Clustering Document Embeddings via Self-organizing Maps for Plagiarism Detection

Atanas Denkov[1]

Abstract: Academic plagiarism greatly increases the complexity of assesing student competence. Institutions invest hundreds of working hours in detecting plagiarised work. This is inefficient in terms of cost and time. Automated plagiarism detection methods have gained traction with the advances in natural language processing and machine learning. This work introduces a system for aggregating and filtering documents based on the similarities of their contexts using machine learning techniques. Users can input previously unseen documents and the system will filter and aggregate potential plagiarism candidates from a fixed-length dataset crawled from Wikipedia. The user need only examine the candidates returned by the system for potential plagiarisms. This reduces the manual labour cost associated with plagiarism detection.

Keywords: Machine Learning; Natural Language Processing; Self-organizing Maps; Unsupervised Learning

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

1 Architecture

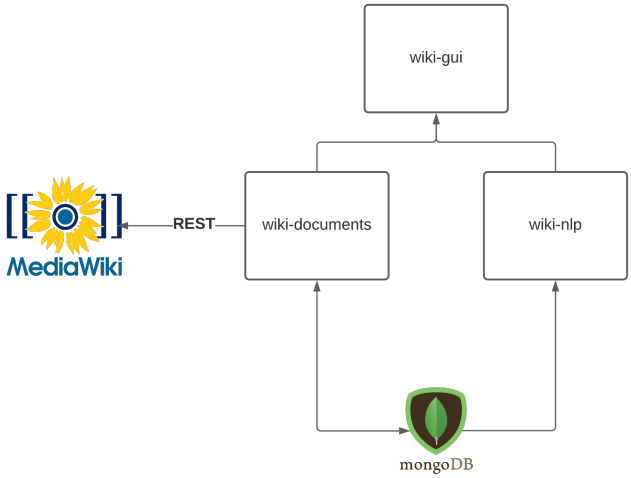

Fig. 1: System architecture

The `wiki-documents` application represents an extract-transform-load pipeline that extracts Wikipedia articles and categories from the MediaWiki HTTP server. The extracted documents are processed and saved into a non-relational document database - MongoDB. `wiki-documents` also provides an easy to use REST interface for creating, reading, updating and deleting documents and categories from the database.
The `wiki-nlp` application reads the extracted articles from the database and trains the machine learning algorithms described in Chapter 2. Every document is converted into a multidimensional vector via an unsupervised shallow neural network. The set of all document vectors forms a vector space in which the distance between two vectors represents the similarity between the two documents. The multidimensional vector space is projected onto a discrete two-dimensional coordinate map. Each document is mapped to a coordinate in the map. Documents mapped to the same coordinate are classified as similar. `wiki-nlp` provides a REST interface that receives previously unseen documents as input. The unseen documents are converted into multidimensional vectors and are assigned to clusters in the coordinate map. Documents from the dataset that are mapped to the same coordinate in the cluster are considered potential plagiarism sources and are returned to the caller for further examination.
The `wiki-gui` applicaion provides a graphical user interface that allows the user to input a text document and to receive potential plagiarism sources. The application implements an HTTP client that internally communicates with `wiki-documents` and `wiki-nlp` to provide the necessary services to the user.

1.1 Extract Transform Load Pipeline

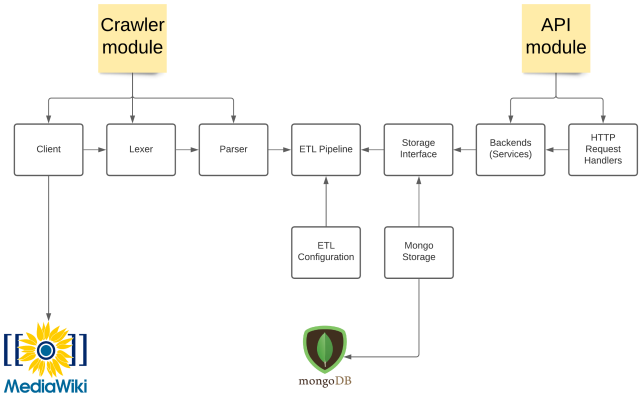

Fig. 2: Information flow diagram between all software components in the extract-transform-load pipeline

The ETL pipeline consists of multiple components for extracting, processing and storing Wikipedia articles and categories. The client component shown on the left-side of Figure 2 is an HTTP client that issues paginated queries to the MediaWiki HTTP server. The user specifies a Wikipedia category and the client reads all pages and subcategories that are a part of that category. This allows us to build a n-ary tree where each node is a category that is associated with a set of documents and a set of subcategories - its child nodes.

```
// A wikipedia page
type Page struct {
    Id        int       `json:"pageid"`  // The id of the page
    Namespace Namespace `json:"ns"`      // The namespace the page is a part of
    Title     string    `json:"title"`   // The title of the page
    Text      string    `json:"extract"` // Any text related to the page
}

// A MediaWiki client that is able to extract wikipedia pages.
type Client interface {
  ReadPages(request *PageRequest) (pages []Page, err error)
  ReadCategory(request *PageRequest) (*Page, error)
}
```

List. 1: Client interface

Listing 1 shows the `Client` interface and the `Page` structure that it returns. The `Namespace` property of a Page indicates whether the Page refers to an article or a subcategory of the category defined in the `PageRequest`. The `Text` property holds the article text or a brief description of the category, depending on the value of the `Namespace`. The text is stored in the wikitext format and sent to the `Lexer` component also shown in Figure 2. The

`Lexer` analyses the text and produces lexical tokens that are sent to the `Parser` for further processing. The `Parser` transforms the tokens into a list of paragraphs for an article or into a brief description for a category. The entities produced by the `Parser` are shown in Listing 2.

```
type Id string // The id type is a string

type Category struct {
  Id          Id     // The category identifier
  Source      string // The source of the category (mediawiki)
  Name        string // The name of the category
  Description string // Brief description of the category
}

type Paragraph struct {
  Title    string // The title of the paragraph
  Position int    // The position of the paragraph in the table of contents
  Text     string // The clean text of the paragraph
}

type Document struct {
  Id         Id           // The document identifier
  Title      string       // The title of the document
  Source     string       // The source of the document (mediawiki)
  Categories []Id         // The categories that this document belongs to
  Paragraphs []Paragraph  // The paragraphs that belong to this document
}
```

List. 2: Application entities

Together, the `Client`, `Lexer` and `Parser` allow us to crawl Wikipedias' category tree. The `Crawler` interface is quite simple. It takes in a category to visit to and produces a list of subcategories and a list of documents that belong to that category, as shown by Listing 3.

```
func (cw *Crawler) Walk(node Category) ([]Category, []Document, error) {
    // Create the request to send to the MediaWikiClient
    request, err := mw.NewPageRequest(cw.Language, node.Name)
    if err != nil {
        return nil, nil, ErrCrawl.from(err)
    }
    // Extract the Page structures from MediaWiki
    pages := cw.extract(request)
    // transform lexes and parses the pages into Category and Document entities
    categories, documents := cw.transform(pages)
    // Return the results to the caller.
    // The caller can now take a subcategory and call Walk again, thus
    // traversing the category tree
    return categories, documents, nil
}
```

List. 3: Walk function that visits a node in the Wikipedia category tree and returns its subcategories and documents

The ETL pipeline uses the Breadth-First-Search algorithm together with the crawler and the storage interface for persisting documents into MongoDB. After visiting a node, the documents are asynchronously put in the database and the subcategories are put in a queue for later processing. The user can configure the behaviour of the pipeline through a JSON configuration file. The database object specifies the MongoDB endpoint to store the data into. The etl object specifies the language subsystem to crawl, the root category to start crawling from, the duration to crawl for and the duration to sleep after visiting a node. The sleep duration is required to avoid clogging the MediaWiki API with requests, which may lead to a permanent ban.

```
{
    "database": {
        "uri": "mongodb://localhost:27017",
        "user": "",
        "password": ""
    },
    "etl": {
        "language": "de",
        "root": {
            "name": "Strukturwissenschaft",
            "source": "mediawiki",
            "description": ""
        },
        "duration": "1h",
        "interval": "30s"
    }
}
```

List. 4: Example configuration file. The ETL pipeline will begin crawling the Strukturwissenschaft category. The crawler will proceed for 1 hour and will pause every 30 seconds. The documents and categories will be stored in the MongoDB instance listening on localhost on port 27017.

```
$ pwd
/home/user
$ ls -l
drwxr-xr-x 7 user user      4096 Jan  9 14:07 etl-config.json
drwxr-xr-x 7 user user      4096 Jan  9 14:07 wiki-plag
$ cd ./wiki-plag/wiki-documents/cmd/wiki-etl
$ go build main.go -o ./wiki-crawler
$ ./wiki-crawler -cfg /home/user/etl-config.json
```

List. 5: Example pipeline usage

After the tree has been crawled, the user can use the REST API to perform CRUD operations on documents and categories saved in the database. The REST API runs in a docker container.

1.2 Natural Language Processing Pipeline

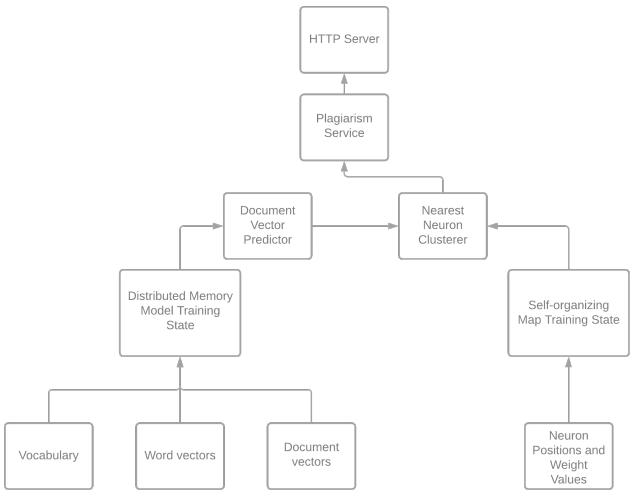

Fig. 3: Information flow diagram in the inference pipeline that searches for plagiarism candidates from Wikipedia.

The NLP pipeline shown in Figure 3 is responsible for finding documents that may have been plagiarised. The user communicates with the system through the `HTTP Server` by sending a POST request containing a potentially copied document. The HTTP Server delegates the request to the `Plagiarism Service` which preprocesses the document by lemmatising all words and removing all punctuation tokens. Afterwards, the DM-PV model introduced in Chapter 2.2 is loaded into memory and a vector is predicted for the document. The vector is then assigned to a neuron in the self-organizing map introduced in Chapter 2.3. All documents in the training set that have been clustered to the same neuron are considered plagiarism candidates, i.e documents that a user has copied content from. These documents are returned to the user along with their similarity to the input document which is measured via the *cosine similarity* metric. Listing 6 shows a summary of the detection procedure in Python code.

```
def detect_plagiarisms(self, input_document: str) -> Iterable[PlagCandidate]:
    # Words are lemmatized and punctuations removed
    words: Iterable[str] = lemmatize([input_document])
    # Vector is predicted
    vector: List[float] = dmpv.predict_vector(words)
    # The x, y coordinates of the neuron that is closest to the vector are computed
    neuron_coordinates: Tuple[int, int] = som.winner(vector)
    # All documents in the training set at the same coordinates are candidates
    return som.map[neuron_coordinates]
```

List. 6: Python code for NLP pipeline

1.3 Graphical user interface

2 Algorithms

2.1 Continuous Bag of Words

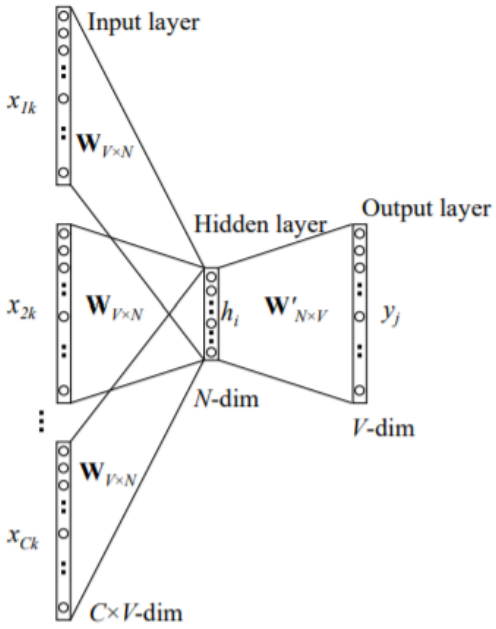

Fig. 4: The Continuous Bag of Words model

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.

2.1.1 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 $v_1, v_2, \ldots, v_c$ 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

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

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 represended 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

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

2.1.2 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

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

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

$$\frac{\partial H}{\partial v'_j} = \frac{\partial H}{\partial u_j} \cdot \frac{\partial u_j}{\partial v'_j}. \qquad (8)$$

Calculating the derivative of the loss yields

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

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

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

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

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

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 $v_1, v_2, \ldots, v_c$. The chain rule tells us that

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

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:

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

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

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

We now derive the update rule

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

2.1.3 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.

2.2 Distributed Memory Model of Paragraph Vectors

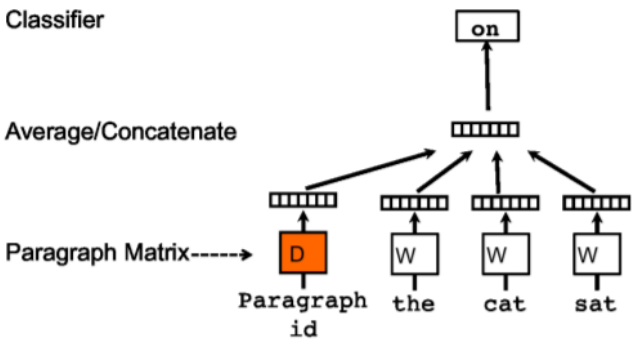

Fig. 5: The DM-PV model.

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 5 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.

2.2.1 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.

2.3 Self-Organizing Maps

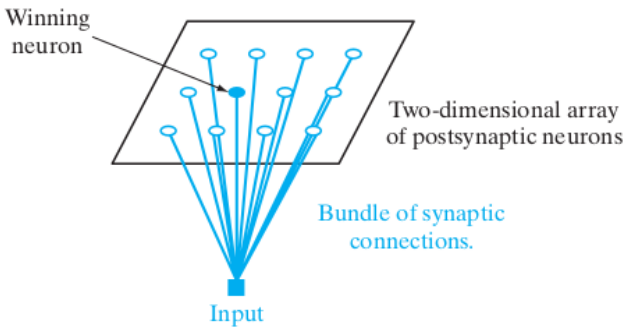

Fig. 6: An image showing the connection between the input data and the neurons of a self-organizing map. Each input is connected to all of the neurons in the lattice. The neurons compete among each other and the neuron that resembles the input the most is marked as the winner neuron. After multiple iterations, 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.

2.3.1 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;

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

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 $\mathbf{x}$.

2.3.2 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.

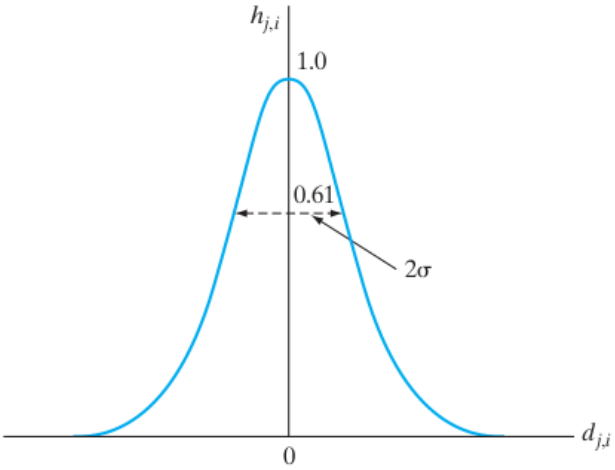

Fig. 7: The gaussian neighbourhood function. σ controls the size of the neighbourhood. Note that the peak of the bell curve maps to all neurons whose distance to the winner is 0. In other words, neurons that have large distances to the winner have low probabilities of being activated.

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 7) 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.

2.3.3 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

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

2.3.4 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.