**FastText algorithm**

**Unsupervised**

1. Initialiser un dictionnaire de mots distincts en parcourant un fichier texte d’entrainement

int Dictionary.MAX\_VOCAB\_SIZE

* La constante MAX\_VOCAB\_SIZE borne le nombre maximum de mots distincts qui peuvent être stockés dans le dictionnaire

List<entry> Dictionary.words\_

* Liste d’entrées (class entry) enregistrant les propriétés des mots distincts stockés dans le dictionnaire

int Dictionary.size\_

* Nombre effectif de mots distincts stockés dans le dictionnaire (= taille de la liste \_words)
* Pendant l’initialisation du dictionnaire, représente l’indice de position dans le dictionnaire du dernier mot distinct rencontré

int Dictionary.ntokens\_

* Nombre total de mots examinés pour initialiser le contenu du dictionnaire (doublons compris)

void Dictionary.readFromFile(fileReader)

* Initialiser à 1 le nombre minimum d’occurrences d’un mot qui le rendent éligible à un stockage dans le dictionnaire (dans une taille de vocabulaire contrainte, on privilégie les mots les plus fréquemment rencontrés)
* Examiner l’un après l’autre les mots lus dans le fichier texte (étape 2.)
  + Créer une entrée pour le mot dans le dictionnaire, ou incrémenter le nombre d’occurrences du mot dans une entrée déjà existante
  + Incrémenter le compteur ntokens\_
  + Si le nombre de mots stockés dans le dictionnaire (size\_) dépasse 75% de la taille maximale du dictionnaire (MAX\_VOCAB\_SIZE)
    - Incrémenter le nombre minimum d’occurrences d’un mot qui le rendent éligible à un stockage dans le dictionnaire
    - Supprimer du dictionnaire tous les mots qui ne présentent pas le nouveau nombre d’occurrences minimum (cf ci-dessous)

void Dictionary.threshold(long t, long tl)

* Supprimer du dictionnaire tous les mots qui ne présentent pas au moins args\_.minCount occurrences, et tous les labels qui ne présent pas au moins args\_.minCountLabel occurrences
  + Trier les entrées du dictionnaire par nombre d’occurrences décroissant, en respectant l’ordre initial pour les nombres d’occurrences égaux
  + Supprimer toutes les entrées dont le nombre d’occurrences est inférieur à args\_.minCount (mots) et args\_.minCountLabel (labels), redimensionner le dictionnaire
  + Réinitialiser le tableau word2int\_
    - Positionner toutes les cellules à la valeur -1
    - Parcourir les mots restant dans le dictionnaire
    - Recalculer le code numérique associé à chaque mot
    - Valoriser la cellule du tableau word2int\_ à la position du code numérique avec le nouvel indice de position du mot dans le dictionnaire
    - Au passage, recompter le nombre de mots (nwords\_) et le nombre de labels (nlabels\_)

void Dictionary.initTableDiscard()

* Précalculer pour chaque mot, en fonction de sa fréquence dans le fichier, un facteur qui limitera le nombre de fois où les termes les plus récurrents sont pris en compte dans l’entrainement non supervisé (mots de liaison qui ne portent pas beaucoup de sens et n’apportent rien de plus après un certain nombre d’occurrences)
  + Initialiser un tableau de nombres de la taille du dictionnaire (size\_)
  + Soient args\_.t le paramètre qui définit le seuil d’échantillonnage en terme de fréquence d’un mot dans le corpus (t = 10-4 par défaut) et f la fréquence du mot en position i dans le dictionnaire, alors la cellule i du tableau est valorisée à : sqrt(t/f) + t/f
  + Un mot du jeu d’entrainement commence à être très rarement rejeté s’il est 2.6x plus fréquent que le seul d’échantillonnage, il est rejeté : 20% du temps si 3.5x plus fréquent, 50% s’il est 7 fois plus fréquent, 80% s’il est 33x plus fréquent, 95% s’il est 500x plus fréquent

*In order to subsample the most frequent words, we use a rejection threshold of 10−4 (for more details, see (Mikolov et al., 2013b* [*https://arxiv.org/pdf/1310.4546.pdf*](https://arxiv.org/pdf/1310.4546.pdf) *: Subsampling of Frequent Words  
In very large corpora, the most frequent words can easily occur hundreds of millions of times (e.g., “in”, “the”, and “a”). Such words usually provide less information value than the rare words. For example, while the Skip-gram model beneﬁts from observing the co-occurrences of “France” and “Paris”, it beneﬁts much less from observing the frequent co-occurrences of “France” and “the”, as nearly every word co-occurs frequently within a sentence with “the”. This idea can also be applied in the opposite direction; the vector representations of frequent words do not change signiﬁcantly after training on several million examples.  
To counter the imbalance between the rare and frequent words, we used a simple subsampling approach: each word wi in the training set is discarded with probability computed by the formula P(wi) = 1 − sqrt( t / f(wi)) where f(wi) is the frequency of word wi and t is a chosen threshold, typically around 10−5. We chose this subsampling formula because it aggressively subsamples words whose frequency is greater than t while preserving the ranking of the frequencies. Although this subsampling formula was chosen heuristically, we found it to work well in practice. It accelerates learning and even signiﬁcantly improves the accuracy of the learned vectors of the rare words, as will be shown in the following sections.*

void Dictionary.initNgrams()

void Dictionary.computeNgrams(string word, List<int> ngrams)

* Générer toutes les séquences de 3 à 6 lettres contenues dans chaque mot et les stocker dans la propriété "subwords" de chaque élément du dictionnaire (entry)
  + Préfixer et suffixer le mot par '<' et '>' pour distinguer les chaines de caractères en début et en fin de mot
  + Ajouter le mot lui-même ainsi encadré dans la liste "subwords"
  + Parcourir l’un après l’autre les caractères du mot
    - Sauter les caractères de contrôle dont le code est compris entre 128 (Hex80) et 191 (HexC0) comme première lettre
    - Générer toutes les suites de caractères de longueur comprise entre args\_.minn (3 par défaut) et args\_.maxn (6 par défaut) commençant par le caractère courant (à l’exclusion du mot entier)
    - Ne pas compter les caractères de contrôle dont le code est compris entre 128 (Hex80) et 191 (HexC0) dans le calcul de la longueur d’un ngram
  + Appliquer la function de hachage "Fowler–Noll–Vo" sur les caractères de chaque ngram, puis restreindre le nombre obtenu dans la plage de valeurs [0, args\_.bucket] (modulo) - valeur par défaut = 2 000 000, et ajouter le nombre de mots (nwords\_)
  + A la fin de cette procédure, la propriété subwords de chaque élément du dictionnaire contient : l’indice de position du mot, des hash compris entre [0, args\_.bucket], décalés du nombre de mots dans le dictionnaire (de sorte qu’il ne peut y avoir de collision entre les indices des mots et des ngrams), qui représentent toutes les séquences de 3 à 6 lettres contenues dans chaque mot

*Subword model. By using a distinct vector representation for each word, the skip-gram model ignore the internal structure of words. In this section, we thus propose a different scoring function s, in order to take into account this information. Given a word w, let us denote by Gw ⊂ {1,... ,G} the set of n-grams appearing in w. We associate a vector representation zg to each n-gram g. We represent a word by the sum of the vector representations of its n-grams. We always include the word w in the set of its ngrams, to also learn a vector representation for each word. The set of n-grams is thus a superset of the vocabulary. It should be noted that different vectors are assigned to a word and a n-gram sharing the same sequence of characters. This simple model allows sharing the representations across words, thus allowing to learn reliable representation for rare words.  
Dictionary of n-grams. The presented model is simple and leaves room for design choices in the deﬁnition of Gw. In this paper, we adopt a very simple scheme: we keep all the n-grams with a length greater or equal than 3 and smaller or equal than 6. Different sets of n-grams could be used, for example preﬁxes and sufﬁxes. We also add a special character for the beginning and the end of the word, thus allowing to distinguish preﬁxes and sufﬁxes. In order to bound the memory requirements of our model, we use a hashing function that maps n-grams to integers in 1 to K. In the following, we use K equal to 2 millions. In the end, a word is represented by its index in the word dictionary and the value of the hashes of its n-grams. To improve the efﬁciency of our model, we do not use n-grams to represent the P most frequent words in the vocabulary. There is a trade-off in the choice of P, as smaller values imply higher computational cost but better performance. When P = W, our model is the skip-gram model of Mikolov et al. (2013b).*

2. Convertir un flux de caractères (fichier ou saisie) en un flux de labels et de mots

bool Dictionary.readWord(textReader, out word)

* Les caractères séparateurs de mots sont ' ', '\r', '\t', '\v', '\f', '\0'
* Ils sont ignorés, quel que soit leur nombre d’occurrences successives
* Le caractère séparateur de phrases est '\n'
* Il produit directement le mot standard "</s>" = fin de phrase
* Tous les autres caractères s’accumulent pour constituer un mot
* La fonction retourne de manière répétée "true" et un mot non vide
* Puis, quand la fin du flux est atteinte, "false" (et un mot vide)

Déterminer si le terme qui vient d’être lu est un label ou un mot

entry\_type Dictionary.getType(word)

* Si le terme lu commence par la chaine de caractères stockée dans le paramètre args\_.label, c’est un label (entry\_type.label), sinon c’est un mot (entry\_type.word)
* Valeur par défaut de args\_.label : "\_\_label\_\_"

3. Préparer une structure de donnée qui permettra de retrouver très rapidement la position d’un mot dans le dictionnaire à partir de ses caractères (hashtable)

int[] Dictionary.word2int\_

* Le tableau word2int\_ permet de retrouver l’indice de position d’un mot dans le dictionnaire à partir d’un code numérique calculé rapidement en fonction de ses caractères
* Ce tableau est de taille MAX\_VOCAB\_SIZE
* Ses cellules sont initialisées à la valeur -1

uint Dictionary.hash(string str)

* Appliquer la function de hachage "Fowler–Noll–Vo" sur les caractères du mot  
  *A hash function is any function that can be used to map data of arbitrary size to data of fixed size.   
  The values returned by a hash function are called hash values, hash codes, digests, or simply hashes.   
  One use is a data structure called a hash table, widely used in computer software for rapid data lookup.  
  The FNV hash function was designed for fast hash table use:*[*https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo\_hash\_function*](https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function)
* On produit un nombre entier non signé (32 bits)
* A ce stade, plusieurs mots peuvent avoir le même code (collisions)

int Dictionary.find(word)

* Ramener le nombre entier dans l’intervalle [0, MAX\_VOCAB\_SIZE] (modulo)
* Ce nombre devient un indice de position dans le tableau word2int\_
* Regarder si un mot différent est déjà stocké à cet indice de position dans le tableau word2int\_
* Si c’est le cas, incrémenter le nombre entier et reprendre au point 1   
  *(cette boucle permet d’éviter les collisions de la fonction de hachage)*
* Sinon, retourner le code numérique finalement retenu pour le mot

void Dictionary.add(word)

* Si on rencontre ce mot pour la première fois = mot pas encore stocké dans le dictionnaire = valeur -1 à l’indice de position correspondant au code numérique retenu dans le tableau word2int\_
* Incrémenter le compteur size\_ (à ce stade, sa valeur indique l’indice de position du mot dans le dictionnaire)
* Enregistrer : word2int\_[code numérique du mot] = [indice de position du mot dans le dictionnaire (size\_)]

4. Stocker un mot dans le dictionnaire

void Dictionary.add(word)

* Calculer le code numérique du mot (étape 3.), puis utiliser le tableau de correspondance word2int\_ pour identifier la position du mot dans la liste d’entrée words\_
* Si on rencontre ce mot pour la première fois = mot pas encore stocké dans le dictionnaire = valeur -1 à l’indice de position correspondant au code numérique retenu dans le tableau word2int\_ :
  + Créer une nouvelle entrée dans le dictionnaire (class entry) : mot, 1 occurrence, type = label ou mot
  + Ajouter cette nouvelle entrée dans la liste de mots words\_
* Sinon :
  + Récupérer l’entrée du mot déjà existante dans le dictionnaire
  + Incrémenter la propriété « nombre d’occurrences » dans cette entrée

5. Initialiser une matrice d’entrée, de sortie, puis lancer l’entrainement

void FastText.train(args)

(option 1) Charger un fichier de vecteurs précalculés dans la matrice d’entrée

void FastText.loadVectors(filename)

* Lire un fichier texte, valeurs séparées par des caractères de type « whitespace »
* Lire deux entiers 32 bits : nombre de mots, dimension des vecteurs
* Répéter pour le nombre de mots :
  + Lire un mot, l’ajouter dans une liste temporaire, l’ajouter dans le dictionnaire (cf 4. Dictionary.add)
  + Lire autant de décimaux 32 bits que de dimensions : stocker le vecteur de décimaux dans une matrice temporaire
* Re-trier les entrées du dictionnaire par nombre d’occurrences décroissant (cf 1. Dictionary.threshold)
* Initialiser une matrice input\_ de taille (nombre mots + nombre buckets) x dim, remplie de nombres aléatoires compris entre -1/dim et +1/dim
* Parcourir les mots du fichier de vecteur dans la liste temporaire
  + Récupérer l’indice de position du mot dans le dictionnaire
  + Copier le vecteur de la matrice temporaire à l’indice de position cible du mot dans la matrice input\_

(option 2) Initialiser une matrice d’entrée vide

* Initialiser une matrice input\_ de taille (nombre mots + nombre buckets) x dim, remplie de nombres aléatoires compris entre -1/dim et +1/dim

Initialiser une matrice de sortie

* Initialiser une matrice de sortie output\_ remplie de zéros, de taille :
  + Non supervisé : nombre de mots x dim
  + Supervisé : nombre de labels x dim

Lancer l’entrainement

* Réinitialiser le compteur de tokens partagé entre tous les threads (tokenCount)
* Appeler args.thread\_ fois en parallèle la méthode trainThread(index)   
  => voir 6.
* Encapsuler les matrices d’entrée, de sortie, et les arguments dans un objet de type Model, model\_
* Sauvegarder le modèle dans un fichier binaire

void FastText.saveModel()

* + Utiliser l’extension .bin si le modèle n’est pas quantifié, .ftz si le modèle est quantifié
  + Ecrire deux entiers : un nombre magique pour reconnaitre le type de fichier + une version de l’algorithme
  + Ecrire les arguments
  + Ecrire le dictionnaire
  + Si le modèle n’est pas quantifié :
    - Ecrire la matrice input\_
    - Ecrire quant\_ = false
    - Ecrire la matrice output\_
  + Si le modèle est quantifié :
    - Ecrire la matrice qinput\_
    - Ecrire quant\_ = false
    - Ecrire la matrice qoutput\_

6. Entrainer les matrices input\_ et output\_

void FastText.trainThread(threadId)

* Ouvrir le fichier d’entrainement au chemin args\_.input
* Avancer dans le fichier jusqu’à l’octet (thread id \* taille fichier) / (threads count)
* Encapsuler les matrices d’entrée, de sortie, et les arguments dans un objet de type Model, en passant le thread id comme seed pour le générateur de nombres aléatoires
* Extraire le nombre d’occurrences pour chacune des entrées du dictionnaire
  + Entrainement non supervisé : entrées de type word
  + Entrainement supervisé : entrées de type label

List<long> Dictionary.getCounts(entry\_type type)

* Parcourir l’ensemble des entrées du dictionnaire du type donné
* Ajouter dans une liste le nombre d’occurrences de chaque entrée

void Model.setTargetCounts(List<long> counts)

* Si la fonction de coût est « negative sampling » : initTableNegatives()
  + Initialiser une liste d’entiers appelée negatives, de taille Model.NEGATIVE\_TABLE\_SIZE (valeur par défaut 10 000 000)
  + Calculer la somme des racines carrées de chacun des nombres d’occurrences
  + Ajouter l’indice de chaque mot dans la liste negatives autant de fois que : (racine carrée du nombre d’occurrences du mot x NEGATIVE\_TABLE\_SIZE) / (somme des racines carrées des nombres d’occurrences de mots)
  + Mélanger les élements de la liste negatives
* Si la fonction de coût est « hierarchical softmax » : buildTree()
  + Initialiser une liste de nœuds (Node) appelée tree, de taille 2 x nombre d’outputs (= nombre de prédictions osz\_, = nombre de mots en non supervisé ou nombre de labels en supervisé), avec Node.parent/left/right = -1, Node.binary = false, Node.count = 1e15,
  + Initialiser chaque nœud d’indice entre 0 et nombre de labels avec Node.count = nombre d’occurrences du label en position i
  + Exemple de la suite de l’algorithme :

N0:1,N1:3,N2:5,N3:2,N4:4,N5:1e15,N6:1e15,N7:1e15,N8:1e15,N9:1e15

leaf=4, node=5

Pour i=5

Pour j=0

Si N[leaf=4]=4 < N[node=5]=1e15

mini[j=0] = leaf = 4  
 leaf = 3

Pour j=1

Si N[leaf=3]=2 < N[node=5]=1e15

mini[j=1] = leaf = 3

leaf = 2

N[5].left = mini[0] = 4

N[5].right = mini[1] = 3

N[5].count = N[4].count + N[3].count = 4+2 = 6

N[4].parent = 5

N[3].parent = 5

N[3].binary = true

Pour i=6

Pour j=0

Si N[leaf=2]=5 < N[node=5]=6

mini[j=0] = leaf = 2  
 leaf = 1

Pour j=1

Si N[leaf=1]=3 < N[node=5]=6

mini[j=1] = leaf = 1

leaf = 0

N[6].left = mini[0] = 2

N[6].right = mini[1] = 1

N[6].count = N[2].count + N[1].count = 5+3 = 8

N[2].parent = 6

N[1].parent = 6

N[1].binary = true

Pour i=7

Pour j=0

Si N[leaf=0]=1 < N[node=5]=6

mini[j=0] = leaf = 0  
 leaf = -1

Pour j=1

Si leaf <0

mini[j=1] = node = 5

node = 6

N[7].left = mini[0] = 0

N[7].right = mini[1] = 5

N[7].count = N[0].count + N[5].count = 1+6 = 7

N[0].parent = 7

N[5].parent = 7

N[5].binary = true

Pour i=8

Pour j=0

Si leaf <0

mini[j=0] = node = 6

node = 7

Pour j=1

Si leaf <0

mini[j=1] = node = 7

node = 8

N[8].left = mini[0] = 6

N[8].right = mini[1] = 7

N[8].count = N[6].count + N[7].count = 8+7 = 15

N[6].parent = 8

N[7].parent = 8

N[7].binary = true

Pour i=9

Pour j=0

Si leaf <0

mini[j=0] = node = 8

node = 9

Pour j=1

Si leaf <0

mini[j=1] = node = 9

node = 10

N[9].left = mini[0] = 8

N[9].right = mini[1] = 9

N[8].count = N[9].count + N[9].count = 15+1e15 = 1e15

N[8].parent = 9

N[9].parent = 9

N[9].binary = true

Résultat :

N9-1e15

N8-15(b)

N6-8 N7-7(b)

N2-5 N1-3(b) N0-1 N5-6(b)

N4-4 N3-2(b)

>> SUITE de trainThread()

* Initialiser localTokenCount à zero
* Répéter tant que le compteur de tokens partagé entre tous les threads (tokenCount) est inférieur à : nombre de passes (args\_.epoch) x nombre de tokens dans le fichier d’entrainement
  + Calculer l’avancement = tokenCount / (args\_.epoch \* ntokens)
  + Diminuer learning rate de manière proportionelle à l’avancement :   
    lr = args\_.lt \* (1 - progress)
  + Lire une ligne du fichier d’entrainement
  + Incrémenter localTokenCount du nombre de tokens lus sur cette ligne

int Dictionary.getLine(sr, words, labels)

* + Initialiser une liste vide de hashes pour les mots word\_hashes
  + Si on arrive au bout du fichier d’entrainement, recommencer au début
  + Lire mot par mot le contenu de la ligne
    - Calculer le hash du mot
    - Retrouver la position du mot dans le dictionnaire
    - Si le mot n’est pas encore dans le dictionnaire, ajouter son hash à la liste word\_hashes et passer au mot suivant, sinon exécuter les opérations suivantes
    - S’il s’agit d’un mot, et si un tirage au sort ne décide pas d’ignorer le mot (en apprentissage non supervisé uniquement, avec une probabilité liée à la valeur précalculée pour le mot dans pdiscard)
      * Ajouter l’indice de position du mot dans la liste words
      * Aouter le hash du mot dans la liste word\_shashes
    - S’il s’agit d’un label
      * Ajouter l’indice du label - décalé du nombre de mots – dans la liste labels
    - Sortir de la boucle après avoir traité le mot de fin de ligne </s>, ou (en mode non supervisé seulement) lorsque le nombre de tokens lus dans la boucle est supérieur à la constante MAX\_LINE\_SIZE (1024)
  + En apprentissage supervisé, ajouter les word Ngrams – addWordNgrams()
    - Parcourir la liste des hashes des mots
    - Si le paramètre wordNgram >= 2
      * Calculer un hash combiné du hash courant avec la liste des hash des wordNgram-1 mots suivants
      * Un tableau pruneidx\_ est créé dans le dictionnaire si on a appliqué la méthode quantize() avec un paramètre cutoff > 0
        + FastText.selectEmbeddings

Calcul de la norme l2 de chacune des lignes de la matrice input\_

Création d’une liste d’entiers 0 1 2 3 4 … reprenant tous les indices de lignes de la matrice input\_

Trier cette liste d’indices par ordre décroissant de la norme l2 du vecteur correspondant, le mot EOS étant considéré le plus grand de tous

Ne conserver que les indices des cutoff premiers mots (ou ngrams), par ordre de d’importance décroissante

* + - * + Dictionary.prune()

Retrier la liste d’indices de mots (hors ngrams) par ordre croissant de l’indice

Pour chaque ngram conservé

pruneidx\_[ngram hash – nwords\_] = j++

ajout du hash de ngram dans la liste d’indices

Décalage des mots conservés dans le tableau words\_, réduction de la taille du tableau words\_

* + - * Si le hash produit est trouvé dans pruneidx\_, on remplace le hash par pruneidx\_[hash]
      * Ajouter dans la liste des indexes : nwords\_ + hash des words ngrams

Traitement d’une ligne de mots et de labels : au final, la liste words (line) contient

+ indices de position des mots de la phrase dans le dictionnaire

+ hash des wordNgrams (si wordNgrams >= 2), décalés du nombre de mots dans le dictionnaire, hash remplacé par un compteur si on tombe sur une valeur d’un hash de subword conservé après quantize + cutoff

SI supervised

* Choix d’un label au hasard parmi ceux de la ligne
* Mise à jour du modèle pour : la liste des indices de mots de la ligne, le label sélectionné, learning rate – Model.update()
  + computeHidden
    - hidden = moyenne ( lignes de la matrice wi\_ ) pour les indices de mots (et hashes de wordNgrams) de la liste
  + si paramètre loss = negative sampling
    - initialiser loss et gradient à zéro
    - exemple positif
      * score = sigmoid ( vecteur hidden \* ligne de la matrice wo\_ correspondant à l’indice du label )
      * alpha = learning rate \* (1 - score)
      * grad\_ = alpha \* ( ligne de la matrice wo\_ correspondant à l’indice du label)
      * loss = - log(score)
    - exemples négatifs : répéter autant de fois qu’indiqué par le paramètre neg
      * récupérer un indice de label différent de la cible, selon une probabilité égale à la racine carrée de la fréquence de chaque label dans les exemples
      * score = sigmoid ( vecteur hidden \* ligne de la matrice wo\_ correspondant à l’indice du label )
      * alpha = - learning rate \* score
      * grad\_ = alpha \* ( ligne de la matrice wo\_ correspondant à l’indice du label)
      * loss = - log(1 - score)
  + si paramètre loss = hierarchical softmax
    - initialiser gradient à zéro
    - bool[] binaryCode = codes[indice du label]
    - int[] pathToRoot = paths[indice du label]
    - Pour chaque indice étape de pathToRoot : loss += binaryLogistic(…)
      * score = sigmoid ( vecteur hidden \* ligne de la matrice wo\_ correspondant à l’indice du label )
      * alpha = -learning rate \* (binaryCode[i] - score)
      * grad\_ = alpha \* ( ligne de la matrice wo\_ correspondant à l’indice du label)
      * loss = binaryCode[i] ? -log(score) : - log(1 - score)
  + si paramètre loss = softmax
    - initialiser gradient à zéro
    - output = matrice wo\_ x vecteur hidden
    - max = nombre maximum du vecteur output
    - Pour chaque label i :
    - output[i] = exp(output[i] – max)
    - z = somme des output[i]
    - output[i] = output[i] / z
    - label attendu
      * alpha = learning rate \* (1 – output[i])
    - Label non attendu
      * alpha = - learning rate \* output[i]
    - grad\_ = alpha \* ( ligne de la matrice wo\_ correspondant à l’indice du label)
    - ( ligne de la matrice wo\_ correspondant à l’indice du label) += alpha \* hidden
    - Loss = -log(output\_[indice du label])
  + nexamples\_++
  + si apprentissage supervisé
    - diviser le vecteur de gradient par le nombre d’indices dans la liste de mots
  + pour chaque indice de mot de la ligne
    - ajouter à la ligne de la matrice wi\_ correspondant à l’indice le vecteur de gradient grad\_

SI cbow

Pour chaque mot de la ligne :

* bow = liste des indices de mots et sous-mots qui représentent le contexte
* choix aléatoire d’une taille de fenêtre de contexte entre 1 et le paramètre ws
* Parcours de mots en position -taille de fenêtre / +taille de fenêtre, à l’exclusion du mot central
* Ajout dans bow de dict.getSubwords pour chacun des mots du contexte
* Mise à jour du modèle pour bow (cf ci-dessus)

SI skipgram

Pour chaque mot de la ligne :

* ngrams = dict.getSubwords pour le mot de la ligne
* choix aléatoire d’une taille de fenêtre de contexte entre 1 et le paramètre ws
* Parcours de mots en position -taille de fenêtre / +taille de fenêtre, à l’exclusion du mot central
  + Mise à jour du modèle pour : input = ngrams, target = mot du contexte(cf ci-dessus)

FIN SI

* Si localTokenCount est supérieur à lrUpdateRate (val par défaut 100 tokens)
  + Ajout thread-safe de localTokenCount au tokenCount partagé entre les différents threads d’entrainement
  + Réinitialisation de localTokenCount
  + Affichage d’une info d’avancement si le paramètre verbose > 1
* Lorsque tokenCount atteint ntokens \* epochs
  + Affichage dans le thread 0 d’une unique info avec la valeur finale du coût (loss) si le paramètre verbose > 0

RETOUR dans la méthode Train()

* Encapsulation des matrices input\_, output\_, args\_ dans une classe Model
* Enregistrement du modèle cf FastText.saveModel()
* Si apprentissage non supervisé
* Enregistrement des vecteurs - FastText.saveVectors()
  + - Nombre de mots dans le dictionnaire, espace, dimension des vecteurs
    - Mot sous forme de chaine de caractères, liste des valeurs float qui constituent le vecteur de ce mot dans le dictionnaire
      * Récupérer la liste des indices des ngrams qui constituent le mot – Dictionary.getSubwords()
        + Si le mot se trouve dans le dictionnaire

Retourner la liste d’indices de ngrams stockée dans l’entrée du dictionnaire

* + - * + Si le mot ne se trouve pas dans le dictionnaire

Encadrer le mot par < et >

Créer la liste des indices de ngrams du mot encadré – cf Dictionary.computeSubwords()

* + - * Constituer un vecteur par addition des lignes correspondant à chaque indice de ngram dans la matrice input\_
      * Diviser ce vecteur par le nombre de ngrams pour en calculer la moyenne
  + Si argument saveOutput > 0, enregistrement de la matrice output\_ - FastText.saveOutput()
    - Nombre de mots dans le dictionnaire, espace, dimension des vecteurs
    - Mot sous forme de chaine de caractères, liste des valeurs float qui constituent le vecteur de ce mot dans le dictionnaire
      * Récupérer l’indice du mot dans le dictionnaire
      * Renvoyer la ligne correspondant à cet indice dans la matrice output\_

7. Prédire un résultat : FastText.predict

dict\_.getLine => words

model\_.predict(words, k)

* hidden = moyenne les lignes de la matrice wi\_ correspondant aux indices de mots passés dans words
* SI Hierarchical Softmax : Model.dfs
  + Départ du nœud racine
  + Si noeud feuille, ajout du score et du noeud dans la liste triée
  + Sinon parcours left-first de l’ensemble de l’arbre :
    - F = sigmoid ( hidden \* ligne de wo\_ en position [indice du nœud – nombre de labels]) {car en cas de hierarchical softmax, on a appris dans la matrice wo\_ la probabilité des nœuds non feuilles de l’arbre, pas celle des labels}
    - Accumulation dans score par addition du Log de la prédiction ou (1 – la prédiction) {car le produit des probabilités au fil du parcours des noeuds est équivalent à la somme des logs des probabilités}
* SINON : Model : findKBest
  + Output = wo\_ \* h idden
  + Softmax : output[i] = exp(output[i] - max) / sum(exp(output[i] - max))  
    (voir ci-dessous : The max trick when computing softmax)
  + Heap est une liste triée par ordre décroissant du log de output[i]
  + On parcourt toutes les valeurs de output[i], on conserver les k plus grandes dans heap
  + Une entrée de heap contient : (log(output[i]), i)  
    (on imagine que le log facilite la comparaison des très faibles probabilités entre elles)

8. Trouver les mots les plus proches : FastText.nn

precomputeWordVectors(wordVectors)

getVector(queryVec)

findNN(queryVec, k)

* Pour chacun des mots du dictionnaire :
  + Récupérer le vecteur du mot
  + Calculer le dot product de queryVec avec le vecteur du mot et diviser par la norme de queryVec (dotproduct(a,b)=norm(a)\*norm(b)\*cos(angle dans le plan défini par les deux vecteurs))
  + Ajouter le mot trié par score décroissant dans une liste
* Parcourir et retrouver les k premiers mots de la liste triée (sauf le mot query)

9. Compresser le modèle : FastText.quantize

loadModel()

Si le paramètre cutoff est > 0 et < nombre de mots dans le dictionnaire :

* selectEmbeddings
  + Calculer la norme de chacun des vecteurs des mots
  + Initialiser une liste contenant les indices de 0 à (nombre de mots – 1)
  + Trier la liste des indices de mots par ordre décroissant de la norme de leurs vecteurs, en plaçant [End of sentence] en tête
  + Supprimer tous les indices en position supérieure à cutoff
  + Retourner la liste des indices de mots ou ngrams à conserver
* Dictionary.prune()
  + Partager la liste des indices de mots à conserver en liste d’indices de mots et liste d’indices de ngrams
  + Trier la liste d’indice d’indices de mots par indices croissants
  + Créer un tableau pruneidx qui fait correspondre à chaque indice de position original des ngrams conservés (déplacé du nombre de mots) son nouvel indice de position (de 0 à nombre de ngrams conservés - 1)
  + Réinitialiser toutes les entrées du tableau word2int à -1
  + Parcourir tous les mots et ngrams du dicitonnaire original : déplacer les mots conservés aux premières places du tableau dans l’ordre où ils apparaissent à l’origine, mettre à jour l’entrée word2int\_ correspondante
  + Recalcul du nombre de mots et des tailles de tableaux
* Créer une matrice ninput de taille cutoff, avec le sous-ensemble des lignes de input original qui sont conservées, remplacer input par ninput
* Si le paramètre retrain est vrai
  + Ré-exécuter trainThread()

Dans tous les cas, continuer avec :

* qinput\_ = new QMatrix(input, paramètre dsub, paramètre qnorm)
* Si paramètres quout est vrai : qoutput\_ = new QMatrix(input, 2, paramètre qnorm)

New QMatrix / QMatrix.quantize :

* Initialiser un tableau de bytes codes\_ de taille m \* ((n + dsub – 1) / dsub) { = nombre maximal de sous-vecteurs de taille dsub qu’on peut extraire d’un vecteur de taille n}
* Si le paramètre qnorm est vrai :
  + initialiser un tableau de bytes norm\_codes\_ de taille m \* 1
  + calculer dans un vecteur les normes de toutes les lignes de la matrice d’origine
  + diviser toutes les lignes de la matrice par la norme de la ligne
  + npq\_ ProductQuantizer.train()
  + npq\_ ProductQuantizer.compute\_codes(norm\_codes\_)
* pq\_ ProductQuantizer.train()
* pq\_ ProductQuantizer.compute\_codes(norm\_codes\_)

ProductQuantizer.train(n = m\_)

* nbits\_ = nombre de bits alloués pour désigner un centroid pour un sous-vecteur donné
* ksub\_ = 1 << nbits = nombre de centroids qui peuvent être calculés pour chaque sous-vecteur
* max\_points\_per\_cluster\_ = 256 = nombre maximum de points qu’on peut rattacher à un centroid => max\_points\_ = 256\*256 = 65 536 = nombre maximum d’exemples qui peuvent être considérés pour l’entrainement
* np = Math.Min(n, max\_points\_ =) {np strictement inférieur à n ssi le dictionnaire est d’un taille totale supérieure à 65 536}
* Répéter autant de fois que le parameter threadCount :
* trainThread(int threadIndex, bool isLastThreadIndex, int kmeansPerThread, float[] x, int n, int np)
* startIndex = threadIndex \* kmeansPerThread
* lastIndexExcluded = isLastThreadIndex ? nsubq\_ : (threadIndex + 1) \* kmeansPerThread
* Initialiser un tableau xslice de taille np \* dsub\_
* Pour les index de startIndex à lastIndexEcluded :
  + Copier dans xslice les valeurs de d = (dsub\_ ou lastdsub\_) colonnes en position index \* dsub\_ (indice de colonnes), pour toutes les lignes si n = np, ou pour un sous-ensemble aléatoire de np lignes si np < n
  + ProductQuantizer.kmeans(xslice, centroids[(index \* ksub\_) \* dsub\_], np, d)
    - Initialiser ksub\_ centroids à partir de points distincts pris au hazard dans xslice
    - Initialiser un tableau de bytes : codes de taille np = nombre de points exemples, qui sert pendant le travail d’entrainement, mais qu’on réinitialise et ne renvoie pas en retour de la fonction
    - Répéter niter\_ = 25 fois
      * Estep { !!PERF !! : dans cette méthode elle-même exécutée dim / dsub = 50 fois, on exécute 25 \* 65 536 \* 256 = 420 millions de calculs distL2, où distL2 comprend dsub \* (2 accès de tableau, 5 additions, 1 multiplication), soit au total 168 milliards d’opérations 🡺 A vectoriser !!}
        + Parcourir chacun des exemples considérés
        + Ré-assigner un centroid à chaque exemple

dis = distL2(exemple | premier centroid)

initialiser le code pour l’exemple à l’indice de centroid 0

Parcourir les centroids du deuxième au dernier

disij = distL2(exemple | centroid courant)

Si disij < dis : code pour l’exemple = indice du centroid courant, dis = disij

* + - * MStep
        + Réinitialisation de la valeur des centroids
        + Parcourir chacun des exemples considérés

Récupérer dans codes l’indice du centroid affecté à l’exemple

Ajouter le vecteur exemple au centroid, augmenter le compteur du nombre de points affectés au centroid

* + - * + Parcourir chacun des centroids

Si le nombre d’exemples affectés au centroid est >0 : diviser le centroid par le nombre de points {moyenne des vecteurs affectés}

* + - * + Parcourir chacun des centroids

Si le nombre d’éléments affectés au centroid = 0 {dans ce cas le centroid est nul pour l’instant} :

Recherche aléatoire d’un indice de centroid tel que : nombre d’exemples affectés > (nombre aléatoire entre 0 et 1) \* (nombre d’exemples – nombre de centroids) + 1

Copie du centroid choisi aléatoirement dans le centroid nul

Retraits / ajouts compensés d’une petite valeur (eps\_ = 1e-7) sur chacune des cordonnées des deux centroids considérés

Nombre d’élements partagés en deux entre les deux centroids

ProductQuantizer.compute\_codes()

* Parcourir l’ensemble des exemples
  + Parcourir l’ensemble des sous-vecteurs
    - Récupérer les centroids pour le sous-vecteur
    - Appeler assign\_centroid pour le sous-vecteur

10. Faire des opérations sur une matrice compressée :

* addToVector(Vector x, int t)
  + norm = 1F, if (qnorm\_) norm = npq\_.get\_centroids(0, norm\_codes\_[t])[0]
  + Parcourir les sous-vecteurs
    - Récupérer le centroid correspondant au code du sous-vecteur
  + Reconstruire le vecteur en concaténant les centroids

[FNV Hash](http://www.isthe.com/chongo/tech/comp/fnv/)

[Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent](https://arxiv.org/pdf/1106.5730.pdf)

**Algorithmic steps for k-means clustering**

Let  X = {x1,x2,x3,……..,xn} be the set of data points and V = {v1,v2,…….,vc} be the set of centers.

1) Randomly select ‘c’ cluster centers.

2) Calculate the distance between each data point and cluster centers.

3) Assign the data point to the cluster center whose distance from the cluster center is minimum of all the cluster centers..

4) Recalculate the new cluster center

5) Recalculate the distance between each data point and new obtained cluster centers.

6) If no data point was reassigned then stop, otherwise repeat from step 3).

**Product Quantization**

Another way to see thisistoconsiderthatPQsplitsagivenvector x into k subvectors xi, i = 1...k,eachofdimension d/k: x = [x1 ...xi ...xk], and quantizes each sub-vector using a distinct k-means quantizer. Each subvector xi is thus mapped to the closest centroid amongst 2b centroids, where b is the number of bits required to store the quantization index of the subquantizer, typically b = 8. The reconstructed vector can take 2kb distinct reproduction values, and is stored in kb bits.

In practice, the vector estimate ˆ x is trivially reconstructed from the codes, i.e., from the quantization indexes, by concatenating these centroids. ThetwoparametersinvolvedinPQ,namelythenumberofsubquantizers k andthenumberofbits b per quantization index, are typically set to k ∈ [2,d/2], and b = 8 to ensure byte-alignment.

using PQ just before the last layer incurs a very limited loss in accuracy when combined with a support vector machine.

Inthecontextoftextclassiﬁcation,thenormsofthevectorsarewidelyspread,typicallywitharatio of 1000 between the max and the min. Therefore kmeans performs poorly because it optimizes an absolute error objective, so it maps all low-norm vectors to 0. A simple solution is to separate the norm and the angle of the vectors and to quantize them separately. This allows a quantization with no loss of performance, yet requires an extra b bits per vector.

Memory savings with PQ. In practice, the bottom-up PQ strategy offers a compression factor of 10 without any noticeable loss of performance. Without re-training, we notice a drop in accuracy between 0.1% and 0.5%, depending on the dataset and setting

**The max trick when computing softmax**

<https://jamesmccaffrey.wordpress.com/2016/03/04/the-max-trick-when-computing-softmax/>

In practice, calculating softmax values can go wrong if any x value is very large — the exp() of even a moderate-magnitude positive number can be astronomically huge, which makes the scaling sum huge, and dividing by a huge number can cause arithmetic computation problems.

A trick to avoid this computation problem is subtract the largest x value from each x value. It turns out that the properties of the exp() function give you the same result but you avoid extreme large numbers.

<https://stackoverflow.com/questions/17187507/why-use-softmax-as-opposed-to-standard-normalization>

The values of q\_i represent log-likelihoods. In order to recover the probability values, you need to exponentiate them.

One reason that statistical algorithms often use log-likelihood loss functions is that they are more numerically stable: a product of probabilities may be represented be a very small floating point number. Using a log-likelihood loss function, a product of probabilities becomes a sum.

Another reason is that log-likelihoods occur naturally when deriving estimators for random variables that are assumed to be drawn from multivariate Gaussian distributions. See for example the Maximum Likelihood (ML) estimator and the way it is connected to least squares.

> Maximum likelihood estimation

The method of maximum likelihood corresponds to many well-known estimation methods in statistics. For example, one may be interested in the heights of adult female penguins, but is unable to measure the height of every single penguin in a population due to cost or time constraints. Assuming that the heights are normally distributed with some unknown mean and variance, the mean and variance can be estimated with MLE while only knowing the heights of some sample of the overall population. MLE would accomplish this by taking the mean and variance as parameters and finding particular parametric values that make the observed results the most probable given the model.

> Normal distribution

In probability theory, the normal (or Gaussian) distribution is a very common continuous probability distribution. Normal distributions are important in statistics and are often used in the natural and social sciences to represent real-valued random variables whose distributions are not known.[1][2]

The normal distribution is useful because of the central limit theorem. In its most general form, under some conditions (which include finite variance), it states that averages of samples of observations of random variables independently drawn from independent distributions converge in distribution to the normal, that is, become normally distributed when the number of observations is sufficiently large. Physical quantities that are expected to be the sum of many independent processes (such as measurement errors) often have distributions that are nearly normal.[3] Moreover, many results and methods (such as propagation of uncertainty and least squares parameter fitting) can be derived analytically in explicit form when the relevant variables are normally distributed.

**Logistic regression**

**Hypothesis function**

Logistic regression is used when the variable *y* that is wanted to be predicted can only take discrete values (i.e.: classification).

Considering a binary classification problem (*y* can only take two values), then having a set of parameters *θ* and set of input features *x*, the hypothesis function could be defined so that is bounded between [0, 1], in which *g()* represents the sigmoid function:

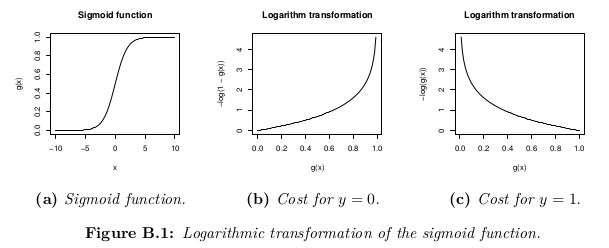
[enter image description here](https://i.stack.imgur.com/usPHR.png)

This hypothesis function represents at the same time the estimated probability that *y = 1* on input *x* parameterized by *θ*:

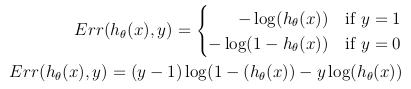
[enter image description here](https://i.stack.imgur.com/4m3LO.png)

**Cost function**

The cost function represents the optimization objective.

[](https://i.stack.imgur.com/ufmSH.png)

Although a possible definition of the cost function could be the mean of the Euclidean distance between the hypothesis *h\_θ(x)* and the actual value *y* among all the *m* samples in the training set, as long as the hypothesis function is formed with the sigmoid function, this definition **would result in a non-convex cost function**, which means that a local minimum could be easily found before reaching the global minimum. In order to ensure the cost function is convex (and therefore ensure convergence to the global minimum), **the cost function is transformed using the logarithm of the sigmoid function**.

[](https://i.stack.imgur.com/9C6w5.png)

This way the optimization objective function can be defined as the mean of the costs/errors in the training set:

[enter image description here](https://i.stack.imgur.com/LHRDJ.png)

**Hierarchical softmax**

<http://ruder.io/word-embeddings-softmax/index.html#hierarchicalsoftmax>

<https://arxiv.org/pdf/1310.4546.pdf>

2.1 Hierarchical Softmax

A computationally efﬁcient approximation of the full softmax is the hierarchical softmax. In the context of neural network language models, it was ﬁrst introduced by Morin and Bengio [12]. The main advantage is that instead of evaluating W output nodes in the neural network to obtain the probability distribution, it is needed to evaluate only about log2(W) nodes. The hierarchical softmax uses a binary tree representation of the output layer with the W words as its leaves and, for each node, explicitly represents the relative probabilities of its child nodes. These deﬁne a random walk that assigns probabilities to words.

More precisely, each word w can be reached by an appropriate path from the root of the tree. Let n(w,j) be the j-th node on the path from the root to w, and let L(w) be the length of this path, so n(w,1) = root and n(w,L(w)) = w. In addition, for any inner node n, let ch(n) be an arbitrary ﬁxed child of n and let [[x]] be 1 if x is true and -1 otherwise. Then the hierarchical softmax deﬁnes p(wO|wI) as follows:

p(w|wI) =

L(w)−1 Y j=1

σ[[n(w,j + 1) = ch(n(w,j))]] · v′ n(w,j)

⊤vwI (3)

where σ(x) = 1/(1 + exp(−x)). It can be veriﬁed thatPW w=1 p(w|wI) = 1. This implies that the cost of computing logp(wO|wI) and ∇logp(wO|wI) is proportional to L(wO), which on average is no greater than logW. Also, unlike the standard softmax formulation of the Skip-gram which assigns two representations vw and v′w to each word w, the hierarchical softmax formulation has one representation vw for each word w and one representation v′ n for every inner node n of the binary tree.

The structure of the tree used by the hierarchical softmax has a considerable effect on the performance. Mnih and Hinton explored a number of methods for constructing the tree structure and the effect on both the training time and the resulting model accuracy [10]. In our work we use a binary Huffman tree, as it assigns short codes to the frequent words which results in fast training. It has been observed before that grouping words together by their frequency works well as a very simple speedup technique for the neural network based language models [5, 8].

<http://www.trevorsimonton.com/blog/2016/12/15/huffman-tree-in-word2vec.html>

**Huffman tree**

Le principe du codage de Huffman repose sur la création d'une structure d'[arbre](https://fr.wikipedia.org/wiki/Arbre_enracin%C3%A9) composée de nœuds.

On recherche tout d'abord le nombre d'[occurrences](https://fr.wiktionary.org/wiki/occurrence) de chaque caractère. Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids égal à son nombre d'occurrences.

L'arbre est créé suivant un principe simple : on associe à chaque fois les deux nœuds de plus faibles poids, pour donner un nouveau nœud dont le poids équivaut à la somme des poids de ses fils. On réitère ce processus jusqu'à n'en avoir plus qu'un seul nœud : la racine. On associe ensuite par exemple le code 0 à chaque embranchement partant vers la gauche et le code 1 vers la droite .

Pour obtenir le [code binaire](https://fr.wikipedia.org/wiki/Syst%C3%A8me_binaire) de chaque caractère,on remonte l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code un 0 ou un 1 selon la branche suivie. La phrase « this is an example of a huffman tree » se code alors sur 135 bits au lieu de 288 bits (si le codage initial des caractères tient sur 8 bits).