# Rapport – Obligatorisk innlevering 2a

adamdos, joannaek, justynao

## Oppgave 1

**a)** Transisjonssansynlighetene:
Sannsynligheten P(t|TAG) beregner vi ved å dele antall ganger t forekommer etter TAG (a) på antall ganger TAG forekommer (b).

P(ADP|DET) = a/b = 0/2 = 0
P(NOUN|DET) = ½
P(DET|DET) = ½
P(PUNCT|DET) = 0/2 = 0

Emissjonssannsynlighetene:
Sannsynligheten P(ord|TAG) beregner vi ved å dele antall ganger ordet forekommer tagget som TAG (a) på antall ganger TAG forekommer (b).
P(tanke|ADP) = a/b = 0/2 = 0
P(tanke|NOUN) = ½
P(de|DET) = ½

**b)** Viterbi er en dekoding algoritme og er et eksempel på dynamisk programmering. Viterbi dekoderer sekvens av skjulte tilstander, som med høyeste sannsynlighet kunne gi en sekvens av observasjoner. Hver celle av trellis, representerer sannsynligheten at HMM er i en tilstand *j* etter å ha sett den første t observasjoner og passerer gjennom de mest sannsynlige tilstand sekvens $q_1,\ldots,q_{t-1}$ gitt HMM λ. Kompleksitet er $O(N^2 T)$.

## Oppgave 2

I denne oppgaven fullførte vi read_dataset() metode.
Denne metoden leser inn filen og bruker split() metoden for å splitte teksten i setninger. Etterpå laget vi to lister sentences[] og labels[]. For å legge til korrekte elementer fra filen i listene har vi brukt en for løkke som går gjennom setningene, splitter dem inn i ord og tagger og legger dem adskilt «tilbake i en setning» og deretter i de riktige listene (labels eller sentences).

## Oppgave 3

**a)** Her har vi skrevet ferdig funksjonen bigrams().
Vi begynte ved å lage en liste bgs[] som skal ta vare på sekvensen av bigramer.
Vi legger til "<s>" på begynnelsen av sekvensen.
Så brukte vi en while løkke som setter sammen to elementer fra listen (index = i og index = i+1) og lagrer de i bgs[] listen.

**b)** I denne oppgaven skreiv vi ferdig funksjonen fit() som utfyller emissions- og transisjonstabellene.
Funksjonen tellte først tag – ord kombinasjoner. Den gikk gjennom setningene i en while løkke som terminerte når variablene i og j nådde siste indexene i de gitte listene.
For hver kombinasjon (forekomst av ordet tagget som gitt tag), så økte variabelen i emission tabellen med 1.
I andre delen av metoden så telte programmet bigram forekomster i en nøstet for-løkke. For hver setning i labels kalte vi på metoden bigrams(setning) og for hver element i den returnerte listen økte vi verdi til tag – tag kombinasjonen i transisions tabellen med 1.
Linje 133, 134 i koden oppretter en instansvariabel som holder styr på taggene modellen består av, som trengs senere i programmet.

**c)** I denne delen av oppgaven endret vi tallene til sannsynlighetene.
Vi brukte ConditionalProbDist metode på nøstet liste som hold styr på kombinasjon antall. Deretter på transisjons tabell.

## Oppgave 4

Vi har implementert viterbi algoritmen i en egen metode for enkelthet skyld. Det gjør det mer oversiktlig og enklere å kjøre algoritmen hvis vi kaller transform() på flere setninger om gangen.
Transform() kjører viterbi algoritmen på hver enkelt setning i listen den får som argument og returnerer resultatet (liste av predikerte variabler for hver setning).
Viterbi() tar som argument et setning. Så brukte vi ordbok for å kunne bruke tag navn og ord.
Vi lagger liste av ordbøker hvor hver ordbok svarer til steget i beregning, altså den første ordbok holder styr på sannsynlighetene for det første tagget gitt det første ordet i setningen.

Det lagres i en ny ordbok, hvor sannsynligheten og den forrige tilstand lagres.
Så kjører vi en nøstet for-løkke som beregner sannsynlighetene for alle mulige kombinasjoner og lagrer de høyeste sannsynlighetene.
Til slutt er løsningen vår den høyeste verdi av «prob» (altså sannsynlighet) i den siste ordboken i viterbi listen.
Når vi har funnet den verdien, finner vi den siste taggen i sekvensen som ga den største sannsynlighet. Så går vi «bakover» (ved bruk av prev variabelen) og finner hele sekvensen.

## Oppgave 5

**a)** Accuracy – Her har vi skrevet ferdig accuracy() metoden som beregner accuracy score. Vi har oppnådd dette ved å bruke en dobbel for loop som sammenligner elementer fra gullstandard listen med elementer fra listen med predikerte pos sekvenser. Vi tellte antall elementer i for-løppen, samt antall riktige gjett og returnerte accuracy som er lik antall riktige delt på totalt antall tagger.

**b)** Smoothing – Her fullførte vi funksjonen prob() i SmoothProbDist.
Vi brukte en if test som sjekker om verdien er 0 og returner 1e-20 hvis det er tilfelle, ellers returnerer funksjonen value.
Vi skreiv super(SmoothProbDIst, self), fordi python vi brukte på universitets sine maskiner er python 2.7 og tom super() ville ikke fungere.
Accuracy with MLEProbDist: 0.545794495312
Accuracy with SmoothProbDist: 0.911545547032
Vi ser her at smoothing er en viktig verktøy når det gjelder sannsynlighetene.
Verdiene er ofte 0. Siden sannsynlighetene baserer seg på å multiplisere så er jobbing med 0 en ganske vanskelig oppgave. Ved å bruke smoothing får vi mye høyere verdiene.