CSC3101 - CF1

24 janvier 2025

Prénom Nom: _____

Lisez attentivement les instructions données ci-dessous, et vérifiez que vous avez bien toutes les pages (18 au total). Cet examen papier dure deux heures, et les documents sont autorisés. L'usage d'un appareil connecté (ordinateur, tablette, etc) est interdit. Vous pouvez répondre à chaque question dans l'espace dédié. Soyez sûr que votre réponse est lisible. Si vous avez besoin de davantage d'espace, vous pouvez utiliser le verso des feuilles ou demander du papier supplémentaire. Le cas échéant, merci de le préciser sur le recto. Cet examen est noté sur 20 points + 1 point de bonus. N'hésitez pas à sauter des questions si vous êtes bloqués. Vous pouvez répondre aux questions suivantes en utilisant les fonctions supposées implémentées aux questions précédentes (pensez à regarder les signatures).

Les modèles de langage modernes, comme ceux utilisés par les assistants virtuels, fonctionnent comme des boîtes noires capables de générer des probabilités sur un vocabulaire donné. Ces modèles prennent un contexte textuel en entrée et produisent une distribution de probabilités pour chaque mot possible suivant. Cependant, leur puissance brute ne suffit pas pour générer un texte cohérent et pertinent : il est nécessaire d'utiliser des algorithmes traditionnels pour exploiter ces probabilités et construire des phrases complètes.

Prenons un exemple concret : supposons que le modèle reçoive le contexte "Il fait très". Il pourrait retourner les probabilités suivantes : "chaud" (0,6), "froid" (0,3), "beau" (0,1). L'objectif des algorithmes que vous allez implémenter est de transformer cette distribution de probabilités en une séquence de mots, comme "Il fait très chaud aujourd'hui", en respectant des contraintes de cohérence et de fluidité du texte généré.

Dans cet examen, vous allez mettre en œuvre certains de ces algorithmes. Vous commencerez par transformer des mots en identifiants uniques et calculer des probabilités dans un vocabulaire. Ensuite, vous implémenterez des méthodes de recherche telles que la recherche gloutonne (Greedy Search) et la recherche par échantillonnage (Sampling), chacune permettant d'explorer différentes stratégies pour générer du texte à partir des résultats du modèle.

Le but de cet examen est de vous familiariser avec les concepts qui entourent l'utilisation pratique des modèles de langage et de renforcer vos compétences en algorithmique. En complétant les différentes étapes, vous comprendrez mieux comment l'algorithmique traditionnelle s'intègre à l'intelligence artificielle pour résoudre des problèmes complexes.

1 Manipulation de texte et identifiants (5,5 points)

Avant d'associer des mots à des identifiants, il est important de comprendre pourquoi une telle opération est utile. Les ordinateurs travaillent plus efficacement avec des nombres qu'avec des chaînes de caractères. Par exemple, au lieu de manipuler directement le mot "chat", nous pourrions lui attribuer l'identifiant 42, permettant ainsi de stocker des mots de façon plus compacte et de travailler avec des matrices qui nécessitent un index.

Exemple : Pour un vocabulaire composé des mots suivants : ["chat", "chien", "oiseau"], une association possible serait :

- "chat" $\rightarrow 0$

- "chien" $\rightarrow$ 1
- "oiseau" $\rightarrow$ 2

1. (1 point) Écrivez une méthode *main* dans une classe *TSPAI* qui initialise un tableau contenant les mots "chat", "chien", et "oiseau" puis les affiche.

Solution:

```
1  public class TSPAI {
2
3      public static void main(String[] args){
4          String[] tab = {"chat", "chien", "oiseau"};
5          for (String s: tab){
6              System.out.println(s);
7          }
8      }
9
10 }
```

2. (0.5 points) Quelle structure de données permet d'associer des chaînes de caractères (String) à des entiers (int) ? Citez une classe Java qui peut être utilisée à cette fin.

Solution: Une table d'association, avec un HashMap par exemple.

On pourra accepter un map ou un dictionnaire.

3. (1 point) Créez une classe Vocabulary qui permet de gérer ces associations. Cette classe devra :

   - Contenir un tableau de chaînes de caractères appelé *words* (le vocabulaire).
   - Contenir une structure de données appelée *wordToIndex* permettant de faire correspondre chaque mot à un entier distinct (voir la question précédente, si vous ne connaissez pas la classe Java appropriée, vous pouvez utiliser une classe fictive appelée *Association* ayant des méthodes *put* et *get*).
   - Avoir un constructeur prenant en entrée un tableau de chaînes de caractères **uniques** et initialisant seulement le tableau de chaînes de caractères *words* pour l'instant (on n'initialise pas *wordToIndex*).

Solution:

```
1 public class Vocabulary {
2     private String[] words;
3     private Association<String, int> wordToIndex;
4
5     public Vocabulary(String[] words) {
6         this.words = words;
7         this.associations = getAssociations(words); // Not required
8     }
9 }
```

On ne demande pas les visibilités ici.

4. (0.5 points) Supposons que nous ayons un tableau de chaînes de caractères **uniques**. Écrivez une méthode d'instance *String getWordAtIndex(int idx)* dans la classe *Vocabulary*, qui retourne le mot associé à un indice donné.

Solution:

```
1 public String getWordAtIndex(int idx) {
2     return this.words[idx];
3 }
```

5. (1 point) Écrivez une méthode appelée *Association<String, int> getAssociations(String[] words)* dans la classe *Vocabulary*. Cette méthode prend en entrée un tableau de chaînes de caractères **uniques** et retourne une structure de donnée où chaque mot est associé à un entier **distinct**, correspondant à sa position dans le tableau. Vous pouvez supposer que la classe *Association* a les méthodes suivantes qui vous seront utiles dans cette question et la suivante :

   - *void put(String key, int value)* : ajoute une association clé-valeur.
   - *int get(String key)* : retourne la valeur associée à une clé.
   - On initialise une nouvelle association avec *new Association<>()*.

Solution:

```
1  private Association<String, int> getAssociations(String[] words) {
2      Association<String, int> assoc = new Association<>();
3      for (int i = 0; i < words.length; i++) {
4          assoc.put(words[i], i);
5      }
6      return assoc;
7  }
```

6. (0.5 points) On suppose que l'on rajoute la ligne *this.wordToIndex = getAssociations(words);* dans le constructeur. Implémentez la méthode *int getWordIndex(String word)* de la classe *Vocabulary* permettant d'obtenir l'index associé au mot *word* (on supposera que le mot a bien un index).

Solution:

```
1          public int getWordIndex(String word) {
2              return this.wordToIndex.get(word);
3          }
```

7. (1 point) Quelle visibilité recommandez-vous pour les deux attributs Java, le constructeur, et les trois méthodes implémentées ? Justifiez clairement votre réponse.

Solution:

Les attributs sont privés, le constructeur, *getWordAtIndex* et *getWordIndex* sont public, et *getAssociations* est privée. On justifie cela en expliquant que l'utilisateur de la classe n'a pas besoin d'accéder à tout le fonctionnement interne et veut juste un moyen d'associer des mots à des index et vice versa.

La visibilité par défaut peut être acceptée si bien expliquée à la place des visibiltés publiques.

2 Représentation des probabilités (5 points)

Pour générer un texte, les modèles de langage renvoient une distribution de probabilités sur un vocabulaire. Ces probabilités permettent de choisir le mot suivant à partir des probabilités associées aux mots possibles. Une probabilité est une valeur comprise entre 0 et 1, et la somme des probabilités pour l'ensemble du vocabulaire doit être égale à 1.

1. (0.5 points) Définissez une nouvelle exception *NotAProbabilityException*. Cette exception sera utilisée pour signaler qu'un tableau de flottants donné en entrée n'est pas une distribution de probabilités valide. Le constructeur n'est pas demandé.

Solution:

```
1 public class NotAProbabilityException extends Exception {
2     public NotAProbabilityException(String message) {
3         // Constructor not required
4         super(message);
5     }
6 }
```

2. (2 points) Implémentez une classe *Probability* pour stocker une distribution de probabilités. La classe devra :
    1. Contenir un tableau de flottants nommé *probabilities*.
    2. Avoir un constructeur prenant en paramètre un tableau de flottants.

3. Vérifier dans le constructeur que chaque élément du tableau est compris entre 0 et 1, et que la somme totale des éléments est égale à 1. Si ces conditions ne sont pas respectées, levez une exception *NotAProbabilityException*.

Solution:

```
1  public class Probability {
2      private float[] probabilities;
3
4      public Probability(float[] probabilities) throws NotAProbabilityException {
5          float sum = 0;
6          for (float p : probabilities) {
7              if (p < 0 || p > 1) {
8                  throw new NotAProbabilityException("The probabilities must be
9                  between 0 and 1.");
10             }
11             sum += p;
12         }
13         if (Math.abs(sum - 1.0) > 1e-6) {
14             throw new NotAProbabilityException("The probabilities must sum to one.");
15         }
16         this.probabilities = probabilities;
17     }
18 }
```

On acceptera aussi la comparaison *sum == 1.0*.

On supposera par la suite l'existence d'une méthode *float getProbability(int idx)* dans la classe *Probability*, permettant de retourner la probabilité associée à un index donné. Vous n'avez pas à implémenter cette méthode.

De plus, la gestion de l'exception *NotAProbabilityException* **n'est pas demandée** par la suite quand vous utilisez le constructeur de la classe *Probability*.

3. (1 point) Écrivez une fonction *int getIndexMaxProbability()* qui retourne l'index auquel la probabilité est maximale. En cas d'égalité, on pourra retourner n'importe quel index ayant une probabilité maximale.

Solution:

```
1  public int getIndexMaxProbability() {
2          float maxProbability = -1;
3          int maxIndex = -1;
4
5          for (int i = 0; i < probabilities.length; i++) {
6              if (this.getProbability(i) > maxProbability) {
7                  maxProbability = this.getProbability(i);
8                  maxIndex = i;
9              }
10         }
11
12         return maxIndex;
13     }
```

4. (1.5 points) Souvent, il arrive que nous générions une liste de **flottants positifs** qui ne somment pas à 1. Nous pouvons transformer cette liste de flottants en une distribution de probabilité en la normalisant, c'est-à-dire en divisant chaque élément par la somme de tous les éléments. Écrivez une **méthode de classe** *float[] normalize(float[] elements)* dans *Probability* qui fait cette normalisation.

Solution:

```
static float[] normalize(float[] probabilities) {
    // Calcul de la somme des probabilites
    float sum = 0;
    for (float p : probabilities) {
        sum += p;
    }

    // Creation d'un tableau pour les probabilites normalisees
    float[] normalized = new float[probabilities.length];
    for (int i = 0; i < probabilities.length; i++) {
        normalized[i] = probabilities[i] / sum; // Division par la somme
    }

    return normalized;
}
```

3 Représentation du modèle de langage (4 points)

Les modèles de langage peuvent varier considérablement dans leur complexité et leur manière de générer des probabilités. Nous décidons donc de les représenter à travers une interface standardisée.

1. (0.5 points) Expliquez brièvement pourquoi l'utilisation d'une interface est utile pour représenter un modèle de langage.

Solution: On peut implémenter des algorithmes sans être dépendant de l'implémentation.

2. (0.5 points) Créez une interface *LanguageModel* contenant une méthode unique *Probability getNextWordProbability(int[] context)*. Ici, *context* est un tableau d'entiers représentant l'historique des mots précédents (i.e., les identifiants des mots générés jusque-là).

Solution:

```
1 public interface LanguageModel {
2     Probability getNextWordProbability(int[] context);
3 }
```

3. (1 point) Implémentez une classe *RandomLanguageModel* qui réalise une probabilité uniforme sur un vocabulaire donné. Vous supposerez que le constructeur suivant est déjà implémenté :

```
1 public RandomLanguageModel(int vocabularySize) {
2     this.vocabularySize = vocabularySize;
3 }
```

Complétez la classe pour qu'elle respecte l'interface *LanguageModel* et retourne une distribution uniforme des probabilités.

Solution:

```
1  public class RandomLanguageModel implements LanguageModel {
2      private int vocabularySize; // Non demande
3
4      @Override
5      public Probability getNextWordProbability(int[] context) {
6          float[] probabilities = new float[vocabularySize];
7          for (int i = 0; i < vocabularySize; i++) {
8              probabilities[i] = 1.0f / vocabularySize;
9          }
10         return new Probability(probabilities);
11     }
12 }
```

4. (2 points) Un modèle de langage statistique couramment utilisé avant l'ère du deep learning est le modèle bigramme. L'idée est d'approximer la probabilité du mot suivant uniquement à partir du mot précédent. À partir d'un large corpus de texte, on peut obtenir la probabilité de chaque bigramme, c'est-à-dire la probabilité de chaque paire de deux mots. Par exemple, à partir de la phrase "Le chat saute", le mot suivant est déduit du mot "saute".

   Supposons que nous ayons la classe *BigramLanguageModel* avec l'attribut et le constructeur suivant :

```
1      private int[][] numberCooccurrence;
2
3      // Constructeur
4      public BigramLanguageModel(int[][] numberCooccurrence) {
5          this.numberCooccurrence = numberCooccurrence;
6      }
```

   Ici, *numberCooccurrence[i][j]* le nombre de fois que j a été observé après i dans le corpus. Dans le cas où il n'y a pas de mot précédent, on considérera qu'il s'agit du mot vide "". Faites en sorte que

BigramLanguageModel mette en œuvre l'interface *LanguageModel* en implémentant la ou les méthodes adéquates.

Solution:

```
1  public class BigramLanguageModel implements LanguageModel {
2
3      @Override
4      public Probability getNextWordProbability(int[] context) {
5          String previousWord = "";
6          if (context.length > 0) previousWord = context[context.length - 1];
7          float[] probabilities = new float[vocabularySize];
8          for (int i = 0; i < vocabularySize; i++) {
9              probabilities[i] = this.numberCooccurrence[previousWord][i];
10         }
11         return new Probability(Probability.normalize(probabilities));
12     }
13 }
```

4 Génération gloutonne (6 points)

La génération de texte peut être vue comme l'exploration d'un arbre où chaque nœud représente une combinaison de mots. Le premier niveau correspond aux premiers mots possibles, le deuxième niveau aux seconds mots possibles, et ainsi de suite. Chaque chemin dans cet arbre forme une séquence complète.

1. (0.5 points) Si la taille du vocabulaire est V et que l'on souhaite générer une séquence de longueur L, combien de séquences différentes faut-il examiner dans une **exploration exhaustive** ? Quelle est la complexité de cette approche en termes de V et L ? Concluez sur la faisabilité de cette méthode.

Solution: $\mathcal{O}(V^L)$, ce qui est exponentiel et donc non possible en pratique.

La génération gloutonne est une méthode plus simple pour produire une séquence à partir d'un modèle de langage. À chaque étape, elle sélectionne le mot ayant la probabilité la plus élevée dans la distribution fournie par le modèle. Ce processus est répété jusqu'à atteindre une longueur maximale ou rencontrer un critère d'arrêt (par exemple, un mot de fin).

Exemple : Considérons un modèle de langage et le contexte initial suivant : [Il, fait]. À une certaine itération, le modèle retourne les probabilités suivantes pour le prochain mot :

1. "très" : 0,5
2. "pas" : 0,3
3. "beau" : 0,2

Le mot "très" est choisi, car il a la probabilité la plus élevée. Le contexte devient alors [Il, fait, très], et le processus est répété.

2. (0.5 points) Supposons que le modèle de langage génère les probabilités suivantes au cours des itérations :

   1. Itération 1 : "chat" (0,4), "chien" (0,3), "oiseau" (0,3)
   2. Itération 2 : "joue" (0,6), "court" (0,4)
   3. Itération 3 : "dans" (0,8), "sur" (0,2)

   En utilisant la génération gloutonne, quelle est la phrase produite si le contexte initial est "Le" et que vous générez trois mots ?

Solution: Le chat joue dans

3. (2 points) Implémentez une méthode *int[] generate(int[] initialContext, LanguageModel languageModel, int nIterations)* dans une classe GreedyGenerator. Cette méthode produit une séquence de mots de manière gloutonne, en utilisant un contexte initial et un modèle de langage. Le nombre de mots générés est *nIterations*

   On pourra utiliser la méthode *Arrays.copyOfRange(array, début, fin)* pour extraire le sous-tableau de *array* allant de l'indice *début* (inclu) à l'indice *fin* (exclu).

Solution:

```
1  public class GreedyGenerator {
2      public int[] generate(int[] initialContext, LanguageModel languageModel,
3                            int nIterations) {
4          int[] sequence = new int[ititialContext.length + nIterations];
5          int length = initialContext.length;
6
7          // Copy the initial context
8          for (int i = 0; i < length; i++) {
9              sequence[i] = initialContext[i];
10         }
11
12         for (int i = length; i < ititialContext.length + nIterations; i++) {
13             Probability probabilities =
14             languageModel.getNextWordProbability(Arrays.copyOfRange(sequence, 0, i));
15             int nextWord = probability.getIndexMaxProbability();
16             sequence[i] = nextWord;
17         }
18
19         return sequence;
20     }
21 }
```

4. (1 point) Quelle est la complexité de cet algorithme en fonction de la complexité de *getNextWordProbability* (notée M), de la longueur maximale *maxLength* et de la taille du vocabulaire V (on ignorera la taille du contexte initial) ?

Solution:
$\mathcal{O}((M + V) * maxLength)$

5. (0.5 points) Est-ce que l'approche gloutonne donne toujours la séquence de probabilité maximale (la probabilité d'une séquence est le produit de toutes les probabilités des générations sur la séquence) ? Expliquez pourquoi.

Solution: Non, elle n'est pas optimal. Par exemple :

1. P("elle") = 0.51, P("il") = 0.49
2. P("boit"|"elle") = 0.51, P("dort"|"elle") = 0.49
3. P("boit"|"il") = 1.0, P("dort"|"il") = 0

L'approche gloutonne produit la séquence "elle boit" de probabilité 0.26, mais la phrase "il boit" a une probabilité supérieure de 0.49.

6. (1 point) Une limitation importante de la génération gloutonne est qu'elle peut produire des textes trop prévisibles. Pour remédier à cela, on peut utiliser une méthode basée sur l'échantillonnage. Cette méthode consiste à choisir un mot aléatoirement en fonction de sa probabilité, introduisant ainsi plus de diversité dans la génération. Expliquez en pseudo-code comment choisir un mot aléatoirement selon une distribution de probabilité donnée. Vous pouvez supposer qu'une fonction rand() génère un nombre aléatoire entre 0 et 1. Quelle est la complexité de cette approche ?

Solution:

```
float random = rand();
float cumulativeProbability = 0;

for (int i = 0; i < probabilities.size(); i++) {
    cumulativeProbability += probabilities.getProbability(i);
    if (random <= cumulativeProbability) {
        return i;
    }
}
```

La complexité est linéaire en la taille du vocabulaire.

7. (0.5 points) L'approche par échantillonage est très utilisée en pratique avec deux modifications. La première consiste à accentuer ou réduire les différences de probabilités à travers un paramètre appelé la température. La deuxième consiste à ne considérer que les top K mots les plus probables lors de l'échantillonage afin d'éviter de sélectionner des mots peu probables qui, ensemble, forment souvent une probabilité non négligeable. Expliquez brièvement et sans code comment vous pourriez trouver les K plus grand éléments d'un tableau avec une complexité en $\mathcal{O}(N * log(N))$, où N est la taille du tableau.

Solution: On pourrait trier la liste par ordre décroissant et retourner le K premiers éléments.