibliotek för att genereringen skall fungera. För att underlätta denna hantering har jag implementerat ett grafiskt gränssnitt som hjälper till med att starta dessa genereringsskript.

Administrationskonsolen erbjuder användaren ett enkelt sätt att hantera texter och körningar. För varje körning kan användaren välja vilka filer som skall ingå i de olika verktygen. De filer som användaren väljer kopieras sedan till rätt bibliotek varvid genereringsskripten kan köras. När verktygen är genererade kan de också startas med hjälp av konsolen.

Administrationskonsolen består dels av en meny där användaren kan välja aktuellt projekt och körning, dels ett fönster där inställningar sker. I figur 8 kan vi se ett exempel på hur administrationskonsolen kan se ut. I menyn kan användaren även enkelt skapa både nya körningar och projekt.

I inställningsfönstret presenteras fem olika fönster under fem ”flikar”. Det första fönstret innehåller en filhanterare där användaren kan välja vilka filer som skall användas. Hela tiden ser användaren vilka filer som är valda och vilka som återstår att välja. De fyra övriga fönstren korresponderar mot varsitt verktyg. Först ett fönster för att hantera annoteringsverktyget. Här kan användaren välja vilka filer som skall användas, generera ett annoteringsverktyg och slutligen starta verktyget. Sedan finns ett fönster för att göra mappning av vilka annoteringar som skall detekteras av vilka regler. Här kan även användaren generera ett mappningsverktyg och sedan starta detta. Det tredje fönstret innehåller möjligheter att generera statistik och studera denna. Det fjärde och sista fönstret används för att kontrollera automatiserings-verktyget. Här kan användaren välja automatiseringsbeskrivning, sedan generera regeluppsättningar och slutligen analysera resultatet med hjälp av statistikverktyget.

Liksom de andra verktygen är administrationskonsolen implementerad med hjälp av HTML. För att kunna hantera lokala filer och starta nya processer, för att till exempel starta en XSLT-transformeringsprocess, så måste HTML-filerna ha speciella rättigheter att göra mer än vad webbsidor vanligtvis kan göra. Administrations-konsolen använder en teknik som endast fungerar i Microsoft Internet Explorer och heter HTML Application. Detta innebär att HTML-sidan får rättigheter att göra allt som ett vanligt program på datorn kan göra. Man kan exempelvis komma åt användarens lokala hårddisk och skapa och ändra i filer. Man kan dessutom starta nya processer och program.

Automatisk konstruktion av regler

Den bästa hjälp en regelkonstruktör kan få är förstås att datorn utför hela konstruktörens arbete. Detta skulle innebära att datorn skulle konstruera reglerna själv. Enligt en undersökning gjord av Ngai och Yarowsky (Ngai & Yarowsky, 2000) så tyder mycket på att det långsiktigt är både billigare och snabbare att låta en dator endast använda sig av stora korpus istället för att använda både manuellt konstruera regler och korpus. Ngai och Yarowsky studerade specifikt uppgiften att finna substantivfraser, men resonemanget och resultatet antyder att det även kan gälla för andra tillämpningar, såsom grammatikgranskning. Gojenol och Oronoz menar att med en tillräckligt stor korpus torde det gå att använda maskinlärning för att skapa mönsterigenkännande regler för att finna grammatiska fel (Gojenol & Oronoz, 2000).

Vi antar för den vidare diskussionen att datorprogrammet har tillång till stora mängder felannoterat textmaterial, på liknande sätt som diskuterats tidigare, för det språk som reglerna skall genereras. Den triviala metoden att söka bland alla tänkbara regeluppsättningar och utvärdera dessa med ett statistikverktyg såsom beskrivet tidigare är mycket opraktiskt med tanke på den näst intill oändliga mängden tänkbara regeluppsättningar. Om meningarna i språket kunde ha oändlig längd så skulle det förstås finnas just oändligt många tänkbara regeluppsättningar. Även om vi begränsar meningarnas längd till rimliga längder (säg mindre än 20 ord) så blir mängden tänkbara regeluppsättningar alltför stor för en uttömmande sökning.

När man automatiskt vill konstruera regler måste man bestämma sig för hur man vill prioritera täckning i förhållande till precision. Om man endast prioriterar precision så kan man låta regeluppsättningen bestå av regler som känner igen endast en mening var. Det finns en regel som känner igen varje känd felaktig mening, det vill säga i det här fallet, en regel för varje annotering som känner igen endast den meningen. Om istället täckning är det enda prioriterade så kan vi nöja oss med att skapa en regel som matchar alla tänkbara meningar.

Ett sätt att lösa avvägningen mellan precision och täckning är att kräva att den automatiska regelkonstruktionen konstruerar regler som har en precision som är högre än ett givet tröskelvärde. Utifrån detta bör den automatiska regel-konstruktionen söka maximera täckningen. Ju lägre tröskelvärde som väljs desto lättare för den automatiska konstruktionen att finna regler med hög täckning.

Transformationsbaserad inlärning

En vanlig metod för maskininlärning för tillämpningar inom naturliga språk är så kallad transformationsbaserad inlärning (Brill, 1995, samt Mangu & Brill, 1997). Antag att man till exempel vill lära en dator hur man känner igen ett substantiv i en text (exemplet är i stora drag hämtat från Brill, 1995). Transformationsbaserad inlärning kräver en så kallad basepredictor, som är någon typ av (naiv) algoritm för det problem man vill lära datorn. I vårt exempel skulle en basepredictor kunna vara en algoritm som säger att alla ord som finns i ett givet lexikon är ett substantiv.

För att inlärningen skall fungera krävs ett facit, så att man hela tiden kan se till att inlärningen går framåt, det vill säga att datorn hela tiden blir bättre och bättre. I exemplet med substantivigenkänning skulle ett facit kunna utgöras av en korpus annoterat med ordklasser för alla ingående ord.

Själva inlärningen fungerar sedan så att datorn väljer bland en lista med transformeringar, utvärderar vilken som gav den bästa förbättringen med avseende på facit. Den transformering som gav bäst resultat sparas. Man fortsätter sedan att leta efter nya transformeringar tills det inte längre går att finna transformeringar som ger ett förbättrat resultat. Den inlärda algoritmen utgörs sedan av att man först applicerar sin basepredictor och därefter sin lista med transformeringar i ordning.

För att den transformationsbaserade inlärningen skall fungera bra krävs ett lämpligt val av möjliga transformeringar. Dels kan man inte ha för stor mängd transformeringar, för då tar det för lång tid att testapplicera dem i varje iteration av inlärning. Dels måste transformeringarna vara tillräckliga för att representera all kunskap som krävs för den givna tillämpningen.

För att känna igen substantiv skulle en transformering kunna vara, *Ändra klassificering från **substantiv** till **ej substantiv** om **föregående** ord är klassificerat som **substantiv***. En lämplig mängd transformeringar skulle kunna vara bland annat *Ändra klassificering från **X** till **Y** om ordet Z positioner tidigare är av ordklass **R***. För att inte få för många möjliga transformeringar kan vi begränsa Z till högst 3.

Transformationsbaserad inlärning har bland annat använts för att tagga ord med ordklasser (Brill, 1995). Man har då som ovan använt en basepredictor som är ett lexikon, men eftersom många ord kan ha flera olika ordklasser, använder man transformeringar såsom ovan för att disambiguera. Metoden har också använts för *textchunking*, det vill säga man vill dela upp meningar i mindre sammanhängande enheter såsom exempelvis nominalfras (Ramshaw & Marcus, 1995).

Det borde gå att använda transformationsbaserad inlärning för att lära en dator att känna igen grammatiskt felaktiga meningar. Man skulle kunna ha en regelbaserad basepredictor, det vill säga man skulle kunna använda Granska. Transformationerna skulle kunna likna de tranformationer som exemplifierats för att känna igen

substantiv, men med möjlighet att undersöka fler attribut än ordklass, exempelvis plural/singular eller bestämd/obestämd form. Ett exempel på en transformation skulle kunna vara *Ändra från status **matchad** till **icke-matchad** om matchad av regel **X** och ordet **Y** positioner från början av matchningen har attributet **Z** satt till **R***.

Resultatet av en sådan inlärning blir dock inte en uppsättning regler, utan regler samt ett antal transformationer. Om tranformationerna formuleras i likhet med ovan så har de stora likheter med Granskas regler. Med små modifieringar av Granska skulle de kunna betraktas som en ny sorts regler som appliceras i ordning efter det att de vanliga reglerna applicerats.

Ett problem med att använda transformationer som ovan är att inlärningen kräver att en transformation förbättrar resultatet. Det innebär att mer komplicerade villkor, som förvisso kan uttryckas som en serie av transformationer, men där transformationerna separat kan innebära en temporär försämring, inte prövas. Detta kan lösas genom att tillåta mer komplicerade transformationer, men å andra sidan blir sökrymden betydligt större och inlärningen tar längre tid.

Låt oss studera ett exempel. Antag att vi har regler som istället för att söka efter felaktiga meningar söker efter korrekta meningar. Vi har en regel, *regel1*, som hittar alla meningar som börjar med *Vi sysslar med*. I vår korpus har vi meningarna

1. *Vi sysslar med en process för projektarbete.*
2. *Vi sysslar med dyr avancerad teknologi.*
3. *Vi sysslar med en ny teknologi inom läkemedelsindustrin.*
4. *Vi sysslar med avancerad process teknologi.*
5. *Vi sysslar med en process för att rena vatten.*

Alla dessa meningar matchas av vår regel. Låt oss nu studera två transformationer:

> *Ändra från status **matchad** till **icke-matchad** om matchad av regel **regel1** och ordet **2** positioner från slutet av matchningen har attributet **ord** satt till **process**.*
>
> *Ändra från status **matchad** till **icke-matchad** om matchad av regel **regel1** och ordet **3** positioner från början av matchningen har attributet **ord** satt till **teknologi**.*

tillsammans innebär en förbättring, men att den första transformationen inte prövas eftersom den allena innebär ett försämrat resultat. Den första transformationen ger upphov till ett korrekt val (meningen nummer 4) och två falska alarm (mening nummer 1 och 5). På samma sätt ger den andra transformationen ett försämrat resultat, även här ett korrekt val (mening nummer 4) och två falska alarm (mening nummer 2 och 3). Om vi istället kunde pröva den mer avancerade transformationen

*Ändra från status **matchad** till **icke-matchad** om matchad av regel **regel** och ordet **2** positioner från början av matchningen har attributet **ord** satt till **process** och ordet **3** positioner från början av matchningen har attributet **ord** satt till **teknologi**.*

så skulle vi få en förbättring eftersom vi nu endast hittar mening nummer 4. Å andra sidan har den nya transformationen betydligt fler frihetsgrader än de båda första.

Ett annat problem med transformationerna är att det är svårt att finna vilka transformationer som kan användas för att ändra en icke-matchad mening till en matchad mening. De transformationer som skulle kunna vara tillräckliga för detta skulle behöva lika många frihetsgrader som Granskas regler, vilket gör sökningen bland transformationerna omöjlig.

Alignment based training

Zaanen har använt sig av en metod som han kallar för alignment based training för att skapa en grammatik för ett språk (Zaanen, 2000). Alignment based training bygger på att man studerar en stor mängd meningar och försöker hitta par av meningar som kan paras ihop bra, det vill säga har ett antal likadana ord i samma ordning. Man studerar sedan dessa par av meningar och antar att de delar som är lika utgör en syntaktisk enhet i grammatiken. Svårigheten ligger i att hitta ett bra mått på likhet mellan meningar.

Jag har valt att undersöka en metod för automatisk regelkonstruktion som bygger på att finna likheter mellan de meningar som ger upphov till grammatiska fel, inspirerat av metoden för alignment based training. Dessutom kan man söka efter olikheter mellan de korrekta meningarna och de icke-korrekta meningarna.

Låt oss se hur alignment based training skulle kunna användas för att finna nya regler. Låt oss börja med att betrakta en mängd meningar som alla gör upphov till grammatiska fel. För varje par av meningar i denna mängd försöker vi finna likheter dem emellan. Om det är tillräckligt stora likheter så skapar vi en regel som känner igen denna likhet. Likheter kan vara saker som att de båda meningarna börjar med ett substantiv följt av ett verb i presens, eller att ordet *och* någonstans i meningen förekommer två gånger direkt efter varandra. Vi evaluerar därefter regeln på alla meningar som inte tillhör mängden, det vill säga alla meningar som inte innehåller annoterade fel. Om vi får få träffar bland de korrekta meningarna (och gärna många träffar bland de annoterade med felaktigheter) så behåller vi regeln. Vi fortsätter tills vi har tillräckligt många regler för att detektera alla (eller en stor andel) av meningarna i mängden av felaktiga meningar. Nedan visas algoritmen i pseudokod. S betecknar mängden felaktiga meningar och S* betecknar dess komplement, det vill säga alla korrekta meningar.

```
rules ← 0
while | S | > 0 do
        (x, y) ← a random pair in S
        R ← find best aligment rule(x, y);
        if (match(R, S*) is small) then
                rules ← rules ∪ R
                S ← S – match(R, S)
```

Funktionen *find best alignment rule* beräknar den bästa likheten av x och y givet någon likhetsfunktion. Likhetsfunktionen beräknar likhet genom att bland annat studera ordens särdrag, det vill säga två ord är lika om de har många gemensamma särdrag. Denna likhet som hittas kan lätt översättas till en regel som korresponderar med en Granskaregel.

En brist med algoritmen ovan är att man bara betraktar de största likheterna mellan meningarna. Det kan dock finnas likheter meningarna emellan som inte är de likheter som hittas när man letar efter de största likheterna. Betrakta exempelvis meningarna:

En lok är svart

En tak är svart

Den bästa likheten skulle kunna vara att båda meningarna passar in i mönstret *En xxx är svart* emedan de likheter som skulle ge upphov till en bra regel snarare är *En "substantiv i neutrum"*. Ett sätt att lösa detta är att testa flera "likheter" som genererats med hjälp av att använda ett antal olika likhetsfunktioner. Problemet är dock att det finns alltför många regler att söka bland. Låt oss därför införa två regeltrans-formeringar. Den ena transformeringen gör regler till mindre känsliga för att på så sätt hitta fler felaktiga meningar. Den andra transformeringen gör regler mer känsliga för att göra så att regeln detekterar färre meningar som inte är grammatiskt felaktiga.

Lösgöra restriktioner

Den första transformeringen skall appliceras på regler som matchar få meningar med fel och ger upphov till få falska alarm. Vi vill försöka få regeln att detektera fler meningar, men inte generera fler falska alarm. Ett sätt att göra detta på är att ta ytterligare en mening ur mängden med felaktiga meningar och hitta den minsta förändringen av regeln som gör att den nya meningen detekteras. Vi verifierar därefter att vi inte får oacceptabelt många felaktiga matchningar.

Ett annat sätt är att ändra regeln genom att ta bort krav. Vi kan antingen ta bort hela krav eller göra något av kraven mindre starkt. Om vi till exempel har kravet att det skall finnas ett ord som har ordklass verb och tempus futurum, så kan vi minska kravet genom att endast kräva ordklass verb. Algoritmen fungerar i vissa avseenden

som transformationsbaserad inlärning i det avseende att vi söker bland alla möjliga förändringar av krav för att hitta den förändring som ger det bästa resultatet. Vi fortsätter sedan att leta förändringar till dess inga förbättringar kan ske.

Införa restriktioner

I vissa lägen hittar vi en regel som matchar både många felaktiga meningar, men som samtidigt ger ett stort antal falska alarm. Vi vill här hitta ett sätt att förändra regeln så att vi undviker de falska alarmen. För att göra detta söker vi efter fler krav som vi kan lägga till regeln.

Ett sätt att göra detta är att välja ett falskt alarm och jämföra detta med någon av de meningar som regeln matchar korrekt. Vi studerar likheten mellan dessa meningar för att hitta något som skiljer dem åt. Det som skiljer dem åt översätter vi till ett krav som vi lägger till regeln. Vi utvärderar sedan vår potentiellt förbättrade regel genom att utvärdera den på alla meningar. Om vi uppnått en förbättring så behåller vi vår förbättrade regel, annars väljer vi ett nytt falskt alarm och försöker igen.

Ett annat sätt är att testa nya restriktioner utan att betrakta de falska alarmen. Vi testar här samtliga krav vi kan ställa på olika ord i meningen. För varje nytt krav utvärderar vi regeln gentemot alla våra meningar. Alla krav som innebär en förbättring behåller vi.

Implementation

Som vi ser kan man både införa och lösgöra restriktioner på två olika sätt. Den naiva metoden testar att ta bort alla tänkbara krav respektive att lägga till alla tänkbara krav. Den mer sofistikerade metoden försöker minska sökrymden genom att studera de meningar vi vill att regeln skall matcha respektive inte matcha. Fördelen med den senare metoden är förstås minskade sökningar och ökad prestanda men med risk att vissa restriktioner inte prövas. Eftersom båda metoderna bara inför eller tar bort ett krav per iteration så riskerar vi samma sak som med den transformationsbaserade inlärningen, nämligen att kombinationer av förändringar kan vara bra utan att de i kombinationen ingående förändringarna allena behöver innebära förbättringar.

Jag har valt att implementera införande och lösgörande av restriktioner med den naiva metoden, det vill säga utan att ta hänsyn till falska alarm och icke matchade meningar.

Nedan visas algoritmen i pseudokod. S betecknar mängden felaktiga meningar och S* betecknar dess komplement, det vill säga alla korrekta meningar.

```
rules ← 0
while | S | > 0 do
        (x, y) ← a random pair in S
        R ← find best aligment rule(x, y);
        if (match(R, S⋆) is somwhat small or match(R, S) is large) then
                R ← improveRule(R, S)
                if (match(R, S⋆) is small) then
                        rules ← rules ∪ R
                        S ← S – match(R, S)

function improveRule(R, S)
        while (true) do
                if (match(R, S⋆) is large and match(R, S) is large) then
                        R ← restrictRule(R, S)
                else if (match(R, S⋆) is large and match(R, S) is small) then
                        R ← loosenRule(R, S)
                else return R

function restrictRule(R, S)
        for each possible restriction A do
                Q ← R + A
                if (match(Q, S⋆) < match(R, S⋆) and
                        match(Q, S) ≈ match(R, S) then
                        R ← Q
        return R

function loosenRule(R, S)
        for each restriction A in R do
                Q ← R – A
                if (match(Q, S) > match(R, S) and
                        not match(Q, S⋆) > match(R, S⋆) then
                        R ← Q
        return R
```

Resultat och utvärdering

Jag har evaluerat algoritmen ovan på en felannoterad korpus med ungefär 300 meningar med fel i, och ungefär 1200 meningar utan fel. Eftersom det har varit svårt att få tag i stora felannoterade korpusar så har jag använt Granska för att skapa en korpus. Detta för med sig att innehållet inte är helt korrekt, utan innehåller både det som Granska gör korrekt och de fel som Granska gör. Eftersom algoritmen söker finna regler för Granska, så kan det faktum att de annoterade felen i korpusen redan känts igen av regler, innebära att utvärdering häri ger ett mer positivt resultat än vad som kan förväntas om jag använt en korrekt (manuellt) felannoterad korpus.

Systemet hittar under mina körningar regler som kan känna igen ungefär 90 meningar med fel i och cirka 30 meningar som inte är felaktiga. Endast ett fåtal av de regler som hittades har en hög kvalitet, det vill säga de flesta regler som hittas känner igen en specifik egenskap i en mening, som förvisso finns i flera meningar som är felaktiga, men där egenskapen som känns igen inte har något att göra med felet i meningen. Exempelvis visar det sig att den bästa regel som hittas är att konstatera att alla meningar som innehåller ordet *teknologi* är felaktiga. Detta beror på att det finns ett antal meningar där ordet *processteknologi* är felaktigt särskrivet till *process teknologi*. Eftersom ordet *teknologi* inte finns i någon annan mening så är det tillräckligt att säga att alla meningar som innehåller ordet *teknologi* är felaktiga.

Genom att mata korpusen med meningar med enkla medvetna fel i stil med *En svart lok står där borta* och *Det står ett bil där borta* så har jag verifierat att systemet verkligen fungerar. Med hundra konstruerade meningar med felaktig användning av *en* och *ett* samt ett stort antal korrekta meningar så lyckades systemet finna följande regler (frågetecken står för att vilket ord som helst kan matchas):

- dt.neu nn.utr
- dt.neu ? nn.utr
- dt.utr nn.neu
- dt.utr ? nn.neu

Systemet finner alltså alla regler där en determinerare följs av substantiv och genus inte stämmer överens. En del av de felaktiga meningarna hade också ett adjektiv mellan determinerare och substantiv. Vi ser ovan att systemet även fann dessa meningar. Endast en felaktig mening, *En litet rött hus*, hade två adjektiv mellan determinerare och substantiv varför någon regel för detta fall inte genererades.

En manuellt konstruerad regel skulle troligtvis skrivas (Knutsson, 2001):

```
X(wordcl=dt),
Y(wordcl=jj*),
Z(wordcl=nn & gender!=X.gender)
```

Regeln ovan säger att en mening är felaktig om en determinerare följs av noll eller flera adjektiv och därefter ett substantiv, där substantivet och determinerarens genus inte stämmer överens. Till skillnad från den automatiskt konstruerade regeln kan vi uttrycka att determinerare och substantiv kan skiljas åt med godtyckligt antal adjektiv. De automatiskt konstruerade reglerna har en regel för varje fall, det vill säga en regel för determinerare i utrum och substantiv i neutrum och en regel för determinerare i neutrum och substantiv i utrum emedan den manuellt konstruerade regeln helt enkelt uttrycker att gender på substantivet och determineraren skall vara skilda. Troligtvis kan frihetsgraderna hos de automatiskt konstruerade reglerna ökas så att reglerna kan uttrycka mer och bli mer kompletta, men det är sannolikt svårt att automatiskt konstruera regler som är lika kompakta och uttrycksfulla som de manuellt konstruerade reglerna.

Algoritmen för att konstruera regler söker endast efter regler som känner igen mer är en felaktig mening. Eftersom korpusen som använts är så pass liten innebär detta att en hel del potentiella regler inte testas eftersom det endast finns en mening med det felet representerat i korpusen. Ett sätt att förbättra resultatet är rimligtvis att använda en större korpus. Om varje feltyp finns representerad med ett flertal meningar så ökar sannolikheten att det skapas en regel för det felet. Ett annat sätt att förbättra systemet är att använda mer specifika korpus. Man skulle till exempel kunna använda en korpus där det endast finns grammatiskt korrekta meningar och meningar som innehåller ett specifikt grammatiskt fel. De regler som då skapas är garanterade att endast söka efter det specifika felet.

En annan faktor som skulle kunna förbättra resultatet är att titta på hur likhet mellan meningar söks. I min implementation söks efter par av meningar som har något gemensamt. Ett sätt att få regler med högre täckning skulle kunna vara att istället för att söka efter par av meningar, söka efter grupper om fem eller tio meningar som har någonting gemensamt. Vi skulle då öka sannolikheten att de regler som hittas verkligen känner igen någonting signifikant hos det fel som man försöker känna igen. Man kan också studera vad som skall betraktas när man söker efter likhet. Skall det anses vara en bra likhet om två meningar innehåller samma ord, eller är det bättre att söka efter mönster av andra attribut såsom ordklass och tempus. Ska meningen *En lok är svart* och meningen *En tak är svart* anses vara lika varandra efter-som att de båda slutar med *är svart* eller för att de båda börjar med ett kongruensfel, det vill säga *en lok* och *en tak* istället för *ett lok* och *ett tak*.

Slutsats

I och med att Granska har fått möjlighet att exportera sin kunskap till XML-filer så öppnas en mängd nya möjligheter. Det finns många tillämpningar som kan tänkas använda Granskas kunskap för egen nytta men även för att kunna förbättra Granska på olika sätt. I detta examensarbete har vi sett hur man kan konstruera verktyg för att underlätta regelkonstruktörens arbete. Dessa verktyg har alla nyttjat Granskas kunskap om dels språkets struktur, dels Granskas möjligheter att finna felaktigheter och andra mönster i meningar.

Det huvudsakliga problemet som exjobbet behandlat har lösts genom att annoteringsverktyget och statistikverktyget har implementerats. Det första verktyget, annoteringsverktyget, har under en tid använts för att börja bygga en svensk fel-korpus. Ju större korpus som finns tillgängliga desto lättare blir det att förbättra och forska kring Granska.

Det andra verktyget, rapport- och statistikverktyget, löser problemet att förstå hur regler presterar. Systement har inte börjat användas i samma utsträckning som annoteringsverktyget, men rapportverktyget borde dock kunna användas för att underlätta arbetet med konstruktion av regler. Det torde vara lättare att förbättra en regel om man på ett lättillgängligt sätt kan se vilka meningar som regeln finner och vilka regler som missas.

Att automatiskt konstruera regler är en intressant idé, men de resultat som presenterats i detta exjobb har mycket kvar att önska. Det behövs troligtvis mycket forskning och en mycket stor korpus för att kunna skapa regeluppsättningar med samma täckning och precision som manuellt konstruerade regler. Det är med de metoder som presenterats här svårt att skapa regler med lika stor täckning som manuellt konstruerade regler. Detta kan förstås kompenseras med hjälp av fler regler.

Det finns också andra problem med de automatiskt konstruerade reglerna som återstår att lösa. Exempelvis är det svårt att se hur man skulle kunna konstruera regler som ger rättningsförslag till de fel som detekteras. Om man vill kontrollera texter utan manuellt konstruerade regler är det inte säkert att den bästa metoden är att automatiskt konstruera regler. Andra metoder än regler för att finna grammatiska felaktigheter kan vara enklare att lära upp med maskininlärning.

Referenser

E. Brill. 1995. Transformation-based error-driven learning and natural language processing: a case study in part of speech tagging. *Computational Linguistics*, december 1995, s. 543-565.

J. Carlberger, R. Domeij, V. Kann, O. Knutsson**.** 2000. A Swedish grammar checker. Tillgänglig via http://www.nada.kth.se/theory/projects/granska/rapporter, nerladdad 2002-05-14.

J. Carlberger, V. Kann. 1999. Implementing an efficient part-of-speech tagger. *Software- practice and experience*, nr 9 1999.

J. Carlberger, V. Kann, 2000. Some applications of a statistical tagger for Swedish. I *Proc. 4th conference of the international quantitative linguistics association.*

R. Domeij, O. Knutsson, J. Carlberger, V. Kann**.** 1999. Granska - an efficient hybrid system for Swedish grammar checking. I *Proc. 12th nordic conference in computational linguistics*, december 1999, s. 49-56.

K. Gojenol., M. Oronoz. 2000. Corpus-based syntactic error detection using syntactic patterns. I *Proceedings of the student research workshop at applied natural language processing.*

V. Kann, R. Domeij, J. Hollman, M. Tillenius**.** 1998. Implementation aspects and applications of a spelling correction algorithm. NADA report, TRITA-NA-9813.

O. Knutsson. 2001. Automatisk språkgranskning av svensk text. Licenciatavhandling, TRITA-NA-01-5, ISBN 91-7283-052-2, Nada, KTH.

L. Mangu, E. Brill. 1997. Automatic rule acquisition for spelling correction. I *Proc. 14th International conference on machine learning*, s. 187-194.

G. Ngai, D. Yarowsky. 2000. Rule writing or annotation: cost-efficient resource usage for base noun phrase chunking. I *Proc. 38th ACL conference*, s. 117-125.

L. A. Ramshaw, M. P. Marcus. 1995. Text chunking using transformation-based learning. I *Proceedings of the third workshop on very large corpora*, s. 82-94.

M. van Zaanen. 2000. Bootstrapping syntax or recursion using alignment-based learning. I *Proc. 17th international conference on machine learning*, s. 1063-1070.