

Istraživanje podataka u bioinformatičari - skripta za vežbe

Mart 2026

1 RSCU i CAI

Da bismo dobro razumeli gradivo koje sledi, pogledajmo sledeću analogiju:

DNK = tekst

gen = cela rečenica

kodon = reč od 3 slova

aminokiselina = značenje te reči

T,C,A,G = slova (timin, citozin, adenin, guanin)

Aminokiseline su organska jedinjenja koja sadrže amino ($-NH_2$) i karboksilnu grupu ($-COOH$) i predstavljaju osnovne gradivne elemente proteina (belančevina). One su ključne za rast, razvoj i funkcionisanje organizma, učestvujući u izgradnji mišića, tkiva, enzima i hormona. Poznato je oko 500 vrsta, ali svega 20 gradi proteine u ljudskom telu.

Različite aminokiseline imaju različite kodone koji ih predstavljaju. Jedna aminokiselina može biti predstavljena sa više kodona (jednom značenju može biti pridruženo više reči - na primer "auto", "kola", "automobil" predstavljaju isti objekat). Razlog za ovo je evolucijski. U slučaju da dodje do mutacije, i promeni se jedna baza u kodonu (npr. baza adenin predje u guanin u GAA, GAA $\rightarrow$ GAG), želimo da aminokiselina ostane ista, protein ostane isti i funkcionalan i da organizam ima veću šansu da preživi. U kodonu treća baza je najmanje bitna te tako GCU, GCC, GCA, GCG svi daju alanin (ovde je uracil U baza umesto timina T, jer se radi o RNK, a ne o DNK).

FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG

je standardni genetski kod, sekvenca koja mapira aminokiseline na kodone koji ih predstavljaju (na 64 mogućnosti redom F = TTT, F=TTC, L=TTA,... i * su stop kodoni). Aminokiseline su: F - fenilalanin, L - leucin, S - serin, Y - tirozin, C - cistein, * - stop kodon, itd. Stop kodon signalizira kraj sinteze proteina i posledica toga je kada ribozom naidje na ove triplete, sinteza prestaje i novoformirani protein se oslobadja.

```

bases = ['T', 'C', 'A', 'G']
codons = [a + b + c for a in bases for b in bases for c in
          bases]
amino_acids = 'FFLLSSSSYY**CC*
              WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG'
codon_table = dict(zip(codons, amino_acids))

```

- **zip** – spaja elemente iz više lista u parove (tuple-ove). Na primer: `zip([A,B], [1,2])` → (A,1), (B,2). Funkcija pravi iterator tuple-ova gde se elementi uparuju po poziciji [?].
- **dict** – pravi rečnik (mapu) ključ → vrednost iz parova (key, value).
- **Šta radi kod:**
 - generiše sve moguće kodone (kombinacije T, C, A, G dužine 3)
 - ima niz aminokiselina
 - pomoću `dict(zip(...))` pravi mapu:

kodon → aminokiselina

Sledeći korak je da napravimo rečnik oblika aminokiselina -> lista svih kodona koji je predstavljaju.

```

amino_codons_dict = {}
for codon, amino:
    if amino not in amino_codons_dict:
        amino_codons_dict[amino] = [codon]
    else:
        amino_codons_dict[amino].append(codon)

```

Ovaj kod nam daje rečnik oblika 'F' : ['TTT', 'TTC'],

RSCU (Relative Synonymous Codon Usage) predstavlja meru koliko često u nekom delu genoma koristimo određeni kodon za neku aminokiselinu u odnosu na to koliko bi se očekivalo da su svi kodoni te aminokiseline podjednako zastupljeni.

$$rscu[codon] = \frac{num_{codons} * num_{occurrence}}{num_{all-codons-for-that-amino-in-this-seq}}$$

gde je:

- num_{codons} broj kodona koji kodiraju aminokiselinu koju kodira naš kodon čiji rscu računamo
- $num_{occurrence}$ broj pojavljivanja našeg kodona u posmatranoj sekvenci
- $num_{all-codons-for-that-amino-in-this-seq}$ broj kodona koji se javljaju u sekvenci a kodiraju istu aminokiselinu kao naš

Primer: Aminokiselinu *F* kodiraju dva kodona *TTT* i *TTC*. Neka se u sekvenci (genu npr.) koju proučavamo *TTT* javlja 6 puta, a *TTC* 2 puta. U tom slučaju:

$$rscu['TTT'] = \frac{2 * 6}{6 + 2} = 12/8 = 1.5$$

RSCU koji je veći od 1 govori da se kodon javlja više puta nego što je očekivano, RSCU manji od 1 govori da se javlja manje puta, a RSCU jednak 1 je optimalan. RSCU je mera koja se koristi za analizu gena, poredjenje organizama, itd.

```
def my_RSCU(sequence):
    rscu = {}
    for codon in codons:
        rscu[codon] = 0.5
    num_occurrence_codon = {}
    num_occurrence_all_for_amino = {}
    for pos_codon in range(0, len(sequence)-2, 3):
        found_codon = sequence[pos_codon:pos_codon+3]
        if (found_codon not in num_occurrence_codon):
            num_occurrence_codon[found_codon] = 1
        else:
            num_occurrence_codon[found_codon] += 1
        amino = codon_table[found_codon]
        if (amino not in num_occurrence_all_for_amino):
            num_occurrence_all_for_amino[amino] = 1
        else:
            num_occurrence_all_for_amino[amino] += 1
    for codon in codons:
        if codon in num_occurrence:
            amino = codon_table[codon]
            rscu[codon] = (len(amino_codons_dict[amino]) *
                           num_occurrence_codon[codon] /
                           num_occurrence_all_for_amino[amino])
    return rscu
```

Primetimo da je na početku rscu za svaki kodon postavljen na 0.5, a ne na 0. To je zato što želimo da izbegnemo deljenje 0 a i da isključimo nerealističan slučaj da se neki kodon ne javlja uopšte. Veća je verovatnoća da je došlo do greške ili da sekvenca koju smo izabrali slučajno nije dovoljno ekspresivna za taj kodon. Naime, ljudski genom ima skoro 3 milijarde baza. Skupo je i resursno zahtevno da obradjujemo čitav genom: zahteva se sekvenciranje, analiza, skladištenje podataka, treba spojiti posle milijarde fragmenata, rešiti greške, analizirati varijacije, veoma je kompleksno, dobijamo previše informacija (to je ogroman broj mutacija, pri čemu ne znamo koje su bitne koje nisu, mnogo varijanti od toga je nepoznato, mnogo je bez efekta, teško za interpretaciju). Iz toga razloga obično čitamo samo jedan gen. To nije dovoljno za analizu čitavog organizma, ali jeste ako samo želimo da ispitujemo ponašanje toga gena.

CAI - Codon Adaptation Index, definiše adaptiranost korišćenih kodona u odnosu na kodone gena sa najvećom ekspresijom kod istog organizma. Relativna adaptiranost kodona predstavlja RSCU vrednost kodona u odnosu na najveću

vrednost RSCU sinonimnog kodona:

Primer: Uzmimo AAA i AAG. Neka je AAA kodon sa najvećim RSCU koji iznosi 1.8, a neka AAG ima 0.2. Tada je:

$$w = \frac{0.2}{1.8}$$

koliko je kodon dobar u odnosu na najbolji.

CAI je prosek svih w vrednosti. CAI blizak 1 govori o tome da gen koristi optimalne kodone i da ima visoku ekspresiju, a CAI blizak 0 govori da koristi loše kodone i da ima nisku ekspresiju. Ekspresija je uključenost gena u sintezu proteina. Na primer, gen za insulin ako se ekspresuje pravi se insulin, ako se ne ekspresuje ne pravi se insulin. CAI nam ustvari kaže koliko efikasno gen pravi protein. Ako je kodon "popularan" brzo se prevodi, ako nije, ribozom čeka i proces se usporava. Gen sa popularnijim kodonima ima ubrzan proces sinteze proteina i može da sintetiše veću količinu proteina.

Imamo i ugrađene funkcije koje nam izračunavaju pomenute metrike:

```
sequence = 'CGCATTACATTACGTACTGGTACA'
from CAI import RSCU, relative_adaptiveness, CAI
RSCU([sequence]) # RSCU iz CAI biblioteke ocekuje listu
                  # sekvenci kao argument
relative_adaptiveness([sequence])
```

1.1 Primer 1: Računanje RSCU nad kodirajućim sekvencama e. coli

FASTA fajlovi su jednostavan tekstualni format koji se koristi u bioinformatički za čuvanje bioloških sekvenci, kao što su DNK, RNK ili proteinske sekvence. Ovaj format je jedan od najčešće korišćenih standarda jer je lak za čitanje i obradu pomoću programskih jezika.

Svaka sekvenca u FASTA fajlu sastoji se iz dva osnovna dela:

- **Header (opis)** – linija koja počinje znakom > i sadrži identifikator sekvence i opcioni opis
- **Sekvenca** – niz karaktera koji predstavljaju nukleotide (A, T, C, G) ili aminokiseline

Primer FASTA zapisa:

```
>seq1 Homo sapiens gene
ATGCGTACGTAGCTAG
```

FASTA fajl može sadržati jednu ili više sekvenci, pri čemu svaka nova sekvenca počinje linijom sa znakom >.

Zbog svoje jednostavnosti i kompatibilnosti sa različitim bioinformatičkim alatima, FASTA format se široko koristi za:

- skladištenje sekvenci
- poredjenje sekvenci (npr. BLAST)
- analizu genoma i proteina

FASTA je osnovni i univerzalni format za rad sa biološkim sekvencama, koji omogućava lako čuvanje, razmenu i obradu podataka.

U narednom kodu najpre ćemo pročitati informacije o genima Ešerihije koli.

```
from Bio import SeqIO
import pandas as pd
records_raw = SeqIO.parse('data/e_coli_cds.fna', 'fasta')
sequences = [str(record.seq) for record in records_raw] # -
    Kopiranje radi mogućnosti kasnije ponovne iteracije
e_coli_rscu = RSCU(sequences)
e_coli_rscu_df = pd.DataFrame(data=[e_coli_rscu.values()],
    columns=e_coli_rscu.keys())
```

Ukoliko želimo malo složenije analize, možemo prvo nazive gena da povežemo sa njihovim zapisom:

```
'''Izdvajanje CDS kod e. coli koji odgovaraju imenovanim
    genima'''
import re # - Modul za rad sa regularnim izrazima

records_raw = SeqIO.parse('data/e_coli_cds.fna', 'fasta')
records = [record for record in records_raw] # - Kopiranje
    radi mogućnosti kasnije ponovne iteracije

gene_regex = r'gene=(.*?)\]' # - Regularni izraz za
    izdvajanje naziva gena iz FASTA zaglavlja
data = []
for record in records:
    match = re.search(gene_regex, record.description)
    if match != None:
        gene_name = match.groups()[0]
        sequence = str(record.seq)
        data.append([gene_name, sequence])
```

Od liste parova naziv gena - zapis gena, možemo lako napraviti tabelu na sledeći način:

```
gene_df = pd.DataFrame(data=data, columns=['gene', 'sequence'])
```

Ako hoćemo samo niz sekvenci, bez da znamo koje su možemo izvršiti sledeći kod:

```
gene_sequences = gene_df['sequence'].tolist()
```

A potom možemo saznati i RSCU nad svim sekvencama gena kod e. coli:

```
genome_rscu = RSCU(gene_sequences)
```

Zanima nas koji su geni najekspresivniji:

```
'''
Izracunavanje CAI pojedinačnih gena pri čemu se kao
referentna vrednost koristi RSCU sekvenci svih gena
Napomena: Referentne RSCU vrednost treba racunati na osnovu
sekvenci gena sa najvećom ekspresijom!
'''

cai_data = []

# Iteriranje kroz DataFrame gena - nukleotidnih sekvenci
gena
for index, row in gene_df.iterrows():
    gene = row.gene
    sequence = row.sequence
    cai = CAI(sequence, RSCUs=genome_rscu) # - Racunanje CAI
    cai_data.append([gene, cai])

# Konstrukcija DataFrame-a sa nazivima gena i njihovim CAI
vrednostima
cai_df = pd.DataFrame(data=cai_data, columns=['gene', 'CAI'])
cai_df
```

Računanje CAI vrednosti za gene

Ovaj kod računa **CAI (Codon Adaptation Index)** za svaki gen u datom skupu podataka. CAI predstavlja meru koliko je upotreba kodona u jednom genu slična upotrebi kodona u referentnom skupu gena, koji su najčešće visoko eksprimirani geni. Veće vrednosti CAI ukazuju na veću sličnost i potencijalno viši nivo ekspresije gena.

Opis algoritma

- Na početku se inicijalizuje prazna lista `cai_data`, koja će sadržati rezultate.
- Petlja:

```
for index, row in gene_df.iterrows():
```

prolazi kroz svaki red DataFrame-a `gene_df`, gde svaki red predstavlja jedan gen i njegovu nukleotidnu sekvencu.

- Iz svakog reda izdvajaju se:
 - `gene` – naziv gena

– `sequence` – DNK sekvenca gena

- Za svaki gen se računa CAI:

```
cai = CAI(sequence, RSCUs=genome_rscu)
```

pri čemu se koriste **RSCU vrednosti referentnog skupa gena**. Ove vrednosti treba da potiču od visoko eksprimiranih gena, jer CAI meri prilagodjenost kodonske upotrebe tom referentnom skupu [?].

- Rezultat (naziv gena i njegova CAI vrednost) se dodaje u listu:

```
cai_data.append([gene, cai])
```

- Nakon obrade svih gena, pravi se novi DataFrame:

```
cai_df = pd.DataFrame(data=cai_data, columns=['gene', 'CAI'])
```

- Dobijeni DataFrame sadrži dve kolone:

- naziv gena
- odgovarajuću CAI vrednost

Možemo dobiti spisak najekspresivnijih gena sortiranjem dobijenih CAI vrednosti:

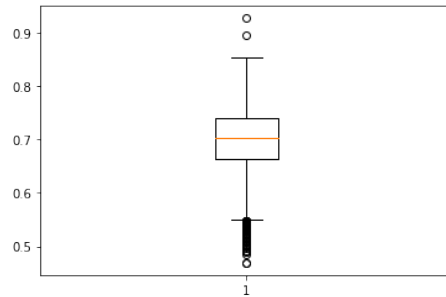
```
'''Generisanje sortiranog DataFrame-a sa CAI vrednostima  
gena u opadajućem redosledu'''
```

```
sorted_cai_df = cai_df.sort_values(by='CAI', ascending=False  
)  
sorted_cai_df
```

Lako se može napraviti i dijagram na osnovu koga se može lako sagledati ceo skup vrednosti za CAI:

```
import matplotlib.pyplot as plt  
_ = plt.boxplot(sorted_cai_df.iloc[:,1])
```

Većina gena (preko 50%) ima vrednosti CAI unutar pravougaonika na slici. Dosta gena ima nisku ekspresiju, a samo malo njih izuzetno visoku (tačkice na slici).



1.2 Primer 2: Konstrukcija filogenetskog stabla na osnovu RSCU

Stablo je konstruisano na osnovu RSCU vrednosti HBB gena kod 5 različitih organizama. Za konstrukciju stabla korišćen je algoritam za hijerarhijsko klasiranje iz biblioteke scipy.

```
base = 'data/hbb_sekvenca'
filepaths = [f'{base}/{f}' for f in os.listdir(base) if not
               f.startswith('.')]

data = []

for path in filepaths:
    record = SeqIO.read(path, 'fasta')
    label = path.split('/')[-1].split('.')[0]
    sequence = record.seq

    rscu_table = RSCU([sequence])

    row = {'label': label}
    row.update(rscu_table)

    data.append(row)

df = pd.DataFrame(data).reset_index(drop=True)
```

Dobićemo sledeću tabelu sa izračunatim RSCU kod HBB gena različitih organizama: Na osnovu dobijene tabele možemo uraditi hijerarhijsko klasterovanje

	label	TTT	TTC	TTA	TTG	TCT	TCC	TCA	TCG	TAT	...	GCA	GCG	GAT	GAC
0	hbb_chinchilla	0.444444	1.555556	0.157895	0.157895	1.600000	1.600000	0.400000	0.400000	0.400000	...	0.108108	0.216216	0.500000	1.500000 0.54
1	hbb_human	1.250000	0.750000	0.150000	0.150000	0.923077	1.846154	0.461538	0.461538	1.333333	...	0.125000	0.125000	1.428571	0.571429 0.50
2	hbb_mullatta	1.250000	0.750000	0.153846	0.153846	0.923077	1.846154	0.461538	0.461538	0.666667	...	0.142857	0.142857	1.428571	0.571429 0.50
3	hbb_cebus	1.000000	1.000000	0.150000	0.150000	0.923077	1.846154	0.461538	0.461538	0.666667	...	0.133333	0.133333	1.000000	1.000000 0.85
4	hbb_sapajus	1.000000	1.000000	0.150000	0.150000	0.923077	1.846154	0.461538	0.461538	0.666667	...	0.133333	0.133333	1.000000	1.000000 0.85
5	hbb_hololepus	1.250000	0.750000	0.146341	0.878049	0.375000	3.000000	0.375000	0.375000	1.000000	...	0.133333	0.133333	1.111111	0.888889 1.25

i dendrogram.

```

import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage

# 1. Priprema podataka
# uklonimo label kolonu (to su imena gena)
X = df.drop(columns=['label']).values

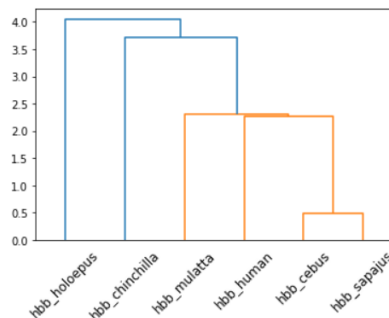
# 2. Hijerarhijsko klasterovanje
Z = linkage(X, method='ward') # moze i 'single', 'complete', 'average'

# 3. Crtanje dendrograma
plt.figure(figsize=(10, 6))
dendrogram(Z, labels=df['label'].values)

plt.title("Dendrogram (Hijerarhijsko klasterovanje gena)")
plt.xlabel("Geni")
plt.ylabel("Distanca")

plt.xticks(rotation=90)
plt.tight_layout()
plt.show()

```



Hijerarhijsko klasterovanje i dendrogram

Hijerarhijsko klasterovanje je metoda nenadgledanog učenja koja grupiše objekte na osnovu njihove međusobne sličnosti i formira hijerarhiju klastera. Za razliku od nekih drugih metoda, nije potrebno unapred zadati broj klastera, već se dobija kompletna struktura grupisanja podataka.

Postoje dva osnovna pristupa:

- **Agglomerativno (bottom-up)** – svaki objekat počinje kao poseban klaster, a zatim se najbliži klasteri postepeno spajaju
- **Divizivno (top-down)** – svi objekti počinju u jednom klasteru koji se zatim deli na manje

Rezultat hijerarhijskog klasterovanja se prikazuje pomoću **dendrograma**, koji predstavlja strukturu klastera u obliku stabla. Listovi stabla predstavljaju pojedinačne objekte, dok unutrašnji čvorovi predstavljaju spajanje klastera.

Ward metoda je posebna vrsta hijerarhijskog (aglomerativnog) klasterovanja koja ne gleda samo rastojanje između klastera, već promenu varijanse unutar klastera.

Dendrogram

Dendrogram je dijagram koji prikazuje kako se objekti grupišu tokom klasterovanja. Visina na kojoj se dve grane spajaju predstavlja njihovu međusobnu udaljenost (razliku).

Važne osobine dendrograma:

- listovi predstavljaju pojedinačne objekte (npr. gene)
- grane predstavljaju spajanje klastera
- visina spajanja označava koliko su objekti različiti

Dendrogram omogućava da se broj klastera odredi tako što se "preseče" na određenoj visini, pri čemu objekti ispod preseka pripadaju istom klasteru.

Primena na RSCU podatke

U ovom primeru, hijerarhijsko klasterovanje se primenjuje na gene opisane njihovim RSCU vrednostima. Svaki gen se posmatra kao vektor numeričkih vrednosti koje predstavljaju upotrebu kodona.

Klasterovanje omogućava:

- identifikaciju gena sa sličnim obrascima upotrebe kodona
- grupisanje gena koji mogu imati sličan nivo ekspresije
- otkrivanje potencijalnih funkcionalnih ili evolutivnih sličnosti

Na dendrogramu:

- geni koji se spajaju na maloj visini imaju sličnu kodonsku upotrebu
- geni koji se spajaju na većoj visini su međusobno različitiji

Zaključak

Hijerarhijsko klasterovanje i dendrogram predstavljaju moćan alat za vizualizaciju i analizu sličnosti između gena. U kontekstu RSCU vrednosti, oni omogućavaju bolje razumevanje obrazaca kodonske upotrebe i potencijalnih bioloških odnosa između gena.

2 Probabilistički grafički modeli - uvod

Kompleksni sistemi su okarakterisani mnogim nekoreliranim aspektima od kojih mnogi utiču na ishod zadatka o kome rezonujemo. Na primer, u slučaju postavljanja dijagnoze, postoji mnogo bolesti koje pacijent može da ima, stotine ili hiljade simptoma i dijagnostičkih testova, lične karakteristike koje predstavljaju predispozicije za neke bolesti i još gomila faktora. Ovi domenii mogu se predstaviti slučajnim promenljivama (*random variables*). Konkretna bolest, kao što je Grip (*Flu*) može imati vrednosti prisutan (*present*) ili odsutan (*absent*), simptom ove bolesti Groznica (*Fever*) može imati neprekidnu vrednost. Skup promenljivih i njihovih vrednosti važno je pitanje dizajna i zavisi od toga na kakvo pitanje o problemu želimo da damo odgovor.

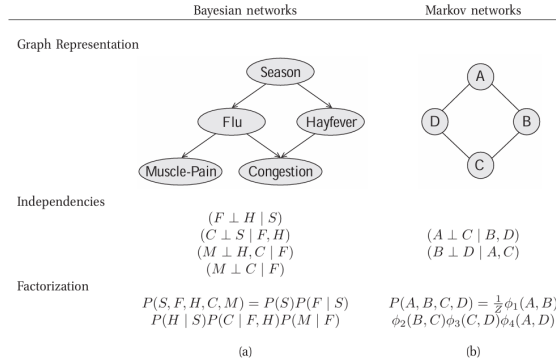
Naš zadatak je da na probabilistički način rezonujemo o vrednosti jedne ili više promenljivih, ako su nam pritom date observacije o drugim promenljivama. Da bismo to uradili, konstruišemo zajedničku raspodelu (*joint distribution*) nad skupom slučajnih promenljivih X . Ovaj tip modela nam omogućuje da odgovorimo na veliki skup interesantnih upita. Na primer, možemo da napravimo pretpostavku da slučajna promenljiva X_i uzima vrednost x_i i da se pitamo kakva je raspodela verovatnoća nad vrednostima druge promenljive X_j .

*Razmotrimo jedan veoma jednostavan primer. Posmatramo Grip i Prehladu. Ove dve bolesti nisu uzajamno isključive, neko može imati i jedno i drugo. Takođe imamo jednu slučajnu veličinu Sezona, koja može da uzima 4 vrednosti - Proleće, Leto, Jesen i Zima. I dva simptoma BolMišića i ZačepljenNos, za koje beležimo da li su prisutni ili ne. Ukupno imamo $2*2*4*2*2 = 64$ moguće kombinacije. Ako nam je data zajednička raspodela, možemo da se pitamo kolika je verovatnoća da pacijent ima Grip ako je Jesen i ima ZačepljenNos, ali ne i BolMišića. Ovo se matematički može zapisati kao:*

$$P(Flu = true)|(Seasson = fall, Congestion = true, MusclePain = false)$$

2.1 Probabilistički grafički modeli

Probabilistički grafički modeli služe da nam omoguće da kompaktno zapišemo komplikovane raspodele u visoko dimenzionim prostorima. Čvorovi na graficima odgovaraju promenljivama, a grane direktnim probabilističkim vezama između njih: Za primer koji smo malopre imali vidimo da nema interakcije između bola u mišićima i sezone, ali oba direktno interaguju sa gripom. Ovo je dualna perspektiva koju možemo iskoristiti da interpretiramo graf. Oznaka $(X \perp Y|Z)$ znači da je X nezavisno od Y ako je vrednost od Z poznata. Na primer, $(ZačepljenNos \perp Sezona | Grip, Prehlada)$ znači da ako je poznato da li pacijent ima grip i da li ima prehladu onda sezona nije relevantna informacija za to da li mu je nos začepljen ili ne. Ipak moramo paziti, ovo ne znači da sezona ne utiče na raspodelu verovatnoća za začepljenost nosa, već da sve informacije koje je sezona proizvela i sav uticaj se preslikao kroz podatke o gripu i prehladi. Drugi važan razlog zašto nam je graf potreban je što nam pomaže da neku komplikovanu verovatnoću sa puno promenljivih razbijemo na faktore koje množimo i koji su



jednostavni. Recimo verovatnoća događaja: "proleće je, osoba nema grip, ima prehladu, začepljen nos i bol u mišićima" može se zapisati kao proizvod sledećih verovatnoća:

$$P(\text{Season} = \text{spring}), P(\text{Flu} = \text{false} \mid \text{Season} = \text{spring}), P(\text{Hayfever} = \text{true} \mid \text{Season} = \text{spring})$$

$$P(\text{Congestion} = \text{true} \mid \text{Flu} = \text{false}, \text{Hayfever} = \text{true}), P(\text{MusclePain} = \text{true} \mid \text{Flu} = \text{False})$$

U ovim izrazima jasno je da verovatnoća neke konkretne vrednosti zavisi samo od čvorova koji su u grafu spojeni sa čvorom te promenljive. Izučavaćemo dve familije grafičkog prikaza distribucija: Bajesove mreže - koje su usmereni grafovi i Markovljeve - koje su neusmereni. Sa obe čitamo nezavisnosti i faktorizacije.

2.2 Bajesove mreže

Naš cilj je da predstavimo zajedničku raspodelu P nad nekim skupom slučajnih promenljivih $X = \{X_1, \dots, X_n\}$. Čak i u najjednostavnijem slučaju kada su sve promenljive binarne, zajednička raspodela bi zahtevala verovatnoće 2^n različitih mogućnosti. Eksplicitno čuvanje takvih informacija je nezamislivo iz svake perspektive. Računski je veoma skupo čuvati i manipulirati sa tom količinom informacija, kognitivno je nemoguće prikupiti toliko informacija od ljudskih eksperata, statistički ako hoćemo raspodelu da naučimo iz podataka, užasno mnogo podataka bi nam trebalo da naučimo parametre robusno. Svi ovi razlozi doveli su do razvoja teorije o kojoj ćemo ovde da govorimo.

Razmotrimo sledeći primer. Kompanija želi da zaposli pametne studente, ali nema način da direktno testira inteligenciju. Jedino što poseduje je pristup SAT skoru, koji jeste informativan ali ne potpuno indikativan. Znači naš prostor verovatnoća indukovanih je dvema važnim promenljivama Inteligencija i SAT. Zbog jednostavnosti možemo pretpostaviti da svaka od njih uzima po dve vrednosti: $\text{Val}(I) = \{i^1, i^0\}$ i $\text{Val}(S) = \{s^1, s^0\}$ pri čemu je 1 oznaka za višu vrednost i 0 za nižu. Primer jedne zajedničke raspodele može biti:

i^0	s^0	0.665
i^0	s^1	0.035
i^1	s^0	0.06
i^1	s^1	0.24

Postoji naravno alternativa. Možemo zapisati: $P(I, S) = P(I) * P(S|I)$. Predstavljamo proces na način koji je intuitivno saglasniji sa uslovnošću. Gomila faktora (geni, okruženje, dostupnost resursa,...) uticali su najpre na razvoj studentove inteligencije. Njegov SAT skor je potom pod uticajem inteligencije. Iz matematičke perspektive, ovakav način razmišljanja nas dovodi do toga da umesto da čuvamo jednu visoko dimenzionu tabelu, čuvamo nekoliko manjih, a vrednosti verovatnoća koje nas zanimaju dobijamo množenjem odgovarajućih faktora. Za ovaj konkretni slučaj čuvamo $P(I)$:

i^0	0.7
i^1	0.3

i $P(S|I)$

I	s^0	s^1
i^0	0.95	0.05
i^1	0.2	0.8

Verovatnoća da student dobije veliki SAT skor, a da nije visoko inteligentan je mala 0.05, sa druge strane verovatnoća da dobije veliki SAT skor ako je visoko inteligentan je velika 0.8, ali nije 100%.

Pretpostavimo sada da osim ova dva, imamo i ocenu (prosečnu, ali jednostavnosti radi smatrajmo je diskretnom) studenta koja može uzeti neke tri vrednosti npr. $Val(G) = \{g^1, g^2, g^3\}$ koje predstavljaju ocene A, B i C. In-

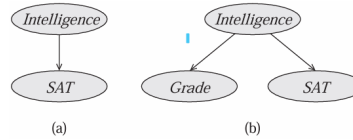


Figure 3.1 Simple Bayesian networks for the student example

teligencija studenta je očigledno povezana sa SAT skorom i sa ocenom. Takođe i SAT skor i ocena su povezani, ali su uslovno nezavisni u smislu koji smo već opisali, tj. ako znamo inteligenciju studenta, skor nam ne nosi nikakvu novu informaciju o oceni. Odnosno, formalnije zapisano:

$$P(g|i^1, s^1) = P(g|i^1)$$

ili

$$S \perp G | I$$

Ovde je bitna pretpostavka da je inteligencija studenta jedini razlog, jedina veza između SAT skora i ocene. Naravno da ovo nije realna pretpostavka, ali

je možda dobra aproksimacija sistema. Ovde se već može naslutiti intuicija sa grafika, za sve informacije koje su poznate, obrišemo njihove čvorove, oni čvorovi koji ostanu nepovezani u takvom grafiku uslovno su nezavisni. Možemo faktorisati raspodelu verovatnoća:

$$P(I, S, G) = P(S, G|I) * P(I)$$

$$P(S, G|I) = P(S|I) * P(G|I)$$

jer I utiče na S i G , a S i G su uslovno nezavisne u odnosu na I , odakle je konačno:

$$P(I, S, G) = P(S|I) * P(G|I) * P(I)$$

2.2.1 Naivni Bajes

U ovoj podsekciji opisaćemo naivni Bajesov model (*Idiot Bayes model*). Pretpostavka je da postoji jedna promenljiva C (u primeru sa studentima je to Inteligencija) koja utiče na ostale $\{X_1, \dots, X_n\}$ (SAT skor, ocena) i jedina je stvar koja utiče, odnosno one su sve međusobno uslovno nezavisne - kad je C poznato, one ne zavise jedna od druge. Raspodela verovatnoća se ovde faktoriše

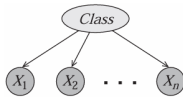


Figure 3.2 The Bayesian network graph for a naive Bayes model

kao:

$$P(C, X_1, \dots, X_n) = P(C) \prod_{i=1}^n P(X_i|C)$$

Ovi faktori se mogu sačuvati sa malim brojem parametara - linearnim, za razliku od eksponencijalnog broja kada bismo čuvali zajedničku raspodelu.

Bez obzira što pravi ozbiljno jake pretpostavke, Bajesov model se u praksi koristi zbog svoje jednostavnosti i malog broja potrebnog parametra. Značajna je primena za klasifikaciju - odlučivanje zasnovano na vrednosti nekih parametara koji predstavljaju dokaze za datu instancu, kojoj klasi pripada. Formalno, to možemo zapisati kao:

$$\frac{P(C = c^1|x_1, \dots, x_n)}{P(C = c^2|x_1, \dots, x_n)} = \frac{P(C = c^1)}{P(C = c^2)} \prod_{i=1}^n \frac{P(x_i|C = c^1)}{P(x_i|C = c^2)}$$

Interesantno je da se u svojim ranim danima, ovaj model koristio za medicinske dijagnoze, promenljiva C je predstavljala bolest (njene vrednosti su odgovarale različitim bolestima) a promenljive $X_1, \dots, X_n$ simptome, i rezultate različitih testova. Ovaj model je korišćen za medicinsku dijagnostiku jer je mali broj interpretabilnih parametara olakšavao njihovo dobijanje od stručnjaka. Na

primer, sasvim je prirodno pitati lekara eksperta kolika je verovatnoća da pacijent sa upalom pluća ima visoku temperaturu. Zaista, nekoliko ranih sistema za medicinsku dijagnostiku bilo je zasnovano na ovoj tehnologiji, a pokazano je da su neki od njih davali bolje dijagnoze od onih koje postavljaju lekari eksperti. Naravno, ovaj metod ima i ozbiljne mane. Pretpostavlja da pacijent nema istovremeno više bolesti, a takodje i da su $X_1, \dots, X_n$ medjusobno nezavisni, a oni mogu biti baš ozbiljno korelisani, na primer gojaznost i visok pritisak su oba znaci za probleme sa srcem, ali pošto su povezani, njihov uticaj se duplo računa. Iz ovog razloga razvijeni su kompleksniji Bajesovi modeli. Ipak, naivni Bajes se tradicionalno koristi u oblasti gde daje dobre rezultate, domenima sa puno featura i malo x -eva, kao što je recimo klasifikacija dokumenta u teme, na osnovu reči koje se javljaju u tim dokumentima.

2.3 Bajesove mreže

U ovom modelu posmatra se nešto složenija situacija u kojoj ocena studenta ne zavisi samo od njegove inteligencije, već i od težine kursa. Pored toga, profesor piše preporuku za studenta, ali pošto ga se ne seća, oslanja se isključivo na njegovu ocenu, pa kvalitet preporuke zavisi od ocene i odredjen je nasumično. U modelu se zato pojavljuje pet slučajnih promenljivih: inteligencija (I), težina kursa (D), ocena (G), SAT rezultat (S) i kvalitet preporuke (L), pri čemu su sve binarne osim ocene koja ima tri vrednosti, pa ukupna zajednička raspodela ima 48 mogućih kombinacija.

Ovaj sistem se predstavlja Bajesovom mrežom, odnosno usmerenim grafom u kome čvorovi predstavljaju promenljive, a grane direktne uticaje medju njima. Ideja je da se ceo proces može posmatrati kao generativni proces u kome svaka promenljiva dobija vrednost na osnovu verovatnoće koja zavisi samo od njenih "roditelja". Tako inteligencija i težina kursa nezavisno utiču na ocenu, inteligencija dodatno utiče na SAT rezultat, dok kvalitet preporuke zavisi samo od ocene. Na taj način model jasno prikazuje lokalne zavisnosti izmedju promenljivih, koje zajedno formiraju celokupno ponašanje sistema.

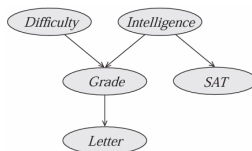
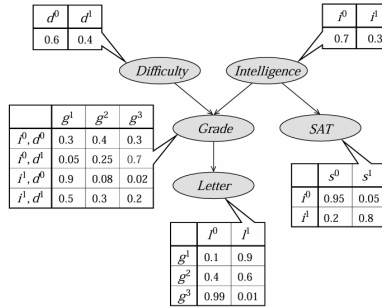


Figure 3.3 The Bayesian Network graph for the Student example

Moramo imati i uslovne verovatnoće kao na slici: da bi mogli da izračunamo npr. verovatnoću da inteligentan student na lakom kursu ima nisku ocenu a da je imao visok SAT skor i loše pismo:

$$P(i^1, d^0, g^2, s^1, l^0) = P(i^1)P(d^0)P(g^2|i^1, d^0)P(s^1|i^1)P(l^0|g^2) = 0.004608$$



Bajesova mreža predstavlja zajedničku raspodelu verovatnoća, iz koje možemo računati bilo koju uslovnu verovatnoću $P(Y|E)$. To radimo tako što uzmemo samo one slučajeve koji su u skladu sa posmatranim dokazima (evidence). Tipovi zaključivanja koje možemo da sprovedemo u ovakvim mrežama su sledeći:

- kauzalno zaključivanje (uzrok $\rightarrow$ posledica)
 - na primer inteligencija $\rightarrow$ ocena $\rightarrow$ preporuka
 - ako znamo da je student slab, smanjuje se verovatnoća dobre preporuke
 ovo je predikcija
- evidencijalno zaključivanje (posledica $\rightarrow$ uzrok)
 - loša ocena $\rightarrow$ manja verovatnoća da je student visoko inteligentan
 - preporuka takodje daje informaciju ali slabiju od ocene
 ovo je objašnjenje
- interkauzalno zaključivanje (izmedju uzroka) - dešava se kada više uzroka utiče na istu posledicu, najvažniji efekat je *explaining away*:
 - ako znamo posledicu npr. loša ocena, moguće je više uzroka (niska inteligencija ili težak kurs)
 - ako otkrijemo jedan uzrok - npr. kurs je težak, verovatnoća drugog uzroka se menja - u ovom slučaju smanjuje se verovatnoća niske inteligencije

2.4 Bajesovi modeli i osnovne komponente u pgmpy

Bajesovi modeli predstavljaju jedan od najznačajnijih pristupa u okviru probabilističkih grafičkih modela. Oni omogućavaju modelovanje zavisnosti izmedju slučajnih promenljivih na formalan i strukturiran način, koristeći usmerene aciklične grafove. U kontekstu bioinformatike, ovakvi modeli se često koriste

za opisivanje složenih bioloških procesa, kao što su regulacija ekspresije gena, signalni putevi ili odnosi između genetičkih i fenotipskih karakteristika.

Implementacija Bajesovih modela u programskom jeziku Python može se realizovati pomoću biblioteke `pgmpy`, koja pruža alate za definisanje strukture modela, zadavanje verovatnoća i izvodjenje zaključivanja. Osnovne komponente koje ćemo koristiti date su u nastavku:

Listing 1: Osnovne komponente za rad sa Bajesovim modelom

```
# Bajesov model
from pgmpy.models import BayesianModel

# Conditional Probability Table, funkcije uslovne raspodele
# tabelarno predstavljena
from pgmpy.factors.discrete import TabularCPD

# Algoritam koji vrsi zakljucivanje
from pgmpy.inference import VariableElimination
```

Klasa `BayesianModel` koristi se za definisanje strukture modela, odnosno zavisnosti između promenljivih. Klasa `TabularCPD` omogućava zadavanje uslovnih raspodela verovatnoća u tabelarnom obliku, što predstavlja ključni korak u parametrizaciji modela. Na kraju, `VariableElimination` predstavlja algoritam za zaključivanje, koji omogućava izračunavanje verovatnoća interesantnih događaja na osnovu poznatih podataka.

U nastavku ćemo, polazeći od ovih osnovnih komponenti, konstruisati konkretan Bajesov model i analizirati njegovo ponašanje.

2.4.1 Algoritam eliminacije promenljivih

Jedan od osnovnih zadataka u radu sa Bajesovim mrežama jeste **zaključivanje**, odnosno izračunavanje verovatnoća određenih promenljivih na osnovu poznatih informacija. U tu svrhu koristi se više algoritama, a jedan od najpoznatijih i najčešće korišćenih jeste **eliminacija promenljivih** (*Variable Elimination*).

Algoritam eliminacije promenljivih zasniva se na ideji da se zajednička raspodela verovatnoće može efikasno izračunati tako što se postepeno eliminišu (sumiraju) promenljive koje nisu od interesa. Umesto da se računa kompletna zajednička raspodela nad svim promenljivima, što je u praksi često neizvodljivo zbog eksponencijalnog rasta, ovaj algoritam koristi strukturu grafa kako bi značajno smanjio broj operacija.

Intuitivno, postupak funkcioniše tako što:

- identifikujemo promenljive koje želimo da izračunamo (upit),
- uzimamo u obzir poznate vrednosti nekih promenljivih (evidenciju),
- ostale promenljive postepeno eliminišemo sumiranjem preko njihovih mogućih vrednosti.

Tokom ovog procesa, koriste se lokalne funkcije verovatnoće (CPD-ovi), koje se međusobno množe i marginalizuju. Redosled eliminacije promenljivih može značajno uticati na efikasnost algoritma, jer određuje veličinu međukoraka (intermedijarnih tabela).

U biblioteci `pgmpy`, algoritam eliminacije promenljivih implementiran je kroz klasu `VariableElimination`. Nakon što definišemo model i njegove raspodele verovatnoća, možemo inicijalizovati objekat za zaključivanje na sledeći način:

Listing 2: Inicijalizacija algoritma eliminacije promenljivih

```
from pgmpy.inference import VariableElimination

inference = VariableElimination(model)
```

Nakon inicijalizacije, moguće je postavljati upite nad modelom. Na primer, možemo izračunati raspodelu verovatnoće za određenu promenljivu ili skup promenljivih, uz eventualno zadatu evidenciju. Ovakav pristup omogućava fleksibilno i efikasno zaključivanje u složenim probablističkim modelima.

Važno je naglasiti da eliminacija promenljivih daje **tačne rezultate**, za razliku od nekih aproksimativnih metoda, ali njena vremenska i memorijska složenost može biti velika kod mreža sa velikim brojem čvorova ili gustim povezivanjem. Uprkos tome, zbog svoje konceptualne jednostavnosti i preciznosti, ovaj algoritam predstavlja standardni alat za zaključivanje u Bajesovim mrežama.

Marginalna verovatnoća Marginalna verovatnoća predstavlja verovatnoću jedne slučajne promenljive, nezavisno od vrednosti ostalih promenljivih u modelu. Drugim rečima, ona se dobija tako što se “zanemare” (eliminšu) ostale promenljive, najčešće sumiranjem preko svih njihovih mogućih vrednosti.

Neka imamo dve slučajne promenljive X i Y sa zajedničkom raspodelom $P(X, Y)$. Marginalna verovatnoća promenljive X definiše se kao:

$$P(X) = \sum_Y P(X, Y)$$

Intuitivno, ovo znači da verovatnoću događaja X dobijamo tako što sabiramo verovatnoće svih mogućih slučajeva u kojima se X pojavljuje, bez obzira na to koja je vrednost promenljive Y .

Na primer, ako promenljiva Y može da uzme dve vrednosti y_1 i y_2 , tada važi:

$$P(X) = P(X, y_1) + P(X, y_2)$$

U kontekstu Bajesovih mreža, marginalna verovatnoća se često računa pomoću algoritma eliminacije promenljivih, gde se neželjene promenljive eliminišu upravo ovakvim sumiranjem. Na taj način dobijamo verovatnoću promenljive od interesa bez potrebe da eksplicitno razmatramo sve kombinacije vrednosti u modelu.

Primer rada algoritma eliminacije promenljivih Da bismo razumeli kako funkcioniše eliminacija promenljivih, razmotrimo jednostavan model sa tri promenljive:

- A – uzrok,
- B – posredna promenljiva,
- C – posledica.

Pretpostavimo da važi struktura:

$$A \rightarrow B \rightarrow C$$

Zajednička raspodela ovih promenljivih može se zapisati kao:

$$P(A, B, C) = P(A) \cdot P(B | A) \cdot P(C | B)$$

Neka nas zanima marginalna verovatnoća promenljive C , odnosno $P(C)$. Direktno računanje bi zahtevalo sumiranje preko svih mogućih vrednosti promenljivih A i B :

$$P(C) = \sum_A \sum_B P(A) \cdot P(B | A) \cdot P(C | B)$$

Algoritam eliminacije promenljivih omogućava da se ovaj izraz izračuna efikasnije, reorganizacijom izraza i postepenim eliminisanjem promenljivih.

Prvo eliminišemo promenljivu A . Grupisanjem odgovarajućih faktora dobijamo:

$$\sum_A P(A) \cdot P(B | A)$$

Rezultat ove sume je nova funkcija koja zavisi samo od promenljive B (to je zapravo marginalna raspodela $P(B)$). Označimo je sa $f(B)$:

$$f(B) = \sum_A P(A) \cdot P(B | A)$$

Sada izraz za $P(C)$ postaje:

$$P(C) = \sum_B f(B) \cdot P(C | B)$$

U sledećem koraku eliminišemo promenljivu B :

$$P(C) = \sum_B \left(\sum_A P(A) \cdot P(B | A) \right) \cdot P(C | B)$$

Na ovaj način izbegavamo računanje kompletne zajedničke raspodele $P(A, B, C)$, već radimo sa manjim, lokalnim funkcijama.

Ključna ideja algoritma je sledeća:

- kombinujemo (množimo) samo relevantne faktore,

- eliminišemo promenljive sumiranjem,
- kreiramo nove, manje faktore koji se koriste u narednim koracima.

Ovakav pristup značajno smanjuje broj operacija u odnosu na naivno računanje, posebno kod modela sa većim brojem promenljivih. Upravo zbog toga, eliminacija promenljivih predstavlja osnovni algoritam za tačno zaključivanje u Bajesovim mrežama.

Numerički primer marginalne verovatnoće i eliminacije promenljivih

Razmotrimo model sa tri binarne promenljive A , B i C :

$$A \rightarrow B \rightarrow C$$

Neka su date sledeće verovatnoće:

- Apriorna verovatnoća:

$$P(A = 1) = 0.3, \quad P(A = 0) = 0.7$$

- Uslovna verovatnoća B dato A :

$$P(B = 1 \mid A = 1) = 0.8, \quad P(B = 0 \mid A = 1) = 0.2$$

$$P(B = 1 \mid A = 0) = 0.2, \quad P(B = 0 \mid A = 0) = 0.8$$

- Uslovna verovatnoća C dato B :

$$P(C = 1 \mid B = 1) = 0.9, \quad P(C = 0 \mid B = 1) = 0.1$$

$$P(C = 1 \mid B = 0) = 0.1, \quad P(C = 0 \mid B = 0) = 0.9$$

Želimo da izračunamo marginalnu verovatnoću $P(C = 1)$.

Korak 1: eliminacija promenljive A

Prvo računamo raspodelu promenljive B :

$$P(B = 1) = P(A = 1)P(B = 1 \mid A = 1) + P(A = 0)P(B = 1 \mid A = 0)$$

$$P(B = 1) = 0.3 \cdot 0.8 + 0.7 \cdot 0.2 = 0.24 + 0.14 = 0.38$$

$$P(B = 0) = 0.3 \cdot 0.2 + 0.7 \cdot 0.8 = 0.06 + 0.56 = 0.62$$

Korak 2: eliminacija promenljive B

Sada računamo $P(C = 1)$:

$$P(C = 1) = P(B = 1)P(C = 1 \mid B = 1) + P(B = 0)P(C = 1 \mid B = 0)$$

$$P(C = 1) = 0.38 \cdot 0.9 + 0.62 \cdot 0.1 = 0.342 + 0.062 = 0.404$$

Dakle, dobijamo:

$$P(C = 1) = 0.404$$

Ovaj primer jasno ilustruje ideju eliminacije promenljivih: prvo smo eliminisali A da bismo dobili raspodelu nad B , a zatim eliminisali B da bismo dobili verovatnoću za C . Na taj način izbegavamo računanje kompletne zajedničke raspodele nad svim promenljivima.

Primer: Bajesova mreža za dijagnostiku COVID-19 Razmotrimo jednostavan model koji opisuje odnos između simptoma i dijagnostičkih nalaza kod bolesti COVID-19. Posmatramo sledeće promenljive:

- `kasalj` – da li pacijent ima kašalj,
- `temperatura` – da li pacijent ima povišenu temperaturu,
- `X-ray` – rezultat rendgenskog snimka,
- `covid` – da li je pacijent zaražen.

Struktura modela je takva da simptomi utiču na rezultat rendgenskog snimka, dok nalaz rendgena dalje utiče na procenu. Na osnovu ove strukture, definišemo odgovarajuće uslovne raspodele verovatnoća (CPD-ove):

Listing 3: Definisanje CPD-ova za model COVID dijagnostike

```
# Kasalj, Temperatura, X-ray, COVID

'''
kasalj    temperatura
  \      /
   X-ray
   |
   covid
'''

K_cpd = TabularCPD(
    variable="kasalj",
    variable_card=2,
    values=[[0.4],[0.6]],
    state_names = { 'kasalj': ['NE', 'DA'] }
)

T_cpd = TabularCPD(
    variable="temperatura",
    variable_card=2,
    values=[[0.3],[0.7]],
    state_names = { 'temperatura': ['N', 'V'] }
)

X_cpd = TabularCPD(
    variable="X-ray",
    variable_card=2,
    evidence=['kasalj','temperatura'],
    evidence_card=[2,2],
    values=[
        [0.1,0.2,0.3,0.4],
        [0.9,0.8,0.7,0.6],
    ],
)
```

```

state_names = {
    'kasalj': ['NE', 'DA'],
    'temperatura': ['N', 'V'],
    'X-ray': ['LOS', 'DOBAR']
}

C_cpd = TabularCPD(
    variable="covid",
    variable_card=2,
    evidence=['X-ray'],
    evidence_card=[2],
    values=[
        [0.2, 0.7],
        [0.8, 0.3],
    ],
    state_names = {
        'X-ray': ['LOS', 'DOBAR'],
        'covid': ['NE', 'DA']
    }
)

```

U ovom modelu, promenljive **kasalj** i **temperatura** imaju apriorne raspodele, dok promenljiva **X-ray** zavisi od obe. Na kraju, promenljiva **covid** zavisi isključivo od rezultata rendgenskog snimka.

Ovakav model omogućava da, na osnovu simptoma i dijagnostičkih nalaza, procenimo verovatnoću prisustva bolesti, kao i da analiziramo kako različiti faktori utiču na konačnu dijagnozu.

Konstrukcija modela Nakon definisanja uslovnih raspodela verovatnoća (CPD-ova), sledeći korak je konstrukcija same Bejzesovske mreže, odnosno definisanje strukture grafa i povezivanje sa odgovarajućim verovatnoćama.

Struktura modela se zadaje navodjenjem usmerenih grana između promenljivih, čime se eksplicitno definišu njihove zavisnosti. U našem primeru, promenljive **kasalj** i **temperatura** utiču na **X-ray**, dok **X-ray** dalje utiče na promenljivu **covid**.

Listing 4: Konstrukcija Bajesovog modela

```

# Konstrukcija modela pomocu CPD i veza izmedju cvorova
model = BayesianModel([
    ('kasalj', 'X-ray'),
    ('temperatura', 'X-ray'),
    ('X-ray', 'covid')
])

model.add_cpds(K_cpd, T_cpd, X_cpd, C_cpd)

```

Pozivom funkcije **BayesianModel** definišemo strukturu grafa, dok metod **add_cpds** služi za pridruživanje prethodno definisanih raspodela verovatnoća

svakom čvoru u modelu.

Na ovaj način dobijamo kompletno definisan Bajesov model, koji se sastoji iz dva ključna dela: strukture (graf zavisnosti) i parametara (verovatnoće). Takav model je spreman za dalje analize, uključujući izvođenje zaključivanja pomoću algoritama kao što je eliminacija promenljivih.

Provera ispravnosti modela Nakon definisanja strukture mreže i dodavanja svih uslovnih raspodela verovatnoća, potrebno je proveriti da li je model ispravno konstruisan. U biblioteci `pgmpy` to se radi pomoću metode `check_model()`.

Listing 5: Provera ispravnosti modela

```
# Provera ispravnosti modela (probati sa izmenama vrednosti
CPD)
model.check_model()
```

Ova metoda proverava da li su svi čvorovi u modelu dobili odgovarajuće CPD-ove, da li su roditelji navedeni u tabelama verovatnoća uskladjeni sa granama u grafu, kao i da li su same tabele korektno definisane. Posebno je važno da verovatnoće u svakoj koloni uslovne tabele daju zbir jednak 1, jer svaka kolona predstavlja raspodelu verovatnoće za fiksirane vrednosti roditeljskih promenljivih.

Ako je model ispravno definisan, metoda vraća vrednost `True`. U suprotnom, javlja se greška, što može pomoći da se uoče problemi u definiciji modela, na primer pogrešno zadate verovatnoće, neusklađenost između strukture mreže i CPD-ova ili izostavljanje raspodele za neki čvor.

Zbog toga je `check_model()` koristan korak u radu, naročito tokom testiranja i izmene modela, jer omogućava da se greške otkriju pre nego što se pristupi zaključivanju.

Izvršavanje probablističkih upita Kada je model konstruisan i provereno da je ispravan, možemo pristupiti zaključivanju. U biblioteci `pgmpy` za to se koristi klasa `VariableElimination`, koja implementira algoritam eliminacije promenljivih. Ovaj algoritam omogućava izračunavanje verovatnoća promenljivih od interesa na osnovu strukture modela i definisanih uslovnih raspodela.

Listing 6: Izvršavanje probablističkog upita nad modelom

```
# Izvršavanje probablističkih upita
inference = VariableElimination(model)
res = inference.query(['covid'])
print(res)
```

U prvoj liniji formira se objekat `inference`, koji predstavlja mehanizam za zaključivanje nad prethodno definisanim modelom. Drugom linijom zadaje se upit kojim se traži raspodela verovatnoće promenljive `covid`. Pošto u ovom slučaju nije navedena nikakva dodatna evidencija, računa se **marginalna verovatnoća** promenljive `covid`, odnosno njena verovatnoća bez uslovljavanja poznatim vrednostima drugih promenljivih.

Drugim rečima, model sabira doprinose svih mogućih kombinacija vrednosti promenljivih `kasalj`, `temperatura` i `X-ray`, kako bi dobio ukupnu raspodelu za promenljivu `covid`. Rezultat se smešta u promenljivu `res`, a zatim se ispisuje pomoću funkcije `print`.

Dobijeni rezultat predstavlja verovatnoće za obe moguće vrednosti promenljive `covid`, na primer za stanja `NE` i `DA`. Na taj način možemo videti kolika je ukupna verovatnoća prisustva, odnosno odsustva bolesti u okviru zadatog modela.

```
Finding Elimination Order: : 100%|██████████| 3/3 [00:00<00:00, 634.92it/s]
Eliminating: temperatura: 100%|██████████| 3/3 [00:00<00:00, 350.26it/s]
+-----+-----+
| covid | phi(covid) |
+=====+=====+
| covid(NE) | 0.5550 |
+-----+-----+
| covid(DA) | 0.4450 |
+-----+-----+
```

Izvršavanje upita uz zadatu evidenciju Pored izračunavanja marginalnih verovatnoća, Bejzsovska mreža omogućava i zaključivanje u situacijama kada su neke vrednosti promenljivih već poznate. Takve poznate vrednosti nazivaju se **evidencija** ili **opažanja**. Na osnovu njih moguće je izračunati uslovnu verovatnoću promenljive koja nas zanima.

Listing 7: Izvršavanje upita uz zadatu evidenciju

```
# Izvršavanje upita uz opazanje pojedinih vrednosti
inference = VariableElimination(model)
res = inference.query(['covid'], evidence={'kasalj': 'DA', '
    temperatura': 'V'})
print(res)
```

U ovom primeru formira se objekat za zaključivanje nad modelom, a zatim se postavlja upit za promenljivu `covid`, pri čemu je zadato da pacijent ima kašalj i povišenu temperaturu, odnosno da važi `kasalj = DA` i `temperatura = V`. Na taj način ne računa se više ukupna, marginalna verovatnoća promenljive `covid`, već njena **uslovna verovatnoća** u prisustvu konkretnih opaženih simptoma.

Preciznije, izračunava se raspodela:

$$P(\text{covid} \mid \text{kasalj} = \text{DA}, \text{temperatura} = \text{V})$$

To znači da model uzima u obzir samo one delove raspodele koji su u skladu sa zadatim vrednostima promenljivih `kasalj` i `temperatura`, a zatim na osnovu njih procenjuje verovatnoću da promenljiva `covid` ima vrednost `DA` ili `NE`. U tom procesu koristi se struktura mreže i odgovarajuće uslovne raspodele verovatnoća, posebno preko promenljive `X-ray`, koja povezuje simptome sa konačnom dijagnozom.

Ovakvi upiti su posebno značajni u bioinformatičkim i medicinskim primenama, jer omogućavaju procenu verovatnoće određenog stanja na osnovu već poznatih simptoma, nalaza ili eksperimentalnih opažanja. Na taj način Bajesove mreže

predstavljaju koristan alat za modelovanje neizvesnosti i donošenje zaključaka u složenim sistemima.

Primer uslovne nezavisnosti i zatvaranja toka u grafu Jedna od ključnih osobina Bajesovih mreža jeste mogućnost uočavanja **uslovne nezavisnosti** izmedju promenljivih na osnovu strukture grafa. Ova osobina se često objašnjava kroz pojam "zatvaranja toka informacija" u grafu.

Listing 8: Primer uslovne nezavisnosti u modelu

```
# Primer nezavisnosti usled zatvaranja toka zavisnosti u
    grafu
inference = VariableElimination(model)

res1 = inference.query(['covid'], evidence={'X-ray': 'LOS'})
print(res1)

res2 = inference.query(['covid'], evidence={'X-ray': 'LOS',
    'kasalj': 'NE'})
print(res2)
```

U ovom primeru posmatramo kako dodatna informacija o promenljivoj *kasalj* utiče na verovatnoću promenljive *covid*, u situaciji kada već znamo vrednost promenljive *X-ray*.

Kada zadamo evidenciju *X-ray = LOS*, čvor *X-ray* "zatvara" tok informacija izmedju njegovih roditelja (*kasalj*, *temperatura*) i potomka (*covid*). To znači da dodatna informacija o promenljivoj *kasalj* više ne utiče na verovatnoću promenljive *covid*.

Drugim rečima, važi približno:

$$P(\text{covid} \mid \text{X-ray}) = P(\text{covid} \mid \text{X-ray}, \text{kasalj})$$

što znači da su promenljive *covid* i *kasalj* **uslovno nezavisne** kada je poznata vrednost promenljive *X-ray*.

Ovo se može proveriti u praksi poredjenjem rezultata *res1* i *res2*, koji će biti jednaki (ili vrlo bliski), što potvrđuje teorijsko svojstvo modela.

Ovaj fenomen je poznat kao **d-separacija** i predstavlja osnovni mehanizam pomoću kojeg struktura grafa određuje koje promenljive utiču jedna na drugu, a koje postaju nezavisne u prisustvu određene evidencije.

MAP upit (Maximum A Posteriori) Pored računanja kompletne raspodele verovatnoće, često nas zanima samo **najverovatnija vrednost** određene promenljive, uz zadatu evidenciju. U tom slučaju koristi se tzv. **MAP upit** (*Maximum A Posteriori*).

Listing 9: Primer MAP upita

```
# Primer MAP upita
inference = VariableElimination(model)
```

```
res = inference.map_query(['covid'], evidence={'X-ray': 'LOS',
        'kasalj': 'NE'})
print(res)
```

MAP upit ne vraća celu raspodelu verovatnoće, već onu vrednost promenljive koja ima **najveću uslovnu verovatnoću** u odnosu na zadatu evidenciju.

Formalno, MAP upit određuje:

$$\arg \max_x P(x \mid \text{evidence})$$

U ovom primeru, računa se koja vrednost promenljive covid (da li DA ili NE) ima najveću verovatnoću pod uslovom da je X-ray = LOS i kasalj = NE.

Rezultat upita je konkretna vrednost promenljive, na primer:

```
{'covid': 'DA'}
```

što znači da je, prema modelu, najverovatnije da pacijent ima COVID u datim uslovima.

Za razliku od standardnog upita `query()`, koji vraća celu raspodelu verovatnoće, `map_query()` daje samo najverovatniji ishod, što je često korisno u situacijama kada je potrebno doneti konkretnu odluku ili klasifikaciju.

Analiza nezavisnosti u modelu Jedna od važnih prednosti Bajesovih mreža jeste mogućnost automatskog određivanja zavisnosti i nezavisnosti između promenljivih na osnovu strukture grafa. Biblioteka `pgmpy` pruža nekoliko korisnih funkcija za analizu ovih odnosa.

Listing 10: Analiza nezavisnosti i zavisnosti u modelu

```
# Pronadjene nezavisnosti u grafu
model.get_independencies()

# Pronadjene lokalne nezavisnosti vezane za promenljivu "
    kasalj"
model.local_independencies('kasalj')

# Pronadjene aktivne putanje zavisnosti u odnosu na cvor "
    kasalj"
model.active_trail_nodes('kasalj')

# Aktivne putanje kada je X-ray opazen
model.active_trail_nodes('kasalj', observed=['X-ray'])
```

Funkcija `get_independencies()` vraća skup svih nezavisnosti koje važe u modelu, izvedenih na osnovu strukture grafa. Ove nezavisnosti su posledica **d-separacije**, odnosno načina na koji su čvorovi povezani.

Funkcija `local_independencies('kasalj')` daje lokalne nezavisnosti za datu promenljivu. U našem modelu, `kasalj` je direktno povezan samo sa promenljivom `X-ray`, pa je nezavisan od ostalih promenljivih kada se uslovimo na odgovarajuće čvorove (npr. potomke ili roditelje u grafu).

Funkcija `active_trail_nodes('kasalj')` vraća sve čvorove koji su **dostupni putem aktivnih putanja** iz čvora `kasalj`. To su čvorovi na koje `kasalj` može da utiče (ili od kojih može da zavisi) kroz graf, bez zadate evidencije. U našem slučaju, to uključuje `X-ray` i dalje `covid`.

Kada uvedemo evidenciju:

```
model.active_trail_nodes('kasalj', observed=['X-ray'])
```

situacija se menja. Čvor `X-ray` je sada opažen, čime se **zatvara putanja** između `kasalj` i `covid`. To znači da `kasalj` više ne utiče na `covid` kada znamo vrednost promenljive `X-ray`. Drugim rečima, ove dve promenljive postaju **uslovno nezavisne**.

Intuitivno, kada znamo rezultat rendgenskog snimka, dodatna informacija o kašlju više ne donosi novu informaciju o bolesti, jer je uticaj kašlja već "ugrađen" u promenljivu `X-ray`.

Ove funkcije omogućavaju da se struktura modela analizira bez eksplicitnog računanja verovatnoća, što je posebno korisno za razumevanje načina na koji se informacije propagiraju kroz mrežu.

3 Izgradnja Bajesove mreže učenjem strukture podataka

U procesu učenja strukture Bayesovih mreža često se koristi pristup zasnovan na optimizaciji, poznat kao *score-based* metoda. U ovom pristupu, cilj je pronaći strukturu grafa (usmerenog acikličnog grafa, DAG) koja najbolje objašnjava posmatrane podatke. Ovaj proces se sastoji iz dve ključne komponente: algoritma pretrage i funkcije ocenjivanja (engl. *scoring function*). Algoritam pretrage, kao što je *Hill Climb Search*, iterativno modifikuje strukturu grafa dodavanjem, uklanjanjem ili obrnutim usmeravanjem grana, pri čemu u svakom koraku bira promenu koja poboljšava vrednost izabrane funkcije ocenjivanja. Ovaj postupak se ponavlja sve dok se ne dostigne lokalni optimum, odnosno stanje u kojem nijedna lokalna promena ne dovodi do poboljšanja rezultata.

Funkcija ocenjivanja, kao što je BIC (Bayesian Information Criterion), meri kvalitet određene strukture tako što kombinuje dva aspekta: koliko dobro model objašnjava podatke i koliko je model složen. Matematički, BIC se može posmatrati kao razlika između log-verovatnoće modela (koja meri prilagodjenost podacima) i kazne za broj parametara (koja penalizuje kompleksnije modele). Time se postiže balans između preciznosti i jednostavnosti modela, čime se sprečava prekomerno prilagođavanje

U okviru biblioteke *pgmpy*, klasa `HillClimbSearch` implementira ovaj pristup tako što započinje od inicijalne strukture (najčešće bez grana) i zatim sistematski istražuje prostor mogućih grafova. Svaka potencijalna modifikacija strukture se evaluira pomoću odabrane funkcije ocenjivanja, kao što je `BicScore`, a

zatim se bira ona promena koja najviše poboljšava vrednost skora. Konačni rezultat ovog procesa je graf koji predstavlja najbolju aproksimaciju zavisnosti izmedju promenljivih u podacima.

Važno je napomenuti da je Hill Climb Search pohlepni algoritam, što znači da ne garantuje pronalaženje globalno optimalnog rešenja, već lokalno optimalnog. Ipak, zbog svoje efikasnosti i relativno dobre performanse u praksi, ovaj pristup je jedan od najčešće korišćenih metoda za učenje strukture Bayesovih mreža iz podataka.

4 CART algoritam

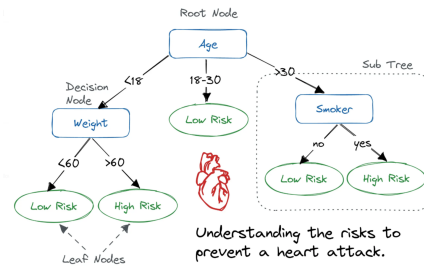
Stablo odlučivanja je drvoliki model koji se koristi za donošenje odluka i predviđanje ishoda na osnovu podataka. Sastoji se od čvorova koji predstavljaju tačke odlučivanja i grana koje predstavljaju moguće ishode tih odluka. Svaki unutrašnji čvor sadrži uslov zasnovan na vrednosti neke ulazne promenljive, dok listovi stabla predstavljaju konačne rezultate, odnosno klasifikacije ili numeričke predikcije.

Stablo odlučivanja se konstruiše rekursivnim deljenjem ulaznog skupa podataka na manje podskupove, u skladu sa vrednostima ulaznih promenljivih. Ovaj proces se ponavlja sve dok se ne dobiju podskupovi koji su dovoljno homogeni ili dok se ne ispuni određeni kriterijum zaustavljanja. Svaka takva podela odgovara jednom čvoru u stablu i predstavlja donošenje odluke na osnovu određenog uslova.

Ključni cilj pri konstruisanju stabla jeste da se izaberu podele koje minimizuju nečistoću (engl. *impurity*) dobijenih podskupova. Nečistoća predstavlja meru nehomogenosti podataka u čvoru, odnosno koliko su različite klase međusobno pomešane. Što je nečistoća manja, to je podskup "čistiji", odnosno sastavljen od elemenata iste klase. Algoritam teži formiranju čvorova sa što manjom nečistoćom, jer takvi čvorovi omogućavaju preciznije donošenje odluka.

Na ovaj način, stablo odlučivanja razlaže složen problem na niz jednostavnih pravila tipa *ako-onda*, gde svaki put od korenskog čvora do lista predstavlja jedno pravilo odlučivanja.

CART (Classification and Regression Trees) algoritam je algoritam zasnovan na stablu odlučivanja koji se može koristiti i za probleme klasifikacije i za probleme regresije u mašinskom učenju. Funkcioniše tako što rekursivno deli skup podataka za treniranje na manje podskupove koristeći binarne podele.



CART je moćan i široko korišćen algoritam zbog svoje sposobnosti da radi sa kategorijskim i numeričkim (kontinualnim) promenljivama, svoje interpretabilnosti, kao i sposobnosti da modeluje nelinearne odnose između ulaznih promenljivih i ciljne promenljive.

Algoritam Classification and Regression Trees (CART) uvek formira binarno stablo, što znači da svaki unutrašnji čvor ima tačno dva potomka. Ovo ga razlikuje od drugih metoda zasnovanih na stablima, koje mogu imati više od dve grane iz jednog čvora.

Algoritam funkcioniše tako što rekurzivno deli skup podataka za treniranje na manje podskupove korišćenjem binarnih podela. Stablo počinje od korenskog čvora, koji sadrži sve podatke za treniranje, i zatim se podaci postepeno dele na manje podskupove sve dok se ne ispuni određeni kriterijum zaustavljanja.

- U svakom čvoru stabla algoritam bira ulaznu promenljivu i odgovarajući prag koji na najbolji način deli podatke na dve grupe, na osnovu vrednosti te promenljive. Ovaj izbor se vrši tako što se pronalazi kombinacija promenljive i praga koja maksimizuje informacijski dobitak ili minimizuje Gini nečistoću, odnosno mere koje procenjuju koliko dobro određena podela razdvaja podatke.
- Proces se nastavlja rekurzivno, tako što svaki čvor deli podatke na dva manja podskupa, sve dok se ne ispuni neki kriterijum zaustavljanja. Ovaj kriterijum može biti maksimalna dubina stabla, minimalan broj instanci u listu ili neki drugi uslov.
- Kada je stablo izgrađeno, ono se koristi za donošenje predikcija tako što se prolazi kroz stablo od korenskog čvora do odgovarajućeg lista za dati ulaz. Kod regresionih problema, predikcija je prosečna vrednost ciljne promenljive u tom listu, dok je kod klasifikacionih problema predikcija najzastupljenija klasa u listu.

Gini nečistoća predstavlja meru nehomogenosti podataka u čvoru i definiše se formulom:

$$Gini = 1 - \sum_{i=1}^K p_i^2 \quad (1)$$

gde je p_i verovatnoća (odnosno udeo) elemenata klase i u datom čvoru, a K ukupan broj klasa. Vrednost Gini nečistoće je jednaka nuli kada svi elementi u čvoru pripadaju istoj klasi, dok veće vrednosti ukazuju na veću nečistoću, odnosno veću izmešanost klasa.

Alternativno, kvalitet podele može se meriti i informacionim dobitkom, koji se zasniva na entropiji:

$$Entropy = - \sum_{i=1}^K p_i \log_2 p_i \quad (2)$$

$$Information\ Gain = Entropy(parent) - \sum_j \frac{N_j}{N} \cdot Entropy(child_j) \quad (3)$$

gde N_j predstavlja broj instanci u podčvoru j , a N ukupan broj instanci u roditeljskom čvoru. Cilj je izabrati podelu koja najviše smanjuje entropiju, odnosno povećava homogenost podskupova.

Entropija je osnovni pojam u teoriji informacija i predstavlja meru neizvesnosti ili količine informacije u sistemu. U kontekstu mašinskog učenja i stabala odlučivanja, entropija se koristi za kvantifikaciju nečistoće (neuredjenosti) podataka u datom čvoru, odnosno za merenje koliko su klase u tom čvoru međusobno pomešane.

Formalno, entropija diskretne slučajne promenljive definiše se kao:

$$H(X) = - \sum_{i=1}^K p_i \log_2 p_i \quad (4)$$

gde p_i predstavlja verovatnoću pojavljivanja klase i , a K ukupan broj klasa. Entropija predstavlja očekivanu količinu informacije potrebnu da se odredi ishod slučajne promenljive.

Vrednost entropije zavisi od raspodele klasa u čvoru. Ako svi elementi pripadaju istoj klasi, entropija je jednaka nuli, što znači da ne postoji neizvesnost. Sa druge strane, entropija dostiže maksimalnu vrednost kada su klase ravnomerno rasporedjene, odnosno kada je neizvesnost najveća.

U algoritmima stabala odlučivanja, entropija se koristi za izbor najbolje podele podataka. Cilj je pronaći takvu podelu koja dovodi do najvećeg smanjenja entropije, čime se dobijaju homogeniji podskupovi. Ova promena entropije naziva se informacijski dobitak (engl. *information gain*) i predstavlja osnovni kriterijum za izgradnju stabla u nekim algoritmima.

Na taj način, entropija omogućava algoritmu da sistematski smanjuje neizvesnost u podacima, razlažući složen problem na niz jednostavnijih i jasnije razdvojenih podskupova.

Da bismo ilustrovali kako se računa Gini nečistoća, razmotrimo sledeći primer. Pretpostavimo da imamo problem binarne klasifikacije, gde želimo da predvidimo da li će osoba kupiti određeni proizvod na osnovu svojih godina i prihoda.

Godine	Prihod	Kupuje proizvod?
30	20000	Da
40	50000	Da
20	30000	Ne
50	60000	Ne
60	80000	Da

Cilj je da izgradimo stablo odlučivanja koje će predvideti da li će osoba kupiti proizvod na osnovu njenih godina i prihoda. Pretpostavimo da želimo da podelimo podatke na osnovu promenljive godine, sa pragom od 35.

Levi potomak (čvor) sadržaće podatke za koje su godine manje ili jednake 35, dok će desni potomak sadržati podatke za koje su godine veće od 35.

Gini nečistoća za levi potomak može se izračunati na sledeći način:

$$G_{\text{left}} = 1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2 = 0.5 \quad (5)$$

U levom čvoru nalaze se dva podatka, od kojih jedan pripada klasi "Da", a drugi klasi "Ne". Stoga je verovatnoća da nasumično izabrani element pripada klasi "Da" jednaka 0.5, dok je verovatnoća da pripada klasi "Ne" takodje 0.5.

Gini nečistoća za desni potomak računa se na sledeći način:

$$G_{\text{right}} = 1 - \left(\frac{2}{3}\right)^2 - \left(\frac{1}{3}\right)^2 \approx 0.444 \quad (6)$$

U desnom čvoru nalaze se tri podatka, od kojih dva pripadaju klasi "Da", a jedan klasi "Ne". Zbog toga je verovatnoća da nasumično izabrani element pripada klasi "Da" jednaka $\frac{2}{3}$, dok je verovatnoća da pripada klasi "Ne" jednaka $\frac{1}{3}$.

Gini nečistoća meri verovatnoću pogrešne klasifikacije nasumično izabranog elementa ako bi mu klasa bila dodeljena na osnovu raspodele klasa u čvoru. Manja vrednost Gini nečistoće ukazuje na homogeniji skup podataka, što znači da je podela kvalitetnija.

Ukupna Gini nečistoća za određenu podelu može se izračunati kao ponderisana sredina Gini nečistoća potomaka, gde su težine proporcionalne broju podataka u svakom čvoru:

$$G_{\text{split}} = \frac{2}{5}G_{\text{left}} + \frac{3}{5}G_{\text{right}} \approx 0.48 \quad (7)$$

- Algoritam stabla odlučivanja ispituje različite pragove i različite ulazne promenljive kako bi pronašao podelu koja minimizuje Gini nečistoću.

Sledeći primer ilustruje primenu CART algoritma na problem binarne klasifikacije.

- Pretpostavimo da imamo skup podataka o pacijentima koji sadrži informacije o njihovim godinama, polu, krvnom pritisku i nivou holesterola, kao i informaciju da li imaju srčano oboljenje. Cilj je da se izgradi model koji predviđa da li će novi pacijent imati srčano oboljenje na osnovu ovih karakteristika.
- Na početku se računa Gini nečistoća celog skupa podataka. Pretpostavimo da u skupu ima 500 pacijenata, od kojih 200 ima srčano oboljenje, a 300 nema. Gini nečistoća se računa kao:

$$G = 1 - \left(\frac{200}{500}\right)^2 - \left(\frac{300}{500}\right)^2 = 0.48 \quad (8)$$

- Zatim se razmatraju sve ulazne promenljive i mogući pragovi za podelu. Pretpostavimo da se kao prva promenljiva bira starost, i da je optimalan prag 50 godina.

Podaci se zatim dele na dva podskupa: pacijente koji imaju 50 ili manje godina i pacijente koji imaju više od 50 godina. Za svaki podskup se formira novi čvor i računa odgovarajuća Gini nečistoća.

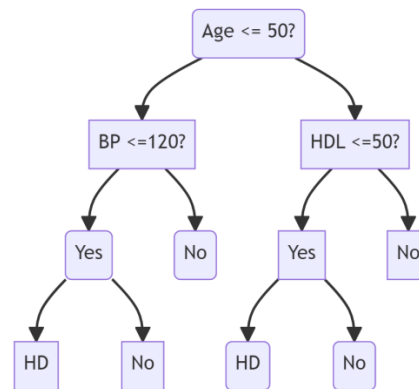
- Pretpostavimo da prvi čvor sadrži 300 pacijenata, od kojih 100 ima srčano oboljenje, a 200 nema. Gini nečistoća ovog čvora iznosi:

$$G_1 = 1 - \left(\frac{100}{300}\right)^2 - \left(\frac{200}{300}\right)^2 = 0.44 \quad (9)$$

- Pretpostavimo da drugi čvor sadrži 200 pacijenata, od kojih 100 ima srčano oboljenje, a 100 nema. Gini nečistoća ovog čvora iznosi:

$$G_2 = 1 - \left(\frac{100}{200}\right)^2 - \left(\frac{100}{200}\right)^2 = 0.5 \quad (10)$$

- Bira se ona podela koja daje najmanju ukupnu Gini nečistoću, odnosno najhomogenije podskupove.
- Proces se nastavlja rekursivno, sve dok se ne ispuni neki kriterijum zaustavljanja, kao što su maksimalna dubina stabla ili minimalan broj instanci u listu.



- Ovo stablo odlučivanja može se koristiti za predviđanje da li će novi pacijent imati srčano oboljenje na osnovu njegovih godina, krvnog pritiska i nivoa holesterola.

Primer koda

- U ovom primeru koriste se prve dve karakteristike iz Iris skupa podataka (dužina i širina čašičnog lista, *sepal length* i *sepal width*) radi jednostavnosti. Formira se klasifikaciono stablo odlučivanja sa maksimalnom dubinom 2 i trenira se nad podacima. Nakon toga se vrše predikcije za nove podatke i prikazuju dobijeni rezultati.

Jedna od značajnih prednosti metoda zasnovanih na stablima, posebno CART algoritma, jeste to što proces donošenja odluka veoma podseća na način na koji ljudi prirodno donose odluke. Zbog toga su rezultati dobijeni ovim modelom lako razumljivi i prihvatljivi korisnicima. Ova intuitivna interpretabilnost predstavlja ključni razlog zašto metode zasnovane na stablima ostaju relevantne u različitim primenama.

Još jedna prednost CART algoritma je njegova sposobnost da obradjuje različite tipove ulaznih podataka, za razliku od mnogih linearnih metoda kao što su logistička regresija ili metode potpornih vektora. CART može raditi sa kombinacijom numeričkih promenljivih (npr. cena ili površina) i kategorijskih promenljivih (npr. tip objekta ili lokacija). Ova fleksibilnost, zajedno sa lakom interpretacijom modela, čini CART veoma moćnim i široko primenljivim algoritmom u različitim zadacima mašinskog učenja.

Prednosti

- Jednostavan i intuitivan algoritam koji je lako razumeti i interpretirati.
- Može da obradjuje i numeričke i kategorijske podatke bez potrebe za dodatnim transformacijama.
- Može da rukuje nedostajućim vrednostima korišćenjem zamenskih podela (engl. *surrogate splits*).
- Može da rešava probleme višeklasne klasifikacije korišćenjem proširenja poznatog kao multi-class CART.

Nedostaci

- Ima tendenciju da preprilagodi (engl. *overfit*) podatke, naročito ako se stablu dozvoli da raste previše duboko.
- Predstavlja pohlepni (engl. *greedy*) algoritam, što znači da ne garantuje pronalaženje globalno optimalnog stabla.
- Može biti pristrasan prema promenljivama sa velikim brojem kategorija ili visokom kardinalnošću.
- Može davati nestabilne rezultate, jer male promene u podacima mogu dovesti do značajno drugačije strukture stabla.

5 HMM i Viterbijev algoritam

U mnogim problemima u računarstvu i statistici susrećemo se sa sekvencijalnim podacima - nizovima koji se razvijaju kroz vreme. Na primer:

- govor (reč po reč),
- tekst (reči u rečenici),
- ponašanje sistema kroz vreme.

Često u tim situacijama: ne možemo direktno da vidimo šta se zaista dešava u sistemu, već samo vidimo neke spoljašnje manifestacije. Tu do izražaja dolazi Hidden Markov Model (HMM) — model koji nam omogućava da opišemo kako se sistem menja kroz vreme i da iz posmatranja zaključimo skrivena stanja. Drugim rečima skriveni Markovljev model je model za situacije gde postoji nevidljiv proces koji generiše vidljive podatke. Pre nego što razumemo HMM, moramo razumeti Markovljevo svojstvo: budućnost zavisi samo od sadašnjosti a ne od prošlosti. Matematički, tu tvrdnju možemo iskazati kao:

$$P(X_{t+1}|X_1X_2...X_t) = P(X_{t+1}|X_t)$$

gde su X_i stanja u kojima se sistem nalazi.

Razmotrimo primer sa psom. Pas može biti srećan, gladan ili umoran. Ali mi ne znamo stanje psa, znamo samo njegovo ponašanje. Pas može da maše repom, laje, spava, traži hranu. Stanje utiče na ponašanje: srećan pas maše repom, umoran spava, a gladan traži hranu. To nije 100% deterministički - i gladan pas može da maše repom. Pas može i da menja stanja - gladan pas posle jela postaje srećan, srećan posle igre se umori, umoran je posle odmora opet gladan, itd. Zamislamo da imamo niz observacija za psa: maše repom → traži hranu → spava → spava. Koje je najverovatnije stanje psa kroz vreme? Odgovor na ovo pitanje daje Viterbijev algoritam ali i neki drugi koje ćemo kasnije pomenuti.



5.1 Viterbijev algoritam na primeru psa

- Skrivena stanja
 - (S) srećan
 - (G) gladan
 - (U) umoran
- Posmatranja
 - (R) maše repom
 - (H) traži hranu
 - (L) laje
 - (Z) spava

- Početne verovatnoće:
 - $P(\text{pas je } S \text{ na početku})=0.5$
 - $P(\text{pas je } G \text{ na početku})=0.3$
 - $P(\text{pas je } U \text{ na početku})=0.2$

- Verovatnoće prelaska iz stanja u stanje:

Tranzicije:				
iz → u	S	G	U	
S	0.6	0.3	0.1	
G	0.4	0.4	0.2	
U	0.3	0.3	0.4	

- Verovatnoće emitovanja određenih signala u određenim stanjima

Emisije:				
stanje	R	H	Z	L
S	0.5	0.2	0.1	0.2
G	0.1	0.6	0.1	0.2
U	0.1	0.1	0.7	0.1

- Posmatrani niz: $R \rightarrow H \rightarrow L \rightarrow Z$
- Tražimo najverovatniji niz stanja u kojima se pas našao

Algoritam se odvija u 4 koraka (broj koraka zavisi od broja observacija u nizu):

- Korak 1:
 - Verovatnoća da je prvo stanje S jednaka je proizvodu dve verovatnoće: verovatnoće da je početno stanje S i verovatnoće da observacija R ako smo u stanju S , odnosno:

$$P(X_1 = S) = P(X_0 = S) * P(O_1 = R | X_0 = S) = 0.5 * 0.5 = 0.25$$

- Slično:

$$P(X_1 = G) = P(X_0 = G) * P(O_1 = R | X_0 = G) = 0.3 * 0.1 = 0.03$$

- i:

$$P(X_1 = U) = P(X_0 = U) * P(O_1 = R | X_0 = R) = 0.2 * 0.1 = 0.02$$

- Korak 2:

- Pod pretpostavkom da idemo u stanje S :

$$P(X_2 = S) = P(X_1 = S) * P(X_2 = S | X_1 = S) * P(O_2 = H | X_2 = S) = 0.25 * 0.6 * 0.2 = 0.03$$

$$P(X_2 = S) = P(X_1 = G) * P(X_2 = S | X_1 = G) * P(O_2 = H | X_2 = S) = 0.03 * 0.4 * 0.2 = 0.0024$$

$$P(X_2 = S) = P(X_1 = U) * P(X_2 = S | X_1 = U) * P(O_2 = H | X_2 = S) = 0.02 * 0.3 * 0.2 = 0.0012$$

Najverovatnija je putanja $S \rightarrow S$ (0.03).

- Pod pretpostavkom da idemo u stanje G :

$$P(X_2 = G) = P(X_1 = S) * P(X_2 = G | X_1 = S) * P(O_2 = H | X_2 = G) = 0.25 * 0.3 * 0.6 = 0.045$$

$$P(X_2 = G) = P(X_1 = G) * P(X_2 = G | X_1 = G) * P(O_2 = H | X_2 = G) = 0.03 * 0.4 * 0.6 = 0.0072$$

$$P(X_2 = G) = P(X_1 = U) * P(X_2 = G | X_1 = U) * P(O_2 = H | X_2 = G) = 0.02 * 0.3 * 0.6 = 0.0036$$

Najverovatnija je putanja $S \rightarrow G$ (0.045).

- Pod pretpostavkom da idemo u stanje U :

$$P(X_2 = U) = P(X_1 = S) * P(X_2 = U | X_1 = S) * P(O_2 = H | X_2 = U) = 0.25 * 0.1 * 0.1 = 0.0025$$

$$P(X_2 = U) = P(X_1 = G) * P(X_2 = U | X_1 = G) * P(O_2 = H | X_2 = U) = 0.03 * 0.2 * 0.1 = 0.0006$$

$$P(X_2 = U) = P(X_1 = U) * P(X_2 = U | X_1 = U) * P(O_2 = H | X_2 = U) = 0.02 * 0.4 * 0.1 = 0.0008$$

Najverovatnija je putanja $S \rightarrow U$ (0.0025).

- Korak 3:

- Pod pretpostavkom da idemo u stanje S :

$$P(X_3 = S) = P(X_2 = S) * P(X_3 = S | X_2 = S) * P(O_3 = L | X_3 = S) = 0.03 * 0.6 * 0.2 = 0.0036$$

$$P(X_3 = S) = P(X_2 = G) * P(X_3 = S | X_2 = G) * P(O_3 = L | X_3 = S) = 0.045 * 0.4 * 0.2 = 0.0036$$

$$P(X_3 = S) = P(X_2 = U) * P(X_3 = S | X_2 = U) * P(O_3 = L | X_3 = S) = 0.0025 * 0.3 * 0.2 = 0.00015$$

Najverovatnije putanje su $S \rightarrow S \rightarrow S$ i $S \rightarrow G \rightarrow S$ (0.0036).

- Pod pretpostavkom da idemo u stanje G :

$$P(X_3 = G) = P(X_2 = S) * P(X_3 = G | X_2 = S) * P(O_3 = L | X_3 = G) = 0.03 * 0.3 * 0.2 = 0.0018$$

$$P(X_3 = G) = P(X_2 = G) * P(X_3 = G | X_2 = G) * P(O_3 = L | X_3 = G) = 0.045 * 0.4 * 0.2 = 0.0036$$

$$P(X_3 = G) = P(X_2 = U) * P(X_3 = G | X_2 = U) * P(O_3 = L | X_3 = G) = 0.0025 * 0.3 * 0.2 = 0.00015$$

Najverovatnija je putanja $S \rightarrow G \rightarrow G$ (0.0036).

– Pod pretpostavkom da idemo u stanje U :

$$P(X_3 = U) = P(X_2 = S) * P(X_3 = U | X_2 = S) * P(O_3 = L | X_3 = U) = 0.03 * 0.1 * 0.1 = 0.0003$$

$$P(X_3 = U) = P(X_2 = G) * P(X_3 = U | X_2 = G) * P(O_3 = L | X_3 = U) = 0.045 * 0.2 * 0.1 = 0.0009$$

$$P(X_3 = U) = P(X_2 = U) * P(X_3 = U | X_2 = U) * P(O_3 = L | X_3 = U) = 0.0025 * 0.4 * 0.1 = 0.0001$$

Najverovatnija je putanja $S \rightarrow G \rightarrow U$ (0.0009).

• Korak 4:

– Pod pretpostavkom da idemo u stanje S :

$$P(X_4 = S) = P(X_3 = S) * P(X_4 = S | X_3 = S) * P(O_4 = Z | X_4 = S) = 0.0036 * 0.6 * 0.1 = 0.000216$$

$$P(X_4 = S) = P(X_3 = G) * P(X_4 = S | X_3 = G) * P(O_4 = Z | X_4 = S) = 0.0036 * 0.4 * 0.1 = 0.000144$$

$$P(X_4 = S) = P(X_3 = U) * P(X_4 = S | X_3 = U) * P(O_4 = Z | X_4 = S) = 0.0009 * 0.3 * 0.1 = 0.000027$$

Najverovatnija je putanja $S \rightarrow G \rightarrow S \rightarrow S$ (0.000216).

– Pod pretpostavkom da idemo u stanje G :

$$P(X_4 = G) = P(X_3 = S) * P(X_4 = G | X_3 = S) * P(O_4 = Z | X_4 = G) = 0.0036 * 0.3 * 0.1 = 0.000108$$

$$P(X_4 = G) = P(X_3 = G) * P(X_4 = G | X_3 = G) * P(O_4 = Z | X_4 = G) = 0.0036 * 0.4 * 0.1 = 0.000144$$

$$P(X_4 = G) = P(X_3 = U) * P(X_4 = G | X_3 = U) * P(O_4 = Z | X_4 = G) = 0.0009 * 0.3 * 0.1 = 0.000027$$

Najverovatnija je putanja $S \rightarrow G \rightarrow G \rightarrow G$ (0.000144).

– Pod pretpostavkom da idemo u stanje U :

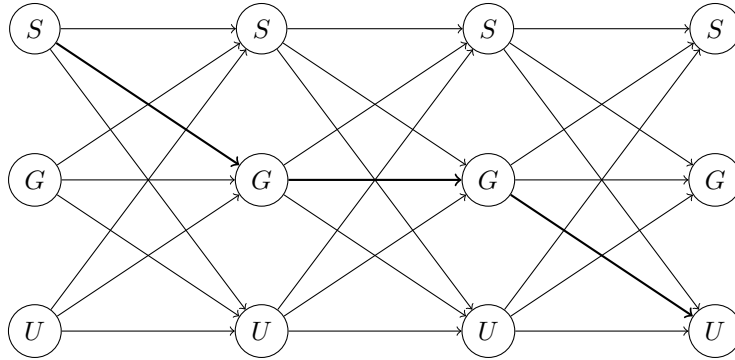
$$P(X_4 = U) = P(X_3 = S) * P(X_4 = U | X_3 = S) * P(O_4 = Z | X_4 = U) = 0.0036 * 0.1 * 0.7 = 0.000252$$

$$P(X_4 = U) = P(X_3 = G) * P(X_4 = U | X_3 = G) * P(O_4 = Z | X_4 = U) = 0.0036 * 0.2 * 0.7 = 0.000504$$

$$P(X_4 = U) = P(X_3 = U) * P(X_4 = U | X_3 = U) * P(O_4 = Z | X_4 = U) = 0.0009 * 0.4 * 0.7 = 0.000252$$

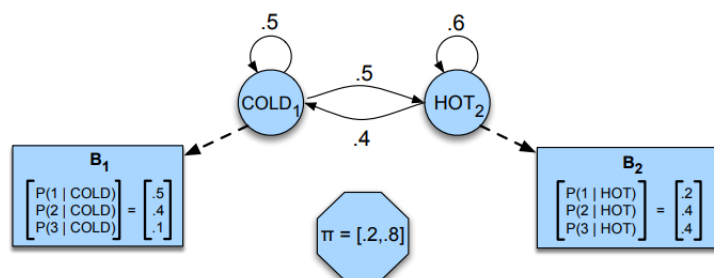
Najverovatnija je putanja $S \rightarrow G \rightarrow G \rightarrow U$ (0.000504).

Najverovatniji niz stanja je: $S \rightarrow G \rightarrow G \rightarrow U$.



6 HMM i Forward algoritam

Započnimo ovo poglavlje primerom koji je dao Džejson Ajsner 2002. godine, tačnije njegovom pojednostavljenom verzijom. Pretpostavimo da postoje dva tipa dana topli (H-HOT) i hladni (C-COLD). Zadatak je sledeći: Za dati niz observacija O (pri čemu je svaka observacija broj sladoleda koje je Džejson pojeo toga dana), pronaći skrivenu sekvencu tipa dana (hladni ili topli) koja je najverovatnija za dati niz observacija. Ovaj zadatak rešavali smo u prethodnom poglavlju Viterbijevim algoritmom. Dovoljno je bilo da imamo ulazne podatke u obliku kao na primeru:



Na slici su prikazane u okviru pravougaonika B_1 verovatnoće da Džejson pojede jedan sladoled na hladan dan, dva i tri. Slično imamo verovatnoće prelaska - da hladan dan postane topao 0.5, da ostane hladan 0.5, da topao postane hladan 0.4, da ostane topao 0.6, kao i verovatnoće da je inicijalno hladan 0.2 i topao 0.8 (očigledno je Džejson živeo u nekim toplijim krajevima kad je smislio ovaj primer).

Sada nas zanima da rešimo jedan malo drugačiji zadatak. Možemo takodje smatrati da nam je data ova ulazna slika i pitamo se: Kolika je verovatnoća niza observacija 3 1 3? Kolika je verovatnoća da Džejson pojede 3 sladoleda pa 1 pa opet 3?

Krenućemo sa nešto jednostavnijom situacijom. Da li bismo znali da izračunamo ovu verovatnoću ako bismo znali kom nizu skrivenih stanja odgovara? Za svaki unapred zadati niz skrivenih stanja (npr. HOT HOT COLD), lako je izračunati verovatnoću niza observacija 3 1 3. Primetimo da ih ima isto - jer važan uslov HMM jeste da jedno stanje proizvodi jednu observaciju i jedna observacija odgovara jednom stanju. Druga važna pretpostavka HMM jeste da ništa drugo nije uticalo na tu observaciju sem pomenutog stanja:

$$P(o_i | q_1, q_2, \dots, q_T, o_1, o_2, \dots, o_T) = P(o_i | q_i)$$

Ako sada imamo poznat niz observacija $O = o_1 o_2 \dots o_T$ i poznat niz skrivenih stanja $Q = q_1 q_2 \dots q_T$, tada je:

$$P(O|Q) = \prod_{i=1}^T P(o_i|q_i)$$

Na našem primeru bi bilo:

$$P(3 \ 1 \ 3 | HOT \ HOT \ COLD) = P(3 | HOT) \times P(1 | HOT) \times P(3 | COLD)$$

Jedini problem ovde je što mi ne znamo skrivenu sekvencu. Umesto ovog, moraćemo da sumiramo verovatnoće od 3 1 3 za sve moguće skrivene sekvence pomnožene njihovim verovatnoćama. Računamo zajedničku raspodelu:

$$P(O, Q) = P(O|Q) \times P(Q) = \prod_{i=1}^T P(o_i|q_i) * \prod_{i=1}^T P(q_i|q_{i-1})$$

Na našem primeru:

$$P(3 \ 1 \ 3, HOT \ HOT \ COLD) = P(HOT | START) \times P(HOT | HOT) \times P(COLD | HOT) \\ \times P(3 | HOT) \times P(1 | HOT) \times P(3 | COLD)$$

Sada kada znamo da izračunamo zajedničku raspodelu,

$$P(O) = \sum_Q P(O, Q)$$

suma po svim zajedničkim raspodelama. Za *skriveni Markovljev model* (*Hidden Markov Model*, *HMM*) sa N skrivenih stanja i sekvencom od T opservacija, postoji N^T mogućih skrivenih sekvenci. U realnim zadacima, gde su i N i T veliki, N^T je izuzetno veliki broj, pa nije praktično izračunati ukupnu verovatnoću posmatrane sekvence tako što bi se za svaku skrivenu sekvencu posebno računala verovatnoća, a zatim sve te vrednosti sabrale.

Umesto takvog eksponencijalnog pristupa, koristi se efikasan algoritam složenosti $O(N^2T)$ koji se naziva *forward algoritam*. Forward algoritam je primer *dinamičkog programiranja*, odnosno algoritma koji koristi tabelu za čuvanje međurezultata dok postepeno gradi verovatnoću posmatrane sekvence.

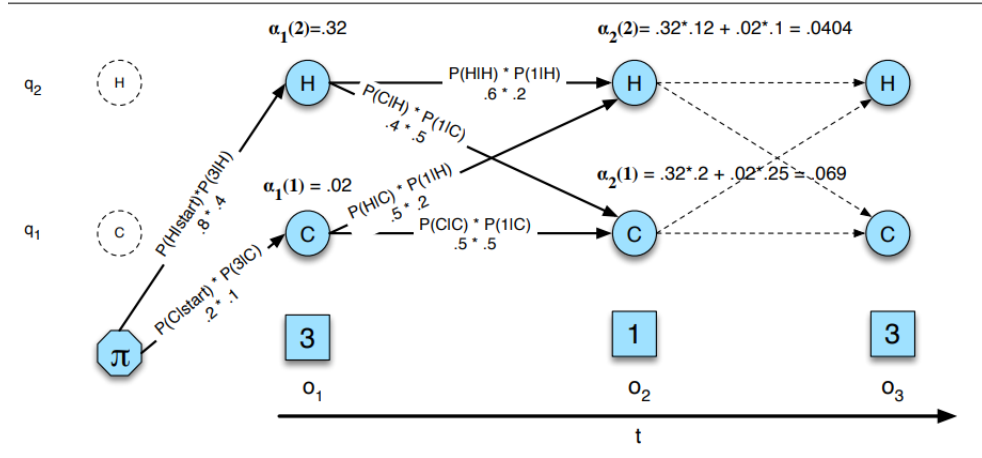
Forward algoritam računa verovatnoću opservacija sabiranjem verovatnoća svih mogućih putanja skrivenih stanja koje mogu generisati datu sekvencu opservacija, ali to čini efikasno tako što implicitno objedinjuje sve te putanje u jednu *forward rešetku* (*trellis*). Pogledajmo na slici: Svaka ćelija $\alpha_t(j)$ predstavlja verovatnoću da budemo u stanju j nakon što smo videli prvih t opservacija. Tu vrednost dobijamo tako što sumiramo verovatnoće svih puteva koji su tu mogli da nas dovedu. Formalno zapisano:

$$\alpha_t(j) = P(o_1 o_2 \dots o_t, q_t = j | \lambda)$$

gde je λ predstavlja ulazne informacije - verovatnoće prelaska, uslovne verovatnoće, itd. Precizno,

$$\alpha_t(j) = \sum_{i=1}^N \alpha_{t-1}(i) a_{ij} b_j(o_t)$$

pri čemu je:



- $\alpha_{t-1}(i)$ verovatnoća da je prethodno $t - 1$ stanje bilo i
- a_{ij} verovatnoća prelaska iz stanja i u stanje j
- verovatnoća observacije t u stanju j

Posmatrajmo izračunavanje $\alpha_2(2)$ na slici, odnosno forward verovatnoće da se u vremenskom koraku 2 nalazimo u stanju 2, pri čemu je generisana parcijalna sekvenca opservacija 3, 1.

Ovu verovatnoću računamo proširivanjem α verovatnoća iz vremenskog koraka 1, putem dve moguće putanje. Svako proširenje sastoji se od tri faktora opisana ranije:

$$\alpha_1(1) \cdot P(HOT | COLD) \cdot P(1 | HOT)$$

i

$$\alpha_1(2) \cdot P(HOT | HOT) \cdot P(1 | HOT).$$

Dakle, $\alpha_2(2)$ se dobija kao zbir doprinosa ovih putanja.

7 CPG ostrva

Da bismo razumeli šta su CpG ostrva, potrebno je da ukratko objasnimo nekoliko osnovnih pojmova iz biologije.

DNK (dezoksiribonukleinska kiselina) je molekul koji nosi genetsku informaciju i sastoji se od niza “slova”, odnosno nukleotida: A, C, G i T. Određeni delovi DNK predstavljaju gene — segmente koji sadrže uputstva za pravljenje proteina.

Pre nego što se gen “aktivira” (odnosno počne da se koristi za stvaranje proteina), postoji deo DNK koji reguliše taj proces. Taj deo nazivamo *promotorski region*. Možemo ga zamisliti kao “prekidač” koji kontroliše da li će se gen uključiti ili ne.

Sada dolazimo do procesa metilacije. Metilacija je hemijska promena DNK pri kojoj se na određene nukleotide (najčešće na C u kombinaciji CG) dodaje mala hemijska grupa. Ova promena često utiče na to da se gen “isključí”, odnosno da se ne koristi.

Tokom vremena, metilacija može dovesti i do promena u DNK sekvenci (mutacija), zbog čega su kombinacije CG u većem delu genoma relativno retke.

Medjutim, postoje regioni DNK gde se motiv CG pojavljuje znatno češće — to su *CpG ostrva*. Ovi regioni su obično manje podložni metilaciji i često se nalaze blizu promotorskih regiona gena, što znači da su povezani sa aktivnim genima.

Zbog toga je zadatak pronalazjenja CpG ostrva važan: ona nam mogu ukazati na delove DNK koji imaju značajnu biološku funkciju. U ovom primeru koristićemo Python biblioteku `hmmlearn`, koja omogućava jednostavan rad sa skrivenim Markovljevim modelima (HMM). Ova biblioteka već implementira osnovne algoritme kao što su forward, backward i Viterbijev algoritam, pa nam omogućava da se fokusiramo na razumevanje modela i interpretaciju rezultata, a ne na samu implementaciju od nule.

Takodje ćemo koristiti biblioteku `numpy`, koja je standard za rad sa numeričkim podacima u Pythonu i omogućava efikasnu manipulaciju nizovima i matricama, što je posebno važno kod probablističkih modela.

```
from hmmlearn import hmm
import numpy as np
```

Sada konstruišemo skriveni Markovljev model koji će imati dva skrivena stanja. U ovom primeru ta stanja možemo interpretirati kao regione koji pripadaju CpG ostrvima i regione koji ne pripadaju CpG ostrvima.

Pošto su opservacije diskretne, odnosno predstavljaju nukleotide iz DNK sekvence, koristimo klasu `MultinomialHMM`.

```
# Konstrukcija modela za pronalazenje CpG ostrva
model = hmm.MultinomialHMM(n_components=2)
```

Klasa `MultinomialHMM` predstavlja skriveni Markovljev model kod koga su opservacije diskretne, odnosno dolaze iz konačnog skupa mogućih vrednosti. Drugim rečima, model pretpostavlja da svaka opservacija pripada jednoj od unapred definisanih kategorija.

U kontekstu bioinformatike, ove kategorije mogu biti, na primer, nukleotidi DNK sekvence: A, C, G, T. Umesto da radimo sa realnim vrednostima, svaku od ovih kategorija kodiramo kao ceo broj (npr. A=0, C=1, G=2, T=3), a model zatim uči verovatnoće pojavljivanja svake od ovih vrednosti u svakom skrivenom stanju.

Za razliku od, na primer, `GaussianHMM`, koji modeluje kontinualne (realne) opservacije pomoću normalne raspodele, `MultinomialHMM` je namenjen upravo ovakvim diskretnim podacima.

U nastavku ručno zadajemo parametre modela. Oznakom `-` predstavljamo regione koji nisu CpG ostrva, dok oznakom `+` predstavljamo regione koji jesu CpG ostrva. Nukleotidi A, C, G i T biće naše opservacije.

Početne verovatnoće su jednake, što znači da model na početku nema prednost ni za jedno stanje. Matrica prelaza kaže da je velika verovatnoća da model ostane u istom stanju, dok je prelazak iz jednog stanja u drugo redji. Na taj način model opisuje ideju da se CpG ostrva i regioni van njih javljaju kao duži povezani segmenti, a ne kao potpuno nasumična pojedinačna slova.

Matrica emisija opisuje verovatnoće pojavljivanja nukleotida u svakom stanju. U stanju $-$ češće se pojavljuju A i T, dok se u stanju $+$ češće pojavljuju C i G. To odgovara ideji da su CpG ostrva bogatija nukleotidima C i G.

Na kraju, DNK sekvencu zapisujemo kao string, a zatim svako slovo pretvaramo u odgovarajući broj, jer model ne radi direktno sa slovima već sa numerički kodiranim opservacijama.

```
# 0: -, 1: +
states = ['- ', '+ ']
observations = ['A', 'C', 'G', 'T']

#                                -      +
model.startprob_ = np.array([0.5, 0.5])

model.transmat_ = np.array([
    # --  -+
    [0.9, 0.1],

    # +-  ++
    [0.1, 0.9]
])

model.emissionprob_ = np.array([
    # A      C      G      T
    [0.37, 0.13, 0.13, 0.37],
    [0.13, 0.37, 0.37, 0.13],
])

X = '
    ATATCATTGCGCGTTTCGCGAGCTATTTTATATATCTAGAGTAGCAGAGGCGCGCATATAGCTACTATCGT
'

X_encoded = []

for x in X:
    X_encoded.append(observations.index(x))

X_encoded = np.array([X_encoded]).T
```

Nakon što je DNK sekvencu kodirana brojevima, možemo primeniti model kako bismo odredili najverovatniji niz skrivenih stanja. To znači da za svaki nukleotid iz sekvence želimo da procenimo da li pripada CpG ostrvu, označenom sa $+$, ili regionu van CpG ostrva, označenom sa $-$.

Metoda `decode` vraća najverovatniju sekvencu skrivenih stanja. Iz rezultata uzimamo drugi element, jer on sadrži upravo niz stanja. Dobijene vrednosti su

brojevi 0 i 1, pa ih zatim prevodimo nazad u oznake - i + pomoću liste `states`.

```
# Kodiranje emisija u brojeve
encoded_y = model.decode(X_encoded)[1]
y_pred = ''.join(list(map(lambda y: states[y], encoded_y)))
```

Sada pravimo novi HMM model, ali ovoga puta nećemo ručno zadavati sve verovatnoće. Umesto toga, pustićemo model da ih nauči iz datih DNK sekvenci. Zato koristimo metodu `fit`.

Parametar `algorithm='viterbi'` znači da će se pri dekodiranju koristiti Viterbijev algoritam, tj. algoritam koji pronalazi najverovatniju sekvencu skrivenih stanja.

```
new_model = hmm.MultinomialHMM(n_components=2, algorithm='
viterbi')
```

Zatim zadajemo nekoliko DNK sekvenci koje će služiti kao skup podataka za obučavanje modela. Pošto biblioteka `hmmlearn` očekuje numeričke vrednosti, svaku sekvencu moramo kodirati na isti način kao ranije:

$$A \rightarrow 0, \quad C \rightarrow 1, \quad G \rightarrow 2, \quad T \rightarrow 3.$$

```
X = ['ATTATATTATATTATATTTAA',
      'CCTAGTCGTGTCGCAAAAAAACTGCTCTGACCTGAGC',
      'ATTATATGCCGCGGC',
      'GGGCCCCGGCTTATATAT']
```

```
X_encoded = None
init = False
```

```
for x in X:
    new_encoded = []
    for c in x:
        new_encoded.append(observations.index(c))

    if not init:
        X_encoded = np.array([new_encoded]).T
        init = True
    else:
        X_encoded = np.concatenate([X_encoded,
                                     np.array([new_encoded]).
                                     T])
```

```
X_encoded
```

Ovde je važno primetiti da se sve sekvence spajaju u jedan veliki niz opservacija. Međutim, model mora da zna gde se jedna sekvenca završava, a gde počinje sledeća. Zato se metodi `fit` kao drugi argument prosledjuje lista dužina pojedinačnih sekvenci:

```
new_model.fit(X_encoded, [len(x) for x in X])
```

Lista

```
[len(x) for x in X]
```

sadrži dužine svih sekvenci iz liste `X`. Na taj način model zna da podaci ne predstavljaju jednu neprekidnu DNK sekvencu, već više odvojenih primera.

Nakon obučavanja modela, možemo ga primeniti na novu DNK sekvencu. Postupak je isti kao ranije: prvo sekvencu kodiramo brojevima, a zatim pomoću metode `decode` određujemo najverovatniji niz skrivenih stanja.

```
X = '
    ATATCATTGCGCGTTTCGCGAGCTATTTTATATATCTAGAGTAGCAGAGGCGCGCATATAGCTACTATCGT
'

X_encoded = []

for x in X:
    X_encoded.append(observations.index(x))

X_encoded = np.array([X_encoded]).T

# new_model.decode(X_encoded)

encoded_y = new_model.decode(X_encoded)[1]
y_pred = ','.join(list(map(lambda y: states[y], encoded_y)))

Promenljiva encoded_y sadrži niz brojeva 0 i 1, gde svaki broj predstavlja
jedno skriveno stanje. Zatim se ti brojevi prevode u oznake - i +, pa promenljiva
y_pred predstavlja procenjenu strukturu sekvence: koji delovi najverovatnije
pripadaju CpG ostrvima, a koji ne. Na kraju ispisujemo originalnu DNK
sekvencu i predvidjeni niz skrivenih stanja. Tako možemo vizuelno uporediti
gde model procenjuje da se nalaze CpG ostrva.

Važno je napomenuti da algoritam za učenje parametara HMM modela ne
garantuje globalno optimalno rešenje. On često pronalazi samo lokalno opti-
malno rešenje, pa se može desiti da dobijeni parametri, naročito u ovako jed-
nostavnom primeru, značajno odstupaju od očekivanih. Zbog toga je obuku
modela često korisno pokrenuti više puta.

'''
Napomena, algoritam za učenje parametara HMM modela
pronalaži lokalno optimalno rešenje te je moguće da
pronađeni parametri (posebno u ovako jednostavnom slučaju)
značajno odstupaju od očekivanih, te je potrebno obuku
pokrenuti više puta
'''

print(X)
print(y_pred)
```

8 Uslovna slučajna polja - Conditional Random Fields (CRF)

Vratimo se na problem CpG ostrva. Kod CpG ostrva imamo DNK sekvencu, na primer:

```
A C G C G T A C G G C A
```

i želimo svakoj poziciji da dodelimo oznaku:

```
A C G C G T A C G G C A
0 1 1 1 1 0 0 1 1 1 0
```

gde je, recimo:

```
I = pozicija je unutar CpG ostrva
0 = pozicija je van CpG ostrva
```

(kao oznake možemo koristiti $i + i -$, ili 1 i 0, itd.) Dakle, ne klasifikujemo celu sekvencu odjednom, nego radimo označavanje svake pozicije u sekvenci. To se zove sequence labeling, odnosno označavanje sekvence. Naivno bismo mogli da kažemo: Za svaku poziciju pogledaj nukleotid i odluči da li je I ili O. Na primer:

```
ako je C ili G -> mozda I
ako je A ili T -> mozda 0
```

Ali to nije dovoljno, zato što oznake nisu nezavisne. Ako je jedna pozicija unutar CpG ostrva, velika je verovatnoća da su i susedne pozicije takodje unutar CpG ostrva. Ima smisla *OOOIIIIIOO* ali nema smisla *OIOIOIOIOI*. Dakle, model ne samo da treba da nauči vezu izmedju nukleotida i oznake, već i vezu izmedju prethodne i trenutne oznake. Zato koristimo modele kao što su HMM i CRF.

Kod HMM-a imamo skrivene oznake/stanja: O, I i posmatrane simbole A, C, G, T . HMM pretpostavlja da skrivena stanja generišu opažanja, dakle da je stanje to koje je uticalo na nukleotid. Na primer:

```
ako sam u stanju I, veka je verovatnoca da emitujem C ili G
ako sam u stanju 0, veka je verovatnoca da emitujem A ili T
```

HMM ima:

```
pocetne verovatnoce stanja
prelazne verovatnoce izmedju stanja
emisione verovatnoce
```

i modeluje zajedničku verovatnoću: $P(x, y)$ gde je:

```
x = posmatrana sekvenca, npr. ACGCGT
y = niz skrivenih oznaka, npr. 0IIII0
```

Znači HMM pokušava da objasni kako je sekvenca nastala. CRF, odnosno Conditional Random Field, razmišlja drugačije. On ne pokušava da modeluje kako je sekvenca nastala. On direktno pita:

Ako mi je data ova sekvenca x , koji niz oznaka y je najverovatniji?

Zapravo traži verovatnoću $P(y|x)$, odnosno verovatnoću oznaka ako već znamo sekvenču. To je prirodnije jer nas ne zanima kako je sekvenca generisana. Ovo je važna razlika. HMM je generativni model. To znači da on opisuje kako se generišu i oznake i opažanja. Kod HMM-a imamo priču:

```
prvo se izabere skriveno stanje
zatim to stanje emituje simbol
zatim se predje u novo stanje
zatim novo stanje emituje novi simbol
...
```

Dakle HMM kao da pravi sekvenču. CRF je diskriminativni model. On ne pokušava da generiše sekvenču. On samo uči granicu između mogućih oznaka. Kod CRF-a imamo priču:

```
sekvenca je vec data
sada na osnovu njenih osobina odluci koje su oznake najbolje
```

Zato je CRF bliži klasifikatoru, ali klasifikatoru koji zna da radi sa sekvencama. Zamislimo da imamo rečenicu:

Marko zivi u Beogradu

i želimo da označimo reči:

```
Marko      PERSON
zivi       0
u          0
Beogradu   LOCATION
```

HMM bi razmišljao:

Ako je stanje PERSON, kolika je verovatnoća da emituje reč "Marko"?
Ako je stanje LOCATION, kolika je verovatnoća da emituje reč "Beogradu"?

CRF bi razmišljao:

Reč počinje velikim slovom.

Prethodna reč je "u".

Reč se završava na "-u".

Možda je to lokacija.

Kod HMM-a uglavnom imamo emisione verovatnoće tipa:

```
P(A | I)
P(C | I)
P(G | I)
P(T | I)
```

Kod CRF-a ne moramo da se ograničimo samo na trenutni simbol. Za svaku poziciju možemo da napravimo opis. Na primer, za sekvenču: *ACGCGT* za poziciju $i = 2$, gde je nukleotid G, možemo gledati:

trenutni nukleotid je G
prethodni nukleotid je C
sledeći nukleotid je C
prethodni + trenutni par je CG
trenutni + sledeći par je GC
da li je trenutni nukleotid C ili G
koliko C i G ima u okolini
da li smo na početku sekvence
da li smo na kraju sekvence

To su osobine, odnosno features. CRF onda uči koje osobine su važne za koju oznaku. Na primer, može naučiti:

ako je trenutni nukleotid C ili G, veća je šansa za I
ako je par CG, veća je šansa za I
ako je u okolini mnogo C i G, veća je šansa za I
ako je prethodna oznaka I, veća je šansa da i trenutna bude I

CRF uči težine feature-a. Na primer, imamo feature:

trenutni nukleotid je G

Model može naučiti da je taj feature pozitivan za oznaku I. To znači:

ako je trenutni nukleotid G, povećaj skor oznake I

Drugi feature može biti:

prethodna oznaka je I, trenutna oznaka je I

Model može naučiti da je to veoma dobar prelaz. To znači:

ako smo već u CpG ostrvu, verovatno ostajemo u CpG ostrvu

A feature:

prethodna oznaka je I, trenutna oznaka je 0

može imati manju težinu, jer izlazak iz ostrva nije toliko čest kao ostajanje u njemu.

8.1 Kod

Feature **bias** predstavlja konstantnu osobinu koja je prisutna na svakoj poziciji sekvence. Njegova vrednost je uvek jednaka 1, a njegova uloga je da modelu omogući učenje osnovne sklonosti ka određenoj oznaci, nezavisno od konkretnog nukleotida na posmatranoj poziciji. Na taj način CRF može da nauči, na primer, da je oznaka van CpG ostrva češća od oznake unutar CpG ostrva, čak i pre nego što uzme u obzir lokalni kontekst sekvence.

bias kaže: šta model misli pre nego što pogleda konkretne osobine?

nuc kaže: kako trenutni nukleotid menja tu odluku?

-1:nuc kaže: kako prethodni nukleotid menja tu odluku?

Još jedna analogija: zamislite da profesor ocenjuje studenta.

Bez informacija, profesor možda zna da većina studenata položi. To je osnovna pretpostavka.

bias: većina studenata položi

Onda dolaze dodatne informacije:

student je radio domaće zadatke	→ povećava šansu da položi
student nije dolazio na vežbe	→ smanjuje šansu da položi
student je dobro uradio kolokvijum	→ povećava šansu da položi

Dakle, bias nije konkretan dokaz. To je početna sklonost.

8.1.1 Konstrukcija feature-a za CRF model

Da bi se DNK sekvenca mogla koristiti kao ulaz za CRF model, potrebno je pretvoriti je u oblik koji model može da obradi. CRF model ne prima direktno string, na primer "ACGT", već očekuje da svaka pozicija u sekvenci bude opisana skupom osobina, odnosno feature-a. Zbog toga se za svaku poziciju u sekvenci konstruiše jedan rečnik koji sadrži osobine te pozicije.

```
# Konstrukcija feature-a ulaznog parametra X
def create_features(X):
    features = []
    for i in range(len(X)):
        x = X[i]

        current_features = {
            'bias': 1,
            'nuc': x.lower()
        }

        if i > 0:
            current_features.update({
                '-1:nuc': X[i - 1].lower()
            })

        features.append(current_features)

    return features
```

U ovoj funkciji promenljiva `X` predstavlja jednu ulaznu sekvencu, na primer DNK sekvencu. Petlja prolazi kroz sve pozicije sekvence. Promenljiva `i` predstavlja indeks trenutne pozicije, dok promenljiva `x` predstavlja nukleotid koji se nalazi na toj poziciji.

Za svaku poziciju formira se rečnik `current_features`. Taj rečnik opisuje trenutnu poziciju pomoću izabranih feature-a. U datom kodu osnovni feature-i su:

```
bias    i    nuc.
```

Feature `nuc` čuva informaciju o trenutnom nukleotidu. Na primer, ako je trenutni nukleotid `A`, tada će odgovarajući deo rečnika biti:

```
'nuc': 'a'
```

Poziv `x.lower()` koristi se da bi se sva slova zapisala malim slovima. Na taj način se izbegava razlikovanje između, na primer, `A` i `a`. Dakle, feature `nuc` modelu govori koji se nukleotid nalazi na posmatranoj poziciji.

Feature `bias` je drugačiji. On ne opisuje konkretan nukleotid, već predstavlja konstantnu osobinu koja je prisutna na svakoj poziciji sekvence. Njegova vrednost je uvek jednaka 1:

```
'bias': 1
```

Iako na prvi pogled deluje da feature koji je uvek jednak 1 ne nosi nikakvu informaciju, njegova uloga je veoma važna. On omogućava CRF modelu da nauči osnovnu sklonost ka određenim oznakama, nezavisno od konkretnog nukleotida. Na primer, u problemu pronalaženja CpG ostrva može se desiti da je oznaka koja predstavlja poziciju van CpG ostrva mnogo češća od oznake koja predstavlja poziciju unutar CpG ostrva. Feature `bias` omogućava modelu da nauči takvu početnu sklonost.

Drugim rečima, `bias` se može posmatrati kao slobodni član u linearnom modelu. Kao što se u izrazu

$$y = ax + b$$

član `b` koristi da pomeri osnovnu vrednost funkcije, tako se i feature `bias` koristi da model ima osnovnu procenu pre nego što uzme u obzir konkretne osobine sekvence. Ostali feature-i, kao što je `nuc`, zatim menjaju tu početnu procenu.

Važno je naglasiti da nazivi `bias`, `nuc` i `-1:nuc` nisu unapred definisani ni u Pythonu ni u CRF biblioteci. To su nazivi koje sami biramo prilikom konstrukcije feature-a. Na primer, umesto:

```
current_features = {
    'bias': 1,
    'nuc': x.lower()
}
```

moglo bi se napisati i:

```
current_features = {
    'konstanta': 1,
    'trenutni_nukleotid': x.lower()
}
```

Takav zapis bi takodje bio ispravan, jer su ključevi u rečniku samo imena feature-a. Ipak, nazivi kao što su **bias** i **nuc** često se koriste zato što su kratki i jasno ukazuju na značenje feature-a.

Ukoliko trenutna pozicija nije prva u sekvenci, dodaje se još jedan feature:

```
'-1:nuc': X[i - 1].lower()
```

Ovaj feature sadrži informaciju o prethodnom nukleotidu. Oznaka **-1** u nazivu feature-a označava poziciju jedan korak ulevo od trenutne pozicije, odnosno poziciju $i - 1$. Na primer, ako je data sekvenca

$X = \text{"ACG"}$,

onda funkcija `create_features` vraća listu:

```
[
    {'bias': 1, 'nuc': 'a'},
    {'bias': 1, 'nuc': 'c', '-1:nuc': 'a'},
    {'bias': 1, 'nuc': 'g', '-1:nuc': 'c'}
]
```

Prvi element liste opisuje prvu poziciju u sekvenci, drugi element opisuje drugu poziciju, a treći element opisuje treću poziciju. Prva pozicija nema feature **-1:nuc**, jer pre nje ne postoji prethodni nukleotid.

CRF model zatim koristi ovako konstruisane feature-e tokom treniranja. Za svaki feature model uči odgovarajuće težine. Na primer, model može naučiti da feature

```
'nuc': 'c'
```

povećava verovatnoću oznake koja označava unutrašnjost CpG ostrva, dok feature

```
'nuc': 'a'
```

može povećavati verovatnoću oznake koja označava poziciju van CpG ostrva. Slično tome, model može naučiti i težinu za feature **bias**, čime uči koja oznaka je generalno verovatnija, nezavisno od konkretnog nukleotida.

Dakle, funkcija `create_features` ne vrši klasifikaciju i ne odlučuje kojoj klasi pripada svaka pozicija. Njena uloga je samo da svaku poziciju sekvence opiše pomoću izabranih osobina. Na osnovu tih osobina i poznatih oznaka u trening skupu, CRF model kasnije uči težine feature-a i koristi ih za predviđanje oznaka na novim sekvencama.

8.1.2 Inicijalizacija i treniranje CRF modela

Nakon što su ulazne sekvence transformisane u nizove feature-a, može se konstruisati i trenirati CRF model. U ovom primeru koristi se biblioteka **sklearn-crfsuite**, koja omogućava rad sa uslovnim slučajnim poljima namenjenim označavanju sekvenci.

```
# Inicijalizacija CRF algoritma i treniranje modela
crf = CRF(
    algorithm='lbfgs',
    c1=0.1,
    c2=0.1,
    max_iterations=100,
    all_possible_transitions=True
)

try:
    crf.fit(X_train, y_train)
except AttributeError:
    pass
```

Ovim kodom se pravi objekat `crf`, odnosno jedan CRF model. Važno je naglasiti da se u liniji

```
crf = CRF(...)
```

model još uvek ne trenira. U toj liniji se samo definišu podešavanja modela: koji algoritam će se koristiti za optimizaciju, kakva regularizacija će se primeniti, koliko iteracija je dozvoljeno i da li će se razmatrati svi mogući prelazi između oznaka.

Samo treniranje počinje tek pozivom metode:

```
crf.fit(X_train, y_train)
```

Ovde `X_train` predstavlja skup ulaznih sekvenci opisanih feature-ima, dok `y_train` predstavlja odgovarajuće tačne oznake za svaku poziciju u tim sekvencama. Drugim rečima, `X_train` sadrži ono što model vidi, a `y_train` sadrži ono što model treba da nauči da predviđa.

Na primer, jedna sekvenca u `X_train` može imati oblik:

```
[
    {'bias': 1, 'nuc': 'a'},
    {'bias': 1, 'nuc': 'c', '-1:nuc': 'a'},
    {'bias': 1, 'nuc': 'g', '-1:nuc': 'c'}
]
```

a odgovarajuće oznake u `y_train` mogu biti:

```
['0', 'I', 'I']
```

To znači da svaka pozicija u sekvenci ima svoj skup feature-a i svoju tačnu oznaku. CRF zatim na osnovu tih primera uči koji feature-i ukazuju na koju oznaku.

Kakav je ovo model? Model koji se ovde koristi je **linear-chain CRF**, odnosno linearno ulančano uslovno slučajno polje. To je probabilistički model namenjen označavanju sekvenci. Kod takvog modela ulaz je sekvenca

$$X = (x_1, x_2, \dots, x_n),$$

a izlaz je niz oznaka

$$Y = (y_1, y_2, \dots, y_n).$$

U problemu pronalaženja CpG ostrva, ulazna sekvenca može biti DNK sekvenca:

$$X = \text{A C G C G T},$$

dok niz oznaka može biti:

$$Y = \text{0 I I I I 0}.$$

Oznaka I može označavati poziciju unutar CpG ostrva, dok oznaka 0 može označavati poziciju van CpG ostrva.

CRF modeluje uslovnu verovatnoću:

$$P(Y | X).$$

To znači da model ne pokušava da objasni kako je DNK sekvenca nastala, već direktno uči koji niz oznaka je najverovatniji ako je data konkretna ulazna sekvenca. Zbog toga kažemo da je CRF **diskriminativni model**. Za razliku od HMM-a, koji je generativni model i modeluje zajedničku verovatnoću

$$P(X, Y),$$

CRF modeluje direktno:

$$P(Y | X).$$

Drugim rečima, HMM opisuje proces generisanja sekvence i skrivenih stanja, dok CRF direktno rešava zadatak označavanja već date sekvence.

Uloga algoritma lbfgs Parametar

`algorithm='lbfgs'`

označava da se za treniranje koristi optimizacioni algoritam L-BFGS. Tokom treniranja CRF treba da nauči težine feature-a i težine prelaza između oznaka. Te težine nisu unapred poznate, već se određuju tako da model što bolje opisuje trening podatke.

Na primer, model može naučiti da feature

`'nuc': 'c'`

povećava verovatnoću oznake I, dok feature

`'nuc': 'a'`

povećava verovatnoću oznake 0. Takođe, model može naučiti da je prelaz

$$I \rightarrow I$$

veoma prirodan, jer ako se jedna pozicija nalazi unutar CpG ostrva, često se i sledeća pozicija nalazi unutar istog ostrva.

Algoritam L-BFGS se koristi upravo za pronalaženje dobrih vrednosti tih težina. Intuitivno, on više puta prilagođava težine modela kako bi se smanjila greška na trening podacima.

Regularizacija: parametri c_1 i c_2 Parametri

```
c1=0.1,  
c2=0.1
```

odnose se na regularizaciju. Regularizacija se koristi da bi se sprečilo preprilagođavanje modela trening podacima. Preprilagođavanje, odnosno *overfitting*, nastaje kada model veoma dobro nauči trening skup, ali loše radi na novim podacima.

Parametar c_1 predstavlja jačinu L1 regularizacije, dok parametar c_2 predstavlja jačinu L2 regularizacije. Intuitivno, L1 regularizacija može dovesti do toga da neke težine postanu jednake nuli, pa se time određeni feature-i praktično odbacuju kao nevažni. L2 regularizacija uglavnom smanjuje prevelike težine i čini model stabilnijim.

U ovom primeru vrednosti su postavljene na 0.1, što znači da se koristi umerena regularizacija.

Maksimalan broj iteracija Parametar

```
max_iterations=100
```

određuje maksimalan broj iteracija optimizacionog algoritma. Pošto se težine modela uče postepeno, algoritam u svakoj iteraciji pokušava da ih popravi. Vrednost 100 znači da se taj postupak može ponoviti najviše sto puta.

Ako algoritam ranije pronadje dovoljno dobro rešenje, može se zaustaviti i pre dostizanja maksimalnog broja iteracija.

Svi mogući prelazi izmedju oznaka Parametar

```
all_possible_transitions=True
```

govori modelu da razmatra sve moguće prelaze izmedju oznaka, čak i ako se neki od njih nisu pojavili u trening podacima. Ovo je važno zato što CRF ne uči samo koji feature-i odgovaraju kojim oznakama, već uči i koji su prelazi izmedju oznaka prirodni.

Ako su oznake 0 i 1, mogući prelazi su:

$$0 \rightarrow 0, \quad 0 \rightarrow 1, \quad 1 \rightarrow 1, \quad 1 \rightarrow 0.$$

U problemu pronalaženja CpG ostrva, prelaz $1 \rightarrow 1$ je važan jer označava nastavljanje CpG ostrva. Prelaz $0 \rightarrow 1$ označava ulazak u CpG ostrvo, dok prelaz $1 \rightarrow 0$ označava izlazak iz CpG ostrva.

Uključivanjem svih mogućih prelaza model postaje fleksibilniji, jer može da dodeli težinu i prelazima koji se nisu direktno pojavili u trening skupu, ali mogu biti relevantni pri predikciji na novim sekvencama.

Treniranje modela Poziv

```
crf.fit(X_train, y_train)
```

pokreće treniranje modela. Na osnovu ulaznih feature-a i poznatih oznaka, CRF uči dve vrste težina:

težine feature-a

i

težine prelaza između oznaka.

Težine feature-a govore modelu koji lokalni obrasci u ulaznoj sekvenci ukazuju na određenu oznaku. Na primer, model može naučiti da nukleotidi **C** i **G** povećavaju verovatnoću oznake **I**. Težine prelaza govore modelu koji su nizovi oznaka prirodni. Na primer, model može naučiti da je prelaz $I \rightarrow I$ čest, dok su česti naizmenični prelazi $I \rightarrow O$ i $O \rightarrow I$ manje poželjni.

Zbog toga CRF ne posmatra svaku poziciju potpuno nezavisno, već traži najbolji niz oznaka za celu sekvencu.

Napomena o bloku try-except U kodu se treniranje nalazi unutar bloka:

```
try:
    crf.fit(X_train, y_train)
except AttributeError:
    pass
```

Ovaj zapis znači da program pokušava da istrenira model, ali ako se pojavi greška tipa `AttributeError`, ona se ignoriše. Takav pristup može biti problematičan, jer može sakriti grešku i dovesti do toga da program nastavi izvršavanje iako model nije uspešno istreniran.

Za učenje i proveru koda često je bolje napisati:

```
try:
    crf.fit(X_train, y_train)
except AttributeError as e:
    print(e)
```

ili privremeno potpuno ukloniti `try-except` blok, kako bi se eventualna greška jasno videla.

Dakle, ovaj deo koda konstruiše i trenira linear-chain CRF model. Model na osnovu trening podataka uči koje osobine sekvence i koji prelazi između oznaka najbolje objašnjavaju tačno označene sekvence, a zatim se može koristiti za predviđanje oznaka na novim DNK sekvencama.