Word Embedding
From Count-based to Prediction-based

Jingbo Xia

Huazhong Agricultural University

xiajingbo.math@gmail.com

2020-02-11

Table of contents I

1. Count-based Vector Space Word Representation 3
   - Information Retrieval and Language Model 4
   - Word w and document d 5
   - Word w and context word c 6
   - From sparse representation to dense one, from SVD to LSA 7
2. Prediction-based Word Embedding 9
   - Word2Vec 10
   - Embedding visualization with t-SNE 14
3. Case Study: Word2Vec term embedding with PyTorch 18
   - Installation of PyTorch 19
   - PyTorch for artificial neural network 20
   - PyTorch for Skipgram 27
   - Appendix: Tensorflow for Skipgram 29

Outline

1. Count-based Vector Space Word Representation 3
   - Information Retrieval and Language Model 4
   - Word w and document d 5
   - Word w and context word c 6
   - From sparse representation to dense one, from SVD to LSA 7
2. Prediction-based Word Embedding 9
   - Word2Vec 10
   - Embedding visualization with t-SNE 14
3. Case Study: Word2Vec term embedding with PyTorch 18
   - Installation of PyTorch 19
   - PyTorch for artificial neural network 20
   - PyTorch for Skipgram 27
   - Appendix: Tensorflow for Skipgram 29

Count-based Vector Space Word Representation

Information Retrieval and Language Model

Information Retrieval and Language Model are two main NLP tasks which need compute the word representations.
Information Retrieval (IR) Compute the relevance of a document for a given word.
(word × document) matrix!

Language Modeling (LM)(e.g., first-order Markov model) Compute the probability of the current word given the previous word
(word × word) matrix!

Count-based Vector Space Word Representation

Word w and document d

Assume there are N documents. The below is a simple way to vectorize a word w^i:

$$\boldsymbol{w}_i = (\boldsymbol{w}_{i1}, \boldsymbol{w}_{i2}, \cdots, \boldsymbol{w}_{iN}),$$

here $\boldsymbol{w}_{in}$ = number of occurrences of w^i in document d^n, $n = 1, 2, \cdots, N$

One may add strength of association, e.g., TF·IDF,

$$tf \cdot idf(w^i, d^n) = tf(w^i, d^n) \times idf(w^i),$$

here $tf(w^i, d^n)$ is the term frequency of w^j in the j-th document, and inverse document frequency, $idf(w^i) = \log_e \frac{N}{\#\{d^n | w^i \text{ in } d^n\}}$.

	d^1	d^2	$\cdots$	d^N	
w^1					$=\boldsymbol{w}_1$
w^2					$=\boldsymbol{w}_2$
$\vdots$			$tf \cdot idf(w^i, d^n)$		$\vdots$
w^I					$=\boldsymbol{w}_I$

Count-based Vector Space Word Representation

Word w and context word c

Assume we consider J context word around the word w^i, the word w^i is vectorized by all the context word c^j, $(j = 1, 2, \cdots, J)$:

$$\boldsymbol{w}_i = (\boldsymbol{w}_{i1}, \boldsymbol{w}_{i2}, \cdots, \boldsymbol{w}_{iJ}),$$

here $\boldsymbol{w}_{ij}$ = number of co-occurrences of w^i with context word c^j.

Point-wise mutual information is used to consider further word-context info.

$$PMI(w^i, w^j) = \log_2 \frac{P(w^i, w^j)}{P(w^i)P(w^j)},$$

here $P(w^i, w^j)$ calculate the probability of two words co-occurred in a context.

Please noted that $\frac{P(w^i, w^j)}{P(w^i)P(w^j)} = \frac{P(w^i|w^j)}{P(w^i)}$, or $\frac{P(w^j|w^i)}{P(w^j)}$.

Count-based Vector Space Word Representation I

From sparse representation to dense one, from SVD to LSA

The representation is always sparse.
Singular Value Decomposition (SVD) –> Latent Semantic Analysis (LSA)

定理 (Singular Value Decomposition)

For an $m \times n$ matrix A of rank r, there exists a factorization (Singular Value Decomposition = SVD) as follows:

$$A = U \Sigma V^T,$$

where U and V are orthogonal matrices, $\Sigma = \begin{pmatrix} \Delta & 0 \\ 0 & 0 \end{pmatrix}$, $\Delta = diag(\sigma_1, \cdots, \sigma_r)$, $\sigma_i = \sqrt{\lambda_i}$ (which is called as singular value of A), $i = 1, 2, \cdots, r$, $r = Rank(A)$, λ_i is the eigenvalue of AA^T.

Generally, $\sigma_{i=1}^R U_i V_i^T$ is a rank-R approximation of an arbitrary A, which yields to a dense matrix!

Count-based Vector Space Word Representation II

From sparse representation to dense one, from SVD to LSA

Idea of LSA.

	Romance	Thriller	Detective	War	History	Cartoon
Term 1						
Term 2						
$\vdots$			Sparse!			
Term V						

SVD convert an original sparse term-document matrix an dense one. LSA use the dense rows to represent the embedding of the terms.

Outline

1. Count-based Vector Space Word Representation 3
   - Information Retrieval and Language Model 4
   - Word w and document d 5
   - Word w and context word c 6
   - From sparse representation to dense one, from SVD to LSA 7
2. Prediction-based Word Embedding 9
   - Word2Vec 10
   - Embedding visualization with t-SNE 14
3. Case Study: Word2Vec term embedding with PyTorch 18
   - Installation of PyTorch 19
   - PyTorch for artificial neural network 20
   - PyTorch for Skipgram 27
   - Appendix: Tensorflow for Skipgram 29

Prediction-based Word Embedding

Word2Vec

Word2vec [1] was created and published in 2013 by a team of researchers led by Tomas Mikolov at Google. Embedding vectors created using the Word2vec algorithm have many advantages compared to earlier algorithms such as latent semantic analysis [2].

1. Word2Vec is a group of related models that are used to produce word embeddings.
2. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words.
3. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space.

[1] https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[2] https://en.wikipedia.org/wiki/Word2vec

Prediction-based Word Embedding I

Word2Vec

1. Predict current word given context words (CBOW).
e.g., “I”, “like”, “”Peppa” $\longrightarrow$ “pig”.
2. Predict context word given current word (Skip-Gram).
e.g., “pig” $\longrightarrow$ “I”, “like”, “”Peppa”.

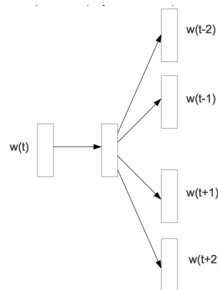


图 1: SkipGram structure

Prediction-based Word Embedding II

Word2Vec

Put Skip-Gram as example, the figure shows clearly the algorithm steps.

1. Skip-Gram does generate two embedding matrices, $W_{d\times V}$, and $W'_{V\times d}$ (Also denoted as U). V is the size of vocabulary, d is the length of embedding.
2. We generate our **one hot** input vector, w_c, for the **center word** w^c.
3. Obtain the word embedding of the center word, as $v_c = Ww_c$. W is the word embedding matrix as representation of center word.
4. For the context word, generate score vectors $W'v_c = (u_1^T v_c, u_2^T v_c, \cdots, u_V^T v_c)^T$.
Note: for a chosen word w^x, u_x is the embedding for the output word, v_x is for the input word. $u_x^T v_c$ is the inner product (similarity) of w^x and w^c.
5. Compute $Softmax(W'v_c)$.
Important! The conditional prob
$P(\mathrm{w}^x|\mathrm{w}^c) = Softmax(W'v_c)_{\text{Max 分量}} = \frac{\exp(u_x^T v_c)}{\sum_{i=1}^V \exp(u_i^T v_c)}$.
6. Minimize Cross Entropy of $Softmax(W'v_c)$ and one hot coding of the actual context word.

Prediction-based Word Embedding III

Word2Vec

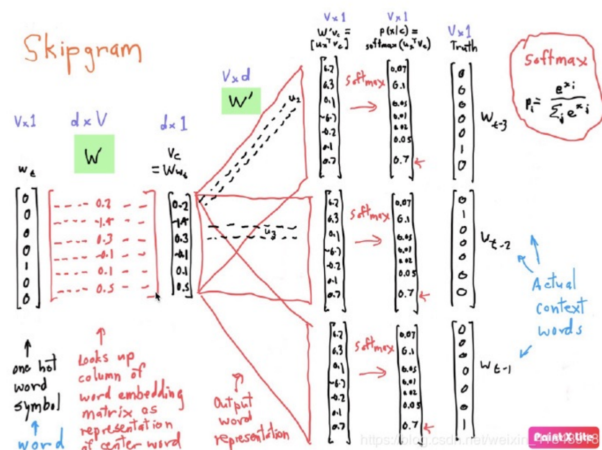


图 2: Neural network is fairly natural to fit SkipGram!

Note:

Prediction-based Word Embedding I

Embedding visualization with t-SNE

The t-distributed stochastic neighbor embedding (t-SNE) is a popular dimension reduction method [4]. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data.

Assume data point x_i in high dimension space R^D, and map point y_i in R^2.

1. Define a conditional similarity between the two data points

$$p_{j|i} = \frac{\exp\left(-\left|x_i - x_j\right|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\left|x_i - x_k\right|^2 / 2\sigma_i^2\right)}.$$

2. Define the similarity as a symmetrized version of the conditional similarity $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$.

Prediction-based Word Embedding II

Embedding visualization with t-SNE

3. Define a similarity matrix for our map points.

$$q_{ij} = \frac{f(|x_i - x_j|)}{\sum_{k \neq i} f(|x_i - x_k|)}$$

with $f(z) = \frac{1}{1+z^2}$.

4. Assume that our map points are all connected with springs. The stiffness of a spring connecting points i and j depends on the mismatch between the similarity of the two data points and the similarity of the two map points, that is, $p_{ij} - q_{ij}$. Now, we let the system evolve according to the laws of physics. If two map points are far apart while the data points are close, they are attracted together. If they are nearby while the data points are dissimilar, they are repelled. The final mapping is obtained when the equilibrium is reached.

Prediction-based Word Embedding III

Embedding visualization with t-SNE

Remarkably, this physical analogy stems naturally from the mathematical algorithm. It corresponds to minimizing the Kullback-Leiber divergence between the two distributions (p_{ij}) and (q_{ij}):

$$\mathbb{KL}(P||Q) = \sum_{i,j} p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

5. To minimize this score, we perform a gradient descent. The gradient can be computed analytically:

$$\frac{\partial\, KL(P||Q)}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})\, g\left(|x_i - x_j|\right) u_{ij}$$

where $g(z) = \frac{z}{1+z^2}$.

Prediction-based Word Embedding IV

Embedding visualization with t-SNE

图 3: t-SNE visualization demo

[4]https:

Outline

1. Count-based Vector Space Word Representation 3
   - Information Retrieval and Language Model 4
   - Word w and document d 5
   - Word w and context word c 6
   - From sparse representation to dense one, from SVD to LSA 7
2. Prediction-based Word Embedding 9
   - Word2Vec 10
   - Embedding visualization with t-SNE 14
3. Case Study: Word2Vec term embedding with PyTorch 18
   - Installation of PyTorch 19
   - PyTorch for artificial neural network 20
   - PyTorch for Skipgram 27
   - Appendix: Tensorflow for Skipgram 29

Case Study: Word2Vec term embedding with PyTorch

Installation of PyTorch

```
1  # Replace pip source
2  mkdir ~/.pip
3  vim ~/.pip/pip.conf
4  # copy this two line to this file.
5  [global]
6  index-url = https://mirrors.aliyun.com/pypi/simple
7
8  # replace source when you downloading.
9  sudo pip3 install package -i source_url
10
11 #清华: https://pypi.tuna.tsinghua.edu.cn/simple
12 #中国科技大学 https://pypi.mirrors.ustc.edu.cn/simple/
13 #华中理工大学: http://pypi.hustunique.com/
14 #山东理工大学: http://pypi.sdutlinux.org/
15 #豆瓣: http://pypi.douban.com/simple/
```

```
1  $sudo pip3 install torch==1.4.0
```


Case Study: Word2Vec term embedding with PyTorch

PyTorch for artificial neural network

A tutorial codes for PyTorch on NN construction [5].

```
1  # -*- coding:utf-8 -*-
2  #! usr/bin/env python3
3  """
4  Created on 09/04/2020 下午4:07
5  @Author: Xinzhi Yao
6  Refrence Link: https://morvanzhou.github.io/tutorials/machine-learning
       /torch/3-02-classification/
7  """
8
9  import torch
10 import matplotlib.pyplot as plt
11 import torch.nn.functional as F    # 激发函数都在这
12 import matplotlib.pyplot as plt
```

[5]This codes refers to https://morvanzhou.github.io/tutorials/machine-learning/torch/3-02-classification/. Another easy-to-follow tutorial is provided by PyTorch page, https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py.

Case Study: Word2Vec term embedding with PyTorch

PyTorch for artificial neural network

```
# 建立模拟数据集
n_data = torch.ones(100, 2)            # 数据的基本形态
x0 = torch.normal(2*n_data, 1)         # 类型0 x data (tensor), shape
    =(100, 2)
y0 = torch.zeros(100)                  # 类型0 y data (tensor), shape
    =(100, )
x1 = torch.normal(-2*n_data, 1)        # 类型1 x data (tensor), shape
    =(100, 1)
y1 = torch.ones(100)                   # 类型1 y data (tensor), shape
    =(100, )

# 注意 x, y 数据的数据形式是一定要像下面一样 (torch.cat 是在合并数据)
x = torch.cat((x0, x1), 0).type(torch.FloatTensor)  # FloatTensor =
    32-bit floating
y = torch.cat((y0, y1), ).type(torch.LongTensor)    # LongTensor = 64-
    bit integer
```

```
# 画图
plt.scatter(x.data.numpy()[:, 0], x.data.numpy()[:, 1], c=y.data.numpy
    (), s=100, lw=0, cmap='RdYlGn')
plt.show()
```


Case Study: Word2Vec term embedding with PyTorch

PyTorch for artificial neural network

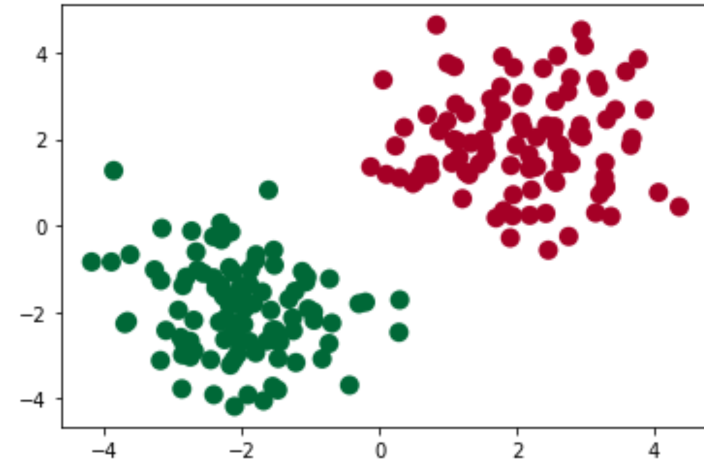


图 4: Plot of samples in two class

Case Study: Word2Vec term embedding with PyTorch

PyTorch for artificial neural network

```
# 建立神经网络
class Net(torch.nn.Module):     # 继承 torch 的 Module
    def __init__(self, n_feature, n_hidden, n_output):
        super(Net, self).__init__()     # 继承 __init__ 功能
        self.hidden = torch.nn.Linear(n_feature, n_hidden)   # 隐藏层
            线性输出
        self.out = torch.nn.Linear(n_hidden, n_output)       # 输出层
            线性输出

    def forward(self, x):
        # 正向传播输入值, 神经网络分析出输出值
        x = F.relu(self.hidden(x))      # 激励函数(隐藏层的线性值)
        x = self.out(x)                 # 输出值, 但是这个不是预测值,
            预测值还需要再另外计算
        return x
```


Case Study: Word2Vec term embedding with PyTorch

PyTorch for artificial neural network

```

net = Net(n_feature=2, n_hidden=10, n_output=2) # 几个类别就几个
    output

print(net)  # net 的结构
"""
Net (
  (hidden): Linear (2 -> 10)
  (out): Linear (10 -> 2)
)
"""
```

Case Study: Word2Vec term embedding with PyTorch
PyTorch for artificial neural network

```
# 训练网络
# optimizer 是训练的工具
optimizer = torch.optim.SGD(net.parameters(), lr=0.02)  # 传入 net 的
    所有参数，学习率
# 算误差的时候，注意真实值!不是! one-hot 形式的，而是1D Tensor，(batch
    ,)
# 但是预测值是2D tensor (batch, n_classes)
loss_func = torch.nn.CrossEntropyLoss()

ffor t in range(100):
    out = net(x)      # 喂给 net 训练数据 x，输出分析值
    loss = loss_func(out, y)     # 计算两者的误差
    optimizer.zero_grad()   # 清空上一步的残余更新参数值
    loss.backward()         # 误差反向传播，计算参数更新值
    prediction = torch.max(F.softmax(out), 1)[1]
    pred_y = prediction.data.numpy().squeeze()
    target_y = y.data.numpy()
    accuracy = sum(pred_y == target_y)/200.  # 预测中有多少和真实值一
        样
    print('Loss: {0}␣\t␣Accuracy: {1}'.format(loss, accuracy))
    optimizer.step()        # 将参数更新值施加到 net 的 parameters 上
```


Case Study: Word2Vec term embedding with PyTorch
PyTorch for artificial neural network

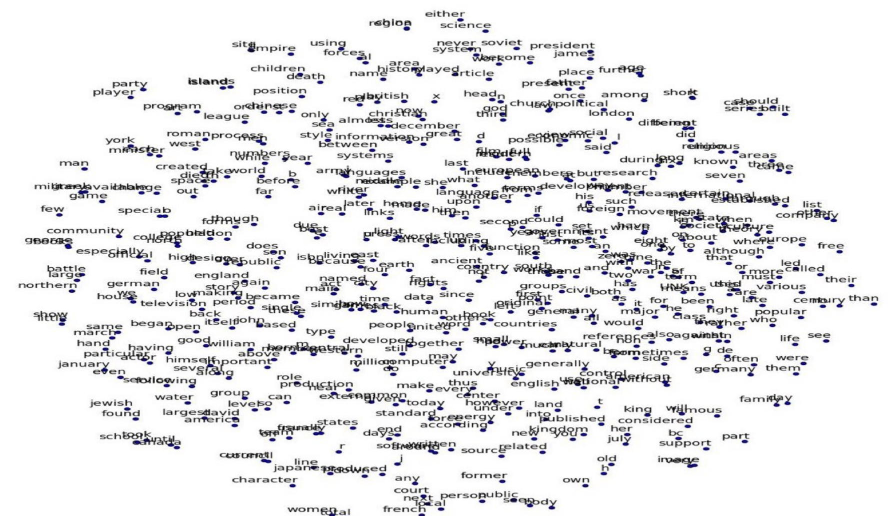
```
# Loss: 1.5146235227584839 ^^I Accuracy: 0.03
# Loss: 1.3551067113876343 ^^I Accuracy: 0.06
# Loss: 1.2119133472442627 ^^I Accuracy: 0.095
# Loss: 1.084607720375061 ^^I Accuracy: 0.185
# Loss: 0.9725099205970764 ^^I Accuracy: 0.26
# Loss: 0.8747918605804443 ^^I Accuracy: 0.39
# Loss: 0.79034823179245 ^^I Accuracy: 0.48
# Loss: 0.7178248763084412 ^^I Accuracy: 0.5
# Loss: 0.6556925773620605 ^^I Accuracy: 0.5
# Loss: 0.6024326682090759 ^^I Accuracy: 0.955
# Loss: 0.5566229820251465 ^^I Accuracy: 0.995
# Loss: 0.5169969797134399 ^^I Accuracy: 0.995
# Loss: 0.482472687959671 ^^I Accuracy: 0.995
# Loss: 0.45217418670654297 ^^I Accuracy: 0.995
# Loss: 0.42538687586784363 ^^I Accuracy: 0.995
# Loss: 0.40153583884239197 ^^I Accuracy: 0.995
# Loss: 0.3801606297492981 ^^I Accuracy: 0.995
# Loss: 0.3608894348144531 ^^I Accuracy: 0.995
# Loss: 0.3434189260005951 ^^I Accuracy: 0.995
# Loss: 0.32750222086906433 ^^I Accuracy: 0.995
# Loss: 0.3129340410232544 ^^I Accuracy: 0.995
# Loss: 0.29954665899276733 ^^I Accuracy: 0.995
# Loss: 0.2872000038623381 ^^I Accuracy: 0.995
```


Case Study: Word2Vec term embedding with PyTorch
PyTorch for Skipgram

```
# -*- coding:utf-8 -*-
#! usr/bin/env python3
"""
Created on 10/04/2020 上午12:33
@Author: Xinzhi Yao
"""

import collections
import string
import os
import re
import random
from sklearn.manifold import TSNE
import matplotlib
import matplotlib.pyplot as plt

import numpy as np
from collections import Counter
import random

import torch
import torch.nn as nn
import torch.nn.functional as F
```


Case Study: Word2Vec term embedding with PyTorch
PyTorch for Skipgram

Visualization result



Case Study: Word2Vec term embedding with PyTorch

Appendix: Tensorflow for Skipgram

```
1  # Copyright 2015 The TensorFlow Authors. All Rights Reserved.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #     http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 #
   ==============================================================================
15
16 from __future__ import absolute_import
17 from __future__ import division
18 from __future__ import print_function
19
20 import collections
```


Acknowledgement

谢谢！