

TDT4113 - Datateknologi, programmeringsprosjekt

Oppgave 4: Tekstanalyse

Dette dokumentet beskriver den fjerde oppgaven i ProgLab 2. For denne oppgaven gjelder at:

- Oppgaven skal løses individuelt
- Oppgaven skal løses med objektorientert kode skrevet i Python
- Fristen for oppgaven er *2 uker*, dvs. implementasjonen din lastes opp på BLACKBOARD og demonstreres senest 20. oktober, 2017, kl 15:00.

NB! Les hele oppgaveteksten før du begynner på implementasjonen. Funksjonaliteten i systemet beskrives skrittvis, men det kan være lurt å se “hele pakka” før du velger designet. Husk at du må bruke et OO-design for å få prosjektet godkjent. Det er en del tips sprett her og der; disse er også nyttige å få med seg før man begynner å kode!

1 Bakgrunn for oppgaven

I dette prosjektet skal vi analysere anmeldelser av filmer som er publisert på `imdb.com`; se <http://www.imdb.com/title/tt0133093/usercomments> for eksempler på hvordan anmeldelsene av en av tidenes beste filmer ser ut. Anmeldelsene vi skal bruke er allerede hentet ned fra `imdb.com`, og er gjort tilgjengelige i en `.zip`-fil på BLACKBOARD; katalog-strukturen blir beskrevet i Seksjon 2. Oppgaven går ut på å analysere teksten i en anmeldelse, for så å gjette på om den er **positiv** eller **negativ**.¹ Slik automatisert analyse av en tekst for å bestemme om den er **positiv** eller **negativ** er kjent under navnet *sentiment-analysis*, og brukes (om enn i litt mer avanserte former enn vi skal jobbe med her) for eksempel til å analysere

¹positive anmeldelser har mer enn seks stjerner, **negative** har mindre enn fem stjerner. Anmeldelsene med fem og seks stjerner er tatt ut, da de er “midt på treet” i denne sammenheng.

børsmeldinger i nær sann tid, for så å kunne trade på en aksje før andre i markedet har rukket å reagere på innholdet.

I prosjektet vil dere få litt innblikk i hvordan man kan lage et dataprogram som selv kan “lære” fra data, bruke enkle teknikker for analyse tekst, og få trening i bruk av Pythons dictionaries/lister/set/tekst-strenger.

2 Om datafilene dere skal bruke

Når man programmerer systemer som skal “lære” noe fra data er det vanlig å dele dataene opp i to deler: Den første delen, *trenings-dataene*, brukes av systemet for å se etter relevante og viktige sammenhenger som kan brukes til å løse oppgaven systemet er satt til (i vår setting: å finne ut hvilke ord som kjennetegner hhv. **positive** og **negative** anmeldelser). Systemet får vite om det er en **positiv** eller **negativ** anmeldelse når det analyserer en tekst fra trenings-dataene, og bruker denne informasjonen til å utlede sammenhengene. Den andre delen av datasettet, *test-dataene*, brukes til å teste hvor godt systemet har blitt etter at det er ferdig med å lære fra trenings-dataene. Når vi bruker test-dataene ber vi systemet gjette på om anmeldelsene der er **positive** eller **negative** – og da selvsagt uten først å fortelle hva som er riktig svar.

Alle datafilene ligger i en egen zip-fil på BLACKBOARD. Strukturen er som følger:

- **data:** Rot-katalogen
 - **stop-words.txt:** En liste med såkalte “stopp-ord”; ett ord per linje. Bruken forklares i neste del av dokumentet.
 - **alle:** Katalogen inneholder alle anmeldelsene vi skal analysere.
 - * **train:** Katalogen inneholder trenings-dataene (som altså skal brukes for å lære opp systemet).
 - **neg:** Katalogen inneholder alle **negative** anmeldelser. Filene har navn etter formatet <løpenummer>_<rating på skala fra 1 til 10>.txt
 - **pos:** Alle **positive** anmeldelser. Filene er navngitt som over.
 - * **test:** Katalogen inneholder test-dataene, som skal brukes for å teste systemet etter at det har analysert treningssettet.
 - **neg**
 - **pos**
 - **subset:** Katalogen inneholder et subset av anmeldelsene. Mens du utvikler koden din kan det være lurt å bruke dette datasettet, da det er

mye kjappere å analysere. Bruk dataene i `alle`-katalogen først når du er ferdig med en del av implementasjonen og vil kjøre programmet ditt for å få “skikkelige” svar. Underkatalogene er som for `alle`-katalogen:

```
* train
  · neg
  · pos
* test
  · neg
  · pos
```

3 Implementasjon for å bli kjent med dataene

Det første vi må gjøre er å lese anmeldelsene fra fil og representere dem på en måte som er egnet for analysen vi skal gjøre senere. For enkelhets skyld gjør vi noen antagelser her, slik at representasjonen skal bli så enkel som mulig:

- Vi er foreløpig kun interessert i *hvilke ord* som er brukt i anmeldelsene, ikke hvilken rekkefølge ordene er skrevet.
- Vi er ikke interessert i hvor mange ganger et ord er brukt innen en anmeldelse, kun om ordet er til stede eller ei.
- Vi skiller ikke mellom stor og liten bokstav, og ser bort fra alle tegn som ikke er alfanumeriske (dvs. at vi fjerner “.”, “,”, “(”, “)” etc.)

Eksempel: Anta teksten i en anmeldelse er slik: “Dette er en bra (BRA) film. Filmen er bra, synes jeg. (Veldig bra!) 9/10!!”. Resultatet skal være en liste med alle ordene representert en gang: [dette, er, en, bra, film, filmen, synes, jeg, veldig, 9, 10]. Merk at ordene som er brukt flere ganger bare er med en gang i resultatet (se for eksempel “er”). Merk også at vi ikke skiller mellom “bra”, “BRA”, “bra,” og “bra!”, men at “film” og “filmen” er ansett som forskjellige.

Implementasjon – Del 1:

Lag funksjonalitet for å lese et dokument i trenings-settet fra fil, og representere det som angitt over.

Tips:

Noen ganger kan `open(filename)` returnerer med ASCII-type feil, og for eksempel si at det er problemer med enkodingen. Prøv i så fall `open(filename, encoding = 'utf-8')` eller `codecs.open(filename, 'r', 'utf-8')` etter først å ha gjort `import codecs`.

Vi kan bruke representasjonen til å finne ut hvilke ord som er mest “populære” i hhv. **positive** og **negative** anmeldelser. La oss bruke sannsynligheten for å se ett gitt ord i en **positiv** anmeldelse som dette ordets *popularitet* blant de **positive** anmeldelsene. Denne beregner vi ved brøken

$$\frac{\text{Antall positive anmeldelser i trenings-dataene med } \langle \text{ord} \rangle}{\text{Antall positive anmeldelser totalt i trenings-dataene}}. \quad (1)$$

Populariteten for **negative** anmeldelser defineres tilsvarende.

Implementasjon – Del 2:

Utvid koden til å lese alle filene i trenings-settet. Analyser representasjonen så du finner de 25 mest populære ordene for hhv. **positive** og **negative** anmeldelser. Er listene som forventet?

Tips:

Om du bruker Mac kan det hende at operativ-systemet lager `.ds_store` - filer i data-katalogen og underkataloger. Når du bruker `os.listdir(path)` vil `.ds_store` filene bli med i listen av filer som programmet ditt tror skal lastes inn. Det betyr at systemet ditt vil forsøke å lese dem som tekstfiler, og det kan gi feil. En mulighet er å slette alle disse filene fra terminal-vinduet, alternativt bruke `glob.glob(path + '/*.txt')` (etter at du har gjort `import glob`) på samme måte som du tidligere brukte `os.listdir(path)`.

Listene som skrives ut i Del 2 av implementasjonen er stort sett såkalte *stopp-ord*, dvs. ord som ikke har spesiell relevans for oss når vi ønsker å skille **positive** anmeldelser fra **negative**. En liste av stopp-ord finnes i `./data/stop-words.txt` i `.zip`-fila på BLACKBOARD.

Implementasjon – Del 3:

Utvid systemet ditt til å fjerne alle stopp-ordene (som altså er listet i `./data/stop-words.txt`) fra representasjonen av dokumentene. Finn de mest populære **positive/negative** ordene etter at stopp-ord er fjernet, og se om det ble bedre.

Tips:

Da filene ble hentet ned og vasket for HTML-tags var det (tydeligvis) et problem med å fjerne alle `
`-tag'ene fra HTML-koden. Du kan legge til “br” i stopp-ord-listen for å bli kvitt dette fra ordene du analyserer.

Mange av de samme ordene går igjen i begge listene, og de er derfor ikke så veldig informative for oss. La oss derfor heller finne de ordene som har mest *informasjonsverdi* med hensyn på hvilken type dokument vi ser på. Vi definerer informasjonsverdien til et ord for **positive** anmeldelser som sannsynligheten for at et tilfeldig dokument med dette ordet skal være **positiv**, og beregner den slik:

$$\frac{\text{Antall positive anmeldelser i trenings-dataene med } \langle \text{ord} \rangle}{\text{Antall anmeldelser } \underline{\text{totalt}} \text{ i trenings-dataene med } \langle \text{ord} \rangle} \quad (2)$$

Definisjonen mhp. **negative** dokumenter er tilsvarende.

Implementasjon – Del 4:

Utvid systemet til å finne informasjonsverdien av ord. Skriv ut de 25 mest informative ordene for **positive** og **negative** anmeldelser.

Disse listene er veldig spesifikke, og forteller oss ikke så mye om hvordan språket benyttes generelt, for eksempel sier det oss ikke mye at et ord som “**brommel**” indikerer at anmeldelsen er **positiv** – det er en typo, og skal relateres til regisøren Henry Brommell. Et alternativ for å rydde opp i slike problemer er å “prune” dataene, med å fjerne alle ord som ikke er brukt minst i en gitt prosent-andel (for eksempel 1%, 2% eller 5%) av alle dokumentene.

Implementasjon – Del 5:

Implementer pruning, og generer listene på nytt. Er de nå slik at vi kan se hvilke ord som brukes for å uttrykke **positive** og **negative** tanker om en film?

Tips:

Pass på at det er andelen av dokumenter *totalt* som definerer om et ord skal prunes bort eller ikke. Dersom du går i fella og bruker kun andelen av **positive** dokumenter (eller **negative** dokumenter, for den del) blir det kaos når vi kommer til klassifikasjonssystemet i Kapittel 4. På samme måte vil det også gi problemer om du lager en egen pruning for hver av disse gruppene, så *ikke* gjør det...

Husk at vi ser på et tekst-dokument (en anmeldelse) kun som et sett med ord, der vi foreløpig ikke har brydd oss om rekkefølgen ordene brukes. Det gjør analysen enklere, men gjør samtidig at vi mister en del av det egentlige innholdet. For eksempel vil de to anmeldelsene “the movie is very good and not boring” og “the movie is not good and very boring” ha samme representasjon “[and, boring, good, is, not, the, very, movie]” (evt. får vi

“`[boring, good, movie]`” om vi fjerner stopp-ord). For å få mer informasjon er det vanlig å bygge såkalte n -grams, der $n = 2$ eller $n = 3$ er mest vanlig. 2-grams (også kalt bigrams) bygger vi her ved at hvert par av ord som følger etter hverandre i den originale teksten settes sammen med en “_” mellom seg og dermed danner et nytt ord. For eksempel vil “`this does not work`” gi 2-grams “`[this_does, does_not, not_work]`”, og en representasjon av setningen der 2-grams er inkludert blir “`[this, does, not, work, this_does, does_not, not_work]`” – før stopp-ord og lignende fjernes. Om vi lager 3-grams vil vi få bidraget “`[this_does_not, does_not_work]`”.

Implementasjon – Del 6:

Implementer n -gram og utvid representasjonen av dokumentene med 2-grams og 3-grams. Husk pruning; for best effekt bør du gjøre operasjonene i denne rekkefølgen:

1. Del opp setningene i ord, fjern ulovlige tegn, sett alt i lowercase.
2. Lag n -grams representasjonen.
3. Fjern stopp-ord og duplikater.
4. Prune bort ord som ikke er brukt i minst 1% av dokumentene i trenings-dataene.

Gir dette bedre lister for 25 mest informative **positive** og **negative** “ord”?

4 Klassifikasjonssystemet

Det å ta et objekt (i dette tilfellet et tekstdokument) og definere det som tilhørende en av et bestemt sett av mulige grupper (her: **positiv** eller **negativ**) kalles *klassifikasjon*, og vi er nå klare til å lage et system som skal gjøre klassifikasjon for oss. Når vi skal klassifisere et dokument regner vi et *godhetsmål* for dokumentet mhp. hver av gruppene vi ser på (en gang for **positiv** og en gang for **negativ** i vårt tilfelle) og tilordner dokumentet til en av gruppene basert på dette målet.

Godhetsmålet av dokumentet med hensyn på en gruppe beregnes ved at vi ser på alle ordene i dokumentet som er i vokabularet vårt (dvs. de ordene som ikke har blitt prune’et bort). Hver slikt ord bidrar med sin popularitet (som vi beregnet i Ligning 1), og bidragene fra alle ordene i dokumentet ganges sammen. Vi gjør dette først for den **positive** gruppen, deretter for den **negative**, så tilegnes dokumentet til den gruppen der godhetsmålet er størst. For disse beregningene kan det bli et problem at tallene som ganges sammen er veldig små, og dette løser

man typisk ved å heller regne ut *logaritmen* til godhetsmålet for hver gruppe. Husk at du i så fall skal *addere* sammen logaritmen til bidragene fra enkelt-ordene i stedet for å gange dem sammen, fordi $\log(x \cdot y) = \log(x) + \log(y)$. Poenget er altså at vi i stedet for å beregne produktet av noen små tall, for eksempel $0.00001 \cdot 0.002 = 0.00000002$ regner summen av logaritmene, i dette tilfellet $\log(0.00001) + \log(0.002) = -11.5129 + (-6.2146) = -17.7275$ (her kan du jo kontrollere at $\log(0.00000002) = -17.7275$ om du gidder). Summen av logaritmene er enklere å representere nøyaktig i datamaskinen, og ettersom vi bare skal finne ut for hvilken gruppe, **positiv** eller **negativ**, godhetsmålet er *størst* er dette tilstrekkelig.

Implementasjon – Del 7:

Implementer klassifikasjons-systemet. Systemet må først læres opp ved å lese alle trenings-dataene for så å beregne ordenes popularitet. Deretter må det ha funksjonalitet for å ta representasjonen av et nytt dokument inn, og gi antatt gruppe for dokumentet (**positiv** eller **negativ**) tilbake. I denne delen av implementasjonen skal du kun inkludere ord som er med i minst 2% av dokumentene, og du skal fjerne stopp-ord. Det er ikke nødvendig å ta med n -grams eller gjøre andre smarte triks.

Tips:

Det er viktig å tenke på hva som er “straffen” for en gruppe som ikke har et ord i listen av beregnede populariteter. La oss si at ordet “**terrible**” ikke finnes i listen for **positive** ord, men har popularitet i 0.90 for den **negative** gruppen. Vi må selvsagt straffe den **positive** gruppen på ett eller annet vis, så vi kan ikke bare overse ordet; om vi overser det er dette det samme som å late som om ordet har popularitet 1.0 i den **positive** gruppen (ettersom $\log(1.0) = 0.0$). Samtidig kan vi ikke bruke verdien 0 fordi $\log(0) = -\infty$, og dersom begge gruppene blir straffet med hvert sitt “ukjente” ord vil dette dominere alle de andre beregningene. En mulig løsning er å straffe den gruppen som ikke har ordet med verdien $\log(\epsilon)$ for et lite tall $\epsilon \leq 0.02$. Grensen 0.02 er valgt fordi vi har cut-off på at 2% av dokumentene skal ha med et ord for at det skal bli med i vokabularet. Her er det lov å prøve seg fram og se hva som virker best (se neste del), men start med $\epsilon = 0.01$.

Systemet er nå klart for å evalueres med test-dataene. Vi definerer systemets nøyaktighet slik:

$$\frac{\text{Antall test-dokumenter systemet tilordner riktig gruppe}}{\text{Antall test-dokumenter systemet får prøve seg på}}$$

Implementasjon – Del 8:

Evaluer systemet på test-dataene. For hvert dokument – både **positive** og **negative** – må du da gjøre disse operasjonene:

- Lese teksten fra fil.
- Transformere teksten til den representasjonen du bruker (del opp setningen i ord, fjern ulovlige tegn, fjern stopp-ord, sett i lowercase, fjern duplikater, fjern ord som ikke er i vokabularet fra trenings-dataene).
- Beregne godhetsmålene for dette dokumentet, og tilordne det til en av gruppene.
- Sjekke om systemet valgte riktig.

Tips:

Dersom du får rare resultater kan det være en god idé å sjekke litt nærmere hva som faktisk skjer. Følgende skritt er naturlige å gjennomføre:

- For hvert dokument der systemet ditt tar feil, skriv ut differansen mellom godhetsmålene for de to gruppene. Velg deg ut et dokument der absolutt-verdien av differansen er “stor”. Dette er et dokument der systemet var rimelig sikker på klassifikasjonen sin, men tok feil.
- Se spesielt på hvordan denne filen blir prosessert:
 - Er teksten tvetydig, eller er det “rart” at systemet feiler?
 - Ser “oversettelsen” fra klar-tekst til listen av ord riktig ut?
 - Skriv ut populariteten til hvert ord i representasjonen med hensyn til både **positive** og **negative** dokumenter. Ser disse “riktige” ut i den forstand at ord du synes er positivt ladet har større popularitet blant **positive** dokumenter og motsatt for ord du synes har negativ ladning? Er de ordene som har mest ekstremt utslag også de som du synes er sterkest ladet? Kan unaturlige resultater forklares fra trenings-dataene?
 - Virker den totale klassifikasjonen meningsfylt, gitt “ladingen” av hvert enkelt ord?

Om alt er gjort riktig bør du få en nøyaktighet på over 70%.²

²Husk at om man bare gjetter, for eksempel at alle dokumentene er **positive**, vil det gi 50% nøyaktighet, så tall på det nivået er ikke noe å skryte av!

5 Frivillig ekstra-oppgave:

I den første delen av prosjektet (Seksjon 3) så vi på hvordan noen ord var mer informative enn andre, og hvordan noen ord som kun var brukt i veldig få dokumenter kunne se veldig betydningsfulle ut – uten at de egentlig er det. Dette gir ideer for optimalisering av systemet:

Implementasjon – Del 9:

Optimaliser systemet ditt ved å skru litt på noen av disse valgene:

- Fjern ord som ikke er benyttet i mange nok trenings-dokumenter; finn ut av hvor mange dokumenter ordene bør være med i for ikke å kastes ut.
- Fjern ord som ikke har sterk nok informasjonsverdi (se Ligning (2)).
- Prøv å ta med n -grams av orden $n = 2$ og/eller $n = 3$.
- Det er også fritt fram å prøve ut egne ideer!

Etter optimalisering bør systemet ditt ha en nøyaktighet på godt over 80%.

6 Hva kreves for å bestå oppgaven

For å bestå denne oppgaven må du:

- Løse alle de obligatoriske del-oppgavene – merk at Del 9 er frivillig.
- Du skal gjøre arbeidet alene, og få det godkjent innen fristen.
- Systemet skal implementeres med objekt-orientert Python.