IDL 10 : Construction d'un HMM pour l'étiquetage morphosyntaxique automatique

Un TP de Y. Dupont.

Exercice 0

Créez un objet `HMM` qui a pour champs trois dictionnaires :

- `initial_prob` : les probabilités initiales
- `transition_prob` : les probabilités de transition d'une étiquette à une autre
- `emission_prob` : les probabilités d'émission d'un token étant donné une étiquette

Définissez aussi les méthodes d'instance (utiliser la fonction `get` de python):

- `HMM.initial` : renvoie la valeur de probabilité initiale d'une étiquette, sinon 0.0
- `HMM.transition` : renvoie la valeur de probabilité de transition d'une étiquette à une autre, sinon 0.0
- `HMM.emission` : renvoie la valeur de probabilité d'émission d'un sachant une étiquette, sinon 0.0

```
class HMM:
    def __init__(self, initial_prob, transition_prob, emission_prob):
        self.tags = sorted(emission_prob.keys())
        # à compléter

    def initial(self, tag):
        # à compléter

    def transition(self, tag_p, tag_c):
        # à compléter

    def emission(self, tag, token):
        # à compléter
```

Exercice 1 : écrire l'algorithme de *décodage* : l'algorithme de Viterbi

Pour un HMM, on appelle *décodage* le processus de retrouver, pour une séquence d'observations donnée, la séquence d'états cachés qui a servi à les générer la plus probable.

Dans le cadre de l'annotation morphosyntaxique, il s'agit de retouver la séquence d'étiquettes la plus probable étant donnée une séquence de mots.

Important : dans un HMM, on suppose que les étiquettes servent à générer les mots et non l'inverse. Cela est logique dans le sens où, dans une phrase syntaxiquement correcte, il est possible de changer un mot par un autre de même catégorie sans que la phrase devienne insensée ("je mange une pomme" –> "je mange une poire").

Note : certaines questions peuvent paraître un peu étranges à première vue, mais l'idée est de construire étape par étape l'algorithme de Viterbi, tout va s'imbriquer correctement à la fin !

Pour cet exercice, nous utiliserons Les probabilités suivantes :

```
exo1_initial = {"DET": 1.0}

exo1_transition = {
    "ADJ": {"NOUN": 1.0},
    "CLO": {"VERB": 1.0},
    "CLS": {"VERB": 1.0},
    "DET": {"NOUN": 0.8, "ADJ": 0.2},
    "NOUN": {"CLO": 0.5, "VERB": 0.5},
    "VERB": {"DET": 1.0}
}

exo1_emission = {
    "ADJ" : {"belle": 1.0},
    "DET" : {"le": 0.6, "la": 0.4},
    "NOUN": {"belle": 0.1, "porte": 0.8, "voile": 0.1},
    "VERB": {"porte": 0.6, "voile": 0.4},
    "CLO" : {"le": 1.0}
}
```

```
exo1_hmm = HMM(exo1_initial, exo1_transition, exo1_emission)

assert exo1_hmm.initial("DET") == 1, exo1_hmm.initial("DET")
assert exo1_hmm.initial("ADJ") == 0, exo1_hmm.initial("ADJ")

assert exo1_hmm.transition("DET", "ADJ") == 0.2, exo1_hmm.transition("DET",
"ADJ")
assert exo1_hmm.transition("DET", "NOUN") == 0.8,
exo1_hmm.transition("DET", "NOUN")
assert exo1_hmm.transition("DET", "VERB") == 0.0,
exo1_hmm.transition("DET", "VERB")

assert exo1_hmm.emission("DET", "le") == 0.6, exo1_hmm.emission("DET",
"le")
assert exo1_hmm.emission("DET", "la") == 0.4, exo1_hmm.emission("DET",
"la")
assert exo1_hmm.emission("DET", "les") == 0.0, exo1_hmm.emission("DET",
"les")
```

a. Calculez, pour chaque étiquette E, la probabilité de "la" sachant E.

La fonction `initials` va donner la probabilité de chaque étiquette pour le premier mot d'une phrase.

La fonction doit renvoyer un dictionnaire où les valeurs stockées seront un dictionnaire avec les clés `probabilité` (qui contiendra la probabilité) et `depuis` qui contient l'étiquette par laquelle la meilleure

transition se fait.

Puisqu'il n'y a pas de prédécesseur aux étiquettes initiales, la valeur contenue dans `depuis` doit systématiquement être None (pas de valeur). Cette convention est importante pour la suite.

Complétez la fonction `initials` jusqu'à ce que la cellule s'exécute sans erreur.

```
def initials(hmm, token):
    # à compléter

assert initials(exo1_hmm, "la") == {
    'ADJ': {'probabilité': 0.0, 'depuis': None},
    'CLO': {'probabilité': 0.0, 'depuis': None},
    'DET': {'probabilité': 0.4, 'depuis': None},
    'NOUN': {'probabilité': 0.0, 'depuis': None},
    'VERB': {'probabilité': 0.0, 'depuis': None}
}, initials(exo1_hmm, "la")
```

b. Calculez la meilleure transition vers l'étiquette `ADJ`

Pour calculer la meilleure transition, vous devez calculer les probabilités de toutes les transitions possibles et garder celle qui a la plus grande probabilité.

Pour calculer la meilleure transition, seule la probabilité de *transition* doit être utilisée. La probabilité d'émission ne doit être utilisée qu'à la fin, après avoir trouvé la meilleure transition.

La fonction `best_transition` doit renvoyer un dictionnaire avec :

- la clé `probabilité` associée à la probabilité la plus élevée
- la clé `depuis` associée à l'étiquette précédente qui a permis d'avoir la probabilité la plus élevée

```
def best_transition_to(hmm, probas_preced, etiquette, token):
    # à compléter

avant = initials(exo1_hmm, "la")
assert (
    best_transition_to(exo1_hmm, avant, "ADJ", "belle") == {'probabilité':
0.08000000000000002, 'depuis': 'DET'}
), best_transition_to(exo1_hmm, avant, "ADJ", "belle")
```

Comme vous pouvez le remarquer, les résultats sont inexacts. Cette limitation dans l'exactitude des nombres à virgule est due à des limitations des ordinateurs, python est innocent.

c. Calculez la meilleure transition vers toutes les étiquettes

En vous basant sur la fonction précédente, écrivez une fonction `best_transitions` qui va calculer la meilleure transition vers l'ensemble des étiquettes possibles.

```
def best_transitions(hmm, probas_preced, token):
    # à compléter

avant = initials(exo1_hmm, "la")
assert best_transitions(exo1_hmm, avant, "belle") == {
    'ADJ': {'probabilité': 0.08000000000000002, 'depuis': 'DET'},
    'CLO': {'probabilité': 0.0, 'depuis': 'ADJ'},
    'DET': {'probabilité': 0.0, 'depuis': 'ADJ'},
    'NOUN': {'probabilité': 0.03200000000000001, 'depuis': 'DET'},
    'VERB': {'probabilité': 0.0, 'depuis': 'ADJ'}
}
```

d. Construire la matrice de Viterbi

La matrice de Viterbi contient les probabilités de chaque état à chaque instant d'une séquence.

Dans notre cas, elle va contenir les probabilités de toutes les étiquettes pour l'ensembles des mots de la phrase. Elle contient également les pointeurs vers les étiquettes précédentes ayant permi les meilleures transitions.

Construction de la matrice

Écrivez la fonction `viterbi_matrix` qui va construire la matrice de Viterbi de manière effective.

La fonction `viterbi_matrix` prend en argument:

- `hmm`, le HMM utilisé pour décoder.
- `words`, la séquence de mots à décoder.

Vous utiliserez les fonctions `initials` et `best_transitions`.

```
def viterbi_matrix(hmm, words):
    # à compléter

matrix = viterbi_matrix(exo1_hmm, ["la", "belle", "porte", "le", "voile"])

def non_zeroes(d):
    return {tag: value for tag, value in d.items() if value["probabilité"]
!= 0}

assert non_zeroes(matrix[0]) == {'DET': {'probabilité': 0.4, 'depuis':
None}}, non_zeroes(matrix[0])
assert non_zeroes(matrix[1]) == {'ADJ': {'probabilité':
0.08000000000000002, 'depuis': 'DET'}, 'NOUN': {'probabilité':
0.03200000000000001, 'depuis': 'DET'}}, non_zeroes(matrix[1])
assert non_zeroes(matrix[2]) == {'NOUN': {'probabilité':
0.06400000000000002, 'depuis': 'ADJ'}, 'VERB': {'probabilité':
0.009600000000000003, 'depuis': 'NOUN'}}, non_zeroes(matrix[2])
assert non_zeroes(matrix[3]) == {'CLO': {'probabilité':
0.03200000000000001, 'depuis': 'NOUN'}, 'DET': {'probabilité':
```

```
0.005760000000000001, 'depuis': 'VERB'}}, non_zeroes(matrix[3])
assert non_zeroes(matrix[4]) == {'NOUN': {'probabilité':
0.00046080000000000014, 'depuis': 'DET'}, 'VERB': {'probabilité':
0.012800000000000004, 'depuis': 'CLO'}}, non_zeroes(matrix[4])
```

e. Reconstruire la séquence

Nous y sommes presque ! À présent que nous avons la matrice de Viterbi, il ne nous reste plus qu'à reconstuire la séquences d'étiquettes en la parcourant correctement.

Pour reconstruire la séquence, il faut partir de la fin et la reconstruire "à l'envers" en naviguant dans les transitions. Pourquoi partir de la fin ? La probabilité de la séquence est celle que l'on trouve sur le dernier élément. La séquence la plus probable est donc la probabilité la plus élevée que l'on trouve à la fin de la matrice.

Comment sait-on que cette séquence est la plus probable ? L'algorithme de Viterbi tire parti de l'hypothèse de Markov, où chaque état ne dépend que de l'état précédent. En naviguant de proche en proche, on sait qu'on a la séquence la plus probable.

Écrivez la fonction `viterbi` qui donne la séquence d'étiquettes la plus probable étant donné une séquence de mots (phrase).

```
def viterbi(hmm, sentence):
    # à compléter

tags, prob = viterbi(exo1_hmm, ["la", "belle", "porte", "le", "voile"])
assert tags == ['DET', 'ADJ', 'NOUN', 'CLO', 'VERB'], tags
assert prob == 0.012800000000000004, prob
```

Enfin, nous avons une implémentation de l'algorithme de Viterbi prête !

Exercice 2 : créer un HMM depuis un jeu d'entraînement

Pour cet exercice, nous travaillerons avec le corpus Sequoia, plus précisément sur sa version publiée dans le cadre du projet Universal Dependencies, accessible à l'adresse suivante : https://github.com/UniversalDependencies/UD_French-Sequoia/

Afin de ne pas trop passer de temps sur le prétraitement du corpus, une version plus simple est disponible dans le dossier `sequoia` sur le Moodle.

Téléchargez les données dans les ressources du TD et mettez-les dans un dossier `sequoia` à côté du notebook.

Les données on la forme suivante :

```
Que/SCONJ la/DET lumière/NOUN soit/VERB !/PUNCT
```

Chaque phrase est sur une ligne, chaque token est séparée par une espace et le mot est associé à son étiquette morphosyntaxique.

Écrivez une fonction `lire` qui, étant donné un chemin de fichier, lit un corpus ayant la forme évoquée précédemment.

Le résultat renvoyé doit être une liste de phrases. Une phrase est la liste des couples mot/étiquette.

```
def lire(path):
    # à compléter

sent = lire("test_lire.txt")
assert sent == [[['Que', 'SCONJ'], ['la', 'DET'], ['lumière', 'NOUN'],
['soit', 'VERB'], ['!', 'PUNCT']]], sent
```

Écrivez une fonction `depuis_corpus` qui, étant donné un corpus lu avec la fonction `lire`, renvoie un HMM.

```
def depuis_corpus(corpus):
    initials = {}
    emissions = {}
    transitions = {}

    # comptage depuis le corpus
    # à compléter

    # transformation des comptes en probabilités
    # à compléter

    return HMM(initials, transitions, emissions)
```

b. Tester le HMM

Exécutez la cellule suivante.

Elle va lire un corpus et créer un HMM depuis les statistiques faites sur ce corpus

```
train = lire("sequoia/fr_sequoia-ud-train.line.txt")
my_hmm = depuis_corpus(train)
sentence = ["Le", "professeur", "Gaston", "parle"]
path, probs = viterbi(my_hmm, sentence)

assert path == ['DET', 'NOUN', 'PROPN', 'VERB'], path

print("proba =", probs)
# La proba peut différer légèrement, mais devrait afficher :
# proba = 7.697081342015525e-17
```

Exécutez la cellule suivante afin d'évaluer la qualité de votre HMM. Plus précisément, vous mesurerez son `accuracy` (taux de bonnes réponses).

```
dev = lire("sequoia/fr_sequoia-ud-dev.line.txt")
total = 0
correct = 0
for sentence in dev:
    tokens = [token for token, tag in sentence]
    gold = [tag for token, tag in sentence]
    total += len(sentence)
    guess, prob = viterbi(my_hmm, tokens)
    correct += sum(guess[i] == gold[i] for i in range(len(gold)))
print("L'accuracy du HMM sur le corpus de développement est de :",
100*correct / total, "%")
```

L'accuracy de votre HMM devrait avoisiner 45%. Ce n'est pas très convainquant…

Afin d'améliorer la qualité, nous allons utiliser une heuristique appelée le "one count smoothing" (l'adoucissement de compte 1) qui donne une probabilité de "repli" (appelée backoff) aux mots inconnus.

Le one-count smoothing

L'idée de cette heuristique est de considérer qu'un mot inconnu (non présent dans les probabilités d'émission) soit considéré comme un mot étant apparu 1 fois dans le corpus.

Il y a une petite précision : il faut considérer que ces mots sont "exédentaires", leur probabilité doit donc être inférieure à celles des mots apparaissant effectivement une fois dans le corpus.

Pour cela, nous donnons une probabilité d'émission unique aux mots inconnus : `1 / (n + V)` où :

- `n` est le nombre de mots du corpus
- `V` est la somme du nombre d'étiquettes et du nombre de mots différents du corpus

Cette probabilité est la probabilité "de repli" (backoff), elle remplacera systématiquement la probabilité nulle des mots inconnus.

Réécrivez la classe `HMM` (faites un copier/coller) afin de prendre en compte cette probabilité `backoff`.

```
class HMM:
    def __init__(self, initial_prob, transition_prob, emission_prob,
backoff):
        self.tags = sorted(emission_prob.keys())
        # à compléter

    def initial(self, tag):
        # à compléter

    def transition(self, tag_p, tag_c):
        # à compléter
```

```
    def emission(self, tag, token):
        # à compléter
```

Ajoutez ensuite le calcul de la probabilité `backoff` à la fonction `depuis_corpus`

```
def depuis_corpus(corpus):
    initials = {}
    emissions = {}
    transitions = {}

    # comptage depuis le corpus
    # à compléter

    # calcul du backoff
    # à compléter

    # transformation des comptes en probabilités
    # à compléter

    return HMM(initials, transitions, emissions, backoff)
```

```
train = lire("sequoia/fr_sequoia-ud-train.line.txt")
dev = lire("sequoia/fr_sequoia-ud-dev.line.txt")
my_hmm = depuis_corpus(train)

total = 0
correct = 0
for sentence in dev:
    tokens = [token for token, tag in sentence]
    gold = [tag for token, tag in sentence]
    total += len(sentence)
    guess, prob = viterbi(my_hmm, tokens)
    correct += sum(guess[i] == gold[i] for i in range(len(gold)))
print("L'accuracy du HMM sur le corpus de développement est de :",
100*correct / total, "%")
```

Vous devriez à présent atteindre approximativement 93% d'accuracy ! C'est déjà plus convainquant !

Plutôt pas mal pour une si petite modification !