Rapport Projet 2

Leader : Yannick Brenning, Follower : Kossi Kossivi

19 novembre 2024

1 Introduction

Ce projet, étudie de quelques manières Word2Vec, un groupe de modèles pour la géneration des plongements de mots statiques publié pour la première fois par Mikolov et al. [2013a]. Dans ce document, deux architectures principales : CBOW (*continuous bag-of-words*), et *Skip-gram* sont présentées. Dans ce cas, j'examinerai le modèle *Skip-gram with negative sampling*, qui était introduit à la suite du document original par Mikolov et al. [2013b]. Les plongements générés par Word2Vec devraient être proches si les mots apparaîssent dans les contextes similaires, comme "bicyclette" et "vélo".

Généralement, le modèle Skip-gram vise à prédire les mots qui apparaissent dans le contexte d'un certain mot cible. Nous entraînons un classifieur binaire sur les mots d'un corpus tel qu'il calcule la probabilité qu'un mot c apparaît dans le contexte d'un mot cible m, écrit $P(+|m, c)$. De manière analogue pour les mots hors du contexte de m, nous notons $P(-|m, c)$. Ces probabilités sont calculées en utilisant la fonction sigmoïde et le produit scalaire des vecteurs $\mathbf{m}$ et $\mathbf{c}$ (voir les équations 1), qui sont les plongements de m et c. Les vecteurs sont stockés dans des matrices $\mathbf{M}$ et $\mathbf{C}$ tel que $\mathbf{M}$ contient les plongements pour chaque mot à la fin de l'apprentissage.

$$\begin{aligned} P(+|m, c) &= \frac{1}{1 + \exp(-\mathbf{m} \cdot \mathbf{c})} \\ P(-|m, c) &= \frac{1}{1 + \exp(\mathbf{m} \cdot \mathbf{c})} \end{aligned} \tag{1}$$

Après la présentation du modèle initial dans la section suivante, nous explorons quelques pistes concernant le sujet de ce projet.

La première piste est l'amélioration des performances. D'abord, nous comparons nos résultats avec le résultat de référence, mais il reste quelques encore des possibilités d'amélioration. Notamment, Mikolov et al. [2013b] propose une méthode d'échantillonnage des exemples négatifs, entre autres. En ce qui concerne le résultat de base, l'objectif d'atteindre des taux de réussite proches de ceux de la version originale de Word2Vec pourrait être intéressant.

Pour la deuxième piste, nous explorons les analogies, c'est-à-dire les associations d'idées entre les mots. Le principe de cette tâche est de faciliter les calculations simples avec les plongements de mots, par exemple `vector(roi) - vector(homme) + vector(femme)` $\approx$ `vector(reine)`. Cette propriété pourrait présenter un intérêt pour montrer que le modèle est capable d'encoder les relations linguistiques différentes.

2 Modèle initial

Pour générer les plongements de mots avec un modèle de Word2Vec, nous avons d'abord besoin des données d'apprentissage. Dans le cadre de ce projet, nous utilisons "Le Comte de Monte-Cristo" comme corpus pour la création des données qui vont entraîner le modèle.

Nous commencons par tokeniser les mots du corpus, c'est-à-dire diviser le texte dans les *tokens* afin de structurer et simplifier le contenu pour le modèle. Sur la base de ces tokens, nous créons un vocabulaire avec lequel nous pouvons construire les les exemples positifs et négatifs.

Pour la construction des exemples positifs, nous sélectionnons les mots qui apparaissent dans le contexte de chaque mot m dans le vocabulaire. La paramètre L donne la taille du contexte, c'est-à-dire le nombre de mots précédant et suivant m qui sont sélectionnées. Pour les exemples négatifs, l'approche naïve prend k mots du lexique de façon aléatoire.

$$L = -[\log \sigma(\mathbf{m} \cdot \mathbf{c}_{pos}) + \sum_{i=1}^{k} \log \sigma(-\mathbf{m} \cdot \mathbf{c}_{neg_i})] \tag{2}$$

En utilisant ces données et la descente du gradient de la fonction de perte donné par l'équation 2, nous réalisons le modèle de base. Un example positif et les k exemples négatifs correspondants sont traités comme un *minibatch*, sur lequel nous calculons le gradient pour la mise à jour des paramètres du modèle (équations 3).

$$
\begin{aligned}
\mathbf{c}_{pos}^{t+1} &= \mathbf{c}_{pos}^{t} - \eta[\sigma(\mathbf{m} \cdot \mathbf{c}_{pos}^{t}) - 1]\mathbf{m} \\
\mathbf{c}_{neg}^{t+1} &= \mathbf{c}_{neg}^{t} - \eta[\sigma(\mathbf{m} \cdot \mathbf{c}_{neg}^{t})]\mathbf{m} \\
\mathbf{m}^{t+1} &= \mathbf{m}^{t} - \eta([\sigma(\mathbf{m}^{t} \cdot \mathbf{c}_{pos}^{t}) - 1]\mathbf{c}_{pos}^{t} + \sum_{i=1}^{k}[\sigma(\mathbf{m}^{t} \cdot \mathbf{c}_{neg_i}^{t})]\mathbf{c}_{neg_i}^{t})
\end{aligned}
\tag{3}
$$

Nous stockons la $\mathbf{M}$, qui contient les plongements de mots sous forme de vecteurs de colonne à l'issue de l'apprentissage, dans un fichier afin de l'évaluer sur un jeu d'évaluation. L'évaluation a lieu sur un ensemble de triplets de mots (m, m_+, m_-) de sorte que le sens de m_+ soit plus proche de m que m_- de m. Pour chaque triplet, nous comparons les plongements des mots pour déterminer lequel est plus proche de m. En comptant le nombre de triplets qui passent le test $\text{sim}(\mathbf{m}, \mathbf{m}_+) > \text{sim}(\mathbf{m}, \mathbf{m}_-)$, nous calculons le taux de réussite pour le modèle. Pour les comparaisons entre deux plongements de mot, nous utilisons la similarité cosinus comme mesure.

Comme notre modèle suit les spécifications de Mikolov et al. [2013a], la fenêtre du contexte change sa taille aléatoirement entre $< 1, L >$ pour chaque mot cible, ce qui accélère considérablement l'apprentissage. L'initialisation aléatoire des matrices suit d'ailleurs une distribution uniforme entre $[-\frac{0{,}5}{d}, \frac{0{,}5}{d})$, où d est la dimension des plongements.

Le point de comparaison sur 10 expériences est un taux de réussite moyen de 53,2% avec un écart type de 3,9 sur un configuration donné de paramètres. [1] Lors de nos expériences, les résultats que nous avons obtenus avec notre implémentation de base sont un taux de réussite moyen de 63.8% et un écart type de 2.2.

Concernant l'approche pour l'échantillonage des exemples négatifs, nous pouvons constater quelques limitations potentiels. Tout d'abord, nous avons la possibilité de tirer un exemple positif par accident, car la sélection s'éffectue sur l'ensemble du lexique. De plus, nous traitons tout les mots du vocabulaire de la même manière, ce qui n'est pas vraiment conforme à la réalité, où les fréquences et les niveaux d'informations contenues peuvent être très différentes d'un mot à l'autre. Par exemple, les mots comme `le` ou `et` sont très fréquents, pourtant ils peuvent ne pas contenir autant d'informations que quelques mots plutôt rares.

La même idée s'applique à la génération des exemples positifs, car la fenêtre contient souvent beaucoup de mots fréquents qui portent moins de

1. Dimension des plongements $d = 100$, taille de fenêtre $L = 2$, nombre d'exemples négatifs par exemple plositif $k = 10$, taux d'apprentissage $\eta = 0{,}1$, nombre d'itérations $n\_iter = 5$, nombre minimum d'occurences $minc = 5$.

signification que les mots moins fréquents. Mikolov et al. [2013b] mentionnent cela avec l'exemple du mot `France` : notre modèle Skip-gram bénéficie beaucoup plus de la cooccurrence avec `Paris` que la cooccurrence avec `la`.

Tous les deux limitations mentionées ci-dessus sont en rapport avec ma première piste à creuser, dans laquelle j'essayerai d'améliorer les résultats du modèle précédemment détaillé.

Un autre limitation est la taille du vocabulaire du corpus utilisé jusqu'à ce point. Pour s'attaquer au problème des analogies, j'utiliserai alors des plongements préexistants, entraînés sur le corpus plus large "French CoNLL17" par Fares et al. [2017]. Pour comparer, le corpus utilisé jusqu'à ce point contient un vocabulaire de 3,210 mots, alors que celui utilisé par Fares et al. [2017] contient 2,567,698 mots.

3 Amélioration des performances

Afin d'améliorer le modèle Word2Vec, Mikolov et al. [2013b] proposent deux méthodes principales qui concernent le processus d'échantillonnage, appelées *negative sampling* (échantillonage négatif) et *subsampling of frequent words* (sous-échantillonage des mots fréquents).

3.1 Description

Premièrement, nous aborderons une différente méthode pour la sélection des exemples négatifs. Jusqu'a maintenant, nous choisissons les exemples négatifs aléatoirement d'une distribution uniforme sur l'ensemble des mots dans le lexique.

Mikolov et al. [2013b] proposent une variation de la distribution d'unigramme en tant que distribution de probabilité pour le processus d'échantillonage, parfois appelé distribution *smooth* (lisse) d'unigramme [Levy et al., 2015]. Nous tirons donc un exemple négatif c_{neg} avec la probabilité $P(m)$ (voir l'équation 4), où $\#(m)$ designe le nombre d'occurences de m dans le corpus. Le document original indique que les meilleurs résultats ont été obtenus en augmentant la distribution à une puissance de $\alpha = 0.75$, ce qui augmente les probabilités des mots rares. Levy et al. [2015] démontrent ensuite les améliorations avec ce choix de paramètre, que nous utilisons aussi dans le cadre de l'expérience.

$$P(m) = \frac{\#(m)^\alpha}{\sum_{i=0}^{n} \#(m_i)^\alpha} \tag{4}$$

L'effet de cette distribution est tel que la probabilité des mots rares augmente, et la probabilité des mots fréquents diminue. PMI (Pointwise mutual information) est une mesure d'association entre les mots [Jurafsky, 2000], tel que les valeurs grandes indiquent des associations fortes. Levy and Goldberg [2014] montrent que le modèle Skip-gram décrit par Mikolov et al. [2013b] calcule implicitement des plongements qui se rapprochent des valeurs PMI entre les mots et les contextes, moins une constante. Selon Levy et al. [2015], l'objectif du smoothing est alors de réduire le biais du PMI en faveur des mots rares.

Avec cette changement, nous souhaitons donc améliorer la diversité des exemples négatifs sur lesquels nous entraînons le modèle afin d'améliorer le PMI et les performances.

La deuxième approche pour l'amélioration de Word2Vec est la sous-échantillonage des exemples positifs. Dans les grands corpus, on observe aussi que les mots fréquents commencent à apparaître dans beaucoup de contextes. Cependant, l'information fourni par les mots comme `le`, `elle`, ou `un` n'est pas très utile. Dans l'autre sens, les plongements genérés pour ces mots fréquents ne se changement pas beaucoup à partir d'un certain moment en raison du nombre considérable d'exemples d'entraînement.

Pour compenser le déséquilibre des mots fréquents, Mikolov et al. [2013b] proposent d'éliminer un exemple positif avec la probabilité donnée par l'équation 5, où $f(m)$ désigne la fréquence relative du mot m dans le corpus, et t est un paramètre *threshold* (seuil), typiquement 10^{-5}. Ce paramètre détermine le degré d'agressivité du sous-échantillonnage en ce qui concerne les mots avec des fréquences inférieures à t.

$$P(m) = 1 - \sqrt{\frac{t}{f(m)}} \tag{5}$$

En gardant à l'esprit ces approches, nous proposons les hypothèses suivantes :

1. Le negative sampling améliorera les résultats des modèles de base de Word2Vec.
2. En ajoutant le subsampling, on peut s'attendre à une amélioration des performances des modèle de base de Word2Vec.
3. L'utilisation des deux méthodes ensemble devrait permettre une amélioration des performances des modèles de base de Word2Vec.

3.2 Mise en œuvre

Pour clarifier la mise en œuvre des améliorations, nous exliquons d'abord brièvement la structure de l'implémentation.

Notre modèle Word2Vec est structuré principalement en deux programmes ; la construction des données d'entraînement et le processus d'entraînement lui-même, ce qui diffère de la plupart des approches, où l'apprentissage et le traitement des données ont lieu au même temps. La structure utilisée dans cette expérience vise à améliorer la clarté et la simplicité de l'implémentation.

La mise en œuvre de la nouvelle méthode d'échantillonage des exemples négatifs est assez simple. Apès avoir construit les exemples positifs, nous créeons une distribution unigramme lisse sur la lexique en utilisant l'équation 4. À l'aide de la bibliothèque `numpy`, nous pouvons effectuer k tirages sans remplacement selon cette distribution pour chaque exemple positif.

L'implémentation de l'échantillonage des exemples positifs est un peu différent, car il existe des façons différentes de procéder.

D'abord, il est intéressant de noter que le code de Word2Vec utilise une formule différente pour l'implémentation de cette méthode, à savoir $P(m) = \frac{f(m)-t}{f(m)} - \sqrt{\frac{t}{f(m)}}$, où $P(m)$ donne la probabilité de conserver le mot m.[2] La raison pour cette différence n'est pas expliqué dans le document originale, mais dans le cadre de cette expérience nous notons des légères améliorations du temps d'exécution avec cette formule. En plus, l'implémentation originale utilise un seuil de 10^{-3} et non 10^{-5}, ce qui nous suivons pour cette mise en œuvre.

En plus, il y a des manières différents concernant quand on effectue le sous-échantillonage : soit on élimine les mots en "glissant" la fenêtre sur le corpus (c'est-à-dire on enleve les mots du contexte), soit nous élimonons les mots avant de glisser la fenêtre. Avec la deuxième option, que nous utiliserons dans cette expérience, nous aurons effectivement quelques mots dans les contextes qui n'auraient pas été présents avant le sous-échontillonage. Levy et al. [2015] ont appelés ce type de sous-échantillonage "dirty" (sale) et ont constatés que les performances entre les deux options étaient comparables. Le processus de ce sous-échantillonage aura donc lieu avant la construction des exemples d'entraînement ; nous parcourons le corpus et gardons chaque mot avec la probabilité $P(m) = \frac{f(m)-t}{f(m)} - \sqrt{\frac{t}{f(m)}}$.

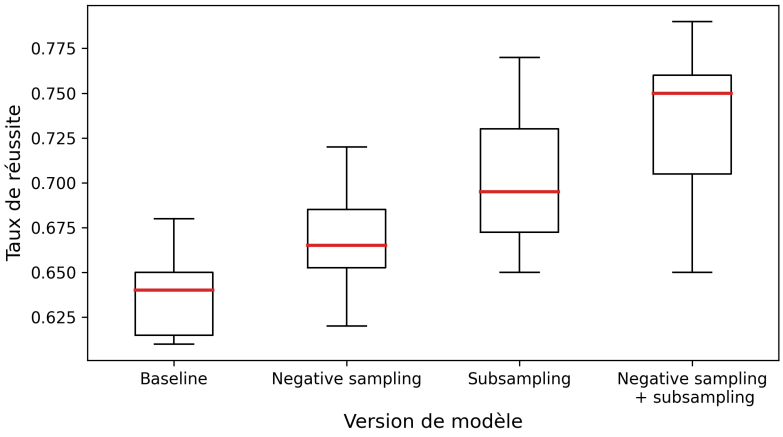

FIGURE 1 – Moyen taux de réussite sur 10 expériences avec les paramètres $d = 100$, $L = 2$, $k = 10$, $\eta = 0{,}1$, $n\_iter = 5$ et $minc = 5$ pour chaque variation de modèle.

3.3 Résultats

Figure 1 montre les résultats de l'expérience. On voit que tout les variations du modèle de base montrent des améliorations à différents degrés. Bien que les taux moyens s'améliorent, les variabilités entre les résultats de chaque modèle augmentent également.

Le modèle de base donne des résultats relativement cohérents vers 63%, ce qui est déja un amélioration de notre point de comparaison. Le negative sampling entraîne de légères améliorations, avec un taux de réussite moyen de 66,8% et un écart type de 2,8%. Pour cette expérience, nous pouvons alors confirmer notre première hypothèse.

Notamment, les taux de réussite moyens du modèle avec subsampling et du modèle avec les deux méthodes ensemble dépassent les 70% avec respectivement 70,2% (écart type de 3,9%) et 73,7% (écart type de 4,3).

Malgré des meilleurs résultats pour les taux de réussite moyens, les écarts types sont assez larges, ce qui peut indiquer des résultats légèrement incohérents. Toutefois, cela peut également résulter de la configuration de notre expérience, puisque nous n'avons suivi que 10 passages pour calculer les scores moyens.

Afin de mieux examiner ces résultats et de les améliorer, il serait possible de comparer les variations en fonction des différentes méthodes d'initialisation et des modifications du type de fenêtre. En plus, on pourrait

2. https ://code.google.com/archive/p/word2vec/source

expérimenter avec des options de tokenisation plus optimales, ainsi que ainsi que l'augmentation du nombre d'expériences, ce qui n'entre pas dans le cadre de ce travail.

4 Analogies

Comme mentionné précédemment, les plongements de mots permettent également de faire des calculs simples, comme `vector(roi) - vector(homme) + vector(femme)` $\approx$ `vector(reine)`. Pour examiner cette capacité, nous suivons la tâche posé par Mikolov et al. [2013a] avec des plongements préexistents de Fares et al. [2017].

4.1 Description

Pour examiner le raisonnement analogique des plongements de mots, nous devons utiliser les distances entre les vecteurs pour déterminer quel plongement est le plus proche du résultat de notre calcul. Pour un ensemble de plongements $V = \{e_1, \ldots, e_N\}$ et un plongement particulier $x \notin V$, nous cherchons trouver l'élément de V la plus proche de x (voire l'équation 6). [3]

$$\hat{e} = \underset{e \in V}{\operatorname{argmin}}\ d(e, x) \tag{6}$$

Comme indiqué en section 2, la taille du vocabulaire est très grande, ce qui rend les recherches naïves coûteuses. Afin de trouver les solutions plus efficacement, nous utilisons donc un arbre k-d, qui permet des recherches plus rapides dans les espaces à k dimensions. [Bentley, 1975].

Notre hypothèse pour la présente section est alors que l'utilisation des arbres k-d permettra des calculs d'analogies plus rapides que le parcours naïf de l'espace.

4.2 Mise en œuvre

Car les plongements existent déjà, la mise en œuvre se compose de trois parties principales : la lecture des plongements donnés, l'implémentation de l'arbre k-d, et la construction d'un jeu de données pour l'évaluation.

D'abord, la lecture des plongements a posé une difficulté à cause de la taille immense. Étant donné que ce projet est réalisé avec une infrastructure de calcul limitée, effectuer des opérations sur des données occupant $\approx$ 2,5 GB d'espace en memoire n'est pas possible. Vu que les plongements

3. Pour cette expérience, la mesure de la distance d est toujours euclidienne.

sont triés dans l'ordre décroissant de fréquence, nous utilisons d'abord les premières 500.000 plongements pour nos calculs.

Nous avons d'abord implémenté la structure d'un arbre *k-d* ainsi que l'algorithme de recherche selon la définition de Friedman et al. [1977], en utilisant une classe `KDTree` composée récursivement de nœuds `Node`.

Cette implémentation à entraîné des difficultés en ce qui concerne la mémoire vive disponible, car la construction implémenté dans ce cas nécessite $2 \cdot |V|$ stockés en mémoire. [4] Pour le cadre de cette expérience, nous utilisons alors 100.000 embeddings. Nous comparerons les durées d'exécution des recherches avec les arbres *k-d* à celles des recherches exhaustives (c'est-à-dire des parcours naïfs de l'espace).

Pour l'évaluation, nous creóns un ensemble de 100 analogies différentes en suivant le format `(homme, roi, femme, reine)`, ce qui signifie un calcul `vector(roi) - vector(homme) + vector(femme)` $\approx$ `vector(reine)`. Nous enregistrerons le durée d'exécution de la recherche du plus proche voisin pour le vecteur obtenu. Ensuite, nous comparerons le voisin trouvé avec le quatrième mot de la ligne, en comptant le nombre total des résultats "corrects". [5]

4.3 Résultats

Pour chaque façon de recherche, nous avons suivi les durées d'execution pour les calculs des analogies (c'est-à dire la recherche du plus proche voisin dans l'espace des plongements). Nous montrons les résultats en figure 2. Bien entendu, ces scores dépendent fortement de la machine et ne sont donc pas nécessairement représentatifs, mais ils serviront de points de référence généraux dans le contexte de cette expérience. Pour la

En rapport avec notre hypothèse, nous pouvons affirmer que les calculs sont légèrement plus rapides, mais seulement par une faible marge, et en limitant le nombre de plongements utilisés. Globalement, nous pouvons dire que l'application de cette implémentation de l'arbre *k-d* n'est pas nécessairement utile dans cette tâche spécifique.

Une raison pour ces résultats pourrait être un phénomène appelé *curse of dimensionality* (le fléau de la dimension) par Bellman [1961], qui s'applique nottement aux tâches de recherche des plus proches voisins. Comme règle génerale, on constate que pour des données en dimension k, le nombre de

4. La construction de l'arbre *k-d* s'effectue à partir de la liste de plongements, ce qui signifie qu'à la fin de la construction, nous avons au moins $2 \cdot |V|$ élélements stockés en memoire.

5. Bien entendu, ce nombre restera toujours le même, car les plongements et les analogies sont statiques.

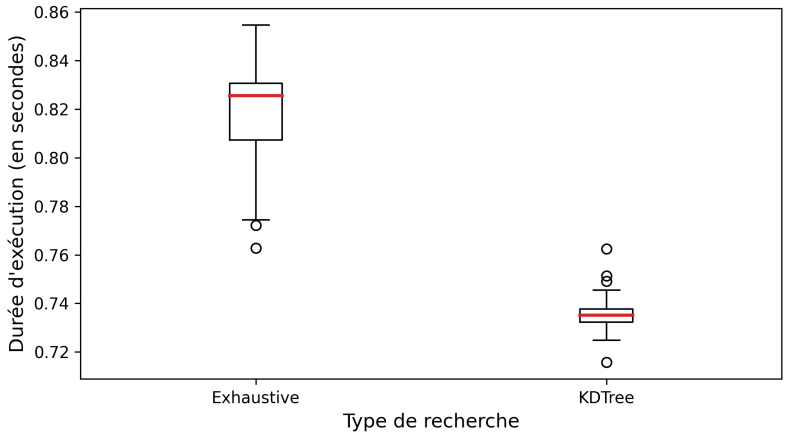

FIGURE 2 – Durée de calcul en secondes pour chaque recherche du plus proche voisin du vecteur calculé par `vector(A) + vector(B) - vector(C)`.

points n devrait être $n \gg 2^k$ pour que cette recherche soit efficace [Indyk, 2004]. Si ce n'est pas le cas, cette méthode pourrait ne pas être beaucoup plus rapide qu'une recherche exhaustive.

En plus, l'implémentation de l'arbre k-d utilisé ici est assez simple, alors que les implémentations utilisés dans les bibliothèques comme Scikit-Learn utilisent des méthodes plus sophistiquées comme le pruning et les fonctionnalités de Cython [6] ; une telle mise en œuvre surpasse le cadre de ce rapport.

5 Conclusions et perspectives

Le but de ce rapport était d'examiner le modèle Word2Vec et ses capacités dans quelques aspects. Surtout, nous avons explorés l'effet des méthodes d'améliorations présentés par Mikolov et al. [2013b] ainsi que l'aspect de l'évaluation des analogies.

En ce qui concerne ses sujets, nous avons posé trois hypothèses sur les méthodes d'amélioration et une hypothèse sur l'évaluation des analogies. Nous avons crées, exécutés, et évalués deux expériences afin de répondre à les hypothèses. Les trois hypothèses sur le premier sujet ont été confirmées en montrant les améliorations sur le modèle de base, tandis que la hypothèse du deuxième sujet avait des résultats mitigés.

6. https ://cython.org/

Références

Richard Bellman. Adaptive control processes : A guided tour. *The Mathematical Gazette*, 46 :160–161, 1961.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9) :509–517, 1975.

Murhaf Fares, Andrey Kutuzov, Stephan Oepen, and Erik Velldal. Word vectors, reuse, and replicability : Towards a community repository of large-text resources. In Jörg Tiedemann and Nina Tahmasebi, editors, *Proceedings of the 21st Nordic Conference on Computational Linguistics*, pages 271–276, Gothenburg, Sweden, May 2017. Association for Computational Linguistics. URL `https://aclanthology.org/W17-0237`.

Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3) :209–226, 1977.

Piotr Indyk. Nearest neighbors in high-dimensional spaces. *Handbook of discrete and computational geometry*, 2004.

Daniel Jurafsky. Speech and language processing, 2000.

Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. *Advances in neural information processing systems*, 27, 2014.

Omer Levy, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the association for computational linguistics*, 3 :211–225, 2015.

Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2–4, 2013, Workshop Track Proceedings*, 2013a. URL `http://arxiv.org/abs/1301.3781`.

Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural*

Information Processing Systems 26 : 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5–8, 2013, Lake Tahoe, Nevada, United States, pages 3111–3119, 2013b. URL https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html.