Opis projektu

Tematem projektu jest:
Zastosowanie algorytmu EM do uproszczonej wersji znajdowania motywów w ciągach DNA

Na początku warto przybliżyć na czym właściwie polega algorytm *EM* (z ang. *Expectation–maximization algorithm*) i w jakim celu jest używany.

Opis algorytmu.

Algorytm *EM* jest skuteczną metodą iteracyjnego obliczania estymatorów największej wiarogodności (*ENW*), stosowaną przy rozwiązywaniu wielu problemów, w których dane są niekompletne. W każdej iteracji algorytmu *EM* wykonywane są dwa kroki: pierwszy krok nazywany E-krokiem (z ang. *expectation step*) oraz drugi krok nazywany M-krokiem (z ang. *maximization step*).

Ze względu na zastosowanie algorytmu *EM* w rozwiązywaniu problemów z brakującymi danymi jest on związany z pewną metodą estymacji *ad hoc*. W praktyce oznacza to, że parametry są estymowane po nadaniu brakującym danym pewnych wartości początkowych, następnie te brakujące dane są "uwiarygadniane" za pomocą wyestymowanych parametrów, a potem parametry są estymowane na nowo i tak aż do uzyskania pewnej zbieżności.

Krok E polega zatem na "stworzeniu" danych dla problemu z danymi kompletnymi przy użyciu niekompletnych danych, tak aby możliwe było wykonanie kroku M dla kompletnych danych. Będąc bardziej precyzyjnym w kroku E tworzona jest funkcja wiarogodności dla problemu z danymi kompletnymi, opiera się ona na nieobserwowanych danych przez co jest zastępowana przez jej warunkową wartość oczekiwaną względem obserwowanych danych. W koroku M szukamy maksimum utworzonej w poprzednim kroku funkcji wiarogodności. Rozpoczynając od odpowiedniej wartości początkowej parametru, powtarzamy kroki E i M aż do uzyskania zbieżności.

Algorytm EM.

Przejdźmy do matematycznego zapisu każdego z kroków w algorytmie *EM*.

Przedstawimy wzory odnoszące się do konkretnego problemu zawartego w naszym zadaniu, tzn X będzie wektorem modelującym obserwacje niekompletne postaci $X = (x_1, ..., x_n)$, gdzie $x_i = (x_{11}, x_{12}, ..., x_{1w})$. Każdym element $x_{1i} \in \sum = A, C, G, T$. Oznaczamy $d = |\sum|$, a więc w naszym przypadku $d = 4$. Przez $\theta_{a,j}$ rozumiem prawdopodobieństwo, że na $j-$tej pozycji występuje literka a. Mają więc macierz $\theta = (\theta_1, ..., \theta_w)$ możemy policzyć prawdopodobieństwo otrzymania ciągu $x_i = (x_{11}, x_{12}, ..., x_{1w})$, jako $P(X = x_i; \theta) = \prod_{i=1}^{w} \theta_{i,x_{1i}}$. Parametrem nieznanym w naszym przypadku jest θ.

Input: Obserwacje$X = (x_1, ..., x_n)$, gdzie każdy $x_i \in \mathbb{R}$

$t = 0$, randomowo inicjalizujemy $\theta^{(t)}$

E(xpectation) step, liczymy $Q_i^{(t)}(j) = P(Z_i = j|X, \theta^{(t)})$

M(aximization) step, najpierw liczymy

$$Q(\theta, \theta^{(t)}) = \sum_{i=1}^{n} \sum_{j=1}^{M} Q_i^{(t)}(j) log P(x_i, Z_i = j; \theta)$$

następnie

$$\theta^{(t+1)} = argmax Q(\theta, \theta^{(t)})$$

Jeśli sa zbieżne to przerywamy, inaczej ponownie powtarzamy krok E oraz M.

Skąd wiemy, że to działa?

Widzimy, że aby pokazać dowód poprawności wystarczy pokazać, że zachodzi: $l(\theta^{(t+1)} \geq \theta^{(t)})$ dla każdego t oraz $\theta^{(t)}$ oznacza estymacje parametru podczas t-tej iteracji.

Przypominając, $Q_i(j) = P(Z_i = j|X, \theta)$, a więc

$$l(\theta^{(t)}) = \sum_{i=1}^{n} \sum_{j=1}^{M} P(Z_i = j|X, \theta^{(t)}) \cdot log \frac{P(X, Z_i = j; \theta^{(t)})}{P(Z_i = j|X, \theta^{(t)})}$$

W kolejnym kroku aktualizujemy nasz parametr do $\theta^{(t+1)}$, który maksymalizuje tak, jak w równaniu poniżej:

$$l(\theta^{(t+1)}) = \sum_{i=1}^{n} \sum_{j=1}^{M} Q_i^{(t+1)}(j) \cdot log \frac{P(X, Z_i = j; \theta^{(t+1)})}{Q_i^{(t+1)}(j)}$$

$$\geq \sum_{i=1}^{n} \sum_{j=1}^{M} Q_i^{(t)}(j) \cdot log \frac{P(X, Z_i = j; \theta^{(t+1)})}{Q_i^{(t)}(j)}$$

$$\geq \sum_{i=1}^{n} \sum_{j=1}^{M} Q_i^{(t)}(j) \cdot log \frac{P(X, Z_i = j; \theta^{(t)})}{Q_i^{(t)}(j)} = l(\theta^{(t)})$$

Dlatego algorytm EM powoduje monotniczną zmianę prawdopodobieństwa. Powszechną praktyką monitorowania zbieżności jest testowanie, czy różnica prawdopodobieństw logarytmicznych przy dwóch iteracjach powodzenia jest mniejsza niż pewien z góry określony parametr tolerancji.

Krótkie podsumowanie treści teoretycznych.

Algorytm *EM* jest postrzegany jako ogólna iteracyjna metoda optymalizacyjna do maksymalizowania wiarygodności funkcji oceny przy zadanym modelu probabilistycznym z brakującymi danymi. Metoda ta jest bardzo wrażliwa na wybór warunków poczatkowych, które mogą prowadzic do uzyskaniach diametralnie różnych maksimów lokalnych. Dlatego bardzo ważne jest, żeby spróbować urochomić algorytm w zależności od różnych wartości początkowych.

Algorytm jest jednak powszechnie stosowany ze względu na dużą uniwersalność struktury i łatwość z jaką może być określony dla wielu różnych problemów. Algorytm EM ma wiele zastosowań w inteligencji obliczeniowej, statystyce matematycznej i uczeniu maszynowym.

Co właściwie zawiera nasz projekt?

Projekt składa się z dwóch plików. Pierwszy z nich *motif_generate.py* ma za zadanie na podstawie przekazanych parametrów wyznaczyć wartość macierzy X. Parametry, które przekazujemy znajdują się w pliku z rozszerzeniem `.json`. Z pomocą parsera odczytujemy je, a odpowiednie wartości przypisujemy do zmiennych i generujemy X.

W drugim pliku natomiast role nieco się odwracają, jak wskazuje nazwa *motif_estimate.py* w tym pliku będziemy estymować parametry, na podstawie X, którego zapisaliśmy w pliku poprzednim. Tak więc podobnie, jak poprzednio przekazujemy plik z roszerzeniem `.json`, w którym znajduje się macierz X oraz wartość α. Naszym zadaniem i równocześnie clue całego projektu jest znalezienie jak najlepiej wyestymowanych wartości θ oraz θ^B (dokładny opis i oznaczenia poniżej).

Algorytm *EM* będziemy wykorzystywały przy estymacji parametrów w pliku drugim.

Opis generowania X.

Opis generowania X to ściślej mówiąc opis działania skryptu pierwszego. Oprócz parsera, którego to opis działania pozwólmy sobie pominąć zawierają się tam dwie funkcje oraz ich wykorzystanie przy generowaniu X. Warto wspomnieć, że na wejściu dostajemy informację o parametrach $(\theta, \theta^B, \alpha, w, k)$, gdzie w oznacza długość jednego ciągu x_i oraz k oznacza ilość takich ciągów. Nasza wyjściowa macierz X jest więc rozmiaru k x w.

W kroku pierwszym na podstawie przekazanego parametru α z rozkładu dwumianowego generujemy probę długości k. Następnie dla tak wygenrowanego ciągu zero-jedynkowego wyznaczamy X. Zero w tym przypadku oznacza, że ciąg x_i pochodzi z θ^B, a jeden, że ciąg x_i pochodzi z θ. Mając te informacje chciałyśmy w jakiś sposób zmapować prawdopodobieństwa wystąpień w odpowiednim θ na element z ciągu.

Jednym z problemów w tym momencie była implementacja algorytmu pozwalającego wygenerować ciągi na podstawie prawdopodobieństwa wystąpnienia poszczególnych wartości (A,C,T,G lub, jak w naszym przypadku 1,2,3,4). Wykorzystana przez nas metoda to mapowanie otrzymanych wartości prawdopodobieństwa na przedziały liczbowe oraz użycie generatora pseudolosowego do generowania liczb w ty zakresie.

Kod do jednej z funkcji przedstawiam poniżej

```
def generate_from_T(Tn):

    randomNumber = randint(1, Precision)

    for i in range(0, w-1):
        if randomNumber <= Tn[0][i]:
            return 1
        elif (randomNumber > Tn[0][i]) and (randomNumber <= Tn[1][i]):
            return 2
        elif (randomNumber > Tn[1][i]) and (randomNumber <= Tn[2][i]):
            return 3
        else:
            return 4
```

Ściślej mówiąc, korzystamy z faktu, że prawdopobieństwo wylosowania liczby z przedziału (a, b) będącego podprzedziałem (L, R), takich że $a < b$, $L \geq a$ oraz $R \geq b$ wynosi $\frac{b-a+1}{R-L+1}$. Ustalamy więc wartości dla $L = 0$ oraz $R = Precision$ (im więszke tym lepiej - w naszym przypadku $Precision$ ma stałą wartość równą 10000), a nstępnie mapujemy poszczególne prawdopodobieństwa na podprzedziały. Losując liczbę z zakresu (L, R) oraz sprawdzając jej przynależność do poszczególnych podprzedziałów otrzymujemy generator działający zgodnie z otrzymanymi założeniami.

Tak wygenerowany X zapisujemy jako wynik działania skryptu razem z wartością α do pliku `generated_data.json`.

Przedstawimy teraz zastosowanie algorytmu i kilka możliwych modyfikacji, które jeszcze bardziej przybliżą nam zastosowanie algorytmu dla naszego problemu.

Implementacja algorytmu EM w projekcie ==========

W drugiej części zadania implementujemy działanie algorytmu *EM*. Parametry, na podstawie których będziemy szacowały θ oraz θ^B, to X oraz α. Jak już wiemy z poprzedniego rozdziału, jako krok startowy musimy w jakiś sposób wygenerować θ oraz θ^B, aby prawdopodobieństwo nie rozłożyło się w sposób równomierny (każda literka w danym ciągu nie miała tej samej częstości wystapień). Skorzystamy w tym celu z parametru α. Podobnie, jak w pliku pierwszym z rozkładu dwumianowego generujemy próbę długości k z zadanym, jako α prawdopodobieństwem. Na tej podstawie ustalamy pewien podział, a mianowicie czy dany ciąg pochodzi z θ, czy też z θ^B.

Następnie dla przekazanej macierzy X generujemy bazowe θ, najpierw zliczając poszczególne wartości w ciągach x_i pochodzących z danego rozkładu, a następnie dzieląc odpowiednie wartości przez sumy wystapień wszystkich wartości w tychże wierszach. Tym sposobem otrzymujemy, jako startowe elementy macierze częstości, które znacznie bardziej przybliżają

nas do rzeczywistych wartości.

Dla tak wygenerowanych wartości początkowych możemy przejść do zastosowania algorytmu *EM*. W naszym przypadku główna implementacja znajduje się w odpowiednio nazwanej funkcji `EM`. Funkcja ta jest tak naprawdę odzworowaniem przedstawionej powyżej modyfikacji. Jako pierwszy krok (*E-step*) wykonujemy zliczanie prawdopodbieństw (... jak to ładnie opisac co to za prawdopodobieństwa... ??) ... W tym celu używamy krótkich, aczkolwiek pomocnych funkcji zliczające prawdopodobieństwa z zadanego rozkładu dla danego ciągu x_i

Implementacja jednej z takich funkcji

```
def probability_Theta(k):

    prob = 1
    for i in range(w):
        prob *= Theta[int(X[k][i]) - 1, i]
    return prob
```

Znając powyższe możemy przejść do kolejnego kroku (*M-step*). Aktualizujemy wartości dla θ i θ^B na podstawie wartości Q_i oraz X.

Kroki E oraz M powtarzamy wielokrotnie, aż wartość pomiędzy kolejnymi wynikami będzie mniejsza od pewnego zadanego epsilona. Wtedy uzyskujemy najlepszą mozliwą estymację θ oraz θ^B. Tak uzyskane wartości przekazujemy, jako wynik do pliku z rozszerzeniem `.json`.

Powyższe zadanie to problem, w którym bardzo dobrze możemy zobaczyć, jako bardzo ułatawia i przyspiesza pracę zastosowanie opisanego algorytmu.