IN2110 våren 2020 – obligatorisk innlevering 1b

*Leveres innen **tirsdag 24. mars kl. 23.59** i Devilry.*

Det er en god idé å lese gjennom hele oppgavesettet før du setter i gang. Har du spørsmål så gå på gruppetime, spør på Piazza eller send epost til `in2110-hjelp@ifi.uio.no`.

Denne obligatoriske oppgaven er ganske omfattende, så planlegg god tid slik at du klarer å fullføre de to delene av oppgaven.

Oppsett

For IN2110 har vi laget et utviklingsmiljø for Python som inneholder programvare og data for obligene. For mer informasjon om bruk av miljøet på IFI's Linux maskiner eller via SSH, eller for installasjon på egen maskin, se;
`https://github.uio.no/IN2110/in2110-shared`

I oppgavene under skal dere bygge videre på pre-kode som er tilgjengelig under mappen `obliger/1b` i kurs-repoet for gruppetimer og obliger:
`https://github.uio.no/IN2110/in2110-lab/`

Innleveringsformat

Innleveringen skal bestå av **to** Python-filer (en for delen om logistisk regresjon og en for delen om sekvensmodeller) med kode og en rapport i PDF-format. I rapporten skal dere beskrive hva dere har gjort og begrunne valgene dere har gjort underveis. Rapporten skal også inneholde de tallene og figurene det spørres om, samt svar på spørsmål. Tall bør presenteres i tabeller. Det forventes at dere bruker hele setninger stikkord er ikke nok. Python-filene skal inneholde de ferdig implementerte klassene og funksjonene i pre-koden, samt koden dere har brukt for å produsere de resultatene dere presenterer i rapporten.

Del 1: Logistisk Regresjon

a) Bakgrunn

I den første delen av innleveringen skal vi bruke logistisk regresjon til å utvikle en enkel *språkidentifikator*, dvs. et lite system som skal predikere hvilket språk et ord hører til. Mer spesifikk skal systemet ta ordets *fonetiske transkripsjon* som input og returnere *navnet på språket* som ordet mest sannsynlig tilhører.

For å trene modellen (dvs. estimere gode verdier for vektene og skjæringspunktene) skal vi utnytte eksisterende lister over ord koblet med deres fonetiske transkripsjon i såkalt IPA-format.[1] Dere trenger ikke å selv kunne lese eller skrive slike fonetiske transkripsjoner i denne oppgaven. Det viktigste er å forstå at disse transkripsjonene beskriver språklydene som utgjør ordet, samt andre egenskaper slik som lengde, tone, trykk og intonasjon. Fonetiske transkripsjoner kan altså fortelle oss hvordan et ord bør uttales.

Vi starter med å importere filen `logistisk_regresjon.py` og laste ned dataene som skal brukes i denne delen av oppgaven:

```
>> import logistisk_regresjon
>> train, test = logistisk_regresjon.extract_wordlist()
```

Metoden `extract_wordlist` laster ned en rekke filer fra Github og samler innholdet av disse filene i en DataFrame, som er en datastruktur fra Python-biblioteket `pandas`. En DataFrame er en slags tabell med kolonner og rader. Biblioteket `pandas` gjør det veldig lett å visualisere og manipulere slike tabeller. Metoden `extract_wordlist` blander deretter sammen alle ordene og fordeler resultatet i to tabeller: et treningsett med 90 % av ordene og et testsett med de resterende 10 %.

Hver DataFrame har tre kolonner: "`ord`", "`IPA`" og "`språk`". Hver rad tilsvarer et ord, og vi kan enkelt bruke `pandas` til å titte inn i hver DataFrame:

```
>> test.head(n=20)
```

	ord	IPA	språk
49760	suggestopedia	ˈsugges¸tope¸dia	finsk
111148	neșifonabil	neʃifonabil	rumensk
76612	Angstgefühlen	ˈʔan¸kstgeːfylən	tysk
92939	justifiable	dʒˈʌstɪf¸aɪəbəl	engelsk
310358	泊低架車	pʰaːkㅓ teiㄱ	kantonesisk
288205	kadar	kadar	malayisk
17552	فكبي ا ين	fakabeɪaːnijj	arabisk
308693	低收入	teiㄱ seuㄱ jepㅓ	kantonesisk
196167	komp	kɔmːp	svensk
139579	régionalisations	ʁeʒjɔnalizasjɔ̃	fransk
267793	杲	kɑʊㅓㅣㅓ	mandarin
73820	Aufhängern	ˈʔaof¸hɛŋɛrn	tysk
314640	鱔	sœːŋㅓㅣ	kantonesisk
303306	santan	santan	malayisk
342490	tannlæknirinn	tʰanlaihknɪrɪn	islandsk
8013	tilbørligste	tɪlˈbøːlɪkstə	norsk
94897	resonates	ɹˈɛzən¸eɪts	engelsk
286415	tertanya	tərtaŋa	malayisk
145466	立論	ɾitsɯɾoɴ	japansk
144310	傾城傾国	keiseikeikokɯ	japansk

[1] IPA står for International Phonetic Alphabet.

b) Trening av modellen

Klassen `LanguageIdentifier` i filen `logistisk_regresjon.py` har flere metoder som ikke er implementert ennå. Vi skal benytte oss av en logistisk regresjonsmodell, mer spesifikk klassen `LogisticRegression` fra `scikit-learn`. Men for å kunne bruke modellen må vi naturligvis først estimere parametrene basert på treningsdata.

Hva slags trekk (*features*) skal vi bruke i modellen vår? I denne oppgaven skal vi gjøre det enkelt for oss selv, og kun ta i betrakning forekomst av bestemte IPA-symboler (som identifiserer ordlyder[2]) i den fonetiske transkripsjonen av ordet. Disse trekkene vil ha binære verdier (1 hvis transkripsjonen inneholder symbolet og 0 ellers) og kan sees som en slags "bag-of-sounds", siden de vil fortelle oss hvilke ordlyder som forekommer i ordet, men ikke i hvilken rekkefølge.

1. Det første skrittet er å lage en liste over alle IPA-symboler som finnes i treningsettet. Metoden `_extract_unique_symbols(transcriptions)` skal ta de fonetiske transkripsjonene fra treningssettet som input, og returnere en liste med alle fonetiske symboler (altså tegn) som finnes i disse transkripsjonene og forekommer minst 10 ganger. Implementer denne metoden. Antall symboler kan variere litt avhengig av den tilfeldige inndelingen mellom treningsettet og testsettet, men bør ligge på rundt 155 unike symboler.

2. Deretter må vi implementere metoden `_extract_feats(transcriptions)` som tar en liste fonetiske transkripsjoner og returnerer en matrise `X` hvor hver rad tilsvarer en transkripsjon og hver kolonne et trekk. La oss anta vi har laget en liste med m unike fonetiske symboler (med metoden ovenfor), og får som input en liste med n transkripsjoner $T = \{t_i \text{ hvor } 0 < i < n\}$. Metoden `_extract_feats(transcriptions)` må lage en matrise X av dimensjon (n, m), hvor hver matrisecelle X_{ij} er definert slik:

$$X_{ij} = \begin{cases} 1 \text{ hvis symbolet } j \text{ forekommer i transkripsjonen } i \\ 0 \text{ ellers} \end{cases} \tag{1}$$

   Tips: den enkleste fremgangsmåten kan være å starte med å lage en tom matrise slik som `X = np.zeros(n,m)` og deretter endre cellene hvor `X[i,j]` skal ha 1 som verdi i stedet for 0.

3. Vi er nå klar for å implementere hovedfunksjonen `train(transcriptions, languages)`. Metoden tar to lister som input, en liste fonetiske transkripsjoner og en liste med språknavn (de to listene må ha samme lengde). Metoden skal trene den logistiske regresjonsmodellen `self.model` ved å

[2]Hvis dere har studert litt fonetikk vil dere kanskje stusse på denne overforenklingen, da IPA-symboler brukes til å kode en god del andre egenskaper knyttet til talelyder slik som lengde, tone, trykk og intonasjon. Men i denne oppgaven skal vi gå den enkle veien og bruke alle symbolene i disse transkripsjonene uten å skille mellom ulike typer.

kalle `fit(X, y)`, hvor `X` er en matrise med alle trekk ekstrahert med metoden `_extract_feats(transcriptions)`, og `y` er outputklassene.

Merk at `scikit-learn` krever at outputklassene `y` må være en liste med heltall (og ikke strenger). Det betyr at dere må lage en mapping mellom språknavn og heltall (f.eks. ved å si at "norsk" er 0, "arabisk" er 1, "finsk" er 2, osv.).

Når dere har både matrisen `X` og output `y` er det bare å kalle metoden `fit(X, y)` for å trene modellen. Trening kan ta 1–2 minutter avhengig av maskinen deres.

c) Prediksjon og evaluering med modellen

Nå når modellen er trent kan vi anvende den på nye fonetiske transkripsjoner.

1. Implementer metoden `predict(transcriptions)`. Metoden tar som input en liste fonetiske transkripsjoner og predikerer det mest sannsynlige språket for hver transkripsjon. Listen som returneres må ha samme lengde som inputlisten. Husk at `scikit-learn` opererer med outputklasser representert som heltall, så dere må forvandle disse tallene tilbake til språknavnene. Deretter kan dere se hvordan modellen fungerer i praksis:

```
predicted_langs = model.predict(["konstituˈθjon", "grʉnlɔʊ",
                                 "stjourtnaɾskrau:ɪn", "bundɛsvɛrfaszʊŋ"])
print("Mest sansynnlige språk for ordene:", predicted_langs)
```

   (Svarene bør være spansk, norsk, islandsk og tysk).

2. Til slutt kan vi gjennomføre en grundigere evaluering av modellen basert på testsettet. Implementer metoden `evaluate(transcriptions, languages)`. Metoden skal beregne og skrive ut de følgende evalueringsmålene:

   - accuracy[3]
   - precision, recall og F1 for hvert språk
   - micro- og macro-averaged F1.

   For å beregne disse evalueringstallene kan dere bruke metodene fra `sklearn.metrics`.

d) Analyse av modellen

Vi kan spørre oss hva modellen egentlig har lært? En stor fordel med logistisk regresjon er at modellene er relativt enkle å tolke. Hvis en vekt w_i i modellen har stor positiv verdi betyr det at sansynnlighet for outputklassen *øker* sammen med trekket x_i. Likeledes betyr en negativ verdi at sannsynlighet for outputklassen *reduseres* med større verdier av x_i. Og jo større verdien er, jo større er effekten.

Det betyr at vi kan inspisere modellen for å finne ut hvilke språklyder som har størst effekt på prediksjonene. I `scikit-learn` er modellvektene lagret i

[3] Accuracy bør ligge rund 93 % hvis dere har gjort alt riktig.

variabelen `coef_` (merk underscoren ved slutten). Siden modellen vår er multiklasse (med 20 unike språk) og inneholder $m \approx 155$ trekk er vektene i `coef_` en matrise av dimensjon $(20, m)$.

- Finn ut hvilket fonetisk symbol som bidrar mest til å øke sansynnligheten for at et ord er klassifisert som norsk. Sjekk om det gir mening ved å telle hvor ofte symbolet forekommer i et norsk ord vs. ikke-norsk ord.
- Finn ut hvilket fonetisk symbol som bidrar mest til å redusere sansynnligheten for at et ord er klassifisert som norsk.

Søk gjerne på nettet for å finnes ut hva disse symbolene står for hvis dere er interessert!

Del 2: Sekvensmodeller

e) Bakgrunn

Vi skal nå jobbe på en viktig anvendelse av sekvensmodeller, nemlig å gjenkjenne navngitte entiteter (*Named Entity Recognition* eller NER på engelsk). For å gjøre det så enkelt som mulig vil vi bruke en *Hidden Markov Model* som sekvensmodell.[4] Hvert ord skal assosieres med en bestemte klasse, og vi skal ta i bruk såkalt BIO-annotering (også kalt IOB i boken til Jurafsky og Martin) for å spesifisere hvilke som ord hører til en navngitt entitet.

Vi skal bruke 2 filer, `norne_train.txt` som treningsett og `norne_test.txt` som testsett. Filene inneholder tokeniserte setninger (en per linje) hvor de navngitte entitetene er markert med XML-tags, som f.eks:

```
De første 43 minuttene hadde <ORG>Rosenborg</ORG> all makt og
tilnærmet full kontroll på <LOC>Fredrikstad Stadion</LOC> .
```

I eksempelet over har vi 2 navngitte enheter, "Rosenborg" (en organisasjon) og "Fredrikstad Stadion" (et sted). Python-filen `ner.py` inneholder prekoden som skal hjelpe oss med å utvikle en modul som automatisk skal markere tekst med slike navngitte enheter.

f) BIO-markering

I filen `ner.py` har vi allerede implementert en funksjon `preprocess(tagged_text)` som tar en tekst som input (som f.eks. setningene i trening- eller testsett) og ekstraherer lister over setninger og navngitte entiteter i disse setningene. De navngitte entiteter er spesifisert som tupler (i, j, tag) hvor i er indeksen for starten av entiteten, j er indeksen for slutten, og tag er entitetstypen, som f.eks.

[4] I praksis er ikke HMM den mest hensiktsmessige sekvensmodellen for å gjenkjenne navngitte enheter. Andre modeller slik som LSTMs (som er en typ nevralnettverk for sekvensdata) eller CRFs (en statistisk modell som ofte brukes på sekvenser eller komplekse strukturer) vil være bedre egnet til denne oppgaven, med disse modellene er mye mer kompliserte.

ORG eller LOC. Indekstallene er på ordnivå. I eksempelet over har vi altså to enheter: (5, 6, ORG) og (13, 15, LOC).

For å trene en HMM må hvert ord kobles til en merkelapp / *label*. En vanlig måte å gjøre dette er å bruke en såkalt BIO-markering, hvor hvert ord markeres som:

- 'O' (hvis ordet ikke tilhører en navngitt entitet)
- 'B-X' (hvis ordet er det første ordet i en navngitt entitet av type 'X')
- 'I-X' (hvis ordet tilhører en entitet av type 'X', men ikke er det første ordet)

Implementer funksjonen `get_BIO_sequence(spans, sentence_length)` som tar som input en liste med "text spans" og setningslengden og gir tilbake en rekke (av samme lengde som setningen) med BIO-markeringer.

g) Telling

Som vi så i forelesningen definerer vi en Hidden Markov Model med to sannsynlighetsfordelinger:

Den første fordelingen er kalt transisjonsmodelle og definert som $P(label_t|label_{t\text{-}1})$. Transisjonsmodellen forteller oss hvor sannsynlig det er at $label_{t\text{-}1}$ (assosiert med ord $w_{t\text{-}1}$) følges av $label_t$ (assosiert med ord w_t).

Den andre fordelingen er emisjonsmodellen, definert som $P(w_t|label_t)$. Emisjonsmodellen forteller oss hvor sannsynlig det er å observere ordet w_t hvis merkelappen for dette ordet er $label_t$.

For å estimere disse to sannsynlighetsfordelinger må vi telle:

- Alle ordene som forekommer i treningssettet
- Alle BIO-labels som forekommer i treningssettet
- Antall ganger hver BIO-label forekommer i treningssettet
- Antall ganger to BIO-labels følger hverandre i treningssettet
- Antall ganger et ord er observert med en BIO-label i treningssettet

Implementer metoden `_add_counts(sentence, label_sequence)` som oppdaterer variablene som inneholder disse tallene.

h) Sannsynligheterfordelinger

Med hjelp av disse tallene kan vi nå estimere transisjonsmodellen og emisjonsmodellen. Implementer metoden `_fill_probs()` som beregner disse to fordelingene.

For emisjonsmodellen bør dere legge til *Laplace smoothing* for å gjøre modellen mer robust, da treningssettet er relativt lite. La oss si at $C(label)$ er antall ganger BIO-merkelappen $label$ ble observert, og $C(label, token)$ antall ganger

ordet *token* ble observert sammen med *label*. Med Laplace smoothing definerer vi sannsynligheten $P(token|label)$ slik:

$$P(token|label) = \frac{C(label, token) + \alpha}{C(label) + \alpha V} \tag{2}$$

hvor V er størrelsen på vokabularet.

i) Dekoding med Viterbi

Til slutt skal vi implementere metoden `_viterbi(sentence)` som skal finne den mest sannsynlige label-sekvensen for en setning ved hjelp av Viterbi-algoritmen.

Algoritmen er beskrevet i kapittel 8 av Jurafsky og Martin. Kort forklart fungerer algoritmen ved å fylle ut en matrise L med en kolonne for hvert ord og en rad for hver mulig BIO-label. Hver celle $L_t(j)$ i matrisen representerer sannsynligheten for at HMM'en er i tilstand j etter å ha sett de første t ordene og passert gjennom den mest sannsynlige label-sekvensen. Verdien av hver celle beregnes rekursivt basert på cellene i den forrige kolonnen L_{t-1}. I tillegg holder vi også rede på de meste sannsynlige stiene ved å lagre såkalte "backpointers". Med disse backpointers kan vi enkelt ekstrahere den meste sannsynlige label-sekvensen når vi er ferdig med å beregne sannsynlighetsmatrisen L.

For å gjøre det litt lettere er metoden `_viterbi(sentence)` allerede delvis implementert, og den eneste som gjenstår er å programmere delen av algoritmen som fylle ut verdiene for matrisen L (latticen) og de "backpointers".

Når dere har implementert modellen ferdig kan dere teste ut hvordan den fungerer ved å kalle metoden `label(text)`:

```
>> label("Kjell Magne Bondevik var statsminister i Norge .")
'<PER>Kjell Magne Bondevik</PER> var statsminister i <GPE>Norge</GPE> .'
```

Valgfritt spørsmål, for de modigste blant dere:

Det er ofte en dårlig idé å gange mange små sannsynligheter, da vi kan havne i numerisk *underflow*. En vanlig løsning er å benytte log-sannsynligheter som kan summeres i stedet for å multipliseres, slik som forklart her:
`http://cs.columbia.edu/ mcollins/notes-on-logs.pdf`

Prøv å implementere Viterbi algoritmen med log-sannsynligheter.