Clustering Document Embeddings via a Multilayered Self-Organizing Map

Atanas Denkov[1]

Abstract: TODO

Keywords: Machine Learning; Distributed Computing; Self-organizing Maps; Unsupervised Learning

[1] Hochschule für Technik und Wirtschaft, Angewandte Informatik, Wilhelminenhofstr., 12459 Berlin, Germany
s0559025@htw-berlin.de

1 Continuous Bag of Words

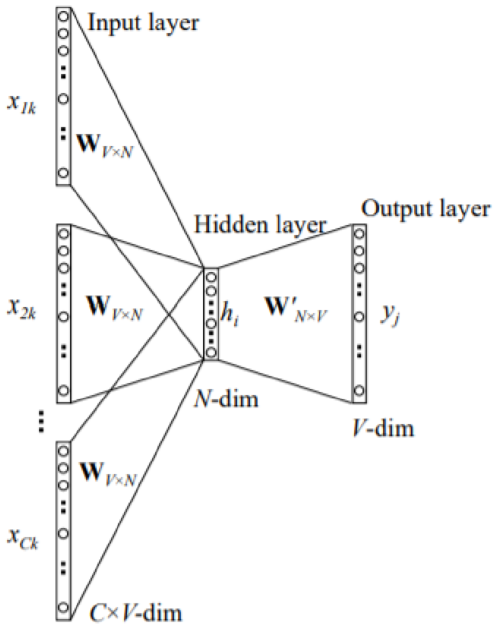

Fig. 1: The Continuous Bag of Words model

The Continuous Bag of Words model is a shallow neural network used for creating word vectors, also refered to as word *embeddings*. The set of word vectors forms a vector space that captures the similarity between words. The objective of the model is to predict the true center word w_o given the context words $w_1, w_2, \ldots, w_c$ that surround it. For example, in the sentence *The big brown fox jumped over the puddle*, we would like to predict the word *jumped* given the context words *[the, big, brown, over, the, puddle]*. The algorithm takes a probabilistic approach and tries to model the probability $p(w_o|w_1, w_2, \ldots, w_c)$. The objective is then formulated as maximizing that probability.

1.1 Forward pass

The input to our model is a set of context words represented as one-hot encoded vectors $x_1, x_2, \ldots, x_c$ where $x_c \in \mathbb{R}^V$. V corresponds to the size of the vocabulary that the model can perceive. Each one-hot vector has a 1 at the index that is unique to the word it represents ($x_k = 1$) and 0's everywhere else ($\{x_{k'} = 0|k' \in [1, 2, \ldots, V], k' \neq k\}$). The neural network has a hidden matrix $\mathbf{W} \in \mathbb{R}^{V\times N}$ that holds a word vector $v_w \in \mathbb{R}^N$ for each word w when it appears as a context word. Given that the context words are the input to our model, this matrix is called the *input matrix*. The hidden layer of the network functions as a lookup table to extract the input vectors $v_1, v_2, \dots, v_c$ by computing the following dot products

$$\begin{aligned} v_1 &= \mathbf{W}^T x_1 \\ v_2 &= \mathbf{W}^T x_2 \\ &\vdots \\ v_c &= \mathbf{W}^T x_c. \end{aligned} \tag{1}$$

These vectors are then averaged to form the hidden state

$$\mathbf{h} = \frac{1}{C}(v_1 + v_2 + \dots + v_c),\ \mathbf{h} \in \mathbb{R}^N. \tag{2}$$

N denotes the size of the embedding space and is typically a hyperparameter. The model has an additional weight matrix $\mathbf{W'} \in \mathbb{R}^{N \times V}$ that holds a word vector $v'_w \in \mathbb{R}^N$ for each word w when it appears as a center word. We refer to this matrix as the *output matrix*. If we compute the dot product between the hidden state $\mathbf{h}$ and a word vector v'_j, we obtain a score that measures the similarity between the context and the word represented by that word vector.

$$u_j = {v'_j}^T \mathbf{h} \tag{3}$$

We convert the scores into probabilities via the softmax function and obtain our prediction

$$p(w_j|w_I) = \hat{y}_j = \frac{\exp u_j}{\sum_{j'=1}^{V} \exp u_{j'}} = \frac{\exp {v'_j}^T \mathbf{h}}{\sum_{j'=1}^{V} \exp {v'_{j'}}^T \mathbf{h}}. \tag{4}$$

1.2 Backward pass

The objective of our model is to maximize the probability $p(w_j|w_I)$ defined in Equation 4 when w_j really is the center word given the context w_I. That is, we want to compare the target probability $y_j = 1$ with our estimate $\hat{y}_j$. We do this by using a simplified version of the cross-entropy loss function

$$\begin{aligned} H(\hat{y}_j, y_j) &= -y_j \log \hat{y}_j \\ &= -\log \hat{y}_j \\ &= -\log \frac{\exp u_j}{\sum_{j'=1}^{V} \exp u_{j'}} \qquad (5) \\ &= -u_j + \log \sum_{j'=1}^{V} \exp u_{j'}. \qquad (6) \end{aligned}$$

The transition from (5) to (6) follows the logarithm rule for fractions

$$\log_b\left(\frac{A}{B}\right) = \log_b(A) - \log_b(B). \tag{7}$$

We use gradient descent to update v'_j and $v_1, v_2 \ldots, v_c$ such that $H(\hat{y}_j, y_j) \approx 0$.
The chain rule tells us that

$$\frac{\partial H}{\partial v'_j} = \frac{\partial H}{\partial u_j} \cdot \frac{\partial u_j}{\partial v'_j}. \tag{8}$$

Calculating the derivative of the loss yields

$$\begin{aligned}
\frac{\partial H}{\partial u_j} &= \frac{\partial}{\partial u_j} - u_j + \log \sum_{j'=1}^{V} \exp u_{j'} \\
&= \frac{\partial}{\partial u_j} - u_j + \frac{\partial}{\partial u_j} \log \sum_{j'=1}^{V} \exp u_{j'} \\
&= -1 + \frac{1}{\sum_{j'=1}^{V} \exp u_{j'}} \frac{\partial}{\partial u_j} \sum_{j'=1}^{V} \exp u_{j'} \\
&= -1 + \frac{1}{\sum_{j'=1}^{V} \exp u_{j'}} \frac{\partial}{\partial u_j} \exp u_j \\
&= -1 + \frac{\exp u_j}{\sum_{j'=1}^{V} \exp u_{j'}} \\
&= \hat{y}_j - 1.
\end{aligned} \tag{9}$$

Next we take the derivative of u_j with respect to v'_j.

$$\frac{\partial u_j}{\partial v'_j} = \frac{\partial}{\partial v'_j} v'_j \cdot \mathbf{h} = \mathbf{h}. \tag{10}$$

By substituting our calculations in Equation 8 and considering that the index of the true center word is j, then our derivative is

$$\frac{\partial H}{\partial v'_{j'}} = \begin{cases} (\hat{y}_{j'} - 1) \cdot \mathbf{h}, & \text{if } j' = j \\ (\hat{y}_{j'}) \cdot \mathbf{h} & \text{otherwise.} \end{cases} \tag{11}$$

Since we are using gradient descent, our update rule for $v'_{j'}$ becomes

$$v'^{(new)}_{j'} = \begin{cases} v'^{(old)}_{j'} - \alpha \cdot ((\hat{y}_{j'} - 1) \cdot \mathbf{h}), & \text{if } j' = j \\ v'^{(old)}_{j'} - \alpha \cdot \hat{y}_{j'} \cdot \mathbf{h} & \text{otherwise.} \end{cases} \tag{12}$$

Note that when $j' = j$, the negative sign of the expression $\hat{y}_{j'} - 1$ cancels out the negative sign of the update rule, thus pushing the value of v'_j towards the context $\mathbf{h}$. Conversely,

when $j' \neq j$, the output vector is pushed away from the context. In other words, vectors that have similar contexts are clustered together by being pushed away from unrelated contexts and at the same time being pushed towards contexts that are meaningful. This notion is refered to as *distributional semantics*.
The next step is to compute the partial derivatives of the loss function with respect to the input vectors $v_1, v_2, \dots, v_c$. The chain rule tells us that

$$\frac{\partial H}{\partial v_i} = \sum_{j=1}^{V} \frac{\partial H}{\partial u_j} \cdot \frac{\partial u_j}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial v_i}, \text{ for } i \in [1, 2, \dots, C]. \tag{13}$$

The first derivative in the summation has already been computed. The second one yields

$$\frac{\partial u_j}{\partial \mathbf{h}} = \frac{\partial}{\partial \mathbf{h}} v_j'^T \mathbf{h} = v_j', \tag{14}$$

and the third one:

$$\frac{\partial \mathbf{h}}{\partial v_i} = \frac{\partial}{\partial v_i} \frac{1}{C}(v_1 + v_2 + \dots + v_c) = \frac{1}{C}. \tag{15}$$

Substituting (14) and (15) into (13) gives us

$$\frac{\partial H}{\partial v_i} = \begin{cases} \frac{1}{C} \cdot \sum_{j'=1}^{V} (\hat{y}_{j'} - 1) \cdot v_{j'}', & \text{if } j' = j \\ \frac{1}{C} \cdot \sum_{j'=1}^{V} \hat{y}_{j'} \cdot v_{j'}' & \text{otherwise.} \end{cases} \quad \text{for } i \in [1, 2, \dots, C] \tag{16}$$

We now derive the update rule

$$v_i^{(new)} = \begin{cases} v_i^{(old)} - \alpha \cdot \frac{1}{C} \cdot \sum_{j'=1}^{V} (\hat{y}_{j'} - 1) \cdot v_{j'}', & \text{if } j' = j \\ v_i^{(old)} - \alpha \cdot \frac{1}{C} \cdot \sum_{j'=1}^{V} \hat{y}_{j'} \cdot v_{j'}' & \text{otherwise.} \end{cases} \quad \text{for } i \in [1, 2, \dots, C] \tag{17}$$

1.3 Optimization

Notice that Equation 4 requires iterating over the whole vocabulary to compute the normalization factor. The vocabulary usually exceeds hundreds of thousands of words, which renders this computation rather inefficient. A technique called *negative sampling* deals with this by reformulating the objective function, the weights and the update equations. Instead of going over the whole vocabulary, we can sample a set of words from it that are unlikely to occur near the context and update their weights in such a way that they are pushed further away from the context, while the true center word is pushed towards it. First, we remodel the objective function to be

$$H(\hat{y}_j, y_j) = -\log \sigma(u_j) - \sum_{k=1}^{K} \log \sigma(-u_k). \tag{18}$$

The words $\{u_k | k \in [1, 2, \ldots, K]\}$ are sampled from a noise distribution $P(w)$, which is defined in terms of word frequencies. A good heuristic value for K is 10. Note that K is a lot smaller than V. If $f(w)$ defines the relative frequency of the word w in our corpus, then the probability of using w as a negative sample is defined by

$$P(w) = \frac{f(w)^{3/4}}{\sum_{i=1}^{V} f(w_i)^{3/4}}. \tag{19}$$

The power factor $3/4$ is determined heuristically and tends to favor words that appear less frequently.
Now that we have changed the objective, we need to recalculate the partial derivatives for our weights. Backpropagating the update signal to the output vector v'_j is done via the following derivative chain:

$$\frac{\partial H}{\partial v'_j} = \frac{\partial H}{\partial \sigma(u_j)} \cdot \frac{\partial \sigma(u_j)}{\partial u_j} \cdot \frac{\partial u_j}{\partial v'_j}. \tag{20}$$

The first derivative yields

$$\begin{aligned}
\frac{\partial H}{\partial \sigma(u_j)} &= \frac{\partial}{\partial \sigma(u_j)} - \log \sigma(u_j) - \sum_{k=1}^{K} \log \sigma(-u_k) \\
&= \frac{\partial}{\partial \sigma(u_j)} - \log \sigma(u_j) - \frac{\partial}{\partial \sigma(u_j)} \sum_{k=1}^{K} \log \sigma(-u_k) \\
&= -\frac{1}{\sigma(u_j)} \frac{\partial}{\partial \sigma(u_j)} \sigma(u_j) \\
&= -\frac{1}{\sigma(u_j)}.
\end{aligned} \tag{21}$$

The derivative of the sigmoid function is defined as

$$\frac{\partial \sigma(u_j)}{\partial u_j} = \sigma(u_j) \cdot (1 - \sigma(u_j)). \tag{22}$$

The third derivative has been calculated in Equation 10. Chaining these produces

$$\frac{\partial H}{\partial v'_{j'}} = \begin{cases} (\sigma(u_{j'}) - 1) \cdot \mathbf{h}, & \text{if } j' = j \\ \sigma(u_{j'}) \cdot \mathbf{h} & \text{otherwise.} \end{cases} \tag{23}$$

Note that we need to compute this just for the center word w_j and for all negative samples w_k instead of for the whole vocabulary. We substitute this equation in (12) to obtain the new update rule.
We now need to backpropagate the update signal to the context vectors $v_1, v_2, \ldots, v_c$. Again, we determine the derivative chain and then compute each derivative. First, we define the set S as the intersection between the index of the current center word j and the indices of the negative samples $k,\ k \in [1, 2, \ldots, K]$.

$$\frac{\partial H}{\partial v_i} = \sum_{j' \in S} \frac{\partial H}{\partial \sigma(u_{j'})} \cdot \frac{\partial \sigma(u_{j'})}{\partial u_{j'}} \cdot \frac{\partial u_{j'}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial v_i} \tag{24}$$

$$\frac{\partial H}{\partial v_i} = \begin{cases} \frac{1}{C} \cdot \sum_{j' \in S} \sigma(u_{j'}) - 1 \cdot v'_{j'}, & \text{if } j' == j \\ \frac{1}{C} \cdot \sum_{j' \in S} \sigma(u_{j'}) \cdot v'_{j'} & \text{otherwise. } (j' == k) \end{cases} \tag{25}$$

We substitute the gradient in (17) to obtain the new update rule.

2 Distributed Memory Model of Paragraph Vectors

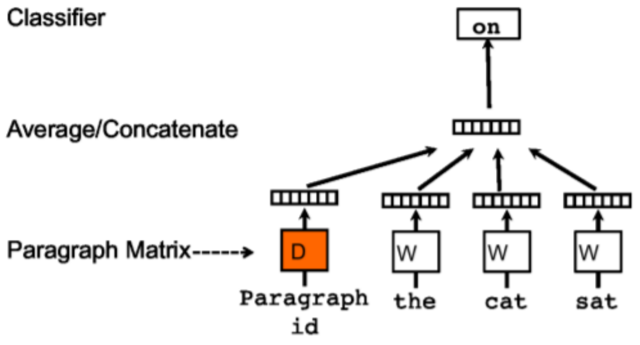

Fig. 2: The DM-PV model.

The distributed memory model of paragraph vectors, also refered to as DM-PV, attemps to generalize the abilities of CBOW to variable-length documents. In DM-PV, every document is mapped to a unique vector represented by a row in a matrix $\mathbf{D} \in \mathbb{R}^{M \times N}$, where M denotes the number of documents. As before, every word is also mapped to a vector in an input matrix $\mathbf{W} \in \mathbb{R}^{V \times N}$ and an output matrix $\mathbf{W'} \in \mathbb{R}^{N \times V}$, respectively. The only difference between DM-PV and CBOW is that the document vector d_j that contains the words is asked to contribute to the prediction task by being integrated into the hidden state in Equation (2). See Figure 2 for a visual representation. Thus, the new hidden state is computed as follows:

$$\mathbf{h} = d_j + \frac{1}{C}(v_1 + v_2 + \ldots + v_c),\ \mathbf{h} \in \mathbb{R}^N. \tag{26}$$

The document vector can be thought of as another word. It acts as a memory that remembers what is missing from the current context, i.e the document's topic. Note that the document vector is shared across all contexts $v_1 + v_2 + \ldots + v_c$ generated from the same document, but not across documents. Conversely, the word vector matrices **W** and **W'** are shared across documents. During backpropagation, the derivative of the cost function with respect to the target document vector

$$\frac{\partial H}{\partial d_j} \tag{27}$$

is computed as well, thus pushing documents with similar words towards those words and therefore, towards each other.

2.1 Inference

So far, we've only considered the case of training the algorithm in order to create numeric representations for **previously seen** documents. But how do we generate such representations for unseen documents? This problem is addressed in the **inference stage**.
Given is an unseen document d. A randomly-initialized vector is added to our model's weight matrix **D**. That vector represents the document vector for d. The vector for d is updated by gradient descending on **D** for i epochs while keeping **W**, **W'** and their gradients fixed. The number of epochs to descent on d should match the number of epochs used during training.

3 Self-Organizing Maps

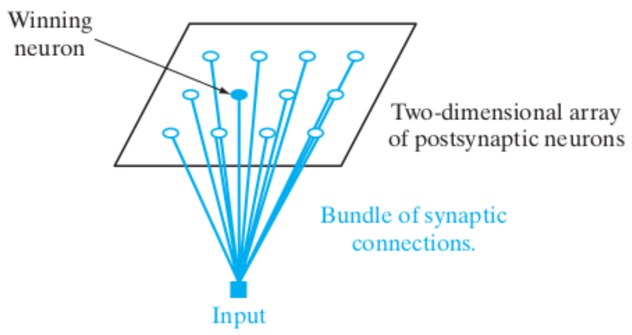
Winning neuron
Two-dimensional array of postsynaptic neurons
Bundle of synaptic connections.
Input

Fig. 3: An image showing the connection between the input data and the neurons of a self-organizing map. Each input is connected to all of the neurons in the lattice. The neurons compete among each other and the neuron that resembles the input the most is marked as the winner neuron. After multiple iterations, the lattice will reflect the feature topology of the inputs.

Self-organizing maps are a special kind of artificial neural networks. They provide the means for clustering and visualsiing highly-dimensional data, e.g document vectors.
In a self-organizing map, neurons are placed at the nodes of a *lattice* that is typically two dimensional. The neurons selectively adapt to the input data during a *competitive learning* procedure. The locations of the neurons on the aforementioned lattice become ordered in such a way that the lattice represents a coordinate system for different input features. In other words, the coordinates of the neurons in the lattice represent the statistical features contained in the input patterns. For this reason, the neurons of the map are said to *self-organize* onto the input data.
The algorithm first initializes the weights of the network. Small values picked from a random-number generator ensure that no prior order is imposed on the lattice. Once initialized, three processes create the map:

- **Competition**. For every input pattern, the neurons in the network compute their values of a function that is used as the basis for competition among the neurons. The neuron that yields the largest value is declared winner of the competition
- **Cooperation**. The winning neuron determines the coordinates of its neighbouring neurons on the lattice and allows them to become partially activated as well.
- **Neuron adaptation**. The winner neuron and its neighbours are adjusted such that their probability of being activated once a similar input pattern is plugged into the model is increased.

3.1 Competition

Let m denote the dimension of the input space. We select a data sample from the input space and denote it by

$$\mathbf{x} = [x_1, x_2, \ldots, x_m]^T. \tag{28}$$

Let $G \in \mathbb{N}^{w \times h \times m}$ represent a rectangular lattice of neurons where w and h denote the width and height of the lattice, respectively. The neuron at position (w, h) in the lattice has a weight vector denoted by

$$\mathbf{w}_{w,h} = [w_1, w_2, \ldots, w_m]^T. \tag{29}$$

To find the best matching neuron relative to the input vector $\mathbf{x}$, we compute the Euclidean distance between each neuron and x;

$$c(\mathbf{x}) = \underset{(i,j)}{\arg\min} \, ||\mathbf{x} - \mathbf{w}_{i,j}||, \ i \in 1, 2, \ldots, w, \ j \in 1, 2, \ldots, h \tag{30}$$

The neuron with the smallest distance is deemed the winner of the competition phase. $c(\mathbf{x})$ is a tuple containing the (x, y)-coordinates of the winner neuron in the lattice for the input sample $\mathbf{x}$.

3.2 Cooperation

The winning neuron locates the center of a neighbourhood of cooperating neurons. Whenever a neuron is activated by winning the competition, its neighbouring neurons will also be activated. This property stems from the human brain, where lateral interactions among a set of excited neurons is quite common.
Let $h_{j,c(\mathbf{x})}$ denote the neighbourhood centered around the winning neuron $c(\mathbf{x})$ and encompassing a set of cooperating neurons, one of which is denoted by j. Let $d_{j,c(\mathbf{x})}$ denote the coordinate distance between neurons j and $c(\mathbf{x})$ in the lattice. Then, the neighbourhood must satisfy the following restrictions:

- The maximum value for $h_{j,c(\mathbf{x})}$ must be attained at the position for which the distance $d_{j,c(\mathbf{x})} = 0$.
- The value of $h_{j,c(\mathbf{x})}$ should decrease as the distance $d_{j,c(\mathbf{x})}$ increases.

These requirements are satisfied by the *Gaussian function*

$$h_{j,c(\mathbf{x})} = \exp(-\frac{d^2_{j,c(\mathbf{x})}}{2\sigma^2}). \tag{31}$$

The parameter σ is the width of the neighbourhood and represents the size of the topological neighbourhood, which is defined as the spread of the gaussian function.

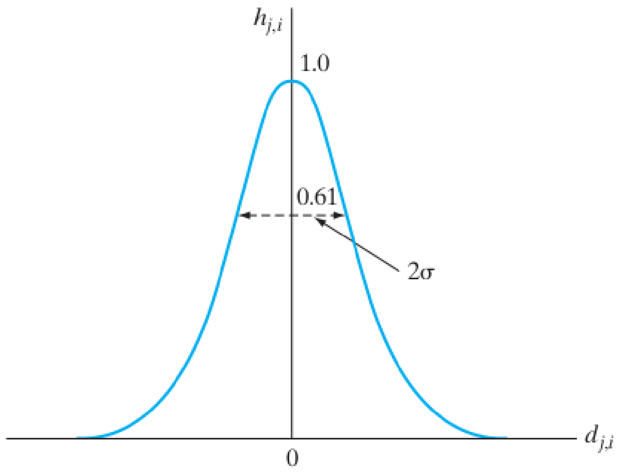

Fig. 4: The gaussian neighbourhood function. σ controls the size of the neighbourhood. Note that the peak of the bell curve maps to all neurons whose distance to the winner is 0. In other words, neurons that have large distances to the winner have low probabilities of being activated.

The distance function is defined in terms of the output space as

$$d^2_{j,c(\mathbf{x})} = \|j - c(\mathbf{x})\|^2. \tag{32}$$

We further shrink the size of the topological neighbourhood with time. This invariant is satisfied by making $h_{j,c(\mathbf{x})}$ decrease with time. We redefine σ as an *exponential decay* function

$$\sigma(n) = \sigma_0 \exp(-\frac{n}{t_1}) \tag{33}$$

and redefine $h_{j,c(\mathbf{x})}$ as

$$h_{j,c(\mathbf{x})}(n) = \exp(-\frac{d^2_{j,c(\mathbf{x})}}{2\sigma^2(n)}). \tag{34}$$

Thus, as the number of iterations (defined by n) increases, the width of the gaussian density (see Figure 4) decreases at an exponential rate. Decreasing the width of the neighbourhood ensures that neurons without any significant relationship in the input space are not influenced by each other in the output space.

3.3 Neuron adaptation

The adaptive process is responsible for changing the weight vector $\mathbf{w}_{i,j}$ of neuron (i, j) in relation to the input vector $\mathbf{x}$. The update rule for neuron (i, j) at iteration n is defined as

$$\mathbf{w}_{i,j}(n + 1) = \mathbf{w}_{i,j}(n) + \eta(n)h_{j,c(\mathbf{x})}(n)(\mathbf{x}(n) - \mathbf{w}_{i,j}(n)). \tag{35}$$

We can see that the probabilities produced by $h_{j,c(\mathbf{x})}$ effectively cancel out any updates to neurons that are not part of the neighbourhood of the winner, because in that case $h_{j,c(\mathbf{x})} = 0$ and therefore $\mathbf{w}_{i,j}(n+1) = \mathbf{w}_{i,j}(n)$.
The learning-rate parameter $\eta(n)$ is also defined as a function of time. It starts at some initial value η_0 and decreases gradually with increasing time n. This requirement is satisfied via the following heuristic function

$$\eta(n) = \eta_0 \exp(-\frac{n}{t_2}). \tag{36}$$

3.4 Summary

The algorithm is summarized as follows:

- **Initialization**. Choose random values for the initial weight vectors $\mathbf{w}_{i,j}(0)$.
- **Sampling**. Draw a sample $\mathbf{x} \in \mathbb{R}^m$ from the input space.
- **Similarity matching**. Find the best-matching neuron $c(\mathbf{x})$ at time-step n by using the minimum-distance criterion specified in Equation 30.
- **Update**. Adjust the weight vectors for all excited neurons by using Equation 35
- **Continuation**. Continue with step 2 until no noticeable changes in the feature map are observed.

4 Self-organizing Document Embeddings for Plagiarism Detection

4.1 Implementation

4.2 Evaluation